Mastering Bitcoin 2da Edition

Prólogo

Escribiendo el libro de Bitcoin

La primera vez que me topé con bitcoin, fue a mediados de 2011. Mi primera reacción fue más o menos "¡Pfft! ¡Dinero nerd!" por lo que lo ignoré durante otros seis meses, sin comprender su importancia. Esta es una reacción que he visto repetirse entre muchas de las personas más inteligentes que conozco, lo que me da un poco de consuelo. La segunda vez que me encontré con Bitcoin, que fue en una discusión de listas de correo, decidí leer el whitepaper original escrito por Satoshi Nakamoto para estudiar la fuente autorizada y ver de qué se trataba todo aquello. Todavía recuerdo el momento en que terminé de leer esas nueve páginas, cuando me di cuenta de que bitcoin no era simplemente una moneda digital, sino una red de confianza que también podía proporcionar las bases para mucho más que solo monedas. La constatación de que "esto no es dinero, es una red de confianza descentralizada", me inició en un viaje de cuatro meses para devorar cada fragmento de información sobre bitcoin que pude encontrar. Me obsesioné y quedé cautivado, pasando 12 horas o más de cada día pegado a una pantalla, leyendo, escribiendo, codificando y aprendiendo todo lo que pude. Salí de este estado de arrobamiento, casi 10 kg más delgado por falta de comidas consistentes, decidido a dedicarme a trabajar en bitcoin.

Dos años después, luego de crear una serie de pequeños emprendimientos para explorar varios servicios y productos relacionados con bitcoin, decidí que era hora de escribir mi primer libro. Bitcoin fue el tema que me llevó a un frenesí de creatividad y consumió mis pensamientos; Era la tecnología más emocionante que había encontrado desde Internet. Ahora era el momento de compartir mi pasión por esta increíble tecnología con un público más amplio.

Audiencia Prevista

Este libro está destinado principalmente a programadores. Si Ud. es capaz de usar un lenguaje de programación, este libro le enseñará cómo funcionan las monedas criptográficas, cómo usarlas y cómo desarrollar un software que funcione con ellas. Los primeros capítulos también son adecuados como una introducción con profundidad al tema de bitcoin para personas sin experiencia en programación; aquellos que intentan comprender el funcionamiento interno de Bitcoin y las criptomonedas.

¿Por Qué Hay Insectos en la Cubierta?

La hormiga cortadora de hojas es una especie que exhibe un comportamiento altamente complejo en un super-organismo de colonias, pero cada hormiga individual opera en un conjunto de reglas simples impulsadas por la interacción social y el intercambio de aromas químicos (feromonas). Según la Wikipedia: "Junto a los humanos, las hormigas cortadoras de hojas forman las sociedades animales más grandes y complejas de la Tierra". Las hormigas cortadoras de hojas en realidad no comen hojas, sino que las usan para cultivar un hongo, que es la fuente central de alimento para la colonia. ¿Lo captan? ¡Estas hormigas están cultivando!

Aunque estas hormigas forman una sociedad basada en castas y dependen de una reina para reproducirse no hay ninguna autoridad central ni líder en la colonia. El comportamiento inteligente y sofisticado que exhibe la colonia de millones es una propiedad que surge de la interacción entre los individuos de una red social.

La naturaleza nos enseña que los sistemas descentralizados pueden ser extremadamente resistentes y dar pie a una increíblemente sofisticada complejidad que emerge sin necesidad de un conductor, autoridad central, jerarquía o complicados componentes.

Bitcoin es una red de confianza descentralizada altamente sofisticada que puede manejar un gran número de procesos financieros. Sin embargo, cada nodo en la red bitcoin sigue algunas reglas matemáticas simples. La interacción entre muchos nodos es lo que conduce a la aparición del comportamiento sofisticado, no a la complejidad inherente o la confianza en un solo nodo. Al igual que una colonia de hormigas, la red bitcoin es una red resistente de nodos simples que siguen reglas simples que juntas pueden hacer cosas increíbles sin ninguna coordinación central.

Convenciones Usadas en este Libro

Las siguientes convenciones tipográficas se utilizan en este libro:

Cursiva

Indica términos nuevos, URLs, direcciones de email, nombres de archivo y extensiones de archivo.

Ancho constante

Usado para listados de programas, así como dentro de párrafos para referirse a elementos de un programa como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, sentencias y palabras clave.

```
Ancho constante con negrita
```

Muestra comandos u otro texto que debe ser tecleado literalmente por el usuario.

Ancho constante con cursiva

Muestra texto que debe ser reemplazado por valores provistos por el usuario o valores determinados por contexto.

TIP	Este icono significa un consejo o una sugerencia.
NOTE	Este icono significa una nota general.
WARNING	Este icono indica una advertencia o cuidado.

Ejemplos de Código

Los ejemplos se han ilustrado en Python y C ++ y utilizan la línea de comandos de un sistema operativo tipo Unix como Linux o macOS. Todos los fragmentos de código están disponibles en el siguiente repositorio de GitHub (https://github.com/bitcoinbook/bitcoinbook) en el subdirectorio *code* del repositorio principal. El código del libro puede bifurcarse, probarse los ejemplos de código y pueden enviarse correcciones a través de GitHub.

Todos los fragmentos de código pueden ejecutarse en la mayoría de los sistemas operativos con una instalación mínima de compiladores e intérpretes para los lenguajes correspondientes. Cuando sea necesario, proporcionamos instrucciones de instalación básicas y ejemplos paso a paso.

Algunos de los fragmentos de código y sus salidas se han reformateado en la impresión. En todos estos casos, las líneas se han dividido por un carácter de barra invertida (\), seguido de un carácter de nueva línea. Cuando se quiera transcribir los ejemplos, elimine esos dos caracteres y una las líneas de nuevo para ver los resultados idénticos a como se muestran en el ejemplo.

Siempre que sea posible, los fragmentos de código utilizan valores y cálculos reales, por lo que se puede construir cada uno de los ejemplos y obtener los mismos resultados en cualquier código que escriba para calcular los mismos valores. Por ejemplo, las llaves privadas, las llaves públicas correspondientes y las direcciones son reales. Las transacciones de la muestra, los bloques y las referencias a la cadena de bloques se han introducido en la cadena de bloques real de bitcoin y forman parte del libro contable, por lo que puede verificarse en cualquier sistema bitcoin.

Usando Ejemplos de Código

Este libro está aquí para ayudarle a hacer su trabajo. En general, puede utilizar el código de ejemplo que se ofrece con este libro en sus programas y en documentación. No es necesario ponerse en contacto con nosotros para obtener permiso a menos que esté reproduciendo una parte importante del código. Por ejemplo, escribir un programa que utiliza varios trozos de código de este libro no requiere permiso. La venta o distribución de un CD-ROM de ejemplos de los libros de O'Reilly sí requiere permiso. Responder a una pregunta que cite a este libro o que replique el código de ejemplo no requiere permiso. La incorporación de una cantidad significativa del código de ejemplo de este libro en la documentación de su producto sí requiere permiso.

Agradecemos, pero no exigimos, ninguna atribución. Una atribución generalmente incluye el título, el autor, el editor y el ISBN. Por ejemplo: "*Mastering Bitcoin* por Andreas M. Antonopoulos (O'Reilly). Copyright 2017 Andreas M. Antonopoulos, 978-1-491-95438-6".

Algunas ediciones de este libro se ofrecen bajo una licencia de código abierto, como CC-BY-NC, en cuyo caso se aplican los términos de esa licencia.

Si considera que el uso de los ejemplos de código, está fuera del uso justo o del permiso otorgado anteriormente, no dude

en contactarnos en permissions@oreilly.com.

Direcciones y transacciones de Bitcoin en este libro

Las direcciones de bitcoin, transacciones, llaves, códigos QR y datos de la cadena de bloques utilizados en este libro son, en su mayor parte, reales. Eso significa que puede navegar por la cadena de bloques, ver las transacciones ofrecidas como ejemplos y recuperarlas con sus propios scripts o programas, etc.

Sin embargo, tenga en cuenta que las llaves privadas utilizadas para construir direcciones están impresas en este libro o han sido "quemadas". Eso significa que si envía dinero a cualquiera de estas direcciones, el dinero se perderá para siempre o, en algunos casos, todos los que puedan leer el libro pueden tomarlo utilizando las llaves privadas impresas aquí.

> **WARNING** NO ENVÍE DINERO A NINGUNA DE LAS DIRECCIONES DE ESTE LIBRO. Otro lector podría tomar su dinero o bien, se perderá para siempre.

O'Reilly Safari

> **NOTE** *Safari* (anteriormente Safari Books Online) es una plataforma de capacitación y consulta basada en membresía para empresas, gobiernos, educadores e individuos.

Los miembros tienen acceso a miles de libros, videos de capacitación, rutas de aprendizaje, tutoriales interactivos y listas de reproducción seleccionadas de más de 250 editoriales, incluidos O'Reilly Media, Harvard Business Review, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Adobe, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett y Course Technology, entre otros.

Para obtener más información, visite *http://oreilly.com/safari*.

Cómo Contactar con Nosotros

Dirija los comentarios y preguntas sobre este libro al editor:

- O'Reilly Media, Inc.
- 1005 Gravenstein Highway North
- Sebastopol, CA 95472
- 800-998-9938 (en los Estados Unidos o Canada)
- 707-829-0515 (internacional o local)
- 707-829-0104 (fax)

Para comentar o hacer preguntas técnicas sobre este libro, envíe un correo electrónico a *bookquestions@oreilly.com*.

Para obtener más información acerca de nuestros libros, cursos, conferencias y noticias, visite nuestro sitio web en http://www.oreilly.com.

Encuéntrenos en Facebook: http://facebook.com/oreilly

Síganos en Twitter: http://twitter.com/oreillymedia

Véanos en YouTube: http://www.youtube.com/oreillymedia

Contactando al Autor

Puede contactarme a mí, (Andreas M. Antonopoulos), a través de mi página personal: https://antonopoulos.com/

La información sobre *Dominando el Bitcoin*, así como la edición abierta y las traducciones están disponibles en: enlace:https://bitcoinbook.info/[]

Sígame en Facebook: enlace:https://facebook.com/AndreasMAntonopoulos[]

Sígame en Twitter: enlace:https://twitter.com/aantonop[]

Sígame en Linkedin: enlace:https://linkedin.com/company/aantonop[]

Muchas gracias a todos mis “patreons” que apoyan mi trabajo a través de donaciones mensuales. Puede seguir mi página de Patreon por aquí: enlace:https://patreon.com/aantonop[]

Agradecimientos

Este libro representa los esfuerzos y contribuciones de muchas personas. Estoy agradecido por toda la ayuda que recibí de amigos, colegas e incluso desconocidos, que se unieron a mí en este esfuerzo por escribir el libro técnico definitivo sobre criptomonedas y bitcoins.

Resulta imposible distinguir entre la tecnología de bitcoin y la comunidad de bitcoin, siendo este libro un producto de esa comunidad tanto como lo es un libro sobre la tecnología. De principio a fin he recibido de la comunidad de bitcoin en su totalidad el entusiasmo, el ánimo, la recompensa y apoyo que me han permitido la realización de este libro. Mas que ninguna otra cosa, este libro me ha incluido en un maravillosa comunidad durante dos años y no puedo daros suficientes gracias por aceptarme como miembro. Hay demasiadas personas como para mencionarlas a todas por nombre - la gente que he conocido durante conferencias, eventos, quedadas, cenando pizza y otras pequeñas tertulias, así como aquellos que se han comunicado conmigo a través de Twitter, en reddit, bitcointalk.org y en GitHub y han dado algo que ha contribuido a este libro. Cada idea, analogía, pregunta, respuesta y explicación que encontrarás en este libro ha sido en algún momento inspirado, contrastado o mejorado por medio de mis interacciones con la comunidad. Gracias a todos por vuestro apoyo; sin vosotros este libro existiría. Estaré agradecido para siempre.

El viaje para convertirse en autor comienza mucho antes del primer libro, por supuesto. Mi primer idioma (y escuela) fue el griego, por lo que tuve que tomar un curso de redacción de inglés correctivo en mi primer año de la universidad. Le debo las gracias a Diana Kordas, mi maestra de escritura en inglés, que me ayudó a desarrollar confianza y habilidades ese año. Más tarde, como profesional, desarrollé mis habilidades de escritura técnica sobre el tema de los centros de datos, escribiendo para la revista *Network World*. Debo agradecer a John Dix y John Gallant, quienes me dieron mi primer trabajo de redacción como columnista en *Network World* y a mi editor Michael Cooney y mi colega Johna Till Johnson, quienes editaron mis columnas y las hicieron aptas para su publicación. Escribir 500 palabras por semana durante cuatro años me dio suficiente experiencia para considerar eventualmente convertirme en autor.

Gracias también a quienes me apoyaron cuando presenté mi propuesta de libro a O’Reilly, proporcionando referencias y revisando la propuesta. Específicamente, gracias a John Gallant, Gregory Ness, Richard Stiennon, Joel Snyder, Adam B. Levine, Sandra Gittlen, John Dix, Johna Till Johnson, Roger Ver y Jon Matonis. Un agradecimiento especial a Richard Kagan y Tymon Mattoszko, quienes revisaron las primeras versiones de la propuesta y Matthew Taylor, quien corrigió la propuesta.

Gracias a Cricket Liu, autor del título de O’Reilly *DNS y BIND*, quien me presentó a O’Reilly. Gracias también a Michael Loukides y Allyson MacDonald de O’Reilly, que trabajaron durante meses para ayudar a que este libro se hiciera realidad. Allyson fue especialmente paciente cuando se incumplieron los plazos y los resultados se retrasaron a medida que la vida intervino en nuestro horario planificado. Para la segunda edición, agradezco a Timothy McGovern por guiar el proceso, Kim Cofer por editar pacientemente y Rebecca Panzer por ilustrar muchos diagramas nuevos.

Los primeros borradores de los primeros capítulos fueron los más difíciles, porque bitcoin es un tema difícil de descifrar. Cada vez que halaba de un hilo en la tecnología bitcoin, tenía que halarlos todos. En repetidas ocasiones me quedé atascado y un poco abatido mientras luchaba por hacer que el tema fuera fácil de entender, creando una narrativa digerible en torno a un tema técnico tan denso. Finalmente, decidí contar la historia de bitcoin a través de las historias de las personas que usan bitcoin y todo el libro se hizo mucho más fácil de escribir. Le debo las gracias a mi amigo y mentor, Richard Kagan, quien me ayudó a desentrañar la historia y superar los momentos en que me bloqueé al escribir. Agradezco a Pamela Morgan, quien revisó los primeros borradores de cada capítulo en la primera y segunda edición del libro, y me formuló las preguntas difíciles que me ayudaron a mejorarlos. Además, gracias a los desarrolladores del grupo “Desarrolladores Meetup de Bitcoin de San Francisco”, así como a Taariq Lewis y Denise Terry por ayudar a probar el material inicial. Gracias también a Andrew Naugler por el diseño infográfico.

Durante el desarrollo del libro, puse los primeros borradores disponibles en GitHub e invité a realizar comentarios públicamente. En respuesta, se presentaron más de cien comentarios, sugerencias, correcciones y contribuciones. Esas contribuciones se reconocen explícitamente, con mi agradecimiento, en [github_contrib]. Sobre todo, mi sincero agradecimiento a mis editores voluntarios de GitHub, Ming T. Nguyen (1ra edición) y Will Binns (2da edición), que trabajaron incansablemente para subsanar, administrar y resolver la incorporación de mejoras (las “pull requests”), emitir reportes y realizar correcciones de errores de código en GitHub.

Una vez redactado, el libro pasó por varias rondas de revisión técnica. Gracias a Cricket Liu y Lorne Lantz por su

minuciosa revisión, comentarios y apoyo.

Varios desarrolladores de bitcoin contribuyeron con muestras de código, reseñas, comentarios y estímulos. Gracias a Amir Taaki y Eric Voskuil, por ejemplo, fragmentos de código y muchos comentarios excelentes; Chris Kleeschulte por contribuir con el apéndice Bitcore; Vitalik Buterin y Richard Kiss por su ayuda con las matemáticas de curva elíptica y contribuciones de código; Gavin Andresen por correcciones, comentarios y aliento; Michalis Kargakis para comentarios, contribuciones y redacción de btcd; y Robin Inge por las presentaciones de erratas que mejoran la segunda impresión. En la segunda edición, nuevamente recibí mucha ayuda de muchos desarrolladores de Bitcoin Core, incluidos Eric Lombrozo, que desmitificó a Segregated Witness, Luke Dashjr, que ayudó a mejorar el capítulo sobre transacciones, Johnson Lau, que revisó Segregated Witness y otros capítulos, y muchos otros. Les debo también las gracias a Joseph Poon, Tadge Dryja y Olaoluwa Osuntokun, quienes explicaron Lightning Network, revisaron mis escritos y respondieron preguntas cuando me quedé atrapado.

Mi amor por los libros y la palabra se lo debo a mi madre Teresa quién me crió en una casa donde los libros cubrían las paredes. Mi madre también me compró mi primer ordenador en 1982 aun cuando se confesó como tecnófoba. Mi padre Menelaos, un ingeniero civil que acaba de publicar su primer a los 80 años de edad, me enseñó el pensamiento lógico y analítico y el amor por la ciencia y la ingeniería.

Gracias a todos ustedes por ayudarme a lo largo de esta travesía.

Glosario Rápido

Este glosario rápido contiene muchos de los términos relacionados con bitcoin. Estos términos se usan en todo el libro, así que márquelo como favorito para una referencia rápida.

dirección

Una dirección bitcoin se parece a 1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV. Consiste en una cadena de letras y números. Es realmente una versión codificada en formato "base58check" del hash de una llave pública de 160 bits. Tal y como podríamos pedirle a otros que nos envíen un correo a nuestra dirección de correo electrónico, así también le podemos pedir a otros que nos envíen bitcoin a una de nuestras direcciones de bitcoin.

bip

Propuestas de Mejora de Bitcoin. Son un conjunto de propuestas que los miembros de la comunidad bitcoin han presentado para mejorar a bitcoin. Por ejemplo, BIP-21 es una propuesta para mejorar el esquema uniforme de identificación de recursos (del inglés, el "URI") de bitcoin.

bitcoin

Es el nombre de la unidad monetaria (la moneda), la red y el software.

bloque

Una agrupación de transacciones, marcada con un sello de tiempo y una huella digital del bloque anterior. El encabezado del bloque se procesa mediante una función hash para generar una prueba de trabajo, validando así las transacciones. Los bloques válidos se agregan a la cadena de bloques principal por consenso de la red.

cadena de bloques

Es una lista de bloques validados, cada uno vinculado a su predecesor hasta el bloque génesis.

Problema de los Generales Bizantinos

Un sistema informático confiable debe ser capaz de hacer frente a las fallas de uno o más de sus componentes. Un componente fallido puede exhibir un tipo de comportamiento que a menudo se pasa por alto, es decir, enviar información conflictiva a diferentes partes del sistema. El problema de hacer frente a este tipo de falla se expresa de manera abstracta como el "Problema de los Generales Bizantinos".

coinbase

Un campo especial utilizado como la única entrada para las transacciones de coinbase. La entrada coinbase permite reclamar la recompensa del bloque y proporciona hasta 100 bytes para datos arbitrarios. No debe confundirse con la Transacción Coinbase.

transacción coinbase

Es la primera transacción en un bloque. Siempre es creada por un minero; incluye una sola entrada, la entrada coinbase. No debe confundirse con "Coinbase".

almacenamiento en frío

Se refiere a mantener una reserva de bitcoin fuera de línea. El almacenamiento en frío se logra cuando las llaves privadas de Bitcoin se crean y se almacenan en un entorno seguro fuera de línea. El almacenamiento en frío es importante para cualquier persona que posea bitcoin. Las computadoras en línea son vulnerables a los piratas informáticos y no deben usarse para manejar una cantidad significativa de bitcoin.

confirmaciones

Una vez que una transacción es incluida en un bloque, tiene una confirmación. Tan pronto como se extrae otro bloque en la misma cadena de bloques, la transacción tiene dos confirmaciones, y así sucesivamente. Seis o más confirmaciones se consideran prueba suficiente de que una transacción no puede ser revertida.

consenso

Es lo que sucede cuando varios nodos, generalmente la mayoría de los nodos en la red, poseen todos los mismos bloques en su mejor cadena de bloques validada localmente. No debe confundirse con "reglas de consenso".

reglas de consenso

Son las reglas de validación de los bloques que siguen los nodos completos para mantenerse en consenso con todos los demás nodos. No debe confundirse con “consenso”.

dificultad

La configuración de toda la red que controla cuánto cómputo es necesario para producir una prueba de trabajo válida.

reajuste de dificultad

Es un recálculo de la dificultad de toda la red, que ocurre una vez cada 2.016 bloques y toma en cuenta la potencia de hash de los 2.016 bloques anteriores.

objetivo de dificultad

Es un nivel de dificultad en el que la capacidad de cómputo total de la red encontrará bloques aproximadamente cada 10 minutos.

doble gasto

Un doble gasto es el resultado de gastar con éxito algo de dinero más de una vez. Bitcoin se protege contra el doble gasto, verificando cada transacción que se agrega a la cadena de bloques, para garantizar que las entradas de dichas transacciones no se hayan utilizado previamente.

ECDSA

El algoritmo de firma digital de curva elíptica o del inglés, “ECDSA” es un algoritmo criptográfico utilizado por Bitcoin para garantizar que los fondos solo puedan ser gastados por sus legítimos propietarios.

nonce extra

A medida que aumentaba la dificultad, los mineros a menudo recorrían los 4 mil millones de valores del nonce sin encontrar un bloque. Debido a que el script de la entrada coinbase puede almacenar entre 2 a 100 bytes de datos, los mineros comenzaron a usar ese espacio como espacio de “nonce extra”, lo que les permite explorar un rango mucho más grande de valores de encabezados de bloque para encontrar bloques válidos.

comisiones

El remitente de una transacción a menudo incluye una comisión a la red para procesar la transacción solicitada. La mayoría de las transacciones requieren una tarifa mínima de 0.5 mBTC.

bifurcación

Las bifurcaciones, también conocidas como bifurcaciones accidentales, ocurren cuando dos o más bloques tienen la misma altura de bloque, bifurcando la cadena de bloques. Por lo general, ocurre cuando dos o más mineros encuentran bloques casi al mismo tiempo. También puede suceder como parte de un ataque.

bloque génesis

El primer bloque en la cadena de bloques, utilizado para inicializar la criptomoneda.

bifurcación dura

Una bifurcación dura, también conocida como Cambio por “Hard-Forking”, es una divergencia permanente en la cadena de bloques, que ocurre comúnmente cuando los nodos no actualizados no pueden validar los bloques creados por los nodos actualizados que siguen nuevas reglas de consenso. No debe confundirse con bifurcaciones a secas, una bifurcación suave, una bifurcación de software o una bifurcación de código en el estándar “Git”.

monedero hardware

Un monedero hardware es un tipo especial de billetera de bitcoin que almacena las llaves privadas del usuario en un dispositivo hardware seguro.

hash

Una huella digital de alguna entrada binaria.

bloqueos por hash o “hashlocks”

Un bloqueo por hash es un tipo de acertijo que restringe el gasto de una salida hasta que se revela públicamente un dato secreto específico. Los bloqueo por hash o “hashlock”, tienen la propiedad útil de que una vez que se abre públicamente el hashlock, también se puede abrir cualquier otro hashlock asegurado con el mismo secreto. Esto hace

posible crear múltiples salidas que están aseguradas por el mismo hashlock y que todas se pueden gastar al mismo tiempo.

protocolo de Jerarquía Determinista o HD

Es un protocolo de creación y transferencia de llaves jerárquico determinista (o del inglés “HD”) (propuesta de mejora BIP32), que permite crear llaves secundarias a partir de llaves primarias bajo una misma jerarquía.

cartera HD

Carteras que utilizan el protocolo de creación y transferencia de llaves jerárquico-determinista (Protocolo HD) (BIP32).

semilla de cartera HD

La semilla de un monedero HD o semilla raíz, es un valor potencialmente corto utilizado como semilla para generar la llave privada maestra y el código de cadena maestro para una billetera HD.

HTLC

Un Contrato blindado por un acertijo Hash y un Bloqueo Temporal (o del inglés “TimeLock”) o también “HTLC” (Hash TimeLock Contract) es un tipo de promesa de pagos que utiliza “hashlocks” y “timelocks” para exigir que el receptor de un pago reconozca haber recibido el pago antes de cierta fecha límite al generar una prueba criptográfica de pago, so pena de perder la capacidad de reclamar dicho pago, devolviéndolo al pagador.

KYC

Conozca a su cliente (del inglés: “KYC”) es el procedimiento que sigue un negocio, identificando y verificando la identidad de sus clientes. El término también se utiliza para referirse a la regulación bancaria que rige estas actividades.

LevelDB

Una “LevelDB” es una biblioteca de almacenamiento de código abierto de llaves-valores en disco. LevelDB es una biblioteca liviana y de un solo propósito para mantener enlaces de manera persistente con muchas plataformas.

Red “Lightning Network”

La red Lightning Network es una implementación de contratos blindados por Hash y Timelocks (HTLC) con canales de pago bidireccionales que permite que los pagos sean enrutados de manera segura a través de múltiples canales de pago entre pares. Esto permite la formación de una red donde cualquier par en la misma puede pagar a cualquier otro par, incluso si no tienen un canal abierto directamente entre sí.

Bloqueo Temporal o “Locktime”

El bloqueo temporal “Locktime”, o más técnicamente nLockTime, es la parte de una transacción que indica la hora más temprana o el bloque más temprano a partir del cual esa transacción puede agregarse a la cadena de bloques.

Tanque de Memoria o “mempool”

El tanque de memoria de bitcoin (o el “pool” de memoria) es una colección de todas las transacciones agrupadas en un bloque de datos, que han sido verificadas por los nodos de bitcoin, pero que aún no se han confirmado.

raíz de merkle

Es el nodo raíz de un árbol de merkle, un descendiente de todos los pares de hash que hay en el árbol. Las cabeceras de bloque deben incluir una raíz de merkle válida que descienda de todas las transacciones que existan en ese bloque.

árbol de merkle

Es una base de datos con estructura de árbol construida mediante el cálculo de los hash de emparejados de datos (las hojas), luego emparejando y calculando el hash sucesivo de los resultados hasta que quede un solo hash, la raíz de merkle. En Bitcoin, las hojas son casi siempre transacciones de un mismo bloque.

minero

Un nodo de red que encuentra pruebas de trabajo válidas para nuevos bloques, mediante procesos de cómputo donde se calcula una función hash repetitivamente.

multifirma

Una firma múltiple (multifirma) hace referencia a aquel procedimiento donde se requiere más de una llave para

autorizar una transacción de bitcoin.

red

Es una red entre pares iguales que propaga transacciones y bloques a cada nodo bitcoin en la red.

nonce

El "nonce" en un bloque de bitcoin es un campo de 32 bits (4 bytes) cuyo valor se establece de modo que el hash del bloque contenga una serie de ceros iniciales. El resto de los campos no se deben cambiar, ya que tienen un significado definido.

transacciones fuera de la cadena u “off-chain”

Una transacción fuera de la cadena es un movimiento de valores que ocurre al margen la cadena de bloques. Si bien una transacción confirmada en la cadena—generalmente se conoce simplemente como *una transacción*—ésta modifica la cadena de bloques y depende de la cadena de bloques para determinar su validez; pero una transacción fuera de la cadena se basa en otros métodos para registrar y validar dicha transacción.

código transaccional u “opcode”

Son códigos de operación en el lenguaje de script de Bitcoin que o bien realizan “empuje” de datos o bien ejecutan funciones específicas dentro de instrucciones script de llave pública (o pubkey) o en instrucciones script de verificación de firmas.

protocolo Open Assets

El Protocolo “Open Assets” es una capa de servicios simple y poderosa construida sobre la cadena de bloques de bitcoin. Permite la emisión y transferencia de activos creados por el usuario.

OP_RETURN

Un código operativo o transaccional, utilizado en una de las salidas, de las así llamadas transacciones “OP_RETURN”. No debe confundirse con una transacción OP_RETURN.

transacción OP_RETURN

Un tipo de transacción que agrega datos arbitrarios a una secuencia de comandos de salida o de llave pública demostrablemente indestructible que los nodos completos no tienen que almacenar en su base de datos UTXO. No debe confundirse con el código de operación OP_RETURN.

bloque huérfano

Son bloques cuyo bloque padre no ha sido procesado por el nodo local, por lo que aún no se pueden validar por completo. No debe confundirse con el bloque vencido.

transacciones huérfanas

Transacciones que no pueden ingresar al tanque de memoria debido a que están faltando una entrada válida o más en dichas transacciones.

salida

“Salida”, salida de una transacción o “TxOut” hace referencia a una salida en una transacción que contiene dos campos: un campo de valor para transferir cero o más satoshis y una secuencia de comandos de salida o “pubkey” para indicar qué condiciones deben cumplirse para que esos satoshis se puedan gastar o utilizar más adelante.

P2PKH

(Pagar-al-Hash-de-una-Llave-Pública) Son aquellas transacciones que pagan a una dirección bitcoin que contiene scripts del tipo P2PKH o en inglés “Pay-To-Pub-Key-Hash”. Una salida bloqueada por un script P2PKH puede desbloquearse (gastarse) presentando una llave pública y una firma digital creada por la llave privada que le corresponde.

P2SH

Pagar-al-Hash-de-un-Script (o del inglés: P2SH o “Pay-to-Script-Hash”) es un nuevo y poderoso tipo de transacción que simplifica enormemente el uso de scripts de transacción complejas. Con el estándar P2SH, el script complejo que detalla las condiciones para gastar una salida (o el “script de canje”) no se presenta en el script de bloqueo. En cambio, solo un hash está en el script de bloqueo.

dirección P2SH

Las direcciones P2SH son codificaciones bajo formato “Base58Check” del hash de 20 bytes de un script, las direcciones P2SH usan el prefijo de versión "5", que da como resultado direcciones codificadas Base58Check que comienzan con un "3". Las direcciones P2SH ocultan toda la complejidad, de modo que la persona que realiza un pago no ve el script.

P2WPKH

La firma de un P2WPKH (Pagar-al-Testigo-del-Hash-de-una-Llave-Pública, o del inglés “Pay-to-Witness-Public-Key-Hash”) contiene la misma información que el estándar P2PKH, pero se encuentra en el campo “testigo” de la transacción en lugar del campo “scriptSig”. El “scriptPubKey” también se modifica.

P2WSH

La diferencia entre P2SH y P2WSH (Pay-to-Witness-Script-Hash) está en el cambio de ubicación de la prueba criptográfica del campo “scriptSig” que se reubica en el campo del testigo y la “scriptPubKey”, que también se modifica.

cartera en papel

En el sentido más específico, una cartera en papel es un documento que contiene todos los datos necesarios para generar cualquier número de llaves privadas de Bitcoin, formando una billetera de llaves. Sin embargo, las personas a menudo usan el término para referirse a cualquier forma de almacenar bitcoin fuera de línea como un documento físico. Esta segunda definición también incluye llaves en papel y códigos redimibles.

canales de pago

Un canal de micropagos o canales de pago es una clase de técnicas diseñadas para permitir a los usuarios realizar múltiples transacciones de bitcoin sin tener que encomendar todas estas transacciones a la cadena de bloques de bitcoin. En un canal de pago típico, solo se agregan dos transacciones a la cadena de bloques, pero se puede realizar un número ilimitado o casi ilimitado de pagos entre los participantes.

minería en agrupaciones

La minería en agrupación es un enfoque de minería en el que múltiples clientes generadores contribuyen con la producción de un bloque y luego dividen la recompensa del bloque según la potencia de procesamiento aportada.

Prueba de Participación

La Prueba de Participación (o del inglés Proof-of-Stake; PoS) es un método por el cual la red de la cadena de bloques de una criptomoneda tiene como objetivo lograr un consenso distribuido. La prueba de participación pide a los usuarios que demuestren la propiedad de una cierta cantidad de moneda (su "participación" en la moneda).

Prueba-de-Trabajo

Es un paquete de datos que requiere de una cantidad de cómputo significativo para poder calcularlo. En bitcoin, los mineros deben encontrar una solución numérica para el algoritmo SHA256 que cumpla con un objetivo universal de toda la red, el objetivo de dificultad.

recompensa

Una cantidad incluida en cada nuevo bloque como recompensa de la red para el minero que encontró la solución de la Prueba-de-Trabajo. Actualmente la recompensa es de 6.25 BTC por bloque.

RIPEMD-160

RIPEMD-160 es una función hash criptográfica de 160 bits. RIPEMD-160 es una versión reforzada de RIPEMD con un resultado hash de 160 bits, y se espera que sea seguro durante los próximos diez años o más.

satoshi

Un satoshi es la denominación más pequeña de bitcoin que se puede grabar en la cadena de bloques. Es el equivalente de 0,00000001 bitcoin y lleva el nombre del creador de Bitcoin, Satoshi Nakamoto.

Satoshi Nakamoto

Satoshi Nakamoto es el nombre utilizado por la persona o personas que diseñaron Bitcoin y crearon su implementación de referencia original, Bitcoin Core. Como parte de la implementación, también idearon la primera base de datos para la cadena de bloques. En el proceso, fueron los primeros en resolver el problema del gasto duplicado para la moneda digital. Su identidad real sigue siendo desconocida.

Script

Bitcoin utiliza un sistema de secuencias de comandos para las transacciones. Basados en el lenguaje FORTH de programación, el script es simple, basado en comandos de pila y es procesado de izquierda a derecha. A propósito no es un lenguaje de Turing completo, ya que este lenguaje no posee bucles.

ScriptPubKey (también conocido como el script “pubkey”)

ScriptPubKey o pubkey script, es un script incluido en las salidas de una transacción que establece las condiciones que deben cumplirse para que los satoshis que estas resguardan, se gasten. Los datos para cumplir las condiciones se pueden proporcionar en un script de firma.

ScriptSig (también conocido como script de firma)

ScriptSig o el script de firma, son los datos generados por un usuario que ejecuta un gasto que casi siempre se utilizan como variables para satisfacer el desafío del script conocido como “pubkey”.

llave secreta (también conocida como llave privada)

Es el número secreto que desbloquea los bitcoins enviados a la dirección correspondiente. Un llave secreta tiene el siguiente aspecto:

```
5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh
```

Testigo Segregado

El Testigo Segregado es una mejora del protocolo de Bitcoin en el que la data de la firma (o el "testigo") se separa de la data del remitente/receptor para optimizar aún más la estructura de las transacciones. El Testigo Segregado se implementó como una bifurcación suave; un cambio que técnicamente hace que las reglas del protocolo de Bitcoin sean más restrictivas.

SHA

El algoritmo de hash seguro o de sus siglas en inglés el “SHA”, es una familia de funciones de hash criptográfico publicadas por el Instituto Nacional (Norteamericano) de Estándares y Tecnología (NIST).

Verificación de Pago Simplificado (SPV)

La verificación de pago simplificado o SPV es un método para verificar qué transacciones particulares han sido incluidas en un bloque, sin tener que descargar todo el bloque. Este método de verificación a menudo es utilizado por el software de clientes ligeros de Bitcoin.

bifurcación suave

Una Bifurcación Suave o Cambio de “Soft-Forking” es una bifurcación temporal en la cadena de bloques, que ocurre comúnmente cuando los mineros que usan nodos no actualizados no siguen una nueva regla de consenso sobre la que sus nodos no conocen. No debe confundirse con una bifurcación a secas, una bifurcación fuerte, una bifurcación de software o bifurcación bajo el protocolo Git.

bloque vencido

Es todo aquel bloque que ha sido minado con éxito, pero que no está incluido en la mejor cadena de bloques actual, probablemente porque otro bloque a la misma altura fue colocado en su lugar primero. No debe confundirse con el concepto de bloque huérfano.

Bloqueos Temporales o timelocks

Un bloqueo temporal es un tipo de blindaje que restringe el gasto de algunos bitcoins hasta que se alcance algún momento del tiempo específico en el futuro o se haya alcanzado cierta altura de bloque en la cadena. Los bloqueos temporales ocupan un lugar destacado en muchos contratos de Bitcoin, incluidos los canales de pago y los contratos por bloqueo de tiempo y de hash.

transacción

En términos simples, se trata de una transferencia de bitcoins desde una dirección a otra. Más precisamente, una transacción es una estructura de datos firmada que expresa una transferencia de valor. Las transacciones se transmiten a través de la red bitcoin, son recopiladas por los mineros, e incluidas en los bloques, lo que las perpetúa en la cadena de bloques.

pool de transacciones

Se trata de la misma colección desordenada de transacciones, también conocida como “tanque de memoria” (mempool) y que no están todavía en ningún bloque de la cadena principal, pero para las cuales existen ya las transacciones que convalidan sus entradas.

Completitud de Turing

Un lenguaje de programación se denomina “Turing completo” si puede ejecutar los mismos programas que una máquina de Turing también puede ejecutar, con suficiente tiempo y memoria.

salida de transacción sin gastar (de sus siglas en inglés: UTXO)

Una UTXO es cualquier salida de una transacción que no ha sido gastada y que se puede gastar al referirse como la entrada de una nueva transacción.

cartera/monedero

Se trata de aquel software que contiene todas las direcciones de bitcoin de un usuario y sus correspondientes llaves secretas. Puede utilizarse para enviar, recibir y almacenar nuestros bitcoins.

Formato de Importación de Cartera (de sus siglas en inglés: WIF)

El WIF o el Formato de Importación de Cartera es un estándar de intercambio de datos diseñado para permitir la exportación e importación de una sola llave privada a la vez, con un banderín que indica si se utiliza o no una llave pública comprimida.

Introducción

¿Qué es Bitcoin?

Bitcoin es una colección de conceptos y tecnologías que construyen la base de un ecosistema de dinero digital. Unidades monetarias llamadas Bitcoin, son utilizadas para almacenarlo transmitir valor entre los participantes de la red de Bitcoin. Los usuarios de Bitcoin se comunican entre ellos usando el protocolo de bitcoin principalmente usando internet, aunque es posible usar otras redes de transmisión. El stack del protocolo de Bitcoin está disponible como software de código abierto, puede utilizarse en una gran variedad de computadoras personales, incluyendo computadoras portátiles y teléfonos inteligentes, creando una tecnología muy accesible.

Los usuarios pueden transferir bitcoins a través de la red para hacer casi cualquier cosa que se pueda hacer con monedas convencionales, incluida la compra y venta de bienes, el envío de dinero a personas u organizaciones o la extensión de créditos. Bitcoin se puede comprar, vender e intercambiar por otras monedas en casas de cambio especializadas. Bitcoin, en cierto sentido, es la forma perfecta de dinero para Internet porque es rápido, seguro y sin fronteras.

A diferencia de las divisas tradicionales, bitcoin es completamente virtual. No hay monedas físicas o incluso monedas digitales per se. Las monedas están implícitas en transacciones que transfieren valor del remitente al destinatario. Los usuarios de bitcoin poseen llaves que les permiten probar la propiedad de sus bitcoins en la red bitcoin. Con estas llaves, pueden firmar transacciones para desbloquear el valor y gastarlo transfiriéndolo a un nuevo propietario. Las llaves a menudo se almacenan en una billetera digital en la computadora o teléfono inteligente de cada usuario. La posesión de la llave que puede firmar una transacción es el único requisito previo para gastar los bitcoins, poniendo el control completamente en manos de cada usuario.

Bitcoin es un sistema distribuido en una red entre pares, de igual a igual. Como tal, no hay un servidor "central" o punto de control. Los bitcoins se crean a través de un proceso llamado "minería", que consiste en competir para encontrar soluciones a un problema matemático mientras se procesan las transacciones de bitcoin. Cualquier participante en la red bitcoin (es decir, cualquier persona que use un dispositivo que ejecute la pila completa de protocolos bitcoin) puede operar como minero, utilizando la potencia de procesamiento de su computadora para verificar y registrar transacciones. Cada 10 minutos, en promedio, un minero de bitcoin puede validar las transacciones de los últimos 10 minutos y es recompensado con nuevos bitcoins. Esencialmente, la minería de bitcoins descentraliza las funciones de emisión y compensación de divisas de un banco central y reemplaza la necesidad de cualquier banco central.

El protocolo bitcoin incluye algoritmos integrados que regulan la función de minería en toda la red. La dificultad de la tarea de procesamiento que los mineros deben realizar se ajusta dinámicamente para que, en promedio, alguien tenga éxito cada 10 minutos, independientemente de cuántos mineros (y cuánto procesamiento) compitan en cualquier momento. El protocolo también reduce a la mitad la velocidad a la que se crean nuevos bitcoins cada 4 años, y limita el número total de bitcoins que se crearán a un total fijo justo por debajo de 21 millones de monedas. El resultado es que el número de bitcoins en circulación sigue de cerca una curva fácilmente predecible que se aproxima a 21 millones para el año 2140. Debido a la tasa de emisión decreciente de bitcoin, a largo plazo, la moneda bitcoin es deflacionaria. Además, bitcoin no se puede inflar "imprimiendo" dinero nuevo más allá de la tasa de emisión esperada.

Tras bambalinas, bitcoin es también el nombre del protocolo, una red entre pares y una innovación de computación distribuida. La moneda bitcoin es tan solo la primera aplicación de esta invención. Bitcoin representa la culminación de décadas de investigación en criptografía y sistemas distribuidos e incluye cuatro innovaciones clave reunidas en una combinación única y poderosa. Bitcoin consiste en:

- Una red entre pares distribuida (el protocolo bitcoin)
- Un libro contable público (la cadena de bloques, o "blockchain")
- Un conjunto de reglas para la validación de transacciones de forma independiente y para la emisión de moneda (reglas de consenso)
- Un mecanismo para alcanzar un consenso global descentralizado sobre la cadena de bloques válida (algoritmo de Prueba-de-Trabajo)

Como desarrollador, veo a bitcoin como algo similar a la internet del dinero, una red para propagar valor y asegurar la propiedad de activos digitales mediante computación distribuida. Bitcoin es mucho más de lo que inicialmente aparenta.

En este capítulo, comenzaremos explicando algunos de los conceptos y términos principales, obteniendo el software

necesario y usando bitcoin para transacciones simples. En los siguientes capítulos, comenzaremos a desenvolver las capas de tecnología que hacen posible bitcoin y examinaremos el funcionamiento interno de la red y el protocolo bitcoin.

Monedas Digitales Antes de Bitcoin

El surgimiento de dinero digital viable se encuentra estrechamente relacionado a desarrollos en criptografía. Esto no es una sorpresa cuando uno considera los desafíos fundamentales involucrados en utilizar bits para representar valor intercambiable por bienes y servicios. Tres preguntas básicas para cualquiera que acepte dinero digital son:

1. ¿Puedo confiar en que el dinero es auténtico y no una falsificación?
2. ¿Puedo confiar en que el dinero digital solo puede gastarse una única vez? (también conocido como el problema del "doble gasto" o "double-spend").
3. ¿Puedo estar seguro de que nadie más aparte de mí puede alegar que ese dinero le pertenece?

Los emisores de moneda en papel están constantemente luchando contra el problema de la falsificación utilizando papeles y tecnología de impresión cada vez más sofisticados. El dinero físico aborda el problema del doble gasto fácilmente porque el mismo billete no puede estar en dos lugares a la vez. Por supuesto, el dinero convencional también a menudo se almacena y transmite digitalmente. En estos casos, los problemas de falsificación y doble gasto se manejan mediante la compensación de todas las transacciones electrónicas a través de las autoridades centrales que tienen una visión global de la moneda en circulación. Para el dinero digital, que no puede aprovechar las tintas esotéricas o las tiras holográficas, la criptografía proporciona la base para confiar en la legitimidad del reclamo de valor de un usuario. Específicamente, las firmas digitales criptográficas permiten a un usuario firmar un activo digital o una transacción que demuestre la propiedad de ese activo. Con la arquitectura adecuada, las firmas digitales también se pueden utilizar para atacar el problema del doble gasto.

Cuando la criptografía comenzaba a estar más ampliamente disponible y entendida a finales de la década de 1980, muchos investigadores comenzaron a intentar utilizar la criptografía para construir monedas digitales. Estos primeros proyectos de monedas digitales emitían dinero digital, generalmente respaldado por una moneda nacional o un metal precioso como el oro.

Aunque estas monedas digitales anteriores funcionaban, estaban centralizadas y, como resultado, eran fáciles de atacar por gobiernos y hackers. Las primeras monedas digitales utilizaban una cámara de compensación central para liquidar todas las transacciones a intervalos regulares, al igual que un sistema bancario tradicional. Desafortunadamente, en la mayoría de los casos, estas monedas digitales nacientes fueron atacadas por gobiernos preocupados y eventualmente litigaron hasta dejar de existir. Algunas fracasaron en colapsos espectaculares cuando su empresa matriz era liquidada abruptamente. Para ser robusto contra la intervención de antagonistas, ya sean gobiernos legítimos o elementos criminales, se necesitaba una moneda digital *descentralizada* para evitar un único punto de ataque. Bitcoin es un sistema de este tipo, descentralizado por diseño y libre de cualquier autoridad central o punto de control que pueda ser atacado o corrompido.

Historia de Bitcoin

Bitcoin se inventó en 2008 mediante la publicación de la memoria titulada "Bitcoin: A Peer-to-Peer Electronic Cash System,"[1] y escrita bajo el seudónimo de Satoshi Nakamoto (ver El Whitepaper (Libro Blanco) de Bitcoin por Satoshi Nakamoto). Nakamoto combinó varias invenciones previas tales como b-money y HashCash para crear un sistema de efectivo electrónico completamente descentralizado el cual no depende de una autoridad central para su emisión o la liquidación y validación de transacciones. La innovación clave fue el uso de un sistema de computación distribuida (llamado un algoritmo de "Prueba-de-Trabajo") para llevar a cabo una "elección" global cada 10 minutos, permitiéndole a la red descentralizada llegar a un consenso acerca del estado de transacciones. Esto resuelve de forma elegante el problema del doble gasto por el cual una misma unidad de moneda puede gastarse dos veces. Hasta entonces el problema del doble gasto era una limitación de las monedas digitales, que se resolvía mediante la verificación por cámara de compensación de todas las transacciones a través de una autoridad central (cámara de compensación).

La red de bitcoin se inició en 2009, basada en una implementación de referencia publicada por Nakamoto y luego revisada por muchos otros programadores. La implementación del algoritmo de Prueba-de-Trabajo (minería) que proporciona seguridad y resiliencia para bitcoin ha aumentado exponencialmente en potencia, y ahora supera la potencia de procesamiento combinada de las principales supercomputadoras del mundo. El valor de mercado total de Bitcoin en ocasiones superó los 135 mil millones de dólares estadounidenses, dependiendo del tipo de cambio de bitcoin a dólar. La transacción más grande procesada hasta ahora por la red fue de 400 millones de dólares estadounidenses, transmitida instantáneamente y procesada con una comisión de 1 dólar.

Satoshi Nakamoto se retiró del público en abril de 2011, dejando la responsabilidad de desarrollar el código y la red a un

creciente grupo de voluntarios. La identidad de la persona o personas detrás de bitcoin es aún desconocida. Sin embargo, ni Satoshi Nakamoto ni nadie más ejerce el control individual sobre el sistema bitcoin, que funciona sobre la base de principios matemáticos completamente transparentes, código libre y consenso entre los participantes. El invento en sí es innovador y ya ha derivado en una nueva ciencia en los campos de la computación distribuida, la economía y la econometría.

Una Solución a un Problema de Computación Distribuida

La invención de Satoshi Nakamoto también es una solución práctica y novedosa a un problema en la computación distribuida, conocido como el "Problema de los Generales Bizantinos". Brevemente, el problema consiste en tratar de acordar un curso de acción o el estado de un sistema intercambiando información a través de una red no confiable y potencialmente comprometida. La solución de Satoshi Nakamoto, que utiliza el concepto de Prueba-de-Trabajo para lograr el consenso sin una autoridad de confianza central, representa un avance en la computación distribuida y tiene una amplia aplicabilidad más allá de la moneda. Puede utilizarse para lograr un consenso en redes descentralizadas para demostrar la imparcialidad en elecciones, loterías, registros de activos, notarización digital, y más.

Usos de Bitcoin, Usuarios y Sus Historias

Bitcoin es una innovación en la antigua tecnología del dinero. En su esencia, el dinero simplemente facilita el intercambio de valor entre las personas. Por lo tanto, para entender completamente el bitcoin y sus usos, lo examinaremos desde la perspectiva de las personas que lo usan. Cada una de las personas y sus historias, como se enumeran aquí, ilustran uno o más casos de uso específicos. Los veremos a lo largo del libro:

Venta de artículos de bajo valor en Norteamérica

Alice vive en el área de la bahía del norte de California. Escuchó sobre bitcoins de sus amigos tecnófilos y quiere comenzar a usarlo. Seguiremos su historia mientras aprende sobre bitcoin, adquiere algunos y luego gasta parte de sus bitcoin para comprar una taza de café en Bob's Cafe en Palo Alto. Esta historia nos presentará el software, las casas de intercambio y las transacciones básicas desde la perspectiva de un consumidor minorista.

Venta de artículos de alto valor en Norteamérica

Carol es la dueña de una galería de arte en San Francisco. Vende pinturas costosas a cambio de bitcoins. Esta historia presentará los riesgos de un ataque de consenso del "51%" para vendedores de artículos de valor elevado.

Subcontratación de servicios al extranjero

Bob, el dueño de la cafetería en Palo Alto, está construyendo un nuevo sitio web. Ha contratado a un desarrollador web indio, Gopesh, que vive en Bangalore, India. Gopesh ha aceptado ser pagado en bitcoin. Esta historia examinará el uso de bitcoin para la subcontratación, contrato de servicios y transferencias electrónicas internacionales.

Tienda virtual

Gabriel es un joven empresario en Río de Janeiro, que administra una pequeña tienda web que vende camisetas, tazas de café y pegatinas con la marca de bitcoin. Gabriel es demasiado joven para tener una cuenta bancaria, pero sus padres están fomentando su espíritu emprendedor.

Donaciones de caridad

Eugenia es la directora de una organización benéfica infantil en Filipinas. Recientemente, descubrió bitcoin y quiere usarlo para llegar a un nuevo grupo de donantes extranjeros y nacionales para recaudar fondos para su organización benéfica. También está investigando formas de usar bitcoin para distribuir fondos rápidamente a áreas de necesidad. Esta historia mostrará el uso de bitcoin para la recaudación global de fondos a través de monedas y fronteras, y el uso de un libro de contabilidad abierto para la transparencia en organizaciones de caridad.

Importación/exportación

Mohammed es un importador de electrónica en Dubai. Está intentando usar bitcoin para comprar productos electrónicos de Estados Unidos y China para importarlos a los Emiratos Árabes Unidos y acelerar el proceso de pago de importaciones. Esta historia mostrará cómo se puede utilizar bitcoin para grandes pagos internacionales de empresa a empresa vinculados a bienes físicos.

Minería en bitcoin

Jing es un estudiante de ingeniería informática en Shanghai. Ha construido una plataforma de "minería" para extraer

bitcoin utilizando sus habilidades de ingeniería para complementar sus ingresos. Esta historia examinará la base "industrial" de bitcoin: el equipo especializado utilizado para asegurar la red bitcoin y emitir nueva moneda .

Cada una de estas historias se basa en personas e industrias reales que actualmente utilizan bitcoin para crear nuevos mercados, nuevas industrias y soluciones innovadoras para los problemas económicos globales.

Primeros Pasos

Bitcoin es un protocolo al que se puede acceder mediante una aplicación cliente que habla el protocolo. Una "billetera bitcoin" es la interfaz de usuario más común para el sistema bitcoin, al igual que un navegador web es la interfaz de usuario más común para el protocolo HTTP. Hay muchas implementaciones y marcas de carteras de bitcoin, al igual que hay muchas marcas de navegadores web (por ejemplo, Chrome, Safari, Firefox e Internet Explorer). Y al igual que todos tenemos nuestros navegadores favoritos (Mozilla Firefox, ¡Guay!) y nuestros villanos (Internet Explorer, ¡Buff!), las carteras de bitcoin varían en calidad, rendimiento, seguridad, privacidad y confiabilidad. También hay una implementación de referencia del protocolo de bitcoin que incluye una cartera, conocida como "Cliente Satoshi" o "Bitcoin Core", que se deriva de la implementación original escrita por Satoshi Nakamoto.

Elegir una Cartera Bitcoin

Las carteras bitcoin son una de las aplicaciones desarrolladas más activamente en el ecosistema de bitcoin. Hay una competencia intensa, y mientras que probablemente se está desarrollando una nueva cartera en este momento, varias carteras del año pasado ya no se mantienen activamente. Muchas carteras se enfocan en plataformas específicas o usos específicos y algunas son más adecuadas para principiantes, mientras que otras están llenas de características para usuarios avanzados. La elección de una cartera es muy subjetiva y depende del uso y la experiencia del usuario. Por lo tanto, es imposible recomendar una marca o cartera específica. Sin embargo, podemos clasificar las carteras de bitcoin de acuerdo con su plataforma y función, y proporcionar cierta claridad sobre los diferentes tipos de carteras que existen. Mejor aún, mover llaves o semillas entre las carteras bitcoin es relativamente fácil, por lo que vale la pena probar varias carteras diferentes hasta que encuentres una que se ajuste a tus necesidades.

Las carteras bitcoin se pueden clasificar de la siguiente manera, según la plataforma:

Cartera de PC

Una cartera de PC (o "de escritorio") fue el primer tipo de cartera de bitcoin creada como implementación de referencia y muchos usuarios ejecutan hoy día carteras de PC por las características, la autonomía y el control que ofrecen. Sin embargo, la ejecución en sistemas operativos de uso general, como Windows y Mac OS, tiene ciertas desventajas de seguridad, ya que estas plataformas a menudo son inseguras y están precariamente configuradas.

Cartera móvil

Una cartera móvil es el tipo más común de cartera bitcoin. Al ejecutarse en sistemas operativos de teléfonos inteligentes como Apple iOS y Android, estas carteras a menudo son una excelente opción para los nuevos usuarios. Muchos están diseñados para ser simples y fáciles de usar, pero también existen carteras móviles con todas las funciones para usuarios avanzados.

Cartera web

Las carteras web se acceden a través de un navegador web y se almacena la cartera del usuario en un servidor propiedad de un tercero. Esto es similar al correo web, ya que depende completamente de un servidor externo. Algunos de estos servicios funcionan ejecutando código del cliente en el navegador del usuario, manteniendo así las llaves de bitcoin en sus manos. La mayoría, sin embargo, presentan un riesgo al delegar el control de las llaves de bitcoin de los usuarios a cambio de su facilidad de uso. No es aconsejable almacenar grandes cantidades de bitcoin en sistemas de terceros.

Cartera de hardware

Las carteras de hardware son dispositivos que operan una cartera bitcoin segura e independiente en un hardware especial. Se acceden a través de USB con un navegador web de escritorio o mediante comunicación de campo cercano (NFC) en un dispositivo móvil. Al gestionar todas las operaciones relacionadas con bitcoin en el hardware especializado, estas carteras se consideran muy seguras y adecuadas para almacenar grandes cantidades de bitcoin.

Cartera de papel

Las llaves que controlan bitcoin también se pueden imprimir y almacenarlas a largo plazo. Se conocen como carteras de papel aunque se pueden usar otros materiales (madera, metal, etc.). Las carteras de papel ofrecen un medio de baja

tecnología pero muy seguro para almacenar bitcoin a largo plazo. El almacenamiento sin conexión también se conoce como *almacenamiento en frío*.

Las carteras bitcoin también se pueden clasificar por su grado de autonomía y por cómo interactúan con la red bitcoin:

Cliente completo

Un cliente completo, o "nodo completo", es un cliente que almacena la totalidad del historial de transacciones de bitcoin (cada transacción realizada por cada usuario, desde siempre), administra carteras de usuarios, y puede iniciar transacciones directamente en la red bitcoin. Un nodo completo gestiona todos los aspectos del protocolo y puede validar independientemente la cadena de bloques completa y cualquier transacción. Un cliente completo consume gran cantidad de recursos informáticos (por ejemplo, más de 125 GB de disco, 2 GB de RAM) pero ofrece autonomía completa y verificación independiente de transacciones.

Cliente ligero

Un cliente ligero, también conocido como cliente de verificación de pago simple (SPV), se conecta a los nodos completos de bitcoin (mencionados anteriormente) para acceder a la información de las transacciones de bitcoin, pero almacenan la cartera del usuario de forma local, e independientemente crean, validan y transmiten las transacciones. Los clientes ligeros interactúan directamente con la red bitcoin, sin intermediarios.

Cliente API de terceros

Un cliente API de terceros interactúa con bitcoin a través de un sistema de interfaces de programación de aplicaciones (API) de terceros, en lugar de conectarse a la red bitcoin directamente. La cartera puede almacenarse por el usuario o por servidores de terceros, pero todas las transacciones pasan por un tercero.

Combinando estas categorizaciones, muchas carteras de bitcoin se agrupan en varios grupos, y las tres más comunes son el cliente completo de escritorio, la cartera ligera para dispositivos móviles y la cartera web de terceros. La separación entre las diferentes categorías a menudo son borrosas, ya que muchas carteras se ejecutan en múltiples plataformas y pueden interactuar con la red de diferentes maneras.

En este este libro mostraremos el uso de varios clientes bitcoin que se pueden descargar por internet, desde la implementación de referencia (Bitcoin Core) hasta carteras móviles y web. Algunos de los ejemplos requerirán el uso de Bitcoin Core, que, además de ser un cliente completo, también expone APIs a los servicios de cartera, red y transacciones. Si tienes intención de explorar las interfaces de programa hacia el sistema de bitcoin, necesitarás ejecutar Bitcoin Core o uno de los clientes alternativos (ver Clientes Alternativos, Bibliotecas y Kits de Herramientas).

Inicio Rápido

Alice, a quien ya presentamos en Usos de Bitcoin, Usuarios y Sus Historias, no es un usuario técnico y solo recientemente escuchó acerca de bitcoin de su amigo Joe. En una fiesta, Joe está explicando con entusiasmo una vez más el bitcoin a todos a su alrededor y ofrece una demostración. Intrigada, Alice le pregunta cómo puede comenzar a usar bitcoin. Joe dice que una cartera móvil es lo mejor para los nuevos usuarios y recomienda algunas de sus carteras favoritas. Alice descarga "Mycelium" para Android y lo instala en su teléfono.

Cuando Alice ejecuta Mycelium por primera vez, como sucede con muchas carteras de bitcoin, la aplicación crea automáticamente una nueva cartera para ella. Alice ve la cartera en su pantalla, como se muestra en La Cartera Móvil Mycelium (nota: no envíes bitcoin a esta dirección de muestra, se perderá para siempre).


12:12
ACCOUNTS
BALANCE
TRANSACTIONS
AD
Alice
1Cdid9KFAaat
wczBwBttQcwX
YCpvK8h7FK
0 mBTC
0.00 USD
Receive
1 BTC ~ USD 449.08 (BitcoinAverage)
Buy / Sell Bitcoin
v2.5.9

Figure 1. La Cartera Móvil Mycelium

La parte más importante de esta pantalla es la *dirección bitcoin* de Alice. En la pantalla aparece como una larga cadena de letras y números: 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK. Junto a la dirección bitcoin de la cartera hay un código QR, una especie de código de barras que contiene la misma información en un formato que puede ser escaneado por la cámara de un teléfono inteligente. El código QR es el cuadrado con un patrón de puntos blancos y negros. Alice puede copiar la dirección bitcoin o el código QR en su portapapeles tocando el código QR o el botón Recibir. En la mayoría de las carteras, tocar el código QR también lo ampliará, de modo que pueda ser escaneado más fácilmente por la cámara de un teléfono inteligente.

TIP

Una dirección bitcoin comienza con 1, 3 o bc1. Al igual que las direcciones de correo electrónico, se pueden compartir con otros usuarios de bitcoin que pueden usarlas para enviar bitcoins directamente a nuestra billetera. No hay nada sensible, desde una perspectiva de seguridad, sobre la dirección de bitcoin. Se pueden publicar en cualquier lugar sin arriesgar la seguridad de la cuenta. A diferencia de las direcciones de correo electrónico, se pueden crear nuevas direcciones con la frecuencia que se desee, todo lo cual dirigirá los fondos a nuestra billetera. De hecho, muchas billeteras modernas crean automáticamente una nueva dirección para cada transacción para maximizar la privacidad. Una billetera es simplemente una colección de direcciones y de las llaves que desbloquean los fondos que administra.

Alice ya está lista para recibir fondos. Su aplicación de cartera generó aleatoriamente una llave privada (descrita con más detalle en Llaves Privadas) junto con su correspondiente dirección bitcoin. En este punto, su dirección bitcoin no es conocida por la red bitcoin ni está "registrada" en ninguna parte del sistema bitcoin. Su dirección bitcoin es simplemente un número que se corresponde con una llave que puede usar para controlar el acceso a los fondos. Se generó de manera independiente por su cartera, sin referencia ni registro con ningún servicio. De hecho, en la mayoría de las carteras, no hay asociación entre la dirección bitcoin y cualquier otra información que lo pueda identificar externamente, incluida la identidad del usuario. Hasta el momento en que se hace referencia a esta dirección como destinatario de valor en una transacción publicada en el libro de contabilidad de bitcoin, la dirección bitcoin es simplemente una más de la gran cantidad de direcciones posibles que son válidas en bitcoin. Solo cuando se ha asociado a una transacción, se convierte en una de las direcciones conocidas en la red.

Alice ya está lista para comenzar a usar su nueva cartera bitcoin.

Obteniendo Tu Primer Bitcoin

La primera tarea y, con frecuencia, la más difícil para los nuevos usuarios es adquirir algunos bitcoins. A diferencia de otras monedas extranjeras, aún no puedes comprar bitcoin en un banco ni en un quiosco de divisas extranjeras.

Las transacciones bitcoin son irreversibles. La mayoría de las redes de pago electrónico, como tarjetas de crédito, tarjetas de débito, PayPal y transferencias de cuentas bancarias, son reversibles. Para alguien que vende bitcoin, esta diferencia presenta un riesgo muy alto de que el comprador revierta el pago electrónico después de haber recibido bitcoin, defraudando así al vendedor. Para mitigar este riesgo, las compañías que aceptan pagos electrónicos tradicionales a cambio de bitcoin generalmente requieren que los compradores se sometan a una verificación de identidad y controles de solvencia, que pueden tardar varios días o semanas. Para un nuevo usuario, esto significa que no puedes comprar bitcoin instantáneamente con una tarjeta de crédito. Sin embargo, con un poco de paciencia y pensamiento creativo, no necesitarás hacerlo así.

Aquí hay algunos métodos para obtener bitcoin como nuevo usuario:

- Encuentre un amigo que tenga bitcoin y cómprele algo directamente. Muchos usuarios de bitcoin comienzan de esta manera. Este método es el menos complicado. Una forma de conocer gente con bitcoin es asistir a una reunión local de bitcoin que se detalla en Meetup.com.
- Usa un servicio clasificado como localbitcoins.com para encontrar un vendedor en tu zona a quien comprar bitcoin por dinero en efectivo en una transacción persona a persona.
- Gana bitcoin vendiendo algún producto o servicio por bitcoin. Si eres un programador, vende tus habilidades de programación. Si eres peluquero, corta el pelo por bitcoin.
- Use un cajero automático bitcoin en su ciudad. Un cajero automático bitcoin es una máquina que acepta efectivo y envía bitcoin al monedero de su teléfono inteligente. Encuentre un cajero automático de bitcoin cerca de usted utilizando un mapa en línea de Coin ATM Radar.

- Use una casa de cambio de moneda bitcoin vinculado a su cuenta bancaria. Muchos países ahora tienen intercambios de divisas que ofrecen un mercado para que compradores y vendedores intercambien bitcoins con moneda local. Los servicios de listado de tipos de cambio, como BitcoinAverage, a menudo muestran una lista de intercambios de bitcoins para cada moneda.

> **TIP**
>
> Una de las ventajas de bitcoin sobre otros sistemas de pago es que, cuando se usa correctamente, ofrece a los usuarios mucha más privacidad. La adquisición, almacenamiento y gasto de bitcoin no requiere que divulgues información confidencial y de identificación personal a terceros. Sin embargo, donde bitcoin toca sistemas tradicionales, como los intercambios de divisas, a menudo se aplican las regulaciones nacionales e internacionales. Para intercambiar bitcoin por tu moneda nacional, a menudo se te pedirá que proporciones un documento de identidad e información bancaria. Los usuarios deben tener en cuenta que una vez que una dirección bitcoin se vincula a una identidad, todas las transacciones de bitcoin asociadas también son fáciles de identificar y rastrear. Esta es una de las razones por las que muchos usuarios optan por mantener cuentas de intercambio dedicadas sin vinculación a sus carteras.

Alice conoció bitcoin a través de un amigo, por lo que ya tiene una manera fácil de adquirir su primer bitcoin. A continuación, veremos cómo compra bitcoins a su amigo Joe y cómo Joe envía el bitcoin a su cartera.

Encontrando el Precio Actual de Bitcoin

Antes de que Alice pueda comprar bitcoin a Joe, tienen que acordar el tipo de cambio entre bitcoin y dólares estadounidenses. Esto plantea una pregunta común para aquellos que son nuevos en bitcoin: "¿Quién establece el precio de bitcoin?" La respuesta rápida es que el precio lo establecen los mercados.

Bitcoin, como la mayoría de las otras monedas, tiene un tipo de cambio flotante. Eso significa que el valor de bitcoin con respecto a cualquier otra moneda fluctúa con la oferta y la demanda en los distintos mercados donde se comercializa. Por ejemplo, el "precio" de bitcoin en dólares estadounidenses se calcula en cada mercado en función del comercio más reciente de bitcoin y dólares estadounidenses. Como tal, el precio tiende a fluctuar minuciosamente varias veces por segundo. Un servicio de precios agregará los precios de varios mercados y calculará un promedio ponderado por volumen que representa el tipo de cambio de mercado de un par de divisas (por ejemplo, BTC/USD).

Hay cientos de aplicaciones y sitios web que pueden proporcionar el tipo de cambio actual en el mercado. Estos son algunos de los más populares:

Bitcoin Average

Un sitio que proporciona una vista simple del promedio ponderado por volumen para cada moneda.

CoinCap

Un servicio que enumera la capitalización de mercado y los tipos de cambio de cientos de criptomonedas, incluido bitcoin.

Chicago Mercantile casa de cambio Bitcoin Reference Rate

Una tasa de referencia que se puede utilizar para referencia institucional y contractual, proporcionada como parte de los datos de inversión suministrados por el CME.

Además de estos sitios y aplicaciones, la mayoría de las carteras bitcoin convierten automáticamente las cantidades entre bitcoin y otras monedas. Joe usará su cartera para convertir el precio automáticamente antes de enviar bitcoin a Alice.

Enviando y Recibiendo Bitcoin

Alice ha decidido cambiar 10 dólares estadounidenses por bitcoin, para no arriesgar demasiado dinero en esta nueva tecnología. Ella da a Joe 10 dólares en efectivo, abre su aplicación de cartera Mycelium y selecciona Recibir. Esto muestra un código QR de la primera dirección bitcoin de Alice.

Después, Joe selecciona Enviar en la cartera de su teléfono inteligente y aparece una pantalla con dos entradas:

- Una dirección bitcoin de destino
- La cantidad a enviar, en bitcoin (BTC) o en su moneda local (USD)

En el campo de entrada de la dirección bitcoin, hay un pequeño icono que parece un código QR. Esto permite a Joe escanear el código de barras con la cámara de su teléfono inteligente para que no tenga que teclear la dirección bitcoin de

Alice, que es bastante larga y difícil de escribir. Joe toca el ícono del código QR y activa la cámara del teléfono inteligente, escaneando el código QR que se muestra en el teléfono inteligente de Alice.

Ahora, Joe ya tiene la dirección bitcoin de Alice establecida como destinatario. Joe introduce la cantidad de 10 dólares estadounidenses y su cartera la convierte accediendo al tipo de cambio más reciente de un servicio en línea. El tipo de cambio en ese momento es de 100 dólares estadounidenses por bitcoin, por lo que 10 dólares estadounidenses valen 0,10 bitcoins (BTC), o 100 milibitcoins (mBTC) como se muestra en la captura de pantalla de la cartera de Joe (ver Pantalla de envío de la cartera bitcoin móvil de Airbitz).

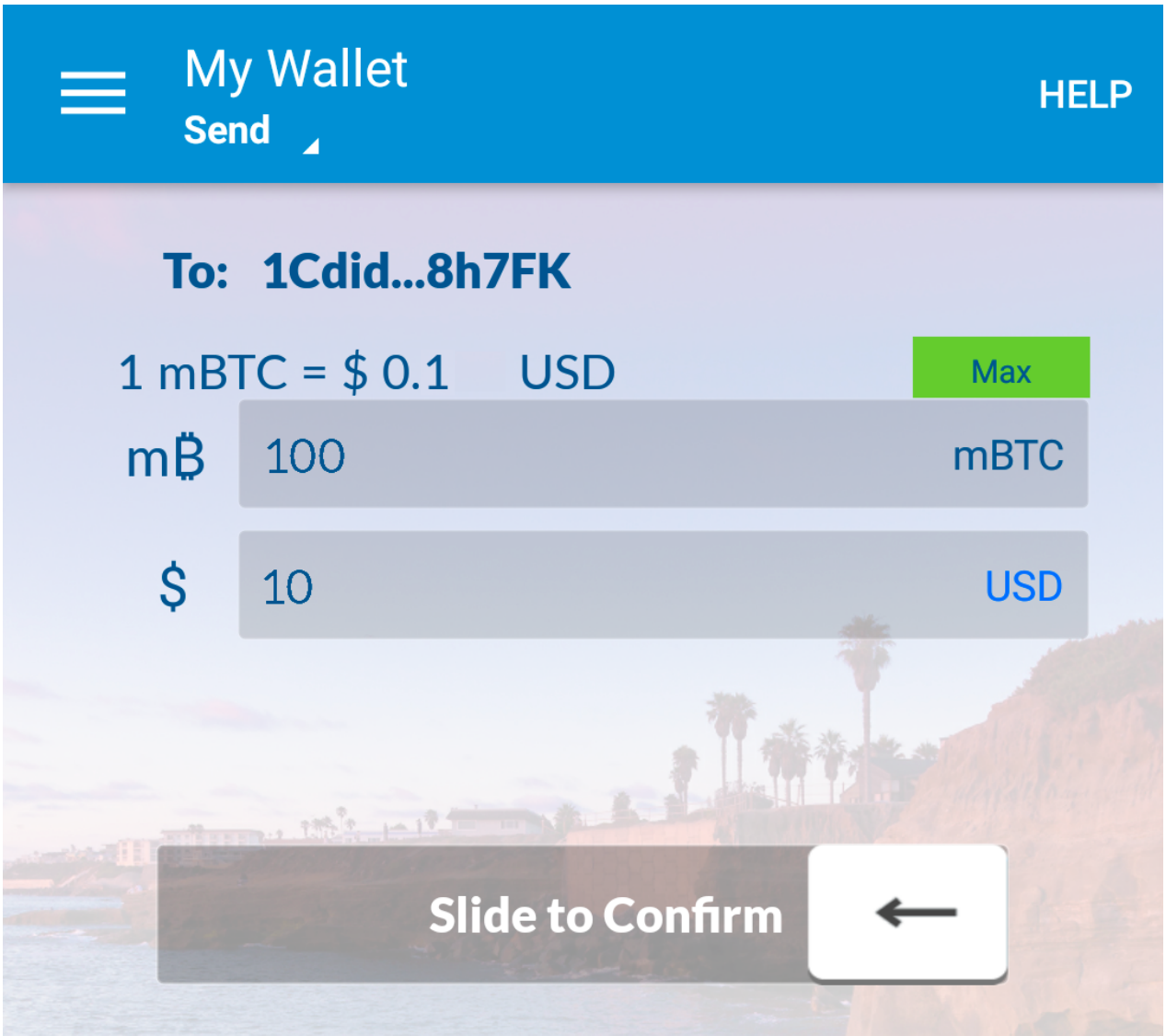


Figure 2. Pantalla de envío de la cartera bitcoin móvil de Airbitz

Después, Joe verifica cuidadosamente que ha introducido la cantidad correcta, porque está a punto de transmitir dinero y los errores son irreversibles. Después de verificar la dirección y el monto, presiona Enviar para transmitir la transacción. La cartera móvil bitcoin de Joe construye una transacción que asigna 0.10 BTC a la dirección proporcionada por Alice, que obtiene los fondos de origen de la cartera de Joe y firma la transacción con las llaves privadas de Joe. Esto le dice a la red bitcoin que Joe ha autorizado una transferencia de valor a la nueva dirección de Alice. A medida que la transacción se transmite a través del protocolo entre pares, se propaga rápidamente a través de la red bitcoin. En menos de un segundo, la mayoría de los nodos bien conectados en la red reciben la transacción y ven la dirección de Alice por primera vez.

Mientras tanto, la cartera de Alice está constantemente "escuchando" las transacciones publicadas en la red bitcoin, buscando alguna que coincida con las direcciones de sus carteras. Unos segundos después de que la billetera de Joe transmita la transacción, la billetera de Alice indicará que está recibiendo 0.10 BTC.

Confirmaciones

Al principio, la dirección de Alice mostrará la transacción de Joe como "Sin confirmar". Esto significa que la transacción se ha propagado a la red, pero aún no se ha registrado en el libro de contabilidad de bitcoin, conocido como blockchain o

cadena de bloques. Para confirmarse, una transacción debe incluirse en un bloque y agregarse a la cadena de bloques, lo que ocurre cada 10 minutos en promedio. En términos financieros tradicionales, esto se conoce como *compensación*. Para obtener más detalles sobre la propagación, validación y compensación (confirmación) de transacciones de bitcoin, consulte Minería y Consenso.

Alice es ahora la orgullosa propietaria de 0.10 BTC que puede gastar. En el siguiente capítulo veremos su primera compra con bitcoin y examinaremos con más detalle las tecnologías subyacentes de transacción y propagación.

¿Cómo funciona Bitcoin?

Transacciones, Bloques, Minería, y la Cadena de Bloques

El sistema bitcoin, a diferencia de los sistemas bancarios y de pago tradicionales, se basa en la confianza descentralizada. En lugar de una autoridad central de confianza, en bitcoin, la confianza se logra como una propiedad emergente de las interacciones de diferentes participantes en el sistema bitcoin. En este capítulo, examinaremos bitcoin desde un nivel alto al rastrear una sola transacción a través del sistema bitcoin y veremos cómo se hace "confiable" y aceptada por el mecanismo de consenso distribuido de bitcoin y finalmente se registra en la cadena de bloques, el libro de contabilidad distribuido de todas las transacciones. Los capítulos subsiguientes ahondarán en la tecnología detrás de las transacciones, la red y la minería.

Descripción General de Bitcoin

En el diagrama de resumen que se muestra en Descripción general de Bitcoin, vemos que el sistema bitcoin consiste en usuarios con carteras que contienen llaves, transacciones que se propagan a través de la red y mineros que producen (a través de cómputo competitivo) la cadena de bloques de consenso, que es el libro de contabilidad autorizado de todas las transacciones.

Cada ejemplo en este capítulo se basa en una transacción real realizada en la red bitcoin, simulando las interacciones entre los usuarios (Joe, Alice, Bob y Gopesh) al enviar fondos de una cartera a otra. Mientras rastreamos una transacción a través de la red bitcoin hasta la cadena de bloques, usaremos un *explorador de la cadena de bloques* para visualizar cada paso. Un explorador de la cadena de bloques es una aplicación web que funciona como un motor de búsqueda de bitcoin, ya que permite buscar direcciones, transacciones y bloques, y ver las relaciones y los flujos entre ellos.

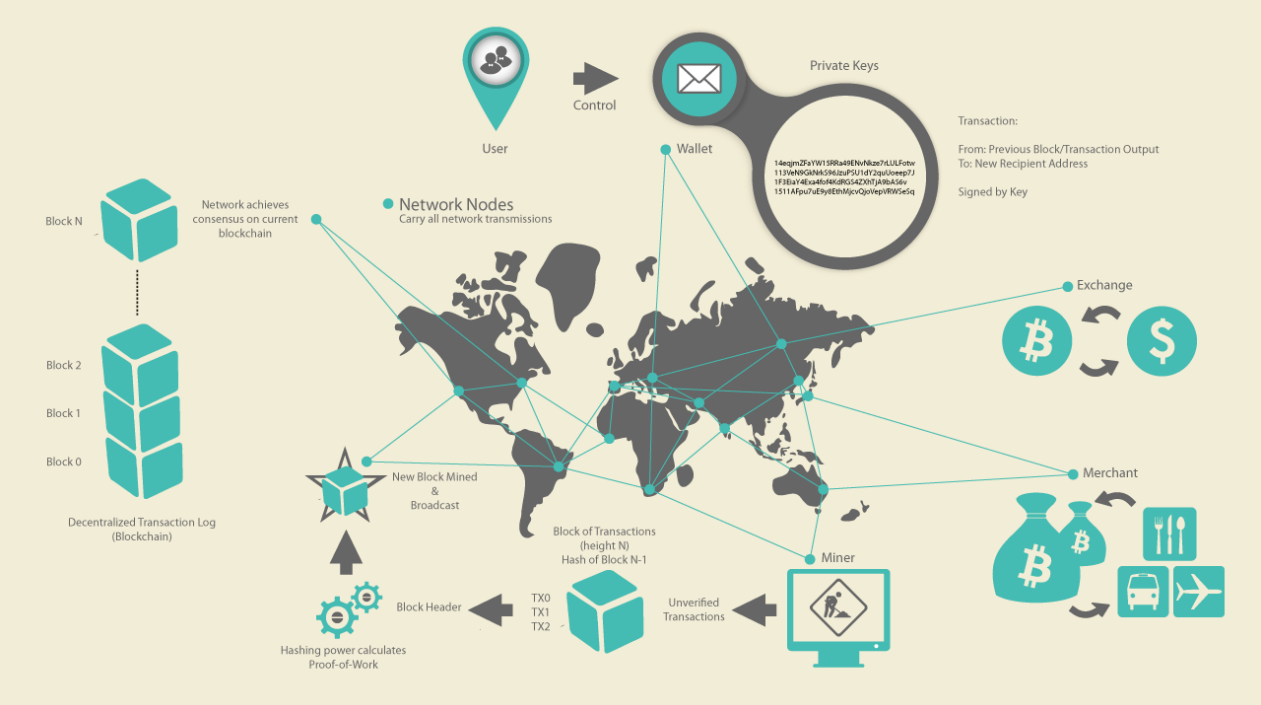


Figure 3. Descripción general de Bitcoin

Algunos exploradores populares de la cadena de bloques son:

- BlockCypher Explorer
- blockchain.info
- BitPay Insight
- Blockstream Explorer

Cada uno de estos tiene una función de búsqueda que puede tomar una dirección bitcoin, un hash de transacción, un número de bloque o un hash de bloque y recuperar la información correspondiente de la red bitcoin. Con cada ejemplo de transacción o bloque, te proporcionaremos una URL para que puedas buscarla y estudiarla en detalle.

Comprando una Taza de Café

Alice, presentada en el capítulo anterior, es una nueva usuaria que acaba de adquirir su primer bitcoin. En Obteniendo Tu Primer Bitcoin, Alice se reunió con su amigo Joe para intercambiar algo de dinero por bitcoin. La transacción creada por Joe financió la billetera de Alice con 0.10 BTC. Ahora Alice realizará su primera transacción, comprando una taza de café en la cafetería Bob's en Palo Alto, California.

Bob's Cafe recientemente comenzó a aceptar pagos de bitcoin al agregar una opción de bitcoin a su sistema de punto de venta. Los precios en Bob's Cafe están listados en la moneda local (dólares estadounidenses), pero en la caja, los clientes tienen la opción de pagar en dólares o bitcoins. Alice hace su pedido de una taza de café y Bob la introduce en máquina registradora, como lo hace para todas las transacciones. El punto de venta convierte automáticamente el precio total de dólares estadounidenses a bitcoin al tipo de cambio vigente y muestra el precio en ambas monedas:

```
Total:
$1.50 USD
0.015 BTC
```

Bob dice, "Son 1.50 dólares, o 15 milibits".

El sistema de punto de venta de Bob también creará automáticamente un código QR especial que contiene una *solicitud de pago* (ver Código QR de solicitud de pago).

A diferencia del código QR que solo contiene una dirección bitcoin de destino, una solicitud de pago es una URL codificada en QR que contiene una dirección de destino, la cantidad a pagar y una descripción genérica como "Bob's Cafe". Esto permite a la aplicación de cartera bitcoin rellenar la información usada para enviar el pago mientras muestra una descripción legible para el usuario. Puedes escanear el código QR con una aplicación de cartera bitcoin para ver lo que vería Alice.

Figure 4. Código QR de solicitud de pago

TIP	Trata de escanear esto con tu cartera para ver la dirección y la cantidad, pero NO ENVÍES DINERO.

El código QR de la solicitud de pago codifica la siguiente URL, definida en BIP-21:

```
bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?
cantidad=0.015&
label=Bob%27s%20Cafe&
message=Purchase%20at%20Bob%27s%20Cafe

Componentes de la URL

Una dirección bitcoin: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
La cantidad a pagar: "0.015"
Una etiqueta para la dirección del receptor: "Bob's Cafe"
Una descripción del pago: "Compra en Bob's Cafe"
```

Alice usa su teléfono inteligente para escanear el código de barras en la pantalla. Su teléfono inteligente muestra un pago de 0.0150 BTC a Bob's Cafe y selecciona Enviar para autorizar el pago. En unos pocos segundos (aproximadamente la misma cantidad de tiempo que una autorización de tarjeta de crédito), Bob ve la transacción en el caja registradora, completando la transacción.

En las siguientes secciones, examinaremos esta transacción con más detalle. Veremos cómo lo construyó la cartera de Alice, cómo se propagó a través de la red, cómo se verificó y, finalmente, cómo Bob puede gastar esa cantidad en transacciones posteriores.

NOTE	bitcoinLa red bitcoin puede realizar transacciones en valores fraccionarios, por ejemplo, desde milibitcoin (1/1000 de un bitcoin) hasta 1/100,000.000 de un bitcoin, que se conoce como satoshi. A lo largo de este

> libro, usaremos el término "bitcoin" para referirnos a cualquier cantidad de moneda bitcoin, desde la unidad más pequeña (1 satoshi) hasta el número máximo (21,000,000) de todos los bitcoin que serán minados.

Puedes examinar la transacción de Alice a Bob's Cafe en la cadena de bloques usando un explorador de bloques (Examina la transacción de Alice en blockchain.info):

Example 1. Examina la transacción de Alice en blockchain.info

```
https://blockchain.info/tx/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2
```

Transacciones Bitcoin

En términos simples, una transacción comunica a la red que el propietario de algún valor de bitcoin ha autorizado la transferencia de ese valor a otro propietario. El nuevo propietario ahora puede gastar el bitcoin creando otra transacción que autorice la transferencia a otro propietario, y así sucesivamente, en una cadena de propiedad.

Entradas y Salidas de Transacción

Las transacciones son como filas en un libro de contabilidad de doble entrada. Cada transacción contiene una o más "entradas", que son como débitos contra una cuenta de bitcoin. En el otro lado de la transacción, hay una o más "salidas", que son como créditos agregados a una cuenta de bitcoin. Las entradas y salidas (débitos y créditos) no necesariamente suman la misma cantidad. En cambio, las salidas suman un poco menos que las entradas y la diferencia representa una *comisión de transacción* implícita, que es un pequeño pago que cobra el minero que incluye la transacción en el libro de contabilidad. Una transacción bitcoin se muestra como una entrada del libro de contabilidad en Transacciones como contabilidad de doble entrada.

La transacción también contiene la prueba de propiedad para cada cantidad de bitcoin (entradas) cuyo valor se está gastando, en forma de una firma digital del propietario, que puede ser validada independientemente por cualquier persona. En términos de bitcoin, "gastar" es firmar una transacción que transfiere el valor de una transacción anterior a un nuevo propietario identificado por una dirección bitcoin.

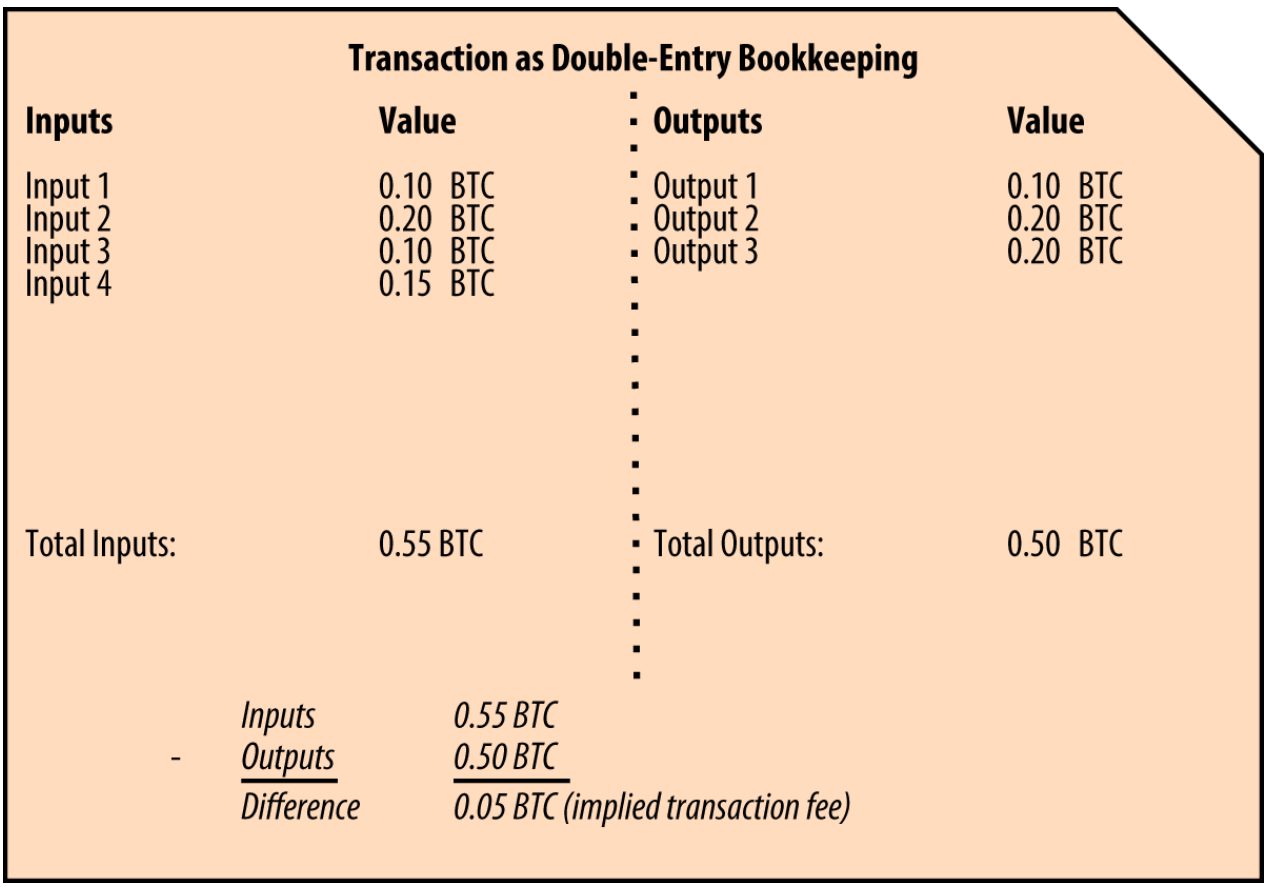


Figure 5. Transacciones como contabilidad de doble entrada

El pago de Alice a Bob's Cafe utiliza la salida de una transacción anterior como entrada. En el capítulo anterior, Alice recibió bitcoin de su amigo Joe a cambio de dinero en efectivo. Esa transacción creó un valor de bitcoin bloqueado por la llave de Alice. Su nueva transacción a Bob's Cafe hace referencia a la transacción anterior como entrada y crea nuevas salidas para pagar la taza de café y recibir el cambio. Las transacciones forman una cadena, donde las entradas de la última transacción corresponden a las salidas de transacciones anteriores. La llave de Alice proporciona la firma que desbloquea esas salidas de la transacción anterior, lo que demuestra a la red bitcoin que es la propietaria de los fondos. Ella registra el pago por el café a la dirección de Bob, por lo que "obstruye" esa salida con el requisito de que Bob produzca una firma para gastar esa cantidad. Esto representa una transferencia de valor entre Alice y Bob. Esta cadena de transacciones, desde Joe a Alice a Bob, se ilustra en Una cadena de transacciones, donde la salida de una transacción es la entrada de la siguiente transacción.

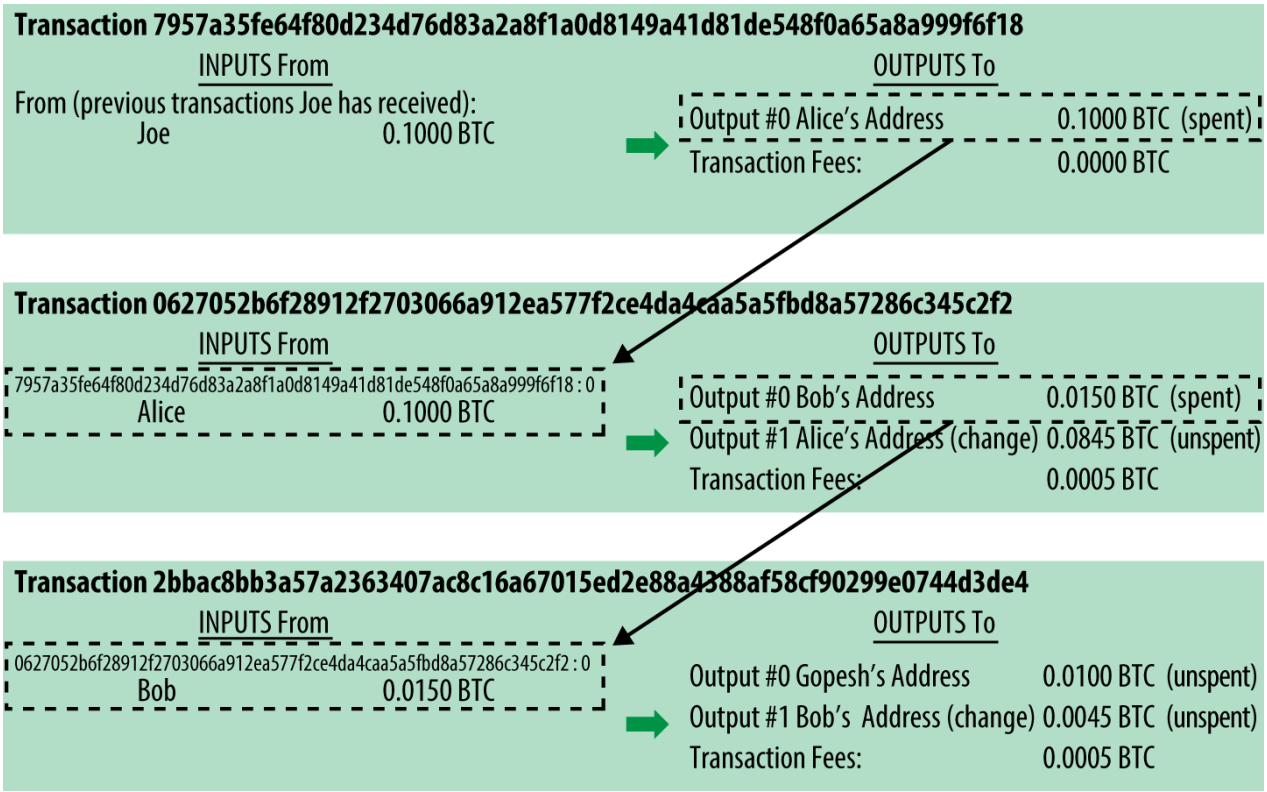


Figure 6. Una cadena de transacciones, donde la salida de una transacción es la entrada de la siguiente transacción

Creando el Cambio

Muchas transacciones bitcoin incluirán salidas que hacen referencia tanto a una dirección del nuevo propietario como a una dirección del propietario actual, llamada la dirección *de cambio*. Esto se debe a que las entradas de transacciones, como los billetes, no se pueden dividir. Si compras un artículo de 5 dólares en una tienda pero usas un billete de 20 dólares para pagar el artículo, esperarás recibir un cambio de 15 dólares. El mismo concepto se aplica a las entradas de transacción de bitcoin. Si compraste un artículo que cuesta 5 bitcoin, pero solo tenías una entrada de 20 bitcoin, enviarías una salida de 5 bitcoin al propietario de la tienda y una salida de 15 bitcoin a ti mismo como cambio (menos cualquier comisión de transacción aplicable). Es importante destacar que la dirección de cambio no tiene por qué ser la misma dirección que la de la entrada y, de hecho, por razones de privacidad, suele ser una dirección nueva de la cartera del propietario.

Diferentes carteras pueden usar diferentes estrategias al agregar entradas para realizar un pago solicitado por el usuario. Podrían agregar muchas entradas pequeñas, o usar una que sea igual o mayor que el pago deseado. A menos que la cartera pueda agregar entradas de tal manera que coincida exactamente con el pago deseado más las comisiones de transacción, la cartera deberá generar algún cambio. Esto es muy similar a cómo las personas manejan el efectivo. Si siempre usas el billete más grande en tu bolsillo, terminarás con un bolsillo lleno de cambio suelto. Si solo usas el cambio suelto, siempre tendrás solo billetes grandes. La gente subconscientemente encuentra un equilibrio entre estos dos extremos, y los desarrolladores de carteras bitcoin se esfuerzan por programar este equilibrio.

En resumen, las *transaccciones* mueven el valor de las *entradas de transacción* a las *salidas de transacción*. Una entrada es una referencia a la salida de una transacción anterior, que muestra de dónde proviene el valor. Una salida de transacción dirige un valor específico a la dirección bitcoin de un nuevo propietario y puede incluir una salida de cambio de vuelta al propietario original. Las salidas de una transacción pueden usarse como entradas en una nueva transacción, creando así

una cadena de propiedad a medida que el valor se mueve de propietario a propietario (ver Una cadena de transacciones, donde la salida de una transacción es la entrada de la siguiente transacción).

Formas Comunes de Transacción

La forma más común de transacción es un pago simple de una dirección a otra, que a menudo incluye algún "cambio" devuelto al propietario original. Este tipo de transacción tiene una entrada y dos salidas y se muestra en Transacción más común.

Transacción más común

image::images/mbc2_0205.png["Common Transaction"

Otra forma común de transacción es una que agrega muchas entradas en una sola salida (ver Transacciones de agregación de fondos). Esto representa el equivalente en el mundo real a intercambiar un montón de monedas y billetes en un único billete más grande. Las transacciones como esas a veces se generan por las aplicaciones de monedero para limpiar muchas cantidades pequeñas recibidas como cambio por pagos.

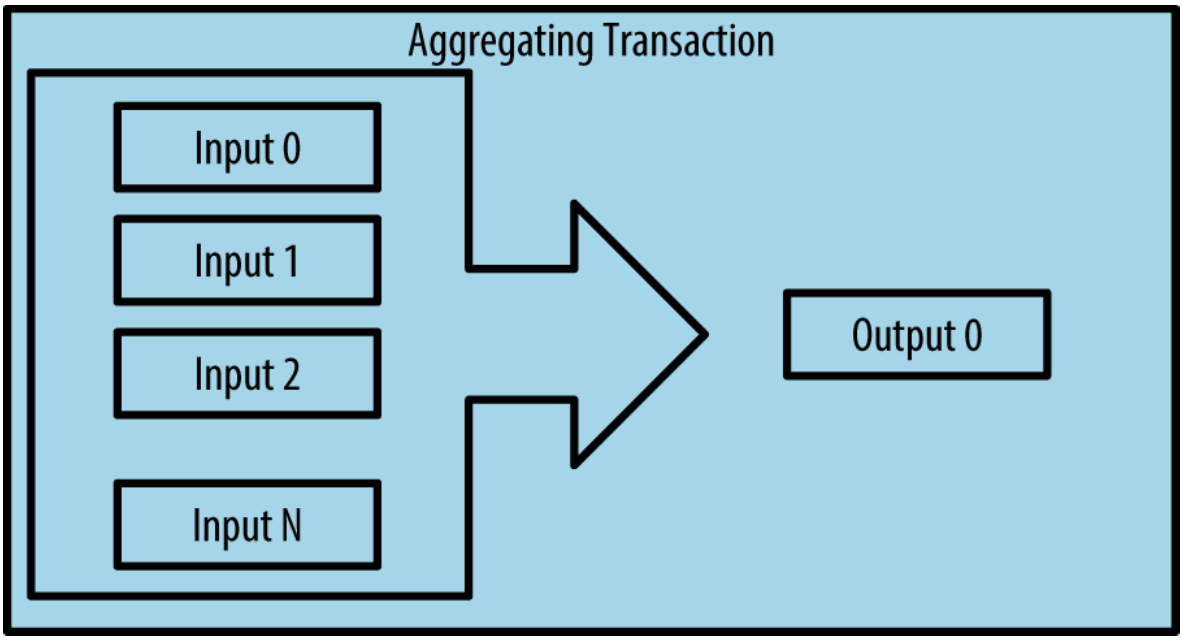


Figure 7. Transacciones de agregación de fondos

Finalmente, otra forma de transacción que se ve a menudo en el libro de contabilidad de bitcoin es una transacción que distribuye una entrada a múltiples salidas que representan múltiples destinatarios (ver Distribución de fondos de transacción). Este tipo de transacción a veces es utilizada por entidades comerciales para distribuir fondos, como cuando se procesan pagos de nómina a varios empleados.

Distribución de fondos de transacción

image::images/mbc2_0207.png["Distributing Transaction"

Construyendo una Transacción

La aplicación de cartera de Alice contiene toda la lógica necesaria para seleccionar entradas y salidas apropiadas al construir una transacción de acuerdo con las especificaciones de Alice. Alice solo necesita especificar un destino y una cantidad, y el resto sucede en la aplicación de cartera sin que ella vea los detalles. Es importante destacar que una aplicación de cartera puede construir transacciones aun cuando esté completamente sin conexión. De la misma forma que se puede escribir un cheque en casa y luego enviarlo al banco en un sobre, la transacción no necesita ser construida y firmada mientras se está conectado a la red bitcoin.

Consiguiendo las Entradas Correctas

La aplicación de cartera de Alice primero tendrá que encontrar entradas con las que poder pagar la cantidad que quiere enviar a Bob. La mayoría de las carteras mantienen un registro de todas las salidas disponibles que pertenecen a las direcciones en la cartera. Por lo tanto, la cartera de Alice guardaría una copia de la salida de transacción que se creó con la transacción de Joe, que se creó a cambio de dinero en efectivo (ver Obteniendo Tu Primer Bitcoin). Una aplicación de cartera bitcoin que se ejecuta en un nodo completo en realidad contiene una copia de cada salida no gastada de cada transacción en la cadena de bloques. Esto permite a una cartera construir entradas de transacción, así como verificar

rápidamente que las transacciones entrantes usan entradas correctas. Sin embargo, debido a que un nodo completo ocupa una gran cantidad de espacio en disco, la mayoría de las carteras se ejecutan como clientes "ligeros" que rastrean únicamente las salidas no gastadas propias de ese usuario.

Si la aplicación de la cartera no mantiene una copia de las salidas de transacción no gastadas, puede consultar a la red bitcoin para que le proporcione esta información utilizando una variedad de APIs disponibles a través de diferentes proveedores o solicitándoselo a un nodo completo mediante una llamada de interfaz de programación de aplicaciones (API). Observa todas las salidas no gastadas de la dirección bitcoin de Alice. muestra una petición API, construida a partir de un comando HTTP GET a una URL específica. Esta URL devolverá todas las salidas de transacciones no gastadas para una dirección, proporcionando a cualquier aplicación la información que necesita para construir las entradas de transacción para ser gastadas. Usamos el cliente HTTP simple de línea de comandos *cURL* para obtener la respuesta.

Example 2. Observa todas las salidas no gastadas de la dirección bitcoin de Alice.

```
$ curl https://blockchain.info/unspent?active=1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK
```

```
{
        "unspent_outputs":[

                {
                        "tx_hash":"186f9f998a5...2836dd734d2804fe65fa35779",
                        "tx_index":104810202,
                        "tx_output_n": 0,
                        "script":"76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
                        "value": 10000000,
                        "value_hex": "00989680",
                        "confirmations":0
                }

        ]
}
```

La respuesta en Observa todas las salidas no gastadas de la dirección bitcoin de Alice. muestra una salida no gastada (una que aún no ha sido gastada) bajo la propiedad de la dirección de Alice 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK. La respuesta incluye la referencia a la transacción en la que esta salida no gastada está contenida (el pago de Joe) y su valor en satoshis, a 10 millones, equivalente a 0.10 bitcoin. Con esta información, la cartera de Alice puede construir una transacción para transferir esa cantidad a la dirección del nuevo propietario.

TIP Ver transacción de Joe a Alice.

Como se puede ver, la cartera de Alice contiene suficiente bitcoin en una sola salida no gastada para pagar la taza de café. Si este no hubiera sido el caso, la cartera de Alice podría tener que "hurgar" a través de un montón de salidas más pequeñas no gastadas, como eligiendo monedas del bolsillo hasta que se encuentren las suficientes para pagar el café. En ambos casos, puede ser necesario que devuelvan el cambio, como veremos en la siguiente sección, donde la cartera crea las salidas de la transacción (pagos).

Creando las Salidas

Una salida de una transacción se crea en utilizando lenguaje script de comandos que crea un desafío protector del valor y solo puede redimirse mediante la introducción de una solución a la secuencia script de comandos. En términos más simples, el resultado de la transacción de Alice contendrá un script que dice algo así como "Esta salida es pagadera a quien quiera que pueda presentar una firma de la llave correspondiente a la dirección de Bob". Debido a que solo Bob tiene la billetera con las llaves correspondientes a esa dirección, solo la billetera de Bob puede presentar dicha firma para redimir esta salida. Por lo tanto, Alice "blinda" el valor de la salida con una solicitud de una firma de Bob.

Esta transacción también incluirá una segunda salida, porque los fondos de Alice están en la forma de una salida de 0.10 BTC, demasiado dinero para la taza de café de 0.015 BTC. Alice necesitará 0.085 BTC como cambio. El pago del cambio de Alice se crea por la cartera de Alice como una salida en la misma transacción que el pago a Bob. En definitiva, la cartera de Alice divide sus fondos en dos pagos: uno para Bob y otro de vuelta para ella misma. Después podrá usar (gastar) la salida del cambio en una transacción posterior.

Finalmente, para que la transacción sea procesada por la red de manera oportuna, la aplicación de cartera de Alice

añadirá una pequeña comisión. Esta no está explícita en la transacción; se obtiene de la diferencia entre entradas y salidas. Si en vez de llevarse 0.085 de cambio, Alice crea solo 0.0845 como segunda salida, habrá perdido 0.0005 BTC (medio milibitcoín). La entrada de 0.10 BTC no se gasta completamente con las dos salidas, ya que sumarán menos de 0.10. La diferencia resultante es la comisión de transacción que cobra el minero como tasa por validar e incluir la transacción en un bloque y registrarlo en la cadena de bloques.

La transacción resultante puede verse usando un explorador web de la cadena de bloques, tal como se muestra en Transacción de Alice a la Cafetería de Bob.

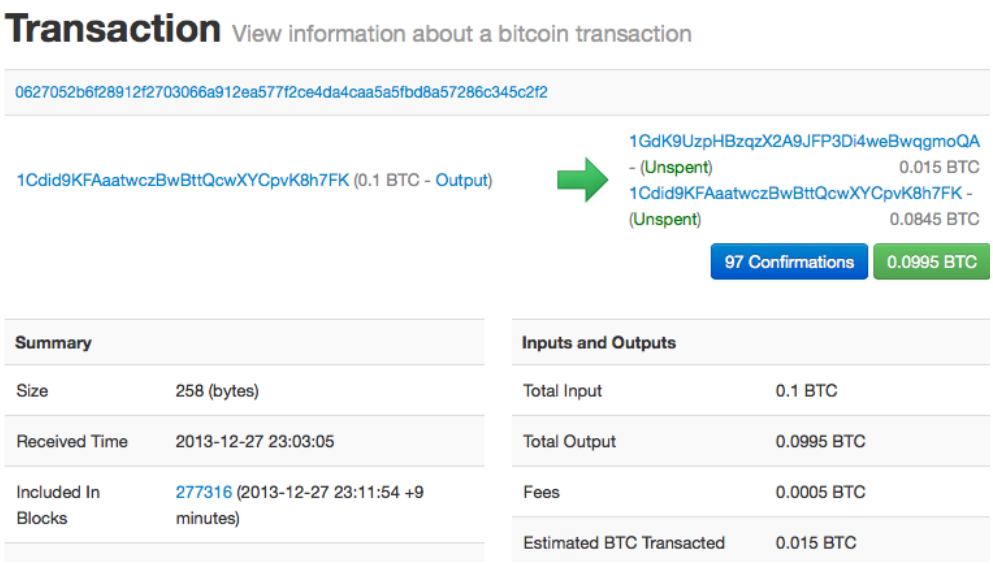


Figure 8. Transacción de Alice a la Cafetería de Bob

TIP | Vea la transacción de Alice a la cafetería de Bob.

Añadiendo la Transacción al Libro de Contabilidad

La transacción creada por la aplicación de cartera de Alice tiene una longitud de 258 bytes y contiene todo lo necesario para confirmar la propiedad de los fondos y asignar nuevos propietarios. Ahora, la transacción debe ser transmitida a la red bitcoin donde se convertirá en parte de la cadena de bloques. En la siguiente sección veremos cómo una transacción se convierte en parte de un nuevo bloque y cómo se "mina" el bloque. Finalmente, veremos cómo el nuevo bloque, una vez añadido a la cadena de bloques, es cada vez más confiable por la red cuantos más bloques se añadan.

Transmitiendo la Transacción

Debido a que la transacción contiene toda la información necesaria para ser procesada, no importa cómo o desde dónde se transmita a la red bitcoin. La red bitcoin es una red de igual a igual (P2P), en la que cada cliente bitcoin participa conectándose a otros muchos clientes bitcoin. El propósito de la red bitcoin es propagar transacciones y bloques a todos los participantes.

Cómo se propaga

Cualquier sistema, como un servidor, una aplicación de escritorio, o una cartera, que participa en la red bitcoin "hablando", el protocolo bitcoin se denomina *nodo bitcoin*. La aplicación de cartera de Alice puede enviar la nueva transacción a cualquier nodo bitcoin al que esté conectada a través de cualquier tipo de conexión: por cable, WiFi, móvil, etc. Su cartera bitcoin no tiene por qué estar conectada a la cartera bitcoin de Bob directamente y no tiene por qué usar la conexión a internet que ofrece la cafetería, aunque ambas opciones también son posibles. Cualquier nodo bitcoin que reciba una transacción válida que no haya visto anteriormente, la reenviará inmediatamente a todos los demás nodos a los que está conectado, en una técnica de propagación conocida como *inundación*. Por lo tanto, la transacción se propaga rápidamente a través de la red de igual a igual, alcanzando un gran porcentaje de los nodos en unos pocos segundos.

El Punto de Vista de Bob

Si la aplicación de billetera de bitcoin de Bob está directamente conectada a la aplicación de la billetera de Alice, la aplicación de billetera de Bob podría ser el primer nodo en recibir la transacción. Sin embargo, incluso si la billetera de Alice envía la transacción a través de otros nodos, llegará a la billetera de Bob en unos pocos segundos. La billetera de Bob identificará inmediatamente la transacción de Alice como un pago entrante porque contiene salidas canjeables por las llaves de Bob. La aplicación de la billetera de Bob también puede verificar de forma independiente que la transacción está

bien formada, y que utiliza salidas aún sin gastar y que contiene comisiones de transacción suficientes para ser incluidas en el siguiente bloque. En este punto, Bob puede asumir, con poco riesgo, que la transacción se incluirá en breve en un bloque y que se confirmará.

TIP

Una concepción errónea común sobre las transacciones de bitcoin es que deben ser "confirmadas" esperando 10 minutos a un nuevo bloque, o hasta 60 minutos para un total de seis confirmaciones. Aunque las confirmaciones aseguran que la transacción ha sido aceptada por toda la red, este retraso es innecesario para artículos de poco valor como una taza de café. Un vendedor puede aceptar una transacción válida de pequeño valor sin confirmaciones, sin más riesgo que un pago con tarjeta de crédito sin una identificación o firma, tal como ya lo hacen normalmente.

Minería de Bitcoin

La transacción de Alice ya se ha propagado en la red bitcoin. No se convierte en parte de la *cadena de bloques* hasta que se verifique y se incluya en un bloque mediante un proceso llamado *minería*. Ver Minería y Consenso para una explicación detallada.

El sistema de confianza de bitcoin se basa en la computación. Las transacciones son empaquetadas en *bloques*, que requieren una enorme capacidad de computación para ser válidos, pero solo una pequeña cantidad de computación para ser validados. El proceso de minería sirve dos propósitos en bitcoin:

- Los nodos de minería validan todas las transacciones mediante referencia a las *reglas de consenso* de bitcoin. Por lo tanto, la minería proporciona seguridad para las transacciones de bitcoin al rechazar transacciones inválidas o con formato incorrecto.
- La minería crea nuevo bitcoin en cada bloque, casi como un banco central que imprime dinero nuevo. La cantidad de bitcoin creada por bloque es limitada y disminuye con el tiempo, siguiendo un programa de emisión fijo.

La minería logra un buen equilibrio entre coste y recompensa. La minería usa electricidad para resolver un problema matemático. Un minero exitoso cobrará una *recompensa* en la forma de nuevo bitcoin y de comisiones de transacción. Sin embargo, la recompensa solo se cobrará si el minero ha validado correctamente todas las transacciones, a satisfacción de las reglas de *consenso*. Este delicado equilibrio proporciona seguridad para bitcoin sin una autoridad central.

Una buena manera de describir la minería es como un juego competitivo de sudoku que se reinicia cada vez que alguien encuentra la solución y cuya dificultad automáticamente se ajusta para que lleve aproximadamente 10 minutos encontrar una solución. Imagina un sudoku gigante, de muchos miles de filas y columnas. Si te lo muestro completado puedes verificarlo rápidamente. Sin embargo, si el puzzle tiene unas pocas casillas completadas y el resto está vacío, ¡lleva mucho trabajo resolverlo! La dificultad del sudoku puede ajustarse cambiando su tamaño (más o menos filas y columnas), pero puede seguir siendo verificado fácilmente aunque sea enorme. El puzzle usado en bitcoin está basado en hashes criptográficos y tienen similares características: es asimétricamente dificil de resolver pero fácil de verificar, y su dificultad se puede ajustar.

En Usos de Bitcoin, Usuarios y Sus Historias, presentamos a Jing, un emprendedor en Shanghai. Jing gestiona una *granja de minería*, que es una empresa que controla miles de plataformas de minería especializadas, compitiendo por la recompensa. Cada 10 minutos aproximadamente, las plataformas de minería de Jing se enfrentan contra miles de sistemas similares en una competición global para encontrar una solución a un bloque de transacciones. Encontrar esa solución, la denominada *Prueba-de-Trabajo* (del inglés, PoW, Proof of Work), requiere ejecutar trillones de hashes por segundo en toda la red bitcoin. El algoritmo de Prueba-de-Trabajo se basa en hacer hash repetidamente de las cabeceras de bloque y un número aleatorio con el algoritmo criptográfico SHA256 hasta que una solución encaje con un patrón predeterminado. El primer minero que encuentra esa solución gana la ronda de competición y publica ese bloque en la cadena de bloques.

Jing comenzó a minar en 2010 usando una computadora de escritorio muy rápida para encontrar una Prueba-de-Trabajo adecuada para los nuevos bloques. A medida que más mineros comenzaron a unirse a la red bitcoin, la dificultad del problema aumentó rápidamente. Pronto, Jing y otros mineros actualizaron su dotación a un hardware más especializado, con unidades de procesamiento gráfico (GPU) dedicadas de alta gama, a menudo utilizadas en computadoras de escritorio o consolas de juegos. En el momento de escribirse este libro, la dificultad es tan alta que solo es rentable la minería con circuitos integrados de aplicación específica (ASIC), esencialmente cientos de algoritmos de minería impresos en hardware, ejecutándose en paralelo en un solo chip de silicio. La compañía de Jing también participa en una *agrupación de minería*, que de forma muy similar a un grupo que juega a la lotería, permite a varios participantes compartir sus

esfuerzos y recompensas. La compañía de Jing ahora maneja un almacén que contiene miles de mineros ASIC para minar bitcoins las 24 horas del día. La compañía paga sus costos de electricidad al vender el bitcoin que puede generar de la minería, creando algunos ingresos de las ganancias.

Minando Transacciones en Bloques

Las nuevas transacciones fluyen constantemente hacia la red desde las carteras de los usuarios y otras aplicaciones. Cuando son vistas por los nodos de la red bitcoin, se añaden a un conjunto temporal de transacciones no verificadas que es mantenido por cada nodo. A medida que los mineros están construyendo un nuevo bloque, añaden transacciones no verificadas de este conjunto al nuevo bloque y luego intentan probar la validez de ese nuevo bloque, con el algoritmo de minería (Prueba-de-Trabajo). El proceso de minería se explica en detalle en Minería y Consenso.

Las transacciones se agregan al nuevo bloque, priorizadas por las transacciones de comisiones más altas primero y algunos otros criterios. Cada minero comienza el proceso de minería de un nuevo bloque de transacciones tan pronto como recibe el bloque anterior de la red, sabiendo que ha perdido esa ronda previa de competencia. Inmediatamente crea un nuevo bloque, lo llena con transacciones y la huella digital del bloque anterior, y comienza a calcular la Prueba-de-Trabajo para el nuevo bloque. Cada minero incluye una transacción especial en su bloque, una que paga a su propia dirección bitcoin, la recompensa del bloque (actualmente 6.25 bitcoins recién creados) más la suma de las comisiones de transacción de todas las transacciones incluidas en el bloque. Si se encuentra una solución que haga que ese bloque sea válido, se "gana" entonces esta recompensa porque su bloque exitoso se agrega a la cadena de bloques global y la transacción de recompensa que fue incluida se podrá gastar. Jing, que participa en un grupo de minería, ha configurado su software para crear nuevos bloques que asignen la recompensa a una dirección del grupo. A partir de ahí, una parte de la recompensa se distribuye a Jing y otros mineros en proporción a la cantidad de trabajo con que contribuyeron en la última ronda.

La transacción de Alice fue recogida por la red e incluida en el conjunto de transacciones no verificadas. Una vez validada por el software de minería, se incluyó en un nuevo bloque, llamado *bloque candidato*, generado por el pool de minería de Jing. Todos los mineros que participan en ese pool de minería comienzan inmediatamente a calcular la Prueba-de-Trabajo para el bloque candidato. Aproximadamente cinco minutos después de que la cartera de Alice transmitiera la transacción por primera vez, uno de los mineros ASIC de Jing encontró una solución para el bloque candidato y la anunció a la red. Una vez que otros mineros validaron el bloque ganador, comenzaron la competición para generar el siguiente bloque.

El bloque ganador de Jing se convirtió en parte de la cadena de bloques como bloque #277316, que contiene 419 transacciones, incluida la transacción de Alice. El bloque que contiene la transacción de Alice se cuenta como una "confirmación" de esa transacción.

> **TIP** Aquí puede verse el bloque que incluye la transacción de Alice.

Aproximadamente 19 minutos más tarde, otro minero minó un nuevo bloque, #277317. Debido a que este nuevo bloque está construido sobre el bloque #277316 que contenía la transacción de Alice, añadió aún más computación a la cadena de bloques, fortaleciendo la confianza en esas transacciones. Cada bloque minado sobre el que contiene la transacción se considera como una confirmación adicional de la transacción de Alice. A medida que los bloques se acumulan unos sobre otros, se vuelve exponencialmente más difícil revertir la transacción, lo que hace sea cada vez más confiable por la red.

transacciónEn el diagrama en Transacción de Alice incluida en el bloque #277316, podemos ver el bloque #277316, que contiene la transacción de Alice. Debajo de ella hay 277.316 bloques (incluido el bloque #0), vinculados entre sí en una cadena de bloques (blockchain) todo el camino atrás hasta el bloque #0, conocido como *bloque génesis*. Con el tiempo, a medida que aumenta la "altura" de los bloques, también lo hace la dificultad de cómputo para cada bloque y la cadena en su conjunto. Los bloques minados después del que contiene la transacción de Alice actúan como una garantía adicional, a medida que acumulan más computación en una cadena más y más larga. Por convención, cualquier bloque con más de seis confirmaciones se considera irrevocable, ya que se requeriría una inmensa cantidad de cálculo computacional para invalidar y recalcular seis bloques. Examinaremos el proceso de minería y la manera en que genera confianza con más detalle en Minería y Consenso.

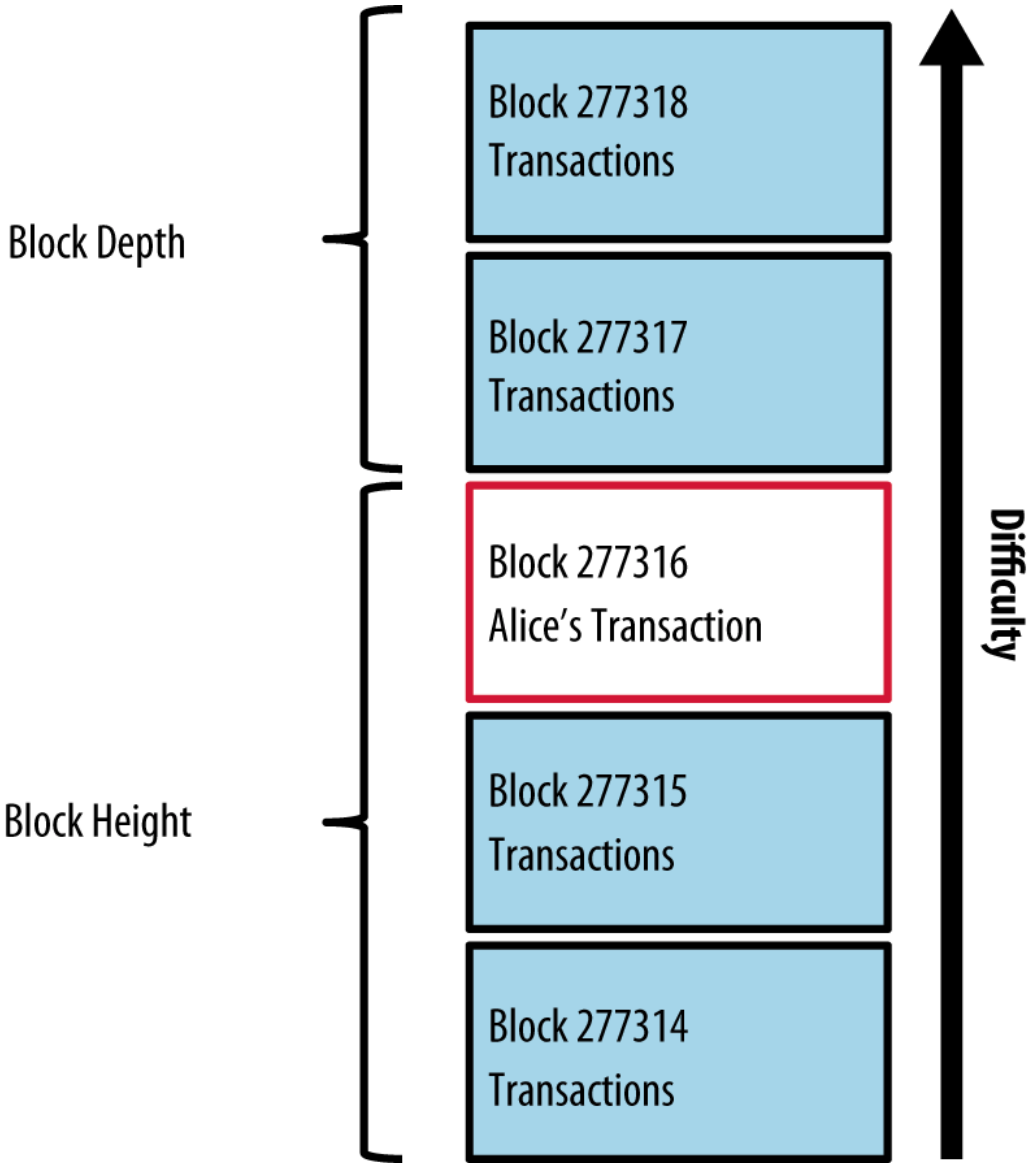


Figure 9. Transacción de Alice incluida en el bloque #277316

Gastando la Transacción

Ahora que la transacción de Alice ha sido incluida en la cadena de bloques como parte de un bloque, forma parte del libro de contabilidad distribuido de bitcoin y es visible para todas las aplicaciones de bitcoin. Cada cliente bitcoin puede verificar de forma independiente la transacción como válida y utilizable. Los clientes de nodo completo pueden rastrear el origen de los fondos desde el momento en que se generaron los bitcoin por primera vez en un bloque, progresivamente de transacción a transacción, hasta llegar a la dirección de Bob. Los clientes ligeros pueden hacer lo que se llama una verificación de pago simplificada (ver Nodos de Verificación de Pago Simplificada (SPV)) confirmando que la transacción está en la cadena de bloques y que tiene unos cuantos bloques minados después de ella, y por tanto asegurando que los mineros la aceptaron como válida.

Bob puede ahora gastar la salida de esta y de otras transacciones. Por ejemplo, Bob puede pagar a un contratista o proveedor transfiriendo valor del pago de la taza de café de Alice a estos nuevos propietarios. Seguramente, el software bitcoin de Bob agregará muchos pequeños pagos en un pago mayor, tal vez concentrando todos los ingresos de bitcoin del día en una sola transacción. Esto agregaría los diversos pagos en una sola salida (y una sola dirección). Para un diagrama de una transacción de agregación, consulte Transacciones de agregación de fondos.

A medida que Bob gasta los pagos recibidos de Alice y otros clientes, extiende la cadena de transacciones. Supongamos que Bob le paga a su diseñador web Gopesh en Bangalore por una nueva página web. Ahora la cadena de transacciones se lucirá como La transacción de Alice como parte de una cadena de transacciones desde Joe a Gopesh.

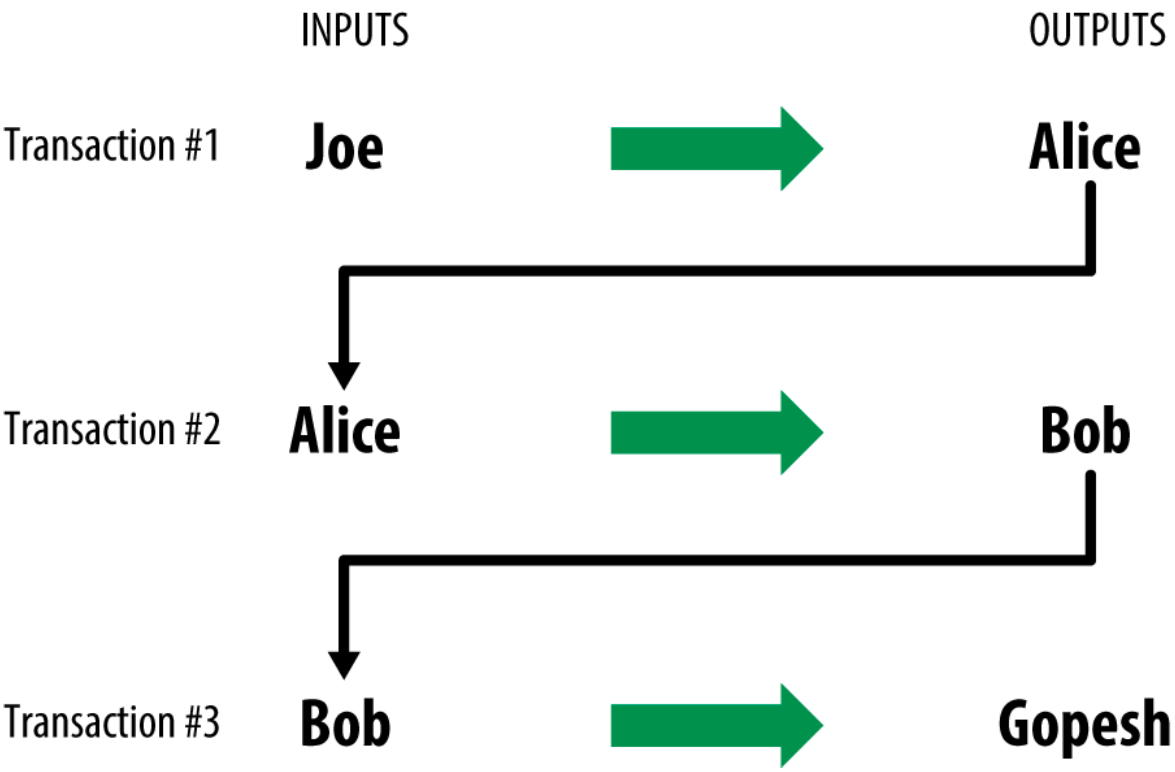


Figure 10. La transacción de Alice como parte de una cadena de transacciones desde Joe a Gopesh

En este capítulo, vimos cómo las transacciones crean una cadena que mueve valor de un propietario a otro. También rastreamos la transacción de Alice, desde el momento en que se creó en su cartera, a través de la red bitcoin y hasta los mineros que la registraron en la cadena de bloques. En el resto de este libro, examinaremos las tecnologías específicas detrás de carteras, direcciones, firmas, transacciones, la red y, finalmente, la minería.

Bitcoin Core: La Implementación de Referencia

Bitcoin es un proyecto de *software libre* y el código fuente está disponible bajo una licencia libre (MIT), se puede descargar gratuitamente y usar para cualquier propósito. Software libre significa algo más que simplemente gratuito. También significa que bitcoin está desarrollado por una comunidad abierta de voluntarios. Al principio, esa comunidad estaba formada únicamente por Satoshi Nakamoto. Para el año 2016, el código fuente de bitcoin tenía más de 400 colaboradores con una docena de desarrolladores trabajando en el código casi a tiempo completo y varias docenas más a tiempo parcial. Cualquiera puede contribuir en su programación— ¡tú también!

Cuando Satoshi Nakamoto creó bitcoin, el software se completó antes de que se escribiera el libro blanco que se reproduce en El Whitepaper (Libro Blanco) de Bitcoin por Satoshi Nakamoto. Satoshi quería asegurarse de que funcionara antes de escribir sobre ello. Esa primera implementación, luego conocida simplemente como "Bitcoin" o "cliente Satoshi", ha sido modificada y mejorada en profundidad. Se ha convertido en lo que se conoce como *Bitcoin Core*, para diferenciarlo de otras implementaciones compatibles. Bitcoin Core es la *implementación de referencia* del sistema bitcoin, lo que significa que es la referencia autorizada sobre cómo se debe implementar cada componente de la tecnología. Bitcoin Core implementa todos los aspectos de bitcoin, incluidas las carteras, un motor de validación de transacciones y bloques, y un nodo de red completo en la red de pares de bitcoin.

WARNING

A pesar de que Bitcoin Core incluye una implementación de referencia de una cartera, esta no está diseñada para ser utilizada como cartera de producción para usuarios o aplicaciones. Se recomienda a los desarrolladores de aplicaciones que programen carteras utilizando estándares modernos como BIP-39 y BIP-32 (ver Palabras Código Mnemónicas (BIP-39) y Carteras HD (BIP-32/BIP-44)). BIP significa *Propuesta de Mejora de Bitcoin* (en inglés, Bitcoin Improvement Proposal).

Arquitectura de Bitcoin Core (Fuente: Eric Lombrozo) muestra la arquitectura de Bitcoin Core.

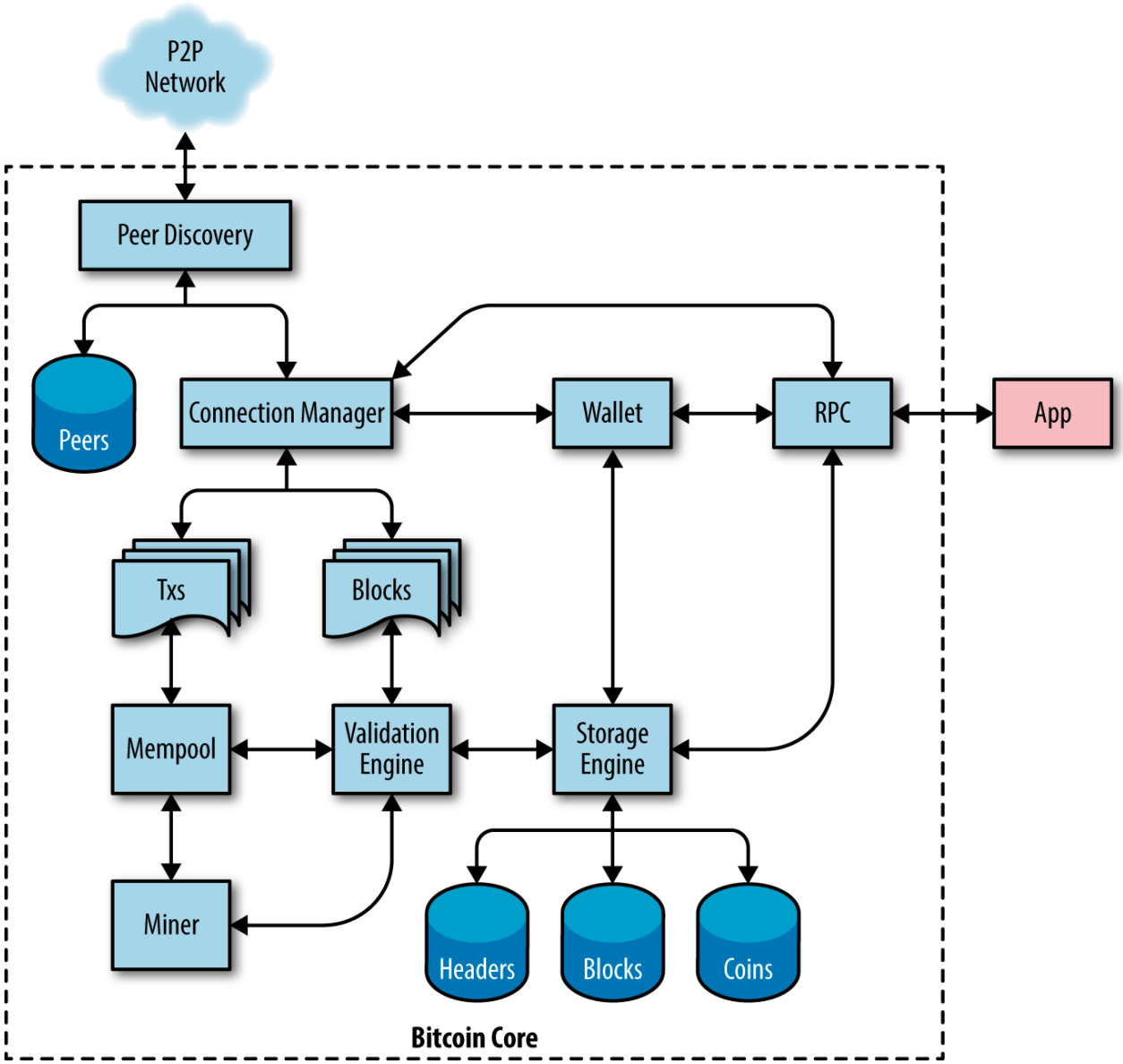


Figure 11. Arquitectura de Bitcoin Core (Fuente: Eric Lombrozo)

Entorno de Desarrollo de Bitcoin

Si eres desarrollador, desearás configurar un entorno de desarrollo con todas las herramientas, bibliotecas y software de soporte para programar aplicaciones de bitcoin. En este capítulo altamente técnico, veremos ese proceso paso a paso. Si el material se vuelve demasiado denso (y no estás configurando un entorno de desarrollo), no dudes en pasar al siguiente capítulo, que es menos técnico.

Compilando Bitcoin Core desde Código Fuente

El código fuente de Bitcoin Core se puede descargar como un archivo comprimido o clonando el repositorio fuente desde GitHub. En la Bitcoin página de descarga de Bitcoin Core, selecciona la versión más reciente y descarga el archivo comprimido del código fuente, por ejemplo, bitcoin-0.15.0.2.tar.gz. Alternativamente, usa la línea de comandos de git para crear una copia local del código fuente desde la GitHub página de bitcoin.

TIP

En muchos de los ejemplos de este capítulo, utilizaremos la interfaz de línea de comandos del sistema operativo (también conocida como "shell"), a la que se accede mediante una aplicación de "terminal". El shell mostrará un indicador; escribes un comando; y el shell responde con algún texto y un nuevo indicador para tu próximo comando. El indicador puede mostrarse de manera diferente en tu sistema, pero en los siguientes ejemplos se denota con un símbolo $. En los ejemplos, cuando veas texto después de un símbolo $, no escribas el símbolo $, escribe el comando inmediatamente después, luego presiona Intro para ejecutar el comando. En los ejemplos, las líneas debajo de cada comando son las respuestas del sistema operativo a ese comando. Cuando veas el siguiente prefijo $, sabrás que es un comando nuevo y que debes repetir el proceso.

En este ejemplo, estamos usando el comando git para crear una copia local ("clonar") del código fuente:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 102071, done.
remote: Compressing objects: 100% (10/10), done.
Receiving objects: 100% (102071/102071), 86.38 MiB | 730.00 KiB/s, done.
remote: Total 102071 (delta 4), reused 5 (delta 1), pack-reused 102060
Resolving deltas: 100% (76168/76168), done.
Checking connectivity... done.
$
```

> **TIP** Git es el sistema distribuido de control de versiones más utilizado, una parte esencial del kit de herramientas de cualquier desarrollador de software. Es posible que debas instalar el comando git, o una interfaz gráfica de usuario para git, en tu sistema operativo si aún no lo tienes.

Cuando la operación de clonación de git haya finalizado, tendrás una copia local completa del repositorio de código fuente en el directorio *bitcoin*. Cambia a este directorio tecleando ** cd bitcoin** tras el indicador:

```
$ cd bitcoin
```

Seleccionando una Versión de Bitcoin Core

De forma predeterminada, la copia local se sincronizará con el código más reciente, que podría ser una versión inestable o beta de bitcoin. Antes de compilar el código, selecciona una entrega específica descargando (en inglés, checkout) a partir de su *etiqueta* de versión. Esto sincronizará la copia local con una copia instantánea específica del repositorio identificada por esa etiqueta. Los desarrolladores utilizan las etiquetas para marcar entregas específicas del código por número de versión. Primero, para encontrar las etiquetas disponibles, usamos el comando git tag:

```
$ git tag
v0.1.5
v0.1.6test1
v0.10.0
...
v0.11.2
v0.11.2rc1
v0.12.0rc1
v0.12.0rc2
...
```

La lista de etiquetas muestra todas las versiones publicadas de bitcoin. Por convención, las *entregas candidatas* (en inglés, release candidates), que están destinados a pruebas, tienen el sufijo "rc". Las entregas estables que se pueden ejecutar en sistemas de producción no tienen sufijo. De la lista anterior, selecciona la entrega con la versión más alta, que en el momento de esta impresión era v0.15.0. Para sincronizar el código local con esta versión, usa el comando git checkout:

```
$ git checkout v0.15.0
HEAD is now at 3751912... Merge #11295: doc: Old fee_estimates.dat are discarded by 0.15.0
```

Puede confirmar que tienes la versión deseada "desprotegida" (en inglés, checked out) lanzando el comando git status:

```
$ git status
HEAD detached at v0.15.0
nothing to commit, working directory clean
```

Configurando la Construcción de Bitcoin Core

El código fuente incluye documentación, que se puede encontrar en varios archivos. Revisa la documentación principal ubicada en *README.md* en el directorio *bitcoin* tecleando **more README.md** en el indicador y usa la barra espaciadora para avanzar a las páginas siguientes. En este capítulo, construiremos el cliente bitcoin de línea de comandos, también conocido como bitcoind en Linux. Revisa las instrucciones para compilar el cliente de línea de comandos bitcoind en tu plataforma tecleando **more doc/build-unix.md**. Existen instrucciones alternativas para macOS y Windows en el directorio *doc*, como *build-osx.md* o *build-windows.md*, respectivamente.

Revisa cuidadosamente los prerrequisitos para hacer la construcción, que están en la primera parte de la documentación.

Esas son las bibliotecas que deben estar presentes en tu sistema antes de que puedas comenzar a compilar bitcoin. Si faltaran estos requisitos previos, la construcción dará un error. Si esto sucede porque has omitido algún requisito previo, puedes instalarlo y luego reanudar el proceso de construcción desde donde lo dejaste. Suponiendo que se instalaron los requisitos previos, puedes iniciar el proceso de construcción mediante la generación de un conjunto de scripts de construcción usando el script *autogen.sh*.

```
$ ./autogen.sh
...
glibtoolize: copying file 'build-aux/m4/libtool.m4'
glibtoolize: copying file 'build-aux/m4/ltoptions.m4'
glibtoolize: copying file 'build-aux/m4/ltsugar.m4'
glibtoolize: copying file 'build-aux/m4/ltversion.m4'
...
configure.ac:10: installing 'build-aux/compile'
configure.ac:5: installing 'build-aux/config.guess'
configure.ac:5: installing 'build-aux/config.sub'
configure.ac:9: installing 'build-aux/install-sh'
configure.ac:9: installing 'build-aux/missing'
Makefile.am: installing 'build-aux/depcomp'
...
```

El script *autogen.sh* crea un conjunto de scripts de configuración automática que recogen información de su sistema para descubrir los ajustes correctos y asegurarse de que tiene todas las bibliotecas necesarias para compilar el código. El más importante de ellos es el script configure que ofrece diferentes opciones para personalizar el proceso de construcción. Teclea **. /configure --help ** para ver las distintas opciones:

```
$ ./configure --help
`configure' configura Bitcoin Core 0.15.0 para adaptarlo a muchos tipos de sistemas.

Uso: ./configure [OPTION]... [VAR=VALUE]...

...
Funcionalidades opcionales:
  --disable-option-checking ignora lo no reconocido --enable/--with opciones
  --disable-CARACTERISTICA no incluye CARACTERISTICA (es lo mismo que --enable-FEATURE=no)
  --enable-CARACTERISTICA[=ARGUMENTO] incluye CARACTERISTICA [ARGUMENTO=yes]

  --enable-wallet         activar cartera (predeterminado es sí)

  --with-gui[=no|qt4|qt5|auto]
...
```

El script configure te permite activar o desactivar ciertas características de bitcoind mediante el uso de las opciones –enable-CARACTERISTICA y --disable-CARACTERISTICA, donde `CARACTERISTICA` se reemplaza por el nombre de la característica, escrita como se muestra en la ayuda. En este capítulo, vamos a construir el cliente bitcoind con todas las características predeterminadas. No utilizaremos las etiquetas de configuración, pero deberías revisarlos para comprender qué características opcionales forman parte del cliente. Si te encuentras en un entorno académico, es posible que las restricciones del laboratorio de computación requieran que instales las aplicaciones en tu directorio de inicio (por ejemplo, usando --prefix=$HOME).

Aquí hay algunas opciones útiles que se anteponen el comportamiento predeterminado del script configure:

`--prefix=$HOME`

Esto cambia la ubicación de instalación predeterminada (que es */usr/local/*) para el ejecutable resultante. Usa `$HOME` para poner todo en tu directorio personal o en otro diferente.

`--disable-wallet`

Esto se usa para deshabilitar la implementación de la cartera de referencia.

`--with-incompatible-bdb`

Si estás construyendo una cartera, permite el uso de una versión incompatible de la librería Berkeley DB.

`--with-gui=no`

No crea la interfaz gráfica de usuario, que requiere la biblioteca Qt. Esto solo construye el servidor y la línea de comandos de bitcoin.

A continuación, ejecuta el script configure para descubrir automáticamente todas las bibliotecas necesarias y crear un script de construcción personalizado para tu sistema:

```
$ ./configure
checking build system type... x86_64-unknown-linux-gnu
checking host system type... x86_64-unknown-linux-gnu
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
checking for a thread-safe mkdir -p... /bin/mkdir -p
checking for gawk... gawk
checking whether make sets $(MAKE)... yes
...
[siguen varias páginas de tests de configuración]
...
$
```

Si todo salió bien, el comando configure finalizará creando los scripts de construcción personalizados que nos permitirán compilar bitcoind. Si faltan bibliotecas o hay errores, el comando configure terminará con error en vez de crear los scripts de construcción. Si ocurre un error, es muy probable que se deba a una biblioteca faltante o incompatible. Revisa nuevamente la documentación de construcción y asegúrate de instalar los requisitos previos que faltan. Después ejecuta configure otra vez y comprueba si eso corrige el error.

Construyendo los Ejecutables de Bitcoin Core

A continuación, compilarás el código fuente, un proceso que puede tardar hasta una hora en completarse, dependiendo de la velocidad de tu CPU y de la memoria disponible. Durante el proceso de compilación deberías ver resultados cada pocos segundos o cada pocos minutos, o un error si algo sale mal. Si se produce un error, o se interrumpe el proceso de compilación, se puede reanudar en cualquier momento tecleando make nuevamente. Teclea **make** para comenzar a compilar la aplicación:

```
$ make
Making all in src
  CXX      crypto/libbitcoinconsensus_la-hmac_sha512.lo
  CXX      crypto/libbitcoinconsensus_la-ripemd160.lo
  CXX      crypto/libbitcoinconsensus_la-sha1.lo
  CXX      crypto/libbitcoinconsensus_la-sha256.lo
  CXX      crypto/libbitcoinconsensus_la-sha512.lo
  CXX      libbitcoinconsensus_la-hash.lo
  CXX      primitives/libbitcoinconsensus_la-transaction.lo
  CXX      libbitcoinconsensus_la-pubkey.lo
  CXX      script/libbitcoinconsensus_la-bitcoinconsensus.lo
  CXX      script/libbitcoinconsensus_la-interpreter.lo

[... siguen muchos otros mensajes de compilación ...]

$
```

En un sistema rápido con más de una CPU, es posible que desees establecer el número de trabajos de compilación en paralelo. Por ejemplo, make -j 2 utilizará dos núcleos si están disponibles. Si todo va bien, ahora se ha compilado Bitcoin Core. Deberías ejecutar el conjunto de tests unitarios con make check para asegurarte de que las bibliotecas vinculadas no están rotas de manera obvia. El último paso es instalar los distintos ejecutables en tu sistema usando el comando make install. Es posible que se te solicite tu contraseña de usuario, porque este paso requiere privilegios administrativos:

```
$ make check && sudo make install
Password:
Making install in src
 ../build-aux/install-sh -c -d '/usr/local/lib'
libtool: install: /usr/bin/install -c bitcoind /usr/local/bin/bitcoind
libtool: install: /usr/bin/install -c bitcoin-cli /usr/local/bin/bitcoin-cli
libtool: install: /usr/bin/install -c bitcoin-tx /usr/local/bin/bitcoin-tx
...
$
```

La instalación predeterminada de bitcoind se coloca en */usr/local/bin*. Puedes confirmar que Bitcoin Core está instalado correctamente solicitando al sistema la ruta de los ejecutables, de la siguiente manera:

```
$ which bitcoind
/usr/local/bin/bitcoind
```

```
$ which bitcoin-cli
/usr/local/bin/bitcoin-cli
```

Ejecutando un Nodo de Bitcoin Core

La red P2P de Bitcoin está compuesta por "nodos" de red, que se ejecutan principalmente por voluntarios y algunas de las empresas que crean aplicaciones de bitcoin. Aquellos que ejecutan nodos bitcoin tienen una visión directa y autorizada de la cadena de bloques de bitcoin, con una copia local de todas las transacciones, validadas independientemente por su propio sistema. Al ejecutar un nodo, no dependes de ningún tercero para validar una transacción. Además, al ejecutar un nodo bitcoin, contribuyes a la red bitcoin haciéndola más robusta.

Sin embargo, la ejecución de un nodo requiere un sistema conectado permanentemente con recursos suficientes para procesar todas las transacciones de bitcoin. Si además de mantener una copia completa de la cadena de bloques, también eliges indexar todas las transacciones, puedes necesitar una gran cantidad de espacio en disco y RAM. A principios de 2018, un nodo completo indexado necesita 2 GB de RAM y un mínimo de 160 GB de espacio en disco (consulta https://blockchain.info/charts/blocks-size). Los nodos bitcoin también transmiten y reciben transacciones y bloques de bitcoin, que consumen ancho de banda de internet. Si tu conexión a internet es limitada, tienes un límite de datos bajo, o no tienes tarifa plana (pagando por gigabit), probablemente no deberías ejecutar un nodo bitcoin en ella, o si no, ejecutarlo limitando su uso de ancho de banda (consulte Configuración de muestra de un sistema con recursos limitados).

TIP

Bitcoin Core mantiene una copia completa de la cadena de bloques de forma predeterminada, incluyendo todas las transacciones que han ocurrido en la red bitcoin desde su creación en 2009. Este conjunto de datos tiene un tamaño de decenas de gigabytes y se descarga de forma gradual durante varios días o semanas, dependiendo de la velocidad de tu CPU y de tu conexión a internet. Bitcoin Core no podrá procesar transacciones ni actualizar los saldos de cuenta hasta que se descargue el conjunto de datos completo de la cadena de bloques. Asegúrate de que tienes suficiente espacio en disco, ancho de banda y tiempo para completar la sincronización inicial. Puedes configurar Bitcoin Core para reducir el tamaño de la cadena de bloques descartando los bloques antiguos (ver Configuración de muestra de un sistema con recursos limitados), pero aun así se descargará el conjunto de datos completo antes de descartar datos.

A pesar de estos requisitos de recursos, miles de voluntarios ejecutan nodos bitcoin. Algunos se ejecutan en sistemas tan simples como una Raspberry Pi (una computadora de $35 USD del tamaño de un estuche de naipes). Muchos voluntarios también ejecutan nodos bitcoin en servidores alquilados, generalmente en alguna variante de Linux. Se puede usar una instancia de *Virtual Private Server* (VPS) o *Cloud Computing Server* para ejecutar un nodo bitcoin. Dichos servidores se pueden alquilar a distintos proveedores por entre $25 a $50 USD al mes.

¿Por qué querrías ejecutar un nodo? Estas son algunas de las razones más comunes:

- Si estás desarrollando software de bitcoin y necesitas confiar en un nodo bitcoin para el acceso programable (API) a la red y a la cadena de bloques.
- Si estás creando aplicaciones que deben validar transacciones según las reglas de consenso de bitcoin. Normalmente, las compañías de software de bitcoin ejecutan varios nodos.
- Si quieres apoyar a bitcoin. La ejecución de un nodo hace que la red sea más robusta y capaz de servir más carteras, más usuarios y más transacciones.
- Si no deseas confiar en un tercero para procesar o validar tus transacciones.

Si estás leyendo este libro y estás interesado en desarrollar software de bitcoin, deberías ejecutar tu propio nodo.

Configurando el Nodo de Bitcoin Core

Bitcoin Core buscará un archivo de configuración en su directorio de datos en cada inicio. En esta sección examinaremos las diversas opciones de configuración y crearemos un archivo de configuración. Para ubicar el archivo de configuración, ejecuta bitcoind -printtoconsole en tu terminal y busca en el primer par de líneas.

```
$ bitcoind -printtoconsole
Bitcoin version v0.15.0
Using the 'standard' SHA256 implementation
Using data directory /home/ubuntu/.bitcoin/
Using config file /home/ubuntu/.bitcoin/bitcoin.conf
...
```

```
[muchos más datos de depuración]
...
```

Podemos presionar Ctrl–C para cerrar el nodo, una vez que determinemos la ubicación del archivo de configuración. Por lo general, el archivo de configuración está dentro del directorio de datos *.bitcoin* debajo del directorio de inicio de nuestro usuario. Este no se crea automáticamente, pero puede crear un archivo de configuración de inicio copiando y pegando desde el ejemplo Configuración de muestra de un nodo de índice completo. que se muestra a continuación. Podemos crear o modificar el archivo de configuración en nuestro editor preferido.

Bitcoin Core ofrece más de 100 opciones de configuración que modifican el comportamiento del nodo de red, el almacenamiento de la cadena de bloques y muchos otros aspectos de su funcionamiento. Para ver una lista de estas opciones, ejecuta bitcoind --help:

```
$ bitcoind --help
Bitcoin Core Daemon version v0.15.0

Usage:
  bitcoind [options]                     Start Bitcoin Core Daemon

Options:

  -?
       Print this help message and exit

  -version
       Print version and exit

  -alertnotify=<cmd>
       Execute command when a relevant alert is received or we see a really
       long fork (%s in cmd is replaced by message)
...
[muchas más opciones]
...

  -rpcthreads=<n>
       Establecer el número de subprocesos para atender llamadas RPC (predeterminado: 4)
```

Estas son algunas de las opciones más importantes que puedes establecer en el archivo de configuración, o como parámetros de línea de comandos para bitcoind:

alertnotify

Ejecuta un comando o script específico para enviar alertas de emergencia al propietario de este nodo, generalmente por correo electrónico.

conf

Una ubicación alternativa para el archivo de configuración. Esto solo tiene sentido como parámetro de línea de comandos para bitcoind, ya que no puede estar dentro del archivo de configuración al que hace referencia.

datadir

Selecciona el directorio y el sistema de archivos en el que colocar todos los datos de la cadena de bloques. Por defecto, este es el subdirectorio *.bitcoin* de tu directorio de inicio. Asegúrate de que este sistema de archivos tenga varios gigabytes de espacio libre.

prune

Reduce los requisitos de espacio en disco a tantos megabytes, eliminando bloques antiguos. Usa esto en un nodo con recursos limitados que no dispone del espacio suficiente para guardar la cadena de bloques completa.

txindex

Mantiene un índice de todas las transacciones. Esto significa una copia completa de la cadena de bloques que te permite obtener mediante programación cualquier transacción por ID.

dbcache

El tamaño de la caché UTXO. El valor predeterminado es 300 MiB. Aumenta esto en el hardware de gama alta, y redúcelo en el hardware de gama baja para ahorrar memoria a costa de una E/S de disco lenta.

maxconnections

Establece el número máximo de nodos desde los que aceptar conexiones. Poner un valor más bajo del predeterminado reducirá el consumo de ancho de banda. Utilízalo si tienes un límite de datos o pagas por gigabyte.

maxmempool

Limita el tanque de memoria de transacciones a tantos megabytes. Úsalo para reducir el uso de memoria en nodos con limitaciones de memoria.

maxreceivebuffer/maxsendbuffer

Limita el búfer de memoria por conexión a este número de múltiplos de 1000 bytes. Úsalo en nodos con limitaciones de memoria.

minrelaytxfee

Establece la comisión mínima para retransmitir una transacción. Por debajo de este valor, la transacción se trata como no estándar, rechazándola del pool de transacciones y no se retransmite.

Índice de Base de Datos de Transaccion y Opción txindex

De forma predeterminada, Bitcoin Core construye una base de datos que contiene *solo* las transacciones relacionadas con la cartera del usuario. Si deseas poder acceder a cualquier transacción con comandos como getrawtransaction (ver Explorando y Decodificando Transacciones), debes configurar Bitcoin Core para crear un índice completo de transacciones, que se puede lograr con la opción txindex. Establece txindex=1 en el archivo de configuración de Bitcoin Core. Si no estableces esta opción al principio y la pones después para indexación completa, debes reiniciar bitcoind con la opción -reindex y esperar a que a se regenere el índice.

Configuración de muestra de un nodo de índice completo. muestra cómo podrías combinar las opciones anteriores, con un nodo completamente indexado, ejecutándose como un backend API para una aplicación de bitcoin.

Example 3. Configuración de muestra de un nodo de índice completo.

```
alertnotify=myemailscript.sh "Alert: %s"
datadir=/lotsofspace/bitcoin
txindex=1
```

Configuración de muestra de un sistema con recursos limitados muestra un nodo con recursos limitados que se ejecuta en un servidor más pequeño.

Example 4. Configuración de muestra de un sistema con recursos limitados

```
alertnotify=myemailscript.sh "Alert: %s"
maxconnections=15
prune=5000
dbcache=150
maxmempool=150
maxreceivebuffer=2500
maxsendbuffer=500
```

Una vez que hayas editado el archivo de configuración y hayas configurado las opciones que mejor cubran tus necesidades, puedes probar bitcoind con esta configuración. Ejecuta Bitcoin Core con la opción printtoconsole para ejecutar en primer plano con salida a la consola:

```
$ bitcoind -printtoconsole

Bitcoin version v0.15.0
InitParameterInteraction: parameter interaction: -whitelistforcerelay=1 -> setting -whitelistrelay=1
Assuming ancestors of block 000000000000000003b9ce759c2a087d52abc4266f8f4ebd6d768b89defa50a have valid signatures.
Using the 'standard' SHA256 implementation
Default data directory /home/ubuntu/.bitcoin
Using data directory /lotsofspace/.bitcoin
Using config file /home/ubuntu/.bitcoin/bitcoin.conf
Using at most 125 automatic connections (1048576 file descriptors available)
```

```
Using 16 MiB out of 32/2 requested for signature cache, able to store 524288 elements
Using 16 MiB out of 32/2 requested for script execution cache, able to store 524288 elements
Using 2 threads for script verification
HTTP: creating work queue of depth 16
No rpcpassword set - using random cookie authentication
Generated RPC authentication cookie /lotsofspace/.bitcoin/.cookie
HTTP: starting 4 worker threads
init message: Verifying wallet(s)...
Using BerkeleyDB version Berkeley DB 4.8.30: (April  9, 2010)
Using wallet wallet.dat
CDBEnv::Open: LogDir=/lotsofspace/.bitcoin/database ErrorFile=/lotsofspace/.bitcoin/db.log
scheduler thread start
Cache configuration:
* Using 250.0MiB for block index database
* Using 8.0MiB for chain state database
* Using 1742.0MiB for in-memory UTXO set (plus up to 286.1MiB of unused mempool space)
init message: Loading block index...
Opening LevelDB in /lotsofspace/.bitcoin/blocks/index
Opened LevelDB successfully

[... más mensajes de inicialización ...]
```

Puede presionar Ctrl-C para interrumpir el proceso una vez que estés satisfecho de que estás cargando las configuraciones correctas y funcionando de la manera que quieres.

Para ejecutar Bitcoin Core en segundo plano como un proceso, inícialo con la opción daemon, como bitcoind -daemon.

Para monitorizar el progreso y el estado de ejecución de tu nodo bitcoin, usa el comando bitcoin-cli getblockchaininfo:

```
$ bitcoin-cli getblockchaininfo


{
  "chain": "main",
  "blocks": 0,
  "headers": 83999,
  "bestblockhash": "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
  "difficulty": 1,
  "mediantime": 1231006505,
  "verificationprogress": 3.783041623201835e-09,
  "chainwork": "0000000000000000000000000000000000000000000000000000000100010001",
  "pruned": false,
  [...]
}
```

Esto muestra un nodo con una altura de cadena de bloques de 0 bloques y 83999 cabeceras. El nodo actualmente obtiene las cabeceras de bloque de la mejor cadena y luego continúa descargando los bloques completos.

Una vez que estés satisfecho con las opciones de configuración que has seleccionado, deberías agregar bitcoin a los scripts de inicio en tu sistema operativo, de modo que se ejecute continuamente y se reinicie cuando el sistema operativo se reinicie. Encontrarás varios ejemplos de scripts de inicio para varios sistemas operativos en el directorio del código fuente de bitcoin en *contrib/init* y un archivo *README.md* que muestra qué sistema usa qué script.

Interfaz de Programación de Aplicaciones (API) de Bitcoin Core

El cliente Bitcoin Core implementa una interfaz JSON-RPC a la que también se puede acceder mediante el herramienta de línea de comandos bitcoin-cli. La línea de comandos nos permite experimentar interactivamente con las capacidades que también están disponibles mediante programación a través de la API. Para comenzar, invoca el comando help para ver una lista de los comandos RPC disponibles en bitcoin:

```
$ bitcoin-cli help
addmultisigaddress nrequired ["key",...] ( "account" )
addnode "node" "add|remove|onetry"
backupwallet "destination"
createmultisig nrequired ["key",...]
createrawtransaction [{"txid":"id","vout":n},...] {"address":amount,...}
decoderawtransaction "hexstring"
...
...
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "frase de contraseña" timeout
```

```
walletpassphrasechange "frase vieja" "nueva frase"
```

Cada uno de estos comandos puede tomar una serie de parámetros. Para obtener ayuda adicional, una descripción detallada e información sobre los parámetros, añade el nombre del comando después de help. Por ejemplo, para ver la ayuda sobre el comando RPC getblockhash:

```
$ bitcoin-cli help getblockhash
getblockhash height

Devuelve el hash de bloque en la mejor cadena de bloques a la altura provista.

Argumentos:
1. height         (numerico, requiredo) El índice de altura

Resultado:
"hash"         (una cadena) El hash de bloque

Ejemplos:
> bitcoin-cli getblockhash 1000
> curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getblockhash", "params": [1000] }'
-H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Al final de la información de ayuda, verás dos ejemplos del comando RPC, usando la herramienta bitcoin-cli o el cliente HTTP curl. Estos ejemplos demuestran cómo se puede llamar al comando. Copia el primer ejemplo y mira el resultado:

```
$ bitcoin-cli getblockhash 1000
00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09
```

El resultado es un hash de bloque, que se describe con más detalle en los siguientes capítulos. Pero por ahora, este comando debe devolver el mismo resultado en tu sistema, demostrando que tu nodo Bitcoin Core se está ejecutando, está aceptando comandos y tiene información sobre el bloque 1000 a la que puedes acceder.

En las siguientes secciones mostraremos algunos comandos RPC muy útiles y su salida esperada.

Obteniendo Información del Estado del Cliente Bitcoin Core

Bitcoin Core proporciona informes de estado en diferentes módulos a través de la interfaz JSON-RPC. Los comandos más importantes incluyen getblockchaininfo, getmempoolinfo, getnetworkinfo y getwalletinfo.

El comando RPC getblockchaininfo de bitcoin ya se ha presentado anteriormente. El comando getnetworkinfo muestra información básica sobre el estado del nodo de la red bitcoin. Usa bitcoin-cli para ejecutarlo:

```
$ bitcoin-cli getnetworkinfo


  "version": 150000,
  "subversion": "/Satoshi:0.15.0/",
  "protocolversion": 70015,
  "localservices": "000000000000000d",
  "localrelay": true,
  "timeoffset": 0,
  "networkactive": true,
  "connections": 8,
  "networks": [
    ...
    información detallada sobre todas las redes (ipv4, ipv6 u onion)
    ...
  ],
  "relayfee": 0.00001000,
  "incrementalfee": 0.00001000,
  "localaddresses": [
  ],
  "warnings": ""
}
```

Los datos se devuelven en JavaScript Object Notation (JSON), un formato que todos los lenguajes de programación pueden "consumir" fácilmente, pero que también es bastante legible para los usuarios. Entre estos datos, vemos los números de versión para el cliente software de bitcoin (150000) y el protocolo bitcoin (70015). Vemos el número actual de conexiones (8) y diversa información sobre la red bitcoin y la configuración relacionada con este cliente.

TIP

Tomará algo de tiempo, quizás más de un día, para que el cliente bitcoind "alcance" la altura de la cadena de bloques actual a medida que descarga bloques de otros clientes de bitcoin. Puedes verificar su progreso usando getblockchaininfo para ver el número de bloques conocidos.

Explorando y Decodificando Transacciones

Comandos: getrawtransaction, decoderawtransaction

En Comprando una Taza de Café, Alice compró una taza de café en Bob's Cafe. Su transacción se registró en la cadena de bloques con el ID de transacción (txid) 0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fudoa57286c345c2f2. Usemos la API para recuperar y examinar esa transacción pasando el ID de transacción como parámetro:

```
$ bitcoin-cli getrawtransaction 0627052b6f28912f2703066a912ea577f2ce4da4caa5a↵
5fbd8a57286c345c2f2

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000↵
000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4↵
ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813014↵
10484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767↵
89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adfffffffff0260e3160000000↵
0001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9↵
147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000
```

TIP

Un ID de transacción no es oficial hasta que se haya confirmado la transacción. La ausencia de un hash de transacción en la cadena de bloques no significa que la transacción no se haya procesado. Esto se conoce como "maleabilidad de transacción", ya que los hashes de transacción pueden modificarse antes de la confirmación en un bloque. Después de la confirmación, el txid es inmutable y oficial.

El comando getrawtransaction devuelve una transacción serializada en notación hexadecimal. Para decodificar eso, usamos el comando decoderawtransaction, pasando los datos hexadecimales como parámetro. Puedes copiar el hex devuelto por getrawtransaction y pegarlo como parámetro para decoderawtransaction:

```
$ bitcoin-cli decoderawtransaction 0100000001186f9f998a5aa6f048e51dd8419a14d8↵
a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba474↵
6ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298↵
cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fd↵
e0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa↵
336a8d752adfffffffff0260e3160000000001976a914ab68025513c3dbd2f7b92a94e0581f5↵
d50f654e788acd0ef8000000000001976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8↵
88ac00000000

{
  "txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
  "size": 258,
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2...8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig": {
        "asm":"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1decc...",
        "hex":"483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1de..."
      },
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "n": 0,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 ab68...5f654e7 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788ac",
        "reqSigs": 1,
```

```
        "type": "pubkeyhash",
        "addresses": [
          "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
        ]
      }
    },
    {
      "value": 0.08450000,
      "n": 1,
      "scriptPubKey": {
        "asm": "OP_DUP OP_HASH160 7f9b1a...025a8 OP_EQUALVERIFY OP_CHECKSIG",
        "hex": "76a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac",
        "reqSigs": 1,
        "type": "pubkeyhash",
        "addresses": [
          "1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK"
        ]
      }
    }
  ]
}
```

La decodificación de la transacción muestra todos los componentes de esta transacción, incluidas las entradas y salidas de la transacción. En este caso, vemos que la transacción que abonó en nuestra nueva dirección con 15 milibits utilizó una entrada y generó dos salidas. La entrada a esta transacción fue la salida de una transacción confirmada previamente (que se muestra como el vin txid que empieza con 7957a35fe). Las dos salidas corresponden a 15 milibits de saldo y la salida con cambio de vuelta al remitente.

Podemos explorar más a fondo la cadena de bloques al examinar la transacción previa referida por su txid en esta transacción usando los mismos comandos (por ejemplo, getrawtransaction). Saltando de transacción en transacción, podemos seguir una cadena de transacciones hacia atrás a medida que las monedas se transmiten de la dirección de un propietario a la dirección de otro propietario.

Explorando Bloques

Comandos: getblock, getblockhash

Explorar bloques es similar a explorar transacciones. Sin embargo, los bloques pueden ser referenciados por la *altura* de bloque o por el *hash* de bloque. Primero, encontremos un bloque por su altura. En Comprando una Taza de Café, vimos que la transacción de Alice estaba incluida en el bloque 277316.

Usamos el comando getblockhash, que toma la altura de bloque como parámetro y devuelve el hash de bloque para ese bloque:

```
$ bitcoin-cli getblockhash 277316
000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

Ahora que sabemos en qué bloque se incluyó la transacción de Alice, podemos consultar ese bloque. Usamos el comando getblock con el hash de bloque como parámetro:

```
$ bitcoin-cli getblock 000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b3↵
1b2cc7bdc4
{
  "hash": "000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
  "confirmations": 37371,
  "size": 218629,
  "height": 277316,
  "version": 2,
  "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
  "tx": [
    "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "B268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",
    "04905ff987ddd4cfe603b03cfb7ca50ee81d89d1f8f5f265c38f763eea4a21fd",
    "32467aab5d04f51940075055c2f20bbd1195727c961431bf0aff8443f9710f81",
    "561c5216944e21fa29dd12aaa1a45e3397f9c0d888359cb05e1f79fe73da37bd",
[... cientos de transacciones ...]
    "78b300b2a1d2d9449b58db7bc71c3884d6e0579617e0da4991b9734cef7ab23a",
```

```
    "6c87130ec283ab4c2c493b190c20de4b28ff3caf72d16ffa1ce3e96f2069aca9",
    "6f423dbc3636ef193fd8898dfdf7621dcade1bbe509e963ffbff91f696d81a62",
    "802ba8b2adabc5796a9471f25b02ae6aeee2439c679a5c33c4bbcee97e081196",
    "eaaf6a048588d9ad4d1c092539bd571dd8af30635c152a3b0e8b611e67d1a1af",
    "e67abc6bd5e2cac169821afc51b207127f42b92a841e976f9b752157879ba8bd",
    "d38985a6a1bfd35037cb7776b2dc86797abbb7a06630f5d03df2785d50d5a2ac",
    "45ea0a3f6016d2bb90ab92c34a7aac9767671a8a84b9bcce6c019e60197c134b",
    "c098445d748ced5f178ef2ff96f2758cbec9eb32cb0fc65db313bcac1d3bc98f"
  ],
  "time": 1388185914,
  "mediantime": 1388183675,
  "nonce": 924591752,
  "bits": "1903a30c",
  "difficulty": 1180923195.258026,
  "chainwork": "0000000000000000000000000000000000000000000000000934695e92aaf53afa1a",
  "previousblockhash": "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569",
  "nextblockhash": "000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
```

El bloque contiene 419 transacciones y la transacción en la posición 64 (0627052b...) es el pago de café de Alice. La entrada height nos dice la altura del bloque, que en este caso es el bloque 277316 en la cadena de bloques.

Usando la Interfaz de Programación de Bitcoin Core

La herramienta bitcoin-cli es muy útil para explorar la API de Bitcoin Core y las funciones de prueba. Pero la razón esencial de una interfaz de programación de aplicaciones es acceder a las funciones mediante programación. En esta sección mostraremos cómo acceder a Bitcoin Core desde otro programa.

La API de Bitcoin Core es una interfaz JSON-RPC. JSON significa JavaScript Object Notation y es una forma muy conveniente de representar datos que tanto los humanos como los programas pueden leer fácilmente. RPC significa Remote Procedure Call (Llamada a Procedimiento Remoto), lo que significa que estamos llamando a procedimientos (funciones) que son remotos (en el nodo del Bitcoin Core) a través de un protocolo de red. En este caso, el protocolo de red es HTTP, o HTTPS (para conexiones cifradas).

Cuando usamos el comando bitcoin-cli para obtener ayuda sobre un comando, se nos mostró un ejemplo del uso de curl, el versátil cliente HTTP de línea de comandos para construir una de esas consultas JSON-RPC:

```
$ curl --user myusername --data-binary '{"jsonrpc": "1.0", "id":"curltest", "method": "getblockchaininfo", "params": []
}' -H 'content-type: text/plain;' http://127.0.0.1:8332/
```

Este comando muestra que curl envía una solicitud HTTP al host local (127.0.0.1), conectándose al puerto de bitcoin predeterminado (8332) y envía una solicitud jsonrpc para el método getblockchaininfo usando la codificación text/plain.

Puede notarse que curl solicitará que se envíen las credenciales junto con la solicitud. Bitcoin Core creará una contraseña aleatoria en cada inicio y la colocará en el directorio de datos bajo el nombre .cookie. El ayudante de bitcoin-cli puede leer este archivo de contraseña una vez proveído el directorio de datos. Del mismo modo, podemos copiar la contraseña y pasarla a curl (o cualquier otro envoltorio de nivel superior de Bitcoin Core RCP). Alternativamente, podemos crear una contraseña estática con el script de ayuda proporcionado en *./share/rpcauth/rpcauth.py* en el directorio de origen de Bitcoin Core.

Si estás implementando una llamada JSON-RPC en tu propio programa, puedes usar una biblioteca HTTP genérica para construir la llamada, similar a lo que se muestra en el ejemplo curl anterior.

Sin embargo, en la mayoría de los lenguajes de programación hay bibliotecas que "envuelven" la API de Bitcoin Core de una manera que hace esto mucho más simple. Usaremos la biblioteca python-bitcoinlib para simplificar el acceso a la API. Recuerda, esto requiere que tengas una instancia de Bitcoin Core en ejecución, que se utilizará para realizar llamadas JSON-RPC.

El script de Python en Ejecutando getblockchaininfo a través del API JSON-RPC de Bitcoin Core realiza una simple llamada getblockchaininfo e imprime el parámetro blocks de los datos devueltos por Bitcoin Core.

Example 5. Ejecutando getblockchaininfo a través del API JSON-RPC de Bitcoin Core

```
from bitcoin.rpc import RawProxy
```

```
# Create a connection to local Bitcoin Core node
p = RawProxy()

# Run the getblockchaininfo command, store the resulting data in info
info = p.getblockchaininfo()

# Retrieve the 'blocks' element from the info
print(info['blocks'])
```

La ejecución nos devuelve el siguiente resultado:

```
$ python rpc_example.py
394075
```

Nos dice que nuestro nodo local de Bitcoin Core tiene 394075 bloques en su cadena de bloques. No es un resultado espectacular, pero muestra el uso básico de la biblioteca como una interfaz simplificada para la API JSON-RPC de Bitcoin Core.

A continuación, usemos las llamadas getrawtransaction y decodetransaction para recuperar los detalles del pago de café de Alice. En Acceder a una transacción e iterar sus resultados., recuperamos la transacción de Alice y listamos las salidas de la transacción. Para cada salida, mostramos la dirección del destinatario y el valor. Como recordatorio, la transacción de Alice tenía una salida que era el pago a Bob's Cafe y una salida para el cambio de vuelta a Alice.

Example 6. Acceder a una transacción e iterar sus resultados.

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# Alice's transaction ID
txid = "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2"

# First, retrieve the raw transaction in hex
raw_tx = p.getrawtransaction(txid)

# Decode the transaction hex into a JSON object
decoded_tx = p.decoderawtransaction(raw_tx)

# Retrieve each of the outputs from the transaction
for output in decoded_tx['vout']:
    print(output['scriptPubKey']['addresses'], output['value'])
```

Ejecutando este código, obtenemos:

```
$ python rpc_transaction.py
([u'1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA'], Decimal('0.01500000'))
([u'1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK'], Decimal('0.08450000'))
```

Los dos ejemplos anteriores son bastante simples. Realmente no necesitas un programa para ejecutarlos; Podrías usar fácilmente la herramienta bitcoin-cli. El siguiente ejemplo, sin embargo, requiere varios cientos de llamadas RPC y muestra más claramente el uso de una interfaz programática.

En Acceder a un bloque y sumar todas las salidas de transacción, primero accedemos al bloque 277316, después accedemos a cada una de las 419 transacciones en su interior haciendo referencia a cada ID de transacción. A continuación, iteramos a través de cada una de las salidas de la transacción y sumamos el valor.

Example 7. Acceder a un bloque y sumar todas las salidas de transacción

```
from bitcoin.rpc import RawProxy

p = RawProxy()

# The block height where Alice's transaction was recorded
blockheight = 277316

# Get the block hash of block with height 277316
```

```
    blockhash = p.getblockhash(blockheight)

    # Retrieve the block by its hash
    block = p.getblock(blockhash)

    # Element tx contains the list of all transaction IDs in the block
    transactions = block['tx']

    block_value = 0

    # Iterate through each transaction ID in the block
    for txid in transactions:
        tx_value = 0
        # Retrieve the raw transaction by ID
        raw_tx = p.getrawtransaction(txid)
        # Decode the transaction
        decoded_tx = p.decoderawtransaction(raw_tx)
        # Iterate through each output in the transaction
        for output in decoded_tx['vout']:
            # Add up the value of each output
            tx_value = tx_value + output['value']

        # Add the value of this transaction to the total
        block_value = block_value + tx_value

    print("Total value in block: ", block_value)
```

Ejecutando este código, obtenemos:

```
$ python rpc_block.py

('Total value in block: ', Decimal('10322.07722534'))
```

Nuestro código de ejemplo calcula que el valor total de las transacciones en este bloque es 10,322.07722534 BTC (incluyendo la recompensa de 25 BTC y 0,0909 BTC en comisiones). Busca el hash de bloque o la altura del bloque en un explorador de bloques en la web y compara la cantidad reportada con los valores anteriores. Algunos exploradores de bloques informan del valor total excluyendo la recompensa y excluyendo las comisiones. Comprueba si puedes detectar la diferencia.

Clientes Alternativos, Bibliotecas y Kits de Herramientas

Hay muchos clientes alternativos, bibliotecas, kits de herramientas e incluso implementaciones de nodo completo en el ecosistema de bitcoin. Estos se implementan en una variedad de lenguajes de programación, ofreciendo interfaces nativas a los programadores en su lenguaje preferido.

Las siguientes secciones enumeran algunas de las mejores bibliotecas, clientes y kits de herramientas, organizados por lenguajes de programación.

C/C++

Bitcoin Core

La implementación de referencia de bitcoin

libbitcoin

Multi-Plataforma con kit de herramientas de desarrollo de C++, nodo, y biblioteca de consenso

bitcoin explorer

Herramienta de línea de comandos de Libbitcoin

picocoin

Una biblioteca de cliente ligero en lenguaje C para bitcoin por Jeff Garzik

JavaScript

bcoin

Una implementación de nodo completo modular y escalable con su API

Bitcore

Nodo completo, API y biblioteca de Bitpay

BitcoinJS

Una biblioteca de bitcoin en JavaScript puro para node.js y navegadores

Java

bitcoinj

Una librería de cliente de nodo completo en Java

PHP

bitwasp/bitcoin

Una biblioteca de bitcoin en PHP y proyectos relacionados

Python

python-bitcoinlib

Una biblioteca de bitcoin en Python, biblioteca de consenso y nodo, por Peter Todd

pycoin

Una biblioteca de bitcoin en Python, por Richard Kiss

pybitcointools

Código bifurcado de una biblioteca Bitcoin de Python por Vitalik Buterin

Ruby

bitcoin-client

Una biblioteca en Ruby que encapsula el API JSON-RPC

Go

btcd

Un cliente bitcoin de nodo completo en lenguaje Go

Rust

rust-bitcoin

Biblioteca bitcoin en lenguaje Rust para la serialización, interpretación, y ejecución de llamadas API

C#

NBitcoin

Biblioteca de bitcoin completa para el framework .NET

Objective-C

CoreBitcoin

Kit de herramientas de Bitcoin para ObjC y Swift

Existen muchas más bibliotecas en otros lenguajes de programación y se crean más todo el tiempo.

Llaves, Direcciones

Es posible que hayas escuchado que bitcoin se basa en *criptografía*, que es una rama de las matemáticas que se usa ampliamente en seguridad informática. Criptografía significa "escritura secreta" en griego, pero la ciencia de la criptografía abarca más que la escritura secreta, que se conoce como cifrado. La criptografía también se puede usar para probar el conocimiento de un secreto sin revelar ese secreto (firma digital), o para probar la autenticidad de los datos (huella digital). Estos tipos de pruebas criptográficas son herramientas matemáticas críticas para bitcoin y se utilizan ampliamente en aplicaciones de bitcoin. Irónicamente, el cifrado no es una parte importante de bitcoin, ya que sus datos de comunicaciones y transacciones no están cifrados y no necesitan estar cifrados para proteger los fondos. En este capítulo presentaremos algunos de los elementos de criptografía que se utilizan en bitcoin para controlar la propiedad de los fondos, en forma de llaves, direcciones y carteras.

Introducción

La propiedad de bitcoin se establece a través de *llaves digitales*, *direcciones bitcoin* y *firmas digitales*. Las llaves digitales no se almacenan realmente en la red, sino que los usuarios las crean y almacenan en un archivo o simple base de datos, denominada *cartera*. Las llaves digitales en la cartera de un usuario son completamente independientes del protocolo bitcoin y pueden ser generadas y administradas por el software de la cartera del usuario sin referencia alguna a la cadena de bloques o acceso a internet. Las llaves permiten muchas de las propiedades interesantes de bitcoin, incluidas la confianza descentralizada y el control, comprobación de propiedad y el modelo de seguridad de resistencia criptográfica.

La mayoría de las transacciones de bitcoin requieren una firma digital válida para ser incluidas en la cadena de bloques, que solo se pueden generar con una llave secreta; por lo tanto, cualquier persona con una copia de esa llave tiene el control del bitcoin. La firma digital utilizada para gastar fondos también se conoce como *testigo* (en inglés, *witness*), un término usado en criptografía. Los datos testigo en una transacción bitcoin certifican la verdadera propiedad de los fondos que se están gastando.

Las llaves vienen en pares que consisten en una llave privada (secreta) y una llave pública. Imagina que la llave pública es similar a un número de cuenta bancaria y que la llave privada es similar al PIN secreto, o la firma en un cheque que proporciona control sobre la cuenta. Estas llaves digitales rara vez son vistas por los usuarios de bitcoin. Normalmente se almacenan dentro del archivo de cartera y son administrados por el software de cartera de bitcoin.

En la parte del pago de una transacción de bitcoin, la llave pública del destinatario está representada por su huella digital, denominada *dirección bitcoin*, que se usa de la misma manera que el nombre del beneficiario en un cheque (es decir, "Páguese a la orden de") . En la mayoría de los casos, una dirección bitcoin se genera a partir de y corresponde a una llave pública. Sin embargo, no todas las direcciones bitcoin representan llaves públicas; también pueden representar otros beneficiarios, como scripts, como veremos más adelante en este capítulo. De esta manera, las direcciones bitcoin abstraen al destinatario de fondos, flexibilizando el destino de las transacciones, de forma similar a los cheques de papel: un único instrumento de pago que puede ser usado para pagar a cuentas de personas, de empresas, pagar facturas o pagar con efectivo. La dirección bitcoin es la única representación de las llaves que los usuarios verán habitualmente, ya que esta es la parte que necesitan compartir con el mundo.

Primero, presentaremos la criptografía y explicaremos las matemáticas utilizadas en bitcoin. A continuación, veremos cómo se generan, almacenan y administran las llaves. Revisaremos los diversos formatos de codificación utilizados para representar llaves privadas y públicas, direcciones, y direcciones de script. Finalmente, veremos el uso avanzado de las llaves y direcciones: vanidad, multifirma, y direcciones de script y carteras de papel.

Criptografía de Llave Pública y Criptomonedas

La criptografía de llave pública se inventó en la década de 1970 y es la base matemática para la seguridad informática y de la información.

Desde la invención de la criptografía de llave pública, se han descubierto varias funciones matemáticas adecuadas, como la exponenciación de números primos y la multiplicación de curva elíptica. Estas funciones matemáticas son prácticamente irreversibles, lo que significa que son fáciles de calcular en una dirección e inviables de calcular en la dirección opuesta. Sobre la base de estas funciones matemáticas, la criptografía permite la creación de secretos digitales y firmas digitales infalsificables. Bitcoin utiliza la multiplicación de curva elíptica como base para su criptografía.

En bitcoin, utilizamos la criptografía de llave pública para crear un par de llaves que controla el acceso a bitcoin. El par de llaves consiste en una llave privada y, derivada de ella, una llave pública única. La llave pública se utiliza para recibir

fondos, y la llave privada se usa para firmar transacciones para gastar los fondos.

Existe una relación matemática entre la llave pública y la privada que permite que la llave privada se use para generar firmas en los mensajes. Estas firmas se pueden validar con la llave pública sin revelar la llave privada.

Cuando los bitcoin son gastados, el dueño actual de los bitcoin presenta su llave pública y firma (diferente cada vez, pero creada a partir de la misma llave privada) en una transacción para gastar esos bitcoin. A través de la presentación de la llave pública y la firma, todos los participantes de la red bitcoin pueden verificar y aceptar la transacción como válida, confirmando que la persona que transfirió el bitcoin era la propietaria en el momento de la transferencia.

TIP

En la mayoría de las implementaciones de carteras, las llaves privadas y públicas se almacenan juntas como un *par de llaves* para mayor comodidad. Sin embargo, la llave pública se puede calcular a partir de la llave privada, por lo que también es posible almacenar solo la llave privada.

Llaves Privadas y Públicas

Una cartera bitcoin contiene una colección de pares de llaves, cada uno de las cuales consta de una llave privada y una pública. La llave privada (k) es un número, generalmente escogido aleatoriamente. A partir de la llave privada, utilizamos multiplicación de curva elíptica, una función criptográfica de sentido único, para generar una llave pública (K). A partir de la llave pública (K), usamos una función hash criptográfica de sentido único para generar una dirección bitcoin (A). En esta sección, comenzaremos con la generación de la llave privada, veremos las matemáticas de curva elíptica que se usa para convertirla en una llave pública y, finalmente generar una dirección bitcoin a partir de la llave pública. La relación entre la llave privada, la llave pública y la dirección bitcoin se muestra en Llave privada, llave pública y dirección bitcoin.

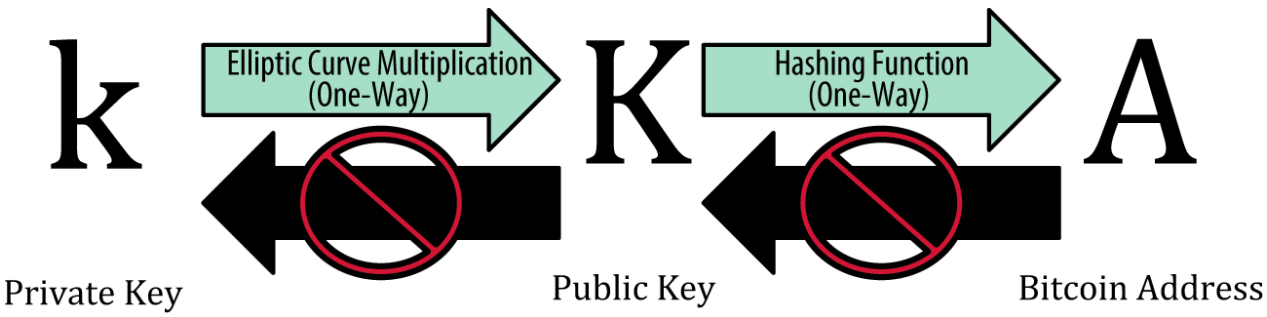


Figure 12. Llave privada, llave pública y dirección bitcoin

¿Por Qué Usar Criptografía Asimétrica (Llaves Públicas/Privadas)?

¿Por qué se utiliza la criptografía asimétrica en bitcoin? No se utiliza para "cifrar" (hacer secretas) las transacciones. Más bien, la propiedad útil de la criptografía asimétrica es la capacidad de generar firmas digitales. Se puede aplicar una llave privada a la huella digital de una transacción para producir una firma numérica. Esta firma solo puede ser producida por alguien con conocimiento de la llave privada. Sin embargo, cualquier persona con acceso a la llave pública y la huella digital de la transacción puede usarla para verificar la firma. Esta útil propiedad de la criptografía asimétrica hace posible que cualquier persona pueda verificar cada firma en cada transacción, mientras se asegura que solo los propietarios de llaves privadas puedan producir firmas válidas.

Llaves Privadas

Una llave privada es simplemente un número, escogido al azar. La propiedad y el control sobre la llave privada es la raíz del control del usuario sobre todos los fondos asociados con la dirección bitcoin correspondiente. La llave privada se usa para crear las firmas requeridas para gastar bitcoin al demostrar la propiedad de los fondos utilizados en una transacción. La llave privada debe permanecer en secreto en todo momento, ya que revelarla a terceros equivale a darles el control sobre el bitcoin asegurado por esa llave. También deben hacerse copias de respaldo de la llave privada para protegerla de pérdidas accidentales, ya que si se pierde, no se puede recuperar y los fondos asegurados por ella también se pierden para siempre.

TIP

La llave privada de bitcoin es solo un número. Puedes elegir tus llaves privadas al azar usando únicamente una moneda, lápiz y papel: lanza una moneda 256 veces y tendrás los dígitos binarios de una llave privada aleatoria que puedes usar en una cartera bitcoin. La llave pública se puede generar después a partir de la llave privada.

Generando una llave privada a partir de un número aleatorio

El primer paso y el más importante para generar llaves es encontrar una fuente segura de entropía o aleatoriedad. Crear una llave de bitcoin es esencialmente lo mismo que "Elija un número entre 1 y 2^{256}". El método exacto utilizado para elegir ese número no importa siempre que no sea predecible o repetible. El software bitcoin utiliza los generadores de números aleatorios del sistema operativo subyacente para producir 256 bits de entropía (aleatoriedad). Usualmente, el generador de números aleatorios del sistema operativo se inicializa mediante una fuente humana de aleatoriedad, por lo que puede que se te solicite que muevas el ratón durante unos segundos.

Más precisamente, la llave privada puede ser cualquier número entre 0 y n - 1 inclusive, donde n es una constante (n = 1.1578 * 10^{77}, ligeramente menor que 2^{256}) definida como el orden de la curva elíptica utilizada en bitcoin (ver Criptografía de Curva Elíptica Explicada). Para crear tal llave, elegimos aleatoriamente un número de 256 bits y verificamos que sea menor que n. En términos de programación, esto generalmente se logra cuando una cadena más grande de bits aleatorios recopilados desde una fuente de aleatoriedad criptográficamente segura, se alimenta en el algoritmo de hash SHA256, que producirá convenientemente un número de 256 bits. Si el resultado es menor que n, hemos obtenido una llave privada apropiada. De lo contrario, simplemente lo intentamos de nuevo con otro número aleatorio.

WARNING

No escribas tu propio código para generar un número aleatorio ni uses un generador de números aleatorios "simple" ofrecido por tu lenguaje de programación. Utiliza un generador de números pseudoaleatorios criptográficamente seguro (CSPRNG) con una semilla a partir de una fuente de entropía suficiente. Estudia la documentación de la biblioteca del generador de números aleatorios que elijas para asegurarte de que sea criptográficamente seguro. La correcta implementación de los CSPRNG es fundamental para la seguridad de las llaves.

La siguiente es una llave privada (k) generada aleatoriamente que se muestra en formato hexadecimal (256 bits se muestran como 64 dígitos hexadecimales, cada uno de 4 bits):

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

TIP

El tamaño del espacio de llaves privadas de bitcoin, (2^{256}) es un número inimaginablemente grande. Es aproximadamente 10^{77} en decimal. A modo de comparación, se estima que el universo visible contiene 10^{80} átomos.

Para generar una llave nueva con el Cliente Principal de Bitcoin (ver Bitcoin Core: La Implementación de Referencia), usa el comando getnewaddress. Por razones de seguridad, solo muestra la llave pública, no la llave privada. Para pedir a bitcoind que muestre la llave privada, usa el comando dumpprivkey. El comando dumpprivkey muestra la llave privada en un formato Base58 que incorpora un checksum, llamado *Wallet Import Format* (WIF), y que examinaremos con más detalle en Formatos de llaves privadas. Aquí hay un ejemplo de cómo generar y mostrar una llave privada usando estos dos comandos:

```
$ bitcoin-cli getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoin-cli dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

El comando dumpprivkey abre la cartera y extrae la llave privada generada por el comando getnewaddress. No es posible que bitcoind conozca la llave privada de la llave pública a menos que ambas estén almacenadas en la cartera.

TIP

El comando dumpprivkey no genera una llave privada a partir de una llave pública, ya que esto es imposible. El comando simplemente revela la llave privada que ya es conocida por la cartera y que fue generada por el comando getnewaddress.

También puedes usar la herramienta de línea de comandos del Explorador de Bitcoin (ver Comandos del "Bitcoin Explorer" (bx)) para generar y mostrar llaves privadas con los comandos seed, ec-new, y ec-to-wif:

```
$ bx seed | bx ec-new | bx ec-to-wif
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Llaves Públicas

La llave pública se calcula a partir de la llave privada mediante la multiplicación de curva elíptica, que es irreversible: $K = k * G$, donde k es la llave privada, G es un punto constante llamado *punto generador*, y K es la llave pública resultante. La operación inversa, conocida como "encontrar el logaritmo discreto" —calcular k si se sabe K— es tan difícil como probar con todos los valores posibles de k, es decir, una búsqueda por fuerza bruta. Antes de mostrar cómo generar una llave pública a partir de una llave privada, veamos la criptografía de curva elíptica con un poco más de detalle.

TIP

La multiplicación de curva elíptica es un tipo de función que los criptógrafos llaman una función de "trampilla": es fácil de hacer en una dirección (multiplicación) e imposible de hacer en la dirección inversa (división). El propietario de la llave privada puede crear fácilmente la llave pública y luego compartirla con todo el mundo sabiendo que nadie puede revertir la función y calcular la llave privada a partir de la llave pública. Este truco matemático es la base de las firmas digitales infalsificables y seguras que atestiguan la propiedad de los fondos de bitcoin.

Criptografía de Curva Elíptica Explicada

La criptografía de curva elíptica es un tipo de criptografía asimétrica o de llave pública basada en el problema del logaritmo discreto expresado por suma y multiplicación sobre puntos de una curva elíptica.

Una curva elíptica es un ejemplo de una curva elíptica, similar a las usadas por bitcoin.

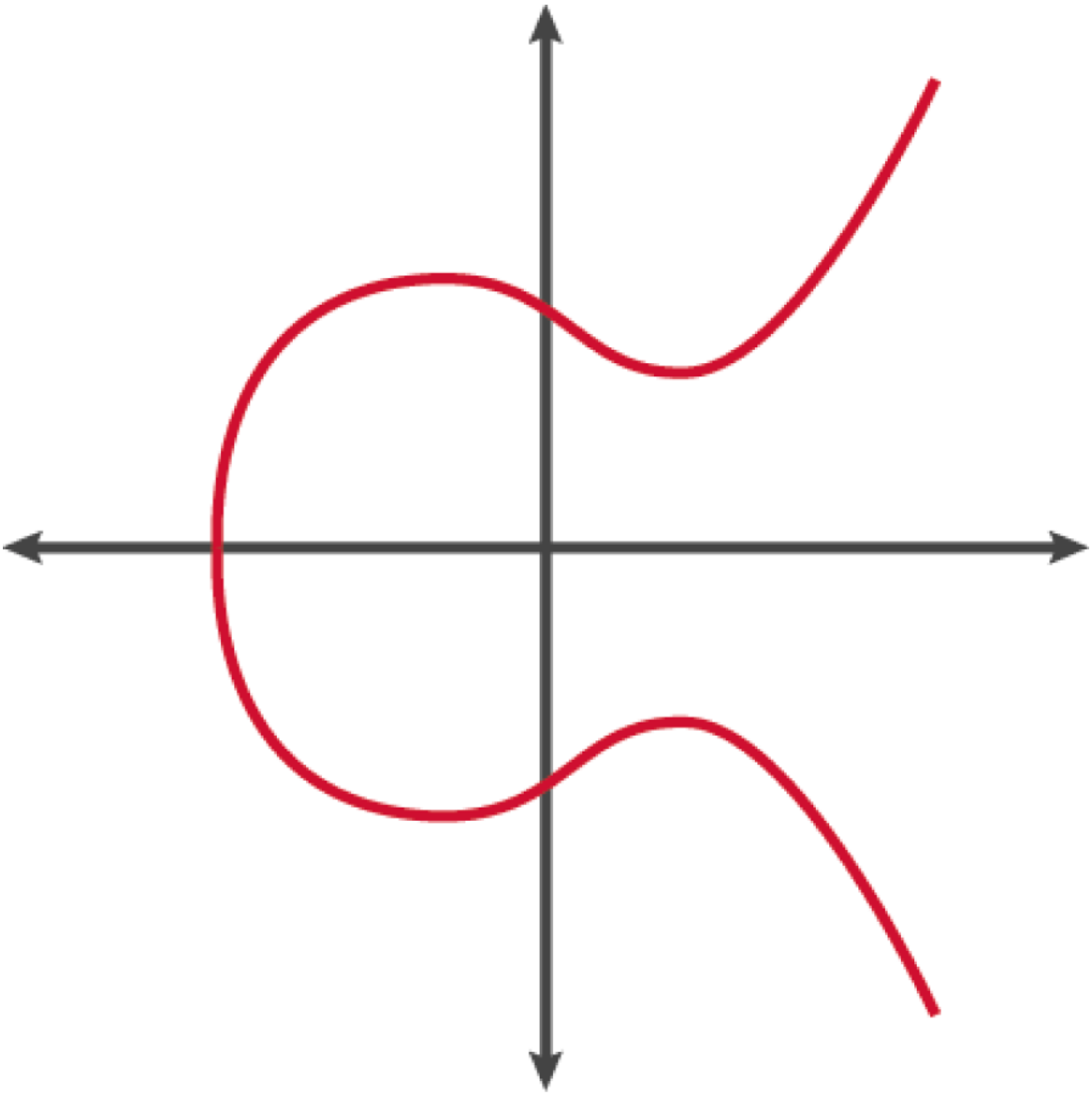


Figure 13. Una curva elíptica

Bitcoin usa una curva elíptica específica y un conjunto de constantes matemáticas definidas en un estándar llamado secp256k1, establecido por el Instituto Nacional de Estándares y Tecnología (National Institute of Standards and Technology, o NIST). La curva secp256k1 se define mediante la siguiente función, la cual produce una curva elíptica:

\[\begin{equation} {y^2 = (x^3 + 7)}~\text{over}~(\mathbb{F}_p) \end{equation}\]

ó

\[\begin{equation} {y^2 \mod p = (x^3 + 7) \mod p} \end{equation}\]

El *mod p* (módulo del número primo p) indica que esta curva se encuentra sobre un campo finito de orden primo *p*, también escrito como \(\(\mathbb{F}_p \)\), donde p = $2^{256} – 2^{32} – 2^9 – 2^8 – 2^7 – 2^6 – 2^4 – 1$, un número primo muy grande.

Debido a que esta curva se define sobre un campo finito de orden primo en vez de sobre los números reales, se ve como un patrón de puntos dispersos en dos dimensiones, lo que dificulta su visualización. Sin embargo, las matemáticas son idénticas a las de una curva elíptica sobre números reales. Como ejemplo, Criptografía de curva elíptica: visualizando una curva elíptica sobre F(p), con p=17 muestra la misma curva elíptica en un campo finito mucho más pequeño de orden primo 17, mostrando un patrón de puntos sobre una cuadrícula. La curva elíptica de bitcoin secp256k1 se puede considerar como un patrón de puntos mucho más complejo sobre una cuadrícula inconmensurablemente grande.

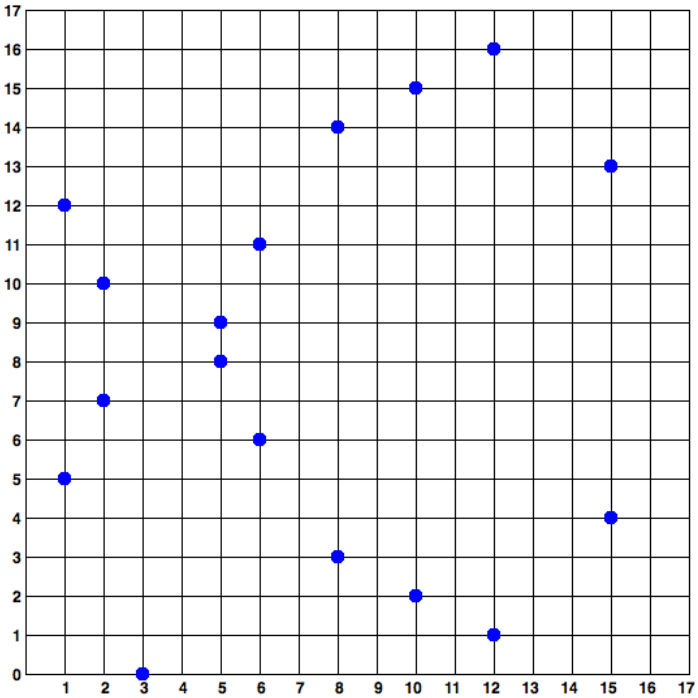


Figure 14. Criptografía de curva elíptica: visualizando una curva elíptica sobre F(p), con p=17

Así que, por ejemplo, lo siguiente es un punto P con coordenadas (x,y) que es un punto en la curva secp256k1:

```
P = (55066263022277343669578718895168534326250603453777594175500187360389116729240,
32670510020758816978083085130507043184471273380659243275938904335757337482424)
```

Usando Python para confirmar que este punto está en la curva elíptica muestra cómo puedes comprobarlo tú mismo usando Python:

Example 8. Usando Python para confirmar que este punto está en la curva elíptica

```
Python 3.4.0 (default, Mar 30 2014, 19:23:13)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> p = 115792089237316195423570985008687907853269984665640564039457584007908834671663
>>> x = 55066263022277343669578718895168534326250603453777594175500187360389116729240
>>> y = 32670510020758816978083085130507043184471273380659243275938904335757337482424
>>> (x ** 3 + 7 - y**2) % p
0
```

En las matemáticas de curva elíptica, hay un punto llamado "punto en el infinito", que se corresponde aproximadamente con el rol de cero en la suma. En las computadoras, a veces se representa con x = y = 0 (lo que no satisface la ecuación de curva elíptica, pero es un caso aislado simple que se puede verificar).

Existe también un operador +, llamado "suma", que tiene algunas propiedades similares a la suma tradicional de números

reales que aprenden los niños en la escuela. Dados dos puntos P_1 y P_2 en la curva elíptica, hay un tercer punto $P_3 = P_1 + P_2$, también en la curva elíptica.

Geométricamente, este tercer punto P_3 es calculado dibujando una línea entre P_1 y P_2. Esta línea intersecará la curva elíptica en exactamente un punto adicional. Llamemos a este punto $P_3' = (x, y)$. Luego reflejamos el eje x para obtener $P_3 = (x, –y)$.

Existen un par de casos especiales que explican la necesidad del "punto en el infinito".

Si P_1 y P_2 son el mismo punto, la línea "entre" P_1 y P_2 debe extenderse para ser la tangente sobre la curva en el punto P_1. Esta tangente intersecará la curva en exactamente un nuevo punto. Puedes usar técnicas de cálculo para determinar la pendiente de la línea tangencial. Estas técnicas curiosamente funcionan a pesar de estar restringiendo nuestro interés a puntos sobre la curva con coordenadas de dos enteros.

En algunos casos (es decir, si P_1 y P_2 poseen los mismos valores x pero diferentes valores y), la línea tangente será exactamente vertical, en cuyo caso P_3 = "punto al infinito".

Si P_1 es el "punto en el infinito", entonces $P_1 + P_2 = P_2$. De manera similar, si P_2 es el punto en el infinito, entonces $P_1 + P_2 = P_1$. Esto muestra cómo el punto en el infinito desempeña el rol de cero.

Resulta que el + es asociativo, lo que significa que (A + B) + C = A + (B + C). Eso significa que podemos escribir A + B + C sin paréntesis y sin ambigüedad.

Ahora que hemos definido la suma, podemos definir la multiplicación en la forma estándar en que extiende a la suma. Para un punto P en la curva elíptica, si k es un número entero, entonces kP = P + P + P + … + P (k veces). Nótese que k es a veces llamado un "exponente" en este caso, lo cual puede causar confusión.

Generando una Llave Pública

Comenzando con una llave privada en la forma de un número *k* generado aleatoriamente, lo multiplicamos por un punto predeterminado en la curva llamado *punto generador G* para producir otro punto en otro lugar en la curva, que es la llave pública correspondiente *K*. El punto generador se especifica como parte del estándar secp256k1 y siempre es el mismo para todas las llaves en bitcoin:

\[\begin{equation} {K = k * G} \end{equation}\]

donde *k* es la llave privada, *G* es el punto generador y *K* es la llave pública resultante, un punto en la curva. Debido a que el punto generador siempre es el mismo para todos los usuarios de bitcoin, una llave privada *k* multiplicada por *G* siempre dará como resultado la misma llave pública *K*. La relación entre *k* y *K* es fija, pero solo se puede calcular en una dirección, desde *k* a *K*. Es por eso que una dirección bitcoin (derivada de *K*) se puede compartir con cualquier persona y no revela la llave privada del usuario (*k*).

> **TIP** Una llave privada puede ser convertida a una llave pública, pero una llave pública no puede ser convertida en una llave privada ya que la matemática solo funciona en un sentido.

Para implementar la multiplicación de curva elíptica, tomamos la llave privada *k* generada previamente y la multiplicamos por el punto generador G para encontrar la llave pública *K*:

```
K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD * G
```

La llave pública *K* se define como un punto K = (x,y):

```
K = (x, y)

donde,

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Para visualizar la multiplicación de un punto con un número entero, usaremos la curva elíptica más simple sobre los números reales—recuerda, la matemática es la misma. Nuestro objetivo es encontrar el múltiplo *kG* del punto generador *G*, que es lo mismo que agregar *G* a sí mismo, *k* veces seguidas. En las curvas elípticas, sumar un punto a sí mismo es el

equivalente a dibujar una línea tangente en el punto y encontrar dónde interseca la curva nuevamente, y luego reflejar ese punto en el eje x.

Criptografía de curva elíptica: visualizando la multiplicación de un punto G por un entero k en una curva elíptica muestra el proceso de derivar *G*, *2G*, *4G*, como una operación geométrica en la curva.

TIP Bitcoin utiliza la secp256k1 optimized C library para hacer los cálculos matemáticos de curva elíptica.

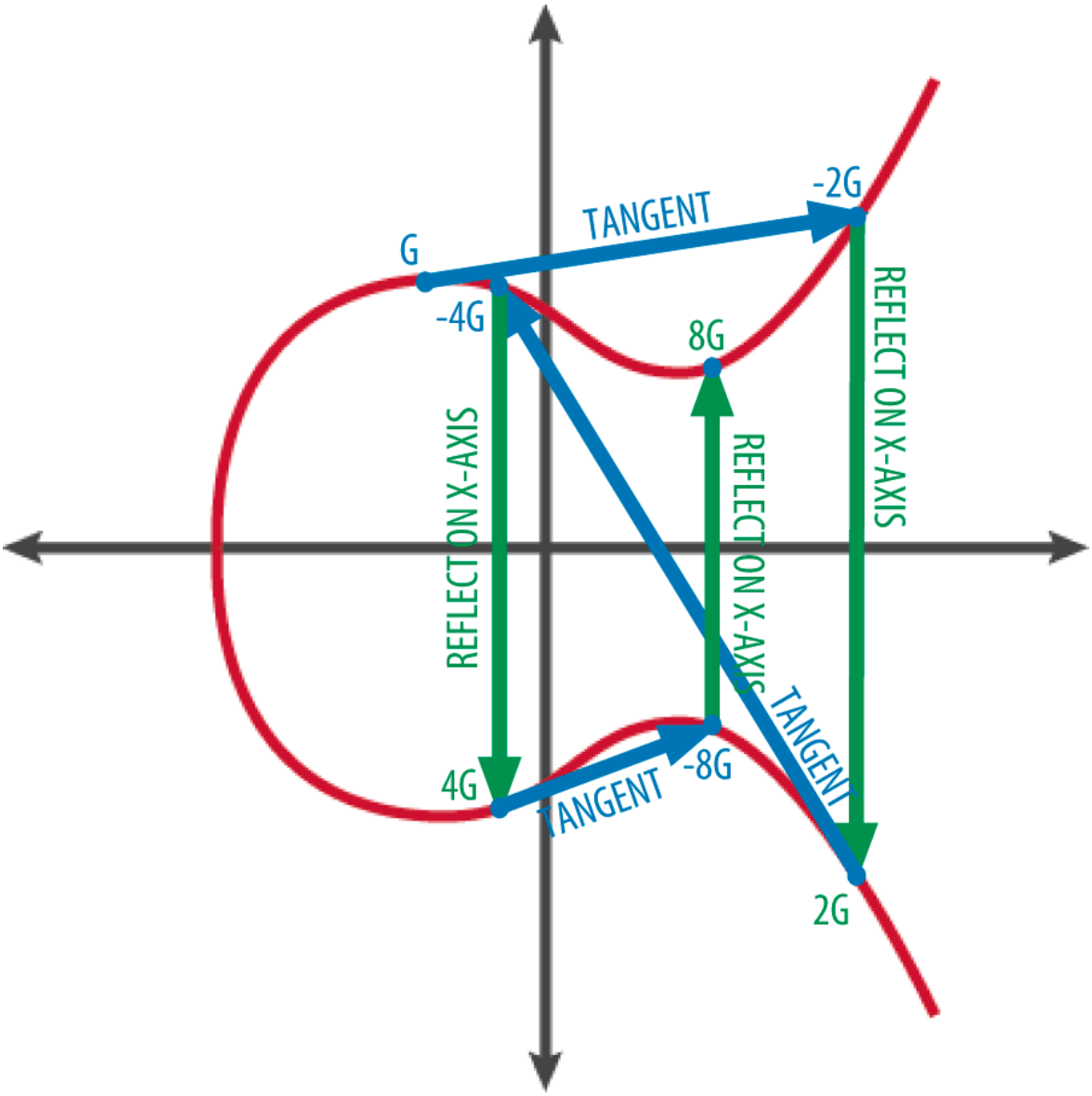


Figure 15. Criptografía de curva elíptica: visualizando la multiplicación de un punto G por un entero k en una curva elíptica

Direcciones Bitcoin

Una dirección bitcoin es una cadena de dígitos y caracteres que se puede compartir con cualquier persona que desee enviarte dinero. Las direcciones producidas a partir de llaves públicas consisten en una serie de números y letras, que comienzan con el dígito "1". Aquí hay un ejemplo de una dirección bitcoin:

```
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
```

La dirección bitcoin es lo que aparece más comúnmente en una transacción como el "destinatario" de los fondos. Si comparamos una transacción de bitcoin con un cheque en papel, la dirección bitcoin es el beneficiario, que es lo que escribimos en la línea después de "Páguese a la orden de". En un cheque en papel, ese beneficiario a veces puede ser el nombre del titular de una cuenta bancaria, pero también puede incluir empresas, instituciones o incluso efectivo. Debido a que los cheques en papel no necesitan especificar una cuenta, sino que usan un nombre abstracto como destinatario de los fondos, son instrumentos de pago muy flexibles. Las transacciones de bitcoin utilizan una abstracción similar, la dirección bitcoin, para hacerlas muy flexibles. Una dirección bitcoin puede representar al propietario de un par de llaves

privada/pública, o puede representar otra cosa, como un script de pago, como veremos en Pay-to-Script-Hash (P2SH). Por ahora, examinemos el caso simple, una dirección bitcoin que representa, y se deriva de, una llave pública.

La dirección bitcoin se deriva de la llave pública mediante el uso de un hash criptográfico unidireccional. Un "algoritmo de cálculo de hash" o simplemente "algoritmo hash" es una función unidireccional que produce una huella digital o "hash" de una entrada de tamaño arbitrario. Las funciones hash criptográficas se utilizan ampliamente en bitcoin: en direcciones bitcoin, en direcciones de script y en el algoritmo de Prueba-de-Trabajo de minería. Los algoritmos utilizados para crear una dirección bitcoin desde una llave pública son el Secure Hash Algorithm (SHA) y el RACE Integrity Primitives Evaluation Message Digest (RIPEMD), específicamente SHA256 y RIPEMD160.

A partir de la llave pública *K*, calculamos el hash SHA256 y después calculamos el hash RIPEMD160 del resultado, produciendo un número de 160 bits (20 bytes):

\[\begin{equation} {A = RIPEMD160(SHA256(K))} \end{equation}\]

donde *K* es la llave pública y *A* es la dirección bitcoin resultante.

> **TIP** Una dirección bitcoin *no* es lo mismo que una llave pública. Las direcciones bitcoin se derivan de una llave pública utilizando una función unidireccional.

Las direcciones bitcoin casi siempre están codificadas como "Base58Check" (ver Codificación Base58 y Base58Check), que utiliza 58 caracteres (un sistema numérico Base58) y un checksum para ayudar a la legibilidad humana, evitar la ambigüedad y proteger contra errores en la transcripción y tecleo de direcciones. También se usa Base58Check de muchas otras formas en bitcoin, siempre que sea necesario que un usuario lea y transcriba correctamente un número, como una dirección bitcoin, una llave privada, una llave cifrada o un hash de script. En la siguiente sección examinaremos la mecánica de la codificación y decodificación de Base58Check y las representaciones resultantes. Llave pública a dirección bitcoin: conversión de una llave pública en una dirección bitcoin ilustra la conversión de una llave pública en una dirección bitcoin.

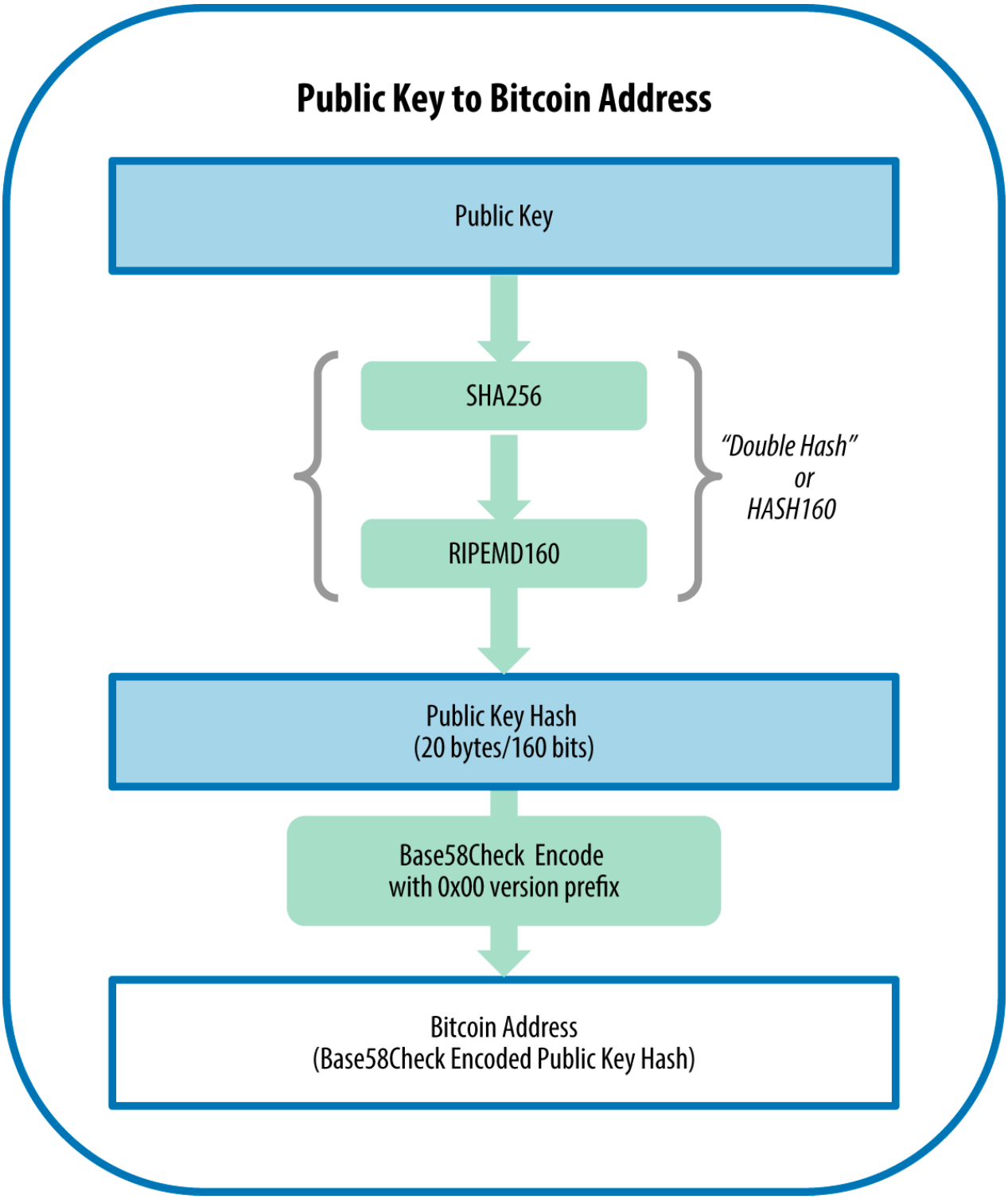


Figure 16. Llave pública a dirección bitcoin: conversión de una llave pública en una dirección bitcoin

Codificación Base58 y Base58Check

Para representar números largos de forma compacta, usando menos símbolos, muchos sistemas informáticos utilizan representaciones alfanuméricas mixtas con una base (o radix) superior a 10. Por ejemplo, mientras que el sistema decimal tradicional utiliza los 10 números del 0 al 9, el sistema hexadecimal usa 16, con las letras de la A a la F como los seis símbolos adicionales. Un número representado en formato hexadecimal es más corto que la representación decimal equivalente. Aún más compacta, la representación en Base64 usa 26 letras minúsculas, 26 letras mayúsculas, 10 números y 2 caracteres más, como “` `” y "/" para transmitir datos binarios a través de medios basados en texto, como el correo electrónico. Base64 se usa comúnmente para adjuntar archivos binarios al correo electrónico. Base58 es un formato de codificación binaria basado en texto desarrollado para su uso en bitcoin y utilizado en muchas otras criptomonedas. Ofrece un equilibrio entre la representación compacta, la legibilidad y la detección y prevención de errores. Base58 es un subconjunto de Base64, que usa letras y números en mayúsculas y minúsculas, pero omite algunos caracteres que con frecuencia se confunden entre sí y pueden parecer idénticos cuando se muestran en ciertas fuentes. Específicamente, Base58 es Base64 sin el 0 (número cero), O (o mayúscula), l (L minúscula), I (i mayúscula) y los símbolos

“``” y "/". Puesto de forma más sencilla, es un conjunto de letras minúsculas y mayúsculas y números, sin los cuatro (0, O, l, I) mencionados. Alfabeto Base58 de bitcoin muestra el alfabeto Base58 completo.

Example 9. Alfabeto Base58 de bitcoin

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

Para agregar seguridad adicional contra errores tipográficos o de transcripción, Base58Check es un formato de codificación de Base58, usado frecuentemente en bitcoin, el cual tiene un código de verificación de errores incorporado. El checksum consiste en cuatro bytes adicionales agregados al final de los datos que se están codificando. El checksum se deriva del hash de los datos codificados y, por lo tanto, se puede utilizar para detectar y prevenir errores de transcripción y de tecleo. Cuando se presenta un código Base58Check, el software de decodificación calculará el checksum de los datos y lo comparará con el checksum incluido en el código. Si no son idénticos, se ha introducido un error y los datos de Base58Check no son válidos. Esto evita que una dirección bitcoin mal escrita sea aceptada por el software de la cartera como un destino válido, un error que de lo contrario resultaría en la pérdida de fondos.

Para convertir datos (un número) al formato Base58Check, primero agregamos un prefijo a los datos, llamado "byte de versión", que sirve para identificar fácilmente el tipo de datos que están codificados. Por ejemplo, en el caso de una dirección bitcoin, el prefijo es cero (0x00 en hexadecimal), mientras que el prefijo utilizado para codificar una llave privada es 128 (0x80 en hexadecimal). Se muestra una lista de los prefijos de versión comunes en Prefijos de versión Base58Check y ejemplos de resultados codificados.

A continuación computamos el checksum "doble SHA", lo que significa que aplicamos el algoritmo de hash SHA256 dos veces sobre resultado previo (prefijo y datos):

```
checksum = SHA256(SHA256(prefijo+datos))
```

Del hash de 32 bytes resultante (hash-de-un-hash), tomamos solo los primeros cuatro bytes. Estos cuatro bytes sirven como código de comprobación de errores o checksum. El checksum se concatena (se anexa) al final.

El resultado está compuesto de tres elementos: un prefijo, los datos y un checksum. Este resultado es codificado usando el alfabeto Base58 descrito anteriormente. Codificación Base58Check: un formato Base58, con versión y checksum para codificar datos bitcoin sin ambigüedades ilustra el proceso de codificación Base58Check.

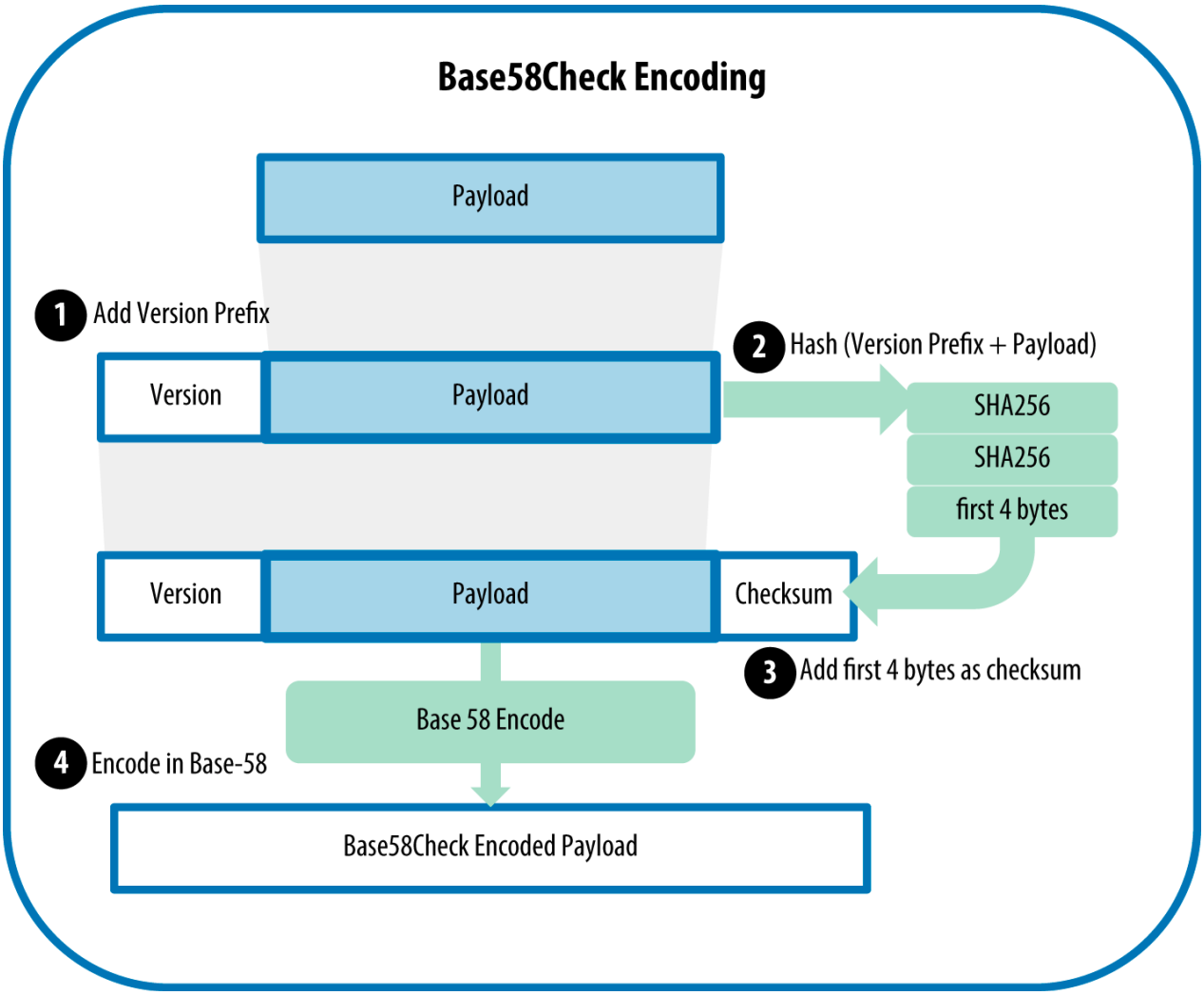


Figure 17. Codificación Base58Check: un formato Base58, con versión y checksum para codificar datos bitcoin sin ambigüedades

En bitcoin, la mayoría de los datos presentados al usuario están codificados en Base58Check para que sean compactos, fáciles de leer y facilitar la detección de errores. El prefijo de versión en la codificación Base58Check se utiliza para crear formatos fácilmente distinguibles, que cuando están codificados en Base58 contienen caracteres específicos al principio de la carga útil codificada en Base58Check. Estos caracteres facilitan a los humanos identificar el tipo de datos que están codificados y cómo usarlos. Esto es lo que diferencia, por ejemplo, una dirección bitcoin codificada en Base58Check que comienza con un 1 de una llave privada codificada en Base58Check que comienza con un 5. Algunos prefijos de versión de ejemplo y los caracteres Base58 resultantes se muestran en Prefijos de versión Base58Check y ejemplos de resultados codificados.

Table 1. Prefijos de versión Base58Check y ejemplos de resultados codificados

Tipo	Prefijo de versión (hexadecimal)	Prefijo del resultado Base58
Dirección Bitcoin	0x00	1
Dirección Pago-a-Hash-de-Script	0x05	3
Dirección Bitcoin para red de pruebas	0x6F	m ó n
Llave Privada WIF	0x80	5, K, ó L
Llave Privada Cifrada BIP-38	0x0142	6P
Llave Pública Extendida BIP-32	0x0488B21E	xpub

Formatos de Llaves

Tanto las llaves privadas como las públicas se pueden representar en varios formatos diferentes. Todas estas representaciones codifican el mismo número, aunque se vean diferentes. Estos formatos se utilizan principalmente para facilitar que las personas lean y transcriban llaves sin introducir errores.

Formatos de llaves privadas

La llave privada se puede representar en varios formatos diferentes, todos los cuales corresponden al mismo número de 256 bits. Representaciones de llaves privadas (formatos de codificación) muestra tres formatos comunes utilizados para representar llaves privadas. Se utilizan diferentes formatos en diferentes circunstancias. Los formatos hexadecimales y binarios sin formato se utilizan internamente en el software y rara vez se muestran a los usuarios. El WIF se usa para importar/exportar llaves entre carteras y, a menudo, en representaciones de llaves privadas mediante códigos QR (código de barras).

Table 2. Representaciones de llaves privadas (formatos de codificación)

Tipo	Prefijo	Descripción
En crudo (en inglés, raw)	Ninguno	32 bytes
Hexadecimal	Ninguno	64 dígitos hexadecimales
WIF	5	codificación Base58Check: Base58 con prefijo de versión de 128 bits- y checksum de 32 bits
WIF comprimido	K ó L	Como el caso anterior, con sufijo 0x01 añadido antes de codificar

La tabla Ejemplo: Misma llave, formatos distintos muestra la llave privada generada en estos tres formatos.

Table 3. Ejemplo: Misma llave, formatos distintos

Formato	Llave privada
Hexadecimal	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF comprimido	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Todas estas representaciones son formas distintas de mostrar el mismo número, la misma llave privada. Se ven diferentes, pero cualquiera de estos formatos puede ser convertido fácilmente a cualquier otro formato. Ten en cuenta que el "binario sin formato" (en inglés, "raw binary") no se muestra en Ejemplo: Misma llave, formatos distintos, ya que, por definición, cualquier codificación para visualización no pueden ser datos binarios sin formato.

Usamos el comando wif-to-ec de Bitcoin Explorer (ver Comandos del "Bitcoin Explorer" (bx)) para mostrar que ambas llaves WIF representan la misma llave privada:

```
$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

$ bx wif-to-ec KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
```

Decodificar a partir de Base58Check

Los comandos de Bitcoin Explorer (ver Comandos del "Bitcoin Explorer" (bx)) facilitan la escritura de scripts de shell y "pipes" de línea de comandos que manipulan llaves, direcciones y transacciones de bitcoin. Puedes usar Bitcoin Explorer para decodificar el formato Base58Check en la línea de comandos.

Usamos el comando base58check-decode para decodificar la llave sin comprimir:

```
$ bx base58check-decode 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
wrapper
{
    checksum 4286807748
    payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd
    version 128
}
```

El resultado contiene la llave como "payload", 128 como prefijo de versión WIF y un checksum.

Observa que el "payload" de la llave comprimida es anexada con el sufijo 01, lo que indica que la llave pública derivada debe comprimirse:

```
$ bx base58check-decode KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
wrapper
{
    checksum 2339607926
    payload 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01
    version 128
}
```

Codificar de hexadecimal a Base58Check

Para codificar a Base58Check (lo opuesto al comando anterior), usamos el comando base58check-encode de Bitcoin Explorer (ver Comandos del "Bitcoin Explorer" (bx)) y proporcionamos la llave privada hexadecimal, seguida por el prefijo de versión 128 de WIF:

```
bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd --version 128
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
```

Codificar desde hexadecimal (llave comprimida) a Base58Check

Para codificar a Base58Check como una llave privada "comprimida" (ver Llaves privadas comprimidas), agregamos el sufijo 01 a la llave hexadecimal y luego codificamos como en la sección anterior:

```
$ bx base58check-encode 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 --version 128
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

El formato resultante WIF-comprimido comienza con una "K". Esto denota que la llave privada en su interior tiene un sufijo "01" y se usará para producir solo llaves públicas comprimidas (ver Llaves públicas comprimidas).

Formatos de llaves públicas

Las llaves públicas también se presentan de diferentes maneras, generalmente como llaves públicas *comprimidas* o *no comprimidas*.

Como vimos anteriormente, la llave pública es un punto en la curva elíptica que consiste en un par de coordenadas (x,y). Generalmente se presenta con el prefijo 04 seguido de dos números de 256 bits: uno para la coordenada *x* del punto, el otro para la coordenada *y*. El prefijo 04 se usa para distinguir las llaves públicas no comprimidas de las llaves públicas comprimidas que comienzan con un 02 o un 03.

Aquí hay una llave pública generada por la llave privada que creamos previamente, mostrada como las coordenadas x e y.

```
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Aquí está la misma llave pública mostrada como un número de 520 bits (130 dígitos hexadecimales) con el prefijo 04 seguido por las coordenadas x y luego y, como 04 x y:

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A↵
07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Llaves públicas comprimidas

Las llaves públicas comprimidas se introdujeron en bitcoin para reducir el tamaño de las transacciones y ahorrar espacio en disco en los nodos que almacenan la base de datos de la cadena de bloques de bitcoin. La mayoría de las transacciones incluyen la llave pública, que se requiere para validar las credenciales del propietario y gastar el bitcoin. Cada llave pública requiere 520 bits (prefijo + x + y), que cuando se multiplica por varios cientos de transacciones por bloque, o decenas de miles de transacciones por día, suman una cantidad significativa de datos a la cadena de bloques.

Como vimos en la sección Llaves Públicas, una llave pública es un punto (x,y) en una curva elíptica. Ya que la curva expresa una función matemática, un punto en la curva representa una solución a una ecuación, y por ende, si conocemos la coordenada *x* podemos calcular la coordenada *y* resolviendo la ecuación $y^2 \bmod p = (x^3 + 7) \bmod p$. Esto nos permite almacenar solamente la coordenada *x* del punto de llave pública, omitiendo la coordenada *y* y reduciendo el tamaño de la llave y el espacio requerido para almacenarla en 256 bits. ¡Una reducción en tamaño de casi el 50% por transacción

representa muchos datos ahorrados con el transcurrir del tiempo!

Mientras que las llaves públicas no comprimidas llevan el prefijo 04, las llaves públicas comprimidas comienzan con un prefijo 02 o 03. Veamos por qué hay dos prefijos posibles: dado que el lado izquierdo de la ecuación es y^2, la solución para y es una raíz cuadrada, que puede tener un valor positivo o negativo. Visualmente, esto significa que la coordenada y resultante puede estar por encima o por debajo del eje x. Como puedes ver en el gráfico de la curva elíptica en Una curva elíptica, la curva es simétrica, lo que significa que se refleja como un espejo en el eje x. Entonces, mientras podemos omitir la coordenada y tenemos que almacenar el *signo* de y (positivo o negativo); o en otras palabras, debemos recordar si estaba por encima o por debajo del eje x, ya que cada una de esas opciones representa un punto diferente y una llave pública diferente. Al calcular la curva elíptica en aritmética binaria en el campo finito de orden primo p, la coordenada y es par o impar, lo que corresponde al signo positivo/negativo como se explicó anteriormente. Por lo tanto, para distinguir entre los dos valores posibles de y, almacenamos una llave pública comprimida con el prefijo 02 si y es par, y 03 si es impar, lo que permite que el software deduzca correctamente la coordenada y a partir de la coordenada x y descomprima la llave pública a las coordenadas completas del punto. La compresión de la llave pública se ilustra en Compresión de llave pública.

Aquí está la misma llave pública generada anteriormente, mostrada como una llave pública comprimida almacenada en 264 bits (66 dígitos hexadecimales) con el prefijo 03 que indica que la coordenada y es impar:

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

Esta llave pública comprimida corresponde a la misma llave privada, lo que significa que se genera a partir de la misma llave privada. Sin embargo, se ve diferente de la llave pública no comprimida. Más importante aún, si convertimos esta llave pública comprimida en una dirección bitcoin usando la función de doble hash (RIPEMD160(SHA256(K))) producirá una dirección bitcoin *diferente*. Esto puede ser confuso, porque significa que una sola llave privada puede producir una llave pública expresada en dos formatos diferentes (comprimido y no comprimido) que producen dos direcciones bitcoin diferentes. Sin embargo, la llave privada es idéntica para ambas direcciones bitcoin.

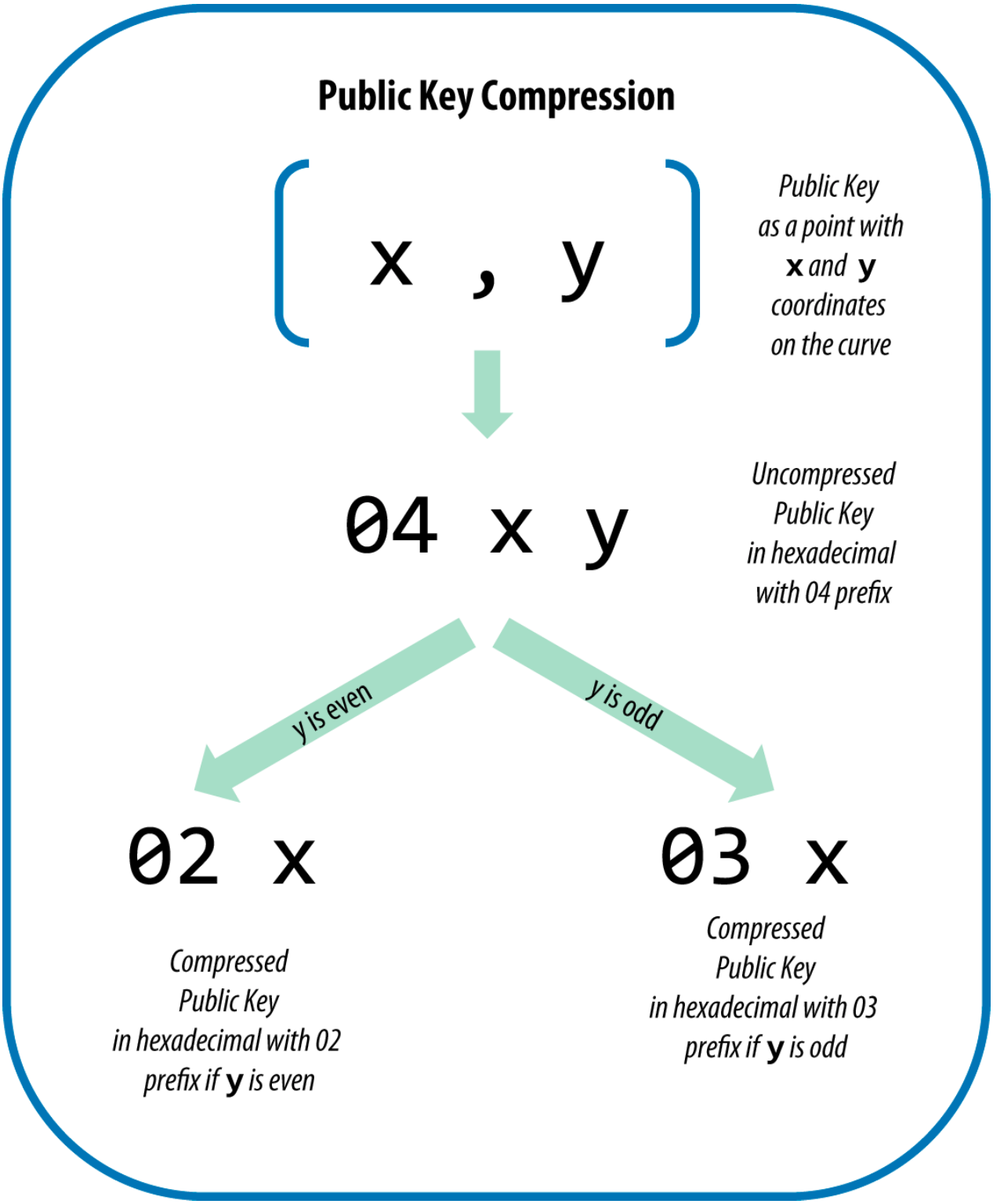


Figure 18. Compresión de llave pública

Las llaves públicas comprimidas se están convirtiendo gradualmente en la opción por defecto en los clientes de bitcoin, lo cual está teniendo un impacto significativo en la reducción del tamaño de las transacciones y, por lo tanto, de la cadena de bloques. Sin embargo, no todos los clientes admiten llaves públicas comprimidas todavía. Los clientes más recientes que admiten llaves públicas comprimidas tienen que tener en cuenta las transacciones de clientes más antiguos que no admiten llaves públicas comprimidas. Esto es especialmente importante cuando una aplicación de cartera está importando llaves privadas de otra aplicación de cartera de bitcoin, porque la nueva cartera necesita escanear la cadena de bloques para encontrar transacciones correspondientes a esas llaves importadas. ¿Qué direcciones bitcoin debe escanear la cartera de bitcoin? ¿Las direcciones bitcoin producidas por llaves públicas no comprimidas, o las direcciones bitcoin producidas por llaves públicas comprimidas? Ambas son direcciones bitcoin válidas, y pueden ser firmadas por la llave privada, ¡pero son direcciones diferentes!

Para resolver este problema, cuando las llaves privadas se exportan desde una cartera, el WIF que se usa para representarlas se implementa de manera diferente en las carteras de bitcoin más nuevas, para indicar que estas llaves

privadas se han utilizado para producir llaves públicas *comprimidas* y por lo tanto direcciones bitcoin *comprimidas*. Esto permite que la cartera a la que se importa distinga entre llaves privadas que se originan en carteras más antiguas o más nuevas y busca en la cadena de bloques las transacciones con direcciones bitcoin correspondientes a las llaves públicas no comprimidas o comprimidas, respectivamente. Veamos cómo funciona esto con más detalle, en la siguiente sección.

Llaves privadas comprimidas

Irónicamente, el término "llave privada comprimida" es engañoso, porque cuando una llave privada se exporta como WIF-comprimido, en realidad es un byte *más largo* que una llave privada "no comprimida". Esto se debe a que a la llave privada se añade un sufijo de un byte (mostrado como 01 en hexadecimal en Ejemplo: Misma llave, formatos distintos), lo que significa que la llave privada proviene de una cartera más nueva y solo debe usarse para producir llaves públicas comprimidas. Las llaves privadas no están comprimidas y no pueden comprimirse. El término "llave privada comprimida" realmente significa "llave privada de la que solo deberían derivarse llaves públicas comprimidas", mientras que "llave privada no comprimida" realmente significa "llave privada de la que solo deberían derivarse llaves públicas no comprimidas". Solo debe referirse al formato de exportación como "WIF-comprimido" o "WIF" y no referirse a la llave privada como "comprimida" para evitar una mayor confusión.

Ejemplo: Misma llave, formatos distintos muestra la misma llave, codificada en formatos WIF y WIF-comprimido.

Table 4. Ejemplo: Misma llave, formatos distintos

Formato	**Llave privada**
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-comprimido	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01
WIF comprimido	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Observa que el formato de llave privada comprimida hexadecimal tiene un byte adicional al final (01 en hex). Si bien el prefijo de versión de codificación Base58 es el mismo (0x80) para los formatos WIF y WIF-comprimido, la adición de un byte al final del número hace que el primer carácter de la codificación Base58 cambie de 5 a una *K* o *L*. Piensa en esto como el equivalente Base58 en la diferencia de codificación decimal entre el número 100 y el número 99. Mientras que 100 es un dígito más largo que 99, también tiene un prefijo 1 en lugar de un prefijo 9. Cuando la longitud cambia, afecta al prefijo. En Base58, el prefijo 5 cambia a *K* o *L* cuando la longitud del número aumenta en un byte.

Recuerda, estos formatos *no* se usan indistintamente. En una cartera más nueva que implementa llaves públicas comprimidas, las llaves privadas solo se exportarán como WIF-comprimido (con el prefijo *K* o *L*). Si la cartera es una implementación más antigua y no utiliza llaves públicas comprimidas, las llaves privadas solo se exportarán como WIF (con un prefijo 5). El objetivo aquí es señalar a la cartera que importa estas llaves privadas si debe buscar en la cadena de bloques las direcciones y llaves públicas comprimidas o no comprimidas.

Si una cartera bitcoin es capaz de implementar llaves públicas comprimidas, las usará en todas las transacciones. Las llaves privadas en la cartera se utilizarán para derivar los puntos de llave pública en la curva, los cuales serán comprimidos. Las llaves públicas comprimidas se usarán para producir direcciones bitcoin y aquellas se utilizarán en las transacciones. Al exportar llaves privadas de una nueva cartera que implementa llaves públicas comprimidas, se modifica el WIF, con la adición de un sufijo de un byte 01 a la llave privada. La llave privada codificada en Base58Check resultante se denomina "WIF-comprimido" y comienza con la letra *K* o *L*, en lugar de comenzar con "5" como es el caso de las llaves codificadas en WIF (no comprimidas) de las carteras más antiguas.

TIP

¡"Llaves privadas comprimidas" es un nombre poco apropiado! No están comprimidas; más bien, WIF-comprimido significa que las llaves solo deben usarse para derivar llaves públicas comprimidas y sus correspondientes direcciones bitcoin. Irónicamente, una llave privada codificada "WIF-comprimido" es un byte más larga porque tiene el sufijo 01 agregado para distinguirla de una "no comprimida".

Implementando Llaves y Direcciones en C++

Veamos el proceso completo de creación de una dirección bitcoin, partiendo de una llave privada a una llave pública (un punto en la curva elíptica), a una dirección de doble hash y, por último, a la codificación Base58Check. El código C++ en Creando una dirección bitcoin codificada en Base58Check a partir de una llave privada muestra el proceso completo paso

a paso, desde la llave privada hasta la dirección bitcoin codificada en Base58Check. El código de ejemplo utiliza la biblioteca libbitcoin presentada en Clientes Alternativos, Bibliotecas y Kits de Herramientas para algunas funciones de ayuda.

Example 10. Creando una dirección bitcoin codificada en Base58Check a partir de una llave privada

```
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Private secret key string as base16
    bc::ec_secret decoded;
    bc::decode_base16(decoded,
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");

    bc::wallet::ec_private secret(
        decoded, bc::wallet::ec_private::mainnet_p2kh);

    // Get public key.
    bc::wallet::ec_public public_key(secret);
    std::cout << "Public key: " << public_key.encoded() << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //    bc::wallet::payment_address payaddr =
    //        public_key.to_payment_address(
    //            bc::wallet::ec_public::mainnet_p2kh);
    //  const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    bc::data_chunk public_key_data;
    public_key.to_data(public_key_data);
    const auto hash = bc::bitcoin_short_hash(public_key_data);

    bc::data_chunk unencoded_address;
    // Reserve 25 bytes
    //   [ version:1  ]
    //   [ hash:20    ]
    //   [ checksum:4 ]
    unencoded_address.reserve(25);
    // Version byte, 0 is normal BTC address (P2PKH).
    unencoded_address.push_back(0);
    // Hash data
    bc::extend_data(unencoded_address, hash);
    // Checksum is computed by hashing data, and adding 4 bytes from hash.
    bc::append_checksum(unencoded_address);
    // Finally we must encode the result in Bitcoin's base58 encoding.
    assert(unencoded_address.size() == 25);
    const std::string address = bc::encode_base58(unencoded_address);

    std::cout << "Address: " << address << std::endl;
    return 0;
}
```

El código utiliza una llave privada predefinida para producir la misma dirección bitcoin cada vez que se ejecuta, como se muestra en Compilando y ejecutando el código addr.

Example 11. Compilando y ejecutando el código addr

```
# Compilando el código addr.cpp
$ g++ -o addr addr.cpp -std=c++11 $(pkg-config --cflags --libs libbitcoin)
# Correr el ejecutable addr
$ ./addr
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK
```

TIP

El código en Compilando y ejecutando el código addr produce una dirección bitcoin (1PRTT...) a partir de una llave pública *comprimida* (ver Llaves públicas comprimidas). Si utilizaras la llave pública no comprimida, producirías una dirección bitcoin diferente (14K1y...).

Implementando Llaves y Direcciones en Python

La biblioteca de bitcoin más completa en Python es pybitcointools por Vitalik Buterin. En Generación y formato de llaves y direcciones con la biblioteca pybitcointools, usamos la biblioteca pybitcointools (importada como "bitcoin") para generar y mostrar llaves y direcciones en varios formatos.

Example 12. Generación y formato de llaves y direcciones con la biblioteca pybitcointools

```
from __future__ import print_function
import bitcoin

# Generate a random private key
valid_private_key = False
while not valid_private_key:
    private_key = bitcoin.random_key()
    decoded_private_key = bitcoin.decode_privkey(private_key, 'hex')
    valid_private_key = 0 < decoded_private_key < bitcoin.N

print("Private Key (hex) is: ", private_key)
print("Private Key (decimal) is: ", decoded_private_key)

# Convert private key to WIF format
wif_encoded_private_key = bitcoin.encode_privkey(decoded_private_key, 'wif')
print("Private Key (WIF) is: ", wif_encoded_private_key)

# Add suffix "01" to indicate a compressed private key
compressed_private_key = private_key + '01'
print("Private Key Compressed (hex) is: ", compressed_private_key)

# Generate a WIF format from the compressed private key (WIF-compressed)
wif_compressed_private_key = bitcoin.encode_privkey(
    bitcoin.decode_privkey(compressed_private_key, 'hex'), 'wif')
print("Private Key (WIF-Compressed) is: ", wif_compressed_private_key)

# Multiply the EC generator point G with the private key to get a public key point
public_key = bitcoin.fast_multiply(bitcoin.G, decoded_private_key)
print("Public Key (x,y) coordinates is:", public_key)

# Encode as hex, prefix 04
hex_encoded_public_key = bitcoin.encode_pubkey(public_key, 'hex')
print("Public Key (hex) is:", hex_encoded_public_key)

# Compress public key, adjust prefix depending on whether y is even or odd
(public_key_x, public_key_y) = public_key
compressed_prefix = '02' if (public_key_y % 2) == 0 else '03'
hex_compressed_public_key = compressed_prefix + (bitcoin.encode(public_key_x, 16).zfill(64))
print("Compressed Public Key (hex) is:", hex_compressed_public_key)

# Generate bitcoin address from public key
print("Bitcoin Address (b58check) is:", bitcoin.pubkey_to_address(public_key))

# Generate compressed bitcoin address from compressed public key
print("Compressed Bitcoin Address (b58check) is:",
      bitcoin.pubkey_to_address(hex_compressed_public_key))
```

Ejecutando key-to-address-ecc-example.py muestra la salida de ejecutar este código.

Example 13. Ejecutando key-to-address-ecc-example.py

```
$ python key-to-address-ecc-example.py
Private Key (hex) is:
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6
Private Key (decimal) is:
 26563230048437957592232553826663696440606756685920117476832299673293013768870
Private Key (WIF) is:
 5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K
Private Key Compressed (hex) is:
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601
Private Key (WIF-Compressed) is:
 KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S
Public Key (x,y) coordinates is:
 (41637322786646325214887832269588396900663353932545912953362782457239403430124L,
 16388935128781238405526710466724741593761085120864331449066658622400339362166L)
Public Key (hex) is:
 045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec↵
```

```
243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176
Compressed Public Key (hex) is:
 025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec
Bitcoin Address (b58check) is:
 1thMirt546nngXqyPEz532S8fLwbozud8
Dirección Bitcoin Comprimida (b58check) es:
 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3
```

Un script mostrando la matemática de curva elíptica usada para llaves bitcoin es otro ejemplo, usando la biblioteca ECDSA de Python para las matemáticas de curva elíptica y sin usar ninguna biblioteca de bitcoin especializada.

Example 14. Un script mostrando la matemática de curva elíptica usada para llaves bitcoin

```
import ecdsa
import os

# secp256k1, http://www.oid-info.com/get/1.3.132.0.10
_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2F
_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141
_b = 0x0000000000000000000000000000000000000000000000000000000000000007
_a = 0x0000000000000000000000000000000000000000000000000000000000000000
_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798
_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
curve_secp256k1 = ecdsa.ellipticcurve.CurveFp(_p, _a, _b)
generator_secp256k1 = ecdsa.ellipticcurve.Point(curve_secp256k1, _Gx, _Gy, _r)
oid_secp256k1 = (1, 3, 132, 0, 10)
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve_secp256k1,
                               generator_secp256k1, oid_secp256k1)
ec_order = _r

curve = curve_secp256k1
generator = generator_secp256k1


def random_secret():
    # Collect 256 bits of random data from the OS's cryptographically secure
    # random number generator
    byte_array = (os.urandom(32)).hex()

    return int(byte_array,16)

def get_point_pubkey(point):
    if (point.y() % 2) == 1:
        key = '03' + '%064x' % point.x()
    else:
        key = '02' + '%064x' % point.x()
    return key


def get_point_pubkey_uncompressed(point):
    key = ('04' +
           '%064x' % point.x() +
           '%064x' % point.y())
    return key

# Generate a new private key.
secret = random_secret()
print("Secret: ", secret)

# Get the public key point.
point = secret * generator
print("Elliptic Curve point:", point)

print("BTC public key:", get_point_pubkey(point))

# Given the point (x, y) we can create the object using:
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec_order)
assert(point1 == point)
```

Instalando la biblioteca Python ECDSA y ejecutando el script ec_math.py muestra la salida producida al ejecutar este script.

WARNING Un script mostrando la matemática de curva elíptica usada para llaves bitcoin usa os.urandom, que muestra un generador de números aleatorios criptográficamente seguro (CSRNG) proporcionado por el

> sistema operativo subyacente. Precaución: Dependiendo del sistema operativo, os.urandom puede que *no* se implemente con suficiente seguridad o se inicialice correctamente y puede *no* ser apropiado para generar llaves de bitcoin con calidad de producción.

Example 15. Instalando la biblioteca Python ECDSA y ejecutando el script ec_math.py

```
$ # Instalar el administrador de paquetes Python PIP
$ sudo apt-get install python-pip
$ # Instalar la biblioteca Python ECDSA
$ sudo pip install ecdsa
$ # Ejecutar el script
$ python ec-math.py
Secret:  38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point: (700488535318671794898577504976069662723825834713229354546245955400072693126 27,
1052622064786867431910608002634795893299202095272858039357360216860455423533 80)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873
```

Llaves y Direcciones Avanzadas

En las siguientes secciones veremos formas avanzadas de llaves y direcciones, tales como llaves privadas encriptadas, scripts y direcciones multifirma, direcciones de vanidad y carteras de papel.

Pagar-al-Hash-de-un-Script (P2SH) y Direcciones Multi-firma

Como sabemos, las direcciones bitcoin tradicionales comienzan con el número “1” y se derivan de la llave pública, que se deriva de la llave privada. Aunque cualquiera puede enviar bitcoin a una dirección "1", ese bitcoin solo se puede gastar presentando la correspondiente firma de llave privada y el hash de llave pública.

Las direcciones bitcoin que comienzan con el número “3” son direcciones pago-a-script-hash (P2SH, pay-to-script-hash), a veces denominadas erróneamente direcciones multifirma. Designan al beneficiario de una transacción bitcoin como el hash de un script, en lugar del propietario de una llave pública. La función se introdujo en enero de 2012 con BIP-16 (consulte Propuestas de Mejora para Bitcoin) y está siendo ampliamente adoptado porque proporciona la oportunidad de agregar funcionalidad a la dirección en sí. A diferencia de las transacciones que "envían" fondos a direcciones bitcoin "1" tradicionales, también conocidas como pago-a-llave-pública-hash (P2PKH, pay-to-public-key-hash), los fondos enviados a direcciones "3" requieren algo más que la presentación de un hash de llave pública y una firma de llave privada como prueba de propiedad. Los requisitos se designan en el momento en que se crea la dirección, dentro del script, y todas las entradas a esta dirección serán bloqueadas con los mismos requisitos.

Una dirección P2SH se crea a partir de un script de transacción, que define quién puede gastar una salida de transacción (para más detalles, ver Pay-to-Script-Hash (P2SH)). La codificación de una dirección P2SH implica usar la misma función de doble-hash que se utilizó durante la creación de una dirección bitcoin, solo que se aplica al script en lugar de a la llave pública:

```
hash de script = RIPEMD160(SHA256(script))
```

El "hash del script" resultante está codificado con Base58Check con un prefijo de versión de valor 5, lo que resulta en una dirección codificada que comienza con un 3. Un ejemplo de una dirección P2SH es 3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM, que se puede derivar mediante los comandos de Bitcoin Explorer, script-encode, sha256, ripemd160, y base58check-encode (ver Comandos del "Bitcoin Explorer" (bx)) como sigue:

```
$ echo \
'DUP HASH160 [89abcdefabbaabbaabbaabbaabbaabbaabbaabba] EQUALVERIFY CHECKSIG' > script
$ bx script-encode < script | bx sha256 | bx ripemd160 \
| bx base58check-encode --version 5
3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM
```

> **TIP** P2SH no es necesariamente lo mismo que una transacción multifirma estándar. Una dirección P2SH representa *la mayor parte de las veces* un script multi-firma, pero también puede representar un script que codifique otros tipos de transacciones.

Direcciones multifirma y P2SH

Actualmente, la implementación más común de la función P2SH es el script de dirección multi-firma. Como su nombre

indica, el script subyacente requiere más de una firma para demostrar la propiedad y, por lo tanto, gastar fondos. La función de multi-firma de bitcoin está diseñada para requerir M firmas (también conocido como el "umbral") de un total de N llaves, conocido como un multi-firma de M-de-N, donde M es igual o menor que N. Por ejemplo , Bob, el propietario de la cafetería de Introducción podría usar una dirección de multi-firma que requiera de 1-de-2 firmas, una de las llaves de su propiedad y la otra llave perteneciente a su cónyuge, lo que garantiza que cualquiera de los dos pueda firmar para gastar una salida de transacción que se encuentre bloqueada en esta dirección. Esto sería similar a una "cuenta conjunta" tal como se aplica en la banca tradicional, donde cualquiera de los cónyuges puede gastar con una sola firma. O Gopesh, ("casos de uso", "servicios contractuales en el extranjero" el diseñador web pagado por Bob para crear un sitio web, podría tener una dirección multi-firma 2-de-3 para su negocio que garantice que no se pueden gastar los fondos a menos que dos de los socios firmen la transacción.

Exploraremos cómo crear transacciones que gastan fondos de direcciones P2SH (y multifirma) en Transacciones.

Direcciones de Vanidad

Las direcciones de vanidad son direcciones de bitcoin válidas que contienen mensajes legibles. Por ejemplo, 1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33 es una dirección válida que contiene las letras que forman la palabra "Love" como las primeras cuatro letras en Base-58. Las direcciones de vanidad requieren generar y probar miles de millones de llaves privadas candidatas, hasta que se encuentre una dirección bitcoin con el patrón deseado. Aunque hay algunas optimizaciones en el algoritmo de generación de vanidad, el proceso esencialmente consiste en seleccionar una llave privada al azar, derivar la llave pública, derivar la dirección bitcoin y verificar si coincide con el patrón de vanidad deseado, repitiendo miles de millones de veces hasta que se encuentre coincidencia.

Una vez que se encuentra una dirección de vanidad que coincida con el patrón deseado, la llave privada de la que se deriva puede ser utilizada por el propietario para gastar bitcoin exactamente de la misma manera que cualquier otra dirección. Las direcciones de vanidad no son menos o más seguras que cualquier otra dirección. Dependen de la misma criptografía de curva elíptica (ECC) y de SHA como cualquier otra dirección. No es más fácil encontrar la llave privada de una dirección que comienza con un patrón de vanidad de lo que sería cualquier otra dirección.

En Introducción, presentamos a Eugenia, directora de caridad para niños que opera en Filipinas. Digamos que Eugenia está organizando una campaña de recaudación de fondos en bitcoin y quiere usar una dirección bitcoin de vanidad para dar a conocer la recaudación de fondos. Eugenia creará una dirección de vanidad que comienza con "1Kids" para promover la recaudación de fondos de caridad para niños. Vamos a ver cómo se crea esta dirección de vanidad y qué significa para la seguridad de la organización benéfica de Eugenia.

Generando direcciones de vanidad

Es importante darse cuenta de que una dirección bitcoin es simplemente un número representado por símbolos en el alfabeto Base58. La búsqueda de un patrón como "1Kids" se puede ver como la búsqueda de una dirección en el rango de 1Kids11111111111111111111111111111 a 1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz. Hay aproximadamente 58^{29} (aproximadamente $1.4 * 10^{51}$) direcciones en ese rango, todas comenzando con "1Kids". El rango de direcciones de vanidad comenzando con "1Kids" muestra el rango de direcciones que tienen el prefijo 1Kids.

Table 5. El rango de direcciones de vanidad comenzando con "1Kids"

Desde	1Kids11111111111111111111111111
	1Kids11111111111111111111111112
	1Kids11111111111111111111111113
	...
Hasta	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzz

Veamos el patrón "1Kids" como un número y la frecuencia con la que podríamos encontrar este patrón en una dirección bitcoin (ver La frecuencia de un patrón de vanidad (1KidsCharity) y el tiempo de búsqueda promedio en una PC de escritorio). De promedio, un ordenador de sobremesa sin ningún hardware especializado puede buscar aproximadamente 100.000 llaves por segundo.

Table 6. La frecuencia de un patrón de vanidad (1KidsCharity) y el tiempo de búsqueda promedio en una PC de escritorio

Longitud	Patrón	Frecuencia	Tiempo de búsqueda promedio
1	1K	1 en 58 llaves	< 1 milisegundos
2	1Ki	1 en 3.364	50 milisegundos
3	1Kid	1 en 195.000	< 2 segundos
4	1Kids	1 en 11 millones	1 minuto
5	1KidsC	1 en 656 milliones	1 hora
6	1KidsCh	1 en 38 mil millones	2 días
7	1KidsCha	1 en 2.2 billones	3–4 meses
8	1KidsChar	1 en 128 billones	13–18 años
9	1KidsChari	1 en 7 mil billones	800 años
10	1KidsCharit	1 en 400 mil billones	46.000 años
11	1KidsCharity	1 en 23 trillones	2,5 millones de años

Como se puede ver, Eugenia no podrá crear la dirección de vanidad "1KidsCharity" en el corto plazo, incluso si tuviera acceso a varios miles de computadoras. Cada carácter adicional aumenta la dificultad en un factor de 58. Los patrones con más de siete caracteres se encuentran generalmente con hardware especializado, tales como escritorios hechos a medida con múltiples GPU. Estos son a menudo "plataformas" de minería de bitcoin reutilizadas que ya no son rentables para la minería de bitcoin pero que se pueden usar para encontrar direcciones de vanidad. Las búsquedas por vanidad en los sistemas de GPU son varios órdenes de magnitud más rápidas que las de una CPU de propósito general.

Otra forma de encontrar una dirección de vanidad es subcontratar el trabajo a un grupo de mineros de vanidad, como en el pool Vanity Pool. Un pool es un servicio que permite ganar bitcoin a las personas con hardware de GPU buscando direcciones de vanidad para otros. Por un pequeño pago (0.01 bitcoin o aproximadamente $5 al momento de escribir este artículo), Eugenia puede externalizar la búsqueda de una dirección de vanidad con un patrón de siete caracteres y obtener resultados en unas pocas horas en lugar de tener que realizar una búsqueda de CPU durante meses.

Generar una dirección de vanidad es un ejercicio de fuerza bruta: probar una llave aleatoria, verificar la dirección resultante para ver si coincide con el patrón deseado, repetir hasta tener éxito. Minero de direcciones de vanidad muestra un ejemplo de un "minero de vanidad", un programa diseñado para encontrar direcciones de vanidad, escrito en C++. El ejemplo utiliza la biblioteca libbitcoin, que presentamos en Clientes Alternativos, Bibliotecas y Kits de Herramientas.

Example 16. Minero de direcciones de vanidad

```
#include <random>
#include <bitcoin/bitcoin.hpp>

// The string we are searching for
const std::string search = "1kid";

// Generate a random secret key. A random 32 bytes.
bc::ec_secret random_secret(std::default_random_engine& engine);
// Extract the Bitcoin address from an EC secret.
std::string bitcoin_address(const bc::ec_secret& secret);
// Case insensitive comparison with the search string.
bool match_found(const std::string& address);

int main()
{
    // random_device on Linux uses "/dev/urandom"
    // CAUTION: Depending on implementation this RNG may not be secure enough!
    // Do not use vanity keys generated by this example in production
    std::random_device random;
    std::default_random_engine engine(random());

    // Loop continuously...
    while (true)
```

```
    {
        // Generate a random secret.
        bc::ec_secret secret = random_secret(engine);
        // Get the address.
        std::string address = bitcoin_address(secret);
        // Does it match our search string? (1kid)
        if (match_found(address))
        {
            // Success!
            std::cout << "Found vanity address! " << address << std::endl;
            std::cout << "Secret: " << bc::encode_base16(secret) << std::endl;
            return 0;
        }
    }
    // Should never reach here!
    return 0;
}

bc::ec_secret random_secret(std::default_random_engine& engine)
{
    // Create new secret...
    bc::ec_secret secret;
    // Iterate through every byte setting a random value...
    for (uint8_t& byte: secret)
        byte = engine() & 255;
    // Return result.
    return secret;
}

std::string bitcoin_address(const bc::ec_secret& secret)
{
    // Convert secret to payment address
    bc::wallet::ec_private private_key(secret);
    bc::wallet::payment_address payaddr(private_key);
    // Return encoded form.
    return payaddr.encoded();
}

bool match_found(const std::string& address)
{
    auto addr_it = address.begin();
    // Loop through the search string comparing it to the lower case
    // character of the supplied address.
    for (auto it = search.begin(); it != search.end(); ++it, ++addr_it)
        if (*it != std::tolower(*addr_it))
            return false;
    // Reached end of search string, so address matches.
    return true;
}
```

NOTE

Compilando y ejecutando el ejemplo de vanity-miner utiliza std::random_device. Dependiendo de la implementación, puede reflejar un generador de números aleatorios criptográficamente seguro (CSRNG) proporcionado por el sistema operativo subyacente. En el caso de un sistema operativo similar a Unix como Linux, se basa en /dev/urandom. El generador de números aleatorios que se utiliza aquí es para fines de demostración, y *no* es apropiado para generar llaves de bitcoin de calidad de producción, ya que no está implementado con suficiente seguridad.

El código de ejemplo debe compilarse usando un compilador de C++ y enlazado con la biblioteca libbitcoin (que debe instalarse primero en ese sistema). Para ejecutar el ejemplo, lanza el ejecutable vanity-miner sin parámetros (ver Compilando y ejecutando el ejemplo de vanity-miner) e intentará encontrar una dirección de vanidad que empiece por "1kid".

Example 17. Compilando y ejecutando el ejemplo de vanity-miner

```
$ # Compilar el código con g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Ejecutar el ejemplo
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Ejecutarlo otra vez para obtener un resultado distinto
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
```

```
# Usar "time" para ver cuánto tarda en encontrar un resultado
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

real    0m8.868s
user    0m8.828s
sys     0m0.035s
```

El código de ejemplo tardará unos segundos en encontrar una coincidencia para el patrón de tres caracteres "kid", como podemos ver cuando usamos el comando time de Unix para medir el tiempo de ejecución. Cambia el patrón search en el código fuente y ¡mira cuánto tiempo más se tarda para los patrones de cuatro o cinco caracteres!

Seguridad de direcciones de vanidad

Las direcciones de vanidad se pueden usar para mejorar *y* para anular las medidas de seguridad; son verdaderamente una espada de doble filo. Cuando se usa para mejorar la seguridad, una dirección distintiva hace que sea más difícil para los adversarios sustituirla por tu propia dirección y engañar así a los clientes para que les paguen a ellos en lugar de a ti. Desafortunadamente, las direcciones de vanidad también hacen posible que cualquiera pueda crear una dirección que *se parezca* a cualquier dirección aleatoria, o incluso a otra dirección de vanidad, engañando de esta manera a tus clientes.

Eugenia podría publicitar una dirección generada aleatoriamente (por ejemplo, 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy) a la cual la gente podría enviar sus donaciones. O podría generar una dirección de vanidad que comience con 1Kids para hacerla más distintiva.

En ambos casos, uno de los riesgos del uso de una dirección fija única (en lugar de una dirección dinámica separada por donante) es que un ladrón podría ser capaz de infiltrarse en su sitio web y reemplazarla con su propia dirección, desviando así las donaciones a sí mismo. Si ha anunciado su dirección de donación en diferentes lugares, los usuarios pueden inspeccionar visualmente la dirección antes de hacer un pago para asegurarse de que es la misma que vieron en su sitio web, en su correo electrónico y en su propaganda. En el caso de una dirección aleatoria como 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy, el usuario medio querrá quizá inspeccionar los primeros caracteres "1J7mdg" y estar convencido de que la dirección coinde. Mediante el uso de un generador de direcciones de vanidad, una persona con la intención de robar podría sustituir la dirección original con otra de aspecto similar, generada rápidamente mediante la coincidencia en sus primeros caracteres, como se muestra en Generando direcciones de vanidad para coincidir con una dirección aleatoria.

Table 7. Generando direcciones de vanidad para coincidir con una dirección aleatoria

Dirección Aleatoria Original	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
Vanidad (coincidencia de 4 caracteres)	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
Vanidad (coincidencia de 5 caracteres)	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
Vanidad (coincidencia de 6 caracteres)	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

Entonces, ¿una dirección de vanidad aumenta la seguridad? Si Eugenia genera la dirección de vanidad 1Kids33q44erFfpeXrmDSz7zEqG2FesZEN, es probable que los usuarios vean la palabra del patrón de vanidad *y unos pocos caracteres más allá*, por ejemplo, notando la parte "1Kids33" de la dirección. Eso obligaría a un atacante a generar una dirección de vanidad que coincida con al menos seis caracteres (dos más), gastando un esfuerzo que es 3364 veces (58 × 58) más alto que el esfuerzo que Eugenia gastó por su vanidad de 4 caracteres. Esencialmente, el esfuerzo que Eugenia gastó (o pagó a un pool de vanidad) "empuja" al atacante a tener que producir un patrón de vanidad de mayor longitud. Si Eugenia paga a un pool para generar una dirección de vanidad de 8 caracteres, el atacante sería empujado a buscar una de 10 caracteres, que es inviable en una computadora personal y es costoso incluso con un equipo personalizado para minería de vanidad o con un pool de vanidad. Lo que es asequible para Eugenia se vuelve inasequible para el atacante, especialmente si la posible recompensa del fraude no es lo suficientemente alta como para cubrir el costo de la generación de direcciones de vanidad.

Carteras de Papel

Las carteras de papel son llaves privadas de bitcoin impresas en papel. A menudo, la cartera de papel también incluye la dirección bitcoin correspondiente por conveniencia, pero esto no es necesario porque puede ser derivada a partir de la llave privada. Las carteras de papel son una forma muy efectiva de crear copias de seguridad o almacenamiento de bitcoin sin conexión, también conocido como "almacenamiento en frío". Como mecanismo de respaldo, una cartera de

papel puede proporcionar seguridad contra la pérdida de la llave debido a un percance en la computadora, como un fallo del disco duro, un robo o una eliminación accidental. Como un mecanismo de "almacenamiento en frío", si las llaves de la cartera de papel se generan sin conexión y nunca se almacenan en un sistema informático, son mucho más seguras contra piratas informáticos, keyloggers y otras amenazas informáticas en línea.

Las carteras de papel vienen en muchas formas, tamaños y diseños, pero a un nivel muy básico son simplemente una llave y una dirección impresas en papel. Forma más simple de una cartera de papel: una impresión de la dirección bitcoin y de la llave privada. muestra la forma más sencilla de una cartera de papel.

Table 8. Forma más simple de una cartera de papel: una impresión de la dirección bitcoin y de la llave privada.

Dirección pública	**Llave privada (WIF)**
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

Las carteras de papel se pueden generar fácilmente utilizando una herramienta web JavaScript en *bitaddress.org*. Esta página contiene todo el código necesario para generar llaves y carteras de papel, incluso completamente desconectado de Internet. Para usarlo, guarda la página HTML en la unidad local o en una unidad flash USB externa. Desconéctate de Internet y abre el archivo en un navegador. Aún mejor, arranca el ordenador utilizando un sistema operativo original, como por ejemplo, un CD-ROM de arranque del sistema operativo Linux. Cualquier llave generada con esta herramienta sin conexión se puede imprimir en una impresora local mediante un cable USB (no inalámbrica), creando así carteras de papel cuyas llaves sólo existen en el papel y nunca han sido almacenados en ningún sistema en línea. Para implementar una solución sencilla pero muy eficaz de "almacenamiento en frío", pon esas carteras de papel en una caja fuerte a prueba de fuego y "envía" bitcoin a tu dirección bitcoin. Un ejemplo de una cartera de papel simple de bitaddress.org muestra una cartera de papel generada desde el sitio bitaddress.org.

Figure 19. Un ejemplo de una cartera de papel simple de bitaddress.org

La desventaja del sistema de cartera de papel simple es que las llaves impresas son vulnerables al robo. Un ladrón que es capaz de tener acceso al papel puede robar o fotografiar las llaves y tomar el control de los bitcoin bloqueados con dichas llaves. Un sistema de almacenamiento de la cartera de papel más sofisticado se utiliza en BIP-38 mediante el uso de llaves privadas encriptadas. Las llaves impresas en la cartera de papel están protegidas por una frase de contraseña que el propietario ha memorizado. Sin la frase de contraseña, las llaves encriptadas son inútiles. Sin embargo, todavía son superiores a una cartera protegida con una frase de contraseña porque las llaves nunca han estado en línea y deben ser recuperadas físicamente de un almacenamiento seguro o protegido físicamente. Un ejemplo de una cartera de papel cifrada de bitaddress.org. La frase de contraseña es "test." muestra una cartera de papel con una llave privada encriptada (BIP-38) creada en el sitio bitaddress.org.

Figure 20. Un ejemplo de una cartera de papel cifrada de bitaddress.org. La frase de contraseña es "test."

WARNING

Aunque se puede depositar fondos en una cartera de papel varias veces, se debe retirar todos los fondos solo una vez, gastando todo. Esto se debe a que en el proceso de desbloqueo y gasto de fondos, algunas billeteras pueden generar una dirección de cambio si gasta menos del monto total. Si la computadora que se usa para firmar la transacción se ve comprometida, se corre el riesgo de exponer la llave privada, dando acceso a los fondos en la dirección de cambio. Al gastar el saldo completo de una cartera de papel de una vez, se reduce el riesgo del compromiso de la llave. Si solo necesita una pequeña cantidad, envíe los fondos restantes a una nueva cartera de papel en la misma transacción.

Las carteras de papel vienen en muchos diseños y tamaños, con muchas características diferentes. Algunas están destinadas a ser dadas como regalos y tienen temas estacionales, como la Navidad y temas de Año Nuevo. Otras están diseñadas para el almacenamiento en una bóveda bancaria o caja de seguridad con la llave privada oculta de alguna manera, ya sea con pegatinas-rasca opacas o plegados y sellados con una lámina adhesiva a prueba de manipulaciones. Las imágenes de #paper_wallet_bpw a pass: [<a data-type="xref" href="#paper_wallet_spw" data-xrefstyle="select: labelnumber">#paper_wallet_spw] muestran varios ejemplos de carteras de papel con características de seguridad y de copia de respaldo.

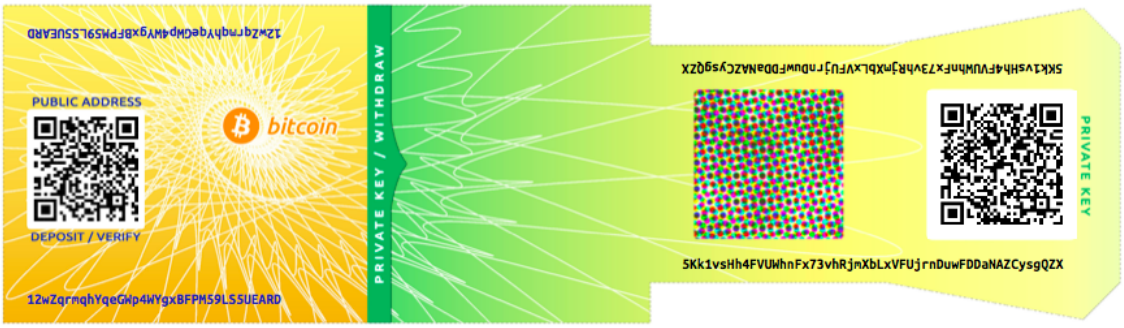


Figure 21. Un ejemplo de una cartera de papel de bitcoinpaperwallet.com con la llave privada en una solapa plegable

Figure 22. La cartera de papel de bitcoinpaperwallet.com con la llave privada oculta

Otros diseños cuentan con copias adicionales de la llave y de la dirección, en forma de fichas separables similares a talones de boletos, lo que le permite almacenar múltiples copias para protegerse contra incendios, inundaciones u otros desastres naturales.

Figure 23. Un ejemplo de una cartera de papel con copias adicionales de las llaves en un "talón" de respaldo

Carteras

La palabra "cartera" se usa para describir cosas diferentes en bitcoin.

De modo general, una cartera es una aplicación que sirve como la interfaz principal de usuario. La cartera controla el acceso al dinero de un usuario, la administración de llaves y direcciones, el seguimiento del saldo y la creación y firma de transacciones.

Más estrictamente, desde la perspectiva de un programador, la palabra "cartera" se refiere a la estructura de datos que se utiliza para almacenar y administrar las llaves de un usuario.

En este capítulo veremos el segundo significado, donde las carteras son contenedores para llaves privadas, generalmente implementadas como archivos estructurados o bases de datos simples.

Resumen de la Tecnología de Carteras

En esta sección resumimos las diversas tecnologías utilizadas para construir carteras bitcoin fáciles de usar, seguras y flexibles.

Un error común acerca de bitcoin es considerar que las carteras bitcoin contienen bitcoin. De hecho, la cartera solo contiene llaves. Las "monedas" se registran en la cadena de bloques en la red bitcoin. Los usuarios controlan las monedas en la red firmando transacciones con las llaves en sus carteras. En cierto sentido, una cartera bitcoin es un *llavero* (en inglés, keychain).

TIP

Las carteras bitcoin contienen llaves, no monedas. Cada usuario posee una cartera conteniendo llaves. Las carteras son en esencia llaveros que contienen pares de llaves privadas/públicas (ver Llaves Privadas y Públicas). Los usuarios firman transacciones con las llaves, demostrando de esa forma que son dueños de las salidas de transacción (sus monedas). Las monedas se almacenan en la cadena de bloques en forma de salidas de transacción (a menudo notadas como vout o txout).

Existen dos tipos principales de carteras, que se distinguen según si las llaves que contienen están relacionadas entre sí o no.

El primer tipo es una cartera no determinista, donde cada llave se genera de forma independiente a partir de un número aleatorio. Las llaves no están relacionadas entre sí. Este tipo de cartera también se conoce como una cartera JBOK, de la frase en inglés "Just a Bunch Of Keys" ("Solo Unas Cuantas Llaves") .

El segundo tipo de cartera es una *cartera determinista*, donde todas las llaves se derivan de una llave maestra única, conocida como *semilla* (en inglés, seed). Todas las llaves en este tipo de cartera están relacionadas entre sí y se pueden generar nuevamente si tienes la semilla original. Existen varios métodos diferentes de *derivación de llaves* que se utilizan en carteras deterministas. El método de derivación usado más habitualmente utiliza una estructura similar a un árbol y se conoce como una cartera *determinista jerárquica* o cartera *HD* (del inglés, Hierarchical Deterministic).

Las carteras deterministas se inicializan a partir de una semilla. Para que sean más fáciles de usar, las semillas se codifican como palabras en inglés, también conocidas como *palabras código mnemónicas*.

Las siguientes secciones introducen cada una de estas tecnologías a nivel general.

Carteras No Deterministas (Aleatorias)

En la primera cartera de bitcoin (ahora llamada Bitcoin Core), las carteras eran colecciones de llaves privadas generadas aleatoriamente. Por ejemplo, el cliente principal, Bitcoin Core, genera 100 llaves privadas aleatorias cuando se inicia por primera vez y genera más llaves según sea necesario, utilizando cada llave solo una vez. Dichas carteras se están reemplazando por carteras deterministas porque son complicadas de administrar, realizar copias de seguridad e importar. La desventaja de las llaves aleatorias es que si generas muchas de ellas debes guardar copias de todas ellas, lo que significa que se debe hacer una copia de seguridad de la cartera con frecuencia. Se debe hacer una copia de seguridad de cada llave, o los fondos que controla se perderán irrevocablemente si la cartera se vuelve inaccesible. Esto entra en conflicto directamente con el principio de evitar la reutilización de direcciones, al usar cada dirección bitcoin para una sola transacción. La reutilización de direcciones reduce la privacidad al asociar varias transacciones y direcciones entre sí. Una cartera no determinista de Tipo-0 es una mala elección de cartera, especialmente si deseas evitar la reutilización

de direcciones porque significa administrar muchas llaves, lo que crea la necesidad de copias de seguridad frecuentes. Aunque el Cliente Principal de Bitcoin incluye una cartera de Tipo-0, los desarrolladores de Bitcoin Core desaconsejan el uso de esta cartera. Cartera no determinista (aleatoria) Tipo-0: una colección de llaves generadas aleatoriamente muestra una cartera no determinista, que contiene un conjunto disperso de llaves aleatorias.

TIP

Se desaconseja el uso de carteras no deterministas para cualquier otra cosa que no sean simples pruebas. Son simplemente demasiado engorrosas para hacer copias de seguridad y para usarlas. En su lugar, úsese una *cartera HD* basada en el estándar de la industria–con una semilla *mnemónica* como copia de seguridad.

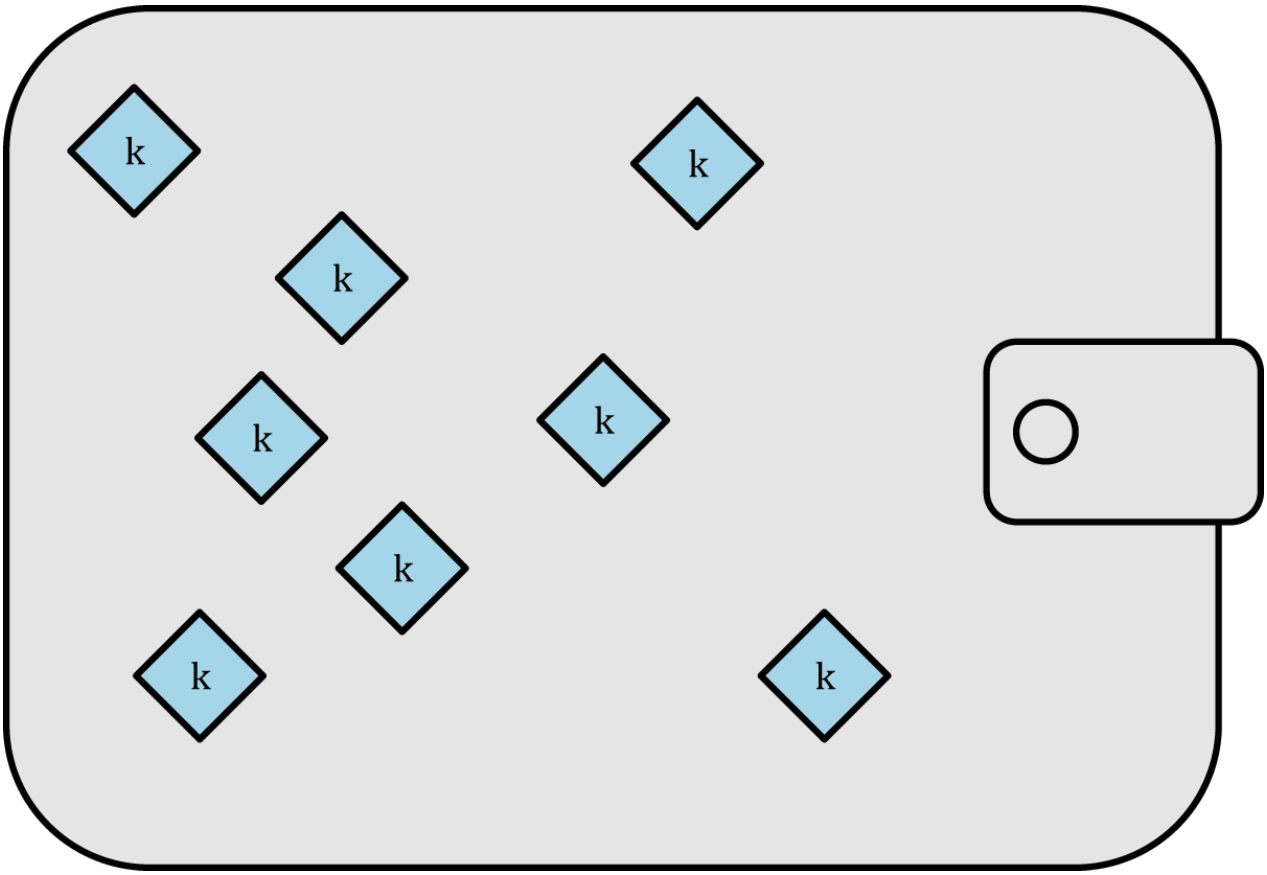


Figure 24. Cartera no determinista (aleatoria) Tipo-0: una colección de llaves generadas aleatoriamente

Carteras Deterministas (A Partir de Semilla)

Las carteras deterministas o "con semilla" son carteras que contienen llaves privadas que surgen a partir de una semilla común, mediante el uso de una función hash unidireccional. La semilla es un número generado aleatoriamente que se combina con otros datos, como un número de índice o "código de cadena" (ver Carteras HD (BIP-32/BIP-44)) para derivar las llaves privadas. En una cartera determinista, la semilla es suficiente para recuperar todas las llaves derivadas, y por lo tanto una única copia de seguridad en el momento de la creación es suficiente. La semilla también es suficiente para una exportación de cartera o de importación, lo que permite una fácil migración de todas las llaves de los usuarios entre diferentes implementaciones de cartera. Cartera determinista (con semilla) Tipo-1: una secuencia determinista de llaves derivada de una semilla muestra un diagrama lógico de una cartera determinista.

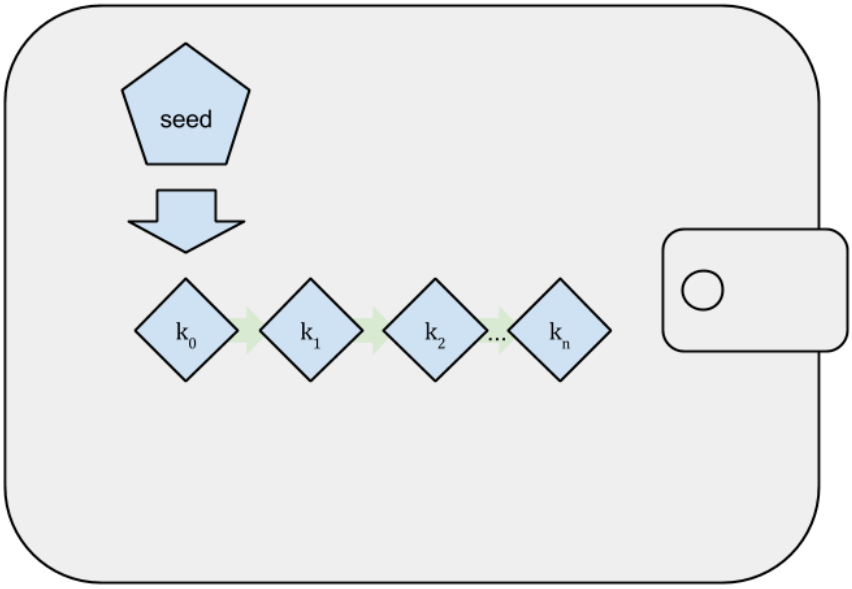


Figure 25. Cartera determinista (con semilla) Tipo-1: una secuencia determinista de llaves derivada de una semilla

Carteras HD (BIP-32/BIP-44)

Las carteras deterministas se desarrollaron para facilitar la obtención de muchas llaves a partir de una sola "semilla". La forma más avanzada de carteras deterministas es la cartera HD definida por el estándar BIP-32. Las carteras HD contienen llaves derivadas en una estructura de árbol, de modo que una llave principal puede derivar una secuencia de llaves secundarias, cada una de las cuales puede derivar una secuencia de llaves de nietos, y así sucesivamente, a una profundidad infinita. Esta estructura de árbol se ilustra en Cartera HD de Tipo-2: un árbol de llaves generado a partir de una única semilla.

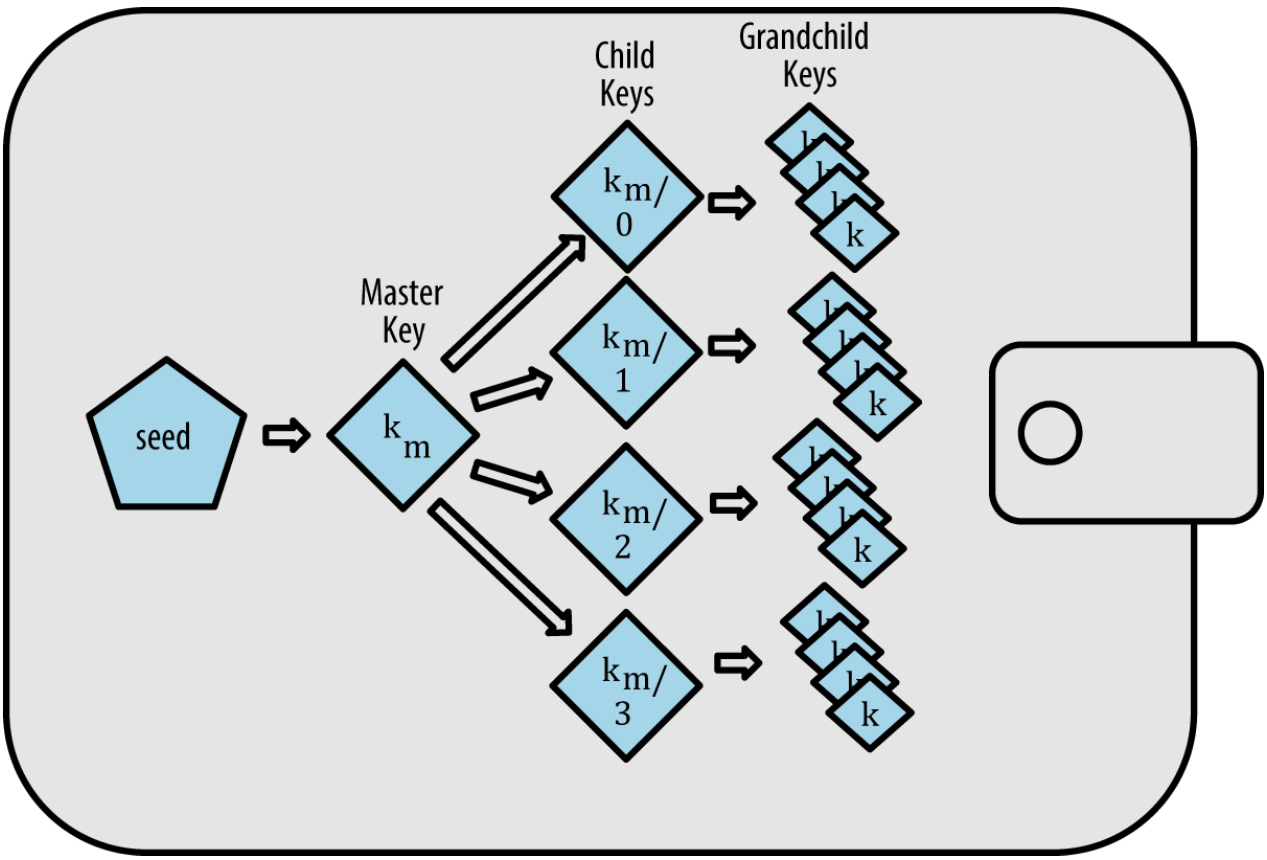


Figure 26. Cartera HD de Tipo-2: un árbol de llaves generado a partir de una única semilla

Las carteras HD ofrecen dos grandes ventajas sobre las llaves aleatorias (no deterministas). En primer lugar, la estructura de árbol se puede utilizar para expresar un significado organizativo adicional, tal como cuando se utiliza una rama específica de sub-llaves para recibir los pagos entrantes y una rama diferente se utiliza para recibir el cambio de los pagos salientes. Las ramas de llaves también se pueden utilizar en un entorno corporativo, asignando diferentes ramas a los

departamentos, filiales, funciones específicas o categorías de contabilidad.

La segunda ventaja de las carteras HD es que los usuarios pueden crear secuencias de llaves públicas sin tener acceso a las llaves privadas correspondientes. Esto permite a las carteras HD ser usadas en servidores inseguros o en capacidad de receptión de fondos únicamente, generando una llave pública distinta para cada transacción. Las llaves públicas no necesitan ser pre-cargadas ni derivadas por adelantado, y aun así el servidor no tiene las llaves privadas que permiten gastar los fondos.

Semillas y Códigos Mnemónicos (BIP-39)

Las carteras HD son un mecanismo muy poderoso para administrar muchas llaves y direcciones. Son aún más útiles si se combinan con una forma estandarizada de crear semillas a partir de una secuencia de palabras en inglés que son fáciles de transcribir, exportar e importar en carteras. Esto se conoce como *mnemónico* y el estándar está definido por BIP-39. Hoy en día, la mayoría de las carteras bitcoin (así como las carteras para otras criptomonedas) utilizan este estándar y pueden importar y exportar semillas para copias de seguridad y recuperación utilizando mnemotécnicos interoperables.

Veamos esto desde una perspectiva práctica. ¿Cuál de las siguientes semillas es más fácil de transcribir, grabar en papel, leer sin error, exportar e importar en otra cartera?

Una semilla para una billetera determinista, en hex

```
0C1E24E5917779D297E14D45F14E1A1A
```

Una semilla para una billetera determinista, a partir de una frase mnemónica de 12 palabras

```
army van defense carry jealous true
garbage claim echo media make crunch
```

Mejores Prácticas de Cartera

A medida que la tecnología de cartera bitcoin ha madurado, han surgido ciertos estándares comunes de la industria que hacen que las carteras bitcoin sean ampliamente interoperables, fáciles de usar, seguras y flexibles. Estos estándares comunes son:

- Palabras código mnemónicas, basado en BIP-39
- Carteras HD, basado en BIP-32
- Estructura de cartera HD multipropósito, basado en BIP-43
- Carteras multimoneda y multicuenta, basado en BIP-44

Estos estándares pueden cambiar o volverse obsoletos por futuros desarrollos, pero por ahora forman un conjunto de tecnologías interdependientes que se han convertido en el estándar de facto de cartera bitcoin.

Los estándares han sido adoptados por una amplia gama de carteras bitcoin de software y hardware, haciendo que todas estas carteras sean interoperables. Un usuario puede exportar un mnemónico generado en una de estas carteras e importarlo en otra cartera, recuperando todas las transacciones, llaves y direcciones.

Algunos ejemplos de carteras software que admiten estos estándares son (enumerados alfabéticamente) Breadwallet, Copay, Multibit HD y Mycelium. Los ejemplos de carteras hardware que admiten estos estándares son (enumerados alfabéticamente) Keepkey, Ledger y Trezor.

Las siguientes secciones examinan en detalle algunas de estas tecnologías.

> **TIP** Si estás implementando una cartera bitcoin, debe construirse como una cartera HD, con una semilla codificada como código mnemotécnico para copia de seguridad, siguiendo los estándares BIP-32, BIP-39, BIP-43 y BIP-44, como se describe en las siguientes secciones.

Usando una Cartera Bitcoin

En Usos de Bitcoin, Usuarios y Sus Historias presentamos a Gabriel, un joven emprendedor en Río de Janeiro, que dirige una tienda web simple que vende camisetas, tazas de café y pegatinas con la marca de bitcoin.

Gabriel usa una cartera hardware Trezor de bitcoin (Un dispositivo Trezor: una cartera HD de bitcoin en hardware) para administrar de forma segura sus bitcoin. El Trezor es un dispositivo USB simple con dos botones que almacena llaves (en

forma de cartera HD) y firma transacciones. Las carteras Trezor implementan todos los estándares de la industria discutidos en este capítulo, por lo que Gabriel no depende de ninguna tecnología patentada o solución de un solo proveedor.

Figure 27. Un dispositivo Trezor: una cartera HD de bitcoin en hardware

Cuando Gabriel usó el Trezor por primera vez, el dispositivo generó un mnemónico y una semilla desde un generador de números aleatorios incorporado en su hardware. Durante esta fase de inicialización, la cartera mostró una secuencia numerada de palabras, una por una, en la pantalla (ver Trezor mostrando una de las palabras mnemónicas).

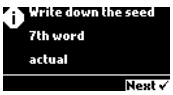


Figure 28. Trezor mostrando una de las palabras mnemónicas

Al escribir este mnemónico, Gabriel creó una copia de seguridad (ver Copia de seguridad en papel del mnemónico de Gabriel) que puede usarse como copia de seguridad en caso de pérdida o daño en el dispositivo Trezor. Este mnemónico sirve de resguardo en un nuevo Trezor o en cualquiera de las muchas carteras de software o hardware compatibles. Ten en cuenta que la secuencia de palabras es importante, por lo que las copias de seguridad de los mnemónicos en papel tienen espacios numerados para cada palabra. Gabriel tuvo que registrar cuidadosamente cada palabra en el espacio numerado para conservar la secuencia correcta.

Table 9. Copia de seguridad en papel del mnemónico de Gabriel

1.	*army*	**7.**	*garbage*
2.	*van*	**8.**	*claim*
3.	*defense*	**9.**	*echo*
4.	*carry*	**10.**	*media*
5.	*jealous*	**11.**	*make*
6.	*true*	**12.**	*crunch*

NOTE

Para simplificar, se muestra un mnemónico de 12 palabras en Copia de seguridad en papel del mnemónico de Gabriel. De hecho, la mayoría de las carteras hardware generan un mnemónico de 24 palabras, que es más seguro. El mnemónico se usa exactamente de la misma manera, independientemente de su longitud.

Para la primera implementación de su tienda web, Gabriel usa una única dirección bitcoin, generada en su dispositivo Trezor. Esta dirección única es utilizada por todos los clientes para todos los pedidos. Como veremos, este enfoque tiene algunos inconvenientes y puede mejorarse con una cartera HD.

Detalles de la Tecnología de Cartera

Examinemos ahora en detalle cada uno de los importantes estándares de la industria que se utilizan en muchas carteras bitcoin.

Palabras Código Mnemónicas (BIP-39)

Las palabras código mnemónicas son palabras en inglés que representan (codifican) un número aleatorio utilizado como semilla para obtener una cartera determinista. La secuencia de palabras es suficiente para volver a crear la semilla y desde allí volver a crear la cartera y todas las llaves derivadas. Una aplicación de cartera que implementa carteras

deterministas con código mnemónico mostrará al usuario una secuencia de 12 a 24 palabras al crear la cartera por primera vez. Esa secuencia de palabras es la copia de seguridad de la cartera y se puede utilizar para recuperar y volver a crear todas las llaves de la misma o de cualquier aplicación de cartera compatible. Las palabras mnemónicas hacen que sea más fácil para los usuarios realizar copias de seguridad de las carteras, ya que son fáciles de leer y transcribir correctamente, en comparación con una secuencia aleatoria de números.

TIP

Las palabras mnemónicas a menudo se confunden con "carteras mentales". No son lo mismo. La principal diferencia es que una cartera mental consiste en palabras elegidas por el usuario, mientras que las palabras mnemónicas se crean aleatoriamente por la cartera y se presentan al usuario. Esta importante diferencia hace que las palabras mnemónicas sean mucho más seguras, porque los humanos somos fuentes muy pobres de aleatoriedad.

Los códigos mnemónicos se definen en BIP-39 (ver Propuestas de Mejora para Bitcoin). Ten en cuenta que BIP-39 es una implementación de un estándar de código mnemónico. Existe un estándar diferente, con un conjunto de palabras diferente, utilizado por la cartera Electrum y anterior a BIP-39. BIP-39 fue propuesto por la compañía que fabrica la cartera de hardware Trezor y es incompatible con la implementación de Electrum. Sin embargo, BIP-39 ya ha logrado un amplio apoyo de la industria a través de docenas de implementaciones interoperables y debe considerarse el estándar de facto de la industria.

BIP-39 define la creación de un código mnemónico y semilla, que describimos aquí en nueve pasos. Para mayor claridad, el proceso se divide en dos partes: los pasos 1 a 6 se muestran en Generando palabras mnemónicas y los pasos 7 a 9 se muestran en De mnemónico a semilla.

Generando palabras mnemónicas

La cartera genera automáticamente las palabras mnemónicas mediante el proceso estandarizado definido en BIP-39. La cartera comienza desde una fuente de entropía, agrega un checksum y luego mapea la entropía a una lista de palabras:

1. Crear una secuencia aleatoria (entropía) de 128 a 256 bits.
2. Crear un checksum de la secuencia aleatoria tomando los primeros (longitud-de-entropía/32) bits de su hash SHA256.
3. Anexar el checksum al final de la secuencia aleatoria.
4. Dividir el resultado en segmentos de 11 bits de longitud.
5. Mapear cada valor de 11 bits a una palabra del diccionario predefinido de 2048 palabras.
6. El código mnemónico es la secuencia de palabras.

Generando entropía y codificándola como palabras mnemónicas muestra cómo se usa la entropía para generar las palabras mnemónicas.

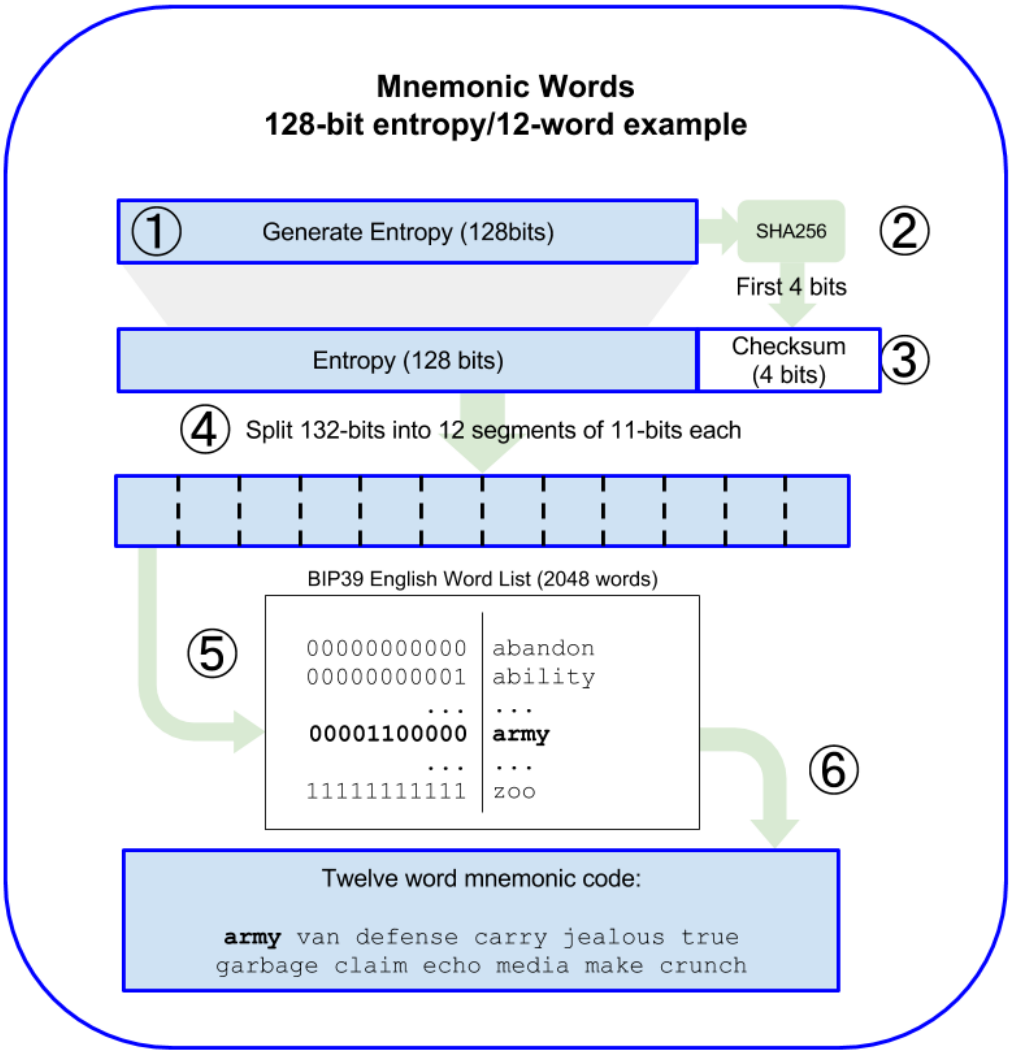


Figure 29. Generando entropía y codificándola como palabras mnemónicas

Códigos mnemónicos: entropía y longitud de palabra muestra la relación entre el tamaño de los datos de entropía y la longitud de los códigos mnemónicos en palabras.

Table 10. Códigos mnemónicos: entropía y longitud de palabra

Entropía (bits)	Checksum (bits)	Entropía + checksum (bits)	Longitud de mnemónico (palabras)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

De mnemónico a semilla

Las palabras mnemónicas representan la entropía con una longitud de 128 a 256 bits. La entropía se utiliza para obtener una semilla más larga (512 bits) mediante el uso de la función de estiramiento de llaves PBKDF2. La semilla producida se usa luego para construir una cartera determinista y derivar sus llaves.

La función de estiramiento de llaves toma dos parámetros: el mnemónico y una *sal*. El propósito de una sal en una función de estiramiento de llaves es hacer que sea difícil construir una tabla de búsqueda que permita un ataque de fuerza bruta. En el estándar BIP-39, la sal tiene otro propósito—permite la introducción de una frase de contraseña que sirve como un factor de seguridad adicional que protege la semilla, como describiremos con más detalle en Frase de contraseña opcional en BIP-39.

El proceso descrito en los pasos 7 a 9 continúa desde el proceso descrito anteriormente en Generando palabras mnemónicas:

7. El primer parámetro de la función de estiramiento de llaves PBKDF2 es el *mnemónico*producido a partir del paso 6.
8. El segundo parámetro de la función de estiramiento de llaves PBKDF2 es la *sal*. La sal se compone de la constante de cadena "`mnemónico`" concatenado con una frase de contraseña opcional proporcionada por el usuario.
9. PBKDF2 estira los parámetros mnemónico y sal mediante 2048 rondas de hash con el algoritmo HMAC-SHA512, produciendo un valor de 512 bits como salida final. Ese valor de 512 bits es la semilla.

De mnemónico a semilla muestra cómo se usa un mnemónico para generar una semilla.

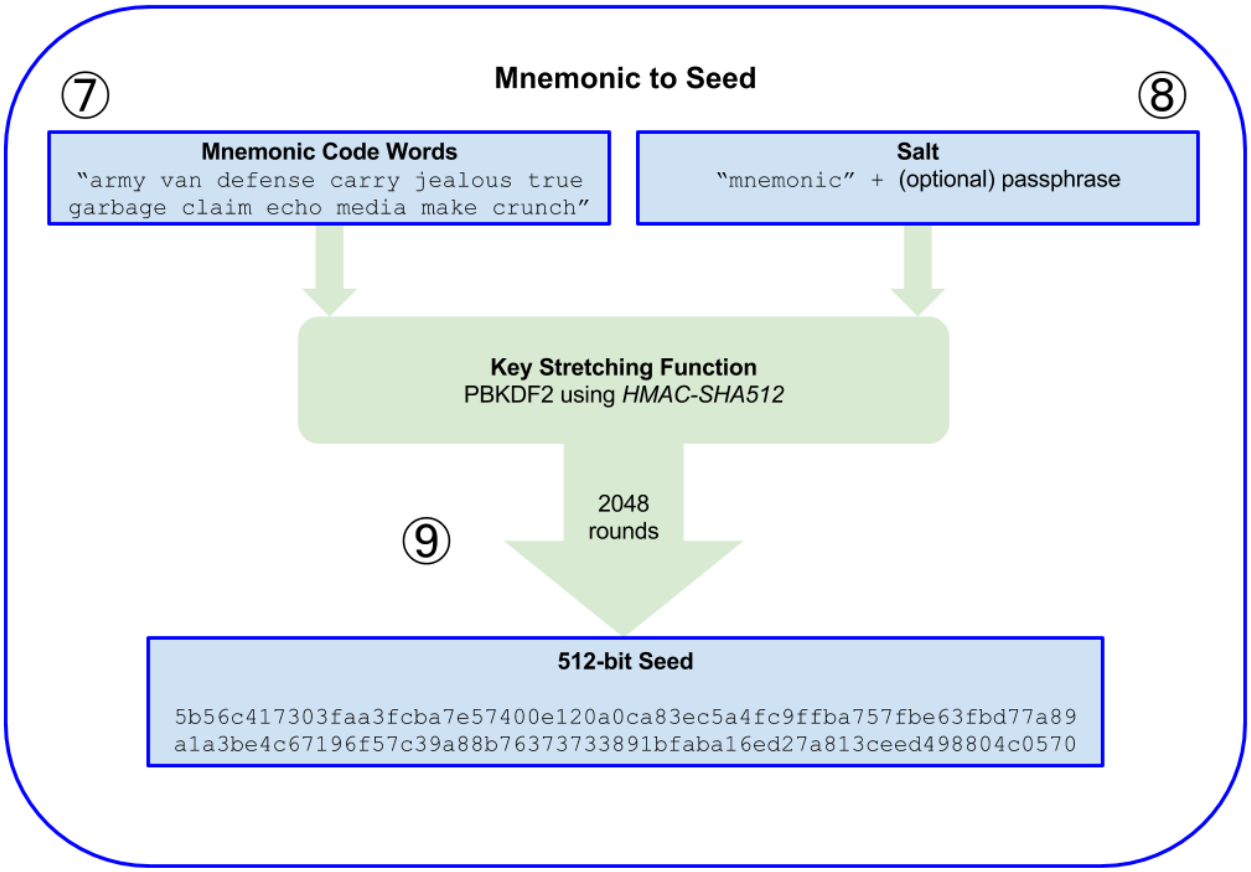


Figure 30. De mnemónico a semilla

> **TIP** La función de estiramiento de llaves, con sus 2048 rondas de hash, es una protección muy efectiva contra los ataques de fuerza bruta contra el mnemónico o la frase de contraseña. Es extremadamente costoso (en computación) probar más de unos pocos miles de combinaciones de frases de contraseña y de mnemónicos, mientras que el número de posibles semillas que se pueden derivar es enorme (2^{512}).

Las tablas #mnemonic_128_no_pass, #mnemonic_128_w_pass, y #mnemonic_256_no_pass muestran algunos ejemplos de códigos mnemónicos y las semillas que éstos producen (con o sin frase de contraseña).

Table 11. Código mnemónico con 128 bits de entropía, sin frase de contraseña, semilla resultante

Entrada de entropía (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Mnemónico (12 palabras)	army van defense carry jealous true garbage claim echo media make crunch
Frase de contraseña	(ninguna)
Semilla (512 bits)	5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39a88b76373733891bfaba16ed27a813ceed498804c0570

Table 12. Código mnemónico con 128 bits de entropía, con frase de contraseña, semilla resultante

Entrada de entropía (128	0c1e24e5917779d297e14d45f14e1a1a

bits)	
Mnemónico (12 palabras)	army van defense carry jealous true garbage claim echo media make crunch
Frase de contraseña	SuperDuperSecret
Semilla (512 bits)	3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0715861dc8a18358f80b79d49acf64142ae57037d1d54

Table 13. Código mnemónico con 256 bits de entropía, sin frase de contraseña, semilla resultante

Entrada de entropía (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Mnemónico (24 palabras)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Frase de contraseña	(ninguna)
Semilla (512 bits)	3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e55f1e0deaa082df8d487381379df848a6ad7e98798404

Frase de contraseña opcional en BIP-39

El estándar BIP-39 permite el uso de una frase de contraseña opcional en la derivación de la semilla. Si no se usa frase de contraseña, el mnemónico se estira con una sal que consiste en la cadena constante "mnemonic", produciendo una semilla específica de 512 bits para cualquier mnemónico dado. Si se usa una frase de contraseña, la función de estiramiento produce una semilla *diferente* para ese mismo mnemónico. De hecho, dado un mnemónico único, para cada frase de contraseña distinta se produce una semilla diferente. Esencialmente, no hay una frase de contraseña "incorrecta". Todas las frases de contraseña son válidas y todas conducen a diferentes semillas, formando un enorme conjunto de posibles carteras sin inicializar. El conjunto de carteras posibles es tan grande (2^{512}) que no es factible atacar por fuerza bruta o adivinar accidentalmente una que esté en uso.

> **TIP** No existen frases de contraseña "incorrectas" en BIP-39. Cada frase de contraseña conduce a una cartera, que, a menos que se haya utilizado anteriormente, estará vacía.

La frase de contraseña opcional crea dos características importantes:

- Un segundo factor (algo memorizado) que hace que un mnemónico que es inútil en sí, proteja copias de seguridad de mnemónicos de ser accedidas por un ladrón.
- Una forma de negación plausible o "cartera ante coacción", donde una frase de contraseña elegida conduce a una cartera con una pequeña cantidad de fondos que se usan para distraer a un atacante de la cartera "real" que contiene la mayoría de los fondos.

Sin embargo, es importante tener en cuenta que el uso de una frase de contraseña también introduce el riesgo de pérdida:

- Si el propietario de la cartera está incapacitado o muerto y nadie más conoce la frase de contraseña, la semilla no sirve para nada y todos los fondos almacenados en la cartera se pierden para siempre.
- Por otro lado, si el propietario guarda la frase de contraseña en el mismo lugar que la semilla, anula el propósito de un segundo factor.

Si bien las frases de contraseña son muy útiles, solo deben usarse en combinación con un proceso de respaldo y recuperación cuidadosamente planificado, considerando la posibilidad de sobrevivir al propietario y permitiendo a su familia recuperar el patrimonio de criptomoneda.

Trabajando con códigos mnemónicos

BIP-39 se implementa como biblioteca en muchos lenguajes de programación diferentes:

python-mnemonic
: La implementación de referencia del estándar por el equipo de SatoshiLabs que propuso BIP-39, en Python

bitcoinjs/bip39

Una implementación de BIP-39, como parte del popular sistema bitcoinJS, en JavaScript

libbitcoin/mnemonic

Una implementación de BIP-39, como parte del popular sistema Libbitcoin, en C++

También hay un generador BIP-39 implementado en una página web independiente, que es extremadamente útil para pruebas y experimentación. Un generador BIP-39 como una página web independiente muestra una página web independiente que genera mnemónicos, semillas y llaves privadas extendidas.

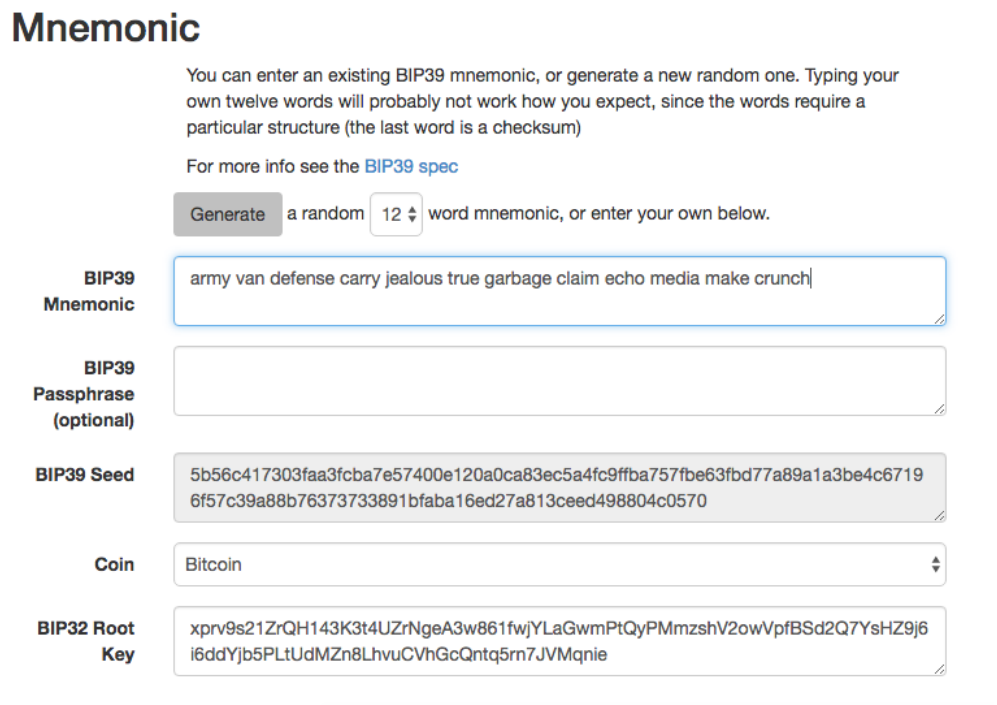


Figure 31. Un generador BIP-39 como una página web independiente

La página (https://iancoleman.github.io/bip39/) se puede usar sin conexión en un navegador, o se puede acceder a ella en línea.

Creando una Cartera HD desde la Semilla

Las carteras HD se crean a partir de una sola *semilla raíz*, que es un número aleatorio de 128, 256 o 512 bits. Más comúnmente, esta semilla se genera a partir de un *mnemónico* como se detalla en la sección anterior.

Cada llave en la cartera HD se deriva determinísticamente a partir de esta semilla raíz, lo que hace posible recrear la cartera HD completa desde esa semilla en cualquier cartera HD compatible. Esto facilita la copia de seguridad, restauración, exportación e importación de carteras HD que contienen miles o incluso millones de llaves, simplemente transfiriendo solo el mnemónico del que se deriva la semilla raíz.

El proceso de creación de llaves maestras y código de cadena maestro para una cartera HD se muestra en Creando llaves y códigos de cadena maestros a partir de una semilla raíz.

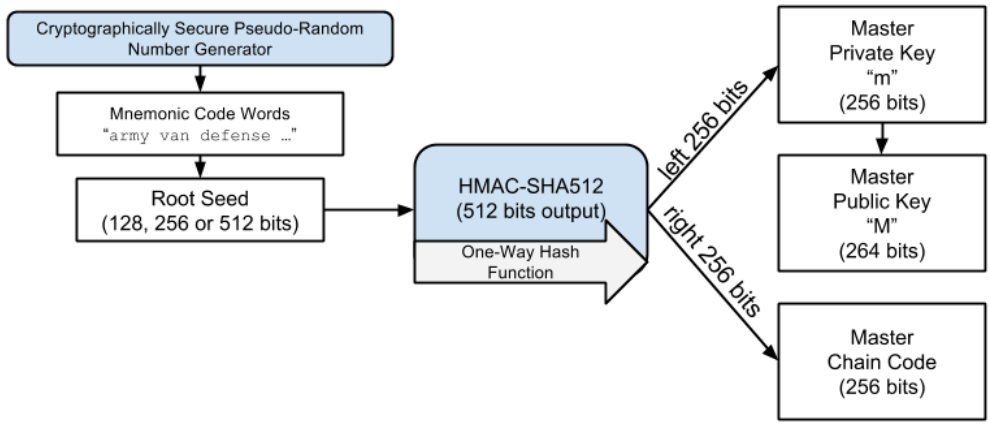


Figure 32. Creando llaves y códigos de cadena maestros a partir de una semilla raíz

La semilla raíz se ingresa en el algoritmo HMAC-SHA512 y el hash resultante se usa para crear una *llave privada maestra* (m) y un *código de cadena maestro* (c).

La llave privada maestra (m) genera una llave pública maestra (M) correspondiente, mediante el proceso normal de multiplicación de curva elíptica m * G que vimos en Llaves Públicas.

El código de cadena (c) se utiliza para introducir la entropía en la función que crea llaves hijas a partir de llaves principales, como veremos en la siguiente sección.

Derivación de la llave privada hija

Las carteras HD utilizan una función de *derivación de llave hija* (CKD, Child Key Derivation) para derivar llaves hijas a partir de llaves principales.

Las funciones de derivación de llaves hijas se basan en una función de hash de sentido único que combina:

- Una llave privada o pública principal (llave comprimida ECDSA)
- Una semilla llamada código de cadena (256 bits)
- Un número índice (32 bits)

El código de cadena se utiliza para introducir datos aleatorios deterministas en el proceso, de modo que conocer el índice y una llave hija no es suficiente para derivar otras llaves hijas. Saber una llave hija no permite encontrar a sus hermanas, a menos que también tengas el código de cadena. La semilla inicial del código de cadena (en la raíz del árbol) se obtiene a partir de la semilla, mientras que los códigos de cadena hijos posteriores se derivan de cada código de cadena padre.

Estos tres elementos (llave padre, código de cadena e índice) son combinados y hasheados para generar llaves hijas, de la siguiente manera.

La llave pública padre, el código de cadena y el número de índice son combinados y hasheados con el algoritmo HMAC-SHA512 para producir un hash de 512 bits. Este hash de 512 bits se divide en dos mitades de 256 bits. Los 256 bits de la mitad derecha de la salida de hash se convierten en el código de cadena para el hijo. Los 256 bits de la mitad izquierda del hash se añaden a la llave privada padre para generar la llave privada hija. En Extendiendo una llave privada padre para crear una llave privada hijo., vemos esto ilustrado con el índice establecido en 0 para producir el hijo "cero" (primero por índice) del padre.

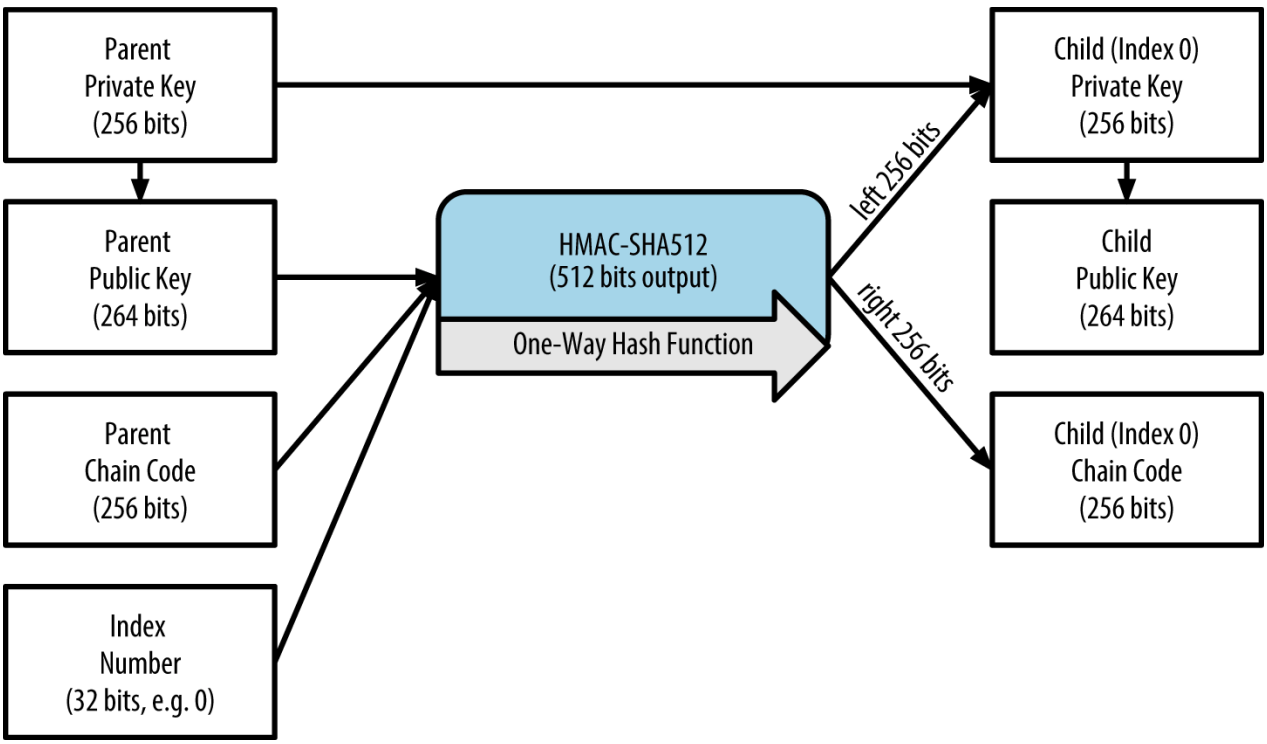


Figure 33. Extendiendo una llave privada padre para crear una llave privada hijo.

Cambiar el índice nos permite extender el padre y crear los otros hijos en la secuencia, por ejemplo, Hijo 0, Hijo 1, Hijo 2, etc. Cada llave padre puede tener 2,147,483,647 (2^{31}) hijos (2^{31} es la mitad del rango completo disponible de 2^{32} porque la

otra mitad está reservada para un tipo especial de derivación que veremos más adelante en este capítulo).

Repitiendo el proceso en un nivel inferior del árbol, cada hijo puede a su vez convertirse en un padre y crear sus propios hijos, en un número infinito de generaciones.

Usando llaves hijas derivadas

Las llaves privadas hijas son indistinguibles de las llaves no deterministas (aleatorias). Debido a que la función de derivación es una función de un único sentido, la llave hija no se puede usar para encontrar la llave padre. La llave hija tampoco se puede usar para encontrar hermanas. Si tienes la hija n_{ava}, no puedes encontrar a sus hermanas, tales como la hija número n–1 o la hija n+1, o cualquier otra hija que forme parte de la secuencia. Solo la llave padre y el código de cadena pueden derivar a todas las hijas. Sin el código de la cadena hija, la llave hija tampoco se puede utilizar para derivar nietos. Necesitas tanto la llave privada hija como el código de la cadena hija para comenzar una nueva rama y obtener nietos.

Entonces, ¿para qué se puede utilizar la llave privada hija por sí sola? Se puede utilizar para crear una llave pública y una dirección bitcoin. Después, se puede utilizar para firmar transacciones para gastar lo que se haya pagado a esa dirección.

TIP

Una llave privada hija, la llave pública correspondiente y la dirección bitcoin son indistinguibles de las llaves y direcciones creadas al azar. El hecho de que formen parte de una secuencia no es visible fuera de la función de cartera HD que los creó. Una vez creados, funcionan exactamente como llaves "normales".

Llaves extendidas

Como vimos anteriormente, la función de derivación de llaves se puede utilizar para crear los hijos en cualquier nivel del árbol, sobre la base de las tres entradas: una llave, un código de cadena, y el índice del hijo deseado. Los dos ingredientes esenciales son la llave y el código de cadena, que cuando se combinan, forman lo que se llama una *llave extendida*. El término "llave extendida" también podría pensarse como "llave extensible" porque dicha llave se puede utilizar para crear los hijos.

Las llaves extendidas se almacenan y se representan simplemente como la concatenación de la llave de 256 bits y el código de cadena de 256 bits en una secuencia de 512 bits. Hay dos tipos de llaves extendidas. Una llave privada extendida es la combinación de una llave privada y el código de cadena, y se puede utilizar para derivar las llaves privadas hijas (y a partir de ellas, las llaves públicas hijas). Una llave pública extendida es una llave pública y el código de cadena, que puede utilizarse para crear las llaves públicas hijas (*solo públicas*), como se describe en Generando una Llave Pública.

Piensa en una llave extendida como el origen de una rama en la estructura de árbol de la cartera HD. Con el origen de la rama, puedes derivar el resto de la rama. La llave privada extendida puede crear una rama completa, mientras que la llave pública extendida *solo* puede crear una rama de llaves públicas.

TIP

Una llave extendida consiste en una llave pública o privada y en un código de cadena. Una llave extendida puede crear hijos, generando su propia rama en la estructura de árbol. Compartir una llave extendida da acceso a toda la rama.

Las llaves extendidas se codifican utilizando Base58Check, para facilitar la exportación e importación de diferentes carteras compatibles con BIP-32–. La codificación Base58Check para las llaves extendidas utiliza un número de versión especial que se traduce en el prefijo "xprv" y "xpub" cuando se codifican en caracteres de Base58, para que sean fácilmente reconocibles. Dado que la llave extendida puede ser de 512 ó 513 bits, es también mucho más larga que otras cadenas codificadas en Base58Check que hemos visto anteriormente.

Aquí hay un ejemplo de una llave *privada* extendida, codificada en Base58Check:

```
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c
```

Aquí está la llave *pública* extendida correspondiente, también codificada en Base58Check:

```
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9
```

Derivación de llave pública hija

Como se mencionó anteriormente, una característica muy útil de las carteras HD es la capacidad para derivar llaves hijas

públicas de las llaves públicas de los padres, *sin* tener las llaves privadas. Esto nos da dos maneras para obtener una llave pública hija: ya sea desde la llave privada hija, o directamente de la llave pública padre.

Una llave pública extendida puede usarse, por tanto, para derivar todas las llaves *públicas* (y solamente las llaves públicas) en esa rama de la estructura de la cartera HD.

Este método simplificado se puede utilizar para crear despliegues muy seguros en servidores que solo requieren de llaves públicas–, mediante una copia de una llave pública extendida, sin llaves privadas de ningún tipo. Ese tipo de despliegue puede producir un número infinito de llaves públicas y direcciones bitcoin, pero no se puede gastar el dinero enviado a esas direcciones. Mientras tanto, en otro servidor, más seguro, la llave privada extendida puede derivar todas las llaves privadas correspondientes para firmar transacciones y gastar el dinero.

Una aplicación común de esta solución es instalar una llave pública extendida en un servidor web que sirve una aplicación de comercio electrónico. El servidor web puede utilizar la función de derivación de llave pública para crear una nueva dirección bitcoin en cada transacción (por ejemplo, para un carrito de la compra del cliente). El servidor web no tendrá ninguna llave privada, que serían vulnerables al robo. Sin carteras HD, la única manera de hacer esto seria generar miles de direcciones de Bitcoin en un servidor seguro por separado y luego cargarlas previamente en el servidor de comercio electrónico. Este enfoque es engorroso y requiere un mantenimiento constante para garantizar que el servidor de comercio electrónico no "agote" las llaves.

Otra aplicación común de esta solución es el almacenamiento en frío o en carteras hardware. En este escenario, la llave privada extendida se puede almacenar en una cartera de papel o en un dispositivo de hardware (tal como una cartera hardware Trezor), mientras que la llave pública extendida puede mantenerse en línea. El usuario puede crear direcciones de "recepción" a voluntad, mientras que las llaves privadas se almacenan de forma segura en un lugar sin conexión. Para gastar los fondos, el usuario puede utilizar la llave privada extendida creando una firma en un cliente bitcoin sin conexión a la red, o firmar las transacciones en una cartera hardware (por ejemplo, Trezor). Extendiendo una llave pública padre para crear una llave pública hija ilustra el mecanismo para extender una llave pública padre para derivar llaves públicas hijas.

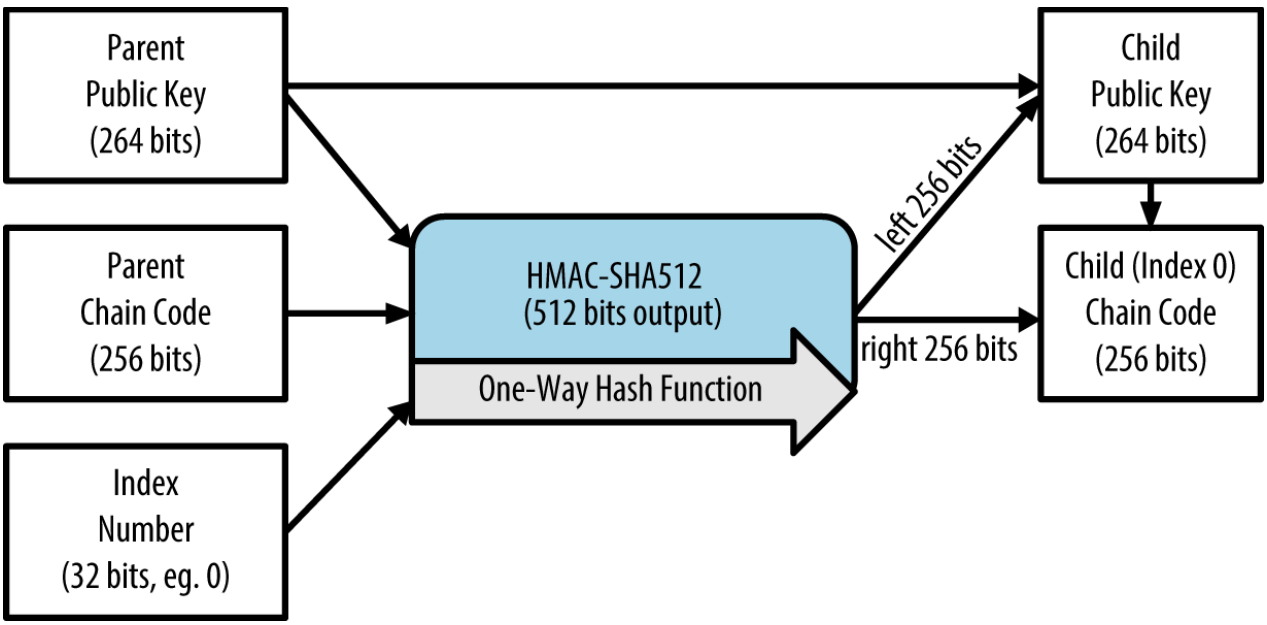


Figure 34. Extendiendo una llave pública padre para crear una llave pública hija

Usando una Llave Pública Extendida en una Tienda Web

Veamos cómo se usan las carteras HD mientras seguimos con nuestra historia de la tienda web de Gabriel.

Gabriel creó su tienda web como un hobby, basada en una página web simple de Wordpress. Su tienda era bastante básica con solo unas pocas páginas y un formulario de pedido con una sola dirección bitcoin.

Gabriel usó la primera dirección bitcoin generada por su dispositivo Trezor como la dirección bitcoin principal para su tienda. De esta manera, todos los pagos entrantes se pagarían a una dirección controlada por su cartera hardware Trezor.

Los clientes solicitarían un pedido utilizando el formulario y enviarían el pago a la dirección bitcoin publicada de Gabriel, lo que generaría un correo electrónico con los detalles del pedido para que Gabriel lo procesara. Para solo unos pocos

pedidos por semana, este sistema funcionó lo suficientemente bien.

Sin embargo, la pequeña tienda web tuvo bastante éxito y atrajo muchos pedidos de la comunidad local. Pronto, Gabriel estaba abrumado. Con todas las órdenes pagando a la misma dirección, se le hacía difícil emparejar correctamente las órdenes y las transacciones, especialmente cuando se tramitaban varias órdenes juntas por la misma cantidad.

La cartera HD de Gabriel ofrece una solución mucho mejor a través de la capacidad de derivar llaves públicas hijas sin conocer las llaves privadas. Gabriel puede cargar una llave pública extendida (xpub) en su sitio web, que puede usarse para obtener una dirección única para cada pedido de los clientes. Gabriel puede gastar los fondos de su Trezor, pero el xpub cargado en el sitio web solo puede generar direcciones y recibir fondos. Esta característica de las carteras HD es una gran característica de seguridad. El sitio web de Gabriel no contiene ninguna llave privada y, por lo tanto, no necesita altos niveles de seguridad.

Para exportar el xpub, Gabriel utiliza el software web junto con la cartera hardware Trezor. El dispositivo Trezor debe estar conectado para exportar las llaves públicas. Tenga en cuenta que las carteras hardware nunca exportarán llaves privadas— esas siempre permanecerán en el dispositivo. Exportando un xpub desde una cartera hardware Trezor muestra la interfaz web que usa Gabriel para exportar el xpub.

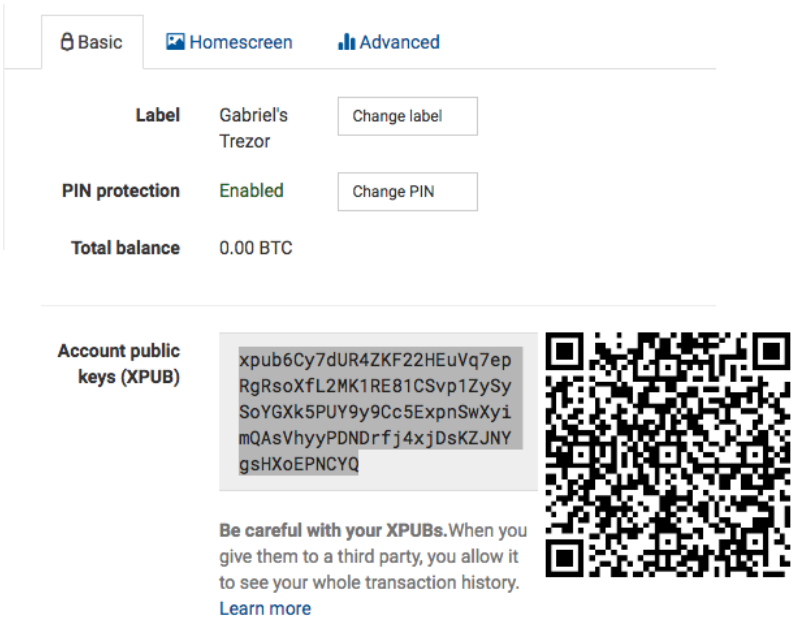


Figure 35. Exportando un xpub desde una cartera hardware Trezor

Gabriel copia el xpub al software de su tienda web de comercio de bitcoin. Utiliza *Mycelium Gear*, que es un complemento de código abierto de la tienda web para una variedad de plataformas de contenido y alojamiento web. Mycelium Gear utiliza el xpub para generar una dirección única para cada compra.

Derivación reforzada de llaves hijas

La capacidad de derivar una rama de llaves públicas de una xpub es muy útil, pero viene con un riesgo potencial. El acceso a una xpub no da acceso a las llaves privadas hijas. Sin embargo, debido a que la xpub contiene el código de cadena, si se conoce una llave privada hija, o de alguna manera se filtró, se puede utilizar el código de cadena para derivar todas las otras llaves privadas hijas. Una única llave privada hija filtrada, junto con un código de cadena padre, revela todas las llaves privadas de todos los hijos. Peor aún, la llave privada hija junto con un código de cadena de los padres se puede utilizar para deducir la llave privada padre.

Para contrarrestar este riesgo, las carteras HD utilizan una función de derivación alternativa llamada *derivación reforzada*, que "rompe" la relación entre la llave pública padre y el código de cadena hijo. La función de derivación reforzada utiliza la llave privada padre para derivar el código de cadena hijo, en lugar de la llave pública padre. Esto crea un "cortafuegos" en la secuencia padre/hijo, con un código de cadena que no puede ser utilizado para comprometer un llave privada padre o hermana. La función de derivación reforzada parece casi idéntica a la derivación normal de la llave privada hija, a excepción de que la llave privada padre se utiliza como entrada a la función hash, en lugar de la llave pública padre, como se muestra en el diagrama en Derivación reforzada de una llave hija; omite la llave pública padre.

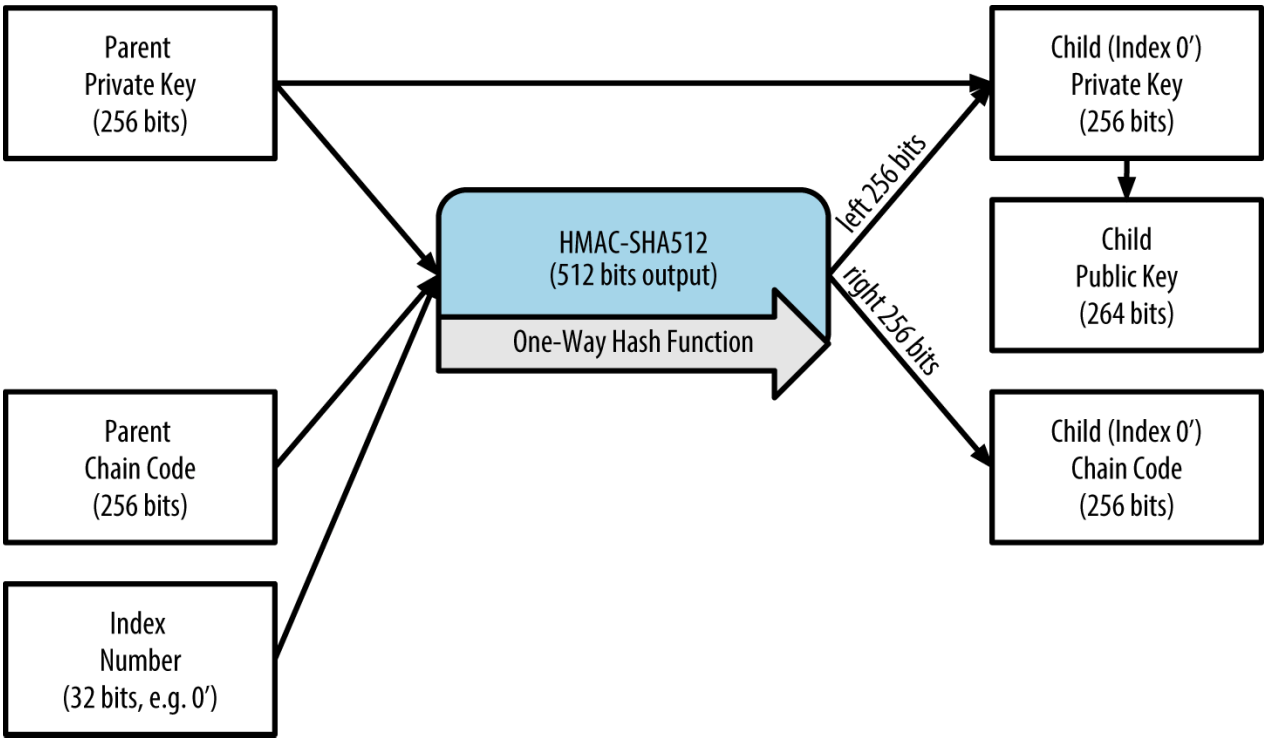


Figure 36. Derivación reforzada de una llave hija; omite la llave pública padre

Cuando se utiliza la función de derivación privada reforzada, la llave privada hija resultante y el código de cadena son completamente diferentes de lo que resultaría de la función normal de derivación. La "rama" resultante de las llaves puede utilizarse para producir las llaves públicas extendidas que no son vulnerables, debido a que el código de cadena que contienen no puede ser explotado para revelar ninguna llave privada. Por lo tanto, la derivación reforzada se utiliza para crear un "espacio" en el árbol por encima del nivel donde se utilizan las llaves públicas extendidas.

En términos simples, si deseas utilizar la conveniencia de un xpub para derivar ramas de llaves públicas, sin exponerte al riesgo de que se filtre un código de cadena, debes derivarlo de un padre reforzado, en lugar de un padre normal. Como práctica recomendada, los hijos de nivel-1 de las llaves maestras siempre se derivan a través de la derivación reforzada, para evitar el compromiso de las llaves maestras.

Números índice para derivación normal y reforzada

El número de índice que se utiliza en la función de derivación es un entero de 32 bits. Para distinguir fácilmente entre llaves derivadas a través de la función normal de derivación frente a llaves derivadas a través de la derivación reforzada, este número de índice se divide en dos rangos. Los números de índice entre 0 y 2^{31}–1 (0x0 a 0x7FFFFFFF), se usan *solo* para la derivación normal. Los números de índice entre 2^{31} y 2^{32}–1 (0x80000000 a 0xFFFFFFFF), se usan *solo* para la derivación reforzada. Por lo tanto, si el número de índice es menor que 2^{31}, eso significa que el hijo es normal, mientras que si el número de índice es igual o superior a 2^{31}, el hijo es reforzado.

Para que el número de índice sea más fácil de leer y de mostrar en pantalla, el número de índice para los hijos reforzados se presenta empezando de cero, pero con un símbolo prima. Por tanto, la primera llave hija normal se muestra como 0, mientras que el primer hijo reforzado (índice 0x80000000) se muestra como 0'. Continuando la secuencia, la segunda llave reforzada tendría índice 0x80000001 y se mostraría como 1', y así sucesivamente. Cuando veas un índice i' en una cartera HD, significa 2^{31}+i.

Identificador de llave de cartera HD (ruta)

Las llaves en una cartera HD se identifican mediante un convenio de descripción de "ruta", con cada nivel del árbol separado por el carácter barra (/) (ver Ejemplos de rutas de cartera HD). Las llaves privadas derivadas de la llave privada maestra empiezan con "m". Las llaves públicas derivadas de la llave pública maestra empiezan con "M". Por lo tanto, la primera llave privada hija de la llave privada maestra es m/0. La primera llave pública hija es M/0. El segundo nieto del primer hijo es m/0/1, y así sucesivamente .

Los "antepasados" de una llave se leen de derecha a izquierda, hasta llegar a la llave maestra de la que deriva. Por ejemplo, el identificador m/x/y/z describe la llave que es el hijo z-ésimo de la llave m/x/y, que a su vez es el hijo y-ésimo de la llave m/x, que es el hijo x-ésimo de m.

Table 14. Ejemplos de rutas de cartera HD

Ruta HD	Llave descrita
m/0	La primera (0) llave privada hija de la llave privada maestra (m)
m/0/0	La primera (0) llave privada derivada de la primera derivación (m/0)
m/0'/0	El primer (0) derivado normal del primer derivado *fortalecido* (m/0')
m/1/0	La primera (0) llave privada derivada de la segunda derivación (m/1)
M/23/17/0/0	La primera (0) llave pública derivada de la primera derivación (M/23/17/0) de la 18va derivación (M/23/17) de la 24va derivación (M/23)

Navegando por la estructura de árbol de la cartera HD

La estructura de árbol de la cartera HD ofrece una gran flexibilidad. Cada llave extendida padre puede tener 4 mil millones de hijos: 2 mil millones de hijos normales y 2 mil millones de hijos reforzados. Cada uno de estos hijos puede tener otros 4 mil millones de hijos, y así sucesivamente. El árbol puede ser tan profundo como se desee, con un número infinito de generaciones. Con toda esta flexibilidad, sin embargo, se hace muy difícil de navegar por este árbol infinito. Es especialmente difícil para transferir carteras HD entre implementaciones, debido a que las posibilidades de organización interna en ramas principales y secundarias son infinitas.

Dos BIPs ofrecen una solución a esta complejidad mediante la creación de algunas de las normas propuestas para la estructura de los árboles de cartera HD. BIP-43 propone el uso del primer índice hijo reforzado como un identificador especial que significa el "propósito" de la estructura de árbol. Basado en BIP-43, una cartera HD debería utilizar solo una rama del árbol de nivel-1, con el número de índice identificando la estructura y el espacio de nombres del resto del árbol mediante la definición de su propósito. Por ejemplo, una cartera HD que utilice una única rama m/i'/ intenta significar un propósito específico y ese propósito es identificado por el número de índice "i".

Ampliando esa especificación, BIP-44 propone una estructura multicuenta cuyo "objetivo" es el número 44' bajo BIP-43. Todas las carteras HD que cumplen con la estructura BIP-44 se identifican por el hecho de que sólo utilizan una rama del árbol: m/44'/.

BIP-44 especifica que la estructura se basa en cinco niveles predefinidos del árbol:

```
m / propósito' / tipo_moneda' / cuenta' / cambio / índice_dirección
```

El primer nivel "propósito" está siempre ajustado a 44'. El segundo nivel "tipo_moneda" especifica el tipo de criptomoneda, permitiendo carteras HD multidivisa donde cada moneda tiene su propio subárbol bajo el segundo nivel. Hay tres monedas definidas por ahora: Bitcoin es m/44'/0', Bitcoin Testnet es m/44'/1' y Litecoin es m/44'/2'.

El tercer nivel del árbol es "cuenta", que permite a los usuarios que subdividan sus carteras en subcuentas lógicas separadas, para la contabilidad o para propósitos organizativos. Por ejemplo, una cartera HD puede contener dos "cuentas" bitcoin: m/44'/0'/0' and m/44'/0'/1'. Cada cuenta es la raíz de su propio subárbol.

En el cuarto nivel, "cambio", una cartera HD tiene dos subárboles, uno para la creación de direcciones que reciben y otro para la creación de direcciones de cambio. Ten en cuenta que mientras que los niveles anteriores utilizaron derivación reforzada, este nivel utiliza derivación normal. Esto se hace para permitir que este nivel del árbol pueda exportar las llaves públicas extendidas para el uso en un entorno no seguro. Las direcciones utilizables se derivan de la cartera HD como hijos del cuarto nivel, haciendo que el quinto nivel del árbol sea el "indice_dirección". Por ejemplo, la tercera dirección de recepción para los pagos bitcoin en la cuenta principal sería M/44'/0'/0'/0/2. Ejemplos de estructuras de carteras HD BIP-44 muestra algunos ejemplos más.

Table 15. Ejemplos de estructuras de carteras HD BIP-44

Ruta HD	Llave descrita
M/44'/0'/0'/0/2	La tercera llave pública receptora para la cuenta bitcoin primaria
M/44'/0'/3'/1/14	La decimoquinta llave pública de la dirección de cambio para la cuarta cuenta bitcoin
m/44'/2'/0'/0/1	La segunda llave privada en la cuenta principal Litecoin, para las transacciones de firma

Transacciones

Introducción

Las transacciones son la parte más importante del sistema bitcoin. Todo lo demás en bitcoin fue diseñado para asegurar que las transacciones puedan ser creadas, propagadas por la red, validadas y finalmente añadidas al libro de contabilidad global (la cadena de bloques). Las transacciones son estructuras de datos que codifican la transferencia de valor entre los participantes en el sistema bitcoin. Cada transacción es una entrada pública en la cadena de bloques de bitcoin, el libro de contabilidad global por partida doble.

En este capítulo examinaremos las varias formas de transacciones, qué contienen, cómo crearlas, cómo se verifican y cómo se vuelven parte del registro permanente de todas las transacciones. Cuando usemos el término "cartera" en este capítulo, nos referimos al software que construye las transacciones, no solo a las llaves de la base de datos.

Transacciones en Detalle

En ¿Cómo funciona Bitcoin?, vimos la transacción que Alice usó para pagar el café en la cafetería de Bob usando un explorador de bloques (Transacción de Alice a la Cafetería de Bob).

La aplicación del explorador de bloques muestra una transacción desde la "dirección" de Alice a la "dirección" de Bob. Esta es una vista mucho más simplificada de lo que contiene una transacción. De hecho, como veremos en este capítulo, gran parte de la información que se muestra está construida por el explorador de bloques y no está realmente en la transacción.

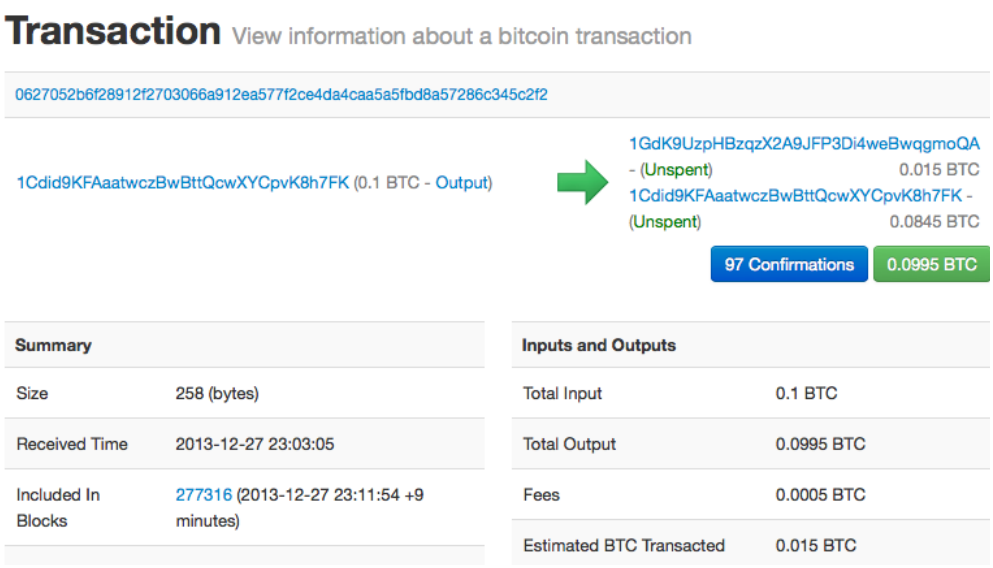


Figure 37. Transacción de Alice a la Cafetería de Bob

Transacciones—Detrás del Telón

Detrás del telón, una transacción real se muestra muy diferente a una transacción proporcionada por un explorador de bloques típico. De hecho, la mayoría de las construcciones de alto nivel que vemos en las diversas interfaces de usuario de las aplicaciones de bitcoin *no existen realmente* en el sistema bitcoin.

Podemos usar la interfaz de línea de comandos de Bitcoin Core (getrawtransaction y decoderawtransaction) para recuperar la transacción "sin formato" de Alice, descodificarla y ver qué contiene. El resultado se ve así:

La transacción de Alice decodificada

```
{
  "version": 1,
  "locktime": 0,
  "vin": [
    {
      "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
      "vout": 0,
      "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863
ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336
```

```
a8d752adf",
      "sequence": 4294967295
    }
  ],
  "vout": [
    {
      "value": 0.01500000,
      "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY OP_CHECKSIG"
    },
    {
      "value": 0.08450000,
      "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
    }
  ]
}
```

Puedes notar algunas cosas sobre esta transacción, ¡principalmente las cosas que faltan! ¿Dónde está la dirección de Alice? ¿Dónde está la dirección de Bob? ¿Dónde está la entrada 0.1 "enviada" por Alice? En bitcoin, no hay monedas, ni remitentes, ni destinatarios, ni saldos, ni cuentas, ni direcciones. Todas esas cosas se construyen en un nivel superior para beneficio del usuario, para que las cosas sean más fáciles de entender.

También puedes notar muchos campos extraños e indescifrables y cadenas hexadecimales. No te preocupes, explicaremos cada campo que se muestra aquí en detalle en este capítulo.

Entradas y Salidas de una Transacción

El componente fundamental de una transacción bitcoin es una *salida de transacción*. Las salidas de transacción son fragmentos indivisibles de la moneda bitcoin, grabados en la cadena de bloques y reconocidos como válidos por toda la red. Los nodos completos de bitcoin realizan un seguimiento de todas las salidas disponibles y consumibles, conocidas como *salidas de transacción sin gastar* o *UTXO* (del inglés, Unspent Transaction Output). La colección de todos los UTXO se conoce como *set UTXO* y actualmente asciende a millones de UTXO. El set UTXO crece a medida que se crea un nuevo UTXO y se reduce cuando se consume un UTXO. Cada transacción representa un cambio (una transición de estado) en el set UTXO.

Cuando decimos que la cartera de un usuario ha "recibido" bitcoin, lo que queremos decir es que la cartera ha detectado una UTXO que se puede gastar con una de las llaves controladas por esa cartera. Por lo tanto, el "saldo" de bitcoin de un usuario es la suma de todas las UTXO que la cartera del usuario puede gastar y que pueden estar dispersas entre cientos de transacciones y cientos de bloques. El concepto de saldo es creado por la aplicación de la cartera. La cartera calcula el saldo del usuario escaneando la cadena de bloques y agregando el valor de cualquier UTXO que la cartera puede gastar con las llaves que controla. La mayoría de las carteras mantienen una base de datos o usan un servicio de base de datos para almacenar un set de referencia rápida de todas las UTXOs que pueden gastar con las llaves que controlan.

Una salida de transacción puede tener un valor arbitrario (entero) denominado como un múltiplo de satoshis. Así como los dólares se pueden dividir en dos decimales como centavos, bitcoin se puede dividir en ocho decimales como satoshis. Aunque una salida puede tener cualquier valor arbitrario, una vez creada, es indivisible. Esta es una característica importante de las salidas que se debe enfatizar: las salidas son unidades de valor *discreto* e *indivisible*, denominadas en satoshis enteros. Una salida sin gastar solo puede ser consumida en su totalidad en una transacción.

Cuando un UTXO es mayor que el valor deseado de una transacción, se debe consumir en su totalidad y en la transacción se debe generar el cambio. En otras palabras, si tienes un UTXO con un valor de 20 bitcoin y quieres pagar solo 1 bitcoin, tu transacción debe consumir todo el UTXO de 20 bitcoin y producir dos salidas: una que paga 1 bitcoin al destinatario deseado y otra que paga 19 bitcoin en cambio de vuelta a tu cartera. Como las salidas de transacción son indivisibles, la mayoría de las transacciones bitcoin tendrán que generar cambio.

Imagina a un cliente que compra una bebida de $1.50, busca en su cartera y trata de encontrar una combinación de monedas y billetes para cubrir el costo de $1.50. El cliente elegirá la cantidad exacta si está disponible, por ejemplo. un billete de dólar y dos cuartos (un cuarto es $0.25), o una combinación de denominaciones más pequeñas (seis cuartos), o si es necesario, una unidad más grande, como un billete de $5. Si entrega demasiado dinero, digamos $5, al propietario de la tienda, esperará un cambio de $3.50, que retornará a su cartera y tendrá disponible para futuras transacciones.

Del mismo modo, una transacción bitcoin debe crearse desde el UTXO de un usuario en cualquiera de las denominaciones que ese usuario tenga disponible. Los usuarios no pueden cortar un UTXO a la mitad de la misma forma que no se puede cortar un billete de dólar a la mitad y usarlo como moneda. La aplicación de cartera del usuario normalmente seleccionará el UTXO disponible del usuario para componer una cantidad mayor o igual a la cantidad deseada de la

transacción.

Al igual que en la vida real, una aplicación bitcoin puede usar varias estrategias para satisfacer el monto de la compra: combinar varias unidades más pequeñas, encontrar el cambio exacto, o usar una única unidad mayor al valor de la transacción y generar cambio. Todo este complejo montaje de UTXOs es calculado por la cartera del usuario automáticamente y es invisible al usuario. Solo es relevante si estás construyendo transacciones sin formato a partir de UTXOs programáticamente.

Una transacción consume salidas de transacciones no gastadas registradas previamente y crea nuevas salidas de transacciones que pueden ser consumidas por una transacción futura. De esta manera, los fragmentos de valor de bitcoin avanzan de un propietario a otro en una cadena de transacciones que consumen y crean UTXO.

La excepción a la cadena de salidas y entradas es un tipo especial de transacción llamada transacción *coinbase*, que es la primera transacción en cada bloque. Esta transacción es colocada allí por el minero "ganador" y crea bitcoin completamente nuevos que se pagan a ese minero como recompensa por la minería. Esta transacción coinbase especial no consume ninguna UTXO; en su lugar, tiene un tipo especial de entrada llamada "coinbase". Así es como se crea la oferta monetaria de bitcoin durante el proceso de minería, como veremos en Minería y Consenso.

> **TIP** ¿Qué estuvo primero? ¿Entradas o salidas, el huevo o la gallina? Hablando en sentido estricto, las salidas están primero porque las transacciones coinbase, las cuales generan nuevos bitcoins, no poseen entradas y generan salidas de la nada.

Salidas de Transacción

Cada transacción de bitcoin crea salidas, que se registran en el libro de contabilidad de bitcoin. Casi todas estas salidas, con una excepción (ver Salida de Registro de Datos (RETURN)) crean trozos de bitcoin que se pueden gastar llamados UTXO, que luego son reconocidos por toda la red y están disponibles para que el propietario los gaste en una transacción futura.

Cada cliente bitcoin de nodo completo hace un seguimiento del set UTXO. Las nuevas transacciones consumen (gastan) una o más de estas salidas del set UTXO.

Las salidas de una transacción consisten en dos partes:

- Una cantidad de bitcoins denominada en *satoshis*, la unidad más pequeña de bitcoin
- Un acertijo criptográfico que determina las condiciones necesarias para gastar la salida

El acertijo criptográfico también se conoce como un script *de bloqueo*, un script *testigo*, o un scriptPubKey.

El lenguaje de script de transacción, utilizado en el script de bloqueo mencionado anteriormente, se describe en detalle en Scripts de Transacción y Lenguaje de Script.

Ahora, veamos la transacción de Alice (que se mostró anteriormente en Transacciones—Detrás del Telón) y veamos si podemos identificar las salidas. En la codificación JSON, las salidas están en un array (lista) llamado vout:

```
"vout": [
  {
    "value": 0.01500000,
    "scriptPubKey": "OP_DUP OP_HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 OP_EQUALVERIFY
    OP_CHECKSIG"
  },
  {
    "value": 0.08450000,
    "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG",
  }
]
```

Como puedes ver, la transacción contiene dos salidas. Cada salida está definida por un valor y un acertijo criptográfico. En la codificación mostrada por Bitcoin Core, el valor se muestra en bitcoin, pero en la transacción en sí se registra como un entero denominado en satoshis. La segunda parte de cada salida es el acertijo criptográfico que establece las condiciones para el gasto. Bitcoin Core muestra esto como scriptPubKey y nos muestra una representación legible del script.

El asunto del bloqueo y desbloqueo de UTXO se tratará más adelante, en Construcción de Scripts (Bloqueo + Desbloqueo).

El lenguaje de script que se usa para el script en scriptPubKey se describe en Scripts de Transacción y Lenguaje de Script. Pero antes de profundizar en esos temas, debemos comprender la estructura general de las entradas y salidas de las transacciones.

Serialización de transacciones—salidas

Cuando las transacciones se transmiten a través de la red o se intercambian entre aplicaciones, se *serializan*. La serialización es el proceso de convertir la representación interna de una estructura de datos en un formato que permite su transmisión un byte cada vez, también conocido como flujo de bytes. La serialización se usa más comúnmente para codificar estructuras de datos para la transmisión a través de una red o para el almacenamiento en un archivo. El formato de serialización de una salida de transacción se muestra en Serialización de salida de transacción.

Table 16. Serialización de salida de transacción

Tamaño	Campo	Descripción
8 bytes (little-endian)	Cantidad	Valor bitcoin en satoshis (10^{-8} bitcoin)
1–9 bytes (VarInt)	Tamaño Script de Bloqueo	Longitud Script de Bloqueo en bytes, a continuación
Variable	Script de Bloqueo	Un script definiendo las condiciones necesarias para gastar la salida

La mayoría de las bibliotecas y los framework de bitcoin no almacenan transacciones internamente como flujos de bytes, ya que eso requeriría un análisis complejo cada vez que se necesite acceder a un solo campo. Por conveniencia y legibilidad, las bibliotecas de bitcoin almacenan transacciones internamente en estructuras de datos (generalmente estructuras orientadas a objetos).

El proceso de conversión desde la representación del flujo de bytes de una transacción a la representación de la estructura de datos interna de una biblioteca se denomina *deserialización* o *parseo de la transacción*. El proceso de conversión inverso, como un flujo de bytes para la transmisión a través de la red, para el cálculo de hashes o para almacenamiento en disco, se denomina *serialización*. La mayoría de las bibliotecas de bitcoin tienen funciones integradas para la serialización y deserialización de transacciones.

Comprueba si puedes decodificar manualmente la transacción de Alice desde la forma hexadecimal serializada, localizando algunos de los elementos que vimos anteriormente. La sección que contiene las dos salidas se resalta en La transacción de Alice, serializada y presentada en notación hexadecimal para que sirva como ayuda:

Example 18. La transacción de Alice, serializada y presentada en notación hexadecimal

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff02 60e3160000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac 00000000
```

Aquí tienes algunas pistas:

- Hay dos salidas en la sección resaltada, cada una serializada como se muestra en Serialización de salida de transacción.
- El valor de 0.015 bitcoin es 1,500,000 satoshis. Eso es 16 e3 60 en hexadecimal.
- En la transacción serializada, el valor 16 e3 60 se codifica en little-endian (el primer byte es el menos significativo), por lo que aparece como 60 e3 16.
- La longitud del scriptPubKey es 25 bytes, que es 19 en hexadecimal.

Entradas de Transacción

Las entradas de transacción identifican (por referencia) qué UTXO se consumirá y proporcionan prueba de propiedad a través de un script de desbloqueo.

Para crear una transacción, una cartera selecciona entre las UTXO que controla, un UTXO con el valor suficiente para realizar el pago solicitado. A veces, un UTXO es suficiente, otras veces se necesita más de uno. Para cada UTXO que se consumirá para realizar este pago, la cartera crea una entrada que apunta al UTXO y la desbloquea con un script de desbloqueo.

Veamos los componentes de una entrada con mayor detalle. La primera parte de una entrada es un puntero a un UTXO mediante la referencia al hash de transacción y un índice de salida, que identifican el UTXO específico en esa transacción. La segunda parte es un script de desbloqueo, que la cartera construye para satisfacer las condiciones de gasto establecidas en el UTXO. La mayoría de las veces, el script de desbloqueo es una firma digital y una llave pública que demuestra la propiedad del bitcoin. Sin embargo, no todos los scripts de desbloqueo contienen firmas. La tercera parte es un número de secuencia, que se discutirá más adelante.

Considera nuestro ejemplo en Transacciones—Detrás del Telón. Las entradas de transacción son un array (lista) llamado vin:

Las entradas de transacción en la transacción de Alice

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863
ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336
a8d752adf",
    "sequence": 4294967295
  }
]
```

Como puedes ver, solo hay una entrada en la lista (porque un UTXO contenía suficiente valor para realizar este pago). La entrada contiene cuatro elementos:

- Un ID de transacción, que referencia a la transacción que contiene el UTXO que se está gastando
- Un índice de salida (vout), que identifica qué UTXO de esa transacción se está referenciando (la primera es cero)
- Un scriptSig, que satisface las condiciones impuestas en el UTXO, desbloqueándolo para gastar
- Un número de secuencia (que se explicará más adelante)

En la transacción de Alice, la entrada apunta al ID de transacción:

```
7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18
```

y al índice de la salida 0 (es decir, el primer UTXO creado por esa transacción). La cartera de Alice construye el script de desbloqueo, recuperando primero el UTXO al que se hace referencia, examinando su script de bloqueo y luego usándolo para construir el script de desbloqueo correcto que lo satisface.

Observando solo la entrada, puedes haber notado que no sabemos nada acerca de este UTXO, aparte de una referencia a la transacción que lo contiene. No sabemos su valor (cantidad en satoshi), y no conocemos el script de bloqueo que establece las condiciones para gastarlo. Para encontrar esta información, debemos recuperar el UTXO al que se hace referencia recuperando la transacción subyacente. Ten en cuenta que debido a que el valor de la entrada no se establece explícitamente, también debemos utilizar el UTXO al que se hace referencia para calcular las comisiones que se pagarán en esta transacción (ver Comisiones de Transacción).

No es solo la cartera de Alice la que necesita recuperar los UTXO a los que se hace referencia en las entradas. Una vez que esta transacción se transmite a la red, cada nodo de validación también deberá recuperar los UTXO a los que se hace referencia en las entradas de la transacción para validar la transacción.

Las transacciones por sí mismas parecen incompletas porque carecen de contexto. Hacen referencia a UTXO en sus entradas pero sin recuperar esas UTXO no podemos conocer el valor de las entradas o sus condiciones de bloqueo. Al

escribir software de bitcoin, cada vez que decodifiques una transacción con la intención de validarla o contar las comisiones o verificar el script de desbloqueo, tu código primero tendrá que recuperar los UTXO referenciados de la cadena de bloques para construir el contexto implícito pero no presente en los UTXO referenciados por las entradas. Por ejemplo, para calcular la cantidad pagada en comisiones, debes conocer la suma de los valores de entradas y salidas. Pero sin recuperar los UTXO a los que se hace referencia en las entradas, no conoces sus valores. Por lo tanto, una operación aparentemente simple como el cálculo de comisiones en una sola transacción involucra múltiples pasos y datos de múltiples transacciones.

Con Bitcoin Core podemos usar la misma secuencia de comandos que usamos al recuperar la transacción de Alice (getrawtransaction y decoderawtransaction). Con eso podemos obtener la referencia UTXO en la entrada anterior y echar un vistazo:

El UTXO de Alice de la transacción anterior, referenciado en la entrada

```
"vout": [
   {
     "value": 0.10000000,
     "scriptPubKey": "OP_DUP OP_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP_EQUALVERIFY OP_CHECKSIG"
   }
 ]
```

Vemos que este UTXO tiene un valor de 0.1 BTC y que tiene un script de bloqueo (scriptPubKey) que contiene "OP_DUP OP_HASH160 ...".

> **TIP** Para comprender completamente la transacción de Alice, tuvimos que recuperar las transacciones anteriores a las que se hace referencia como entradas. Una función que recupera transacciones anteriores y salidas de transacciones no gastadas es muy común y existe en casi todas las bibliotecas y API de bitcoin.

Serialización de transacciones—entradas

Cuando las transacciones se serializan para su transmisión en la red, sus entradas se codifican en un flujo de bytes como se muestra en Serialización de entradas de transacción.

Table 17. Serialización de entradas de transacción

Tamaño	Campo	Descripción
32 bytes	Hash de Transacción	Puntero a la transacción que contiene el UTXO a ser gastado
4 bytes	Índice de Salida	El número de índice del UTXO a ser gastado; comenzando por 0
1–9 bytes (VarInt)	Tamaño Script-de-Desbloqueo	Longitud del Script-de-Desbloqueo en bytes, a continuación
Variable	Script-de-Desbloqueo	Un script que cumple las condiciones del script de bloqueo del UTXO
4 bytes	Número de Secuencia	Usado para locktime o deshabilitado (0xFFFFFFFF)

Al igual que con las salidas, veamos si podemos encontrar las entradas de la transacción de Alice en formato serializado. Primero, las entradas decodificadas:

```
"vin": [
  {
    "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",
    "vout": 0,
    "scriptSig" :
"3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863
ea8f53982c09db8f6e3813[ALL]
0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336
a8d752adf",
    "sequence": 4294967295
  }
],
```

Ahora, veamos si podemos identificar estos campos en la codificación hexadecimal serializada en La transacción de Alice, serializada y presentada en notación hexadecimal:

Example 19. La transacción de Alice, serializada y presentada en notación hexadecimal

```
0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73
4d2804fe65fa35779000000008b483045022100884d142d86652a3f47
ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039
ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84
16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1
7b4a10fa336a8d752adfffffffff0260e31600000000001976a914ab6
8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000
1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000
```

Pistas:

- El ID de transacción se serializa en orden de bytes invertido, por lo que comienza con 18 (hexadecimal) y termina con 79
- El índice de la salida es un grupo de ceros de 4 bytes, fácil de identificar
- La longitud del scriptSig es 139 bytes, o 8b en hexadecimal
- El número de secuencia se establece en FFFFFFFF, de nuevo fácil de identificar

Comisiones de Transacción

La mayoría de las transacciones incluyen comisiones de transacción, que compensan a los mineros de bitcoin por proteger la red. Las comisiones también sirven como un mecanismo de seguridad, al hacer que sea económicamente inviable que los atacantes inunden la red con transacciones. La minería y las comisiones y recompensas recolectadas por los mineros se discuten con más detalle en Minería y Consenso.

Esta sección examina cómo las comisiones de transacción son incluidas en una transacción típica. La mayoría de las carteras calculan e incluyen comisiones de transacción automáticamente. Sin embargo, si estás construyendo transacciones programáticamente o usando una interfaz de línea de comando, debes tenerlo en cuenta e incluir estas comisiones manualmente.

Las comisiones de transacción sirven de incentivo para incluir (minar) una transacción en el siguiente bloque y también como desincentivo contra el abuso del sistema al requerir un pequeño costo en cada transacción. Las comisiones de transacción son recolectadas por el minero que mina el bloque que registra la transacción en la cadena de bloques.

Las comisiones de transacción se calculan según el tamaño de la transacción en kilobytes, no según el valor de la transacción en bitcoins. En general las comisiones de transacción se establecen en base a fuerzas del mercado en la red bitcoin. Los mineros priorizan transacciones basándose en distintos criterios, incluyendo las comisiones y pueden hasta procesar transacciones sin comisión bajo ciertas circunstancias. Las comisiones de transacción afectan la prioridad de procesado, lo cual significa que una transacción con comisiones suficientes será muy probablemente incluida en el próximo bloque minado, mientras que una transacción con pocas comisiones o sin comisiones puede ser demorada, procesada cuando sea posible después de algunos bloques, o jamás procesada. Las comisiones de transacción no son obligatorias y las transacciones sin comisión pueden resultar finalmente procesadas; sin embargo, incluir comisiones en transacciones incentiva al procesado prioritario.

Con el tiempo, ha evolucionado la forma en que se calculan las comisiones de transacción y el efecto que tienen en la priorización de la transacción. Al principio, las comisiones de transacción eran fijas y constantes en toda la red. Gradualmente, la estructura de comisiones se relajó y puede verse influida por las fuerzas del mercado, según la capacidad de la red y el volumen de transacciones. Desde al menos a principios de 2016, los límites de capacidad en bitcoin han creado una competencia entre transacciones, lo que resulta en comisiones más altas y, de hecho, hace que las transacciones gratuitas sean cosa del pasado. Las transacciones de comisión cero o de comisión muy baja rara vez se minan y, a veces, ni siquiera se propagan a través de la red.

En Bitcoin Core, las políticas de comisión de retransmisión se establecen mediante la opción minrelaytxfee. El valor actual predeterminado de minrelaytxfee es 0.00001 bitcoin o una centésima de milibitcoin por kilobyte. Por lo tanto, de forma

predeterminada, las transacciones con una comisión inferior a 0.00001 bitcoin se tratan como gratuitas y solo se retransmiten si hay espacio en el mempool; De lo contrario, se ignoran. Los nodos de Bitcoin pueden anular la política de comisión de retransmisión predeterminada ajustando el valor de minrelaytxfee.

Cualquier servicio de bitcoin que construya transacciones, incluidas carteras, intercambios, aplicaciones al por menor de comercio, etc., deben implementar comisiones dinámicas. Las comisiones dinámicas se pueden implementar a través de un servicio de terceros de estimación de comisiones o con un algoritmo de estimación de comisiones incorporado. Si no estás seguro, comienza con un servicio de terceros y a medida que adquieras experiencia, diseña e implementa tu propio algoritmo si deseas eliminar la dependencia de terceros .

Los algoritmos de estimación de comisiones calculan la comisión apropiada, según la capacidad y las comisiones que ofrecen las transacciones compitiendo entre ellas. Estos algoritmos van desde el simplista (comisión promedio o mediana en el último bloque) hasta sofisticado (análisis estadístico). Estiman la comisión necesaria (en satoshis por byte) que dará a una transacción una alta probabilidad de ser seleccionada e incluida dentro de un cierto número de bloques. La mayoría de los servicios ofrecen a los usuarios la opción de elegir comisiones de prioridad alta, media o baja. Alta prioridad significa que los usuarios pagan comisiones más altas, pero es probable que la transacción se incluya en el siguiente bloque. La prioridad media y baja significa que los usuarios pagan comisiones de transacción más bajas, pero las transacciones pueden tardar mucho más en confirmarse.

Muchas aplicaciones de cartera utilizan servicios tercerizados para el cálculo de las comisiones. Un servicio popular es *https://bitcoinfees.earn.com/*, que provee una API y un gráfico visual mostrando las comisiones en satoshis/byte para diversas prioridades.

> **TIP** Las comisiones estáticas ya no son viables en la red bitcoin. Las carteras que establecen comisiones estáticas producirán una experiencia de usuario deficiente ya que las transacciones a menudo se "atascarán" y no se confirmarán. Los usuarios que no entienden las transacciones y comisiones de bitcoin se sienten frustrados por las transacciones "atascadas" porque creen que han perdido su dinero.

El gráfico en Servicio de estimación de comisiones bitcoinfees.earn.com muestra la estimación de comisiones en tiempo real en incrementos de 10 satoshis/byte y el tiempo de confirmación esperado (en minutos y en número de bloques) para transacciones con comisiones en cada rango. Para cada rango de comisión (por ejemplo, 61–70 satoshis/byte), dos barras horizontales muestran el número de transacciones no confirmadas (1405) y el número total de transacciones en las últimas 24 horas (102.975), con comisiones en ese rango. Según el gráfico, la comisión de alta prioridad recomendada en ese momento era de 80 satoshis/byte, una comisión que probablemente provocaría que la transacción fuese confirmada en el siguiente bloque (retraso nulo de bloques). Para una perspectiva, el tamaño medio de la transacción es de 226 bytes, por lo que la comisión recomendada por el tamaño de la transacción sería de 18.080 satoshis (0.00018080 BTC).

Los datos de estimación de comisiones se pueden recuperar a través de una simple HTTP REST API, en https://bitcoinfees.earn.com/api/v1/fees/recommended. Por ejemplo, en la línea de comando usando el comando curl:

Usando la API de estimación de comisiones

```
$ curl https://bitcoinfees.earn.com/api/v1/fees/recommended

{"fastestFee":80,"halfHourFee":80,"hourFee":60}
```

El API devuelve un objeto JSON con la estimación de comisión actual para la confirmación más rápida (fastFee), confirmación dentro de tres bloques (halfHourFee) y seis bloques (hourFee), en satoshi por byte.

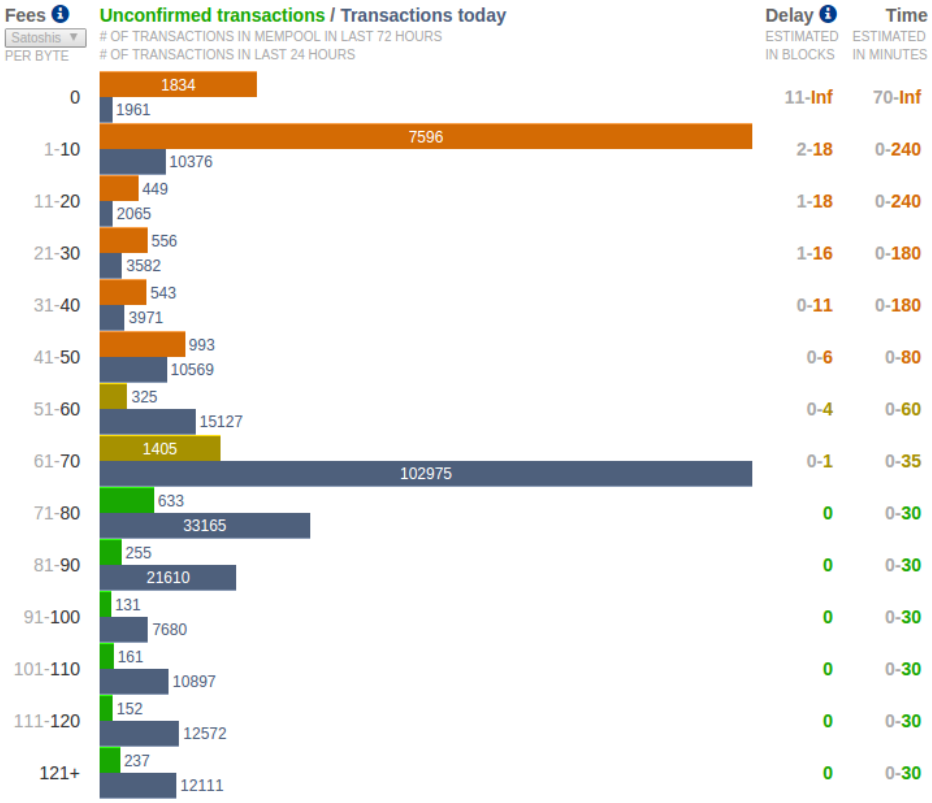


Figure 38. Servicio de estimación de comisiones bitcoinfees.earn.com

Añadiendo Comisiones a Transacciones

La estructura de datos de transacciones no posee un campo para comisiones. En cambio, las comisiones están implícitas como la diferencia entre la suma de las entradas y la suma de las salidas. Cualquier cantidad que sobre después de restar los valores de las entradas menos los de las salidas será la comisión recolectada por los mineros.

Las comisiones de transacción son implícitas como el excedente de entradas menos salidas:

```
Comisiones = Suma(Entradas) - Suma(Salidas)
```

Esto es un elemento un tanto confuso de las transacciones y un punto importante a entender, ya que si estás construyendo tus propias transacciones debes asegurarte de no incluir una comisión muy grande por descuido al gastar las entradas de menos. Esto significa que debes tener en cuenta todas las entradas, y si es necesario, crear una salida para el vuelto, ¡o terminarás dándole a los mineros una propina muy grande!

Por ejemplo, si consumes un UTXO de 20 bitcoin para hacer un pago de 1 bitcoin, debes incluir una salida adicional para un cambio de 19 bitcoin de regreso a tu cartera. De lo contrario, los 19 bitcoin "sobrantes" serán contados como la comisión de transacción y serán recolectados por el minero que mine tu transacción en un bloque. Aunque recibirás un procesado prioritario y harás muy feliz a un minero, esto probablemente no era lo que planeabas hacer.

WARNING	Si te olvidas de añadir una salida de cambio en una transacción construida manualmente terminarás pagando el cambio como comisión de transacción. "¡Quédate el cambio!" puede no haber sido tu intención.

Veamos cómo funciona esto en la práctica usando nuevamente como ejemplo la compra de café de Alice. Alice quiere gastar 0.015 bitcoins para pagar por un café. Para asegurar que esta transacción sea procesada rápidamente, ella querrá incluir una comisión de transacción, digamos de 0.001. Esto significará que el costo total de la transacción será de 0.016 bitcoins o más y, de ser necesario, creará una salida para el cambio. Digamos que su cartera tiene una UTXO de 0.2 bitcoins disponible. Por lo tanto necesitará consumir totalmente esta UTXO, crear una salida para el Café de Bob por 0.015, y una segunda salida con 0.184 bitcoin para el cambio, de regreso a su propia cartera, dejando 0.001 bitcoin sin asignar, lo cual será la comisión implícita para la transacción.

Ahora veamos un caso diferente. Eugenia, nuestra directora de la beneficencia para niños en Filipinas ha completado una recaudación de fondos para adquirir libros para los niños. Ha recibido varios miles de pequeñas donaciones de personas

de alrededor del mundo, las cuales suman 50 bitcoins, por lo que su cartera está llena de pagos muy pequeños (UTXO). Ahora ella quiere comprar cientos de libros escolares a una editorial local, pagando en bitcoin.

Ya que la aplicación de la cartera de Eugenia intenta construir una única gran transacción de pago, debe crearla a partir del set UTXO disponible, el cual está compuesto de múltiples montos más pequeños. Esto significa que la transacción resultante usará como fuente a más de cien UTXOs de pequeño valor como entradas y solo una salida, pagando a la editorial de libros. Una transacción con tantas entradas podrá ser más grande que un kilobyte, quizás de varios kilobytes en tamaño. Por lo tanto requerirá una comisión para la transacción mucho mayor que una transacción de tamaño mediano.

La aplicación de cartera de Eugenia calculará la comisión adecuada midiendo el tamaño de la transacción y multiplicándolo por el valor adecuado de comisión por kilobyte. Muchas carteras pagan comisiones más altas de lo necesario para transacciones muy grandes para asegurarse de que la transacción sea procesada rápidamente. La comisión elevada no es porque Eugenia esté gastando más dinero, sino porque su transacción es más compleja y más grande en tamaño—la comisión es independiente del valor en bitcoin de la transacción.

Scripts de Transacción y Lenguaje de Script

El lenguaje de script de las transacciones bitcoin, llamado *Script*, es un lenguaje basado en la ejecución en la pila con notación polaca inversa, similar a Forth. Si eso te suena como un trabalenguas, probablemente no habrás estudiado lenguajes de programación de la década de 1960, pero no hay de qué preocuparse— lo explicaremos todo en este capítulo. Tanto el script de bloqueo colocado en una UTXO como el script de desbloqueo están escritos en este lenguaje de scripts. Cuando se valida una transacción, el script de desbloqueo en cada entrada se ejecuta junto con el script de bloqueo correspondiente para ver si satisface la condición de gasto.

Script es un lenguaje muy simple que fue diseñado para ser de alcance limitado y ejecutable en un amplio rango de hardware, tal vez tan simple como un dispositivo integrado. Requiere procesamiento mínimo y no puede hacer muchas de las cosas sofisticadas que los lenguajes modernos sí pueden. En el caso del dinero programable, esto es una medida deliberada de seguridad.

Hoy en día, la mayoría de las transacciones procesadas a través de la red bitcoin tienen el formato "Pago a la dirección bitcoin de Bob" y se basan en un script denominado Pay-to-Public-Key-Hash (es decir, Pagar-al-Hash-de-una-Llave-Pública). Sin embargo, las transacciones bitcoin no se limitan a la secuencia de comandos "Pago a la dirección bitcoin de Bob". De hecho, los scripts de bloqueo se pueden escribir para expresar una gran variedad de condiciones complejas. Para comprender estos scripts más complejos, primero debemos entender los conceptos básicos de los scripts de transacción y el lenguaje de script.

En esta sección, demostraremos los componentes básicos del lenguaje de script de las transacciones bitcoin y mostraremos cómo se pueden usar para expresar condiciones simples para gastar y cómo se pueden cumplir esas condiciones mediante el desbloqueo de scripts.

> **TIP** La validación de transacciones bitcoin no se basa en un patrón estático, sino que se consigue a través de la ejecución de un lenguaje de scripts. Este lenguaje permite expresar una variedad casi infinita de condiciones. Así es cómo bitcoin adquiere el poder de "dinero programable".

Incompletitud de Turing

El lenguaje de script de transacciones bitcoin contiene muchos operadores, pero se encuentra deliberadamente limitado en una forma importante—no tiene la capacidad de realizar bucles ni controles de flujo complejos más allá de los controles de flujo condicionales. Esto asegura que el lenguaje no es *Turing Completo*, lo cual significa que los scripts tienen complejidad limitada y tiempos de ejecución predecibles. El lenguaje script no es de propósito general. Estas limitaciones aseguran que el lenguaje no se pueda usar para crear un bucle infinito u otras formas de "bombas lógicas" que pudieran ser incrustadas en una transacción causando un ataque de denegación de servicio contra la red bitcoin. Recuérdese que cada transacción es validada por cada nodo completo en la red bitcoin. Un lenguaje limitado previene que el mecanismo de validación de transacciones pueda suponer una vulnerabilidad.

Verificación Sin Estado

El lenguaje de script de transacciones bitcoin no tiene estados en el sentido de que no existe un estado previo a la ejecución del script, o un estado almacenado después de la ejecución del script. Por lo tanto, toda la información necesaria para ejecutar el script se encuentra contenida en el mismo script. Un script se ejecutará predeciblemente de la misma

forma en cualquier sistema. Si tu sistema verifica un script, puedes estar seguro que cualquier otro sistema en la red bitcoin también verificará ese script, lo cual significa que una transacción es válida para todos y todos lo saben. Esta predictibilidad en los resultados es un beneficio esencial del sistema bitcoin.

Construcción de Scripts (Bloqueo + Desbloqueo)

El motor de validación de transacciones de bitcoin depende de dos tipos de scripts para validar transacciones: un script de bloqueo (locking script) y un script de desbloqueo (unlocking script)

Un script de bloqueo es una condición de gasto que se coloca en una salida: especifica las condiciones que deben cumplirse para gastar la salida en el futuro. Históricamente, el script de bloqueo se llamaba *scriptPubKey*, porque generalmente contenía una llave pública o una dirección bitcoin (el hash de esa llave pública). En este libro nos referimos a él como un "script de bloqueo" para reconocer el rango más amplio de posibilidades de esta tecnología del lenguaje script. En la mayoría de las aplicaciones de bitcoin, lo que llamamos un script de bloqueo aparecerá en el código fuente como scriptPubKey. Encontrarás que al script de bloqueo también se le denomina *script testigo* (en inglés, "witness script") (ver Segregated Witness (Testigos Segregados)) o más generalmente como *acertijo criptográfico*. Todos estos términos significan lo mismo, en diferentes niveles de abstracción.

Un script de desbloqueo es un script que "resuelve", o satisface, las condiciones puestas en una salida por un script de bloqueo y permite que la salida se gaste. Los scripts de desbloqueo forman parte de cada entrada de transacción. La mayoría de las veces contienen una firma digital producida por la cartera del usuario a partir de su llave privada. Históricamente, el script de desbloqueo se llamaba *scriptSig*, porque generalmente contenía una firma digital. En la mayoría de las aplicaciones de bitcoin, el código fuente se refiere al script de desbloqueo como scriptSig. También verás que al script de desbloqueo se le denomina *testigo* (en inglés, "witness") (ver Segregated Witness (Testigos Segregados)). En este libro, nos referimos a él como un "script de desbloqueo" para reconocer el espectro mucho más amplio de requisitos de los scripts de bloqueo, ya que no todos los scripts de desbloqueo requieren firmas.

Cada nodo de validación de bitcoin validará las transacciones ejecutando a la vez los scripts de bloqueo y de desbloqueo. Cada entrada contiene un script de desbloqueo y se refiere a un UTXO preexistente. El software de validación copiará el script de desbloqueo, recuperará el UTXO al que hace referencia la entrada y copiará el script de bloqueo de ese UTXO. El script de desbloqueo y de bloqueo se ejecutan en secuencia. La entrada es válida si el script de desbloqueo cumple con las condiciones del script de bloqueo (ver Ejecución por separado de los scripts de desbloqueo y de bloqueo). Todas las entradas se validan de forma independiente, como parte de la validación general de la transacción.

Ten en cuenta que la UTXO se registra permanentemente en la cadena de bloques, y por lo tanto es invariable y no se ve afectado por los intentos fallidos de gastarlo en una nueva transacción. Solo una transacción válida que satisfaga correctamente las condiciones de la salida hace que la salida se considere "gastada" y se elimine del set de salidas de transacción no gastadas (set UTXO).

Combinando scriptSig y scriptPubKey para evaluar un script de transacción es un ejemplo de los scripts de desbloqueo y bloqueo para el tipo más común de transacción bitcoin (un pago a un hash de llave pública), mostrando el script combinado que resulta de la concatenación de los scripts de desbloqueo y de bloqueo previo a la validación del script.

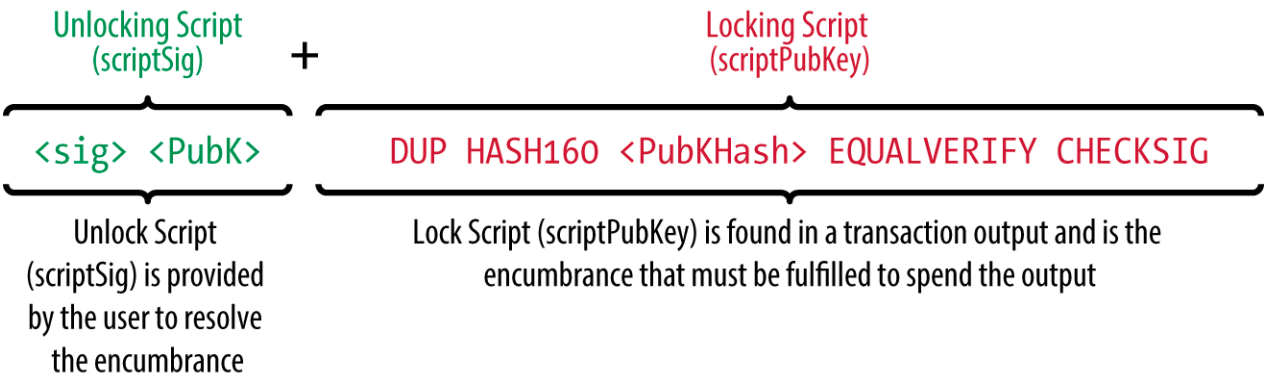


Figure 39. Combinando scriptSig y scriptPubKey para evaluar un script de transacción

La pila para la ejecución de scripts

El lenguaje de scripting de bitcoin es un lenguaje basado en la pila porque utiliza una estructura de datos llamada *pila* (en inglés, "stack"). Una pila es una estructura de datos muy simple que se puede visualizar como una pila de cartas. Una pila permite dos operaciones: push (empujar) y pop (eliminar). Push agrega un elemento en la parte superior de la pila. Pop elimina el elemento superior de la pila. Las operaciones en una pila solo pueden actuar en el elemento superior de la pila.

Una estructura de datos de pila también se llama una cola "LIFO" en la que el "último en entrar, es el primero en salir" (en inglés, Last-In-First-Out).

El lenguaje de script ejecuta el script procesando cada ítem de izquierda a derecha. Los números (valores de data constante) son empujados a la pila. Los operadores empujan o sacan uno o más parámetros de la pila, actúan sobre ellos, y pueden empujar un resultado a la pila. Por ejemplo, OP_ADD sacará dos elementos de la pila, los sumará, y luego empujará la suma resultante a la pila.

Los operadores condicionales evalúan una condición, produciendo un resultado booleano de VERDADERO o FALSO. Por ejemplo, OP_EQUAL saca dos elementos de la pila y empuja VERDADERO (VERDADERO es representado por el número 1) si son iguales y FALSO (representado por cero) si no son iguales. Los scripts de transacción bitcoin usualmente contienen un operador condicional, de forma que puedan producir el valor VERDADERO que significa que la transacción es válida.

Un script simple

Ahora apliquemos lo que hemos aprendido sobre scripts y pilas a algunos ejemplos simples.

En El script de validación de bitcoin haciendo matemática simple, el script 2 3 OP_ADD 5 OP_EQUAL muestra el operador de adición aritmética OP_ADD, el cual suma dos números y coloca el resultado en la pila, seguido por el operador condicional OP_EQUAL, el cual verifica que el resultado de la suma sea igual a 5. Para ser concisos, el prefijo OP_ se omite en el ejemplo paso-a-paso. Para más detalles sobre los operadores y funciones disponibles en los scripts, ver Operadores, Constantes y Símbolos del Lenguaje de Script de Transacción.

Aunque la mayoría de los scripts de bloqueo se refieren al hash de una llave pública (esencialmente, una dirección bitcoin), y por lo tanto requieren una prueba de titularidad para gastar los fondos, el script no necesita ser tan complicado. Cualquier combinación de scripts de bloqueo y desbloqueo que resulte en VERDADERO será válido. La aritmética simple que usamos como ejemplo de lenguaje de script es también un script de bloqueo válido que puede usarse para bloquear una salida de transacción.

Usar parte del script del ejemplo aritmético como el script de bloqueo:

```
3 OP_ADD 5 OP_EQUAL
```

lo cual puede ser satisfecho por una transacción que contenga una entrada con el script de desbloqueo:

```
2
```

El software de validación combina los scripts de bloqueo y desbloqueo y el script resultante es:

```
2 3 OP_ADD 5 OP_EQUAL
```

Como vimos en el ejemplo paso-a-paso en El script de validación de bitcoin haciendo matemática simple, cuando se ejecuta el script, el resultado es OP_TRUE, haciendo a la transacción válida. No solo es esto un script de bloqueo de salida de transacción válido, sino que el UTXO resultante puede gastarse por cualquiera con la habilidad aritmética para saber que el número 2 satisface el script.

TIP

Las transacciones son válidas si el resultado en el tope de la pila es VERDADERO (notado como {0x01}), cualquier valor distinto de cero o si la pila se encuentra vacía luego de la ejecución del script. Las transacciones son inválidas si el valor en lo alto de la pila es FALSO (un valor vacío de longitud cero, notado como {}), o si la ejecución del script se detiene explícitamente por un operador, tal como OP_VERIFY, OP_RETURN, o un cierre de condicional como OP_ENDIF. Ver Operadores, Constantes y Símbolos del Lenguaje de Script de Transacción para más detalles.

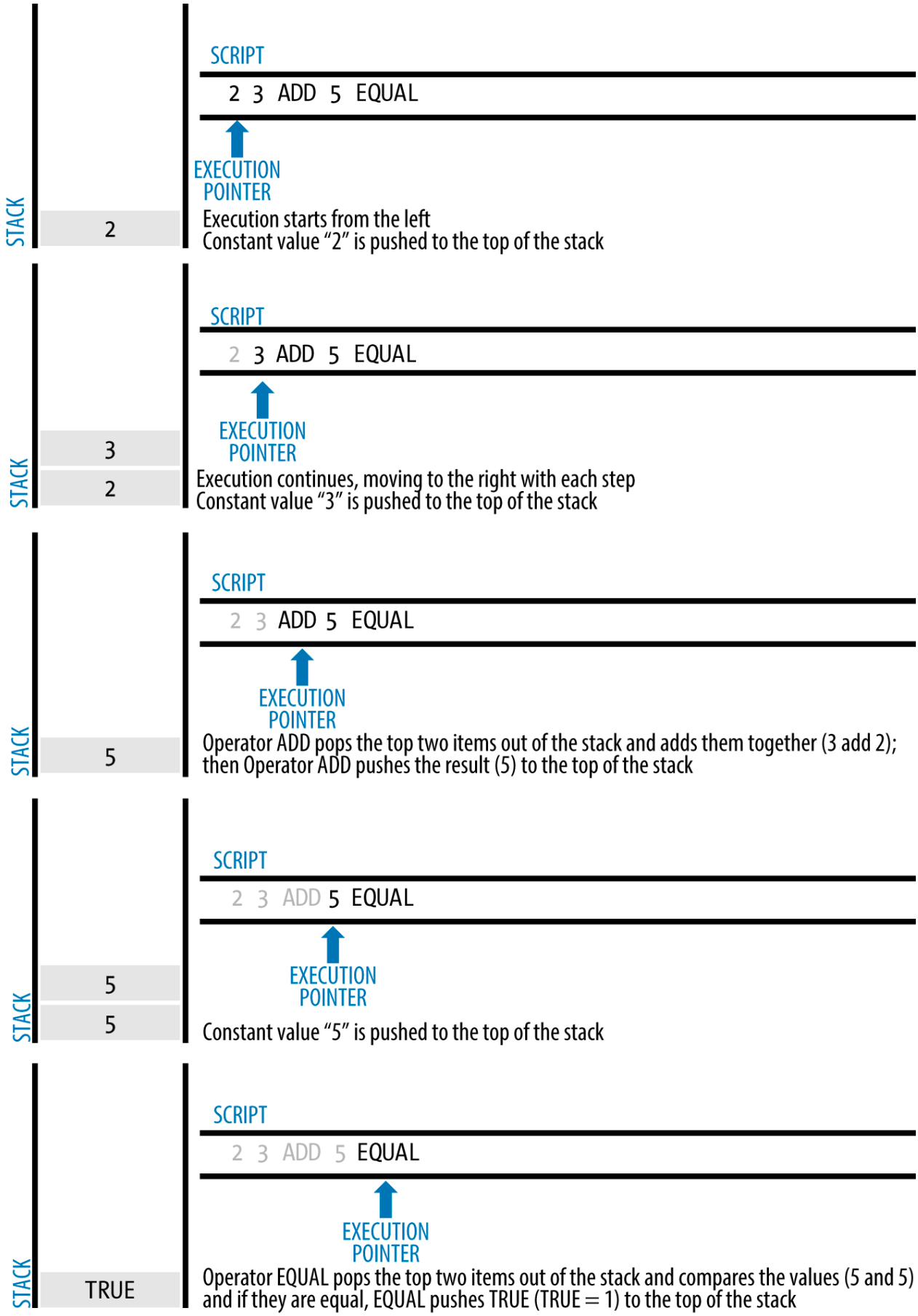


Figure 40. El script de validación de bitcoin haciendo matemática simple

El siguiente es un script un poco más complejo, que calcula 2 + 7 - 3 + 1 . Tenga en cuenta que cuando el script contiene varios operadores en una fila, la pila permite que el siguiente operador actúe sobre los resultados de un operador:

```
2 7 OP_ADD 3 OP_SUB 1 OP_ADD 7 OP_EQUAL
```

Intenta validar el script previo tú mismo usando papel y lápiz. Cuando la ejecución del script acabe, deberías terminar con el valor VERDADERO en la pila.

Ejecución por separado de los scripts de desbloqueo y de bloqueo

En el cliente bitcoin original, los scripts de bloqueo y de desbloqueo se concatenaban y ejecutaban en secuencia. Por razones de seguridad esto se cambió en 2010, debido a una vulnerabilidad que permitía que un script de desbloqueo mal formado enviara datos a la pila y corrompiera el script de bloqueo. En la implementación actual los scripts se ejecutan de forma separada y la pila se transfiere entre las dos ejecuciones, como se describe a continuación.

Primero, se ejecuta el script de desbloqueo, utilizando el máquina de ejecución de la pila. Si la secuencia de comandos del script de desbloqueo se ejecuta sin errores (por ejemplo, no le quedan operadores "colgantes"), la pila principal se copia y se ejecuta la secuencia de comandos de bloqueo. Si el resultado de ejecutar el script de bloqueo con los datos de la pila copiados del script de desbloqueo es "VERDADERO", el script de desbloqueo ha logrado resolver las condiciones impuestas por el script de bloqueo y, por lo tanto, la entrada es una autorización válida para gastar la UTXO . Si después de la ejecución del script combinado permanece cualquier resultado que no sea "VERDADERO" , la entrada no es válida porque no ha cumplido con las condiciones de gasto colocadas en el UTXO.

Pagar-al-Hash-de-una-Llave-Pública (P2PKH)

La gran mayoría de las transacciones procesadas en la red bitcoin gastan salidas bloqueadas por un script de pago-al-hash-de-una-llave-pública o "P2PKH". Estas salidas contienen un script de bloqueo que bloquea la salida para un hash de llave pública, más comúnmente conocida como una dirección bitcoin. Una salida bloqueada por un script P2PKH se puede desbloquear (gastar) presentando una llave pública y una firma digital creada por la llave privada correspondiente (ver Firmas Digitales (ECDSA)).

Por ejemplo, veamos nuevamente el pago de Alice al Café de Bob. Alice hizo un pago de 0.015 bitcoin a la dirección bitcoin del café. Esa salida de transacción tendría un script de bloqueo del tipo:

```
OP_DUP OP_HASH160 <Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

El Hash de Llave Pública del Café es equivalente a la dirección bitcoin del café, sin la codificación Base58Check. La mayoría de las aplicaciones mostrarían el *hash de llave pública* en codificación hexadecimal y no el familiar formato Base58Check de la dirección bitcoin comenzado en "1."

El script de bloqueo anterior puede ser satisfecho con un script de desbloqueo de la forma:

```
<Cafe Signature><Cafe Public Key>
```

Los dos scripts juntos formarían el siguiente script de validación combinado:

```
<Cafe Signature> <Cafe Public Key> OP_DUP OP_HASH160
<Cafe Public Key Hash> OP_EQUALVERIFY OP_CHECKSIG
```

Cuando es ejecutado, este script combinado será evaluado a VERDADERO si, y solo si, el script de desbloqueo cumple las condiciones establecidas por el script de bloqueo. En otras palabras, el resultado será VERDADERO si el script de desbloqueo contiene una firma válida proveniente de la llave privada del café que corresponde al hash de llave pública establecido como obstrucción.

Las figuras #P2PubKHash1 y #P2PubKHash2 muestran (en dos partes) una ejecución paso a paso del script combinado, que probará que esta es una transacción válida.

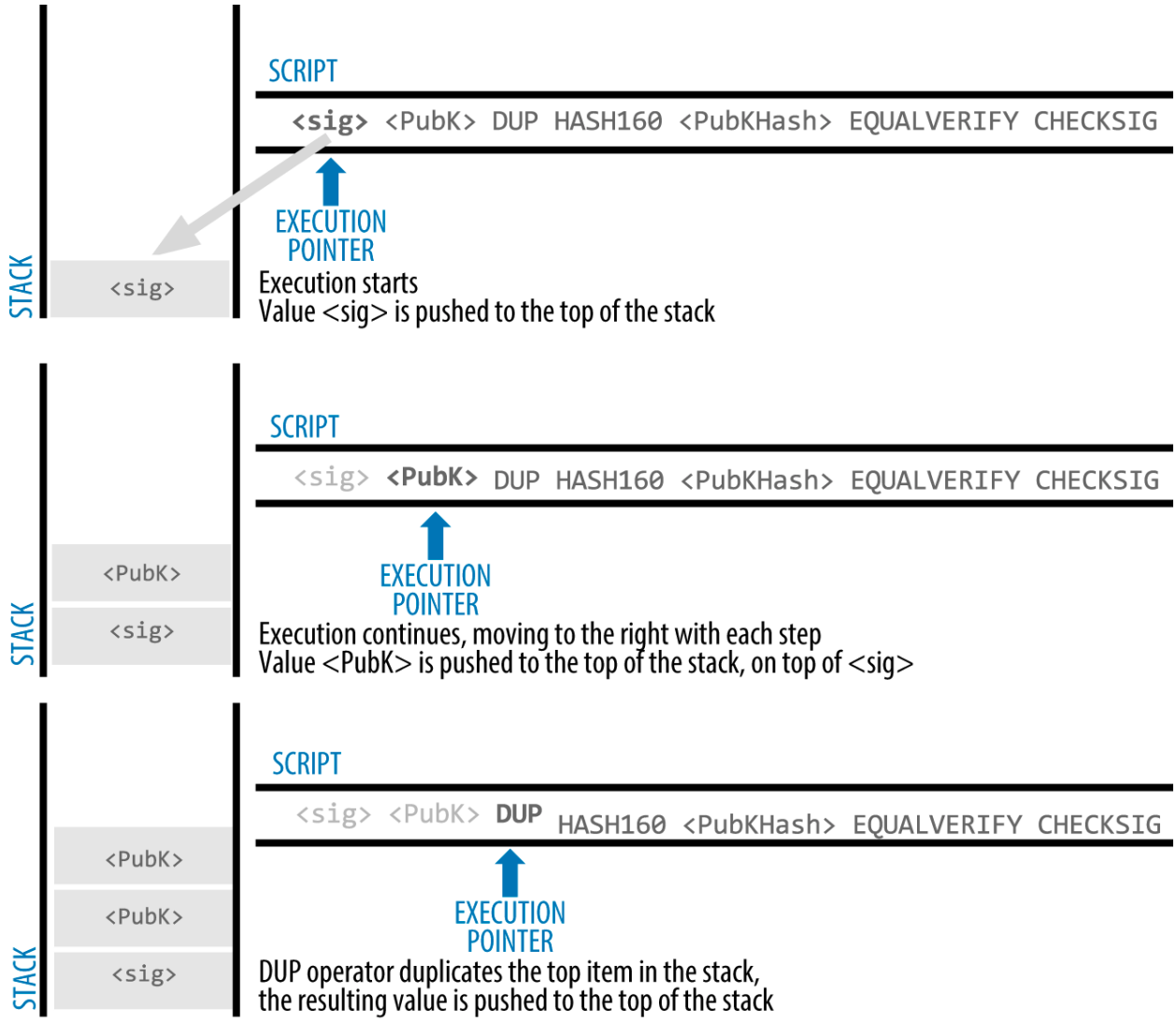


Figure 41. Evaluando un script de una transacción P2PKH (parte 1 de 2)

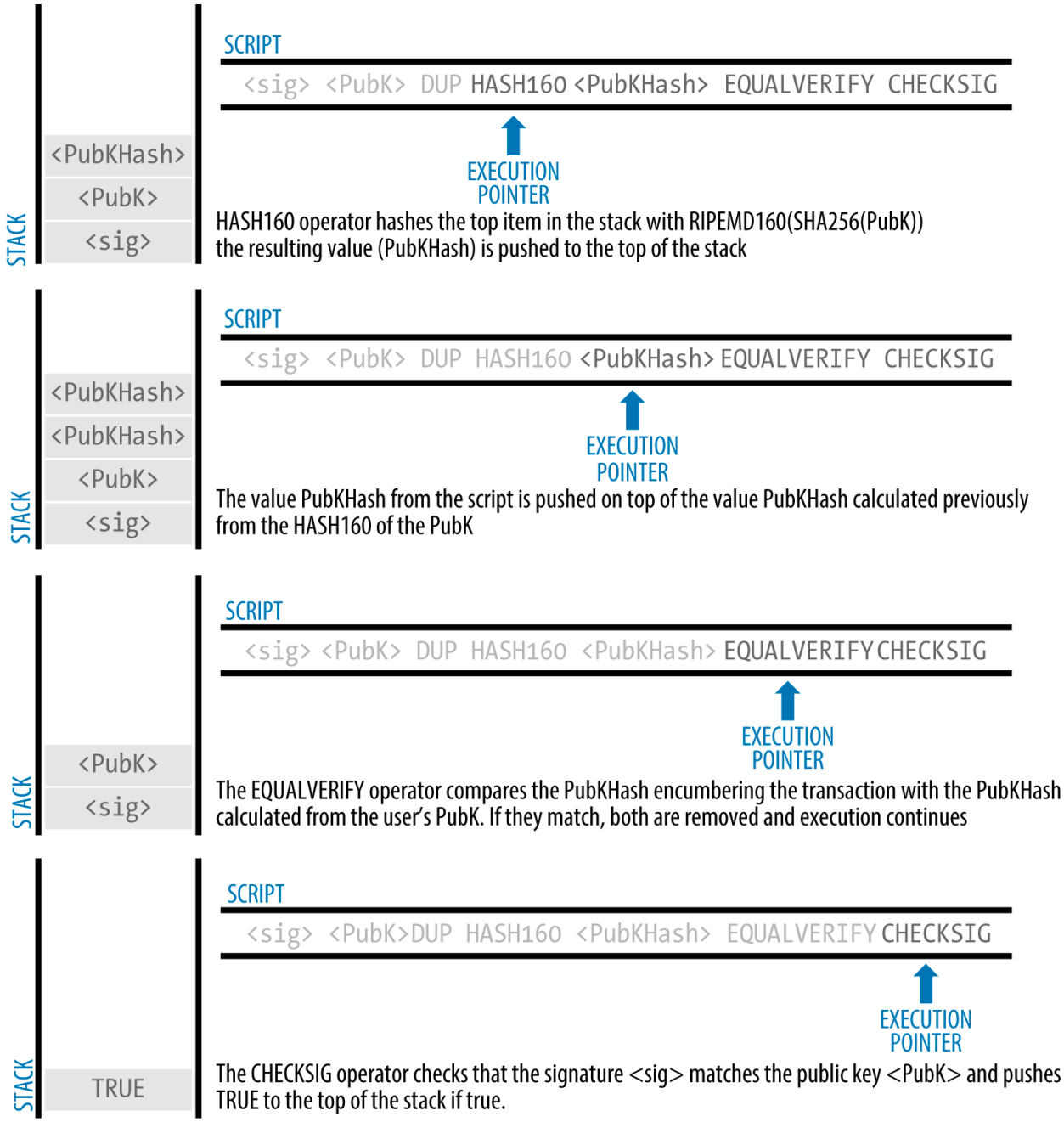


Figure 42. Evaluando un script de una transacción P2PKH (parte 2 de 2)

Firmas Digitales (ECDSA)

Hasta ahora, no hemos profundizado en ningún detalle sobre las "firmas digitales". En esta sección veremos cómo funcionan las firmas digitales y cómo pueden presentar una prueba de titularidad de una llave privada sin revelar esa llave privada.

El algoritmo de firma digital utilizado en bitcoin es el *Algoritmo de Firma Digital de Curva Elíptica* (en inglés, Elliptic Curve Digital Signature Algorithm) o *ECDSA*. ECDSA es el algoritmo utilizado para firmas digitales basadas en pares de llaves privada/pública de curva elíptica, como se describe en Criptografía de Curva Elíptica Explicada. Se usa ECDSA en las funciones de script OP_CHECKSIG, OP_CHECKSIGVERIFY, OP_CHECKMULTISIG, y OP_CHECKMULTISIGVERIFY. Cada vez que los veas en un script de bloqueo, el script de desbloqueo debe contener una firma ECDSA.

Una firma digital sirve para tres propósitos en bitcoin (consulta la siguiente barra lateral). Primero, la firma prueba que el propietario de la llave privada, quien es, en consecuencia, el propietario de los fondos, ha autorizado el gasto de esos fondos. En segundo lugar, constituye una prueba de autorización que es *innegable* (no repudiable). En tercer lugar, la firma prueba que la transacción (o partes específicas de la misma) no ha sido modificada por nadie después de haber sido firmada.

Ten en cuenta que cada entrada de una transacción se firma de forma independiente. Esto es crítico, ya que ni las firmas ni las entradas tienen por qué pertenecer o firmarse por un mismo "propietario". De hecho, un esquema de transacción

específico llamado "CoinJoin" usa este hecho para crear transacciones multipartitas con fines de privacidad.

NOTE

Cada entrada de transacción y cualquier firma que pueda contener es *completamente* independiente de cualquier otra entrada o firma. Múltiples partes pueden colaborar para construir transacciones y firmar solo una entrada cada una.

Definición de "Firma Digital" en Wikipedia

Una firma digital es un esquema matemático para demostrar la autenticidad de un mensaje o documentos digitales. Una firma digital válida le da al destinatario razones para creer que el mensaje fue creado por un remitente conocido (autenticación), que el remitente no puede negar haber enviado el mensaje (no repudiable) y que el mensaje no se modificó en tránsito (integridad).

Fuente: https://en.wikipedia.org/wiki/Digital_signature

Cómo Funcionan las Firmas Digitales

Una firma digital es un *esquema matemático* que consta de dos partes. La primera parte es un algoritmo para crear una firma, utilizando una llave privada (la llave de la firma), y un mensaje (la transacción). La segunda parte es un algoritmo que permite a cualquier persona verificar la firma, dándose a conocer el mensaje y la correspondiente llave pública.

Creando una firma digital

En la implementación del algoritmo ECDSA de bitcoin, el "mensaje" que se firma es la transacción, o más precisamente un hash de un subconjunto específico de los datos de la transacción (ver Tipos de Hash de Firma (SIGHASH)). La llave de la firma es la llave privada del usuario. El resultado es la firma:

\(\(Sig = F_{sig}(F_{hash}(m), dA)\)\)

donde:

- *dA* es la llave privada que firma
- *m* es la transacción (o partes de ella)
- F_{hash} es la función hash empleada
- F_{sig} es el algoritmo de firma
- *Sig* es la firma resultante

Puedes encontrar más detalles sobre las matemáticas de la ECDSA en Matemáticas ECDSA.

La función F_{sig} produce una firma Sig que se compone de dos valores, referidos comúnmente como R y S:

```
Sig = (R, S)
```

Ahora que se han calculado los dos valores R y S, se serializan en un hilo de bytes mediante un esquema de codificación estándar internacional llamado "Reglas de Codificación Distinguidas" o del inglés: *Distinguished Encoding Rules*, o *DER*.

Serialización de firmas (DER)

Veamos de nuevo la transacción que creó Alice. En la entrada de la transacción hay un script de desbloqueo que contiene la siguiente firma codificada en DER desde la cartera de Alice:

```
3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863e
a8f53982c09db8f6e381301
```

Esa firma es un hilo de bytes serializado con los valores R y S producidos por la cartera de Alice para demostrar que posee la llave privada autorizada para gastar esa salida. El formato de serialización consta de nueve elementos como sigue:

- 0x30—indicando el principio de la secuencia DER
- 0x45—la longitud de la secuencia (69 bytes)

- 0x02—sigue un valor entero
- 0x21—la longitud del entero (33 bytes)
- R—00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
- 0x02—sigue otro entero
- 0x20—la longitud del entero (32 bytes)
- S—4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
- Un sufijo (0x01) indicando el tipo de hash usado (SIGHASH_ALL)

Prueba a ver si puedes decodificar la firma serializada de Alice (codificada en DER) usando esta lista. Los números importantes son R y S; El resto de los datos son parte del esquema de codificación DER.

Verificando la Firma

Para verificar la firma, debes tener la firma (R y S), la transacción serializada y la llave pública (que corresponde a la llave privada utilizada para crear la firma). Esencialmente, la verificación de una firma significa "Sólo el propietario de la llave privada que generó esta llave pública podría haber producido esta firma en esta transacción".

El algoritmo de verificación de firma toma el mensaje (un hash de la transacción o partes de la misma), la llave pública del firmante y la firma (valores R y S), y devuelve VERDADERO si la firma es válida para este mensaje y llave pública.

Tipos de Hash de Firma (SIGHASH)

Las firmas digitales se aplican a los mensajes, que en el caso de bitcoin, son las transacciones en sí mismas. La firma implica un *compromiso* por parte del firmante con los datos específicos de la transacción. En la forma más simple, la firma se aplica a toda la transacción, confirmando así todas las entradas, salidas y otros campos de la transacción. Sin embargo, una firma puede comprometerse a solo un subconjunto de los datos de una transacción, lo cual es útil en varios escenarios como veremos en esta sección.

Las firmas de bitcoin tienen una forma de indicar qué parte de los datos de una transacción se incluyen en el hash firmado por la llave privada mediante un indicador SIGHASH. El indicador SIGHASH es un byte único que se agrega a la firma. Todas las firmas tienen un indicador SIGHASH y el indicador puede ser diferente de entrada a entrada. Una transacción con tres entradas firmadas puede tener tres firmas con diferentes indicadores SIGHASH, cada firma firmando (o comprometiendo) diferentes partes de la transacción.

Recuérdese, que cada entrada puede contener una firma con su propio script de desbloqueo. Como resultado, una transacción que contiene varias entradas puede tener firmas con diferentes indicadores SIGHASH que comprometen diferentes partes de la transacción en cada una de las entradas. Téngase en cuenta también que las transacciones bitcoin pueden contener entradas de diferentes "propietarios", que pueden firmar solo una entrada en una transacción construida de manera parcial e inválida, colaborando con otros titulares para reunir todas las firmas necesarias para realizar una transacción válida. Muchos de los tipos de indicador SIGHASH solo tienen sentido si se piensa en varios participantes que colaboran fuera de la red bitcoin y que actualizan una transacción parcialmente firmada.

Existen tres indicadores SIGHASH: ALL, NONE, y SINGLE, como se muestra en Tipos SIGHASH y sus significados.

Table 18. Tipos SIGHASH y sus significados

Indicador SIGHASH	Valor	Descripción
ALL	0x01	Firma que se aplica a todas las entradas y todas las salidas
NONE	0x02	Firma aplicada a todas las entradas, pero no toma en cuenta a ninguna de las salidas
SINGLE	0x03	Firma aplicada a todas las entradas pero solo toma en cuenta a aquella salida con el mismo número de índice que la entrada firmada

Además, hay un indicador modificador SIGHASH_ANYONECANPAY, que se puede combinar con cada uno de los

indicadores anteriores. Cuando se establece ANYONECANPAY, solo se firma una entrada, dejando el resto (y sus números de secuencia) abiertos para su modificación. El ANYONECANPAY tiene el valor 0x80 y se aplica mediante OR en modo bit, lo que da como resultado los indicadores combinados como se muestran en Tipos SIGHASH con modificadores y sus significados.

Table 19. Tipos SIGHASH con modificadores y sus significados

Indicador SIGHASH	Valor	Descripción
ALL|ANYONECANPAY	0x81	Firma aplica a una sola entrada pero para todas las salidas
NONE|ANYONECANPAY	0x82	Firma aplica a una sola entrada, sin tomar en cuenta a ninguna de las salidas
SINGLE|ANYONECANPAY	0x83	Firma aplica a una sola entrada y sólo a la salida con el mismo número de índice que la entrada que esta firmando

Estas combinaciones de indicadores se resumen en Resumen de diferentes combinaciones de indicadores sighash.

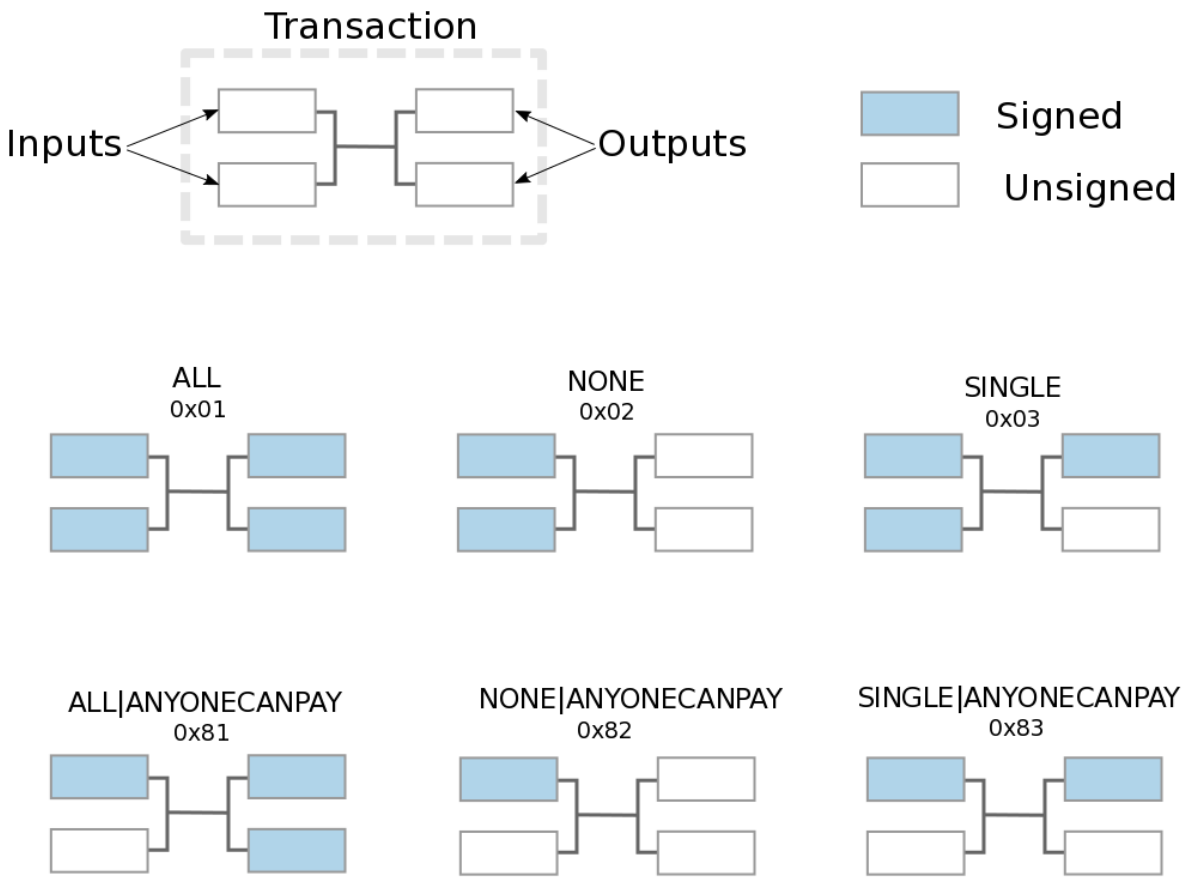


Figure 43. Resumen de diferentes combinaciones de indicadores sighash

La forma en que se aplican los indicadores SIGHASH durante la firma y su verificación es realizando una copia de la transacción y truncando ciertos campos de ésta (se establece la longitud a cero y se vacía). La transacción resultante es serializada. El indicador SIGHASH se agrega al final de la transacción serializada y se calcula el valor hash a este resultado. El hash en sí es el "mensaje" que se está firmado. Dependiendo del indicador SIGHASH que se use, se truncarán diferentes partes de la transacción. El hash resultante depende de los diferentes subconjuntos de los datos de la transacción. Al incluir el SIGHASH como último paso antes del hash, la firma también confirma el tipo SIGHASH, por lo que no puede cambiarse (por ejemplo, por un minero) .

NOTE

Todos los tipos SIGHASH firman el campo nLocktime de la transacción (ver Bloqueo Temporal de Transacción (nLocktime)). Además, el tipo SIGHASH en sí mismo se añade a la transacción antes de que ésta se firme, por lo que no se puede modificar una vez que se haya firmado.

En el ejemplo de la transacción de Alice (ver la lista en Serialización de firmas (DER)), vimos que la última parte de la firma codificada en DER era 01, que es el indicador SIGHASH_ALL. Esto bloquea los datos de la transacción, por lo que la firma de Alice está confirmando el estado de todas las entradas y salidas. Este es la forma de firma más común.

Veamos otros tipos de SIGHASH y cómo se pueden usar en la práctica:

ALL|ANYONECANPAY

Esta construcción se puede usar para realizar una transacción de tipo "crowdfunding". Alguien que intente recaudar fondos puede construir una transacción con una sola salida. La única salida paga la cantidad "objetivo" al recaudador de fondos. Semejante transacción obviamente no es válida, ya que no tiene las suficientes entradas. Sin embargo, otras personas ahora pueden modificarla agregando entradas propias, como donación. Firman su propia entrada con ALL|ANYONECANPAY. A menos que se recopilen suficientes entradas para alcanzar el valor de la salida, la transacción no es válida. Cada donación es una "promesa", que no puede ser reclamada por el recaudador de fondos hasta que no se acumule la cantidad objetivo total.

NONE

Esta construcción se puede usar para crear un "cheque al portador" o "cheque en blanco" de una cantidad específica. Se compromete con la entrada, pero permite cambiar el script de bloqueo de salida. Cualquiera puede escribir su propia dirección bitcoin en el script de bloqueo de salida y gastar la transacción. Sin embargo, el valor mismo de la salida estará bloqueado por la firma.

NONE|ANYONECANPAY

Esta construcción se puede usar para ensamblar un "colector de polvo". Los usuarios que tienen diminutas UTXOs en sus billeteras no pueden gastarlas sin que el costo en las comisiones sea superior al valor del polvo. Con este tipo de firma, el polvo UTXO puede ser donado para que cualquiera lo agregue y lo gaste cuando lo desee.

Hay algunas propuestas para modificar o expandir el sistema SIGHASH. Una de estas propuestas es *Bitmask Sighash Modes* de Glenn Willen de Blockstream, como parte del proyecto Elements. Esto apunta a crear un reemplazo flexible para los tipos SIGHASH que permite "máscaras de bits de entradas y salidas arbitrarias y reescribibles por minería" que pueden expresar "esquemas de compromiso contractual más complejos, como ofertas firmadas con cambio en un intercambio de activos distribuidos".

NOTE

Nadie verá los indicadores SIGHASH presentados como opción en la aplicación de cartera de un usuario. Con pocas excepciones, las carteras construyen sus scripts P2PKH y firman con los indicadores SIGHASH_ALL. Para usar un indicador SIGHASH diferente, tendríamos que escribir un software especial para construir y firmar transacciones. Más importante aún, los indicadores SIGHASH pueden ser utilizados por aplicaciones de bitcoin para propósitos especiales que permiten usos novedosos.

Matemáticas ECDSA

Como se mencionó anteriormente, las firmas se crean mediante una función matemática F_{sig} que produce una firma compuesta por dos valores R y S. En esta sección veremos la función F_{sig} con más detalle.

El algoritmo de firma genera primero un par de llaves público-privadas *fugaces* (temporales). Este par de llaves temporales se usan en el cálculo de los valores R y S, después de una transformación que involucra la llave privada de la firma y el hash de transacción.

El par de llaves temporales se basan en un número aleatorio k, que se utiliza como llave privada temporal. Desde k, generamos la correspondiente llave pública temporal P (calculada como $P = k*G$, de la misma manera en que se derivan las llaves públicas de bitcoin; ver Llaves Públicas). El valor R de la firma digital es entonces la coordenada x de la llave pública efímera P.

A partir de ahí, el algoritmo calcula el valor S de la firma, de manera que:

$S = k^{-1} (Hash(m) + dA * R) \ mod \ n$

donde:

- k es la llave privada efímera
- R es la coordenada x de la llave pública efímera

- *dA* es la llave privada que firma
- *m* son los datos de la transacción
- *n* es el orden de núero primo de la curva elíptica

La verificación es la inversa de la función de generación de firmas, utilizando los valores *R*, *S* y la llave pública para calcular un valor *P*, que es un punto en la curva elíptica (la llave pública efímera utilizada en la creación de la firma):

$P = S^{-1} * Hash(m) * G + S^{-1} * R * Qa$

donde:

- *R* y *S* son los valores de la firma
- *Qa* es la llave pública de Alice
- *m* son los datos de la transacción que ha sido firmada
- *G* es el punto generador de la curva elíptica

Si la coordenada x del punto calculado *P* es igual a *R*, entonces el verificador puede concluir que la firma es válida.

Téngase en cuenta que al verificar la firma, la llave privada no se conoce ni se revela.

TIP

ECDSA es necesariamente una parte bastante complicada de las matemáticas; Una explicación completa está fuera del alcance de este libro. Una serie de excelentes guías en línea pueden orientar al lector paso a paso: búsquese "ECDSA explained" o consúltese: http://bit.ly/2r0HhGB.

La Importancia de la Aleatoriedad en las Firmas

Como vimos en Matemáticas ECDSA, el algoritmo de generación de firmas utiliza una llave aleatoria *k*, como base para generar un par de llaves público/privada efímeras. El valor de *k* no es importante, *siempre que sea aleatorio*. Si se usa el mismo valor *k* para producir dos firmas con diferentes mensajes (transacciones), entonces la *llave privada* firmante podría ser calculada por cualquier persona. ¡La reutilización del mismo valor para *k* en un algoritmo de firma conduce a la exposición de la llave privada!

WARNING

Si se usa el mismo valor *k* en el algoritmo de firma en dos transacciones diferentes, ¡la llave privada se puede calcular y puede quedar expuesta al mundo!

Esto no es solo una posibilidad teórica. Hemos visto que este problema lleva a exponer las llaves privadas en algunas implementaciones diferentes de algoritmos de firma de transacciones en bitcoin. Se han robado fondos debido a la reutilización involuntaria de un valor *k*. La razón más común para la reutilización de un valor *k* es un generador de números aleatorios incorrectamente inicializado.

Para evitar esta vulnerabilidad, la mejor práctica de la industria es no generar *k* con un generador de números aleatorios inicializados con entropía, sino usar un proceso aleatorio determinista inicializado con los datos de la transacción en sí. Esto asegura que cada transacción produce un *k* diferente. El algoritmo estándar de la industria para la inicialización determinista de *k* se define en RFC 6979, publicado por Internet Engineering Task Force.

Si estás implementando un algoritmo para firmar transacciones en bitcoin, debes usar RFC 6979 o un algoritmo aleatorio determinista similar para asegurarte de generar un *k* diferente para cada transacción.

Direcciones Bitcoin, Saldos y Otras Abstracciones

Comenzaremos este capítulo con el descubrimiento de que las transacciones se ven muy diferentes "detrás del telón" en comparación con la forma en que se presentan en carteras, exploradores de la cadena de bloques y otras aplicaciones orientadas al usuario. Muchos de los conceptos simplistas y familiares de los capítulos anteriores, como las direcciones y saldos de bitcoin, parecen estar ausentes de la estructura de una transacción. Vimos que las transacciones no contienen direcciones bitcoin, de por sí, sino que operan a través de scripts que bloquean y desbloquean valores discretos de bitcoin. Los saldos no están presentes en ninguna parte de este sistema y, sin embargo, cada aplicación de cartera muestra de manera destacada el saldo de la cartera del usuario.

Ahora que hemos explorado lo que realmente se incluye en una transacción bitcoin, podemos examinar cómo las

abstracciones de alto nivel se derivan de los componentes aparentemente primitivos de la transacción.

Veamos de nuevo cómo la transacción de Alice se presentó en un explorador de bloques popular (Transacción de Alice a la Cafetería de Bob).

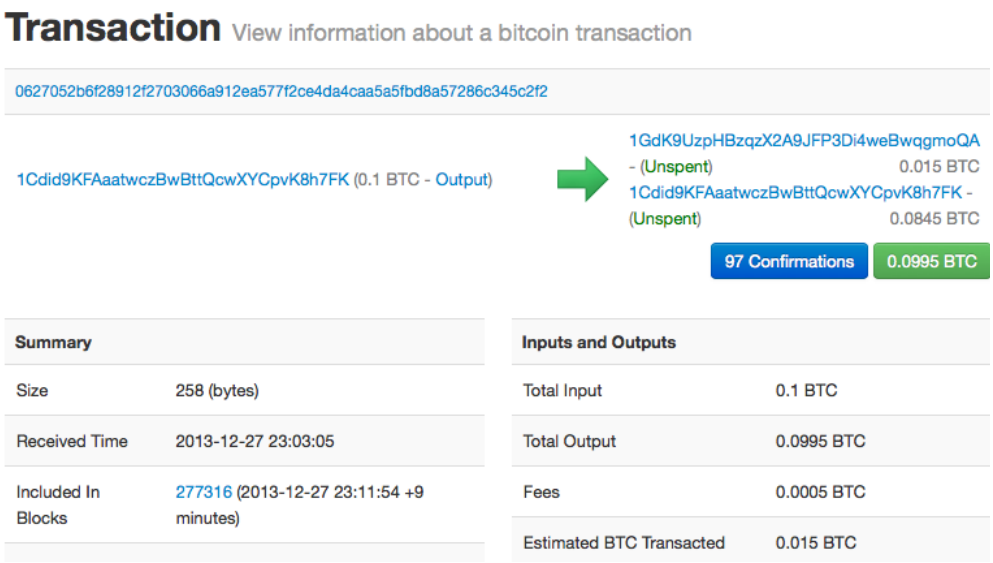


Figure 44. Transacción de Alice a la Cafetería de Bob

En el lado izquierdo de la transacción, el explorador de la cadena de bloques muestra la dirección bitcoin de Alice como el "remitente". De hecho, esta información no está en la transacción en sí. Cuando el explorador de la cadena de bloques hace referencia a la transacción, también hace referencia a la transacción anterior asociada con la entrada y extrae la primera salida de esa transacción anterior. Dentro de esa salida hay un script de bloqueo que bloquea la UTXO al hash de la llave pública de Alice (un script P2PKH). El explorador de la cadena de bloques extrajo el hash de la llave pública y lo codificó utilizando la codificación Base58Check para producir y mostrar la dirección bitcoin que representa esa llave pública.

De manera similar, en el lado derecho, el explorador de la cadena de bloques muestra las dos salidas; la primera a la dirección bitcoin de Bob y la segunda a la dirección bitcoin de Alice (como cambio). Una vez más, para crear estas direcciones bitcoin, el explorador de la cadena de bloques extrajo el script de bloqueo de cada salida, lo reconoció como un script P2PKH y extrajo el hash de llave pública desde dentro. Finalmente, el explorador de la cadena de bloques recodificó ese hash de llave pública con el estándar Base58Check para producir y mostrar las direcciones bitcoin.

Si hicieras clic en la dirección bitcoin de Bob, el explorador de la cadena de bloques te mostraría la imagen en El saldo de la dirección bitcoin de Bob.

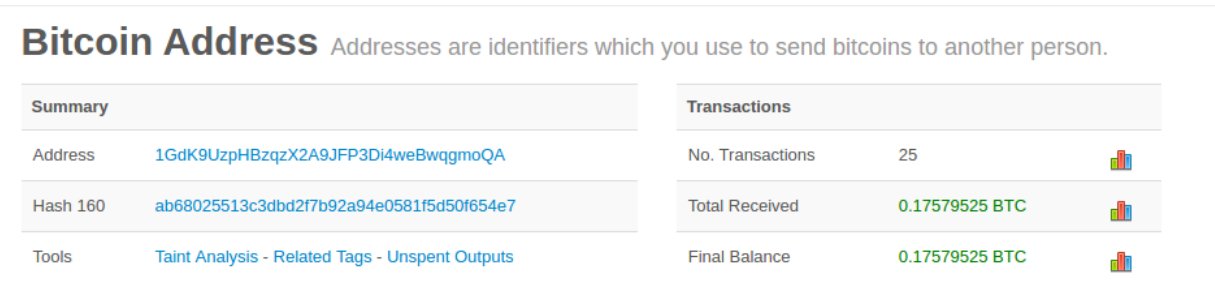


Figure 45. El saldo de la dirección bitcoin de Bob

El explorador de la cadena de bloques muestra el saldo de la dirección bitcoin de Bob. Pero en ninguna parte del sistema de bitcoin hay un concepto de "saldo". Más bien, los valores que se muestran aquí son construidos por el explorador de la cadena de bloques de la siguiente manera.

Para construir la cantidad "Total recibida", el explorador de la cadena de bloques primero decodificará la codificación Base58Check de la dirección bitcoin para recuperar el hash de 160 bits de la llave pública de Bob que está codificada dentro de la dirección. Después, el explorador de la cadena de bloques buscará en la base de datos de transacciones, intentando hallar resultados con scripts de bloqueo P2PKH que contengan el hash de llave pública de Bob. Al sumar el valor de todas las salidas, el explorador de la cadena de bloques puede mostrar el valor total recibido.

La construcción del saldo actual (que se muestra como "Saldo Final") requiere un poco más de trabajo. El explorador de la cadena de bloques mantiene una base de datos de las salidas que actualmente no se han gastado; el set UTXO. Para mantener esta base de datos, el explorador de la cadena de bloques debe monitorizar la red bitcoin, agregar las UTXOs recién creadas y eliminar las UTXOs gastadas, en tiempo real, tal como aparecen en las transacciones no confirmadas. Este es un proceso complicado que requiere realizar un seguimiento de las transacciones a medida que se propagan, así como mantener el consenso con la red bitcoin para garantizar que se siga a la cadena correcta. A veces, el explorador de la cadena de bloques se desincroniza y su visión del set UTXO es incompleta o incorrecta.

A partir del set UTXO, el explorador de la cadena de bloques suma el valor de todas las salidas sin gastar que hacen referencia al hash de la llave pública de Bob y produce el número de "Saldo Final" que se muestra al usuario.

Para producir esta imagen, con estos dos "saldos", el explorador de la cadena de bloques debe indexar y buscar entre docenas, cientos o incluso cientos de miles de transacciones.

En resumen, la información presentada a los usuarios a través de las aplicaciones de billetera, exploradores de la cadena de bloques y otras interfaces de usuario de bitcoin, a menudo se componen de abstracciones de nivel superior que se obtienen al buscar muchas transacciones diferentes, inspeccionar sus contenidos y manipular los datos ubicados en ellas. Al presentar esta visión simplista de las transacciones de bitcoin que se parecen a los cheques bancarios de un remitente a un destinatario, estas aplicaciones deben abstraer muchos detalles subyacentes. Se centran principalmente en los tipos comunes de transacciones: P2PKH con firmas SIGHASH_ALL en cada entrada. Por lo tanto, si bien las aplicaciones de bitcoin pueden exhibir a más del 80% de todas las transacciones de una manera fácil de leer, a veces se ven confundidas por transacciones que se desvían de la norma. Las transacciones que contienen scripts de bloqueo más complejos, o diferentes indicadores SIGHASH, o muchas entradas y salidas, demuestran la simplicidad y debilidad de estas abstracciones.

Todos los días, cientos de transacciones que no contienen salidas P2PKH se confirman en la cadena de bloques. Los exploradores de cadena de bloques a menudo presentan estos con mensajes de advertencia en rojo, que dicen que no pueden decodificar ninguna dirección.

Como veremos en el siguiente capítulo, estas no son necesariamente transacciones extrañas. Son transacciones que contienen scripts de bloqueo más complejos que los P2PKH comunes. Aprenderemos cómo descifrar y comprender scripts más complejos y las aplicaciones que las admiten a continuación.

Transacciones avanzadas y Scripting

Introducción

En el capítulo anterior, presentamos los elementos básicos de las transacciones de bitcoin y analizamos el tipo más común de script de transacción, el script P2PKH. En este capítulo veremos scripts más avanzados y cómo podemos usarlos para generar transacciones con condiciones complejas.

Primero, veremos los scripts *multifirma*. A continuación, examinaremos el segundo script de transacción más común, *Pay-to-Script-Hash* (en español, Paga-A-Hash-de-Script), que permite todo un mundo de scripts complejos. Luego, examinaremos los nuevos operadores de script que agregan una dimensión de tiempo a bitcoin, a través de *timelocks* (en español, *bloqueos de tiempo*). Finalmente, veremos *Segregated Witness*, un cambio arquitectónico en la estructura de las transacciones.

Multifirma

Los scripts multifirma establecen una condición en la que N llaves públicas se registran en el script y al menos M de ellas deben proporcionar firmas para desbloquear los fondos. Esto también se conoce como un esquema M-de-N, donde N es el número total de llaves y M es el umbral de firmas requerido para la validación. Por ejemplo, una multifirma 2-de-3 enumera tres llaves públicas como firmantes potenciales y al menos dos de ellas deben firmar para crear una transacción válida que gastará los fondos.

En este momento, los scripts multifirma estándar se limitan a un listado máximo de 3 llaves públicas, lo que significa que puedes hacer cualquier cosa desde una firma múltiple 1-de-1 a 3-de-3 o cualquier combinación dentro de ese rango. La limitación de las 3 llaves enumeradas podría levantarse para cuando se publique este libro, así que verifica la función IsStandard() para ver qué acepta actualmente la red. Ten en cuenta que el límite de 3 llaves se aplica solo a los scripts multifirma estándar (también conocidas como "simples"), no a los scripts multifirma contenidos en un script Pay-to-Script-Hash (P2SH). Los scripts multifirma P2SH están limitados a 15 llaves, lo que permite hasta multifirmas 15-de-15. Aprenderemos sobre P2SH en Pay-to-Script-Hash (P2SH).

La forma general de un script de bloqueo que establece una condición multifirma M-de-N es:

```
M <Llave Pública 1> <Llave Pública 2> ... <Llave Pública N> N CHECKMULTISIG
```

donde N es el número total de llaves públicas listadas y M es el umbral de firmas requeridas para gastar la salida.

Un script de bloqueo que establece una condición multifirma 2-de-3 se ve así:

```
2 <Llave Pública A> <Llave Pública B> <Llave Pública C> 3 CHECKMULTISIG
```

El script de bloqueo anterior puede ser satisfecho con un script de desbloqueo conteniendo pares de firmas y llaves públicas:

```
<Firma de B> <Firma de C>
```

o cualquier combinación de dos firmas a partir de las llaves privadas correspondientes a las tres llaves públicas listadas.

Los dos scripts juntos formarían el script de validación combinado:

```
<Firma de B> <Firma de C> 2 <Llave Pública A> <Llave Pública B> <Llave Pública C> 3 CHECKMULTISIG
```

Cuando es ejecutado, el script combinado evaluará a VERDADERO si, y solo si, el script de desbloqueo cumple las condiciones establecidas por el script de bloqueo. En este caso, la condición es si el script de desbloqueo contiene una firma válida proveniente de las dos llaves privadas que corresponden a dos de las tres llaves públicas establecidas como obstrucciones.

Un error en la ejecución de CHECKMULTISIG

Hay un error en la ejecución de CHECKMULTISIG que requiere una ligera solución. Cuando se ejecuta CHECKMULTISIG, debería sacar como parámetros M+N+2 elementos de la pila . Sin embargo, debido al error, CHECKMULTISIG sacará un

valor adicional o un valor más de lo esperado.

Veamos esto en más detalle usando el ejemplo anterior de validación:

```
<Firma de B> <Firma de C> 2 <Llave Pública A> <Llave Pública B> <Llave Pública C> 3 CHECKMULTISIG
```

Primero, CHECKMULTISIG saca el elemento superior, que es N (en este ejemplo "3"). A continuación, aparecen N elementos, que son las llaves públicas que pueden firmar. En este ejemplo, las llaves públicas A, B y C. Luego, muestra un elemento, que es M, el quórum (cuántas firmas son necesarias). Aquí M=2. En este punto, CHECKMULTISIG debe sacar los últimos M elementos, que son las firmas, y ver si son válidas. Sin embargo, desafortunadamente, un error en la implementación hace que CHECKMULTISIG saque un elemento más (M+1 en total) de lo que debería. El elemento adicional no se tiene en cuenta al verificar las firmas, por lo que no tiene ningún efecto directo en CHECKMULTISIG en sí. Sin embargo, debe estar presente un valor adicional porque, si no está presente, cuando CHECKMULTISIG intenta sacar en una pila vacía, provocará un error de pila y fallará el script (lo que marcará la transacción como no válida). Debido a que el elemento adicional no se tiene en cuenta, puede ser cualquier cosa, pero habitualmente se usa 0.

Debido a que este error se convirtió en parte de las reglas de consenso, ahora debe ser replicado para siempre. Por lo tanto, la validación correcta del script se vería así:

```
0 <Firma de B> <Firma de C> 2 <Llave Pública A> <Llave Pública B> <Llave Pública C> 3 CHECKMULTISIG
```

Por lo tanto, el script de desbloqueo realmente utilizado en multifirma no es:

```
<Firma de B> <Firma de C>
```

sino:

```
0 <Firma de B> <Firma de C>
```

A partir de ahora, si ves un script de desbloqueo multifirma, deberías esperar ver un 0 adicional al principio, cuyo único propósito es como solución a un error que accidentalmente se convirtió en una regla de consenso.

Pay-to-Script-Hash (P2SH)

Pay-to-Script-Hash (P2SH) (en español, Paga-a-Hash-de-Script) se introdujo en 2012 como un nuevo y poderoso tipo de transacción que simplifica enormemente el uso de scripts de transacciones complejas. Para explicar la necesidad de P2SH, veamos un ejemplo práctico.

En Introducción presentamos a Mohammed, un importador de productos electrónicos con sede en Dubai. La compañía de Mohammed utiliza ampliamente la función de multifirma de bitcoin para sus cuentas corporativas. Los scripts de multifirma son uno de los usos más comunes de las capacidades avanzadas de scripting de bitcoin y son una característica muy poderosa. La compañía de Mohammed usa un script de multifirma para todos los pagos de los clientes, conocido en términos contables como "cuentas por cobrar" o CC. Con el esquema de multifirma, cualquier pago realizado por los clientes se bloquea de tal manera que se requieren al menos dos firmas para su liberación, de Mohammed y uno de sus socios o de su abogado que tiene una llave de respaldo. Un esquema de multifirma como ese ofrece controles de gobierno corporativo y protege contra robo, malversación o pérdida.

El script resultante es bastante largo y se ve así:

```
2 <Llave Pública de Mohammed> <Llave Pública del Socio1> <Llave Pública del Socio2> <Llave Pública del Socio3> <Llave
Pública del Abogado> 5 CHECKMULTISIG
```

Los scripts de multifirma son una característica poderosa, pero son complicados de usar. Dado el script anterior, Mohammed tendría que comunicar ese script a cada cliente antes del pago. Cada cliente tendría que usar un software especial de cartera bitcoin con la capacidad de crear scripts de transacción personalizados, y cada cliente tendría que entender cómo crear una transacción utilizando scripts personalizados. Además, la transacción resultante sería aproximadamente cinco veces más grande que una transacción de pago simple, porque este script contiene llaves públicas muy largas. La carga de esa transacción extra grande sería asumida por el cliente en forma de comisiones. Finalmente, un script de transacción de gran tamaño como esta se guardaría en el set UTXO en RAM en cada nodo

completo, hasta que se gastara. Todos estos problemas hacen que el uso de scripts de bloqueo complejos sea difícil en la práctica.

Para resolver estas dificultades prácticas se desarrolló P2SH y para hacer que el uso de scripts complejos sea tan fácil como un pago a una dirección bitcoin. Con los pagos P2SH, el script de bloqueo complejo se reemplaza con su huella digital, un hash criptográfico. Cuando una transacción que intenta gastar el UTXO se presenta más adelante, debe contener el script que coincida con el hash, además del script de desbloqueo. En términos simples, P2SH significa "pagar a un script que coincida con este hash, un script que se presentará más adelante cuando se gaste esta salida".

En las transacciones P2SH, el script de bloqueo que se reemplaza por un hash se conoce como *scripit de canje* (en inglés, *redeem script*) porque se presenta al sistema en el momento del canje en lugar de como un script de bloqueo. Script complejo sin P2SH muestra el script sin P2SH y Script complejo como P2SH muestra el mismo script codificado en P2SH.

Table 20. Script complejo sin P2SH

Script de Bloqueo	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Script de Desbloqueo	Firma1 Firma2

Table 21. Script complejo como P2SH

Script de Canje	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG
Script de Bloqueo	HASH160 <hash de 20 bytes para el script de canje> EQUAL
Script de Desbloqueo	Sig1 Sig2 <script de canje>

Como puedes ver de las tablas, con P2SH el script complejo que detalla las condiciones para gastar la salida (script de canje) no se presenta en el script de bloqueo. En cambio, solo su hash se encuentra en el script de bloqueo, y el script de canje en sí se presenta después, como parte del script de desbloqueo cuando se gasta la salida. Esto traslada la carga de comisiones y la complejidad del remitente al destinatario (gastador) de la transacción.

Veamos la empresa de Mohammed, el complejo script multifirma, y los scripts P2SH resultantes.

Primero, el script multifirma que usa la empresa de Mohammed para todos los pagos entrantes de clientes:

```
2 <Llave Pública de Mohammed> <Llave Pública del Socio1> <Llave Pública del Socio2> <Llave Pública del Socio3> <Llave
Pública del Abogado> 5 CHECKMULTISIG
```

Si los marcadores de posición son reemplazados por llaves públicas reales (mostradas aquí como números de 520 bits que comienzan por 04) se puede observar que el script se vuelve muy largo:

```
2
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5
DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D
0E34224858008E8B49047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8
F3020DECDBC3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D08722744064
5ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C0
22DD618DA774D207D137AAB59E0B000EB7ED238F4D800 5 CHECKMULTISIG
```

En su lugar, este script completo puede representarse mediante un hash criptográfico de 20 bytes, aplicando primero el algoritmo de hash SHA256 y luego aplicando el algoritmo RIPEMD160 al resultado.

Usamos libbitcoin-explorer (bx) en la línea de comandos para producir el hash del script, de la siguiente manera:

```
echo \
2 \
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF
5DE762C587] \
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E342248
58008E8B49] \
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD3
89D9977965] \
```

```
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE
5C2AFD94A5] \
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7E
D238F4D800] \
5 CHECKMULTISIG \
| bx script-encode | bx sha256 | bx ripemd160
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

La serie de comandos anterior codifica en primer lugar el script de canje multifirma de Mohammed como un script bitcoin de codificación hexadecimal serializada. El siguiente comando bx calcula el hash SHA256 de eso. El siguiente comando bx vuelve a hacer hash con RIPEMD160, produciendo el hash del script final:

El hash de 20 bytes del script de canje de Mohammed es:

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

Una transacción P2SH bloquea la salida a este hash en lugar del script de canje más largo, utilizando el script de bloqueo:

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

que, como puedes ver, es mucho más corto. En lugar de "paga a este script de multifirma de 5 llaves", la transacción equivalente de P2SH es "paga a un script con este hash". Un cliente que realiza un pago a la compañía de Mohammed solo necesita incluir en su pago este script de bloqueo mucho más corto. Cuando Mohammed y sus socios deseen gastar este UTXO, deben presentar el script de canje original (aquel cuyo hash bloqueó el UTXO) y las firmas necesarias para desbloquearlo, de esta manera ("PKN" en este contexto significa "Llave-PíblicaN", del inglés, "PublicKeyN"):

```
<Firma1> <Firma2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>
```

Ambos scripts se combinan en dos etapas. Primero, el script de canje se chequea contra el script de bloqueo para asegurar que el hash concuerda:

```
<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG> HASH160 <hash del script de canje> EQUAL
```

Si el hash del script de canje concuerda, el script de desbloqueo se ejecuta automáticamente para desbloquear el script de canje:

```
<Firma1> <Firma2> 2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG
```

Casi todos los scripts descritos en este capítulo solo pueden implementarse como scripts P2SH. No se pueden utilizar directamente en el script de bloqueo de un UTXO.

Direcciones P2SH

Otra parte importante de las propiedades de P2SH es la capacidad de codificar un hash de script como una dirección, como se define en BIP-13. Las direcciones P2SH son codificaciones Base58Check del hash de 20 bytes de un script, al igual que las direcciones bitcoin son codificaciones Base58Check del hash de 20 bytes de una llave pública. Las direcciones P2SH usan el prefijo de versión "5", que da como resultado direcciones codificadas en Base58Check que comienzan con un "3".

Por ejemplo, el script complejo de Mohammed, hasheado y codificado en Base58Check como una dirección P2SH, se convierte en 39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw. Podemos confirmarlo con el comando bx:

```
echo \
'54c557e07dde5bb6cb791c7a540e0a4796f5e97e'\
 | bx address-encode -v 5
39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw
```

Ahora, Mohammed puede dar esta "dirección" a sus clientes y pueden usar casi cualquier cartera bitcoin para hacer un pago simple, como si fuera una dirección bitcoin. El prefijo 3 les da una pista de que este es un tipo especial de dirección, una correspondiente a un script en lugar de una llave pública, pero de lo contrario, funciona exactamente de la misma manera que un pago a una dirección bitcoin.

Las direcciones P2SH esconden toda la complejidad de forma que la persona realizando el pago no vea el script.

Beneficios del P2SH

La funcionalidad P2SH ofrece los siguientes beneficios en comparación con el uso directo de scripts complejos en el bloqueo de salidas:

- Scripts complejos son reemplazados por huellas más cortas en la salida de transacción, reduciendo la transacción.
- Los scripts pueden codificarse como una dirección, de forma que el remitente y la cartera del remitente no necesitan ingeniería compleja para implementar P2SH.
- P2SH desplaza la carga de construir el script al destinatario, no al remitente.
- P2SH desplaza la carga en almacenamiento de datos del script largo desde la salida (que además de almacenarse en la cadena de bloques, está en el set UTXO) a la entrada (solo se almacena en la cadena de bloques).
- P2SH desplaza la carga en almacenamiento de datos para el script largo del tiempo presente (pago) a un tiempo futuro (cuando es gastado).
- P2SH desplaza los costos de comisiones de transacción de un script largo del remitente al destinatario, quien debe incluir el largo script de canje para gastarlo.

Script de Canje y Validación

Antes de la versión 0.9.2 del Cliente Principal de Bitcoin, el Pago-al-Hash-de-un-Script estaba limitado a los tipos de scripts de transacciones bitcoin estándar, mediante la función IsStandard(). Eso significa que el script de canje presentado en la transacción de gasto podía ser tan solo uno de los tipos estándar: P2PK, P2PKH o multifirma.

Hacia la versión 0.9.2 del Cliente Principal de Bitcoin, las transacciones P2SH pueden contener cualquier script válido, haciendo al estándar P2SH mucho más flexible y permitiendo experimentación con muchos tipos de transacciones novedosos y complejos.

No puedes colocar un P2SH dentro de un script de canje de P2SH, porque la especificación de P2SH no es recursiva. Además, si bien es técnicamente posible incluir RETURN (ver Salida de Registro de Datos (RETURN)) en un script de canje, ya que nada en las reglas le impide hacerlo, no tiene uso práctico porque ejecutar RETURN durante la validación hará que la transacción sea marcada como inválida.

Ten en cuenta que debido a que el script de canje no se presenta a la red hasta que intentas gastar una salida P2SH, si bloqueas una salida con el hash de un script de canje no válido, se procesará de todos modos. El UTXO se bloqueará con éxito. Sin embargo, no podrás gastarlo porque la transacción de gasto, que incluye el script de canje, no se aceptará porque es un script no válido. Esto crea un riesgo, porque puedes bloquear bitcoin en un P2SH que no puede gastarse más tarde. La red aceptará el script de bloqueo P2SH incluso si corresponde a un script de canje no válido, porque el hash del script no da ninguna indicación del script que representa.

> **WARNING** Los scripts de bloqueo P2SH contienen el hash de un script de canje, que no da pistas sobre el contenido del script de canje mismo. La transacción P2SH se considerará válida y aceptada incluso si el script de canje no es válido. Puedes bloquear accidentalmente bitcoin de tal manera que no pueda gastarse más tarde.

Salida de Registro de Datos (RETURN)

El libro de contabilidad distribuido y con marca de tiempo de bitcoin, la cadena de bloques, tiene usos potenciales más allá de los pagos. Muchos desarrolladores han intentado utilizar el lenguaje de scripting de transacciones para aprovechar la seguridad y la capacidad de recuperación del sistema para aplicaciones como los servicios de notarios digitales, certificados de acciones y contratos inteligentes. Los primeros intentos de usar el lenguaje de script de bitcoin para estos fines implicaron la creación de salidas de transacción que registraran datos en la cadena de bloques; por ejemplo, para registrar una huella digital de un archivo de tal manera que cualquiera pudiera establecer la prueba de existencia de ese archivo en una fecha específica por referencia a esa transacción.

El uso de la cadena de bloques de bitcoin para almacenar información sin relación a pagos bitcoin es un tema controvertido. Muchos desarrolladores lo consideran un abuso y prefieren desalentarlo. Otros lo ven como una demostración de las poderosas posibilidades de la tecnología de la cadena de bloques y prefieren alentar su experimentación. Quienes objetan a la inclusión de datos no relacionados a pagos argumentan que causa "hinchazón de

la cadena de bloques", colocando una carga sobre quienes corren nodos bitcoin completos al tener que almacenar datos que la cadena de bloques no fue pensada para albergar. Adicionalmente, esas transacciones crean UTXOs que no pueden ser gastados, utilizando la dirección bitcoin de destino como un campo libre de 20 bytes. Como la dirección es utilizada para datos, no se corresponde a una llave privada y la UTXO resultante no puede ser gastada *jamás*; es un falso pago. Estas transacciones que no pueden ser gastadas, no pueden ser eliminadas del set UTXO y provocan que el tamaño de la base de datos de UTXOs crezca o "se hinche" para siempre.

En la versión 0.9 del Cliente Principal de Bitcoin se alcanzó un acuerdo de compromiso con la introducción del operador RETURN. RETURN permite a los desarrolladores añadir 80 bytes de datos no relacionados con pagos a la salida de una transacción. Sin embargo, a diferencia del uso de UTXOs falsas, el operador RETURN crea una salida *demostrablemente ingastable* de manera explícita, la cual no necesita ser almacenada en la colección de UTXOs. Las salidas del tipo RETURN se registran en la cadena de bloques, por lo que consumen espacio en disco y contribuyen al incremento del tamaño de la cadena de bloques, pero no se almacenan en el set UTXO y por ende no hinchan el tanque de memoria de las UTXOs ni cargan a los nodos completos con el costo de memoria RAM adicional.

Los scripts RETURN se ven así:

```
RETURN <data>
```

La porción de datos se limita a 80 bytes y frecuentemente representa un hash, como el resultado del algoritmo SHA256 (32 bytes). Muchas aplicaciones colocan un prefijo delante de los datos para ayudar a identificar la aplicación. Por ejemplo, el servicio de autorización bajo notario digital Proof of Existence usa el prefijo de 8 bytes "DOCPROOF," el cual es ASCII codificado como 44 4f 43 50 52 4f 4f 46 en hexadecimal.

Ten en cuenta que no existe un "script de desbloqueo" que corresponda a un RETURN que pudiera ser usado para "gastar" una salida RETURN. El propósito de RETURN es que no se pueda gastar el dinero bloqueado en esa salida y por lo tanto no requiere almacenarse en el set UTXO como potencialmente gastable—RETURN es *demostrablemente ingastable*. RETURN es generalmente una salida con un monto de cero bitcoin, ya que cualquier monto en bitcoin asignado a tal salida estaría efectivamente perdido para siempre. Si se hace referencia a un RETURN como una entrada en una transacción, el motor de validación del script detendrá la ejecución del script de validación y marcará la transacción como no válida. La ejecución de RETURN esencialmente hace que el script retorne con un FALSE y se detenga. Por lo tanto, si haces referencia accidentalmente a una salida RETURN como una entrada en una transacción, esa transacción no es válida.

Una transacción estándar (una que cumple con los chequeos de isStandard()) puede tener tan solo una salida RETURN. Sin embargo, una única salida RETURN puede combinarse en una transacción con varias salidas de otros tipos.

Se han agregado dos nuevas opciones de línea de comandos en Bitcoin Core a partir de la versión 0.10. La opción datacarrier controla la retransmisión y minería de las transacciones RETURN, con el valor predeterminado establecido en "1" para permitirlas. La opción datacarriersize toma un argumento numérico que especifica el tamaño máximo en bytes del script RETURN, 83 bytes por defecto, lo que permite un máximo de 80 bytes de datos RETURN más un byte del opcode RETURN y dos bytes del opcode PUSHDATA.

NOTE

RETURN se propuso inicialmente con un límite de 80 bytes, pero el límite se redujo a 40 bytes cuando se lanzó la función. En febrero de 2015, en la versión 0.10 de Bitcoin Core, el límite se elevó de nuevo a 80 bytes. Los nodos pueden elegir no retransmitir o minar RETURN, o solo retransmitir y minar RETURN que contengan menos de 80 bytes de datos.

Bloqueos de tiempo (Timelocks)

Los bloqueos de tiempo son restricciones en transacciones o salidas que solo permiten gastar después de un momento en el tiempo. Bitcoin ha tenido una característica de bloqueo de tiempo a nivel de transacción desde el principio. Se implementa mediante el campo nLocktime en una transacción. Se introdujeron dos nuevas funciones de bloqueo de tiempo a finales de 2015 y mediados de 2016 que ofrecen bloqueos de tiempo a nivel de UTXO. Estos son CHECKLOCKTIMEVERIFY y CHECKSEQUENCEVERIFY.

Los bloqueos de tiempo son útiles para transacciones posteriores a la fecha actual y para bloquear fondos hasta una fecha en el futuro. Más importante aún, los bloqueos de tiempo ofrecen a los scripts de bitcoin una dimensión del tiempo, abriendo la puerta a complejos contratos inteligentes de múltiples pasos.

Desde sus comienzos, bitcoin ha tenido una función de bloqueo de tiempo a nivel de transacción. El bloqueo temporal de la transacción es una configuración que se ubica a nivel de la transacción (en un campo en la estructura de datos de la misma) y que define a partir de qué momento una transacción es válida y se puede retransmitir en la red o agregar a la cadena de bloques. El bloqueo temporal también se conoce como nLocktime proveniente del nombre de la variable utilizada en el código base de Bitcoin Core. Este parámetro se establece en cero en la mayoría de las transacciones para indicar la propagación y ejecución inmediata. Si el valor de nLocktime no es cero y está por debajo de 500 millones, se interpreta como una altura de bloque, lo que significa que la transacción no será válida y no se retransmitirá ni se incluirá en la cadena de bloques antes que la referida altura de bloque no haya sido alcanzada. Si es mayor o igual a 500 millones, se interpreta como un sello de tiempo del tipo "Unix Epoch" (es decir, segundos transcurridos desde el 1° de enero de 1970) y la transacción no se admite como válida antes de cumplirse ese momento especificado. Aquellas transacciones con una especificación del parámetro nLocktime que indican un número futuro de bloque un momento futuro del tiempo deben ser retenidas por el sistema que las origina y solo deben ser difundirlas a la red bitcoin una vez que sean validas. Si una transacción se transmite a la red antes que se cumpla la condición del parámetro nLocktime especificado, será rechazada apenas alcance el siguiente nodo interpretándose como una transacción no válida y no será retransmitida a otros nodos. El uso de nLocktime es equivalente a un cheque de papel post-fechado.

Limitaciones del bloqueo temporal de las transacciones

nLocktime tiene la limitación de que si bien permite gastar algunas salidas en el futuro, no es imposible que estas puedan gastarse antes de ese momento. Vamos a explicar eso con el siguiente ejemplo.

Alice firma una transacción y gasta una de sus salidas hacia la dirección de Bob, estableciendo el nLocktime de la transacción a 3 meses en el futuro. Alice envía esa transacción a Bob para que la retenga. Con esta transacción Alice y Bob saben que:

- Bob no puede transmitir la transacción para gastar los fondos hasta que hayan transcurrido 3 meses.
- Bob puede transmitir la transacción después de 3 meses.

Sin embargo:

- Alice puede crear otra transacción, haciendo un "doble gasto" de esas mismas entradas sin requerir bloqueo temporal. Por lo tanto, Alice puede gastar el mismo UTXO antes de que hayan transcurrido los 3 meses.
- Bob no tiene garantía de que Alice no hará eso.

Es importante entender las limitaciones de la transacción nLocktime. La única garantía es que Bob no podrá gastarlo antes de que hayan transcurrido 3 meses. No hay garantía de que Bob obtendrá los fondos. Para lograr dicha garantía, la restricción de bloqueo de tiempo debe colocarse en el propio UTXO y ser parte del script de bloqueo, en lugar de en la transacción. Esto se logra mediante la siguiente forma de bloqueo de tiempo, llamada Check Lock Time Verify.

Check Lock Time Verify (CLTV)

En diciembre de 2015, se introdujo una nueva forma de bloqueo de tiempo en bitcoin como una bifurcación suave. Su especificación se detalla en BIP-65, que incluye un nuevo operador de script llamado *CHECKLOCKTIMEVERIFY* (*CLTV*) que se agregó al lenguaje de scripting. CLTV es un bloqueo de tiempo que afecta a cada salida, en lugar de un bloqueo de tiempo por transacción como es el caso con nLocktime. Esto ofrece una mayor flexibilidad en la forma en que se aplican los bloqueos de tiempo.

En términos simples, al agregar el opcode CLTV en el script de canje de una salida, se restringe la salida, de modo que solo puede gastarse una vez transcurrido el tiempo especificado.

> **TIP** Mientras nLocktime es un bloqueo de tiempo a nivel de transacción, CLTV es un bloqueo de tiempo basado en salida.

CLTV no sustituye a nLocktime, sino que restringe UTXO específicos, de modo que solo se puedan gastar en una transacción futura que tenga establecida su nLocktime a un valor mayor o igual.

El opcode CLTV toma un parámetro como entrada, expresado como un número en el mismo formato que nLocktime (ya sea una altura de bloque o un tiempo de época de Unix). Como indica el sufijo VERIFY, CLTV es el tipo de opcode que detiene la ejecución del script si el resultado es FALSE. Si resulta en VERDADERO, la ejecución continúa.

Para bloquear una salida con CLTV, lo incluyes en el script de canje de la salida en la transacción que crea la salida. Por

ejemplo, si Alice está pagando a la dirección de Bob, la salida normalmente contendría un script P2PKH como este:

```
DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG
```

Para bloquearlo a un momento en el tiempo, digamos que dentro de 3 meses, la transacción sería una transacción P2SH con un script de canje como este:

```
<now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG
```

donde <now {plus} 3 months> es una altura de bloque o un valor de tiempo estimado de 3 meses desde el momento en que se mina la transacción: altura de bloque actual {más} 12,960 (bloques) o tiempo actual de época de Unix {más} 7,760,000 (segundos). Por ahora, no te preocupes por el código de operación DROP que sigue a CHECKLOCKTIMEVERIFY; Se explicará en breve.

Cuando Bob intenta gastar este UTXO, construye una transacción que hace referencia al UTXO como una entrada. Utiliza su firma y llave pública en el script de desbloqueo de esa entrada y establece el nLocktime de la transacción a un valor igual o mayor que el bloqueo de tiempo en el conjunto CHECKLOCKTIMEVERIFY de Alice. Bob después transmite la transacción en la red bitcoin.

La transacción de Bob se evalúa de la siguiente manera. Si el parámetro de CHECKLOCKTIMEVERIFY que Alice establece es menor o igual que el nLocktime de la transacción de gasto, la ejecución del script continúa (actúa como si fuera “no operación” o se hubiera ejecutado NOP). De lo contrario, la ejecución del script se detiene y la transacción se considera inválida.

Más concretamente, CHECKLOCKTIMEVERIFY falla y se detiene la ejecución, marcando la transacción como inválida si (fuente: BIP-65):

1. la pila está vacía; o
2. el elemento superior en la pila es menor que 0; o
3. el tipo de tiempo de bloqueo (altura frente a sello de tiempo) del elemento superior de la pila y el campo nLocktime no son los mismos; o
4. el elemento superior de la pila es mayor que el campo nLocktime de la transacción; o
5. el campo nSequence de la entrada es 0xffffffff.

> **NOTE** CLTV y nLocktime usan el mismo formato para describir los bloqueos de tiempo, ya sea una altura de bloque o el tiempo transcurrido en segundos desde la época Unix. Necesariamente, cuando se usan juntos, el formato de nLocktime debe coincidir con el de CLTV en las salidas—ambos deben hacer referencia a la altura de bloque o al tiempo en segundos.

Después de la ejecución, si se satisface el CLTV, el parámetro de tiempo que lo precedió permanece como el elemento superior en la pila y es posible que deba eliminarse, con DROP, para la ejecución correcta de los siguientes opcodes del script. Por este motivo, a menudo verás en los scripts CHECKLOCKTIMEVERIFY seguido de DROP.

Al usar nLocktime junto con CLTV, el escenario descrito en Limitaciones del bloqueo temporal de las transacciones cambia. Alice ya no puede gastar el dinero (porque está bloqueado con la llave de Bob) y Bob no puede gastarlo antes de que expire el bloqueo temporal de 3 meses.

Al introducir la funcionalidad de bloqueo de tiempo directamente en el lenguaje de scripting, CLTV nos permite desarrollar algunos scripts complejos muy interesantes.

El estándar se define en BIP-65 (CHECKLOCKTIMEVERIFY).

Bloqueos de Tiempo Relativos

Tanto nLocktime como CLTV son ambos *bloqueos de tiempo absolutos* en el sentido de que especifican momentos absolutos de tiempo. Las siguientes dos características de bloqueo de tiempo que examinaremos son los *bloqueos de tiempo relativos* que especifican, como condición para gastar una salida, un tiempo transcurrido desde la confirmación de la salida en la cadena de bloques.

Los bloqueos de tiempo relativos son útiles porque permiten que una cadena de dos o más transacciones interdependientes se mantengan fuera de la cadena, y simultáneamente impone una restricción de tiempo en una transacción que depende del tiempo transcurrido desde la confirmación de una transacción anterior. En otras palabras, el reloj no comienza a contar hasta que se registra el UTXO en la cadena de bloques. Esta funcionalidad es especialmente útil en los canales de estado bidireccionales y en la Red Lightning, como veremos en Canales de Pago y Canales de Estados.

Los bloqueos de tiempo relativos, como los bloqueos de tiempo absolutos, se implementan tanto con una característica a nivel de transacción como con un opcode a nivel de script. El bloqueo de tiempo relativo a nivel de transacción se implementa como una regla de consenso sobre el valor de nSequence, un campo de transacción que se establece en cada entrada de transacción. Los bloqueos de tiempo relativos a nivel de script se implementan con el opcode CHECKSEQUENCEVERIFY (CSV).

Los bloqueos de tiempo relativos se implementan según las especificaciones en BIP-68, Tiempo de bloqueo relativo usando números de secuencia impuestos por consenso y BIP-112, CHECKSEQUENCEVERIFY.

BIP-68 y BIP-112 se activaron en mayo de 2016 como una bifurcación suave a las reglas de consenso.

Bloqueos de Tiempo Relativos con nSequence

Los bloqueos de tiempo relativos se pueden establecer en cada entrada de una transacción, estableciendo el campo nSequence en cada entrada.

Significado original de nSequence

El campo nSequence fue originalmente pensado (pero nunca implementado correctamente) para permitir la modificación de transacciones en el mempool. En ese uso, una transacción que contuviera entradas con un valor de nSequence por debajo de 2^{32} - 1 (0xFFFFFFFF) indicaba una transacción que aún no se había "finalizado". Dicha transacción se mantendría en el mempool hasta que fuera reemplazada por otra transacción que gastara las mismas entradas con un valor de nSequence más alto. Una vez que se recibiera una transacción cuyas entradas tenían un valor de nSequence de 0xFFFFFFFF, se consideraría "finalizada" y se minaría.

El significado original de nSequence nunca se implementó correctamente y el valor de nSequence se establece habitualmente en 0xFFFFFFFF en transacciones que no utilizan bloqueos temporales. Para transacciones con nLocktime o CHECKLOCKTIMEVERIFY, el valor nSequence debe establecerse en menos de 2^{31} para que los guardias de bloqueo temporal tengan efecto, como se explica a continuación.

nSequence como bloqueo de tiempo relativo impuestos por consenso

Desde la activación de BIP-68, se aplican nuevas reglas de consenso para cualquier transacción que contenga una entrada cuyo valor nSequence sea menor que 2^31 ^ (el bit 1<<31 no está establecido). Programáticamente, eso significa que si el bit más significativo (bit 1<<31) no se establece, es un indicador que significa "bloqueo temporal relativo". De lo contrario (bit 1<<31 establecido), el valor nSequence se reserva para otros usos, como habilitar CHECKLOCKTIMEVERIFY, nLocktime, Opt-In-Replace-By-Fee, y otros desarrollos futuros .

Las entradas de transacciones con valores de nSequence menores que 2^{31} se interpretan como que tienen un bloqueo de tiempo relativo. Dicha transacción solo es válida una vez que la entrada haya vencido por la cantidad relativa de bloqueo de tiempo. Por ejemplo, una transacción con una entrada y un bloqueo de tiempo relativo nSequence de 30 bloques solo es válida cuando hayan transcurrido al menos 30 bloques desde el momento en que se minó el UTXO al que se hizo referencia en la entrada. Como existe un campo nSequence por cada entrada, una transacción puede contener cualquier número de entradas con bloqueo de tiempo, de las cuales, todas deben tener una antigüedad suficiente para que la transacción sea válida. Una transacción puede incluir tanto entradas con bloqueo de tiempo (nSequence < 2^{31}) como entradas sin bloqueo de tiempo relativo (nSequence >= 2^{31}).

El valor de nSequence se especifica en bloques o en segundos, pero en un formato ligeramente diferente al que vimos en nLocktime. Se utiliza un indicador de tipo para diferenciar entre el conteo de valores en bloques y el conteo de valores de tiempo en segundos. El indicador de tipo se establece en el bit 23 menos significativo (es decir, el valor 1<<22). Si se establece el indicador de tipo, el valor nSequence se interpreta como un múltiplo de 512 segundos. Si el indicador de tipo no está establecido, el valor nSequence se interpreta como un número de bloques.

Al interpretar nSequence como un bloqueo de tiempo relativo, solo se consideran los 16 bits menos significativos. Una vez que se evalúan los indicadores (bits 32 y 23), el valor de nSequence generalmente se "enmascara" con una máscara de 16 bits (por ejemplo, nSequence & 0x0000FFFF).

Definición en BIP-68 de la codificación de nSequence (Fuente: BIP-68) muestra el diseño binario del valor nSequence, como se define en BIP-68.

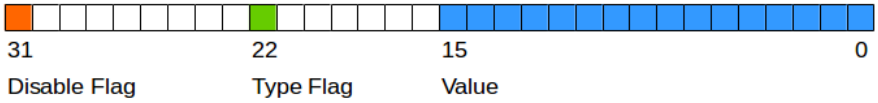


Figure 46. Definición en BIP-68 de la codificación de nSequence (Fuente: BIP-68)

Los bloqueos de tiempo relativos basados en la imposición por consenso del valor nSequence se definen en BIP-68.

El estándar se define en BIP-68, Tiempo de bloqueo relativo usando números de secuencia impuestos por consenso.

Bloqueos de Tiempo Relativos con CSV

Al igual que CLTV y nLocktime, hay un opcode de script para los bloqueos de tiempo relativos que aprovecha el valor de nSequence en los scripts. Ese opcode es CHECKSEQUENCEVERIFY, comúnmente denominado CSV para abreviar.

El opcode CSV, cuando se evalúa en un script de canje de UTXO, permite gastar solo en una transacción cuyo valor nSequence de entrada es mayor o igual que el parámetro CSV. Esencialmente, esto restringe el gasto del UTXO hasta que haya transcurrido un cierto número de bloques o segundos desde que se minó el UTXO.

Al igual que con CLTV, el valor en CSV debe coincidir con el formato del valor nSequence correspondiente. Si se especifica CSV en términos de bloques, entonces también debe nSequence. Si se especifica CSV en términos de segundos, entonces también debe nSequence.

Los bloqueos de tiempo relativos con CSV son especialmente útiles cuando se crean y firman varias transacciones (encadenadas), pero no se propagan, cuando se mantienen "fuera de la cadena". Una transacción hija no se puede usar hasta que la transacción padre se haya propagado, minado y dejado envejecer por el tiempo especificado en el bloqueo de tiempo relativo. Una aplicación de este caso de uso se puede ver en Canales de Pago y Canales de Estados y Canales de Pago Enrutados (Lightning Network).

CSV se define en detalle en BIP-112, CHECKSEQUENCEVERIFY.

Tiempo-Mediano-Pasado

Como parte de la activación de los bloqueos de tiempo relativos, también hubo un cambio en la forma en que se calcula el "tiempo" para los intervalos de tiempo (tanto absolutos como relativos). En bitcoin hay una diferencia sutil, pero muy significativa, entre el tiempo de reloj de pared y el tiempo de consenso. Bitcoin es una red descentralizada, lo que significa que cada participante tiene su propia perspectiva del tiempo. Los eventos en la red no ocurren instantáneamente en todas partes. La latencia de la red se debe tener en cuenta en la perspectiva de cada nodo. Finalmente, todo se sincroniza para crear un libro de contabilidad común. Bitcoin alcanza el consenso cada 10 minutos sobre el estado del libro de contabilidad tal como existía en el *pasado*.

Los mineros establecen los sellos de tiempo en las cabeceras de bloque. Las reglas de consenso permiten un cierto grado de flexibilidad para tener en cuenta las diferencias en la precisión del reloj de los nodos descentralizados. Sin embargo, esto crea un incentivo desafortunado para que los mineros mientan sobre el tiempo en un bloque para ganar comisiones adicionales al incluir transacciones bloqueadas en el tiempo que aún no están maduras. Lee la siguiente sección para más información.

Para eliminar el incentivo a mentir y así fortalecer la seguridad de los bloqueos de tiempo, se propuso y activó un BIP al mismo tiempo que los BIP para bloqueos de tiempo relativos. Se trata del BIP-113, que define una nueva medida de consenso de tiempo llamada *Tiempo-Mediano-Pasado*.

Tiempo-Mediano-Pasado se calcula tomando las marcas de tiempo de los últimos 11 bloques y encontrando la mediana. Ese tiempo mediano se convierte en tiempo de consenso y se utiliza para todos los cálculos de bloqueo de tiempo. Al tomar el punto medio de aproximadamente dos horas en el pasado, se reduce la influencia del sello de tiempo de cualquier bloque. Al incorporar 11 bloques, ningún minero puede influir en las marcas de tiempo para obtener comisiones de transacción con un bloqueo de tiempo que aún no ha madurado.

Tiempo-Mediano-Pasado cambia la implementación de los cálculos de tiempo para nLocktime, CLTV, nSequence y CSV. El tiempo de consenso calculado por Tiempo-Mediano-Pasado es siempre aproximadamente una hora detrás del reloj de pared. Si creas transacciones de bloqueo de tiempo, debes tenerlo en cuenta para estimar el valor deseado codificado en nLocktime, nSequence, CLTV y CSV.

Tiempo-Mediano-Pasado se especifica en BIP-113.

Defensa de Bloqueo de Tiempo Contra Disparo de Comisiones

El francotirador de comisiones es un escenario de ataque teórico, en el que los mineros que intentan volver a escribir los bloques anteriores "roban" las transacciones de mayor comisión de futuros bloques para maximizar su rentabilidad.

Por ejemplo, digamos que el bloque más alto que existe es el bloque #100,000, y en lugar de intentar minar el bloque #100,001 para extender la cadena, algunos mineros intentan reminar el #100,000. Estos mineros pueden elegir incluir cualquier transacción válida (que aún no se haya minado) en su bloque candidato #100,000. No tienen por qué reminar el bloque con las mismas transacciones. De hecho, tienen el incentivo de seleccionar las transacciones más rentables (comisión más alta por kB) para incluirlas en su bloque. Pueden incluir cualquier transacción que estuviera en el bloque "antiguo" #100,000, así como cualquier transacción del mempool actual. Esencialmente tienen la opción de llevar las transacciones del "presente" al "pasado" reescrito cuando recrean el bloque #100,000.

Hoy en día, este ataque no es muy lucrativo, porque la recompensa por encontrar el bloque es mucho más alta que las comisiones totales por bloque. Pero en algún momento en el futuro, las comisiones de transacción serán la mayoría de la recompensa (o incluso la totalidad de la recompensa). En ese momento, este escenario se vuelve inevitable.

Para evitar el "disparo de comisiones", cuando Bitcoin Core crea transacciones utiliza nLocktime para limitarlas al "bloque siguiente", de forma predeterminada. En nuestro escenario, Bitcoin Core establecería nLocktime en 100,001 en cualquier transacción que haya creado. En circunstancias normales, este nLocktime no tiene ningún efecto— las transacciones solo se podrían incluir en el bloque #100,001 de todos modos; es el siguiente bloque.

Pero bajo un ataque de bifurcación de la cadena de bloques, los mineros no podrían obtener transacciones con altas comisiones del mempool, porque todas esas transacciones se bloquearían en el bloque #100,001. Solo pueden reminar el #100,000 con cualquier transacción que fuera válida en ese momento, esencialmente sin obtener nuevas comisiones.

Para lograr esto, Bitcoin Core establece el nLocktime en todas las transacciones nuevas a <current block # + 1> y establece el nSequence en todas las entradas a 0xFFFFFFFE para habilitar nLocktime.

Scripts con Control de Flujo (Cláusulas Condicionales)

Una de las características más poderosas de Bitcoin Script es el control de flujo, también conocido como cláusulas condicionales. Probablemente estés familiarizado con el control de flujo en varios lenguajes de programación que usan la construcción IF...THEN...ELSE. Las cláusulas condicionales de Bitcoin se muestran un poco diferentes, pero son esencialmente la misma construcción.

En un nivel básico, los opcodes condicionales de bitcoin nos permiten construir un script de canje que tiene dos formas de desbloquear, dependiendo del resultado TRUE/FALSE que se devuelva al evaluar una condición lógica. Por ejemplo, si x es TRUE, el script de canje es A y el script de canje para el "ELSE" es B.

Además, las expresiones condicionales de bitcoin se pueden "anidar" indefinidamente, lo que significa que una cláusula condicional puede contener otra dentro de ella, que contiene otra, etc. El control de flujo de Bitcoin Script se puede usar para construir scripts muy complejos con cientos o incluso miles de posibles vías de ejecución. No hay límite en la anidación, pero las reglas de consenso imponen un límite en el tamaño máximo, en bytes, de un script.

Bitcoin implementa el control de flujo utilizando los opcodes IF, ELSE, ENDIF y NOTIF. Además, las expresiones condicionales pueden contener operadores booleanos como BOOLAND, `BOOLOR` , y NOT.

A primera vista, puedes encontrar confusos los scripts de control de flujo de bitcoin. Eso es porque Bitcoin Script es un lenguaje de pila. De la misma manera que 1 {plus} 1 se ve "hacia atrás" cuando se expresa como 1 1 ADD, las cláusulas de control de flujo en bitcoin también parecen verse "hacia atrás".

En la mayoría de los lenguajes de programación (procedurales), el control de flujo se ve así:

Pseudocódigo del control de flujo en la mayoría de los lenguajes de programación

```
if (condición):
  código a ejecutar cuando la condición es true
else:
  código a ejecutar cuando la condición es false
código a ejecutar en cualquier caso
```

En un lenguaje basado en pila como el Bitcoin Script, la condición lógica viene antes del IF, que hace que se vea "hacia atrás", así:

Control de flujo de Bitcoin Script

```
condición
IF
  código a ejecutar cuando la condición es true
ELSE
  código a ejecutar cuando la condición es false
ENDIF
código a ejecutar en cualquier caso
```

Al leer Bitcoin Script, recuerda que la condición que se evalúa se produce *antes* que el código de operación IF.

Cláusulas Condicionales con Opcodes VERIFY

Otra forma de condicional en Bitcoin Script es cualquier opcode que termina en VERIFY. El sufijo VERIFY significa que si la condición evaluada no es TRUE, la ejecución del script termina inmediatamente y la transacción se considera inválida.

A diferencia de una cláusula IF, que ofrece rutas de ejecución alternativas, el sufijo VERIFY actúa como una *cláusula guarda*, que continúa solo si se cumple una condición previa.

Por ejemplo, el siguiente script requiere la firma de Bob y una preimagen (secreto) que produce un hash específico. Ambas condiciones deben satisfacerse para desbloquear:

Un script de canje con una cláusula guarda EQUALVERIFY.

```
HASH160 <hash esperado> EQUALVERIFY <Llave Pública de Bob> CHECKSIG
```

Para redimir esta salida, Bob debe construir un script de desbloqueo que presente una preimagen y una firma válidas:

Un script de desbloqueo que satisface el script de canje anterior

```
<Firma de Bob> <preimagen de hash>
```

Sin presentar la preimagen, Bob no puede acceder a la parte del script que comprueba su firma.

En su lugar, este script se puede escribir con un IF:

Un script de canje con una cláusula de guardia, IF

```
HASH160 <hash esperado> EQUAL
IF
   <Llave Pública de Bob> CHECKSIG
ENDIF
```

El script de desbloqueo de Bob es idéntico:

Un script de desbloqueo que satisface el script de canje anterior

```
<Firma de Bob> <preimagen de hash>
```

El script con IF hace lo mismo que usar un opcode con un sufijo VERIFY; ambos operan como cláusulas de guarda. Sin embargo, la construcción VERIFY es más eficiente, ya que utiliza dos opcodes menos.

Entonces, ¿cuándo usamos VERIFY y cuándo usamos IF? Si lo único que intentamos hacer es adjuntar una condición previa (cláusula de guarda), entonces VERIFY es mejor. Sin embargo, si queremos tener más de una ruta de ejecución (control de flujo), necesitamos una cláusula de control de flujo IF...ELSE.

TIP

Un opcode como EQUAL empujará el resultado (TRUE/FALSE) en la pila, dejándolo allí para su evaluación

> por los opcodes subsiguientes. Por contra, el sufijo de opcode EQUALVERIFY no deja nada en la pila. Los opcodes que terminan en VERIFY no dejan el resultado en la pila.

Usando el Control de Flujo en Scripts

Un uso muy común en el control de flujo en Bitcoin Script es construir un script de canje que ofrezca múltiples caminos de ejecución, cada una de las cuales es una forma diferente de redimir la UTXO.

Veamos un ejemplo simple, donde tenemos dos firmantes, Alice y Bob, y cualquiera de los dos puede redimir. Con multifirmas, esto se expresaría como un script multifirma 1-de-2. Como ejemplo, haremos lo mismo con una cláusula IF:

```
IF
 <Llave Pública de Alice> CHECKSIG
ELSE
 <Llave Pública de Bob> CHECKSIG
ENDIF
```

Al observar este script de canje, puede que te preguntes: "¿Dónde está la condición? ¡No hay nada antes de la cláusula IF!"

La condición no es parte del script de canje. En su lugar, la condición aparecerá en el script de desbloqueo, lo que permitirá a Alice y Bob "elegir" qué camino de ejecución desean.

Alice redime esto con el script de desbloqueo:

```
<Firma de Alice> 1
```

El 1 al final sirve como la condición (TRUE) que hará que la cláusula + IF ejecute el primer camino de canje para el cual Alice tiene una firma.

Para que Bob pueda redimir esta salida, tendría que elegir el segundo camino de ejecución dando un valor FALSE a la cláusula IF:

```
<Firma de Bob> 0
```

El script de desbloqueo de Bob pone un 0 en la pila, lo que hace que la cláusula IF ejecute el segundo script (ELSE), que requiere la firma de Bob.

Como las cláusulas IF se pueden anidar, podemos crear un "laberinto" de caminos de ejecución. El script de desbloqueo puede proporcionar un "mapa" que selecciona qué camino de ejecución se ha ejecutado realmente:

```
IF
  script A
ELSE
  IF
    script B
  ELSE
    script C
  ENDIF
ENDIF
```

En este escenario, hay tres caminos de ejecución (script A, script B y script C). El script de desbloqueo proporciona un camino en forma de una secuencia de TRUE o FALSE. Para seleccionar el camino script B, por ejemplo, el script de desbloqueo debe terminar en 1 0 (TRUE, FALSE). Estos valores se insertarán en la pila, de modo que el segundo valor (FALSE) termine en la parte superior de la pila. La cláusula IF externa muestra el valor FALSE y ejecuta la primera cláusula ELSE. Después, el valor TRUE se mueve a la parte superior de la pila y se evalúa por el IF interno (anidado), seleccionando el camino de ejecución B.

Usando esta estructura, podemos construir scripts de canje con decenas o cientos de caminos de ejecución, cada uno de las cuales ofrece una forma diferente de redimir la UTXO. Para gastarla, construimos un script de desbloqueo que navega por el camino de ejecución al colocar los valores apropiados TRUE y FALSE en la pila en cada punto de control del flujo.

Ejemplo de Script Complejo

En esta sección combinamos muchos de los conceptos de este capítulo en un solo ejemplo.

Nuestro ejemplo utiliza la historia de Mohammed, el propietario de la empresa en Dubai que opera un negocio de importación/exportación.

En este ejemplo, Mohammed desea construir una cuenta de capital de empresa con reglas flexibles. El esquema que crea requiere diferentes niveles de autorización según los intervalos de tiempo. Los participantes en el esquema de multifirma son Mohammed, sus dos socios, Saeed y Zaira, y el abogado de su compañía, Abdul. Los tres socios toman decisiones basadas en una regla de mayoría, por lo que dos de los tres deben estar de acuerdo. Sin embargo, en el caso de que exista algún problema con sus llaves, quieren que sus abogados puedan recuperar los fondos con una de las firmas de los tres socios. Finalmente, si todos los socios no están disponibles o están incapacitados por un tiempo, quieren que el abogado pueda administrar la cuenta directamente.<

Aquí está el script de canje que Mohammed diseña para lograr esto (prefijo de número de línea como XX):

Multi-firma Variable con Bloqueo de Tiempo

```
01  IF
02    IF
03      2
04    ELSE
05      <30 días> CHECKSEQUENCEVERIFY DROP
06      <Llave Pública de Abdul el abogado> CHECKSIGVERIFY
07      1
08    ENDIF
09    <Llave Pública de Mohammed> <Llave Pública de Saeed> <Llave Pública de Zaira> 3 CHECKMULTISIG
10  ELSE
11    <90 días> CHECKSEQUENCEVERIFY DROP
12    <Llave Pública del abogado> CHECKSIG
13  ENDIF
```

El script de Mohammed implementa tres caminos de ejecución usando cláusulas de control de flujo anidadas IF ... ELSE.

En el primer camino de ejecución, este script funciona como un multifirma simple 2 de 3 entre los tres socios. Este camino de ejecución consta de las líneas 3 y 9. La línea 3 establece el quórum del multifirma en 2 (2-de-3). Este camino de ejecución se puede seleccionar poniendo TRUE TRUE al final del script de desbloqueo:

Script de desbloqueo para el primer camino de ejecución (multifirma 2-de-3)

```
0 <Firma de Mohammed> <Firma de Zaira> TRUE TRUE
```

> **TIP**
>
> El 0 al principio de este script de desbloqueo se debe a un error en CHECKMULTISIG que saca un valor de más de la pila. CHECKMULTISIG ignora el valor de más, pero debe estar presente para que el script no falle. Empujar 0 (habitualmente) es una solución al error, como se describe en Un error en la ejecución de CHECKMULTISIG.

El segundo camino de ejecución solo se puede utilizar después de que hayan transcurrido 30 días desde la creación del UTXO. En ese momento, se requiere la firma del abogado Abdul y uno de los tres socios (un multifirma 1-de-3). Esto se logra mediante la línea 7, que establece el quórum para el mutifirma en 1. Para seleccionar esta camino de ejecución, el script de desbloqueo terminaría en FALSE TRUE:

Script de desbloqueo para el segundo camino de ejecución (Abogado + 1-de-3)

```
0 <Firma de Saeed> <Firma de Abdul> FALSE TRUE
```

> **TIP**
>
> ¿Por qué FALSE TRUE? ¿No sería al revés? Porque los dos valores se empujan en la pila, primero FALSE, y después se empuja TRUE. Por lo tanto, primero se saca TRUE en el primer código de operación IF.

Finalmente, el tercer camino de ejecución le permite al abogado Abdul gastar los fondos por sí mismo, pero solo después de 90 días. Para seleccionar este camino de ejecución, el script de desbloqueo debe terminar en FALSE:

Script de desbloqueo para el tercer camino de ejecución (solo Abogado)

```
<Firma de Abdul> FALSE
```

Intenta ejecutar el script en papel para ver cómo se comporta en la pila.

Algunas cosas más a considerar al leer este ejemplo. Intenta acertar las respuestas:

- ¿Por qué no puede el abogado redimir mediante el tercer camino de ejecución en cualquier momento seleccionándolo con FALSE en el script de desbloqueo?
- ¿Cuántos caminos de ejecución se pueden usar 5, 35 y 105 días, respectivamente, después de que se mine el UTXO?
- ¿Se pierden los fondos si el abogado pierde su llave? ¿Cambia tu respuesta si han transcurrido 91 días?
- ¿Cómo "reinician" los socios el reloj cada 29 u 89 días para evitar que el abogado acceda a los fondos?
- ¿Por qué algunos opcodes CHECKSIG en este script tienen el sufijo VERIFY mientras que otros no?

Segregated Witness (Testigos Segregados)

Segregated Witness (segwit) es una actualización de las reglas de consenso de bitcoin y de su protocolo de red, propuesto e implementado como una bifurcación blanda BIP-9 que se activó en la red principal de bitcoin el 1 de agosto de 2017.

En criptografía, el término "testigo" (en inglés, "witness") se usa para describir una solución a un acertijo criptográfico. En el caso de bitcoin, el testigo satisface una condición criptográfica colocada en una salida de transacción no gastada (UTXO).

En el contexto de bitcoin, una firma digital es *un tipo de testigo*, pero un testigo es, en un sentido más amplio, cualquier solución que pueda satisfacer las condiciones impuestas en un UTXO y desbloquear ese UTXO para gastarlo. Generalmente, el término "testigo" es un término para un "script de desbloqueo" o "scriptSig".

Antes de la introducción de segwit, cada entrada en una transacción era seguida por los datos testigo que la desbloqueaban. Los datos testigo se incluían en la transacción como parte de cada entrada. El término *segregated witness*, (en español, *testigo segregado*) o *segwit* para abreviar, simplemente significa separar la firma o el script de desbloqueo de una salida específica. Piensa en "scriptSig separado" o "firma separada" en la forma más simple.

Segregated Witness es, por lo tanto, un cambio de arquitectura en bitcoin con el objetivo de mover los datos testigo desde el campo scriptSig (script de desbloqueo) de una transacción a una estructura de datos *testigo* separada que acompaña a una transacción. Los clientes pueden solicitar datos de transacción con o sin los datos testigo adjuntos.

En esta sección analizaremos algunos de los beneficios de Segregated Witness, describiremos el mecanismo utilizado para usar y desplegar este cambio de arquitectura y demostraremos el uso de Segregated Witness en transacciones y direcciones.

En los siguientes BIPs se define Segregated Witness:

BIP-141

La definición principal de Segregated Witness.

BIP-143

Transaction Signature Verification for Version 0 Witness Program

BIP-144

Peer Services—New network messages and serialization formats

BIP-145

getblocktemplate Updates for Segregated Witness (para minería)

BIP-173

Base32 address format for native v0-16 witness outputs

¿Por qué Segregated Witness?

Segregated Witness es un cambio de arquitectura que tiene varios efectos en la escalabilidad, seguridad, incentivos económicos y rendimiento de bitcoin:

Maleabilidad de transacciones: al mover el testigo fuera de la transacción, el hash de transacción utilizado como identificador ya no incluye los datos testigo. Dado que los datos testigo son la única parte de la transacción que puede ser modificada por un tercero (ver Identificadores de transacción), eliminarla también elimina la posibilidad de ataques de

maleabilidad de la transacción. Con Segregated Witness, los hashes de transacciones se vuelven inmutables por cualquier persona que no sea el creador de la transacción, lo que mejora en gran medida la implementación de muchos otros protocolos que se basan en la construcción avanzada de transacciones de bitcoin, como los canales de pago, las transacciones encadenadas y las redes lightning.

Versionado de script: con la introducción de los scripts Segregated Witness, cada script de bloqueo está precedido por un número de *versión de script*, similar a los números de versión de transacciones y bloques. Añadir un número de versión de script permite que el lenguaje de scripting se actualice de una manera compatible con versiones anteriores (es decir, mediante el uso de actualizaciones de bifurcación suave) para introducir nuevos operandos, sintaxis o semánticas de script. La capacidad de actualizar el lenguaje de scripting de manera no disruptiva acelerará en gran medida la velocidad de innovación en bitcoin.

Escalado y almacenamiento de red: habitualmente, los datos testigo contribuyen en gran medida al tamaño total de una transacción. Los scripts más complejos, como los que se utilizan para canales de pago o multifirma son muy grandes. En algunos casos, estos scripts representan la mayoría (más del 75%) de los datos en una transacción. Al mover los datos testigo fuera de la transacción, Segregated Witness mejora la escalabilidad de bitcoin. Los nodos pueden eliminar los datos del testigo después de validar las firmas, o ignorarlos por completo cuando se realiza una verificación de pago simplificado. Los datos de los testigos no necesitan ser transmitidos a todos los nodos y no necesitan ser almacenados en el disco por todos los nodos.

Optimización en la verificación de firmas: Segregated Witness actualiza las funciones de firmas (CHECKSIG, CHECKMULTISIG, etc.) para reducir la complejidad computacional del algoritmo. Antes de segwit, el algoritmo utilizado para producir una firma requería varias operaciones hash que eran proporcionales al tamaño de la transacción. Los cálculos de hashing de datos aumentaban en $O(n^2)$ con respecto al número de operaciones de firma, lo que introduce una carga computacional sustancial en todos los nodos que verifican la firma. Con segwit, el algoritmo se cambia para reducir la complejidad a O(n).

Mejora del firmado fuera de línea: las firmas de Segregated Witness incorporan el valor (cantidad) referenciado por cada entrada en el hash que se firma. Anteriormente, un dispositivo de firma fuera de línea, como una cartera hardware, tenía que verificar la cantidad de cada entrada antes de firmar una transacción. Esto generalmente se logra mediante la transmisión de una gran cantidad de datos sobre las transacciones anteriores a las que se hace referencia como entradas. Dado que la cantidad ahora es parte del hash de compromiso que se firma, un dispositivo fuera de línea no necesita las transacciones anteriores. Si las cantidades no coinciden (están falsificadas por un sistema en línea comprometido), la firma no será válida.

Cómo Funciona Segregated Witness

A primera vista, Segregated Witness parece ser un cambio en la forma en que se construyen las transacciones y, por lo tanto, en una función de nivel de transacción, pero no lo es. Más bien, Segregated Witness es un cambio en la forma en que se gastan los UTXO individuales y, por lo tanto, es una característica de cada salida.

Una transacción puede gastar salidas de Segregated Witness o salidas tradicionales (testigo en línea) o ambas. Por lo tanto, no tiene mucho sentido referirse a una transacción como una "transacción de Segregated Witness". Más bien, deberíamos referirnos a salidas específicas de transacciones como "salidas de Segregated Witness".

Cuando una transacción gasta un UTXO, debe proporcionar un testigo. En un UTXO tradicional, el script de bloqueo requiere que se proporcionen datos testigo *en línea* en la parte de entrada de la transacción que gasta el UTXO. Un UTXO de Segregated Witness, sin embargo, especifica un script de bloqueo que puede satisfacerse con datos testigo fuera de la entrada (segregado).

Bifurcación suave (Compatibilidad hacia Atrás)

Segregated Witness es un cambio significativo en la forma en que se diseñan las salidas y las transacciones. Un cambio de este tipo normalmente requeriría un cambio simultáneo en cada nodo y cartera de bitcoin para modificar las reglas de consenso— lo que se conoce como una bifurcación fuerte (en inglés, hard fork). En cambio, segregated witness se introduce con un cambio mucho menos perturbador, que es compatible con versiones anteriores, conocido como bifurcación suave (en inglés, soft fork). Este tipo de actualización permite que el software no actualizado ignore los cambios y continúe operando sin interrupciones.

Las salidas de Segregated Witness se construyen de modo que los sistemas más antiguos que todavía no son conscientes de segwit puedan validarlos. Para una cartera o nodo antiguo, una salida de Segregated Witness parece una salida que

"cualquiera puede gastar". Dichas salidas se pueden gastar con una firma vacía, por lo tanto, el hecho de que no haya una firma dentro de la transacción (está segregada) no invalida la transacción. Sin embargo, los monederos y nodos de minería más nuevos ven la salida de Segregated Witness y esperan encontrar un testigo válido para ello en los datos testigo de la transacción.

Salida de Segregated Witness y Ejemplos de Transacciones

Veamos algunas de nuestras transacciones de ejemplo y veamos cómo cambiarían con Segregated Witness. Primero veremos cómo se transforma un pago de Pago-a-Hash-de-Llave-Pública (Pay-to-Public-Key-Hash, P2PKH) con el programa de Segregated Witness. Después, veremos el equivalente de Segregated Witness para los scripts Pago-a-Hash-de-Script (Pay-to-Script-Hash, P2SH). Finalmente, veremos cómo los dos programas anteriores de Segregated Witness se pueden integrar dentro de un script P2SH.

Pago-al-Testigo-Hash-de-Llave-Pública (Pay-to-Witness-Public-Key-Hash, P2WPKH)

En Comprando una Taza de Café, Alice creó una transacción para pagarle a Bob una taza de café. Esa transacción creó una salida P2PKH con un valor de 0.015 BTC que Bob podía gastar. El script de salida se ve así:

Ejemplo de script de salida P2PKH

```
DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 EQUALVERIFY CHECKSIG
```

Con Segregated Witness, Alice crearía un script Pay-to-Witness-Public-Key-Hash (P2WPKH), que se ve así:

Ejemplo de script de salida de P2WPKH

```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

Como puede verse, el script de bloqueo en la salida de un testigo segregado es mucho más simple que una salida tradicional. Consiste en dos valores que se insertan en la pila de evaluación de scripts. Para un cliente bitcoin no actualizado (que no esté al tanto del testigo segregado), los dos ingresos a la pila se verían como una salida que cualquiera puede gastar y no que requiere de una firma (o más bien, puede gastarse con una firma vacía). Para un cliente actualizado, consciente del testigo segregado, el primer número (0) se interpreta como un número de versión (la *versión del testigo*) y el segundo dato (de 20 bytes) es el equivalente de un script de bloqueo conocido como *programa de testigo*. El programa de testigos de 20 bytes es simplemente el hash de la llave pública, como en un script P2PKH.

Ahora, veamos la transacción correspondiente que Bob usa para gastar esta salida. Para el script original (no-segwit), la transacción de Bob debería incluir una firma en la entrada de la transacción:

Transacción decodificada que muestra el gasto de una salida P2PKH con una firma

```
[...]
“Vin” : [
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
"vout": 0,
         "scriptSig": “<scriptSig de Bob>”,
]
[...]
```

Sin embargo, para gastar la salida de Segregated Witness, la transacción no tiene firma en esa entrada. En cambio, la transacción de Bob tiene un scriptSig vacío e incluye un Segregated Witness, fuera de la transacción en sí:

Transacción decodificada que muestra una salida P2WPKH que se gasta con datos testigo separados

```
[...]
“Vin” : [
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",
"vout": 0,
         "scriptSig": “”,
]
[...]
“witness”: “<datos testigo de Bob>”
[...]
```

Construcción de cartera de P2WPKH

Es extremadamente importante tener en cuenta que P2WPKH solo debe crearlo el beneficiario (destinatario) y no debe convertirlo el remitente a partir de una llave pública conocida, un script P2PKH o una dirección. El remitente no tiene

forma de saber si la cartera del destinatario tiene la capacidad de construir transacciones segwit y gastar salidas P2WPKH.

Además, las salidas P2WPKH deben construirse a partir del hash de una llave pública *comprimida*. Las llaves públicas no comprimidas no son estándar en segwit y pueden deshabilitarse explícitamente en una futura bifurcación suave. Si el hash utilizado en el P2WPKH provino de una llave pública no comprimida, es posible que no se pueda gastar y que pierdas fondos. Las salidas P2WPKH deben ser creadas por la cartera del beneficiario al derivar una llave pública comprimida a partir su llave privada.

WARNING

P2WPKH debe ser construido por el beneficiario (destinatario) convirtiendo una llave pública comprimida en un hash P2WPKH. Nunca debes transformar un script P2PKH, una dirección bitcoin o una llave pública no comprimida en un script testigo P2WPKH.

Pago-a-Testigo-Hash-de-Script (Pay-to-Witness-Script-Hash, P2WSH)

El segundo tipo de programa testigo corresponde a un script Pay-to-Script-Hash (P2SH). Vimos este tipo de script en Pay-to-Script-Hash (P2SH). En ese ejemplo, se utilizó P2SH en la empresa de Mohammed para expresar un script de multifirma. Los pagos a la empresa de Mohammed se codificaron con un script de bloqueo como este:

Ejemplo de script de salida P2SH

```
HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL
```

Este script P2SH hace referencia al hash de un *script de canje* que define un requisito de multifirma 2-de-3 para gastar fondos. Para gastar esta salida, la compañía de Mohammed presentaría el script de canje (cuyo hash coincide con el hash del script en la salida P2SH) y las firmas necesarias para satisfacer ese script de canje, todo dentro de la entrada de la transacción:

Transacción decodificada que muestra el gasto de una salida P2SH

```
[...]
“Vin” : [
"txid": "abcdef12345...",
"vout": 0,
         "scriptSig": “<Firma A> <Firma B> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”,
]
```

Ahora, veamos cómo este ejemplo completo se actualizaría a segwit. Si los clientes de Mohammed estuvieran usando una cartera compatible con segwit, harían un pago, creando una salida Pay-to-Witness-Script-Hash (P2WSH) que se vería así:

Ejemplo de un script de salida P2WSH

```
0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89
```

Nuevamente, como en el ejemplo de P2WPKH, puedes ver que el script equivalente de Segregated Witness es mucho más simple y omite los diversos operandos de script que se ven en los scripts de P2SH. En su lugar, el programa Segregated Witness consta de dos valores que se empujan en la pila: una versión testigo (0) y el hash SHA256 de 32 bytes del script de canje.

TIP

Mientras que P2SH usa el hash de 20 bytes: RIPEMD160 (SHA256 (script)), el programa testigo P2WSH usa un hash de 32 bytes: SHA256 (script). Esta diferencia en la selección del algoritmo de hash es deliberada y se usa para diferenciar los dos tipos de programas testigos (P2WPKH y P2WSH) por la longitud del hash y para proporcionar mayor seguridad a P2WSH (128 bits de seguridad en P2WSH versus 80 bits de seguridad en P2SH).

La compañía de Mohammed puede gastar la salida P2WSH presentando el script de canje correcto y las firmas suficientes para satisfacerlo. Tanto el script de canje como las firmas se segregarían *fuera* de la transacción de gastos como parte de la data del testigo. Dentro de la entrada de la transacción, la billetera de Mohammed pondría un scriptSig vacío:

1. Transacción codificada que muestra una salida P2WSH que se gasta con datos de testigos separados

```
[...]
“Vin” : [
"txid": "abcdef12345...",
```

```
"vout": 0,
         "scriptSig": “”,
]
[...]
“witness”: “<Firma A> <Firma B> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”
[...]
```

Diferenciando entre P2WPKH y P2WSH

En las dos secciones anteriores, demostramos dos tipos de programas de testigo: Pago-al-Testigo-Hash-de-Llave-Pública (Pay-to-Witness-Public-Key-Hash, P2WPKH) y Pago-a-Testigo-Hash-de-Script (Pay-to-Witness-Script-Hash, P2WSH). Ambos tipos de programas de testigo consisten en un número de versión de un solo byte seguido de un hash más largo. Se ven muy similares, pero se interpretan de manera muy diferente: uno se interpreta como un hash de llave pública, que se satisface con una firma y el otro como un hash de script, que se satisface con un script de canje. La diferencia crítica entre ellos es la longitud del hash:

- El hash de llave pública en P2WPKH es de 20 bytes
- El hash de script in P2WSH es de 32 bytes

Esta es la única diferencia que permite que una cartera diferencie entre los dos tipos de programas testigo. Al observar la longitud del hash, una cartera puede determinar qué tipo de programa testigo es, P2WPKH o P2WSH.

Actualización a Segregated Witness

Como podemos ver en los ejemplos anteriores, la actualización a Segregated Witness es un proceso de dos pasos. Primero, las carteras deben crear salidas especiales de tipo segwit. Después, estas salidas pueden ser gastadas por carteras que saben cómo construir transacciones de Segregated Witness. En los ejemplos, la cartera de Alice era consciente de segwit y era capaz de crear salidas especiales con scripts de Segregated Witness. La cartera de Bob también es consciente de segwit y es capaz de gastar esas salidas. Lo que puede no ser obvio en el ejemplo es que, en la práctica, la cartera de Alice debe *saber* que Bob usa una cartera que es consciente de segwit y puede gastar esas salidas. De lo contrario, si la cartera de Bob no se actualiza y Alice intenta realizar pagos de segwit a Bob, la cartera de Bob no podrá detectar estos pagos.

> **TIP** Para los tipos de pago P2WPKH y P2WSH, tanto las carteras del remitente como del destinatario deben actualizarse para poder utilizar segwit. Además, la cartera del remitente debe saber que la cartera del destinatario es consciente de segwit.

Segregated Witness no se implementará simultáneamente en toda la red. Por el contrario, Segregated Witness se implementa como una actualización compatible con versiones anteriores, donde *clientes antiguos y nuevos* pueden coexistir. Los desarrolladores de carteras actualizarán el software de cartera de forma independiente para agregar capacidades de segwit. Los tipos de pago P2WPKH y P2WSH se utilizan cuando tanto el remitente como el destinatario son conscientes de segwit. Los P2PKH y el P2SH tradicionales continuarán funcionando en las carteras no actualizadas. Así emergen dos escenarios importantes, que se abordan en la siguiente sección:

- Capacidad de la cartera de un remitente que no es consciente de segwit para realizar un pago a la cartera del destinatario que puede procesar transacciones segwit.
- Capacidad de la cartera de un remitente que es consciente de segwit para reconocer y distinguir entre destinatarios que son conscientes de segwit y los que no lo son, por sus *direcciones*.

Integración de Segregated Witness dentro de P2SH

Supongamos, por ejemplo, que la cartera de Alice no se actualiza a segwit, pero la cartera de Bob se actualiza y puede manejar transacciones segwit. Alice y Bob pueden usar transacciones "antiguas" no segwit. Pero es probable que Bob quiera usar Segwit para reducir las comisiones de transacción, aprovechando el descuento que se aplica a los datos testigo.

En este caso, la cartera de Bob puede construir una dirección P2SH que contenga un script segwit en su interior. La cartera de Alice ve esto como una dirección P2SH "normal" y puede realizar pagos sin ningún conocimiento de segwit. La cartera de Bob puede gastar este pago con una transacción de segwit, aprovechando segwit al máximo y reduciendo las comisiones de transacción.

Ambas formas de scripts de testigos, P2WPKH y P2WSH, se pueden integrar dentro de una dirección P2SH. Al primero se le referiere como P2SH (P2WPKH) y al segundo se le refiere como P2SH (P2WSH).

Pay-to-Witness-Public-Key-Hash dentro de Pay-to-Script-Hash (Pago-a-Testigo-Hash-de-Llave-Pública dentro de Pago-a-Hash-de-Script)

La primera forma de script de testigo que examinaremos es P2SH (P2WPKH). Este es un programa de testigo Pay-to-Witness-Public-Key-Hash, integrado dentro de un script de Pay-to-Script-Hash, de modo que pueda ser utilizado por una cartera que no tenga conocimiento de segwit.

La cartera de Bob construye un programa testigo P2WPKH con la llave pública de Bob. Este programa testigo se hashea y el hash resultante se codifica como un script P2SH. El script P2SH se convierte en una dirección bitcoin, una que comienza con un "3", como vimos en la sección Pay-to-Script-Hash (P2SH).

La cartera de Bob comienza con el programa testigo P2WPKH que vimos anteriormente:

El programa testigo P2WPKH de Bob

```
0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7
```

El programa testigo P2WPKH consta de la versión testigo y el hash de llave pública de 20 bytes de Bob.

Después, la cartera de Bob hashea el programa testigo anterior, primero con SHA256, luego con RIPEMD160, produciendo otro hash de 20 bytes.

Usemos bx desde la línea de comandos para replicar eso:

HASH160 del programa testigo P2WPKH

```
echo \
'0 [ab68025513c3dbd2f7b92a94e0581f5d50f654e7]'\
 | bx script-encode | bx sha256 | bx ripemd160
3e0547268b3b19288b3adef9719ec8659f4b2b0b
```

A continuación, el hash de script de canje se convierte en una dirección bitcoin. Usemos de nuevo bx desde la línea de comando:

Dirección P2SH

```
echo \
'3e0547268b3b19288b3adef9719ec8659f4b2b0b' \
| bx address-encode -v 5
37Lx99uaGn5avKBxiW26HjedQE3LrDCZru
```

Ahora, Bob puede mostrar esta dirección para que los clientes paguen por su café. La cartera de Alice puede realizar un pago a 37Lx99uaGn5avKBxiW26HjedQE3LrDCZru, tal como lo haría con cualquier otra dirección bitcoin.

Para pagar a Bob, la cartera de Alice bloquearía la salida con un script P2SH:

```
HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL
```

Aunque la cartera de Alice no tiene soporte para segwit, el pago que crea puede ser gastado por Bob con una transacción segwit.

Pay-to-Witness-Script-Hash dentro de Pay-to-Script-Hash (Pago-a-Testigo-Hash-de-Script dentro de Pago-a-Hash-de-Script)

De manera similar, un programa testigo P2WSH para un script multifirma o cualquier otro script complicado se puede integrar dentro de un script y dirección P2SH, lo que hace posible que cualquier cartera realice pagos que sean compatibles con segwit.

Como vimos en Pago-a-Testigo-Hash-de-Script (Pay-to-Witness-Script-Hash, P2WSH), la compañía de Mohammed está usando pagos del Testigo Segregado con scripts de múltiples firmas. Para hacer posible que cualquier cliente le pague a su compañía, independientemente de si sus billeteras están actualizadas para el Testigo Segregado, la billetera de Mohammed puede incrustar el programa de testigo P2WSH dentro de un script P2SH.

Primero, la cartera de Mohammed hashea el script de canje con SHA256 (solo una vez). Usemos bx para hacer eso desde la línea de comandos:

La cartera de Mohammed crea un programa testigo P2WSH

```
echo \
2 \
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF
5DE762C587] \
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E342248
58008E8B49] \
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD3
89D9977965] \
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE
5C2AFD94A5] \
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7E
D238F4D800] \
5 CHECKMULTISIG \
| bx script-encode | bx sha256
9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

A continuación, el script de canje hasheado se convierte en un programa testigo P2WSH:

```
0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73
```

Después, el programa testigo en sí mismo se hashea con SHA256 y RIPEMD160, produciendo un nuevo hash de 20 bytes, tal como se usa en el P2SH tradicional. Usemos bx desde la línea de comandos para hacer eso:

El HASH160 del programa testigo P2WSH

```
 echo \
'0 [9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73]'\
 | bx script-encode | bx sha256 | bx ripemd160
86762607e8fe87c0c37740cddee880988b9455b2
```

A continuación, la cartera construye una dirección bitcoin P2SH a partir de este hash. Nuevamente, usamos bx para calcular desde la línea de comando:

Dirección bitcoin P2SH

```
echo \
'86762607e8fe87c0c37740cddee880988b9455b2'\
 | bx address-encode -v 5
3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG
```

Ahora, los clientes de Mohammed pueden realizar pagos a esta dirección sin necesidad de soportar segwit. Para enviar un pago a Mohammed, una cartera bloquearía la salida con el siguiente script P2SH:

Script P2SH usado para bloquear pagos al multifirma de Mohammed

```
HASH160 86762607e8fe87c0c37740cddee880988b9455b2 EQUAL
```

La compañía de Mohammed puede después construir transacciones segwit para gastar estos pagos, aprovechando las características de segwit que incluyen comisiones de transacción más bajas.

Direcciones de Segregated Witness

Incluso después de la activación del Testigo Segregado, tomará algún tiempo hasta que se actualicen la mayoría de las billeteras. Al principio, Testigo Segregado se integrará en un P2SH, como vimos en la sección anterior, para facilitar la compatibilidad entre billeteras al tanto del Testigo Segregado y las que no estén al tanto.

Sin embargo, una vez que las carteras sean ampliamente compatibles con segwit, tiene sentido codificar scripts de testigos directamente en un formato de dirección nativo diseñado para segwit, en lugar de integrarlo en P2SH.

El formato nativo de direcciones segwit se define en BIP-173:

BIP-173

> Base32 address format for native v0-16 witness outputs

BIP-173 solo codifica los scripts del testigo (P2WPKH y P2WSH). No es compatible con scripts P2PKH o P2SH que no sean acordes al Testigo Segregado. BIP-173 es una codificación Base32 con suma de comprobación, en comparación con la

codificación Base58 de una dirección bitcoin "tradicional". Las direcciones BIP-173 también se llaman direcciones *bech32*, pronunciadas "bej-che treinta y dos", aludiendo al uso de un algoritmo de detección de errores "BCH" y un conjunto de codificación de 32 caracteres.

Las direcciones BIP-173 utilizan un set de 32 caracteres alfanuméricos en minúsculas, seleccionados cuidadosamente para reducir errores de lectura o de escritura errónea. Al elegir un conjunto de caracteres en minúsculas, bech32 es más fácil de leer, deletrear y un 45% más eficiente para la codifcación en códigos QR.

El algoritmo de detección de errores BCH es una gran mejora con respecto al algoritmo de checksum anterior (de Base58Check), ya que permite no solo la detección sino también la corrección de errores. Las interfaces de entrada de direcciones (como las cajas de texto en los formularios) pueden detectar y resaltar qué carácter fue mal escrito cuando detectan un error.

A partir de la especificación BIP-173, aquí tienes algunos ejemplos de direcciones bech32:

Mainnet P2WPKH

bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4

Testnet P2WPKH

tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx

Mainnet P2WSH

bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3

Testnet P2WSH

tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7

Como puede ver en estos ejemplos, una cadena segwit bech32 tiene hasta 90 caracteres y consta de tres partes:

La parte legible para humanos

este prefijo "bc" o "tb" identifica mainnet o testnet.

El separador

el dígito "1", que no forma parte del set de codificación de 32 caracteres y solo puede aparecer en esta posición como separador

La parte de datos

Un mínimo de 6 caracteres alfanuméricos, el script de testigo codificado con checksum

En este momento, solo unas pocas carteras aceptan o producen direcciones bech32 segwit nativas, pero a medida que aumenta la adopción de segwit, se verán cada vez más a menudo.

Identificadores de transacción

Uno de los mayores beneficios de Segregated Witness es que elimina la maleabilidad de las transacciones de terceros.

Antes de segwit, las firmas de las transacciones podían ser sutilmente modificadas por terceros, cambiando su ID de transacción (hash) sin cambiar ninguna propiedad fundamental (entradas, salidas, cantidades). Esto creó oportunidades para ataques de denegación de servicio, así como ataques contra software de carteras mal programadas que presuponían erróneamente que los hashes de transacción no confirmados eran inmutables.

Con la activación de Segregated Witness, las transacciones tienen dos identificadores, txid y wtxid. El ID de transacción tradicional txid es el hash doble SHA256 de la transacción serializada, sin los datos del testigo. El wtxid de una transacción es el hash doble SHA256 del nuevo formato de serialización de la transacción con dato testigo.

El txid tradicional se calcula exactamente de la misma manera que con una transacción no segwit. Sin embargo, dado que la transacción segwit tiene los scriptSig s vacíos para todas las entradas, no hay ninguna parte de la transacción que pueda ser modificada por un tercero. Por lo tanto, en una transacción segwit, el txid es inmutable por un tercero, incluso cuando la transacción no está confirmada.

El wtxid es como un ID "extendido", ya que el hash también incorpora los datos del testigo. Si una transacción se transmite

sin datos de testigos, entonces wtxid y txid deben ser idénticos. Tenga en cuenta que dado que wtxid incluye datos de testigos (firmas) y dado que los datos de testigos pueden ser maleables, el wtxid debe considerarse maleable hasta que se confirme la transacción. Solo el txid de una transacción a corde al Testigo Segregado puede considerarse inmutable por terceros y solo si *todas* las entradas de la transacción son entradas acordes al testigo Segregado.

TIP

Las transacciones de Segregated Witness tienen dos IDs: txid y wtxid. El txid es el hash de la transacción sin los datos testigo y el wtxid es el hash que incluye los datos testigo. El txid de una transacción donde todas las entradas son entradas segwit no es susceptible a la maleabilidad de la transacción por parte de terceros.

Nuevo Algoritmo de Firma de Segregated Witness

Segregated Witness modifica la semántica de las cuatro funciones de verificación de firma (CHECKSIG, CHECKSIGVERIFY, CHECKMULTISIG, y CHECKMULTISIGVERIFY), cambiando la forma en que se calcula un hash de compromiso de transacción.

Las firmas en transacciones bitcoin se aplican a un *hash de compromiso*, que se calcula a partir de los datos de transacción, bloqueando partes específicas de los datos que indican el compromiso del firmante con esos valores. Por ejemplo, en una firma de tipo SIGHASH_ALL simple, el hash de compromiso incluye todas las entradas y salidas.

Desafortunadamente, la forma en que se calculó el hash de compromiso introdujo la posibilidad de que un nodo que verifique la firma pueda ser forzado a realizar un número significativo de cálculos de hash. Específicamente, las operaciones de hash aumentan en $O(n^2)$ con respecto al número de operaciones de firma en la transacción. Por lo tanto, un atacante podría crear una transacción con una gran cantidad de operaciones de firma, lo que provocaría que toda la red bitcoin tuviera que realizar cientos o miles de operaciones hash para verificar la transacción.

Segwit representó una oportunidad para abordar este problema cambiando la forma en que se calcula el hash de compromiso. Para los programas testigo de segwit versión 0, la verificación de firmas se realiza utilizando un algoritmo hash de compromiso mejorado como se especifica en BIP-143.

El nuevo algoritmo logra dos objetivos importantes. En primer lugar, el número de operaciones de hash aumenta mucho más gradualmente, en O(n) al número de operaciones de firma, lo que reduce la oportunidad de crear ataques de denegación de servicio con transacciones demasiado complejas. En segundo lugar, el hash de compromiso ahora también incluye el valor (cantidades) de cada entrada como parte del compromiso. Esto significa que un firmante puede comprometerse con un valor de entrada específico sin necesidad de "buscar" y verificar la transacción anterior a la que hace referencia la entrada. En el caso de los dispositivos sin conexión, como las carteras de hardware, esto simplifica enormemente la comunicación entre el host y la cartera de hardware, eliminando la necesidad de transmitir transacciones anteriores para su validación. Una cartera hardware puede aceptar el valor de entrada "como se indica" por un host no confiable. Dado que la firma no es válida si el valor de entrada no es correcto, la cartera hardware no necesita validar el valor antes de firmar la entrada.

Incentivos Económicos para Segregated Witness

Los nodos de minería de bitcoin y los nodos completos incurren en costos por los recursos utilizados para soportar la red bitcoin y la cadena de bloques. A medida que aumenta el volumen de transacciones de bitcoin, también aumenta el costo de los recursos (CPU, ancho de banda de la red, espacio en disco, memoria). Los mineros son compensados por estos costos a través de comisiones que son proporcionales al tamaño (en bytes) de cada transacción. Los nodos completos que no son mineros no son compensandos, por lo que incurren en estos costos porque tienen la necesidad de ejecutar un nodo de índice completo con validación autoritativa, tal vez porque utilizan el nodo para operar un negocio de bitcoin.

Sin comisiones de transacción, el crecimiento en los datos de bitcoin sin duda aumentaría dramáticamente. Las comisiones se destinan a alinear las necesidades de los usuarios de bitcoin con la carga que sus transacciones imponen en la red, a través de un mecanismo de descubrimiento de precios basado en el mercado.

El cálculo de las comisiones según el tamaño de la transacción trata todos los datos de la transacción como iguales en costo. Pero desde la perspectiva de los nodos completos y los mineros, algunas partes de una transacción tienen costos mucho más altos. Cada transacción agregada a la red bitcoin afecta el consumo de cuatro recursos en los nodos:

Espacio en disco

Cada transacción se almacena en la cadena de bloques, lo que aumenta el tamaño total de la cadena de bloques. La cadena de bloques se almacena en el disco, pero el almacenamiento se puede optimizar "recortando" las transacciones

anteriores.

CPU

Cada transacción debe ser validada, lo que requiere tiempo de CPU.

Ancho de banda

Cada transacción se transmite (a través de propagación por inundación) a través de la red al menos una vez. Sin ninguna optimización en el protocolo de propagación de bloques, las transacciones se transmiten nuevamente como parte de un bloque, duplicando el impacto en la capacidad de la red.

Memoria

Los nodos que validan las transacciones mantienen el índice UTXO o todo el set UTXO configurado en memoria para acelerar la validación. Debido a que la memoria es al menos un orden de magnitud más costosa que el disco, el crecimiento del conjunto UTXO contribuye de manera desproporcionada al costo de ejecutar un nodo.

Como puedes ver en la lista, no todas las partes de una transacción tienen el mismo impacto en el costo de ejecutar un nodo o en la capacidad de escala de bitcoin para admitir más transacciones. La parte más costosa de una transacción son las salidas recién creadas, ya que se agregan al set UTXO en memoria. En comparación, las firmas (también conocidas como datos testigo) agregan la menor carga a la red y al costo de ejecutar un nodo, ya que los datos testigo solo se validan una vez y luego no se vuelven a utilizar. Además, inmediatamente después de recibir una nueva transacción y validar los datos testigo, los nodos pueden descartar esos datos testigo. Si las comisiones se calcularan según el tamaño de la transacción, sin discriminar entre estos dos tipos de datos, entonces los incentivos de mercado de las comisiones no estarían alineados con los costos reales impuestos por una transacción. De hecho, la estructura de comisiones actual en realidad fomenta el comportamiento opuesto, porque los datos de testigo son la mayor parte de una transacción.

Los incentivos creados por las comisiones son importantes porque afectan al comportamiento de las carteras. Todas las carteras deben implementar alguna estrategia para ensamblar transacciones que tengan en cuenta una serie de factores, como la privacidad (reduciendo la reutilización de direcciones), la fragmentación (generando mucho cambio) y las comisiones. Si las comisiones motivan de manera abrumadora que las carteras usen la menor cantidad posible de entradas en las transacciones, esto puede llevar a seleccionar UTXO y a estrategias de direcciones de cambio que aumenten inadvertidamente el set UTXO.

Las transacciones consumen UTXO en sus entradas y crean nuevas UTXO con sus salidas. Por lo tanto, una transacción que tiene más entradas que salidas dará como resultado una disminución en el set UTXO, mientras que una transacción que tiene más salidas que entradas dará como resultado un aumento en el set UTXO. Consideremos la *diferencia* entre entradas y salidas y llamemos a eso “Nuevo-UTXO-neto”. Esa es una métrica importante, ya que nos dice qué impacto tendrá una transacción en el recurso más costoso de la red, el conjunto UTXO en memoria. Una transacción con un Nuevo-UTXO-neto positivo se suma a esa carga. Una transacción con un Nuevo-UTXO-neto negativo reduce la carga. Por lo tanto, deseamos fomentar las transacciones que sean negativas en Nuevo-UTXO-neto o neutrales con cero Nuevo-UTXO-neto.

Veamos un ejemplo de qué incentivos se crean mediante el cálculo de las comisiones de transacción, con y sin Segregated Witness. Vamos a ver dos transacciones diferentes. La transacción A es una transacción de 3 entradas y 2 salidas, que tiene una métrica Nuevo-UTXO-neto de –1, lo que significa que consume una UTXO más de la que crea, reduciendo el set UTXO en una unidad. La transacción B es una transacción de 2 entradas y 3 salidas, que tiene una métrica Nuevo-UTXO-neto, lo que significa que agrega una UTXO al set UTXO, lo que impone un costo adicional en toda la red bitcoin. Ambas transacciones utilizan scripts de multifirma (2-de-3) para demostrar cómo los scripts complejos aumentan el impacto de segregated witness en las comisiones. Asumamos unas comisiones de transacción de 30 satoshi por byte y un descuento del 75% en los datos testigo:

Sin Segregated Witness

Comisión de transacción A: 25,710 satoshi

Comisión de transacción B: 18,990 satoshi

Con Segregated Witness

Comisión de transaction A: 8,130 satoshi

Comisión de Transacción B: 12,045 satoshi

Ambas transacciones son menos costosas cuando se implementa segregated witness. Pero comparando los costos entre las dos transacciones, vemos que antes de Segregated Witness, la comisión es más alta para la transacción que tiene un Nuevo-UTXO-neto negativo. Después de Segregated Witness, las comisiones de transacción se alinean con el incentivo para minimizar la creación de nuevos UTXO al no penalizar inadvertidamente las transacciones con muchas entradas.

Segregated Witness, por lo tanto, tiene dos efectos principales en las comisiones pagadas por los usuarios de bitcoin. En primer lugar, segwit reduce el costo general de las transacciones al descontar los datos testigo y aumentar la capacidad de la cadena de bloques de bitcoin. En segundo lugar, el descuento de segwit en los datos testigo corrige una desalineación de incentivos que podría haber creado, sin querer, más hinchazón en el set UTXO.

La Red Bitcoin

Arquitectura de Red Entre Pares (P2P)

Bitcoin se estructura como una arquitectura de red P2P sobre Internet. El término de igual a igual, o P2P, significa que las computadoras que participan en la red son iguales entre sí, que no hay nodos "especiales", y que todos los nodos comparten la carga de proveer servicios a la red. Los nodos de la red se interconectan en una malla de redes con una topología "plana". No hay servidor, no hay servicio centralizado, ni jerarquía dentro de la red. Los nodos en una red P2P proporcionan servicios y consumen servicios al mismo tiempo, con la reciprocidad como incentivo para participar. Las redes P2P son inherentemente resistentes, decentralizadas y abiertas. Un ejemplo destacado de una arquitectura de red P2P fue la internet temprana, donde los nodos de la red IP eran iguales. La estructura del internet actual es más jerárquica, pero el Protocolo de Internet mantiene la esencia de topología plana. Más allá de bitcoin, la más grande y exitosa aplicación de tecnologías P2P es compartir archivos, con Napster como pionera y la red BitTorrent como la evolución más reciente de la arquitectura.

La arquitectura de red entre pares (P2P) de bitcoin es mucho más que una elección topológica. Bitcoin es un sistema P2P de dinero en efectivo por diseño, y la arquitectura de red es tanto una reflexión y un fundamento de esa característica base. La descentralización del control es un principio de diseño base y eso solo puede alcanzarse y mantenerse mediante una red P2P de consenso plana y descentralizada.

El término "red bitcoin" se refiere a la colección de nodos que ejecutan el protocolo p2p bitcoin. Además del protocolo P2P bitcoin, hay otros protocolos tales como Stratum, que se utilizan para la minería y carteras ligeras o móviles. Estos protocolos adicionales son proporcionados por servidores de enrutamiento de puerta de enlace que acceden a la red bitcoin utilizando el protocolo P2P bitcoin, y que luego se extienden por esa red de nodos que ejecutan otros protocolos. Por ejemplo, los servidores Stratum conectan los nodos de minería Stratum través del protocolo Stratum a la red bitcoin principal y hacen de puente entre el protocolo Stratum y el protocolo P2P bitcoin. Utilizamos el término "red bitcoin extendida" para referirnos a la red global que incluye el protocolo p2p bitcoin, los protocolos de pool de minería, el protocolo de Stratum, y cualesquiera otros protocolos relacionados que conectan los componentes del sistema de bitcoin.

Tipos de Nodos y Roles

Aunque los nodos en la red P2P bitcoin son iguales, puede que asuman roles distintos dependiendo de la funcionalidad que soporten. Un nodo bitcoin es una colección de funciones: enrutamiento, la base de datos de la cadena de bloques (en inglés, "blockchain"), minería y servicios de cartera. Un nodo completo con todas estas funciones se detalla en Un nodo de la red bitcoin con todas sus cuatro funciones: cartera, minero, base de datos de cadena de bloques completa, y enrutamiento de red.

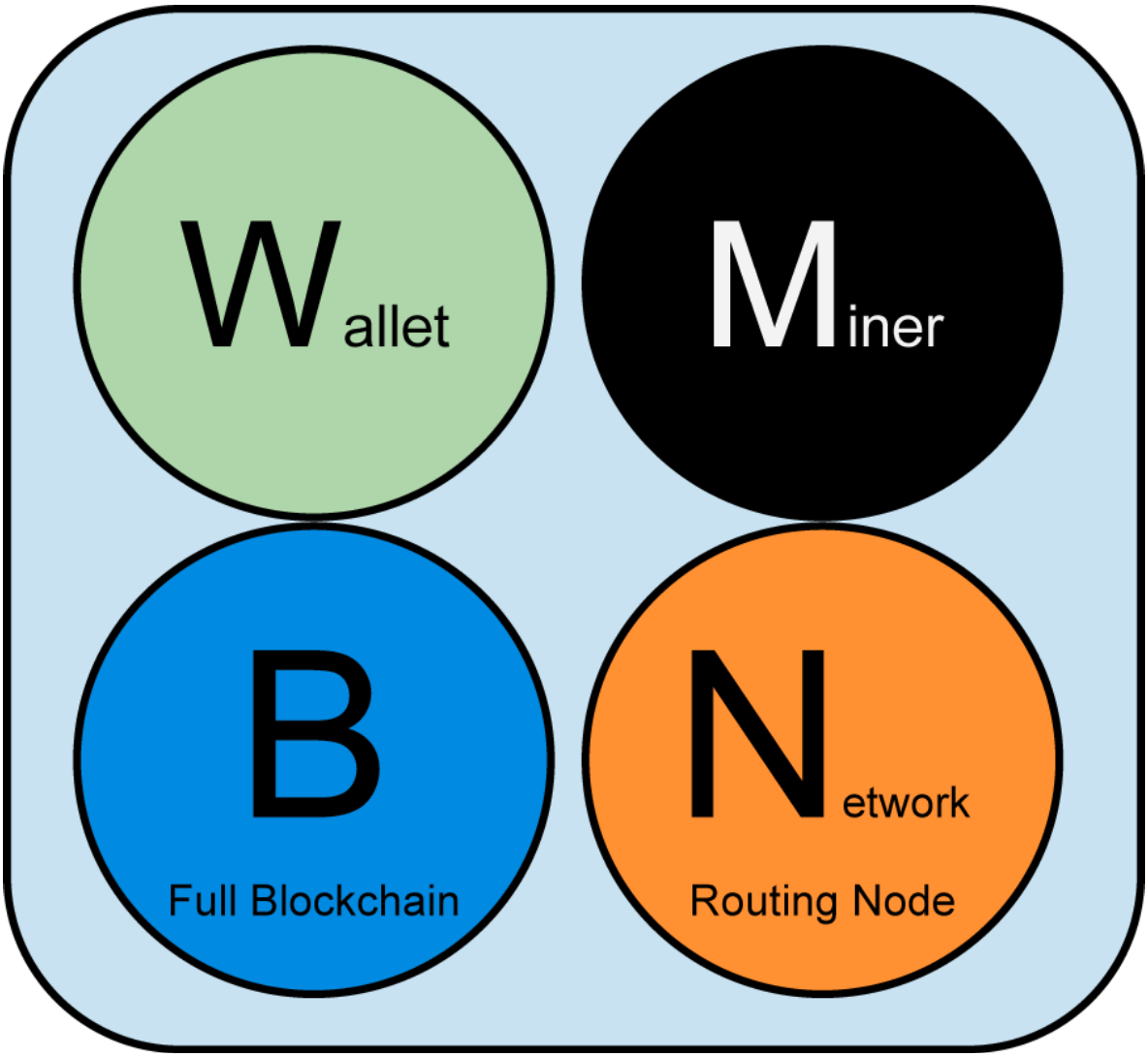


Figure 47. Un nodo de la red bitcoin con todas sus cuatro funciones: cartera, minero, base de datos de cadena de bloques completa, y enrutamiento de red

Todos los nodos incluyen la función de enrutamiento para participar en la red y pueden incluir otra funcionalidad. Todos los nodos validan y propagan transacciones y bloques, y descubren y mantienen conexiones con sus pares. En el ejemplo de nodo completo en Un nodo de la red bitcoin con todas sus cuatro funciones: cartera, minero, base de datos de cadena de bloques completa, y enrutamiento de red, la función de enrutamiento se indica mediante un círculo llamado "Nodo de enrutamiento de red" o con la letra "N".

Algunos nodos, denominados nodos completos, también mantienen una completa y actualizada copia de la cadena de bloques. Los nodos completos pueden verificar cualquier transacción de forma autónoma, concluyente y sin referencia externa. Algunos nodos mantienen solo un subconjunto de la cadena de bloques y verifican las transacciones utilizando un método llamado *verificación de pago simplificado* (en inglés, *simplified payment verification*), o SPV. Estos nodos se conocen como nodos SPV o nodos ligeros. En el nodo completo en la figura de ejemplo, la función de base de datos de la cadena de bloques de nodo completo se indica mediante un círculo llamado "Full Blockchain" o la letra "B". En La red bitcoin extendida muestra varios tipos de nodos, puertas de enlace y protocolos, los nodos SPV se dibujan sin el círculo "B", lo que indica que no tienen una copia completa de la cadena de bloques.

Los nodos de minería compiten para crear nuevos bloques ejecutando hardware especializado para resolver el algoritmo de Prueba-de-Trabajo. Algunos nodos de minería también son nodos completos, manteniendo una copia completa de la cadena de bloques, mientras que otros son nodos livianos que participan en agrupaciones de minería y que dependen de un servidor del grupo para mantener un nodo completo. La función de minería se muestra en el nodo completo como un círculo llamado "Miner" o la letra "M".

Las billeteras de los usuarios pueden ser parte de un nodo completo, como suele ser el caso con los clientes bitcoin de escritorio. Cada vez más, muchas billeteras de usuarios, especialmente las que se ejecutan en dispositivos con recursos limitados, como los teléfonos inteligentes, son nodos SPV. La función de cartera se muestra en <> como un círculo llamado "Wallet" o la letra "W".

Además de los principales tipos de nodos en el protocolo P2P bitcoin, hay servidores y nodos que ejecutan otros protocolos, como los protocolos de pool de minera especializados y protocolos de cliente de acceso ligeros.

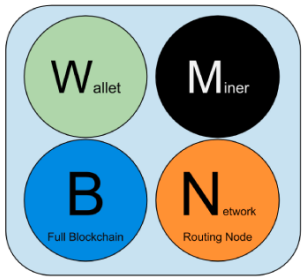
Diferentes tipos de nodos sobre la red bitcoin extendida muestra los tipos de nodos más comunes en la red bitcoin extendida.

La Red Bitcoin Extendida

La red principal de bitcoin, que ejecuta el protocolo P2P de bitcoin, consta de entre 5,000 y 8,000 nodos de escucha que ejecutan varias versiones del cliente de referencia de bitcoin (Bitcoin Core) y unos pocos cientos de nodos que ejecutan otras implementaciones del protocolo P2P de bitcoin, como Bitcoin Classic , Bitcoin Unlimited, BitcoinJ, Libbitcoin, btcd y bcoin. Un pequeño porcentaje de los nodos en la red P2P de bitcoin también son nodos de minería, compitiendo en el proceso de minería, validando transacciones y creando nuevos bloques. Varias grandes compañías se interconectan con la red bitcoin mediante la ejecución de clientes de nodo completo basados en el Cliente Principal de Bitcoin, con copias completas de la cadena de bloques y un nodo de red, pero sin funciones de minería o de cartera. Estos nodos actúan como enrutadores de borde de red, lo que permite que otros servicios (casas de intercambio, carteras, exploradores de bloques, procesadores de pagos para comerciantes) se construyan en la parte superior .

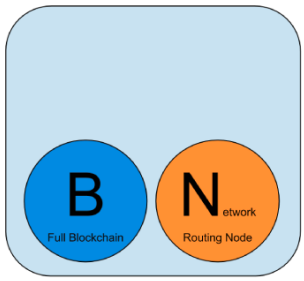
La red bitcoin extendida incluye la red que ejecuta el protocolo P2P bitcoin, descrito anteriormente, así como nodos que ejecutan protocolos especializados. Adjuntos a la red P2P bitcoin principal hay una serie de servidores de pool y pasarelas de protocolo que conectan nodos que ejecutan otros protocolos. Estos otros nodos de protocolo son en su mayoría los nodos de minería del pool (ver Minería y Consenso) y los clientes de carteras ligeros, que no llevan una copia completa de la cadena de bloques.

La red bitcoin extendida muestra varios tipos de nodos, puertas de enlace y protocolos muestra la red bitcoin extendida con los distintos tipos de nodos, servidores de puerta de enlace, los routers de borde, y los clientes de cartera y los distintos protocolos que utilizan para conectarse entre sí.



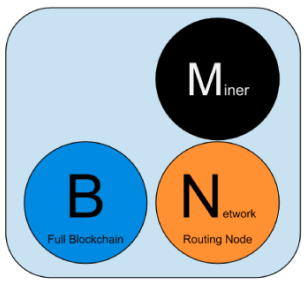
Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



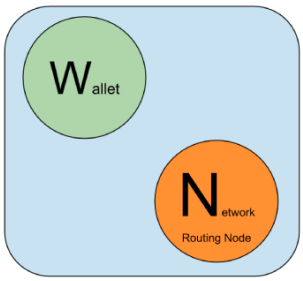
Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



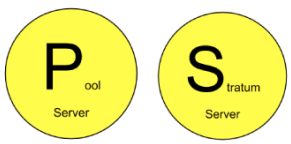
Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.

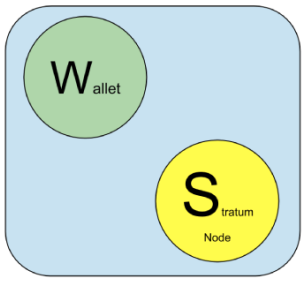


Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.

Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.

Figure 48. Diferentes tipos de nodos sobre la red bitcoin extendida

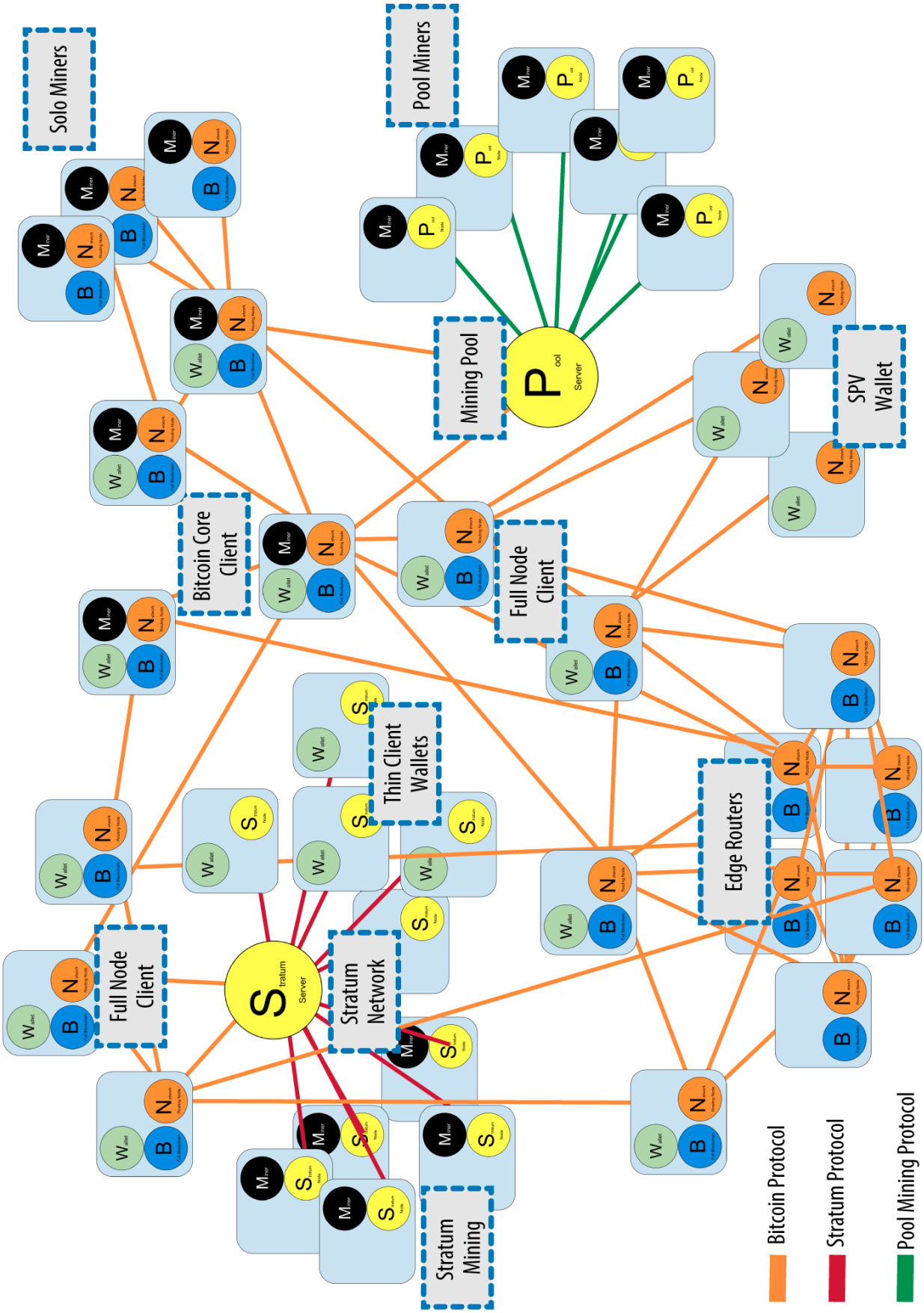


Figure 49. La red bitcoin extendida muestra varios tipos de nodos, puertas de enlace y protocolos

Redes de Retransmisión de Bitcoin

Si bien la red P2P de bitcoin satisface las necesidades generales de una amplia variedad de tipos de nodos, presenta una latencia de red demasiado alta para las necesidades especializadas de los nodos de minería de bitcoin.

Los mineros de bitcoin participan en una competición sensible al tiempo para resolver el problema de la Prueba-de-Trabajo y extender la cadena de bloques (ver Minería y Consenso). Mientras participan en esta competición, los mineros

de bitcoin deben minimizar el tiempo entre la propagación de un bloque ganador y el comienzo de la próxima ronda de competición. En minería, la latencia de la red está directamente relacionada con los márgenes de beneficio.

Una *Red de Retransmisión de Bitcoin* es una red que intenta minimizar la latencia en la transmisión de bloques entre los mineros. La original Red de Retransmisión de Bitcoin fue creada por el desarrollador del núcleo Matt Corallo en 2015 para permitir la rápida sincronización de bloques entre los mineros con una latencia muy baja. La red estaba formada por varios nodos especializados alojados en la infraestructura de los servicios web de Amazon en todo el mundo y servía para conectar a la mayoría de los mineros y los pools de minería.

La Red de Retransmisión de Bitcoin original fue reemplazada en 2016 con la introducción del *Fast Internet Bitcoin Relay Engine* o *FIBRE*, también creado por el desarrollador de núcleo Matt Corallo. FIBRE es una red de retransmisión basada en UDP que retransmite bloques dentro de una red de nodos. FIBRE implementa la optimización de *bloque compacto* para reducir aún más la cantidad de datos transmitidos y la latencia de la red.

Otra red de retransmisión (aún en fase de propuesta) es *Falcon*, basada en investigaciones de la Universidad de Cornell. Falcon utiliza el "enrutamiento directo" en lugar de "almacenar y reenviar" para reducir la latencia al propagar partes de bloques a medida que se reciben, en lugar de esperar hasta que se reciba un bloque completo.

Las redes de retransmisión no son reemplazos para la red P2P de bitcoin. En cambio, son redes superpuestas que proporcionan conectividad adicional entre nodos con necesidades especializadas. Al igual que las autopistas no son reemplazos para carreteras rurales, sino accesos directos entre dos puntos con mucho tráfico, también se necesitan carreteras pequeñas para conectarse a las autopistas.

Descubrimiento de Red

Cuando se inicia un nuevo nodo, debe descubrir otros nodos bitcoin en la red para poder participar. Para iniciar este proceso, un nuevo nodo debe descubrir al menos un nodo existente en la red y conectarse a él. La ubicación geográfica de otros nodos es irrelevante; la topología de la red de bitcoin no está definida geográficamente. Por lo tanto, cualquier nodo bitcoin existente puede ser seleccionado al azar.

Para conectarse a un compañero conocido, los nodos establecen una conexión TCP, por lo general en el puerto 8333 (puerto conocido generalmente como el utilizado por bitcoin), o un puerto alternativo si se proporciona. Al establecer una conexión, el nodo se iniciará un "apretón de manos" (en inglés, "handshake") (ver <<network_handshake> >) mediante la transmisión de un mensaje de versión, que contiene información básica de identificación, incluyendo:

nVersion
: La versión del protocolo P2P de bitcoin que "habla" el cliente (por ejemplo, 70002)

nLocalServices
: Una lista de los servicios locales soportados por el nodo, actualmente solo NODE_NETWORK

nTime
: La fecha y hora actuales

addrYou
: La dirección IP del nodo remoto como se ve desde este nodo

addrMe
: La dirección IP del nodo local, tal como se descubrió por el nodo local

subver
: Una sub-version que muestra el tipo de software que se está ejecutando en este nodo (por ejemplo, `/Satoshi:0.9.2.1/`)

BestHeight
: La altura de bloque de la cadena de bloques de este nodo

(Ver GitHub para un ejemplo del mensaje de red version).

El mensaje version es siempre el primer mensaje enviado por cualquier nodo P2P a otro nodo P2P. El nodo local que

recibe un mensaje version examinará el nVersion reportado por el par remoto y decidirá si el par remoto es compatible. Si el interlocutor remoto es compatible, el interlocutor local reconocerá el mensaje version y establecerá una conexión enviando un verack.

¿Cómo funciona un nuevo nodo para encontrar pares? El primer método consiste en hacer una consulta DNS utilizando una serie de "semillas de DNS", que son servidores DNS que proporcionan una lista de direcciones IP de nodos bitcoin. Algunas de esas semillas DNS proporcionan una lista estática de direcciones IP de los nodos bitcoin estables que están a la escucha. Algunas de las semillas de DNS son implementaciones personalizadas de BIND (Berkeley Internet Name Daemon) que devuelven un subconjunto aleatorio de una lista de direcciones de nodos bitcoin recogidos por un rastreador o por un nodo bitcoin de larga duración. El Cliente Principal, el Bitcoin Core, contiene los nombres de cinco semillas DNS diferentes. La diversidad de la propiedad y la diversidad de la implementación de las diferentes semillas DNS ofrece un alto nivel de fiabilidad en el proceso de arranque inicial. En el Cliente Principal de Bitcoin, la preferencia de utilizar las semillas de DNS se controla con la opción -dnsseed (ajustado a 1 por defecto, para usar la semilla DNS).

Alternativamente, un nodo nuevo en el proceso de arranque que no sabe nada de la red debe tener la dirección IP de al menos un nodo bitcoin, después de lo cual se pueden establecer conexiones a través de nuevas presentaciones. El argumento de línea de comandos -seednode se puede utilizar para conectarse a un nodo solo para presentaciones, usándolo como una semilla. Después de utilizar el nodo de semilla inicial para hacer las presentaciones, el cliente se desconecta de ella y utiliza los pares recién descubiertos.

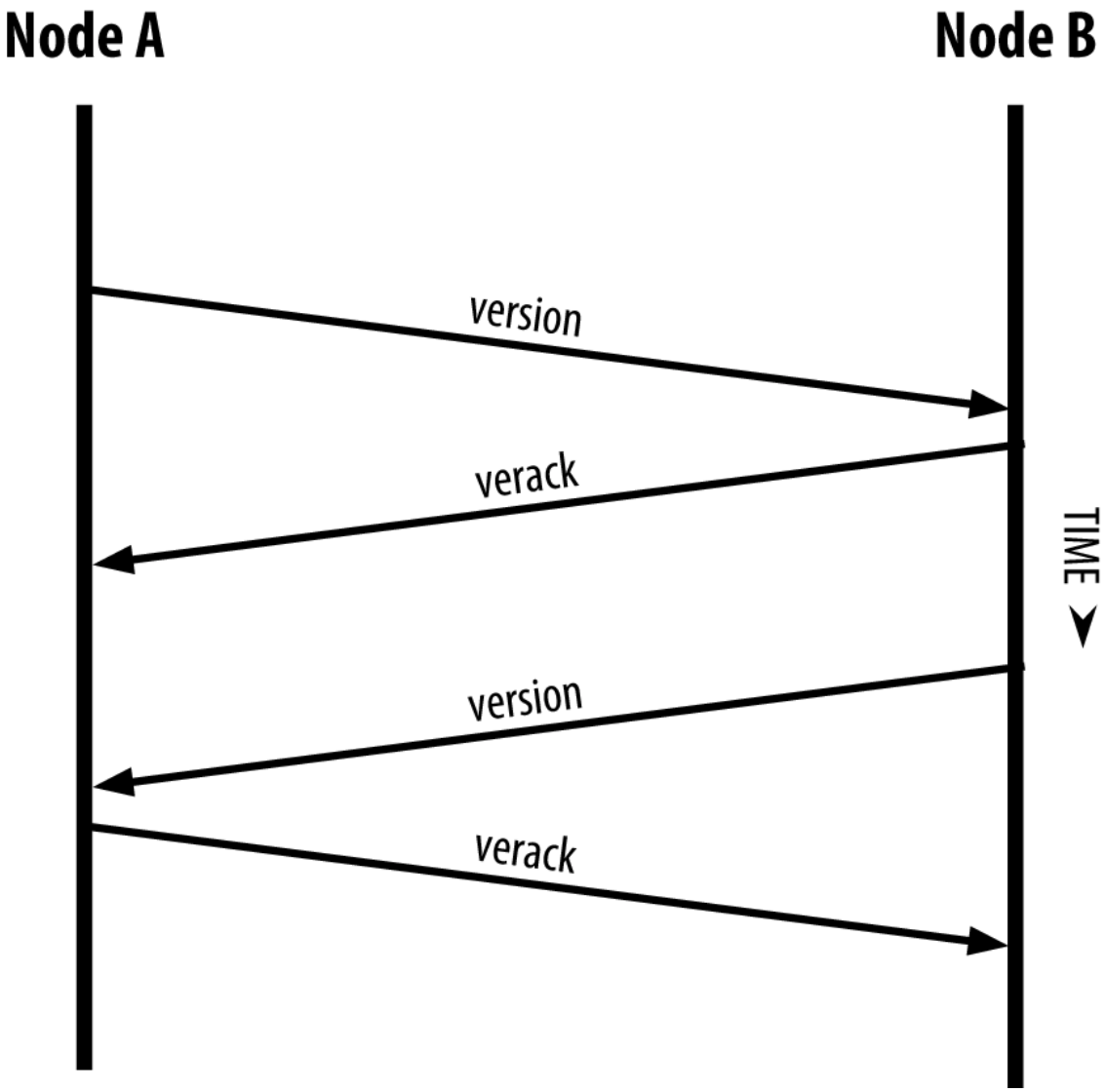


Figure 50. El apretón de manos inicial entre pares

Una vez que se han establecido una o más conexiones, el nuevo nodo enviará un mensaje addr que contiene su propia dirección IP para sus vecinos. Los vecinos, a su vez, remitirán el mensaje addr a sus vecinos, lo que garantiza que el nodo recién conectado se convierte en bien conocido y mejor conectado. Además, el nodo recién conectado puede enviar getaddr a los vecinos, pidiéndoles que le devuelvan una lista de direcciones IP de otros compañeros. De esa manera, un nodo puede encontrar compañeros para conectarse y anunciar su existencia en la red para que otros nodos puedan

encontrarlo. Descubrimiento y propagación de la dirección muestra el protocolo de descubrimiento de direcciones.

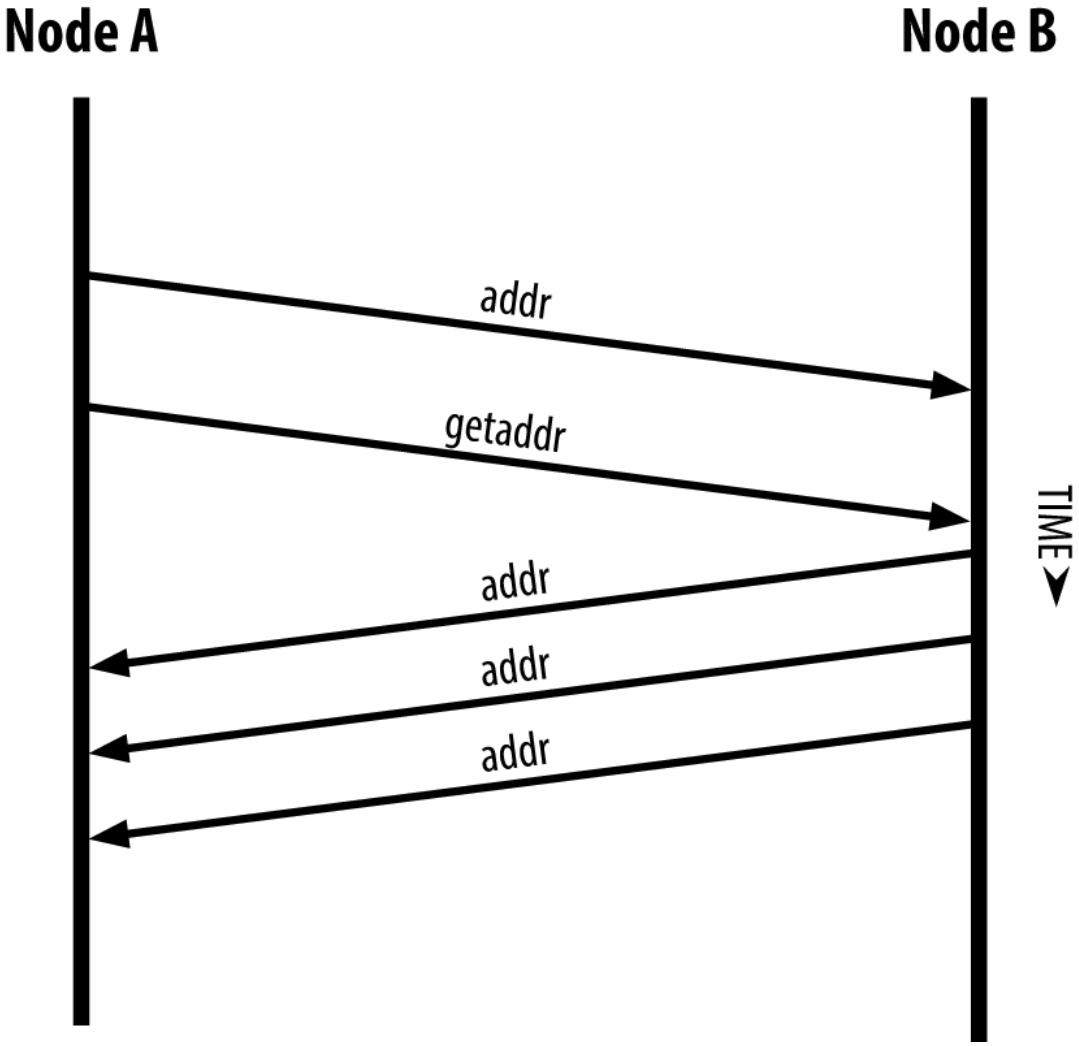


Figure 51. Descubrimiento y propagación de la dirección

Un nodo debe conectarse con algún puñado de pares diferentes para establecer diversas rutas en la red bitcoin. Las rutas no son persistentes –los nodos van y vienen–, por lo que el nodo debe continuar descubriendo nuevos nodos, conforme pierde conexiones viejas, así como también ayudará a otros nodos cuando éstos arrancan. Solo se necesita una conexión para arrancar, puesto que el primer nodo puede ofrecer introducciones a sus nodos pares y esos pares pueden ofrecer a su vez, introducciones adicionales. También es innecesario y es un derroche de recursos de red, el conectarse a más de un puñado de nodos. Después del arranque, un nodo recordará sus conexiones de pares exitosas más recientes, de modo que si se reinicia puede restablecer rápidamente las conexiones con su red de pares anterior. Si ninguno de los pares anteriores responde a su solicitud de conexión, el nodo puede usar los nodos semilla para iniciar nuevamente.

En un nodo que ejecuta el Cliente Principal Bitcoin Core, puedes listar las conexiones del nodo con el comando getpeerinfo:

```
$ bitcoin-cli getpeerinfo


[
    {
        "addr" : "85.213.199.39:8333",
        "services" : "00000001",
        "lastsend" : 1405634126,
        "lastrecv" : 1405634127,
        "bytessent" : 23487651,
        "bytesrecv" : 138679099,
        "conntime" : 1405021768,
        "pingtime" : 0.00000000,
        "version" : 70002,
        "subver" : "/Satoshi:0.9.2.1/",
        "inbound" : false,
        "startingheight" : 310131,
        "banscore" : 0,
```

```
        "syncnode" : true
    },
    {
        "addr" : "58.23.244.20:8333",
        "services" : "00000001",
        "lastsend" : 1405634127,
        "lastrecv" : 1405634124,
        "bytessent" : 4460918,
        "bytesrecv" : 8903575,
        "conntime" : 1405559628,
        "pingtime" : 0.00000000,
        "version" : 70001,
        "subver" : "/Satoshi:0.8.6/",
        "inbound" : false,
        "startingheight" : 311074,
        "banscore" : 0,
        "syncnode" : false
    }
]
```

Para anular la administración automática de pares y para especificar una lista de direcciones IP, los usuarios pueden proporcionar la opción -connect=<IPAddress> y especificar una o más direcciones IP. Si se usa esta opción, el nodo solo se conectará a las direcciones IP seleccionadas, en lugar de descubrir y mantener las conexiones de pares automáticamente.

Si no hay tráfico en una conexión, los nodos enviarán periódicamente un mensaje para mantener la conexión. Si un nodo no se ha comunicado en una conexión durante más de 90 minutos, se supone que está desconectado y se buscará un nuevo par. Por lo tanto, la red se ajusta dinámicamente a los nodos transitorios y a los problemas de la red, y puede crecer y decrecer orgánicamente según sea necesario sin ningún control central.

Nodos Completos

Los nodos completos son nodos que mantienen una cadena de bloques completa con todas las transacciones. Más exactamente, probablemente deberían llamarse "nodos completos de la cadena de bloques". En los primeros años de bitcoin, todos los nodos eran nodos completos y actualmente el Cliente Principal, Bitcoin Core es un nodo completo de la cadena de bloques. En los últimos dos años, sin embargo, se han introducido nuevas formas de clientes de bitcoin que no mantienen una cadena de bloques completa, sino que se ejecutan como clientes ligeros. Los examinaremos con más detalle en la siguiente sección.

Los nodos completos de la cadena de bloques mantienen una copia completa y actualizada de la cadena de bloques de bitcoin con todas las transacciones, que construyen y verifican de forma independiente, comenzando con el primer bloque (bloque génesis) y construyendo hasta el último bloque conocido en la red. Un nodos completos de la cadena de bloques puede verificar de forma independiente y autorizada cualquier transacción sin recurrir ni confiar en ningún otro nodo o fuente de información. El nodos completos de la cadena de bloques depende de la red para recibir actualizaciones sobre nuevos bloques de transacciones, que después verifica e incorpora en su copia local de la cadena de bloques.

La ejecución de un nodo completo de cadena de bloques te brinda la experiencia pura de bitcoin: verificación independiente de todas las transacciones sin la necesidad de depender o confiar en ningún otro sistema. Es fácil saber si estás ejecutando un nodo completo porque requiere más de cien gigabytes de almacenamiento persistente (espacio en disco) para almacenar la cadena de bloques completa. Si necesitas mucho disco y se tarda de dos a tres días en sincronizarse con la red, estás ejecutando un nodo completo. Ese es el precio de la libertad y la independencia completa de autoridades centrales.

Hay algunas implementaciones alternativas en los clientes bitcoin completos de la cadena de bloques, construidas utilizando diferentes lenguajes de programación y arquitecturas de software. Sin embargo, la aplicación más común es el cliente de referencia Bitcoin Core, también conocido como el cliente Satoshi. Más del 75% de los nodos en la red bitcoin ejecutan varias versiones de Bitcoin Core. Se identifica como "Satoshi" en la cadena de sub-versión enviada en el mensaje version y se muestra mediante el comando getpeerinfo como vimos anteriormente; por ejemplo, /Satoshi:0.8.6/.

Intercambiando "Inventario"

La primera cosa que un nodo completo hará una vez que se conecta a los compañeros es tratar de construir una cadena de bloques completa. Si es un nodo nuevo y no tiene cadena de bloques en absoluto, entonces solo conoce un bloque, el bloque génesis, que está integrado de forma estática en el software del cliente. Comenzando con el bloque #0 (el bloque génesis), el nuevo nodo tendrá que descargar cientos de miles de bloques para sincronizarse con la red y volver a establecer la cadena de bloques completa.

El proceso de sincronización de la cadena de bloques comienza con el mensaje version, porque contiene la BestHeight, que es la altura actual de la cadena de bloques de un nodo (número de bloques). Un nodo verá los mensajes version de sus compañeros para saber cuántos bloques tiene cada uno, y ser capaz de comparar con el número de bloques que tiene en su propia cadena de bloques. Los nodos intercambiarán el mensaje getblocks que contiene el hash (huella digital) del bloque de la parte superior de su cadena de bloques local. Uno de los compañeros será capaz de identificar el hash recibido como perteneciente a un bloque que no está en la cima, sino que pertenece a un bloque más antiguo, deduciendo de esta manera que su propia cadena de bloques local es más larga que la de su compañero.

El nodo que tiene la cadena de bloques más larga, tiene mayor cantidad de bloques y puede identificar qué bloques necesita el otro nodo para "ponerse al día". Identificará los primeros 500 bloques a compartir y transmitirá sus valores hash utilizando un mensaje inv + (de inventario). El nodo al que le falten estos bloques podrá luego recuperarlos mediante la emisión de una serie de mensajes +getdata, solicitando los datos del bloque completo e identificando los bloques solicitados mediante los hashes del mensaje inv.

Supongamos, por ejemplo, que un nodo solo tiene el bloque génesis. A continuación, recibirá un mensaje inv de sus pares que contiene los hashes de los próximos 500 bloques en la cadena. Comenzará solicitando bloques de todos sus pares conectados, repartiendo la carga y asegurando que no abrume con sus peticiones a ningún par. El nodo mantiene un registro de cuántos bloques están "en tránsito" por cada conexión de pares, es decir, aquellos bloques que ha solicitado pero que aún no ha recibido, comprobando que no exceden un límite (MAX_BLOCKS_IN_TRANSIT_PER_PEER). De esta manera, si necesita una gran cantidad de bloques, solo solicitará otros nuevos a medida que se completan las solicitudes anteriores, permitiendo a los compañeros controlar el ritmo de las actualizaciones y no sobrecargar la red. A medida que se recibe cada bloque, se va agregando a la cadena de bloques, tal como veremos en La Cadena de Bloques. A medida que la cadena de bloques local se va construyendo gradualmente, se solicitan y se reciben más bloques, y el proceso continúa hasta que el nodo se pone al día con el resto de la red.

Este proceso de comparar la cadena de bloques local con los compañeros y la recuperación de todos los bloques que faltan sucede cada vez que un nodo se desconecta por cualquier período de tiempo. Ya sea un nodo que ha estado desconectado durante unos minutos y faltan pocos bloques, o un mes y faltan unos pocos miles de bloques, se inicia mediante el envío de getblocks, recibe un inv de respuesta, y comienza la descarga de los bloques que faltan. Nodo sincronizando la cadena de bloques pidiendo bloques a un par muestra el protocolo de inventario y la propagación de bloque.

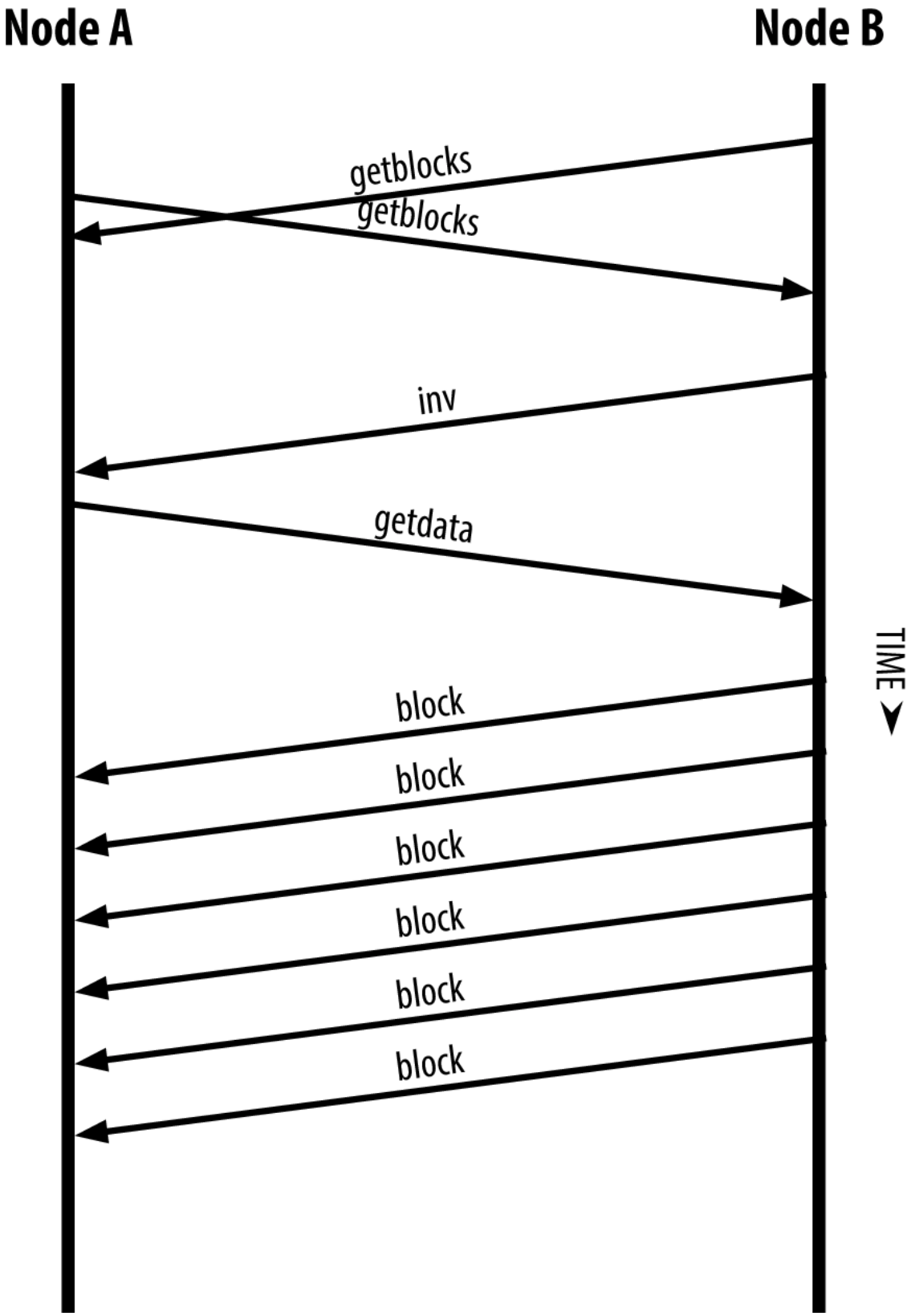


Figure 52. Nodo sincronizando la cadena de bloques pidiendo bloques a un par

Nodos de Verificación de Pago Simplificada (SPV)

No todos los nodos tienen la capacidad de almacenar la cadena de bloques completa. Muchos clientes bitcoin están diseñados para funcionar en dispositivos con restricciones de espacio y de energía, tales como teléfonos inteligentes, tabletas o sistemas embebidos. Para tales dispositivos, se utiliza un método de *verificacion de pago simplificada* (SPV) que permite operar sin almacenar la cadena de bloques completa. Este tipo de clientes se llaman clientes SPV o clientes ligeros. Con el aumento en la adopción de bitcoin, el nodo SPV se está convirtiendo en la forma más común de nodo bitcoin, especialmente para carteras bitcoin.

Los nodos SPV descargan solo las cabeceras de bloque y no descargan las transacciones incluidas en cada bloque. La

cadena resultante de bloques, sin transacciones, es 1000 veces menor que la cadena de bloques completa. Los nodos SPV no pueden construir una imagen completa de todos los UTXOs que están disponibles para el gasto, ya que no saben acerca de todas las transacciones en la red. Los nodos SPV verifican las transacciones utilizando un método ligeramente diferente que depende de los pares para proporcionar vistas parciales de las partes relevantes de la cadena de bloques bajo demanda.

Como analogía, un nodo completo es como un turista en una ciudad extraña, equipado con un mapa detallado de cada calle y de cada dirección. En comparación, un nodo SPV es como un turista en una ciudad extraña preguntando a extraños al azar indicaciones giro a giro conociendo solo una avenida principal. Aunque ambos turistas pueden verificar la existencia de una calle al visitarla, el turista sin un mapa no sabe lo que hay más allá de las calles laterales y no sabe qué otras calles existen. Situado frente a 23 Church Street, el turista sin un mapa no puede saber si hay una docena de otras direcciones "23 Church Street" en la ciudad y si esta es la correcta. La mejor opción del turista sin mapas es preguntar a bastante gente y esperar que algunos de ellos no le estén tratando de robar.

SPV verifica las transacciones en función de su *profundidad* en la cadena de bloques, en vez de en su *altura*. Mientras que un nodo completo de la cadena de bloques construirá una cadena completamente verificada de miles de bloques y transacciones que alcanza (atrás en el tiempo) toda la cadena de bloques hasta el bloque génesis, un nodo SPV verificará la cadena de todos los bloques (pero no todas las transacciones) y vinculará esa cadena a la transacción de interés.

Por ejemplo, al examinar una transacción en el bloque 300.000, un nodo completo enlaza todos los 300.000 bloques desde el bloque génesis y crea una base de datos completa de UTXO, estableciendo la validez de la transacción mediante la comprobación de que el UTXO se encuentra sin gastar. Un nodo SPV no puede validar si el UTXO está sin gastar. En su lugar, el nodo SPV establecerá un vínculo entre la transacción y el bloque que lo contiene, usando un *camino merkle* (ver Árboles Merkle). A continuación, el nodo SPV espera hasta que ve los seis bloques de 300.001 a 300.006, apilados encima del bloque que contiene la transacción y lo verifica mediante el establecimiento de su profundidad bajo los bloques 300.006 a 300.001. El hecho de que otros nodos de la red acepten el bloque 300.000 y luego hayan hecho el trabajo necesario para producir seis bloques más en la parte superior del mismo es la prueba, de forma indirecta, de que la operación no fue un doble gasto.

No se puede convencer a un nodo SPV de que existe una transacción en un bloque cuando la transacción en realidad no existe. El nodo SPV establece la existencia de una transacción en un bloque solicitando una prueba de ruta de merkle y validando la Prueba-de-Trabajo en la cadena de bloques. Sin embargo, la existencia de una transacción puede estar "oculta" para un nodo SPV. Un nodo SPV puede definitivamente probar que existe una transacción, pero no puede verificar que una transacción, como un doble gasto de la misma UTXO, no exista, ya que no tiene un registro de todas las transacciones. Esta vulnerabilidad se puede utilizar en un ataque de denegación de servicio o en un ataque de doble gasto contra nodos SPV. Para defenderse de esto, un nodo SPV necesita conectarse al azar a varios nodos, y así aumentar la probabilidad de que esté en contacto con al menos un nodo honesto. Esta necesidad de conectarse de forma aleatoria tiene como consecuencia que los nodos SPV también sean vulnerables a los ataques de particionamiento de la red o a ataques Sybil, donde están conectados a nodos falsos o a redes falsas y no tienen acceso a nodos honestos o a la red bitcoin real.

A efectos prácticos, los nodos SPV bien conectados son suficientemente seguros, manteniendo un equilibrio entre las necesidades de recursos, practicidad y seguridad. Sin embargo, para una seguridad infalible lo mejor es ejecutar un nodo completo de cadena de bloques.

TIP

Un nodo completo de cadena de bloques verifica una transacción mediante la comprobación de toda la cadena de miles de bloques por debajo de ella con el fin de garantizar que el UTXO no esté gastado, mientras que un nodo SPV comprueba la profundidad del bloque, que estará cubierto solo por un puñado de bloques por encima de ella.

Para obtener las cabeceras de bloque, los nodos SPV utilizan un mensaje getheaders en lugar de getblocks. El compañero que responda, enviará hasta 2.000 cabeceras de bloques utilizando un único mensaje headers. El proceso es similar al utilizado por un nodo completo para recuperar bloques completos. Los nodos SPV también establecen un filtro en la conexión con sus compañeros, filtrando el flujo de bloques y futuras transacciones enviados por los compañeros. Cualquier transacción de interés se recupera mediante una petición getdata. El compañero genera como respuesta un mensaje tx que contiene las transacciones. Nodo SPV sincronizando las cabeceras de bloque muestra la sincronización de las cabeceras de bloque.

El hecho de que los nodos SPV necesiten recuperar transacciones específicas para verificarlas de forma selectiva, hace que

se genere un riesgo para la privacidad. A diferencia de los nodos completos de cadena de bloques, que recogen todas las transacciones dentro de cada bloque, las peticiones de datos específicos por parte de los nodos SPV pueden revelar inadvertidamente las direcciones de su cartera. Por ejemplo, un tercero podría monitorear la red y llevar un registro de todas las transacciones solicitadas por una cartera en un nodo SPV, utilizando las asociaciones de direcciones bitcoin con el usuario de esa cartera y destruyendo la privacidad del usuario.

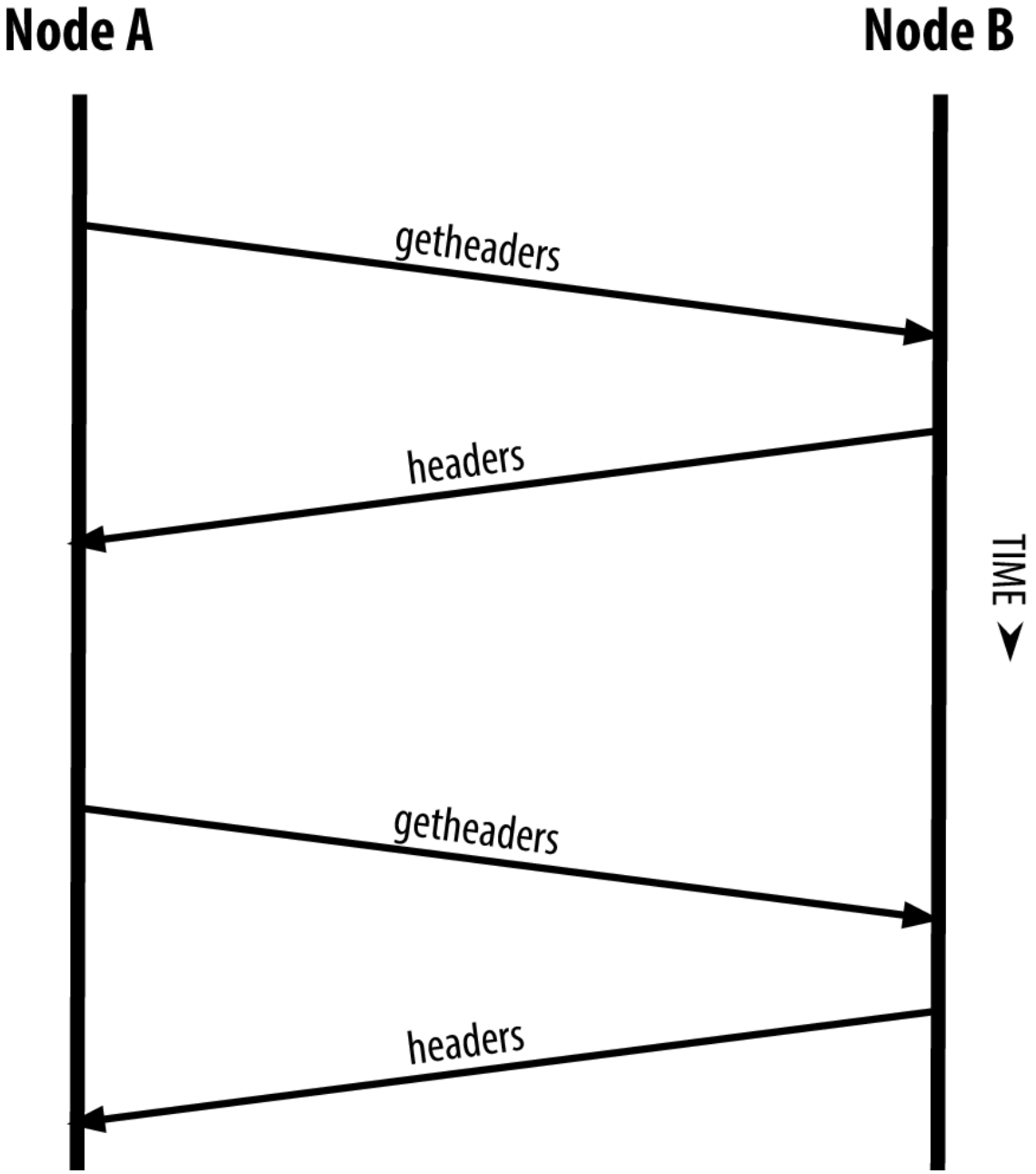


Figure 53. Nodo SPV sincronizando las cabeceras de bloque

Poco después de la introducción de los SPV/nodos ligeros, los desarrolladores de bitcoin añadieron una característica llamada *filtros de bloom* para abordar los riesgos de privacidad de los nodos SPV. Los filtros de bloom permiten a los nodos SPV recibir un subconjunto de transacciones sin revelar con precisión en qué direcciones están interesados, a través de un mecanismo de filtrado que utiliza probabilidades en lugar de patrones fijos.

Filtros de Bloom

Un filtro de bloom es un filtro de búsqueda probabilística, una manera de describir un patrón deseado sin especificarlo exactamente. Los filtros de bloom ofrecen una forma eficiente de expresar un patrón de búsqueda al mismo tiempo que se protege la privacidad. Se utilizan por los nodos SPV para pedir a sus compañeros las transacciones que coincidan con un patrón específico, sin revelar exactamente qué direcciones, laves o transacciones están buscando.

En nuestra analogía anterior, un turista sin mapas está pidiendo direcciones a una dirección específica, "23 Church St." Si preguntara a extraños por las direcciones a esta calle, estaría inadvertidamente revelando su destino. Un filtro de bloom es como preguntar: "¿Hay calles en este barrio cuyos nombres terminen en R-C-H?" Una pregunta como esa revela un

poco menos sobre el destino deseado que pedir directamente "23 Church St." Usando esta técnica, un turista puede especificar la dirección deseada con mayor detalle, como "que termine en U-R-C-H", o con menor detalle, como "que termine en H." Mediante la variación de la precisión de la búsqueda, el turista revela más o menos información, a expensas de obtener resultados más o menos específicos. Si el turista preguntara con un patrón menos específico, obtendría muchas más direcciones posibles y una mejor privacidad, pero muchos de los resultados serían irrelevantes. Si preguntara con un patrón muy específico, obtendría menos resultados, pero perdería privacidad.

Los filtros de bloom consiguen cumplir esta función al permitir que un nodo SPV especifique un patrón de búsqueda para las transacciones que puede ajustarse hacia precisión o hacia privacidad. Un filtro de bloom más específico producirá resultados precisos, pero a expensas de revelar en qué patrones está interesado el nodo SPV, y revelando así las direcciones que utiliza la cartera del usuario. Un filtro de bloom menos específico producirá más datos sobre más transacciones, muchos irrelevantes para el nodo, pero permitirá que el nodo pueda mantener mejor la privacidad.

Cómo Funcionan los Filtros de Bloom

Los filtros de bloom se implementan como un vector (en inglés, "array") de tamaño variable de N dígitos binarios (un campo de un bit) y un número variable M de funciones hash. Las funciones hash están diseñadas para producir siempre una salida que está comprendida entre 1 y N, que corresponde al vector de dígitos binarios. Las funciones hash se generan de manera determinista, de modo que cualquier nodo que ejecute un filtro de bloom siempre utilizará las mismas funciones hash y obtendrá los mismos resultados para una entrada específica. El filtro de bloom puede ajustarse eligiendo diferentes longitudes (N) y un número diferente (M) de funciones de hash, variando así el nivel de precisión y por lo tanto la privacidad.

En Un ejemplo de filtro de bloom simple, con un campo de 16 bits y tres funciones hash, usamos un pequeño vector de 16 bits y un conjunto de tres funciones hash para demostrar cómo funcionan los filtros de bloom.

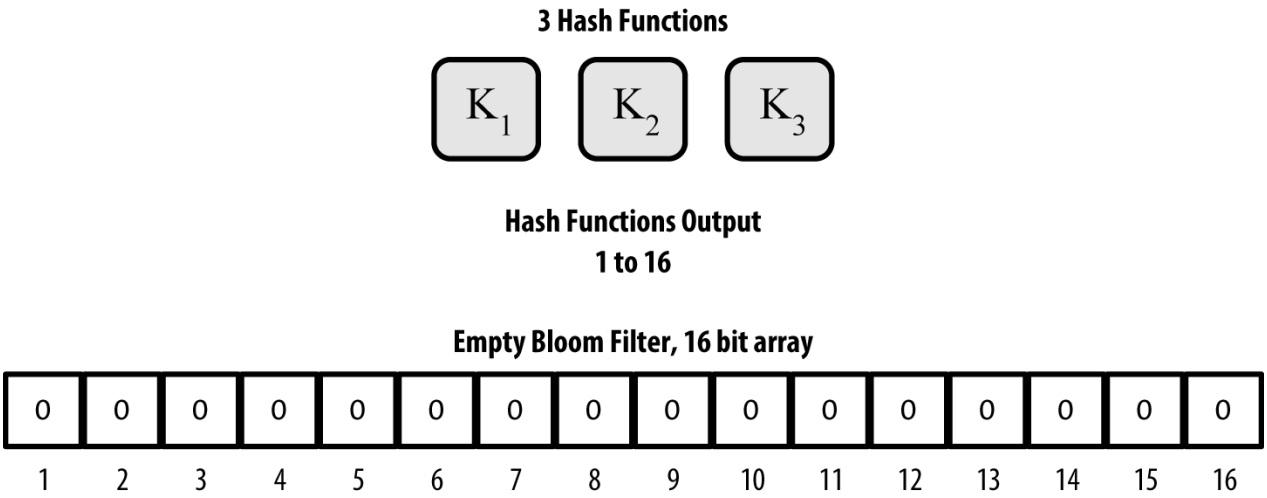


Figure 54. Un ejemplo de filtro de bloom simple, con un campo de 16 bits y tres funciones hash

El filtro de bloom se inicializa para que el vector de bits sea todo ceros. Para agregar un patrón al filtro de bloom, se hace hash del patrón, una vez para cada función hash. La aplicación de la primera función de hash a la entrada da como resultado un número entre 1 y N. Se localiza el bit correspondiente en el vector (indexado de 1 a N) y se pone a 1, quedando registrada así la salida de la función hash. Entonces, se ejecuta la siguiente función hash para establecer otro bit, y así sucesivamente. Una vez de que se han aplicado todas las funciones de hash M, el patrón de búsqueda queda "registrado" en el filtro de bloom como M bits que han cambiado de 0 a 1.

Añadiendo un patrón "A" a nuestro filtro de bloom simple es un ejemplo de la adición de un patrón "A" para el filtro de bloom sencillo mostrado en Un ejemplo de filtro de bloom simple, con un campo de 16 bits y tres funciones hash.

Añadir un segundo patrón es tan simple como repetir este proceso. Se hace hash del patrón mediante la ejecución de cada una de las funciones hash, y el resultado se registra mediante el establecimiento de los bits a 1. Ten en cuenta que a medida que un filtro de bloom se llena con más patrones, algún resultado de la función de hash podría coincidir con uno que ya está marcado a 1, en cuyo caso no se cambia el bit. En esencia, la aparición de más patrones que se registren como bits superpuestos es la señal de que el filtro de bloom comienza a saturarse con más bits establecidos en 1, haciendo que la precisión del filtro disminuya. Por ello, el filtro es una estructura de datos probabilística que se vuelve menos precisa a medida que se agregan más patrones. La precisión depende del número de los patrones agregados en relación con el tamaño del vector de bits (N) y el número de funciones hash (M). Un vector de bits más grande y con más funciones hash

puede registrar más patrones con mayor precisión. Un vector de bits más pequeño o con menos funciones hash registrará menos patrones y producirá menos precisión.

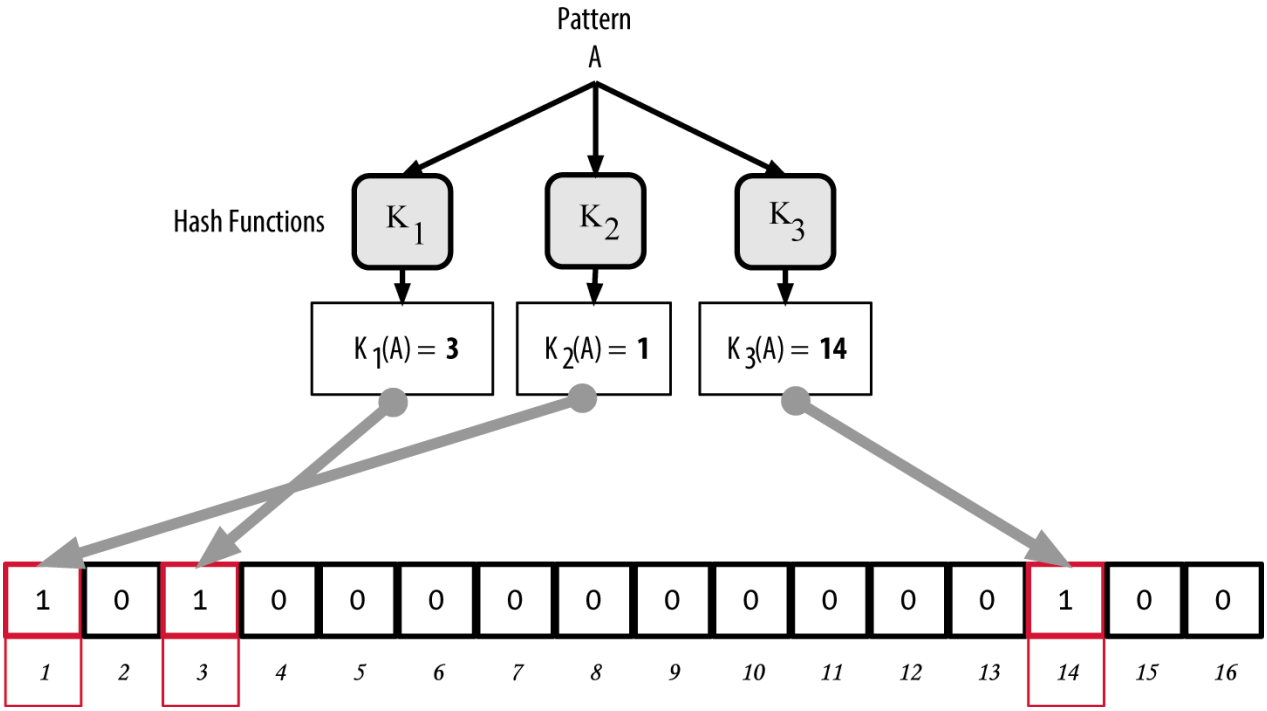


Figure 55. Añadiendo un patrón "A" a nuestro filtro de bloom simple

Añadiendo un segundo patrón "B" a nuestro filtro de bloom simple es un ejemplo que añade un segundo patrón "B" al filtro de bloom simple.

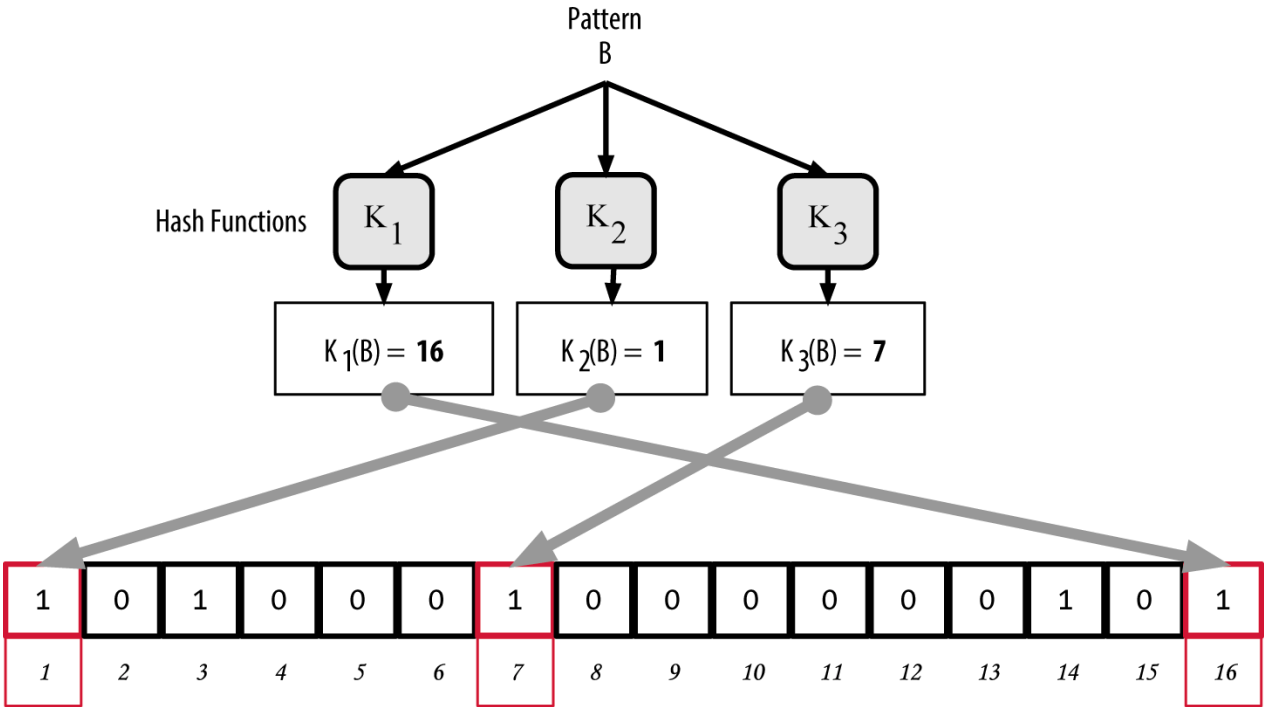


Figure 56. Añadiendo un segundo patrón "B" a nuestro filtro de bloom simple

Para probar si un patrón es parte de un filtro de bloom, se hace hash del patrón con cada una de las funciones hash, y el patrón de bits resultante se chequea contra el vector de bits. Si todos los bits indexados por las funciones hash se establecen en 1, entonces el patrón está *probablemente* registrado en el filtro de bloom. Debido a que los bits se pueden establecer debido a la superposición originada por múltiples patrones, la respuesta no es irrefutable, sino que es probabilística. En términos simples, un resultado positivo de filtro de bloom es un "Tal vez, Sí."

Probando la existencia del patrón "X" en el filtro de bloom. El resultado es una coincidencia positiva probabilística, es decir, "Tal vez." es un ejemplo de pruebas de la existencia del patrón "X" en el filtro de bloom simple. Los bits correspondientes se establecen en 1, por lo que el patrón es probablemente una coincidencia.

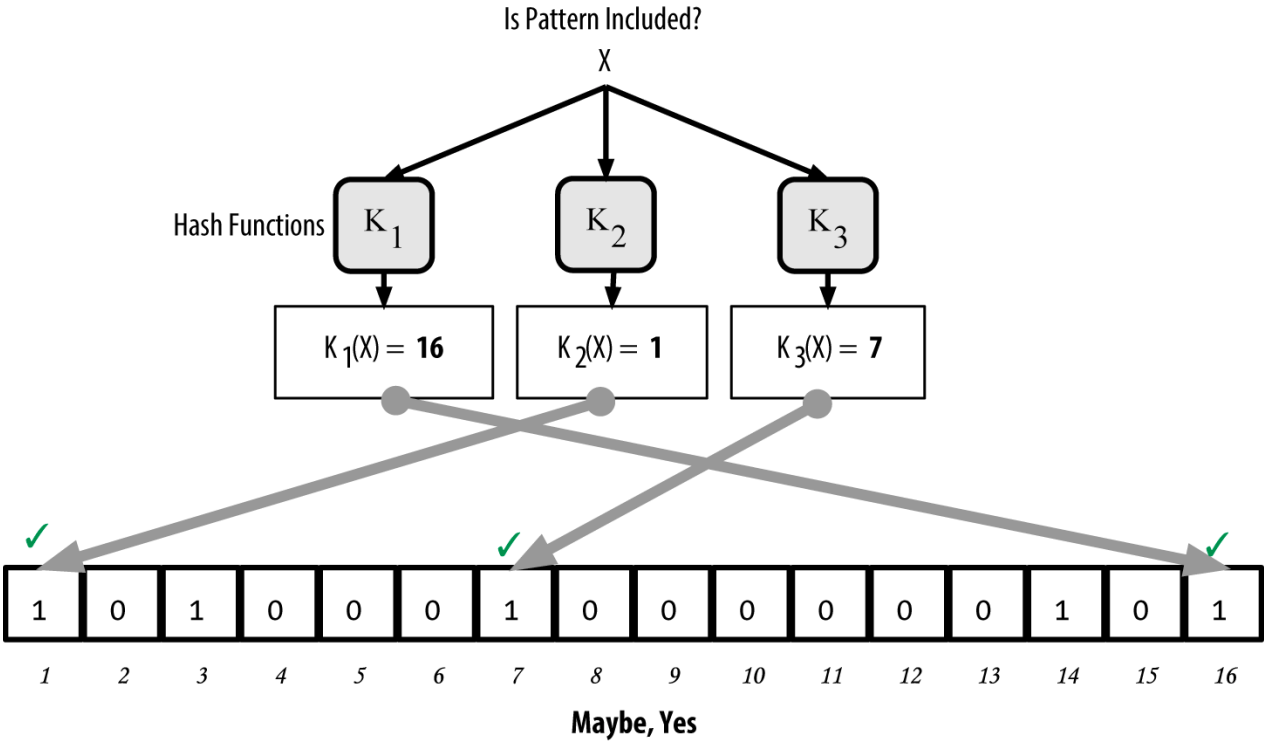


Figure 57. Probando la existencia del patrón "X" en el filtro de bloom. El resultado es una coincidencia positiva probabilística, es decir, "Tal vez."

Por el contrario, si un patrón se prueba contra el filtro de bloom y uno cualquiera de los bits se establece en 0, queda demostrado que el patrón no se registró en el filtro de bloom. Un resultado negativo no es una probabilidad, es una certeza. En términos simples, un resultado negativo en un filtro de bloom es un "¡Definitivamente No!"

Testeando la existencia del patrón "Y" en el filtro de bloom. El resultado es una coincidencia negativa definitiva, que significa "¡Definitivamente No!" es un ejemplo para probar la existencia del patrón "Y" en el filtro de bloom simple. Uno de los bits correspondientes se establece en 0, por lo que el patrón es definitivamente una no coincidencia.

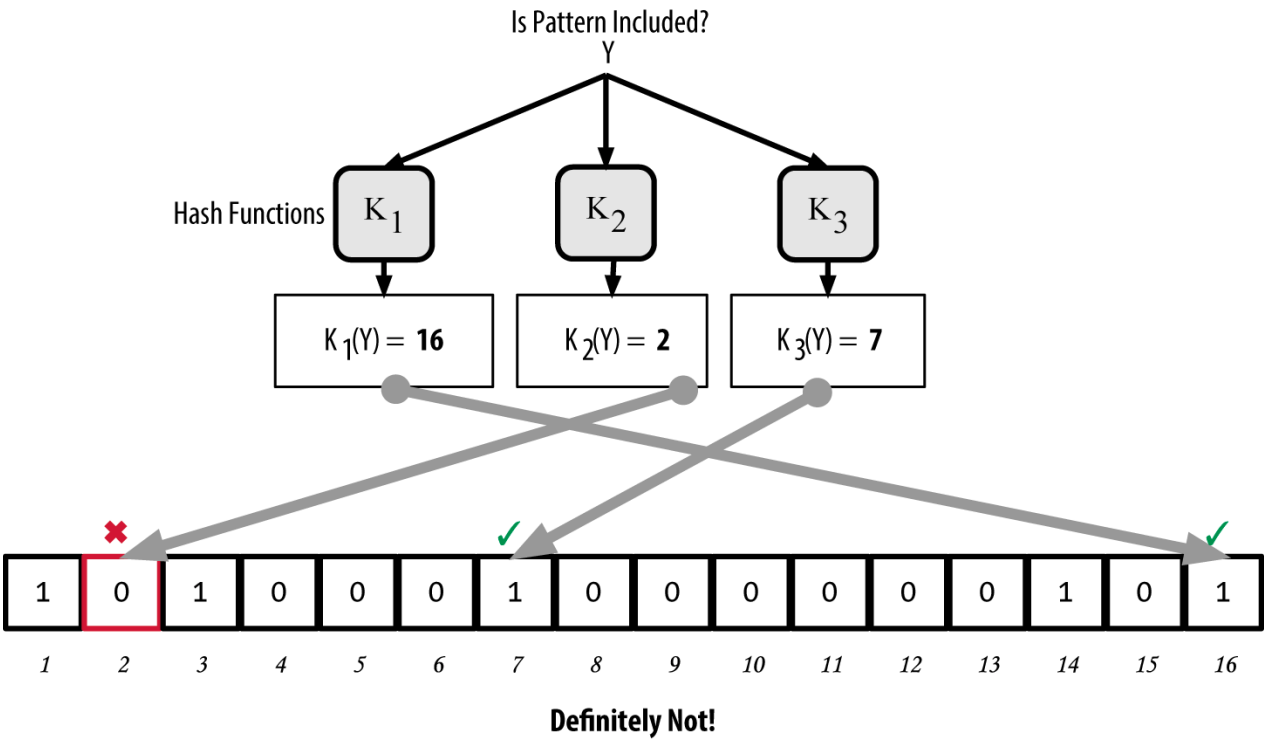


Figure 58. Testeando la existencia del patrón "Y" en el filtro de bloom. El resultado es una coincidencia negativa definitiva, que significa "¡Definitivamente No!"

Cómo los Nodos SPV Usan Filtros de Bloom

Los filtros de bloom se utilizan para filtrar las transacciones (y los bloques que las contienen) que un nodo SPV recibe de sus pares, seleccionando solo las transacciones de interés para el nodo SPV sin revelar en qué direcciones o llaves están

interesados.

Un nodo SPV inicializará un filtro de bloom como "vacío"; en ese estado, el filtro de bloom no coincidirá con ningún patrón. El nodo SPV después hará una lista de todas las direcciones, llaves y hashes que le interesen. Lo hará extrayendo el hash de llave pública, el hash de script y los ID de transacción de cualquier UTXO controlado por su cartera. El nodo SPV después agrega cada uno de estos al filtro de bloom, de modo que el filtro de bloom "coincidirá" si estos patrones están presentes en una transacción, sin revelar los patrones en sí mismos.

El nodo SPV enviará un mensaje filterload al compañero, que contiene el filtro de bloom para usar en la conexión. En el compañero, los filtros de bloom se comparan con cada transacción entrante. El nodo completo verifica varias partes de la transacción contra el filtro de bloom, buscando una coincidencia que incluya:

- El ID de transacción
- Los componentes de datos de los scripts de bloqueo de cada una de las salidas de transacción (cada llave y hash en el script)
- Cada una de las entradas de transacción
- Cada uno de los componentes de datos de las firmas de las entradas (o scripts de testigo)

Al verificar todos estos componentes, se pueden usar filtros de bloom para hacer coincidir los hashes de llave pública, los scripts, los valores OP_RETURN, las llaves públicas en firmas o cualquier componente futuro de un contrato inteligente o un script complejo.

Después de establecer un filtro, el compañero probará las salidas de cada transacción contra el filtro de bloom. Sólo las transacciones que coinciden con el filtro se envían al nodo.

En respuesta a un mensaje getdata desde el nodo, los compañeros enviarán un mensaje merkleblock que contiene solo las cabeceras de bloques para los bloques que coinciden con el filtro y una ruta de merkle (ver Árboles Merkle) para cada transacción correspondiente. El compañero entonces también enviará mensajes tx que contienen las transacciones coincidentes por el filtro.

A medida que el nodo completo envía transacciones al nodo SPV, el nodo SPV descarta los falsos positivos y utiliza las transacciones que coinciden correctamente para actualizar su conjunto UTXO y el saldo de la cartera. A medida que actualiza su propia vista del conjunto UTXO, también modifica el filtro de bloom para que coincida con cualquier transacción futura que haga referencia al UTXO que acaba de encontrar. El nodo completo después usa el nuevo filtro de bloom para coincidir con las nuevas transacciones y todo el proceso se repite.

El nodo que configura el filtro de bloom puede agregar patrones al filtro de forma interactiva enviando un mensaje filteradd. Para borrar el filtro de bloom, el nodo puede enviar un mensaje filterclear. Debido a que no es posible eliminar un patrón de un filtro de bloom, un nodo tiene que borrar y reenviar un nuevo filtro de bloom si ya no se desea un patrón.

El protocolo de red y el mecanismo de filtro de bloom para nodos SPV se define en BIP-37 (Peer Services).

Nodos SPV y Privacidad

Los nodos que implementan SPV tienen una privacidad más débil que un nodo completo. Un nodo completo recibe todas las transacciones y, por lo tanto, no revela información sobre si está usando alguna dirección en su cartera. Un nodo SPV recibe una lista filtrada de transacciones relacionadas con las direcciones que están en su cartera. Como resultado, se reduce la privacidad del propietario.

Los filtros de bloom son una forma de reducir la pérdida de privacidad. Sin ellos, un nodo SPV tendría que enumerar explícitamente las direcciones en las que estaba interesado, creando una violación grave de la privacidad. Sin embargo, incluso con los filtros de bloom, un adversario que supervisa el tráfico de un cliente SPV o se conecta a él directamente como un nodo en la red P2P puede recopilar suficiente información a lo largo del tiempo para conocer las direcciones en la cartera del cliente SPV.

Conexiones Encriptadas y Autenticadas

La mayoría de los usuarios nuevos de bitcoin asumen que las comunicaciones de red de un nodo bitcoin están cifradas. La realidad es que la implementación original de bitcoin se comunica completamente de forma clara. Si bien esto no es un

problema de privacidad importante para los nodos completos, es un gran problema para los nodos SPV.

Existen dos soluciones que proporcionan la encriptación de las comunicaciones para aumentar la privacidad y la seguridad de la red P2P de bitcoin: *Transporte Tor* y *Autenticación y Encriptación P2P* con BIP-150/151.

Transporte Tor

Tor, que significa *The Onion Routing network* (es decir, la red de enrutamiento de cebolla), es un proyecto de software y una red que ofrece encriptación y encapsulación de datos a través de rutas de red aleatorizadas que ofrecen anonimato, intrazabilidad y privacidad.

Bitcoin Core ofrece varias opciones de configuración que te permiten ejecutar un nodo bitcoin con su tráfico transportado a través de la red Tor. Además, Bitcoin Core también puede ofrecer un servicio oculto de Tor (en inglés, Tor hidden service) permitiendo que otros nodos Tor se conecten a tu nodo directamente sobre Tor.

A partir de la versión 0.12 de Bitcoin Core, un nodo ofrecerá automáticamente un servicio oculto de Tor si puede conectarse a un servicio de Tor local. Si tienes Tor instalado y el proceso Bitcoin Core se ejecuta como un usuario con permisos adecuados para acceder a la cookie de autenticación de Tor, debería funcionar automáticamente. Use la marca debug para activar la depuración de Bitcoin Core para el servicio de Tor de esta manera:

```
$ bitcoind --daemon --debug=tor
```

Deberías ver "tor: ADD_ONION successful" en los registros, lo que indica que Bitcoin Core ha agregado un servicio oculto a la red Tor.

Puedes encontrar más instrucciones sobre cómo ejecutar Bitcoin Core como un servicio oculto de Tor en la documentación de Bitcoin Core (*docs/tor.md*) y en varios tutoriales en línea.

Autenticación y Encriptación P2P

Dos propuestas de mejora de bitcoin, BIP-150 y BIP-151, agregan soporte para la autenticación y encriptación P2P en la red P2P de bitcoin. Estos dos BIP definen servicios opcionales que pueden ofrecer los nodos de bitcoin compatibles. BIP-151 permite la encriptación negociada para todas las comunicaciones entre dos nodos que admiten BIP-151. BIP-150 ofrece autenticación de pares opcional que permite que los nodos autentiquen la identidad de cada uno mediante ECDSA y llaves privadas. BIP-150 requiere que, antes de la autenticación, los dos nodos hayan establecido comunicaciones cifradas según BIP-151.

A partir de enero de 2017, BIP-150 y BIP-151 no se implementan en Bitcoin Core. Sin embargo, las dos propuestas han sido implementadas por al menos un cliente de bitcoin alternativo llamado bcoin.

BIP-150 y BIP-151 permiten a los usuarios ejecutar clientes SPV que se conectan a un nodo completo de confianza, utilizando autenticación y encriptación para proteger la privacidad del cliente SPV.

Además, la autenticación se puede usar para crear redes de nodos bitcoin de confianza y evitar los ataques Man-in-the-Middle. Finalmente, la encriptación P2P, si se implementa de forma amplia, fortalecería la resistencia de bitcoin al análisis del tráfico y la vigilancia que erosionan la privacidad, especialmente en países totalitarios donde el uso de internet está fuertemente controlado y monitoreado.

Es estándar está definido en BIP-150 (Peer Authentication) y BIP-151 (Peer-to-Peer Communication Encryption).

Pools de Transacciones

Casi todos los nodos en la red bitcoin mantienen una lista temporal de las transacciones no confirmadas llamada *tanque de memoria* (en inglés, memory pool), *mempool*, o *pool de transacciones* (en inglés, transaction pool). Los nodos utilizan esta reserva (en inglés, "pool") para mantener un registro de las transacciones que son conocidas por la red, pero que aún no están incluidas en la cadena de bloques. Por ejemplo, un nodo de cartera usará el pool de transacciones para rastrear los pagos entrantes a la cartera del usuario que se han recibido en la red pero que aún no se han confirmado.

Cuando se reciben y se verifican las transacciones, se añaden al pool de transacciones y se retransmiten a los nodos vecinos para que se propaguen en la red.

Algunas implementaciones de nodo también mantienen un pool separado de transacciones huérfanas. Si las entradas de una transacción se refieren a una transacción que aún no se conoce, tal como un padre que falta, la transacción huérfana

será almacenada temporalmente en el pool de huérfanos hasta que llegue la transacción padre.

Cuando se añade una transacción al pool de transacciones, se comprueba el pool de huérfanos para cualquier huérfano que haga referencia a las salidas de esta transacción (sus hijos). Se validan los huérfanos que coincidan. Si es válido, se retira del pool de huérfanos y se añade al pool de transacciones, completando la cadena que comenzó con la transacción padre. A la luz de la transacción que se acaba de agregar, que ya no es huérfana, el proceso se repite recursivamente en busca de descendientes, hasta que no se encuentren más descendientes. A través de este proceso, la llegada de una transacción padre desencadena una reconstrucción en cascada de toda una cadena de transacciones interdependientes por volver a unir a los huérfanos con sus padres hasta el final de la cadena.

Tanto el pool de transacciones como el pool de huérfanos (en el caso de que esté implementado) se almacenan en la memoria local y no se guardan en almacenamiento persistente; más bien, se llenan dinámicamente de los mensajes entrantes de la red. Cuando se inicia un nodo, los dos pools están vacíos y son progresivamente ocupados con nuevas transacciones recibidas en la red.

Algunas implementaciones del cliente bitcoin también mantienen una base de datos o pool UTXO, que es el set de todas las salidas sin gastar en la cadena de bloques. Aunque el nombre "pool UTXO" suena similar al pool de transacciones, representa un set diferente de datos. A diferencia de los pools de transacciones y de huérfanos, el pool UTXO no se inicializa vacío, sino que contiene millones de salidas de transacciones sin gastar, todo lo que no se ha gastado desde el bloque génesis. El pool UTXO se puede alojar en la memoria local o como una tabla de base de datos indexada en almacenamiento persistente.

El pool de transacciones y el pool de huérfanos representan la perspectiva local de un solo nodo y pueden variar significativamente de un nodo a otro, dependiendo de cuando se inicia o reinicia el nodo. Sin embargo, el pool UTXO representa el consenso emergente de la red y por lo tanto va a variar poco entre los nodos. Además, los pools de transacciones y de huérfanos solo contienen transacciones no confirmadas, mientras que el pool UTXO solo contiene salidas confirmadas .

La Cadena de Bloques

Introducción

La estructura de datos de la cadena de bloques (en inglés, "blockchain") es una lista ordenada, enlazada hacia atrás en el tiempo, de bloques de transacciones. La cadena de bloques se puede almacenar como un archivo plano, o en una base de datos simple. El Cliente Principal, Bitcoin Core almacena los metadatos de la cadena de bloques usando la base de datos LevelDB de Google. Los bloques están enlazados "hacia atrás en el tiempo", cada uno referenciando al bloque anterior de la cadena. La cadena de bloques a menudo se visualiza como una pila vertical, con los bloques en capas uno encima de otro, sirviendo el primer bloque como la base de la pila. La visualización de bloques apilados unos encima de otros resulta en el uso de términos como "altura" para referirse a la distancia desde el primer bloque, y "arriba" o "punta" para referirse al bloque añadido más recientemente.

Cada bloque dentro de la cadena de bloques se identifica mediante un hash, generado utilizando el algoritmo de hash criptográfico SHA256 en la cabecera de bloque. Cada bloque también hace referencia a un bloque anterior, conocido como el bloque *padre*, a través del campo "hash de bloque anterior" en la cabecera del bloque. En otras palabras, cada bloque contiene el hash de su padre dentro de su propio encabezado. La secuencia de valores hash que une cada bloque a su padre crea una cadena que se remonta hasta el primer bloque creado, conocido como *bloque génesis*.

Aunque un bloque solo tiene un padre, puede tener temporalmente varios hijos. Cada uno de los hijos tiene una referencia al mismo bloque, al que consideran su padre, y cada uno de los hijos contiene también el mismo hash (de padre) en el campo "hash de bloque anterior". Los hijos múltiples surgen durante una bifurcación (en inglés, "fork") de la cadena de bloques, una situación temporal que se produce cuando se descubren diferentes bloques casi simultáneamente por diferentes mineros (ver Bifurcaciones de la Cadena de Bloques). Con el tiempo, un bloque hijo se convierte en parte de la cadena de bloques y la "bifurcación" se resuelve. A pesar de que un bloque puede tener más de un hijo, cada bloque solo puede tener un padre. Esto se debe a que un bloque tiene un solo campo "hash de bloque anterior" que hace referencia a su único padre.

El campo "hash de bloque anterior" está dentro de la cabecera de bloque y por lo tanto afecta al hash del bloque actual. La identidad propia del hijo cambia si la identidad de los padres cambia. Cuando el padre se modifica de alguna manera, los cambios de hash de los padres tambien cambian. Cuando el hash del padre cambia requiere un cambio en el puntero "hash de bloque anterior" del hijo. Esto a su vez hace que el hash del hijo cambie, lo que requiere un cambio en el puntero del nieto, que a su vez cambia el nieto, y así sucesivamente. Este efecto cascada asegura que, una vez que un bloque tiene muchas generaciones siguientes, no puede ser cambiado sin forzar un nuevo cálculo de todos los bloques siguientes. Debido a que tal recálculo requeriría un cálculo enorme (y, por lo tanto, consumo de energía), la existencia de una larga cadena de bloques hace que la historia profunda de la cadena de bloques sea inmutable, lo que es una característica clave de la seguridad de Bitcoin.

Una forma de pensar en la cadena de bloques es considerándola como una serie de capas en una formación geológica o como una muestra de núcleo de glaciar. Las capas superficiales pueden cambiar con las estaciones o incluso volar, antes de que tengan tiempo de asentarse. Pero una vez que nos adentramos unos centímetros de profundidad, las capas geológicas se vuelven cada vez más estables. Cuando miramos unos cientos de pies hacia abajo, estamos mirando una instantánea del pasado que ha permanecido intacta durante millones de años. En la cadena de bloques, los pocos bloques más recientes podrían revisarse, en caso de haber un recálculo de la cadena debido a una bifurcación. Los seis bloques superiores son como unas pocas pulgadas de tierra vegetal. Pero una vez que profundizamos en la cadena de bloques, más allá de seis bloques, los bloques tienen cada vez menos probabilidades de cambiar. Más allá de 100 bloques de profundidad, hay tanta estabilidad que es posible gastarse la transacción coinbase, –la transacción que contiene los bitcoins recién minados–. A más de unos miles de bloques de profundidad (es decir, lo que sucedió hace un mes) y la cadena de bloques es un historial establecido, a todos los efectos prácticos. Si bien el protocolo siempre permite que una cadena se deshaga en pos de otra cadena más larga, y si bien existe la posibilidad de revertir cualquier bloque, la probabilidad de tal evento disminuye a medida que pasa el tiempo hasta que se vuelve infinitesimal.

Estructura de un Bloque

Un bloque es una estructura de datos contenedor que agrupa las transacciones para su inclusión en el libro de contabilidad público, la cadena de bloques. El bloque se compone de una cabecera, que contiene metadatos, seguido por una larga lista de operaciones que componen la mayor parte de su tamaño. La cabecera de bloque es de 80 bytes, mientras que la transacción promedio es de al menos 400 bytes y el bloque promedio contiene más de 1900 transacciones. Un bloque completo, con todas las transacciones, por lo tanto, es 10.000 veces más grande que la cabecera de bloque. La

estructura de un bloque describe la estructura de un bloque.

Table 22. La estructura de un bloque

Tamaño	Campo	Descripción
4 bytes	Tamaño de Bloque	El tamaño del bloque en bytes después de este campo
80 bytes	Cabecera de Bloque	Varios campos componen la cabecera del bloque
1–9 bytes (VarInt)	Contador de Transacciones	Cuántas transacciones siguen
Variable	Transacciones	Las transacciones registradas en este bloque

Cabecera de Bloque

La cabecera de bloque se compone de tres conjuntos de metadatos de bloque. En primer lugar, hay una referencia a un hash de bloque anterior, que conecta este bloque al bloque anterior en la cadena de bloques. El segundo conjunto de metadatos, concretamente, *dificultad*, *sello de tiempo* y *nonce*, están relacionados con la competencia en la minería, como se detalla en Minería y Consenso. La tercera parte de los metadatos es la raíz del árbol de merkle, una estructura de datos utilizada para resumir de manera eficiente todas las transacciones en el bloque. La estructura de la cabecera de bloque describe la estructura de una cabecera de bloque.

Table 23. La estructura de la cabecera de bloque

Tamaño	Campo	Descripción
4 bytes	Versión	Un número de versión para seguir las actualizaciones de software y protocolo
32 bytes	Hash del Bloque Anterior	Una referencia al hash del bloque anterior (padre) en la cadena
32 bytes	Raíz Merkle	Un hash de la raíz del árbol merkle de las transacciones de este bloque
4 bytes	Sello de Tiempo	El tiempo de creación aproximada de este bloque (segundos desde Unix Epoch)
4 bytes	Objetivo de Dificultad	El objetivo de dificultad del algoritmo de Prueba-de-Trabajo para este bloque
4 bytes	Nonce	Un contador usado para el algoritmo de Prueba-de-Trabajo

El nonce, objetivo de dificultad y sello de tiempo son usados en el proceso de minería y serán analizados en mayor detalle en Minería y Consenso.

Identificadores de Bloque: Hash de Cabecera de Bloque y Altura de Bloque

El identificador primario de un bloque es su hash criptográfico, una huella digital, que se obtiene al hacer hash de la cabecera de bloque dos veces a través del algoritmo SHA256. El hash de 32 bytes resultante se llama el *hash de bloque* pero es más preciso llamarlo el *hash de cabecera de bloque*, porque solo se usa la cabecera de bloque para calcularlo. Por ejemplo, 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f es el hash de bloque del primer bloque de bitcoin jamás creado. El hash de bloque identifica un bloque de forma única e inequívoca y se puede derivar de forma independiente por cualquier nodo simplemente haciendo hash de la cabecera de bloque.

Ten en cuenta que el hash de bloque no está realmente incluido dentro de la estructura de datos del bloque, ni cuando el bloque es transmitido en la red, ni cuando se guarda en el almacenamiento persistente de un nodo como parte de la cadena de bloques. En cambio, cada nodo calcula el hash de bloque cuando recibe el bloque de la red. El hash de bloque podría ser almacenado en una tabla separada de la base de datos como parte de los metadatos del bloque, para facilitar la

indexación y hacer más rápida la recuperación de los bloques desde el disco.

Una segunda manera de identificar un bloque es por su posición en la cadena de bloques, denominada la *altura de bloque*. El primer bloque jamás creado está a la altura de bloque 0 (cero) y es el mismo bloque que se ha referenciado anteriormente con el siguiente hash de bloque 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. Así, un bloque se puede identificar de dos maneras: haciendo referencia al hash de bloque o haciendo referencia a la altura de bloque. Cada bloque posterior que se añade "encima" de ese primer bloque está en una posición "superior" en la cadena de bloques, como cajas apiladas una encima de la otra. El 1 de enero de 2017 la altura de bloque era 446.000 aproximadamente, lo que significa que había 446.000 bloques apilados en la parte superior del primer bloque creado en enero de 2009.

A diferencia del hash de bloque, la altura de bloque no es un identificador único. Aunque cada bloque siempre tendrá una altura de bloque específica e invariante, lo contrario no es cierto—la altura de bloque no siempre identifica a un solo bloque. Dos o más bloques que compiten por la misma posición en la cadena de bloques podrían tener la misma altura de bloque. Este escenario se discute en detalle en la sección Bifurcaciones de la Cadena de Bloques. Además, la altura de bloque no forma parte de la estructura de datos del bloque; no se almacena dentro del bloque. Cada nodo identifica dinámicamente la posición de un bloque (altura) en la cadena de bloques cuando se recibe desde la red bitcoin. La altura de bloque también podría almacenarse como metadatos en una tabla indexada de base de datos para recuperarlo más rápidamente.

TIP

El *hash de bloque* de un bloque siempre identifica un bloque de forma única. Un bloque también tiene siempre una *altura de bloque* específica. Sin embargo, no siempre una altura de bloque concreta identifica a un único bloque. Más bien, dos o más bloques pueden competir por una misma posición en la cadena de bloques.

El Bloque Génesis

El primer bloque en la cadena de bloques se llama el bloque génesis y fue creado en 2009. Es el ancestro común de todos los bloques en la cadena de bloques, lo que significa que si comienzas en cualquier bloque y sigues la cadena hacia atrás en el tiempo, finalmente llegarás al bloque génesis.

Cada nodo siempre comienza con una cadena de bloques de al menos un bloque ya que el bloque génesis está codificado de forma estática en el software del cliente bitcoin, de forma que no pueda ser alterado. Cada nodo siempre "sabe" el hash y estructura del bloque génesis, la fecha fija en que fue creado, e incluso la única transacción contenida en él. Por lo tanto, cada nodo tiene el punto de partida para la cadena de bloques, una "raíz" segura desde la que construir una cadena de bloques de confianza.

Mira el bloque génesis codificado estáticamente dentro del Cliente Principal, Bitcoin Core, en *chainparams.cpp*.

El siguiente identificador de hash pertenece al bloque génesis:

```
000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f
```

Puedes buscar ese hash de bloque en cualquier sitio web de explorador de bloques, como *blockchain.info*, y te llevará a una página que describe el contenido de este bloque, con una dirección URL que contiene ese hash:

https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Usando el cliente de referencia Bitcoin Core en la línea de comandos:

```
$ bitcoin-cli getblock 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f


{
    "hash" : "000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f",
    "confirmations" : 308321,
    "size" : 285,
    "height" : 0,
    "version" : 1,
    "merkleroot" : "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b",
    "tx" : [
        "4a5e1e4baab89f3a32518a88c31bc87f618f76673e2cc77ab2127b7afdeda33b"
    ],
```

```
    "time" : 1231006505,
    "nonce" : 2083236893,
    "bits" : "1d00ffff",
    "difficulty" : 1.00000000,
    "nextblockhash" : "00000000839a8e6886ab5951d76f411475428afc90947ee320161bbf18eb6048"
}
```

El bloque génesis contiene un mensaje oculto en su interior. La entrada de transacción coinbase contiene el texto "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks." (traducido al español: "The Times 03/Ene/2009 Canciller preparado para segundo rescate a los bancos."). Este mensaje tenía la intención de ofrecer la prueba de la fecha más antigua en la que este bloque fue creado, haciendo referencia al titular del periódico británico *The Times*. También sirve como un recordatorio irónico de la importancia de un sistema monetario independiente, precisamente cuando el lanzamiento de bitcoin coincide en el tiempo con una crisis monetaria mundial sin precedentes. El mensaje se registró en el primer bloque por Satoshi Nakamoto, creador de bitcoin.

Enlazando Bloques en la Cadena de Bloques

Los nodos completos de bitcoin mantienen una copia local de la cadena de bloques, comenzando en el bloque génesis. La copia local de la cadena de bloques se actualiza constantemente a medida que se encuentran y se utilizan nuevos bloques para extender la cadena. A medida que un nodo recibe bloques entrantes desde la red, validará estos bloques y luego los enlazará a la cadena de bloques existente. Para establecer un enlace, un nodo examinará la cabecera de bloque entrante buscando el "hash de bloque anterior."

Supongamos, por ejemplo, que un nodo tiene 277.314 bloques en la copia local de la cadena de bloques. El último bloque que el nodo conoce es el bloque 277.314, con un hash de cabecera de bloque de:

```
000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249
```

El nodo bitcoin recibe después un nuevo bloque de la red, que se analiza de la siguiente manera:

```
{
    "size" : 43560,
    "version" : 2,
    "previousblockhash" :
        "000000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
    "merkleroot" :
        "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
    "time" : 1388185038,
    "difficulty" : 1180923195.25802612,
    "nonce" : 4215469401,
    "tx" : [
        "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

 #[... muchas otras transacciones omitidas ...]

        "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"
    ]
}
```

Interpretando este nuevo bloque, el nodo encuentra el campo previousblockhash, que contiene el hash de su bloque padre. Es un hash que el nodo ya conocía, y que corresponde al último bloque en la cadena, a la altura de 277.314. Por lo tanto, este nuevo bloque es un hijo del último bloque de la cadena y extiende la cadena de bloques existente. El nodo añade este nuevo bloque al final de la cadena, añadiendo a la cadena de bloques una nueva altura de 277.315. Bloques enlazados en una cadena, por referencia al hash de la cabecera de bloque anterior muestra la cadena de tres bloques, enlazados por referencias en el campo previousblockhash.

Árboles Merkle

Cada bloque en la cadena de bloques bitcoin contiene un resumen de todas las transacciones en el bloque, utilizando un *árbol merkle*.

Un *árbol merkle*, también conocido como un *árbol hash binario*, es una estructura de datos que se usa para resumir y verificar de manera eficiente la integridad de grandes conjuntos de datos. Los árboles merkle son árboles binarios que contienen hashes criptográficos. El término "árbol" se usa en informática para describir una estructura de datos de ramificación, pero estos árboles por lo general aparecen al revés, con la "raíz" en la parte superior y las "hojas" en la parte inferior de un diagrama, como se verá en los ejemplos que siguen.

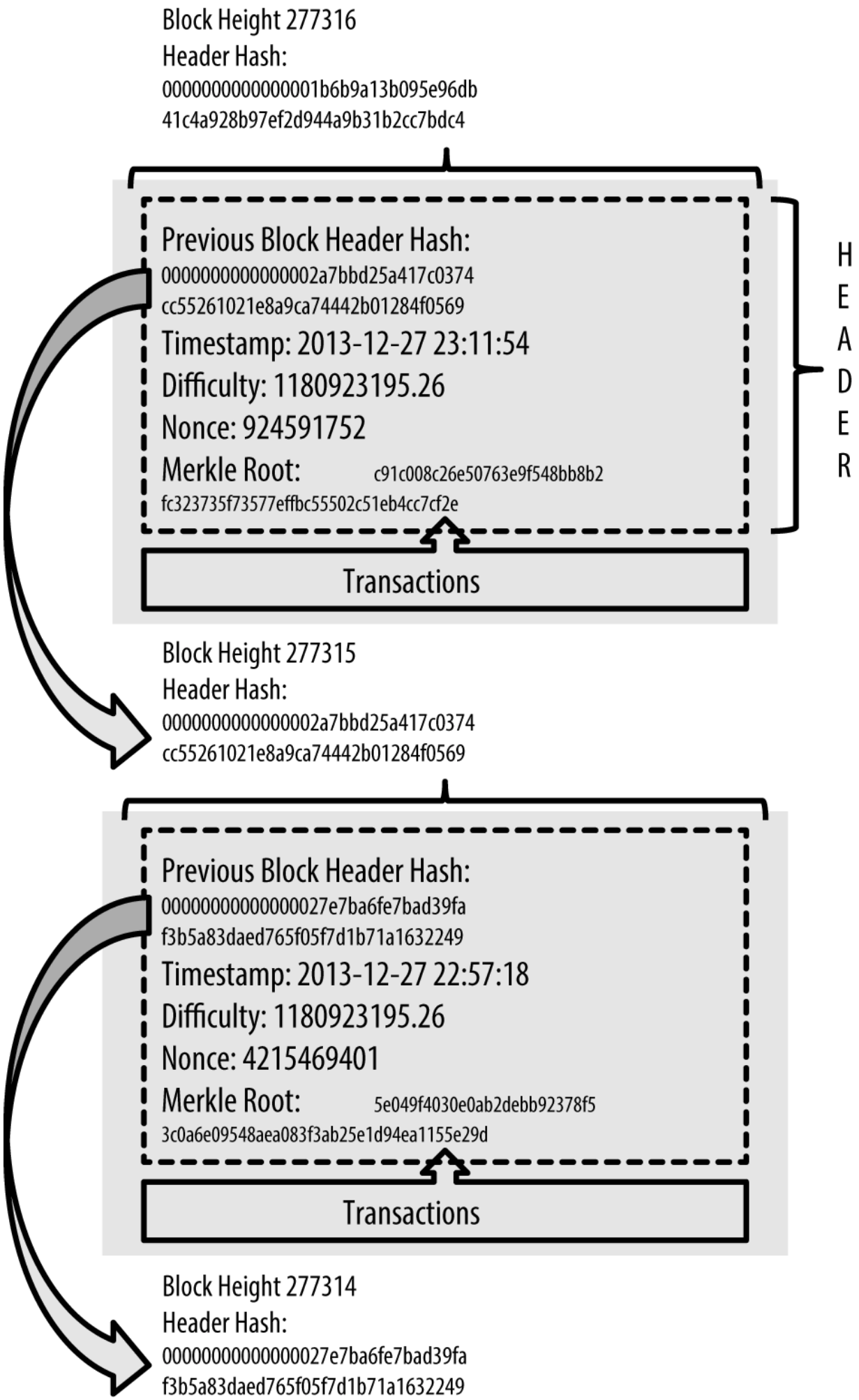

Block Height 277316
Header Hash:
000000000000000001b6b9a13b095e96db
41c4a928b97ef2d944a9b31b2cc7bdc4
Previous Block Header Hash:
000000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Timestamp: 2013-12-27 23:11:54
Difficulty: 1180923195.26
Nonce: 924591752
Merkle Root: c91c008c26e50763e9f548bb8b2
fc323735f73577effbc55502c51eb4cc7cf2e
Transactions
HEADER
Block Height 277315
Header Hash:
000000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Previous Block Header Hash:
000000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
Timestamp: 2013-12-27 22:57:18
Difficulty: 1180923195.26
Nonce: 4215469401
Merkle Root: 5e049f4030e0ab2debb92378f5
3c0a6e09548aea083f3ab25e1d94ea1155e29d
Transactions
Block Height 277314
Header Hash:
000000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249

Figure 59. Bloques enlazados en una cadena, por referencia al hash de la cabecera de bloque anterior

Los árboles de merkle se usan en bitcoin para resumir todas las transacciones en un bloque, produciendo una huella digital completa de todo el conjunto de transacciones, proporcionando un proceso muy eficiente para verificar si una transacción está incluida en un bloque. Un árbol de merkle se construye mediante la ejecución de una función de hash en pares de nodos de forma recursiva hasta que solo queda un único hash, al que se le llama *raíz* o *raíz merkle*. El algoritmo de hash criptográfico utilizado en los árboles de merkle de bitcoin es SHA256 aplicado dos veces, también conocido como doble SHA256.

Cuando se toman N elementos de datos, se hace hash de cada uno de ellos y se resumen en un árbol merkle, se puede comprobar si cualquier elemento de datos está incluido en el árbol con un máximo de 2*log~2~(N) cálculos, convirtiéndolo en una estructura de datos muy eficiente.

El árbol de merkle se construye de abajo hacia arriba. En el siguiente ejemplo, comenzamos con cuatro transacciones, A, B, C y D, que forman la *hojas* del árbol de merkle, como se muestra en Calculando los nodos en un árbol merkle. Las transacciones no se almacenan en el árbol de merkle; más bien, se hace hash de sus datos y el hash resultante se almacena en cada nodo hoja como H_A, H_B, H_C, y H_D:

H_A = SHA256(SHA256(Transaction A))

Después, los pares consecutivos de nodos hoja se resumen en un nodo padre, concatenando los dos hashes y haciendo hash de ese dato concatenado. Por ejemplo, para construir el nodo padre H_{AB}, los dos valores hash de 32 bytes de los hijos se concatenan para crear una cadena de 64 bytes. Se hace entonces un doble-hash de esa cadena para producir el hash del nodo padre:

H_{AB} = SHA256(SHA256(H_A + H_B))

El proceso continúa hasta que solo hay un nodo en la parte superior, el nodo conocido como la raíz merkle. Ese hash de 32 bytes se almacena en la cabecera de bloque y resume todos los datos de las cuatro transacciones. Calculando los nodos en un árbol merkle muestra cómo la raíz se calcula mediante hashes por pares de los nodos.

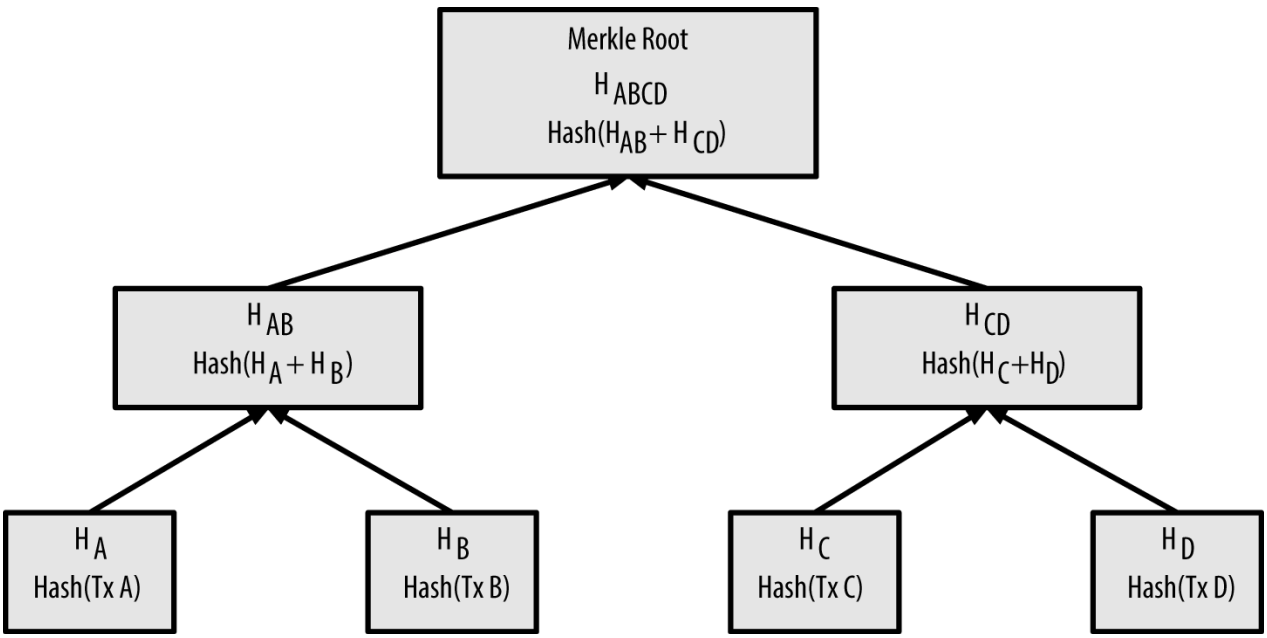


Figure 60. Calculando los nodos en un árbol merkle

Debido a que el árbol de merkle es un árbol binario, necesita un número par de nodos hoja. Si hay un número impar de transacciones para resumir, el último hash de transacción se duplicará para crear un número par de nodos hoja, también conocido como *árbol balanceado*. Esto se muestra en Duplicando un elemento de datos para alcanzar un número par de elementos de datos, donde se duplica la transacción C.

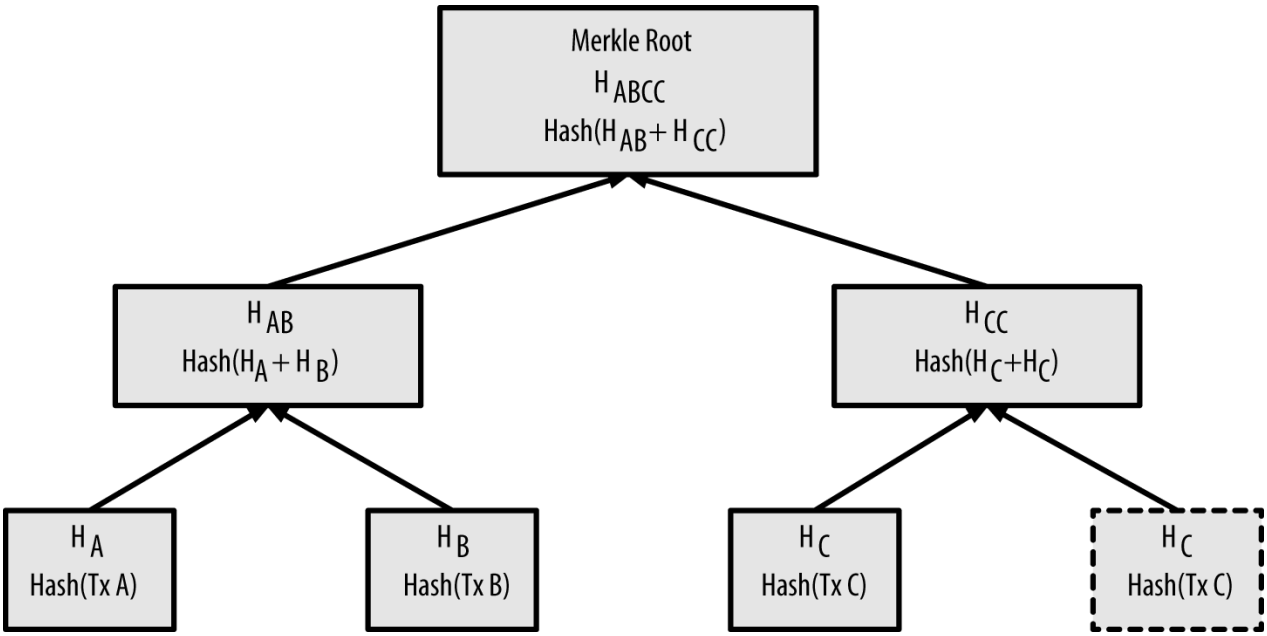


Figure 61. Duplicando un elemento de datos para alcanzar un número par de elementos de datos

El mismo método para la construcción de un árbol de cuatro transacciones se puede generalizar para construir árboles de cualquier tamaño. En bitcoin es común tener de varios cientos a más de mil transacciones en un solo bloque, que se resumen de la misma forma, produciendo solo 32 bytes de datos en una única raíz merkle. En Un árbol merkle resumiendo muchos elementos de datos, verás un árbol construido a partir de 16 transacciones. Ten en cuenta que, aunque la raíz se ve más grande que los nodos hoja en el diagrama, tiene exactamente el mismo tamaño, solo 32 bytes. Independientemente de si existe una transacción o cien mil transacciones en el bloque, la raíz merkle siempre los resume en 32 bytes.

Para demostrar que una transacción específica está incluida en un bloque, un nodo solo necesita producir log~2~(N) hashes de 32 bytes, elaborando un *camino de autenticación* o *camino merkle* que conecte la transacción específica a la raíz del árbol. Esto es especialmente importante a medida que el número de transacciones aumenta, porque el logaritmo en base-2 del número de transacciones aumenta mucho más lentamente. Esto permite que los nodos bitcoin produzcan eficientemente caminos de 10 o 12 hashes (320-384 bytes), que pueden proporcionar la prueba de la existencia de una sola transacción entre más de mil transacciones en un bloque de un megabyte de tamaño.

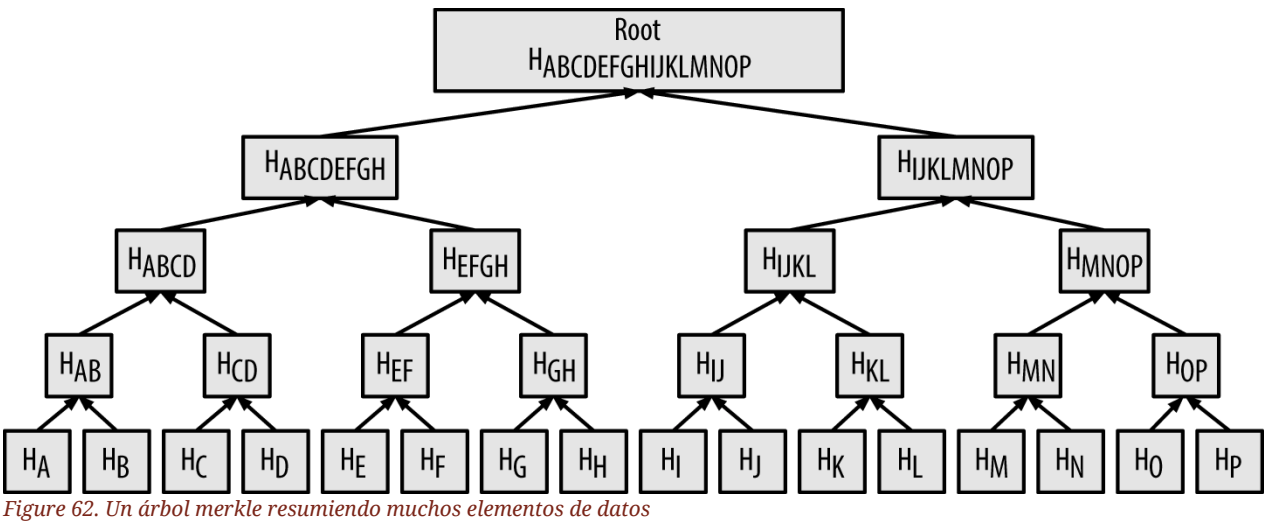


Figure 62. Un árbol merkle resumiendo muchos elementos de datos

En Una ruta de merkle utilizado para probar la inclusión de un elemento de datos, un nodo puede demostrar que una transacción K está incluida en el bloque mediante la producción de un ruta de merkle que ocupa solo cuatro hashes de 32-bytes de largo (128 bytes en total). El camino consta de los cuatro hashes (señalados en fondo sombreado en Una ruta de merkle utilizado para probar la inclusión de un elemento de datos) H_L, H_{IJ}, H_{MNOP} y $H_{ABCDEFGH}$. Con esos cuatro hashes suministrados a modo de ruta de autenticación, cualquier nodo puede demostrar que H_K (con un fondo negro en la parte inferior del diagrama) está incluido en la raíz de merkle mediante el cálculo de cuatro hashes adicionales por pares H_{KL}, H_{IJKL}, $H_{IJKLMNOP}$, y la raíz del árbol de merkle (descrito en una línea de puntos en el diagrama).

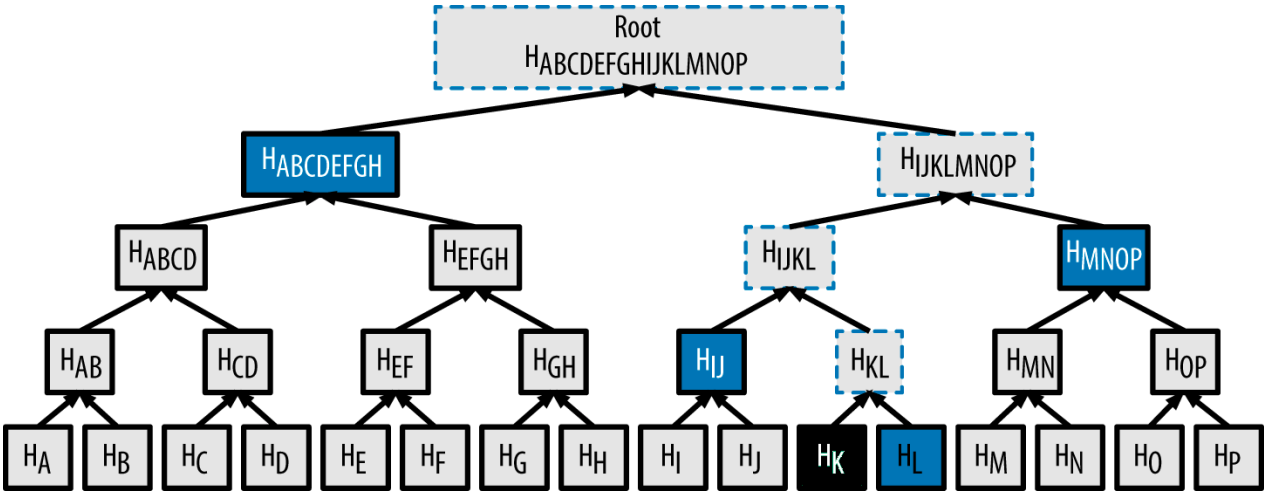


Figure 63. Una ruta de merkle utilizado para probar la inclusión de un elemento de datos

El código en Construyendo un árbol merkle demuestra el proceso de crear un árbol merkle desde el hash del nodo hoja hasta la raíz, utilizando la biblioteca libbitcoin para algunas funciones auxiliares.

Example 20. Construyendo un árbol merkle

```
#include <bitcoin/bitcoin.hpp>

bc::hash_digest create_merkle(bc::hash_list& merkle)
{
    // Stop if hash list is empty.
    if (merkle.empty())
        return bc::null_hash;
    else if (merkle.size() == 1)
        return merkle[0];

    // While there is more than 1 hash in the list, keep looping...
    while (merkle.size() > 1)
    {
        // If number of hashes is odd, duplicate last hash in the list.
        if (merkle.size() % 2 != 0)
            merkle.push_back(merkle.back());
        // List size is now even.
        assert(merkle.size() % 2 == 0);

        // New hash list.
        bc::hash_list new_merkle;
        // Loop through hashes 2 at a time.
        for (auto it = merkle.begin(); it != merkle.end(); it += 2)
        {
            // Join both current hashes together (concatenate).
            bc::data_chunk concat_data(bc::hash_size * 2);
            auto concat = bc::serializer<
                decltype(concat_data.begin())>(concat_data.begin());
            concat.write_hash(*it);
            concat.write_hash(*(it + 1));
            // Hash both of the hashes.
            bc::hash_digest new_root = bc::bitcoin_hash(concat_data);
            // Add this to the new list.
            new_merkle.push_back(new_root);
        }
        // This is the new list.
        merkle = new_merkle;

        // DEBUG output -------------------------------------
        std::cout << "Current merkle hash list:" << std::endl;
        for (const auto& hash: merkle)
            std::cout << "  " << bc::encode_base16(hash) << std::endl;
        std::cout << std::endl;
        // --------------------------------------------------
    }
    // Finally we end up with a single item.
    return merkle[0];
}

int main()
{
    // Replace these hashes with ones from a block to reproduce the same merkle root.
    bc::hash_list tx_hashes{{
```

```
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000000"),
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000011"),
        bc::hash_literal("0000000000000000000000000000000000000000000000000000000000000022"),
    }};
    const bc::hash_digest merkle_root = create_merkle(tx_hashes);
    std::cout << "Result: " << bc::encode_base16(merkle_root) << std::endl;
    return 0;
}
```

Compilando y ejecutando el código de ejemplo merkle muestra el resultado de compilar y ejecutar el código merkle.

Example 21. Compilando y ejecutando el código de ejemplo merkle

```
$ # Compilar el código merkle.cpp
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Ejecutar el ejecutable merkle
$ ./merkle
Lista actual de hash merkle:
  32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006
  30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4

Lista actual de hash merkle:
  d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3
```

La eficiencia de los árboles de merkle se hace evidente a medida que aumenta la escala. Eficiencia de un árbol merkle muestra la cantidad de datos que necesitan intercambiarse como un ruta de merkle para demostrar que una transacción está incluida en un bloque.

Table 24. Eficiencia de un árbol merkle

Número de transacciones	Tamaño aprox. del bloque	tamaño de camino (hashes)	Tamaño de camino (bytes)
16 transacciones	4 kilobytes	4 hashes	128 bytes
512 transacciones	128 kilobytes	9 hashes	288 bytes
2048 transacciones	512 kilobytes	11 hashes	352 bytes
65.535 transacciones	16 megabytes	16 hashes	512 bytes

Como se puede ver en la tabla, mientras que el tamaño de bloque aumenta rápidamente, de 4 KB con 16 transacciones a un tamaño de bloque de 16 MB para incluir a 65.535 transacciones, la ruta de merkle requerido para demostrar la inclusión de una transacción aumenta mucho más lentamente, de 128 bytes a solo 512 bytes. Con árboles de merkle, un nodo puede descargar solo las cabeceras de bloque (80 bytes por bloque) y aún así ser capaz de identificar la inclusión de una transacción en un bloque mediante la recuperación de una ruta de merkle pequeña de un nodo completo, sin almacenar o transmitir la gran mayoría de la cadena de bloques, que puede ser de varios gigabytes de tamaño. Los nodos que no mantienen una cadena de bloques completa, llamados nodos de verificación de pago simplificado (nodos SPV), usan rutas de merkle para verificar las transacciones sin necesidad de descargar bloques completos .

Árboles Merkle y Verificación de Pago Simplificada (SPV)

Los árboles de merkle son ampliamente utilizados por los nodos SPV. Los nodos SPV no tienen todas las transacciones y no descargan bloques completos, solo las cabeceras de bloque. Con el fin de verificar que una transacción está incluida en un bloque sin tener que descargar todas las transacciones del bloque, utilizan una ruta de autenticación, o ruta de merkle.

Consideremos, por ejemplo, un nodo SPV que esté interesado en los pagos entrantes a una dirección incluida en su cartera. El nodo SPV establecerá un filtro de bloom en sus conexiones con sus compañeros para limitar las transacciones recibidas a solo aquellas que contengan direcciones de interés. Cuando un compañero vea una transacción que coincida con el filtro de bloom (ver Filtros de Bloom), enviará ese bloque usando un mensaje merkleblock. El mensaje merkleblock contiene la cabecera de bloque, así como una ruta de merkle que vincula la transacción de interés con la raíz de merkle en el bloque. El nodo SPV puede utilizar esta ruta de merkle para conectar la transacción con el bloque y verificar que la transacción está incluida en el bloque. El nodo SPV también utiliza la cabecera de bloque para vincular el bloque con el resto de la cadena de bloques. La combinación de estos dos enlaces, entre la transacción y bloque, y entre el bloque y la

cadena de bloques, prueba que la transacción está registrada en la cadena de bloques. Con todo, el nodo SPV habrá recibido menos de un kilobyte de datos para la cabecera de bloque y la ruta de merkle, una cantidad de datos que es más de mil veces menor que un bloque completo (aproximadamente 1 megabyte actualmente).

Cadenas de Bloques de Test Bitcoin

Es posible que te sorprenda saber que hay más de una cadena de bloques de bitcoin. La cadena de bloques de bitcoin "principal", la creada por Satoshi Nakamoto el 3 de enero de 2009, la que tiene el bloque génesis que estudiamos en este capítulo, se llama *mainnet*. Hay otras cadenas de bloques de bitcoin que se utilizan para fines de pruebas: en este momento *testnet*, *segnet* y *regtest*. Echemos un vistazo a cada una de ellas.

Testnet—La Zona de Pruebas de Bitcoin

Testnet es el nombre de la cadena de bloques de prueba, la red y la moneda que se utiliza para fines de prueba. Testnet es una red P2P en vivo con todas las funciones, con billeteras, bitcoins de prueba (monedas testnet), minería y todas las demás características de la mainnet. En realidad, solo hay dos diferencias: las monedas de testnet no tienen valor y la dificultad para la minería debe ser lo suficientemente baja como para que cualquiera pueda extraer monedas de la testnet con relativa facilidad (manteniéndolas sin valor).

Cualquier desarrollo de software que esté destinado al uso en producción en la red principal de bitcoin (en inglés, mainnet), debe probarse primero en la red de pruebas (en inglés, testnet) con monedas de prueba. Esto protege tanto a los desarrolladores de pérdidas monetarias debido a errores, como a la red del comportamiento no deseado debido a errores.

Sin embargo, mantener las monedas sin valor y la minería fácil no es tarea fácil. A pesar de las peticiones de los desarrolladores, algunas personas utilizan equipos de minería avanzados (GPU y ASIC) para minar en testnet. Esto aumenta la dificultad, hace que sea imposible minar con una CPU y, finalmente, hace que sea lo suficientemente difícil obtener monedas de prueba para que la gente comience a valorarlas, por lo que no son totalmente sin valor. Como resultado, de vez en cuando, el testnet debe desecharse y reiniciarse desde un nuevo bloque génesis, restableciendo la dificultad.

El testnet actual se llama *testnet3*, la tercera iteración de testnet, que se reinició en febrero de 2011 para restablecer la dificultad del testnet anterior.

Ten en cuenta que testnet3 es una gran cadena de bloques, que supera los 20 GB a principios de 2017. Tardarás aproximadamente un día en sincronizarse por completo y utilizar los recursos de tu computadora. No tanto como mainnet, pero tampoco exactamente "ligero". Una buena manera de ejecutar un nodo de testnet es como una imagen de máquina virtual (por ejemplo, VirtualBox, Docker, Cloud Server, etc.) dedicada para ese propósito.

Usando testnet

Bitcoin Core, como casi todos los demás programas de bitcoin, tiene soporte completo para la operación en testnet en lugar de mainnet. Todas las funciones de Bitcoin Core funcionan en testnet, incluida la cartera, la minería de monedas de testnet y la sincronización de un nodo de testnet completo.

Para iniciar Bitcoin Core en testnet en lugar de mainnet, usa la opción testnet:

```
$ bitcoind -testnet
```

En los logs, deberías ver que bitcoind está creando una nueva cadena de bloques en el subdirectorio testnet3 del directorio de bitcoind predeterminado:

```
bitcoind: Using data directory /home/username/.bitcoin/testnet3
```

Para conectarse a bitcoind, usa la herramienta de línea de comandos bitcoin-cli, pero también debes marcarlo al modo testnet:

```
$ bitcoin-cli -testnet getblockchaininfo
{
  "chain": "test",
  "blocks": 1088,
  "headers": 139999,
  "bestblockhash": "0000000063d29909d475a1c4ba26da64b368e56cce5d925097bf3a2084370128",
  "difficulty": 1,
```

```
  "mediantime": 1337966158,
  "verificationprogress": 0.001644065914099759,
  "chainwork": "0000000000000000000000000000000000000000000000000000044104410441",
  "pruned": false,
  "softforks": [

  [...]
```

También puedes ejecutar en testnet3 con otras implementaciones de nodo completo, como btcd (escrito en Go) y bcoin (escrito en JavaScript), para experimentar y aprender otros lenguajes de programación y frameworks.

A principios de 2017, testnet3 es compatible con todas las funciones de mainnet, incluido Segregated Witness (ver Segregated Witness (Testigos Segregados)). Por lo tanto, testnet3 también se puede utilizar para probar las características de Segregated Witness.

Segnet—El Testnet de Segregated Witness

En 2016, se lanzó una red de prueba de propósito especial para ayudar en el desarrollo y prueba de Segregated Witness (también conocido como segwit; ver Segregated Witness (Testigos Segregados)). Esta cadena de bloques de prueba se llama segnet y te puedes unir a ella ejecutando una versión especial (rama) de Bitcoin Core.

Desde que se agregó segwit a testnet3, ya no es necesario usar segnet para probar las funciones de segwit.

En el futuro, es probable que veamos otras cadenas de bloques de testnet que estén diseñadas específicamente para probar una característica única o un cambio arquitectónico importante, como segnet.

Regtest—La Cadena de Bloques Local

Regtest, que significa "Pruebas de Regresión", es una característica de Bitcoin Core que permite crear una cadena de bloques local para fines de prueba. A diferencia de testnet3, que es una cadena de bloques de prueba pública y compartida, las cadenas de bloques de regtest se ejecutan como sistemas cerrados para pruebas locales. Lanzas una cadena de bloques regtest desde cero, creando un bloque génesis local. Puedes agregar otros nodos a la red, o ejecutar la cadena con un solo nodo para probar únicamente el software de Bitcoin Core.

Para comenzar Bitcoin Core en modo regtest, se usar la opción regtest:

```
$ bitcoind -regtest
```

Al igual que con testnet, Bitcoin Core inicializará una nueva cadena de bloques en el subdirectorio *regtest* de tu directorio predeterminado de bitcoind:

```
bitcoind: Using data directory /home/username/.bitcoin/regtest
```

Para utilizar la herramienta de línea de comandos, también debes especificar la opción regtest. Probemos el comando getblockchaininfo para inspeccionar la cadena de bloques regtest:

```
$ bitcoin-cli -regtest getblockchaininfo
{
  "chain": "regtest",
  "blocks": 0,
  "headers": 0,
  "bestblockhash": "0f9188f13cb7b2c71f2a335e3a4fc328bf5beb436012afca590b1a11466e2206",
  "difficulty": 4.656542373906925e-10,
  "mediantime": 1296688602,
  "verificationprogress": 1,
  "chainwork": "0000000000000000000000000000000000000000000000000000000000000002",
  "pruned": false,
  [...]
```

Como puedes ver, todavía no hay bloques. Vamos a minar algunos (500 bloques) y ganar la recompensa:

```
$ bitcoin-cli -regtest generate 500
[
  "7afed70259f22c2bf11e406cb12ed5c0657b6e16a6477a9f8b28e2046b5ba1ca",
  "1aca2f154a80a9863a9aac4c72047a6d3f385c4eec5441a4aafa6acaa1dada14",
  "4334ecf6fb022f30fbd764c3ee778fabbd53b4a4d1950eae8a91f1f5158ed2d1",
  "5f951d34065efeaf64e54e91d00b260294fcdfc7f05dbb5599aec84b957a7766",
```

```
  "43744b5e77c1dfece9d05ab5f0e6796ebe627303163547e69e27f55d0f2b9353",
   [...]
  "6c31585a48d4fc2b3fd25521f4515b18aefb59d0def82bd9c2185c4ecb754327"
]
```

Solo tomará unos segundos minar todos estos bloques, lo que ciertamente facilita la prueba. Si verificas el saldo de tu cartera, verás que ganaste la recompensa por los primeros 400 bloques (las recompensas de coinbase deben tener una profundidad de 100 bloques antes de poder gastarlas):

```
$ bitcoin-cli -regtest getbalance
12462.50000000
```

Usando Cadenas de Bloques de Prueba para Desarrollo

Las diversas cadenas de bloques de bitcoin (regtest, segnet, testnet3, mainnet) ofrecen un rango de entornos de prueba para el desarrollo de bitcoin. Usa las cadenas de bloques de prueba si estás desarrollando para Bitcoin Core u otro cliente de consenso de nodo completo; una aplicación como una cartera, casa de intercambio, sitio de comercio electrónico; o incluso desarrollando nuevos contratos inteligentes y scripts complejos.

Puedes utilizar las cadenas de bloques de prueba para establecer un procedimiento de desarrollo. Prueba tu código localmente en regtest a medida que lo vas desarrollando. Una vez que esté listo para probarlo en una red pública, cambia a testnet para exponer tu programa a un entorno más dinámico con más diversidad de códigos y aplicaciones. Finalmente, una vez que estés seguro de que tu código funciona como se esperaba, cambia a mainnet para implementarlo en producción. A medida que realices cambios, mejoras, correcciones de errores, etc., vuelve a iniciar el procedimiento, implementando cada cambio primero en regtest, después en testnet, y finalmente en producción.

Minería y Consenso

Introducción

La palabra "minería" es un tanto engañosa. Al evocar la extracción de metales preciosos, se centra nuestra atención en la recompensa por la minería, los nuevos bitcoin creados en cada bloque. Aunque esta recompensa incentiva la minería, el propósito principal de la minería no es la recompensa o la generación de nuevas monedas. Si ves la minería solo como el proceso por el cual se crean las monedas, estás confundiendo los medios (incentivos) con el objetivo del proceso. La minería es el mecanismo que sustenta la cámara de compensación descentralizada, mediante el cual las transacciones se validan y autorizan. La minería es el invento que hace especial a bitcoin, un mecanismo de seguridad descentralizado que es la base del efectivo digital P2P.

La minería *asegura al sistema de bitcoin* y permite el surgimiento en toda la red del *consenso sin una autoridad central*. La recompensa de las monedas recién acuñadas y las comisiones de transacción es un esquema de incentivos que alinea las acciones de los mineros con la seguridad de la red, al mismo tiempo que implementa la oferta monetaria.

> TIP
> El propósito de la minería no es la creación de bitcoin nuevos. Ese es el sistema de incentivos. La minería es el mecanismo que permite que la *seguridad* de bitcoin sea *descentralizada*.

Los mineros validan nuevas transacciones y las graban en el libro contable global. Un nuevo bloque, que contiene las transacciones que tuvieron lugar desde el último bloque, se "extrae" cada 10 minutos de promedio, añadiendo esas transacciones a la cadena de bloques. Las transacciones que pasan a formar parte de un bloque y se agregan a la cadena de bloques se consideran "confirmadas", y permite a los nuevos propietarios de bitcoin gastar los bitcoin que recibieron en esas transacciones.

Los mineros reciben dos tipos de recompensas a cambio de la seguridad proporcionada por la minería: nuevas monedas creadas con cada nuevo bloque y comisiones de transacción de todas las transacciones incluidas en el bloque. Para ganar esta recompensa, los mineros compiten para resolver un problema matemático complejo basado en un algoritmo de hash criptográfico. La solución al problema, llamada Prueba-de-Trabajo, se incluye en el nuevo bloque y actúa como prueba de que el minero realizó un importante esfuerzo de computación. La competición para resolver el algoritmo de Prueba-de-Trabajo para ganar la recompensa y el derecho a registrar transacciones en la cadena de bloques es la base del modelo de seguridad de bitcoin.

El proceso se llama minería porque la recompensa (la nueva emisión de monedas) está diseñada para simular rendimientos decrecientes, al igual que la minería de metales preciosos. La oferta monetaria de Bitcoin se crea a través de la minería, de forma similar a cómo un banco central emite dinero nuevo imprimiendo billetes de banco. La cantidad máxima de bitcoins recién creados que un minero puede agregar a un bloque disminuye aproximadamente cada cuatro años (o precisamente cada 210,000 bloques). Comenzó siendo de 50 bitcoins por bloque en enero de 2009 y se redujo a la mitad, a 25 bitcoins por bloque en noviembre de 2012. Se redujo nuevamente a la mitad, a 12.5 bitcoins en julio de 2016 y nuevamente, a 6.25 bitcoins en mayo de 2020. Según esta fórmula, las recompensas de minería de bitcoin disminuirán exponencialmente hasta aproximadamente el año 2140, cuando se habrán emitido todos los bitcoins (20.99999998 millones). Después de 2140, no se emitirán nuevos bitcoins.

Los mineros de bitcoin también ganan comisiones por las transacciones. Cada transacción puede incluir una comisión de transacción, en forma de un excedente de bitcoin entre las entradas y salidas de la transacción. El minero de bitcoin ganador es el que "se queda con el cambio" en las transacciones incluidas en el bloque ganador. Hoy en día, las comisiones representan el 0,5% o menos de los ingresos de un minero de bitcoin, la gran mayoría proveniente de los bitcoin de nuevo cuño. Sin embargo, a medida que la recompensa disminuye con el tiempo y el número de transacciones por bloque aumenta, una mayor proporción de las ganancias de la minería de bitcoin provendrá de las comisiones. Gradualmente, la recompensa de minería estará dominada por las comisiones de transacción, que constituirán el incentivo principal para los mineros. Después de 2140, la cantidad de bitcoins nuevos en cada bloques se reduce a cero y la minería de bitcoin se incentivará solo por las comisiones de transacción.

En este capítulo, vamos a examinar primero la minería como un mecanismo de oferta monetaria y luego veremos la función más importante de la minería: el mecanismo de consenso descentralizado en el que se basa la seguridad de bitcoin.

Para comprender la minería y el consenso, seguiremos la transacción de Alice a medida que se recibe y se agrega a un

bloque por el equipo de minería de Jing. Después seguiremos el bloque a medida que es minado, se agrega a la cadena de bloques y es aceptado por la red bitcoin a través del proceso de consenso emergente.

Economía Bitcoin y Creación de Moneda

Los bitcoins son "acuñados" durante la creación de cada bloque a un ritmo fijo y decreciente. Cada bloque, generado en promedio cada 10 minutos, contiene bitcoins totalmente nuevos, creados de la nada. Cada 210.000 bloques, o aproximadamente cada cuatro años, la tasa de emisión de moneda se reduce en un 50%. Durante los primeros cuatro años de funcionamiento de la red, cada bloque contenía 50 nuevos bitcoins.

En noviembre de 2012, la nueva tasa de emisión de bitcoins se redujo a 25 bitcoins por bloque. En julio de 2016 se redujo a 12.5 bitcoins por bloque, y en mayo de 2020 se redujo nuevamente a 6.25 bitcoins por bloque. La tasa de monedas nuevas disminuye así exponencialmente en 32 "mitades" hasta el bloque 6,720,000 (que será minado aproximadamente en el año 2137), cuando se alcanzará la unidad monetaria mínima de 1 satoshi. Finalmente, después de 6.93 millones de bloques, en aproximadamente 2140, se emitirán casi 2,099,999,997,690,000 satoshis, o casi 21 millones de bitcoins. A partir de entonces, los bloques no contendrán nuevos bitcoins, y los mineros serán recompensados únicamente a través de las comisiones de las transacciones. La oferta de moneda bitcoin a lo largo del tiempo se basa en una velocidad de emisión geométricamente decreciente muestra el total de bitcoins en circulación a lo largo del tiempo, a medida que disminuye la emisión de monedas.

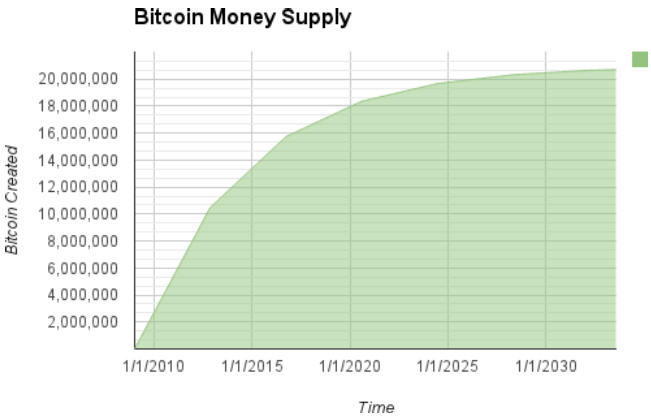


Figure 64. La oferta de moneda bitcoin a lo largo del tiempo se basa en una velocidad de emisión geométricamente decreciente

> **NOTE** El número máximo de monedas minadas es el *limite superior* de posibles recompensas de minería para bitcoin. En la práctica, un minero puede minar intencionadamente un bloque para tomar menos de la recompensa completa. Dichos bloques ya han sido extraídos y más pueden ser extraído en el futuro, lo que resulta en una emisión total más baja de moneda.

En el código de ejemplo Un script para calcular cuántos bitcoin serán emitidos en total calculamos el número total de bitcoin que serán emitidos.

Example 22. Un script para calcular cuántos bitcoin serán emitidos en total

```
# Original block reward for miners was 50 BTC = 50 0000 0000 Satoshis
start_block_reward = 50 * 10**8
# 210000 is around every 4 years with a 10 minute block interval
reward_interval = 210000


def max_money():
    current_reward = start_block_reward
    total = 0
    while current_reward > 0:
        total += reward_interval * current_reward
        current_reward /= 2
    return total


print("Total BTC to ever be created:", max_money(), "Satoshis")
```

Ejecutando el script max_money.py muestra la salida producida al ejecutar el script.

Example 23. Ejecutando el script max_money.py

```
$ python max_money.py
Total BTC to ever be created: 2099999997690000 Satoshis
```

La emisión finita y decreciente crea una oferta monetaria fija que resiste la inflación. A diferencia de una moneda fiduciaria, que un banco central puede imprimir en números infinitos, bitcoin nunca podrá sufrir de inflación mediante la impresión.

Dinero Deflacionario

La más importante y debatida consecuencia de la emisión monetaria fija y decreciente es que la moneda tiende a ser intrínsecamente *deflacionaria*. La deflación es el fenómeno de apreciación del valor debido a un desajuste entre oferta y demanda que hace aumentar el valor (y el tipo de cambio) de una moneda. Siendo lo opuesto a la inflación, la deflación de precios significa que el dinero adquiere mayor poder de compra con el tiempo.

Muchos economistas sostienen que la economía deflacionaria es un desastre y debe ser evitada a toda costa. Eso se debe a que en un período de deflación acelerada la gente tiende a acaparar dinero en vez de gastarlo, esperando que los precios caerán. Tal fenómeno se desató durante la "Década Perdida" de Japón, durante la cual un colapso completo de la demanda empujó a la moneda hacia una espiral deflacionaria.

Los expertos de bitcoin sostienen que la deflación no es mala por sí misma. En cambio, la deflación se asocia con el colapso de la demanda ya que ese es el único tipo de ejemplo de deflación que tenemos para estudiar. En una moneda fiduciaria con la posibilidad de impresión ilimitada es muy difícil entrar en una espiral deflacionaria a menos que ocurra un colapso completo en la demanda sumada a un rechazo a imprimir moneda. La deflación en bitcoin no está causada por un colapso en la demanda, sino por una oferta predecible y restringida.

El aspecto positivo de la deflación, por supuesto, es que es lo opuesto a la inflación. La inflación causa una devaluación lenta pero inevitable de la moneda, lo que resulta en una forma de impuesto oculto que castiga a los ahorradores para rescatar a los deudores (incluidos los deudores más grandes, los gobiernos mismos). Las monedas bajo el control del gobierno sufren el riesgo moral de una fácil emisión de deuda que luego puede borrarse a través de la devaluación a costa de los ahorradores.

Queda por verse si el aspecto deflacionario de una moneda es un problema cuando no se debe a una rápida retracción económica, o una ventaja porque la protección contra la inflación y la devaluación supera con creces los riesgos de la deflación.

Consenso Descentralizado

En el capítulo anterior vimos la cadena de bloques, el libro contable (o lista) público y global de todas las transacciones, que es aceptado como un registro autoritario de propiedad por todos en la red bitcoin.

Pero, ¿cómo puede todo el mundo en la red estar de acuerdo sobre una única "verdad" universal, sobre quién es dueño de qué, sin tener que confiar en nadie? Todos los sistemas de pago tradicionales están basados en un modelo de confianza en el que una autoridad central proporciona un servicio de cámara de compensación, básicamente, verificando y compensando todas las transacciones. Bitcoin no tiene autoridad central, pero de alguna manera cada nodo completo tiene una copia completa de un libro de contabilidad público en el que se puede confiar como registro de autoridad. La cadena de bloques no está creada por una autoridad central, pero se monta de forma independiente por cada nodo de la red. De alguna manera, cada nodo de la red, que actúa sobre la información transmitida a través de conexiones de red inseguras, puede llegar a la misma conclusión y montar una copia del mismo libro de contabilidad público que los demás. En este capítulo se examina el proceso por el cual la red bitcoin logra un consenso global sin autoridad central.

La invención principal de Satoshi Nakamoto es el mecanismo descentralizado para el *consenso emergente*. Emergente, porque el consenso no se logra explícitamente: no hay elección o momento fijo en el que se produce el consenso. En cambio, el consenso es un artefacto emergente de la interacción asíncrona de miles de nodos independientes, todos siguiendo reglas simples. Todas las propiedades de bitcoin se derivan de esta invención, incluyendo moneda, transacciones, pagos, y el modelo de seguridad que no depende de una autoridad central o de la confianza, .

El consenso descentralizado de bitcoin emerge de la interacción de cuatro procesos que ocurren de forma independiente en los nodos de la red:

- La verificación independiente de cada transacción, por cada nodo completo, basado en una amplia lista de criterios

- La incorporación independiente de esas transacciones en nuevos bloques por nodos de minería, junto con la computación demostrada a través de un algoritmo de Prueba-de-Trabajo
- La verificación independiente de los nuevos bloques por cada nodo y el montaje en una cadena
- Selección independiente, por cada nodo, de la cadena con mayor poder computacional demostrado a través de Prueba-de-Trabajo.

En las próximas secciones examinaremos estos procesos y cómo interactúan para crear la propiedad emergente de consenso de toda la red que permite a cualquier nodo bitcoin montar su propia copia de autoridad, confiable, pública, del libro contable global.

Verificación Independiente de Transacciones

En Transacciones, vimos cómo el software de cartera crea transacciones mediante la recopilación de UTXO, proporcionando los scripts de desbloqueo apropiados, y construyendo después nuevas salidas asignadas a un nuevo propietario. La transacción resultante se envía entonces a los nodos vecinos en la red bitcoin de manera que se pueda propagar a través de toda la red bitcoin.

Sin embargo, antes de retransmitir las transacciones a sus vecinos, cada nodo bitcoin que recibe una transacción primero verifica la transacción. Esto garantiza que solo las transacciones válidas se propaguen a través de la red, mientras que las transacciones no válidas se descartan en el primer nodo que las encuentra.

Cada nodo verifica cada transacción a través de una larga lista de criterios:

- La sintaxis de la transacción y su estructura de datos deben ser correctos.
- Ni las listas de entradas ni de salidas estén vacías.
- El tamaño de la transacción en bytes es inferior a MAX_BLOCK_SIZE.
- Cada valor de salida, así como el total, deben estar dentro del rango permitido de valores (menos de 21m de monedas, más que el umbral de *polvo*).
- Ninguna de las entradas tiene de hash=0, N=-1 (transacciones coinbase no deben ser transmitidas).
- nLocktime es igual a INT_MAX, o los valores nLocktime y nSequence se satisfacen según MedianTimePast.
- El tamaño de la transacción en bytes es mayor o igual a 100.
- El número de operaciones de firma (SIGOPS) contenidas en la transacción es menor que el límite de operación de firma.
- El script de desbloqueo (scriptSig) solo puede empujar números en la pila, y el script de bloqueo (scriptPubkey) debe respetar los formatos isStandard (esto rechaza transacciones "no estándar").
- Debe existir una transacción coincidente en el pool, o en un bloque en la rama principal.
- Para cada entrada, si existe la salida de referencia en cualquier otra transacción del pool, la transacción debe ser rechazada.
- Para cada entrada, busca en la rama principal y en el pool de transacciones para encontrar la transacción de salida de referencia. Si la transacción de salida no se encuentra para cualquier entrada, esta será una transacción huérfana. Añadir al pool de transacciones huérfanas, si una transacción coincidente no está ya en el pool.
- Para cada entrada, si la transacción de salida de referencia es una salida coinbase, debe tener por lo menos COINBASE_MATURITY (100) confirmaciones.
- Para cada entrada, debe existir la salida de referencia y ya no puede ser gastada.
- Usando las transacciones de salida de referencia para obtener los valores de entrada, compruebe que cada valor de entrada, así como la suma, están en el rango permitido de valores (menos de 21m monedas, más de 0).
- Rechazar si la suma de los valores de entrada es inferior a la suma de los valores de salida.
- Rechazar si la comisión de transacción sería demasiado baja (minRelayTxFee) para entrar en un bloque vacío.
- Los scripts de desbloqueo para cada entrada se deben validar contra los scripts de bloqueo de salida correspondientes.

Estas condiciones pueden verse en detalle en las funciones AcceptToMemoryPool, CheckTransaction, y CheckInputs en el Bitcoin Core. Ten en cuenta que las condiciones cambian con el tiempo, para hacer frente a los nuevos tipos de ataques de

denegación de servicio o, a veces se relajan las normas a fin de incluir más tipos de transacciones.

Al verificar de forma independiente cada transacción, en el momento en que se recibe y antes de propagarlo, cada nodo construye un conjunto de transacciones válidas (pero sin confirmar) conocido como el *pool de transacciones*, *pool de memoria* o *mempool*.

Nodos de Minería

Algunos de los nodos de la red bitcoin son nodos especializados llamados *mineros*. En Introducción presentamos a Jing, un estudiante de ingeniería informática en Shanghai, China, que es un minero bitcoin. Jing gana bitcoin ejecutando una "plataforma de minería", que es un sistema informático de hardware especializado diseñado para minar bitcoin. El hardware especializado de minería de Jing se conecta a un servidor que ejecuta un nodo bitcoin completo. A diferencia de Jing, algunos mineros minan sin un nodo completo, como veremos en Pools de Minería. Como cualquier otro nodo completo, el nodo de Jing recibe y propaga transacciones sin confirmar en la red bitcoin. El nodo de Jing, sin embargo, también agrega estas transacciones en nuevos bloques.

El nodo de Jing está a la escucha de nuevos bloques, propagados en la red bitcoin, al igual que hacen todos los nodos. Sin embargo, la llegada de un nuevo bloque tiene un significado especial para un nodo de minería. La competencia entre los mineros termina efectivamente con la propagación de un nuevo bloque que actúa como un anuncio del ganador. Para los mineros, recibir un nuevo bloque válido significa que otra persona ganó la competición y que ellos perdieron. Sin embargo, el final de una ronda de la competición marca también el comienzo de la siguiente ronda. El nuevo bloque no es sólo una bandera a cuadros, que marca el final de la carrera; también es el pistoletazo de salida en la carrera por el siguiente bloque.

Agregando Transacciones en los Bloques

Después de validar las transacciones, un nodo bitcoin las añadirá al *tanque de memoria* o *pool de transacciones*, donde las transacciones esperan hasta que puedan ser incluidas (minadas) en un bloque. El nodo de Jing recoge, valida y transmite nuevas transacciones como cualquier otro nodo. Sin embargo, a diferencia de otros nodos, el nodo de Jing agregará estas transacciones en un *bloque candidato* .

Sigamos los bloques que se crearon durante el tiempo en que Alice compró una taza de café de Bob's Cafe (ver Comprando una Taza de Café). La transacción de Alice se incluyó en el bloque 277.316. Con el fin de demostrar los conceptos de este capítulo, vamos a suponer que el bloque fue minado por el sistema de minería de Jing y seguiremos la transacción de Alice, hasta que pasa a formar parte de este nuevo bloque.

El nodo de minería de Jing mantiene una copia local de la cadena de bloques. Para cuando Alice compra la taza de café, el nodo de Jing ha montado una cadena hasta el bloque 277.314. El nodo de Jing está escuchando transacciones, intentando extraer un nuevo bloque y también escuchando los bloques descubiertos por otros nodos. Mientras el nodo de minería Jing está minando, recibe el bloque 277.315 a través de la red bitcoin. La llegada de este bloque significa el final de la competición para el bloque 277.315 y el comienzo de la competición para crear el bloque 277.316.

Durante los 10 minutos anteriores, mientras que el nodo de Jing estaba buscando una solución para el bloque 277.315, estaba al mismo tiempo recogiendo las transacciones en preparación para el siguiente bloque. Para entonces habrá recogido unos pocos cientos de transacciones en el tanque de memoria. Al recibir el bloque 277.315 y validarlo, el nodo de Jing también comprobará todas las transacciones en el tanque de memoria y retirará las que se hayan incluido en el bloque 277.315. Las transacciones que aún permanezcan en el tanque de memoria seguirán sin confirmar y estarán esperando a ser registradas en un nuevo bloque.

El nodo de Jing construye de inmediato un nuevo bloque vacío, un candidato para el bloque 277.316. Este bloque se denomina *bloque candidato* porque aún no es un bloque válido, ya que no contiene una Prueba-de-Trabajo válida. El bloque se vuelve válido sólo si el minero tiene éxito en la búsqueda de una solución para el algoritmo de Prueba-de-Trabajo.

Cuando el nodo de Jing agrega todas las transacciones del tanque de memoria, el nuevo bloque candidato tiene 418 transacciones con comisiones de transacción totales de 0.09094928 bitcoin. Puedes ver este bloque en la cadena de bloques utilizando la interfaz de línea de comandos del cliente principal de bitcoin, como se muestra en Usando la línea de comandos para obtener el bloque 277.316.

Example 24. Usando la línea de comandos para obtener el bloque 277.316

```
$ bitcoin-cli getblockhash 277316

000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

$ bitcoin-cli getblock 000000000000000001b6b9a13b095e96db41c4a928b97ef2d9\
44a9b31b2cc7bdc4


{
    "hash" : "000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35561,
    "size" : 218629,
    "height" : 277316,
    "version" : 2,
    "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
    "tx" : [
        "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
        "B268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",

        ... 417 more transactions ...

       ],
    "time" : 1388185914,
    "nonce" : 924591752,
    "bits" : "1903a30c",
    "difficulty" : 1180923195.25802612,
    "chainwork" : "0000000000000000000000000000000000000000000000934695e92aaf53afa1a",
    "previousblockhash" : "000000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569"
}
```

La Transacción Coinbase

La primera transacción en cualquier bloque es una transacción especial, llamada *transacción coinbase*. Esta transacción es construida por el nodo de Jing y contiene su *recompensa* por el esfuerzo de minería.

NOTE

Cuando el bloque 277,316 fue minado, la recompensa fue de 25 bitcoins por bloque. Desde entonces, han transcurrido dos períodos de "reducción a la mitad". La recompensa por bloque se redujo a 12.5 bitcoins en julio de 2016 y a 6.25 bitcoins en mayo de 2020.

El nodo de Jing crea la transacción coinbase como un pago a su propia cartera: "Páguese a la dirección de Jing 25,09094928 bitcoins." Es la cantidad total de recompensa que Jing recolecta por la minería de un bloque y es la suma de la recompensa coinbase (25 bitcoins nuevos) mas las comisiones de transacción (0,09094928) de todas las transacciones incluidas en el bloque como se muestra en Transacción de Coinbase.

Example 25. Transacción de Coinbase

```
$ bitcoin-cli getrawtransaction d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1


{
    "hex" :
"01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff0f03443b0403858402062f503253482f
ffffffff0110c08d9500000000232102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",
    "txid" : "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
    "version" : 1,
    "locktime" : 0,
    "vin" : [
        {
            "coinbase" : "03443b0403858402062f503253482f",
            "sequence" : 4294967295
        }
    ],
    "vout" : [
        {
            "value" : 25.09094928,
            "n" : 0,
            "scriptPubKey" : {
                "asm" : "02aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21OP_CHECKSIG",
                "hex" : "2102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac",
                "reqSigs" : 1,
                "type" : "pubkey",
                "addresses" : [
```

```
                    "1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N"
                ]
            }
        }
    ]
}
```

A diferencia de las transacciones regulares, la transacción coinbase no consume (no gasta) ninguna UTXO como entrada. En cambio, solo tiene una entrada, llamada *coinbase*, que crea bitcoins de la nada. La transacción coinbase tiene una salida, pagadera a la propia dirección bitcoin del minero. La salida de la transacción coinbase envía el valor de 25,09094928 bitcoins a la dirección bitcoin del minero; en este caso es 1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N.

Recompensa de Coinbase y Comisiones

Para construir la transacción coinbase, el nodo de Jing calcula primero el monto total de las comisiones de transacción al agregar todas las entradas y salidas de las 418 transacciones que se agregaron al bloque. Las comisiones se calculan como:

```
Total Comisiones = Suma(Entradas) - Suma(Salidas)
```

En el bloque 277.316, el total de las comisiones de las transacciones es de 0,09094928 bitcoin.

A continuación, el nodo de Jing calcula la recompensa correcta para el nuevo bloque. La recompensa se calcula en función de la altura de bloque, comenzando en 50 bitcoins por bloque y se reduce a la mitad cada 210.000 bloques. Debido a que este bloque tiene una altura de 277.316, la recompensa correcta es de 25 bitcoins.

El cálculo se puede ver en la función GetBlockSubsidy del Cliente Principal de Bitcoin, como se muestra en Calculando la recompensa de bloque—Función GetBlockSubsidy, Cliente Principal de Bitcoin, main.cpp.

Example 26. Calculando la recompensa de bloque—Función GetBlockSubsidy, Cliente Principal de Bitcoin, main.cpp

```
CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)
{
    int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;
    // Forzar la recompensa de bloque a cero cuando el desplazamiento a la derecha no está definido.
    if (halvings >= 64)
        return 0;

    CAmount nSubsidy = 50 * COIN;
    // La recompensa se reduce a la mitad cada 210.000 bloques que se producirán aproximadamente cada 4 años.
    nSubsidy >>= halvings;
    return nSubsidy;
}
```

La recompensa inicial se calcula en satoshis multiplicando 50 por la constante COIN (100,000.000 satoshis). Esto establece la recompensa inicial (nSubsidy) a 5 mil millones de satoshis.

A continuación, la función calcula el número de halvings que se han producido al dividir la altura de bloque actual por el intervalo de mitades (SubsidyHalvingInterval). En el caso del bloque 277.316, con un intervalo de reducción a la mitad cada 210.000 bloques, el resultado es 1 reducción a la mitad o halving.

El número máximo de reducciones a la mitad permitido es 64, por lo que el código impone una recompensa cero (solo devuelve las comisiones) si se exceden las 64 reducciones a la mitad.

A continuación, la función usa el operador binario de desplazamiento a la derecha para dividir la recompensa (nSubsidy) entre dos, para cada ronda de reducción a la mitad. En el caso del bloque 277,316, esto cambiaría binariamente a la derecha la recompensa de 5 mil millones de satoshis una vez (una mitad) y daría como resultado 2.5 millardos de satoshis, o 25 bitcoins. El operador binario de desplazamiento a la derecha se usa porque es más eficiente que múltiples divisiones repetidas. Para evitar un posible error, la operación de cambio se omite después de 63 reducciones a la mitad, y la recompensa se establece en 0.

Finalmente, se añade la recompensa coinbase (nSubsidy) a las comisiones de transacción (nFees), y se devuelve la suma.

TIP

Si el nodo de minería de Jing redacta la transacción coinbase, ¿qué le impide a Jing "recompensarse" a sí

> mismo con 100 o 1000 bitcoins? La respuesta es que una recompensa incorrecta daría como resultado que el bloque sea considerado inválido por todos los demás nodos, desperdiciando la electricidad que Jing utilizó para la Prueba-de-Trabajo. Jing solo podrá cobrar su recompensa si su bloque es aceptado por todos los demás nodos.

Estructura de una Transacción de Coinbase

Con estos cálculos, el nodo de Jing construye la transacción coinbase para pagarse 25,09094928 bitcoins.

Como puede verse en Transacción de Coinbase, la transacción coinbase tiene un formato especial. En lugar de una entrada de transacción que especifica una UTXO previa a ser gastada, ésta posee una entrada "coinbase". Ya se examinaron las entradas de transacciones en Serialización de entradas de transacción. Comparemos una entrada de transacción regular con una entrada de transacción coinbase. La estructura de una entrada de transacción "normal" muestra la estructura de una transacción regular, mientras que La estructura de una entrada de transacción coinbase muestra la estructura de la entrada de una transacción coinbase.

Table 25. La estructura de una entrada de transacción "normal"

Tamaño	Campo	Descripción
32 bytes	Hash de Transacción	Puntero a la transacción que contiene la UTXO a ser gastada
4 bytes	Índice de Salida	El número de índice de la UTXO que se pasó, primera es 0
1–9 bytes (VarInt)	Tamaño Script-de-Desbloqueo	Longitud del Script-de-Desbloqueo en bytes, a continuación
Variable	Script-de-Desbloqueo	Un script que cumple las condiciones del script de bloqueo del UTXO
4 bytes	Número de secuencia	Generalmente se establece en 0xFFFFFFFF para optar por no participar en BIP 125 y BIP 68

Table 26. La estructura de una entrada de transacción coinbase

Tamaño	Campo	Descripción
32 bytes	Hash Transacción	Todos los bits son cero: No es una referencia de hash transacción
4 bytes	Índice de salida	Todos los bits son requeridos: 0xFFFFFFFF
1–9 bytes (VarInt)	Tamaño de Data Coinbase	Extensión de la data coinbase, desde 2 a 100 bytes
Variable	Data Coinbase	Data extra arbitraria utilizada para el nonce y para etiquetas de minería. En bloques de v2; se debe comenzar con la altura de bloque.
4 bytes	Número de secuencia	Ajuste a 0xFFFFFFFF

En una transacción coinbase, los valores de los dos primeros campos se establecen de tal manera que no representan una referencia ninguna UTXO. En lugar de un "hash de transacción", el primer campo se llena con 32 bytes, todos con números cero. El "índice de salida" se llena con 4 bytes, todos configurados en 0xFF (o en números 255 en notación decimal). El "script de desbloqueo" (scriptSig) se reemplaza por la data de coinbase, un campo de datos utilizado por los mineros, como veremos a continuación.

Datos Coinbase

Las transacciones coinbase no tienen un campo de script de desbloqueo (también conocido como scriptSig). En cambio,

este campo se reemplaza por data de coinbase, que debe tener entre 2 y 100 bytes. Excepto por los pocos primeros bytes, los mineros pueden usar el resto de la data de coinbase de la forma que quieran; son datos arbitrarios.

En el bloque génesis, por ejemplo, Satoshi Nakamoto agregó el texto: "The Times 03/Jan/2009 Cancillería al borde del segundo rescate para los bancos", en el campo de la data coinbase, utilizándola como prueba de la fecha y para transmitir un mensaje. Actualmente, los mineros usan la data coinbase para incluir valores nonce extra y cadenas adicionales que identifican al grupo de minería.

Los primeros pocos bytes de coinbase solían ser arbitrarios, pero ese ya no es el caso. Según el BIP-34, los bloques de la versión-2 (bloques con el campo de versión establecido en 2) deben contener el índice de la altura de bloque como un script u operación de "inserción" al comienzo del campo de coinbase.

En el bloque 277.316 se observa en la data de coinbase (véase Transacción de Coinbase), que corresponde al script de desbloqueo o campo scriptSig de la entrada de la transacción, con el valor hexadecimal 03443b0403858402062f503253482f. A continuación desciframos este valor.

El primer byte, 03, le ordena a la máquina de ejecución de los script que empuje los siguientes tres bytes en la pila de scripts (véase Empujar valor a la pila). Los siguientes tres bytes, 0x443b04, son la altura de bloque codificada en formato “little-endian” (en reversa, se coloca de primero al byte menos significativo). Inviértase el orden de los bytes y el resultado es 0x043b44, que es 277.316 en formato decimal.

Los siguientes dígitos hexadecimales (0385840206) se utilizan para codificar un *nonce* extra (véase La Solución del Nonce Extra), o un valor aleatorio, que se utiliza para encontrar una solución apropiada al desafío de la Prueba-de-Trabajo.

La parte final de los datos de coinbase (2f503253482f) es la cadena de caracteres en código ASCII serializada `/P2SH/`, que indica que el nodo de minería que confirmó este bloque admite la mejora “P2SH” definida en la BIP-16. La introducción de la capacidad P2SH requirió de la señalización de los mineros para respaldar a la BIP-16, o bien a la BIP-17. Aquellos que respaldaban la implementación de BIP-16 debían incluir /P2SH/ en la data de su coinbase. Aquellos que respaldaban la implementación BIP-17 de P2SH debían incluir la cadena p2sh/CHV en la data de su coinbase. El BIP-16 fue elegido como el ganador, y muchos mineros continuaron incluyendo la cadena /P2SH/ en su coinbase para indicar el apoyo a esta función.

Extraer los datos coinbase del bloque génesis utiliza la biblioteca libbitcoin introducida en Clientes Alternativos, Bibliotecas y Kits de Herramientas para extraer los datos coinbase del bloque génesis que muestran el mensaje de Satoshi. Tenga en cuenta que la biblioteca libbitcoin contiene una copia estática del bloque génesis, por lo que el código de ejemplo puede recuperar el bloque génesis directamente desde la biblioteca.

Example 27. Extraer los datos coinbase del bloque génesis

```
/*
  Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::chain::block block = bc::chain::block::genesis_mainnet();
    // Genesis block contains a single coinbase transaction.
    assert(block.transactions().size() == 1);
    // Get first transaction in block (coinbase).
    const bc::chain::transaction& coinbase_tx = block.transactions()[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs().size() == 1);
    const bc::chain::input& coinbase_input = coinbase_tx.inputs()[0];
    // Convert the input script to its raw format.
    const auto prefix = false;
    const bc::data_chunk& raw_message = coinbase_input.script().to_data(prefix);
    // Convert this to a std::string.
    std::string message(raw_message.begin(), raw_message.end());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}
```

Compilamos el código con el compilador GNU C ++ y ejecutamos el ejecutable resultante, como se muestra en Compilando y ejecutando el código de ejemplo satoshi-words.

Example 28. Compilando y ejecutando el código de ejemplo satoshi-words

```
$ # Compilando el código
$  g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Ejecutar el ejecutable
$ ./satoshi-words
^D��<GS>^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks
```

Construyendo la Cabecera de Bloque

Para construir la cabecera de bloque, el nodo de minería necesita completar seis campos, como se enumera en [block_header_structure_ch10].

Table 27. La estructura de la cabecera de bloque

Tamaño	Campo	Descripción
4 bytes	Versión	Un número de versión para seguir las actualizaciones de software y protocolo
32 bytes	Hash del Bloque Anterior	Una referencia al hash del bloque anterior (padre) en la cadena
32 bytes	Raíz Merkle	Un hash de la raíz del árbol merkle de las transacciones de este bloque
4 bytes	Sello de Tiempo	El tiempo de creación aproximada de este bloque (segundos desde Unix Epoch)
4 bytes	Objetivo	El objetivo del algoritmo de Prueba-de-Trabajo para este bloque
4 bytes	Nonce	Un contador usado para el algoritmo de Prueba-de-Trabajo

En el momento que el bloque 277.316 fue minado, el número de versión que describe la estructura de bloques es la versión 2, que está codificada en formato little-endian en 4 bytes como 0x02000000.

A continuación, el nodo de minería debe agregar el "Hash de Bloque Previo" (también conocido como prevhash). Este es el hash de la cabecera de bloque, del bloque 277.315; el bloque previo recibido de la red, que el nodo de Jing ha aceptado y seleccionado como el *padre* del bloque candidato 277.316. El hash de la cabecera de bloque para el bloque 277.315 es:

```
000000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

> **TIP**
>
> Al seleccionar el bloque *padre* específico, indicado por el campo Hash de Bloque Previo en la cabecera de bloque del bloque candidato, Jing está comprometiendo su poder de minería en el trabajo de extender la cadena que termina en ese bloque específico. En esencia, así es como Jing "vota" con su poder de minería, por la cadena válida de mayor dificultad.

El siguiente paso es resumir todas las transacciones en un árbol de merkle, con el fin de agregar la raíz de merkle a la cabecera de bloque. La transacción coinbase aparece como la primera transacción en el bloque. Luego, se agregan 418 transacciones más, para un total de 419 transacciones en el bloque. Como vimos en los Árboles Merkle, debe tenerse un número par de nodos tipo "hoja" en el árbol, por lo que la última transacción debe duplicarse, creando 420 nodos, cada uno con el hash de una transacción. Los valores hash de las transacciones se combinan en pares, creando cada nivel del árbol, hasta que todas las transacciones se resumen en un nodo en la "raíz" del árbol. La raíz del árbol de merkle resume todas las transacciones en un solo valor de 32 bytes, que puede verse listada como la "raíz de merkle" en Usando la línea de comandos para obtener el bloque 277.316, y aquí:

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

El nodo de minería de Jing agregará un sello de tiempo de 4 bytes, que estará codificado como un sello de tiempo del tipo "epoch" de Unix, que se basa en la cantidad de segundos transcurridos desde la medianoche según la hora UTC, del jueves 1° de enero de 1970. El tiempo 1388185914 es igual al viernes, 27 de diciembre de 2013, a las 23:11:54, hora UTC.

El nodo de Jing a continuación rellena el objetivo que define la Prueba-de-Trabajo requerida para hacer de este un bloque válido. El objetivo se almacena en el bloque como una métrica de "objetivo en bits", que es una codificación de exponente de mantisa del objetivo. La codificación tiene un exponente de 1 byte, seguido de una mantisa (coeficiente) de 3 bytes. En el bloque 277.316, por ejemplo, el valor del objetivo en bits es de 0x1903a30c. La primera parte 0x19 es un exponente hexadecimal, mientras que la siguiente parte, 0x03a30c, es el coeficiente. El concepto de un objetivo se explica en [target] y la representación de "objetivo en bits" se explica en [target_bits].

El campo final es el nonce, que se inicializa a cero.

Con todos los demás campos llenos, la cabecera de bloque ahora está completa y el proceso de minería puede comenzar. El objetivo ahora es encontrar un valor para el nonce que dé como resultado un hash de cabecera de bloque que sea menor que el objetivo. El nodo de minería necesitará probar miles de millones o billones de valores nonce antes de encontrar un nonce que satisfaga el requisito.

Minando el Bloque

Ahora que un bloque candidato ha sido ensamblado por el nodo de Jing, es hora de que la plataforma hardware de minería de Jing "mine" el bloque, para encontrar una solución al algoritmo de la Prueba-de-Trabajo que haga que el bloque sea válido. A lo largo de este libro, hemos estudiado las funciones de hash criptográfico que se utilizan en varios aspectos del sistema bitcoin. La función hash SHA256 es la función utilizada en el proceso de minería de bitcoin.

En los términos más simples, la minería es el proceso de obtener el valor hash de la cabecera de bloque repetidamente, cambiando un parámetro, hasta que el hash resultante coincida con un objetivo específico. El resultado de la función hash no se puede determinar de antemano, ni se puede crear un patrón que produzca un valor hash específico. Esta característica de las funciones hash significa que la única forma de producir un resultado hash que coincida con un objetivo específico es intentar una y otra vez, modificando aleatoriamente la entrada hasta que el resultado del hash deseado aparezca por casualidad.

Algoritmo de Prueba-de-Trabajo

Un algoritmo hash toma una entrada de datos de longitud arbitraria y produce un resultado con una longitud de datos determinísticamente prefijada, es decir, una huella digital de la data de entrada. Para cualquier entrada de datos específica, el valor hash resultante siempre será el mismo y puede ser fácilmente calculado y verificado por cualquier persona que implemente el mismo algoritmo hash.La característica esencial de un algoritmo hash criptográfico es que no es computacionalmente factible encontrar dos entradas diferentes que produzcan la misma huella digital (lo que se conoce como *colisión*). Como corolario, también es prácticamente imposible seleccionar una entrada de tal manera que produzca la huella digital deseada, mas allá de intentar ingresos aleatorios.

Con SHA256, la salida es siempre de 256 bits de longitud, independientemente del tamaño de la entrada. En [sha256_example1], vamos a utilizar el intérprete de Python para calcular el hash SHA256 de la frase: "I am Satoshi Nakamoto."

Ejemplo .SHA256

```
$ python


Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

[sha256_example1] muestra el resultado de calcular el hash de "I am Satoshi Nakamoto" 5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e. Este número de 256 bits es el *hash* o *digest* de

la frase y depende de todos y cada uno de los elementos de frase. Si se añade una sola letra, signo de puntuacion, o cualquier otro carácter, producirá un hash diferente.

Por tanto, si cambiamos la frase, deberíamos ver hashes completamente diferentes. Vamos a comprobarlo añadiendo un número al final de nuestra frase, usando el script simple de Python en Comando script SHA256 para generar muchos hashes iterando con un nonce.

Example 29. Comando script SHA256 para generar muchos hashes iterando con un nonce

```
# example of iterating a nonce in a hashing algorithm's input

from __future__ import print_function
import hashlib

text = "I am Satoshi Nakamoto"

# iterate nonce from 0 to 19
for nonce in range(20):

    # add the nonce to the end of the text
    input_data = text + str(nonce)

    # calculate the SHA-256 hash of the input (text+nonce)
    hash_data = hashlib.sha256(input_data.encode()).hexdigest()

    # show the input and hash result
    print(input_data, '=>', hash_data)
```

La ejecución de este script producirá los hashes de varias frases. Se ha añadido un número al final de las frases para hacerlas diferentes entre sí. Incrementando el número, podemos obtener diferentes hashes, como se muestra en Salida SHA256 de un script para generar muchos hashes iterando un nonce.

Example 30. Salida SHA256 de un script para generar muchos hashes iterando un nonce

```
$ python hash_example.py


I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

Cada frase produce un hash resultante completamente diferente. Parecen completamente al azar, pero si replica los valores exactos de este ejemplo en cualquier computadora con Python, obtendrá exactamente los mismos hashes.

El número que se ha utilizado como variable en este escenario se llama *nonce*. El nonce se utiliza para variar la salida de una función criptográfica, en este caso para variar la huella digital SHA256 de la frase.

Para plantear un desafío con este algoritmo, se establecerá un objetivo: encontrar una frase que produzca un hash hexadecimal que comience con un cero. Afortunadamente, esto no es difícil! Salida SHA256 de un script para generar muchos hashes iterando un nonce muestra que la frase "Yo soy Satoshi Nakamoto13" produce el hash 0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba9531041a5, el cual se ajusta a nuestros criterios. Se

necesitaron 13 intentos para encontrarlo. En términos de probabilidades, si la salida de la función hash se distribuye uniformemente, esperaríamos encontrar un resultado con un 0 como prefijo hexadecimal una vez cada 16 hashes (uno de cada 16 dígitos hexadecimales desde el 0 hasta el F). En términos numéricos, eso significa encontrar un valor hash que sea menor que 0x1000000000000000000000000000000000000000000000000000000000000000. Llamamos a este umbral el objetivo y el objetivo es encontrar un hash que sea numéricamente menor que el objetivo. Si disminuimos el objetivo, la tarea de encontrar un hash que sea menor que el objetivo se vuelve cada vez más difícil.

Para dar una analogía simple, imagínese un juego donde los jugadores lanzan un par de dados repetidamente, tratando de lanzar menos de un objetivo específico. En la primera ronda, el objetivo es 12. A menos que arrojes doble seis, ganas. En la siguiente ronda, el objetivo es 11. Los jugadores deben lanzar 10 o menos para ganar, nuevamente una tarea fácil. Digamos que unas rondas más tarde el objetivo se ha reducido a 5. Ahora, más de la mitad de los lanzamientos de dados excederán el objetivo y, por lo tanto, no serán válidos. Se necesitan exponencialmente más lanzamientos de dados para ganar, cuanto más bajo sea el objetivo. Finalmente, cuando el objetivo es 2 (el mínimo posible), solo un lanzamiento de cada 36, o el 2% de ellos, producirá un resultado ganador.

Desde la perspectiva de un observador que sabe que el objetivo del juego de dados es 2, si alguien ha logrado lanzar un lanzamiento ganador, se puede suponer que intentó, en promedio, 36 lanzamientos. En otras palabras, uno puede estimar la cantidad de trabajo que se necesita para tener éxito a partir de la dificultad impuesta por el objetivo. Cuando el algoritmo se basa en una función determinista como SHA256, la entrada en sí constituye una prueba de que se realizó una cierta cantidad de trabajo para producir un resultado por debajo del objetivo. De allí el concepto de *Prueba-de-Trabajo*.

TIP

Aunque cada intento produce un resultado aleatorio, la probabilidad de cualquier resultado posible se puede calcular por adelantado. Por lo tanto, un resultado de dificultad específica constituye la prueba de una cantidad específica de trabajo.

En Salida SHA256 de un script para generar muchos hashes iterando un nonce, el "nonce" ganador es 13 y este resultado puede ser confirmado por cualquiera independientemente. Cualquiera puede agregar el número 13 como sufijo a la frase "Yo soy Satoshi Nakamoto" y calcular el hash, verificando que sea menor que el objetivo. El resultado exitoso es también Prueba-de-Trabajo, porque demuestra que hicimos el trabajo para encontrar ese nonce. Si bien solo se necesita un cómputo hash para realizar la verificación, se necesitaron 13 cómputos hash para encontrar un nonce que funcionara. Si tuviéramos un objetivo más bajo (mayor dificultad), se necesitarían muchos más cómputos hash para encontrar un nonce adecuado, pero solo un cómputo hash para que cualquiera pueda verificarlo. Además, al conocer el objetivo, cualquiera puede estimar la dificultad usando estadísticas y, por lo tanto, saber cuánto trabajo se necesitó para encontrar el susodicho nonce.

TIP

La Prueba-de-Trabajo debe producir un hash que sea *menor que* el objetivo. Un objetivo más alto significa que es menos difícil encontrar un hash que esté debajo del objetivo. Un objetivo más bajo significa que es más difícil encontrar un hash debajo del objetivo. El objetivo y la dificultad están inversamente relacionados.

La Prueba-de-Trabajo de bitcoin es muy similar al desafío que se muestra en Salida SHA256 de un script para generar muchos hashes iterando un nonce. El minero ensambla un bloque candidato lleno de transacciones. Luego, el minero calcula el hash de la cabecera de este bloque y ve si es más pequeño que el *objetivo* actual. Si el hash no es menor que el objetivo, el minero modificará el nonce (por lo general, solo lo incrementará en uno) e intentará nuevamente. En la dificultad actual en la red bitcoin, los mineros tienen que probar miles de billones de veces antes de encontrar un nonce que resulte en un hash de cabecera de bloque lo suficientemente bajo.

Se implementa un algoritmo de prueba de trabajo muy simplificado en Python en Implementación simplificada de prueba de trabajo.

Example 31. Implementación simplificada de prueba de trabajo

```
#!/usr/bin/env python
# example of proof-of-work algorithm

import hashlib
import time

try:
    long        # Python 2
```

```
    xrange
except NameError:
    long = int  # Python 3
    xrange = range

max_nonce = 2 ** 32  # 4 billion


def proof_of_work(header, difficulty_bits):
    # calculate the difficulty target
    target = 2 ** (256 - difficulty_bits)

    for nonce in xrange(max_nonce):
        hash_result = hashlib.sha256((str(header) + str(nonce)).encode()).hexdigest()

        # check if this is a valid result, below the target
        if long(hash_result, 16) < target:
            print("Success with nonce %d" % nonce)
            print("Hash is %s" % hash_result)
            return (hash_result, nonce)

    print("Failed after %d (max_nonce) tries" % nonce)
    return nonce


if __name__ == '__main__':
    nonce = 0
    hash_result = ''

    # difficulty from 0 to 31 bits
    for difficulty_bits in xrange(32):
        difficulty = 2 ** difficulty_bits
        print("Difficulty: %ld (%d bits)" % (difficulty, difficulty_bits))
        print("Starting search...")

        # checkpoint the current time
        start_time = time.time()

        # make a new block which includes the hash from the previous block
        # we fake a block of transactions - just a string
        new_block = 'test block with transactions' + hash_result

        # find a valid nonce for the new block
        (hash_result, nonce) = proof_of_work(new_block, difficulty_bits)

        # checkpoint how long it took to find a result
        end_time = time.time()

        elapsed_time = end_time - start_time
        print("Elapsed Time: %.4f seconds" % elapsed_time)

        if elapsed_time > 0:

            # estimate the hashes per second
            hash_power = float(long(nonce) / elapsed_time)
            print("Hashing Power: %ld hashes per second" % hash_power)
```

Mediante la ejecución de este código, puede establecer la dificultad que desee (en bits, cuántos de los primeros bits deben ser cero) y ver cuánto tiempo se necesita para encontrar una solución en su equipo. En Ejecutar el ejemplo de prueba de trabajo para diversas dificultades, se puede ver cómo funciona en un ordenador portátil normal.

Example 32. Ejecutar el ejemplo de prueba de trabajo para diversas dificultades

```
$ python proof-of-work-example.py*


Difficulty: 1 (0 bits)

[...]

Difficulty: 8 (3 bits)
Starting search...
Success with nonce 9
Hash is 1c1c105e65b47142f028a8f93ddf3dabb9260491bc64474738133ce5256cb3c1
Elapsed Time: 0.0004 seconds
Hashing Power: 25065 hashes per second
Difficulty: 16 (4 bits)
```

```
Starting search...
Success with nonce 25
Hash is 0f7becfd3bcd1a82e06663c97176add89e7cae0268de46f94e7e11bc3863e148
Elapsed Time: 0.0005 seconds
Hashing Power: 52507 hashes per second
Difficulty: 32 (5 bits)
Starting search...
Success with nonce 36
Hash is 029ae6e5004302a120630adcbb808452346ab1cf0b94c5189ba8bac1d47e7903
Elapsed Time: 0.0006 seconds
Hashing Power: 58164 hashes per second

[...]

Difficulty: 4194304 (22 bits)
Starting search...
Success with nonce 1759164
Hash is 0000008bb8f0e731f0496b8e530da984e85fb3cd2bd81882fe8ba3610b6cefc3
Elapsed Time: 13.3201 seconds
Hashing Power: 132068 hashes per second
Difficulty: 8388608 (23 bits)
Starting search...
Success with nonce 14214729
Hash is 000001408cf12dbd20fcba6372a223e098d58786c6ff93488a9f74f5df4df0a3
Elapsed Time: 110.1507 seconds
Hashing Power: 129048 hashes per second
Difficulty: 16777216 (24 bits)
Starting search...
Success with nonce 24586379
Hash is 0000002c3d6b370fccd699708d1b7cb4a94388595171366b944d68b2acce8b95
Elapsed Time: 195.2991 seconds
Hashing Power: 125890 hashes per second

[...]

Difficulty: 67108864 (26 bits)
Starting search...
Success with nonce 84561291
Hash is 0000001f0ea21e676b6dde5ad429b9d131a9f2b000802ab2f169cbca22b1e21a
Elapsed Time: 665.0949 seconds
Hashing Power: 127141 hashes per second
```

Como puede verse, aumentar la dificultad en 1 bit provoca una duplicación en el tiempo que lleva encontrar una solución. Si se piensa en el espacio numérico completo de 256 bits, cada vez que se restringe a cero un bit adicional, se disminuye el espacio de búsqueda a la mitad. En Ejecutar el ejemplo de prueba de trabajo para diversas dificultades, se necesitan 84 millones de intentos de hash para encontrar un nonce que produzca un hash con 26 bits iniciales como cero. Incluso a una velocidad de más de 120.000 hashes por segundo, todavía se requieren 10 minutos en una computadora portátil para encontrar esta solución.

Al momento de escribir este libro, la red está intentando encontrar un bloque cuyo hash de cabecera sea menor que:

```
00000000000000000029AB9000000000000000000000000000000000000000
```

Como puede verse, hay muchos ceros al comienzo de ese objetivo, lo que significa que el rango aceptable de hashes es mucho más pequeño, por lo que es más difícil encontrar un hash válido. Se necesitarán en promedio más de 1.8 zeta-hashes (un zeta-hash es mil millardos de millardos de hashes) para que la red descubra el siguiente bloque. Parece una tarea imposible, pero afortunadamente la red está aportando 3 exa-hashes por segundo (EH/seg) de potencia de procesamiento, que podrá encontrar un bloque en unos 10 minutos en promedio.

Representación del objetivo

En Usando la línea de comandos para obtener el bloque 277.316, se observó que el bloque contiene el objetivo, en una notación llamada "objetivo en bits" o simplemente "bits", que en el bloque 277.316 tiene el valor de 0x1903a30c. Esta notación expresa el objetivo de la Prueba-de-Trabajo bajo un formato de coeficiente/exponente, con los dos primeros dígitos hexadecimales para el exponente y los siguientes seis dígitos hexadecimales para el coeficiente. En este bloque, por lo tanto, el exponente es 0x19 y el coeficiente es 0x03a30c.

La fórmula para calcular el objetivo de dificultad a partir de esta representación es:

- objetivo = coeficiente * $2^{(8 * (\text{exponente} - 3))}$

Usando esa fórmula, y el valor de los bits de dificultad 0x1903a30c, obtenemos:

- objetivo = 0x03a30c * 2 $^{0x08 * (0x19-0x03)}$
- => objetivo = 0x03a30c * 2 $^{(0x08 * 0x16)}$
- => objetivo = 0x03a30c * 2 ^{0xB0}

que en decimal es:

- => objetivo = 238,348 * 2 ^{176}
- => objetivo = 22.829.202.948.393.929.850.749.706.076.701.368.331.072.452.018.388.575.715.328

cambiando de nuevo a hexadecimal:

- => objetivo = 0x0000000000000003A30C0000000000000000000000000000000000000000000

Esto significa que un bloque válido para la altura 277.316 es uno que tiene un hash de cabecera de bloque que es menor que el objetivo. En binario, ese número debe tener más de 60 bits iniciales establecidos en cero. Con este nivel de dificultad, un solo minero que procesa 1 billón de hashes por segundo (1 terahash por segundo o 1 TH/seg) solo encontraría una solución una vez cada 8.496 bloques o una vez cada 59 días, en promedio.

Re-calibrando para ajustar la dificultad

Como se ha visto, el objetivo determina la dificultad y, por lo tanto, afecta el tiempo que lleva encontrar una solución al algoritmo de la Prueba-de-Trabajo. Esto lleva a las preguntas obvias: ¿Por qué es ajustable la dificultad, quién la ajusta y cómo?

Los bloques de Bitcoin se generan cada 10 minutos, en promedio. Este es el latido del corazón de bitcoin y apuntala la frecuencia de emisión de divisas y la velocidad de confirmación de las transacciones. Tiene que permanecer constante no solo a corto plazo, sino durante un período de muchas décadas. Durante este tiempo, se espera que la potencia de computador continúe aumentando a un ritmo rápido. Además, el número de participantes en la minería y de computadoras involucradas, también cambiarán constantemente. Para mantener el tiempo de generación del bloque en 10 minutos, la dificultad de la minería debe ajustarse para tener en cuenta estos cambios. De hecho, el objetivo de la Prueba-de-Trabajo es un parámetro dinámico que se ajusta periódicamente para mantener el objetivo del intervalo de emisión de bloques en 10 minutos. En términos simples, el objetivo se establece de modo que la potencia de minería actual dé como resultado un intervalo de emisión de bloques en 10 minutos.

¿Cómo es que entonces, semejante ajuste puede llevarse a cabo en una red completamente descentralizada? La re-calibración ocurre automáticamente y en cada nodo de forma independiente. Cada 2.016 bloques, todos los nodos reajustan la Prueba-de-Trabajo. La ecuación de re-calibración mide el tiempo que llevó encontrar los últimos 2.016 bloques anteriores y lo compara con el tiempo esperado de 20.160 minutos (el número de 2.016 bloques multiplicado por el intervalo de bloque de 10 minutos deseado). Se calcula la relación entre el intervalo de tiempo real y el intervalo de tiempo deseado y se realiza un ajuste proporcional (hacia arriba o hacia abajo) del objetivo. En términos simples: si la red encuentra bloques más rápido que cada 10 minutos, la dificultad aumenta (el objetivo disminuye). Si el descubrimiento de bloques es más lento de lo esperado, la dificultad disminuye (el objetivo aumenta).

La ecuación se puede resumir como:

```
Nuevo objetivo = Viejo objetivo * (Tiempo real de los últimos 2016 bloques / 20160 minutos)
```

[retarget_code] muestra el código utilizado en el Cliente Principal de Bitcoin.

Example 33. Recálculo de prueba de trabajo—CalculateNextWorkRequired() in pow.cpp

```
// Etapa de ajuste de límite
 int64_t nActualTimespan = pindexLast->GetBlockTime() - nFirstBlockTime;
 LogPrintf("  nActualTimespan = %d  before bounds\n", nActualTimespan);
 if (nActualTimespan < params.nPowTargetTimespan/4)
     nActualTimespan = params.nPowTargetTimespan/4;
 if (nActualTimespan > params.nPowTargetTimespan*4)
     nActualTimespan = params.nPowTargetTimespan*4;

 // Recálculo
```

```
    const arith_uint256 bnPowLimit = UintToArith256(params.powLimit);
    arith_uint256 bnNew;
    arith_uint256 bnOld;
    bnNew.SetCompact(pindexLast->nBits);
    bnOld = bnNew;
    bnNew *= nActualTimespan;
    bnNew /= params.nPowTargetTimespan;

    if (bnNew > bnPowLimit)
        bnNew = bnPowLimit;
```

NOTE

Si bien la calibración del objetivo ocurre cada 2.016 bloques, debido a un error de tipo “por-un-paso”, en el cliente principal original de Bitcoin, el cálculo se basa en el tiempo total de los 2.015 bloques anteriores (no los 2.016 anteriores como debería ser), lo que resulta en un sesgo de re-calibración hacia la mayor dificultad, en 0,05%.

Los parámetros Interval (2.016 bloques) y TargetTimespan (de dos semanas, o bien 1.209.600 segundos) se definen en *chainparams.cpp*.

Para evitar una volatilidad muy extrema en los ajustes de la dificultad, el ajuste de re-calibración debe ser menor a un factor de cuatro (4) por ciclo. Si el ajuste requerido del objetivo es mayor que un factor de cuatro, se ajustará por un factor de 4 y no más. Cualquier ajuste adicional se logrará en el próximo período de re-calibración porque el desequilibrio persistirá durante los próximos 2.016 bloques. Por lo tanto, las grandes discrepancias entre la potencia de hash y la dificultad pueden tomar varios ciclos de 2.016 bloques para estabilizarse.

TIP

La dificultad de minería para un bloque de bitcoin es aproximadamente de '10 minutos de procesamiento' para toda la red, en función del tiempo que llevó minar los 2.016 bloques anteriores, ajustados cada 2.016 bloques. Esto se logra bajando o elevando el objetivo.

Téngase en cuenta que el objetivo es independiente del número de transacciones o del valor de las transacciones. Esto significa que la cantidad de potencia de hash y, por lo tanto, de la cantidad de electricidad gastada para asegurar la plataforma bitcoin, también es completamente independiente de la cantidad de transacciones. Bitcoin puede escalar, lograr una adopción más amplia y permanecer seguro sin ningún aumento en la potencia de hash desde el nivel actual. El aumento en la potencia de hash representa a las fuerzas del mercado a medida que los nuevos mineros ingresan al mismo para competir por la recompensa. Mientras haya suficiente potencia de hash bajo el control de los mineros que actúen honestamente en busca de la recompensa, es suficiente para prevenir ataques de "toma de control" y, por lo tanto, es suficiente para mantener segura a la plataforma bitcoin.

La dificultad de la minería está estrechamente relacionada con el costo de la electricidad y el tipo de cambio de bitcoin con respecto a la moneda utilizada para pagar la electricidad. Los sistemas de minería de alto rendimiento son lo más eficientes posible con el estado del arte actual en la fabricación de silicio, convirtiendo la electricidad en cómputo de hash a la tasa más alta posible. La influencia principal en el mercado de la minería es el precio de un kilovatio-hora de electricidad en bitcoin, porque eso determina la rentabilidad de la minería y, por lo tanto, los incentivos para entrar o salir del mercado de la minería.

Éxito en el Minado de un Bloque

Como se vió anteriormente, el nodo de Jing ha ensamblado un bloque candidato y lo ha preparado para la minería. Jing tiene varias plataformas de minería de hardware con circuitos integrados de aplicación específica, donde cientos de miles de circuitos integrados ejecutan el algoritmo SHA256 en paralelo a velocidades increíbles. Muchas de estas máquinas especializadas están conectadas a su nodo de minería a través de puertos USB o de una red de área local. A continuación, el nodo de minería que se ejecuta en el computador de escritorio de Jing, transmite la cabecera de bloque a su hardware de minería, que comienza a probar billones de nonces por segundo. Debido a que el nonce es de solo 32 bits, después de agotar todas las posibilidades de nonce (alrededor de 4 mil millones), el hardware de minería cambia la cabecera de bloque (ajustando el espacio extra del nonce en la transacción coinbase o en el sello de tiempo) y restablece el contador de nonce, probando nuevas combinaciones.

Casi 11 minutos después de comenzar a minar el bloque 277.316, una de las máquinas de hardware de minería, encuentra una solución y la envía de vuelta al nodo de minería.

Cuando se inserta en la cabecera de bloque, el nonce 924.591.752 produce un hash de bloque de:

```
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

que es menor que el objetivo:

```
00000000000000003A30C0000000000000000000000000000000000000000000
```

Inmediatamente, el nodo de minería de Jing transmite el bloque a todos sus pares. Ellos reciben, validan y luego propagan el nuevo bloque. A medida que el bloque se difunde a través de la red, cada nodo lo agrega a su propia copia de la cadena de bloques, extendiéndola a una nueva altura de 277.316 bloques. A medida que los nodos de minería reciben y validan el bloque, abandonan sus esfuerzos para encontrar un bloque a la misma altura e inmediatamente comienzan a calcular el siguiente bloque de la cadena, utilizando el bloque de Jing como el "padre". Al construir sobre el bloque recién descubierto por Jing, los otros mineros están esencialmente "votando" con su poder de minería, que respaldan el bloque de Jing, extendiendo de este modo la cadena.

En la siguiente sección, veremos el proceso que cada nodo utiliza para validar un bloque y para seleccionar la cadena más larga, creando el consenso que conforma la cadena de bloques descentralizada.

Validación de un Nuevo Bloque

El tercer paso en el mecanismo de consenso de bitcoin es la validación independiente de cada nuevo bloque por cada nodo en la red. A medida que el bloque recién resuelto se transmite a través de la red, cada nodo realiza una serie de pruebas para validarlo antes de retransmitirlo a sus pares. Esto garantiza que solo los bloques válidos se propaguen en la red. La validación independiente también garantiza que los mineros que actúan honestamente logren incorporen sus bloques en la cadena de bloques, obteniendo así su recompensa. A los mineros que actúan de manera deshonesta les repudian sus bloques y no solo pierden la recompensa, sino que también desperdician el esfuerzo realizado para encontrar una solución de la Prueba-de-Trabajo, incurriendo así en un costo sin compensación de la electricidad.

Cuando un nodo recibe un nuevo bloque, validará el bloque comparándolo con una larga lista de criterios que deben cumplirse todos; de lo contrario, el bloque es rechazado. Estos criterios se pueden ver en el Cliente Principal de Bitcoin en las funciones CheckBlock y CheckBlockHeader e incluyen:

- La estructura de datos de bloque es sintácticamente válida
- El hash de la cabecera de bloque es menor que el objetivo (hace cumplir la Prueba-de-Trabajo)
- El sello de tiempo del bloque es menos de dos horas en el futuro (lo que permite errores de tiempo)
- El tamaño del bloque está dentro de límites aceptables
- La primera transacción (y solo la primera) es una transacción de coinbase
- Todas las transacciones dentro del bloque son válidas utilizando la lista de verificación de transacciones discutida en Verificación Independiente de Transacciones

La validación independiente de cada nuevo bloque por cada nodo en la red asegura que los mineros no puedan hacer trampa. En secciones anteriores vimos cómo los mineros escribían una transacción que les otorgaba los nuevos bitcoins creados dentro del bloque y reclamaban las comisiones de transacción. ¿Por qué los mineros no escriben una transacción por mil bitcoins en lugar de la recompensa correcta? Porque cada nodo valida los bloques de acuerdo con las mismas reglas. Una transacción coinbase no válida, invalidaría todo el bloque, lo que resultaría en el rechazo del bloque y, por lo tanto, esa transacción nunca pasaría a formar parte del libro contable. Los mineros tienen que construir un bloque perfecto, basado en las reglas compartidas que siguen todos los nodos, y minarlo con una solución correcta a la Prueba-de-Trabajo. Para hacerlo, gastan mucha electricidad en la minería, y si hacen trampa, se desperdicia toda la electricidad y el esfuerzo. Es por eso que la validación independiente es un componente clave del consenso descentralizado.

Montaje y Selección de Cadenas de Bloques

El último paso en el mecanismo de consenso descentralizado de Bitcoin es el ensamblaje de bloques en cadenas y la selección de la cadena con la mayor cantidad de Prueba-de-Trabajo. Una vez que un nodo ha validado un nuevo bloque, intentará ensamblar una cadena conectando el bloque a la cadena de bloques existente.

Los nodos mantienen tres conjuntos de bloques: los conectados a la cadena de bloques principal, los que forman ramas de la cadena de bloques principal (cadenas secundarias) y, finalmente, los bloques que no tienen un padre conocido en las cadenas conocidas (los “huérfanos”). Los bloques no válidos se rechazan en cuanto falla uno de los criterios de validación

y, por lo tanto, no se incluyen en ningún grupo.

La "cadena principal" es en cualquier momento, la cadena de bloques *valida* y tiene el acumulado más grande de Prueba-de-Trabajo. En la mayoría de las circunstancias, esta es también la cadena con la mayor cantidad de bloques, a menos que haya dos cadenas de igual longitud y una tenga más cantidad de Prueba-de-Trabajo. La cadena principal también tendrá ramas con bloques que son "gemelos" de los bloques de la cadena principal. Estos bloques son válidos pero no forman parte de la cadena principal. Se guardan para referencia futura, en caso de que una de esas cadenas se extienda para exceder la cadena principal en cantidades de trabajo. En la siguiente sección (Bifurcaciones de la Cadena de Bloques), veremos cómo se producen cadenas secundarias como resultado de una minería casi simultánea de bloques a la misma altura.

Cuando se recibe un nuevo bloque, un nodo intentará ubicarlo en la cadena de bloques existente. El nodo tomará nota del campo "hash de bloque previo" de este bloque, que es la referencia al padre del bloque. Luego, el nodo intentará encontrar ese padre en la cadena de bloques existente. La mayoría de las veces, el padre se ubicará en la "punta" de la cadena principal, lo que significa que este nuevo bloque extiende la cadena principal. Por ejemplo, el nuevo bloque 277.316 tiene una referencia al hash de su bloque padre 277.315. La mayoría de los nodos que reciben al bloque 277.316, ya tendrán el bloque 277.315 como la punta de su cadena principal y, por lo tanto, vincularán el nuevo bloque y extenderán esa cadena.

A veces, como veremos en Bifurcaciones de la Cadena de Bloques, el nuevo bloque extiende una cadena que no corresponde a la cadena principal. En ese caso, el nodo unirá el nuevo bloque a la cadena secundaria que éste extiende, y luego comparará el trabajo de la cadena secundaria con la cadena principal. Si la cadena secundaria tiene más trabajo acumulado que la cadena principal, el nodo se *redirigirá* hacia la cadena secundaria, lo que significa que seleccionará la cadena secundaria como su nueva cadena principal, haciendo que la cadena principal anterior sea una cadena secundaria. Si el nodo es un minero, ahora ensamblará un nuevo bloque que extienda esta nueva cadena más larga.

Si se recibe un bloque válido y no se encuentra ningún padre en las cadenas existentes, ese bloque se considera un "huérfano". Los bloques huérfanos se guardan en el pool de bloques huérfanos donde permanecerán hasta que se reciba a su padre. Una vez que se recibe el padre y se enlaza a las cadenas existentes, el huérfano se puede sacar del pool de huérfanos y enlazarlo al padre, haciéndolo parte de una cadena. Los bloques huérfanos suelen ocurrir cuando dos bloques que fueron minados en poco tiempo el uno del otro se reciben en orden inverso (hijo antes del padre).

Al seleccionar la cadena válida con la cantidad de trabajo acumulado más grande, todos los nodos finalmente logran un consenso en toda la red. Las discrepancias temporales entre cadenas se resuelven eventualmente a medida que se agrega más trabajo, al extenderse una de las cadenas posibles. Los nodos de minería "votan" con su poder de minería al elegir qué cadena extender mediante la minería del siguiente bloque. Cuando minan un nuevo bloque y extienden la cadena, el nuevo bloque en sí representa su voto.

En la siguiente sección veremos cómo se resuelven las discrepancias entre las cadenas competidoras (bifurcaciones) mediante la selección independiente de la cadena con la cantidad de trabajo acumulado más grande.

Bifurcaciones de la Cadena de Bloques

Debido a que la cadena de bloques es una estructura de datos descentralizada, las diferentes copias no siempre son consistentes. Los bloques pueden llegar a diferentes nodos en diferentes momentos, haciendo que los nodos tengan diferentes perspectivas de la cadena de bloques. Para resolver esto, cada nodo siempre selecciona e intenta extender la cadena de bloques que representa la mayor cantidad de Prueba-de-Trabajo, también conocida como la cadena más larga o la cadena con mayor cantidad de trabajo acumulado. Al sumar el trabajo registrado en cada bloque de una cadena, un nodo puede calcular la cantidad total de trabajo que se ha gastado para crear esa cadena. Mientras todos los nodos seleccionen la cadena con la mayor cantidad de trabajo acumulado, la red global de bitcoin eventualmente converge a un estado consistente. Las bifurcaciones se producen como inconsistencias temporales entre las distintas versiones de la cadena de bloques, que se resuelven mediante una eventual reconvergencia a medida que se agregan más bloques a una de las bifurcaciones.

> **TIP** Las bifurcaciones de la cadena de bloques descritas en esta sección ocurren de manera natural como resultado de los retardos en las transmisiones en la red global. También veremos las bifurcaciones inducidas deliberadamente más adelante en este capítulo.

En los siguientes diagramas, seguiremos el progreso de un evento de "bifurcación" a lo largo de la red. El diagrama es una representación simplificada de la red bitcoin. Con fines ilustrativos, los diferentes bloques se muestran como formas

diferentes (estrella, triángulo, triángulo invertido, rombo), que se extienden por la red. Cada nodo en la red se representa como un círculo.

Cada nodo tiene su propia perspectiva de la cadena de bloques global. A medida que cada nodo recibe bloques de sus vecinos, actualiza su propia copia de la cadena de bloques, seleccionando la mayor cadena de trabajo acumulativo. Con fines ilustrativos, cada nodo contiene una forma que representa el bloque que cree que actualmente es la punta de la cadena principal. Entonces, si se ve una forma de estrella en el nodo, eso significa que el bloque de estrella es la punta de la cadena principal, en lo que respecta a ese nodo.

En el primer diagrama (Antes de la bifurcación—todos los nodos tienen la misma perspectiva.), la red tiene una perspectiva unificada de la cadena de bloques, con el bloque con forma de estrella como la punta de la cadena principal.

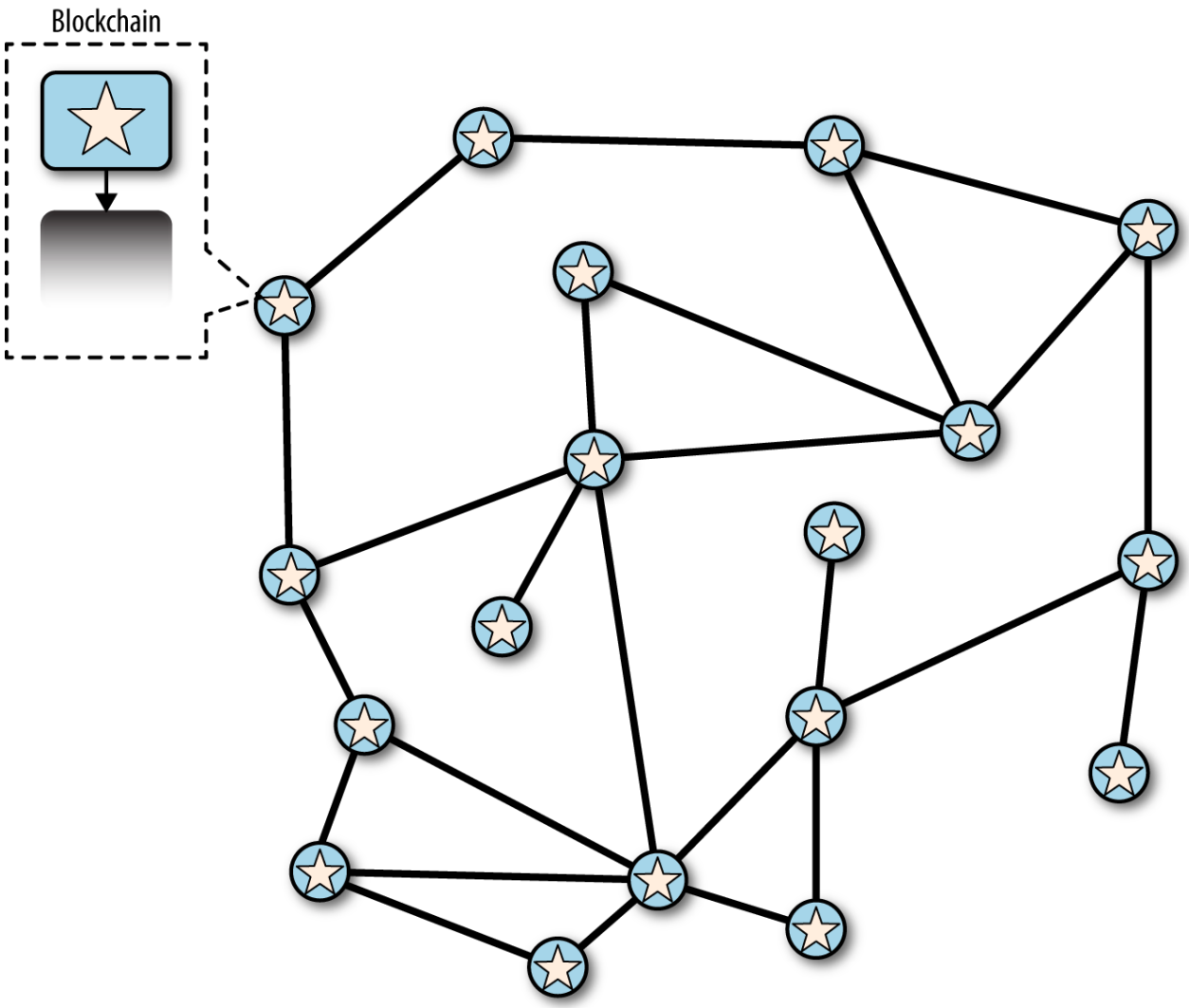


Figure 65. Antes de la bifurcación—todos los nodos tienen la misma perspectiva.

Una "bifurcación" se produce cada vez que dos bloques candidatos compiten para conformar la cadena de bloques más larga. Esto ocurre en condiciones normales cuando dos mineros resuelven el algoritmo de la Prueba-de-Trabajo con una diferencia de tiempo muy corta el uno del otro. A medida que ambos mineros descubren una solución para sus respectivos bloques candidatos, difunden inmediatamente su propio bloque "ganador" a sus vecinos inmediatos que comienzan a propagar el bloque a través de la red. Cada nodo que recibe un bloque válido lo incorporará a su cadena de bloques, extendiendo la cadena de bloques en un bloque adicional. Si ese nodo luego ve otro bloque candidato que se extiende a partir del mismo padre, conectará al segundo candidato en una cadena secundaria. Como resultado, algunos nodos "verán" de primero a uno de los bloques candidato, mientras que otros nodos verán al otro bloque y surgirán dos versiones competidoras de la cadena de bloques.

En Visualización de un evento bifurcación en la cadena de bloques: se encuentran dos bloques simultáneamente, vemos dos nodos mineros (Nodo X y Nodo Y) que minan dos bloques diferentes casi simultáneamente. Ambos bloques son hijos del bloque de estrellas, y extienden la cadena construyendo encima del bloque de estrellas. Para ayudarnos a visualizarlos, uno se ilustra como un bloque triangular que se origina en el Nodo X, y el otro se muestra como un bloque

triangular invertido que se origina en el Nodo Y.

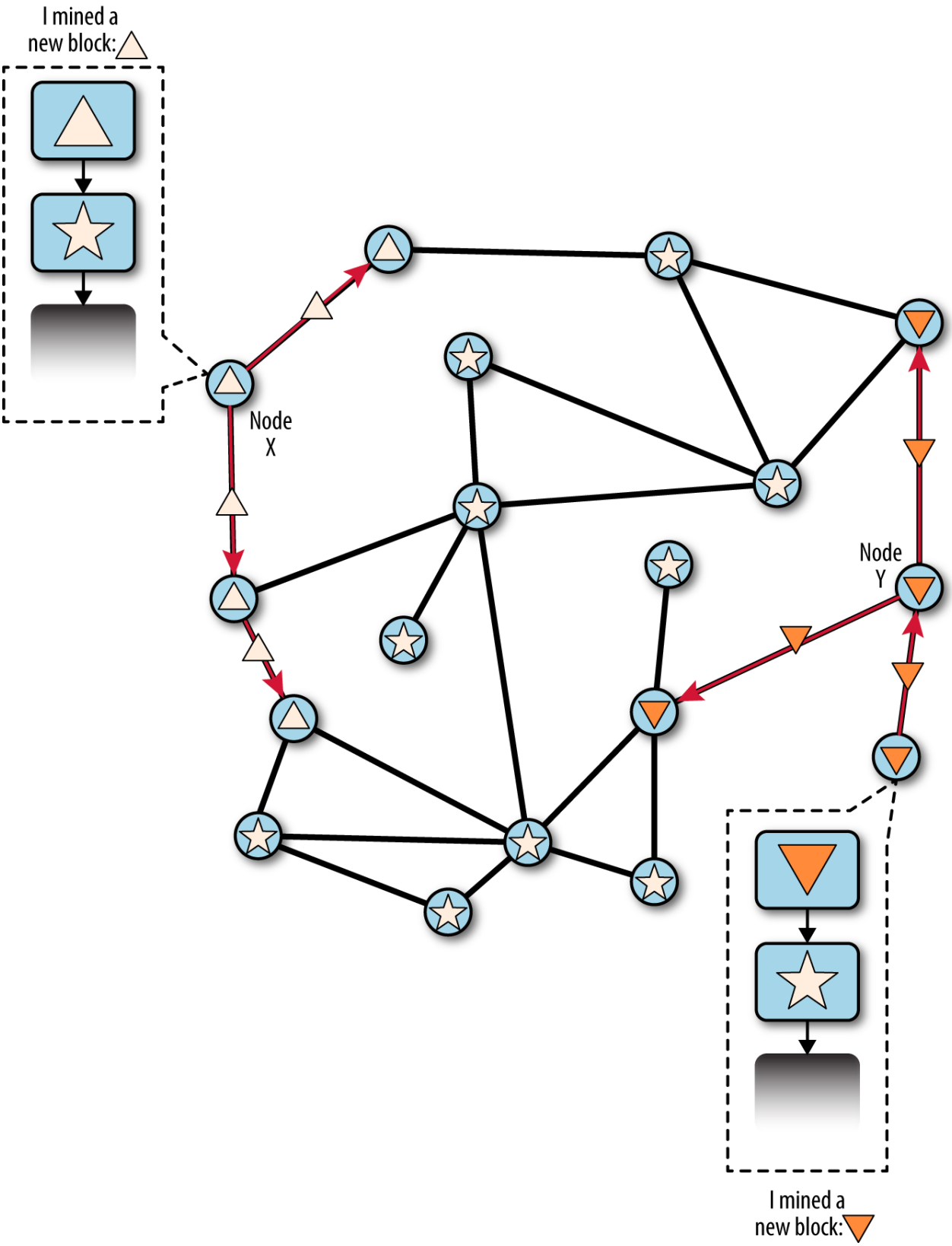


Figure 66. Visualización de un evento bifurcación en la cadena de bloques: se encuentran dos bloques simultáneamente

Supongamos, por ejemplo, que el minero del Nodo X encuentra una solución para la Prueba-de-Trabajo para un bloque "triángulo" que extiende la cadena de bloques, construyéndose encima de bloque padre con forma de "estrella". Casi simultáneamente, el minero del Nodo Y, que también estaba trabajando en extender la cadena desde el bloque "estrella", encuentra una solución para el bloque "triángulo invertido", que es su bloque candidato. Ahora, hay dos posibles bloques; uno que llamamos "triángulo", que se origina en el Nodo X; y uno que llamamos "triángulo invertido", que se origina en el Nodo Y. Ambos bloques son válidos, ambos bloques contienen una solución válida para la Prueba-de-Trabajo, y ambos bloques se extienden a partir del mismo bloque padre (el bloque "estrella"). Es probable que ambos bloques contengan la mayoría de las mismas transacciones, con solo unas pocas diferencias en el orden de las transacciones.

A medida que los dos bloques se propagan, algunos nodos reciben el bloque "triángulo" primero y algunos reciben el bloque "triángulo invertido" primero. Como se muestra en Visualización de un evento bifurcación en la cadena de bloques: se propagan dos bloques, seccionando la red, la red se divide en dos perspectivas diferentes de la cadena de bloques; una que observa al bloque de triángulo en la punta de la cadena, y otra que ve en la punta al bloque de triángulo invertido.

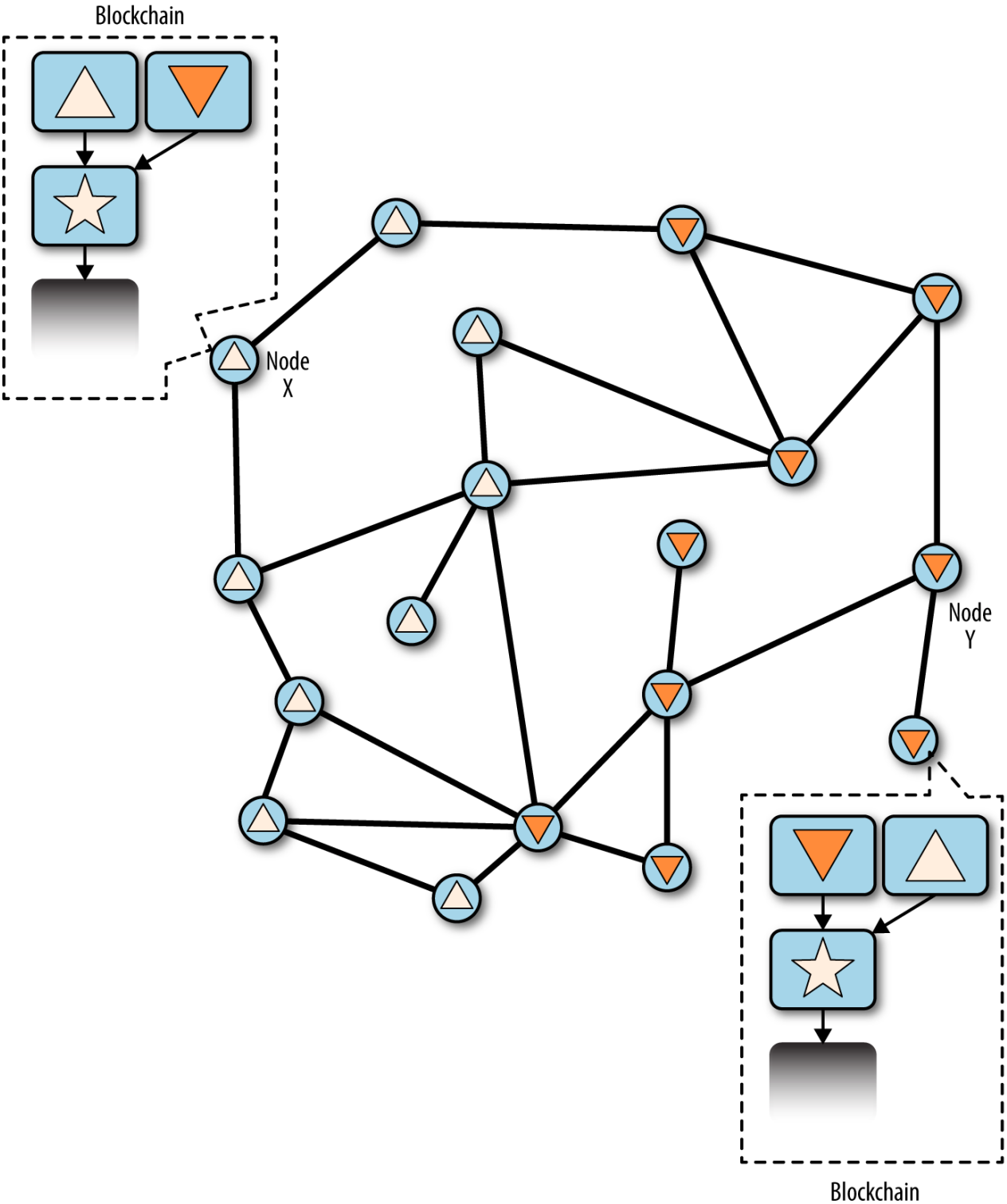


Figure 67. Visualización de un evento bifurcación en la cadena de bloques: se propagan dos bloques, seccionando la red

En el diagrama, un "Nodo X" elegido al azar recibió el bloque triangular primero y extendió la cadena de estrellas con él. El nodo X seleccionó la cadena con el bloque "triángulo" como la cadena principal. Más tarde, el Nodo X también recibió el bloque "triángulo invertido". Como se recibió en segundo lugar, se supone que ha "perdido" la carrera. Sin embargo, el bloque "triángulo invertido" no se descarta. Está vinculado al bloque padre "estrella" y forma una cadena secundaria. Si bien el Nodo X asume que ha seleccionado correctamente la cadena ganadora, mantiene la cadena "perdedora" para que tenga la información necesaria para volver a converger a ella si la cadena "perdedora" termina "ganando" a la larga.

En el otro lado de la red, el Nodo Y construye una cadena de bloques basada en su propia perspectiva de la secuencia de eventos. Este nodo recibió primero el "triángulo invertido" y eligió a la cadena representada con esta punta como el "ganador". Cuando más tarde recibe el bloque "triángulo", lo conecta con el bloque padre con forma de "estrella" en una cadena que presumirá como cadena secundaria.

Ninguno de los dos lados está ni en lo "correcto" ni "equivocado". Ambas son perspectivas válidas de la cadena de bloques. Solo en retrospectiva prevalecerá una de ellas, en función de cómo estas dos cadenas competidoras se extiendan mediante una cantidad de trabajo adicional.

Los nodos de minería cuya perspectiva concuerda con el Nodo X comenzarán inmediatamente a trabajar en la minería de un bloque candidato que extienda la cadena que posea al bloque "triángulo" como punta. Al vincular al "triángulo" como el padre de su bloque candidato, están votando con su potencia de hash. Su voto apoya la cadena que han elegido como la cadena principal.

Cualquier nodo de minería cuya perspectiva concuerde con el Nodo Y comenzará a construir un nodo candidato con el bloque con forma de "triángulo invertido" como su padre, extendiendo la cadena que ellos creen que es la cadena principal. Y así, la carrera comienza de nuevo.

Las bifurcaciones casi siempre se resuelven dentro de un bloque. Si bien parte de la potencia de hash de la red se dedica a construir sobre el "triángulo" como padre, otra parte de la potencia de hash se centrará en construir sobre el "triángulo invertido". Incluso si la potencia de hash se divide casi por igual, es muy probable que un grupo de mineros encuentre una solución y la propague mucho antes que el otro grupo de mineros haya encontrado alguna solución. Digamos, por ejemplo, que los mineros que construyen sobre el "triángulo" encuentran un nuevo bloque "rombo" que extiende la cadena (por ejemplo, estrella-triángulo-rombo). Inmediatamente propagan este nuevo bloque y toda la red lo ve como una solución válida, como se muestra en Visualización de un evento de bifurcación de la cadena de bloques: un nuevo bloque extiende una bifurcación, re-dirigiendo a la red..

Todos los nodos que hayan elegido al "triángulo" como ganador en la ronda anterior simplemente extenderán la cadena un bloque más. Sin embargo, los nodos que eligieron al "triángulo invertido" como el ganador ahora verán dos cadenas: la de estrella-triángulo-rombo y la de estrella-triángulo-invertido. La cadena estrella-triángulo-rombo ahora es más larga (posee más trabajo acumulado) que la otra cadena. Como resultado, esos nodos establecerán a la cadena estrella-triángulo-rombo como la cadena principal y re-clasificarán a la cadena estrella-triángulo-invertido como cadena secundaria, como se muestra en Visualización de un evento bifurcación de la cadena de bloques: la red reconverge en una nueva cadena más larga. Esta es una convergencia de la cadena, porque esos nodos se ven obligados a revisar su visión de la cadena de bloques para incorporar la nueva evidencia de una cadena más larga. Cualquier minero que trabaje para extender la cadena estrella-triángulo-invertido ahora detendrá su trabajo porque su bloque candidato es un "huérfano", ya que su bloque padre "triángulo-invertido" ya no estará más en la cadena más larga. Las transacciones dentro del "triángulo-invertido" que no están dentro del bloque "triángulo" se vuelven a insertar en el tanque de memoria o “mempool” para incluirlas en el siguiente bloque y para que pasen a formar parte de la cadena principal. Toda la red se reconvierte en una sola cadena de bloques estrella-triángulo-rombo, con el bloque "rombo" como el último bloque de la cadena. Todos los mineros comienzan inmediatamente a trabajar en bloques candidatos que hagan referencia al "rombo" como su padre para extender la cadena estrella-triángulo-rombo.

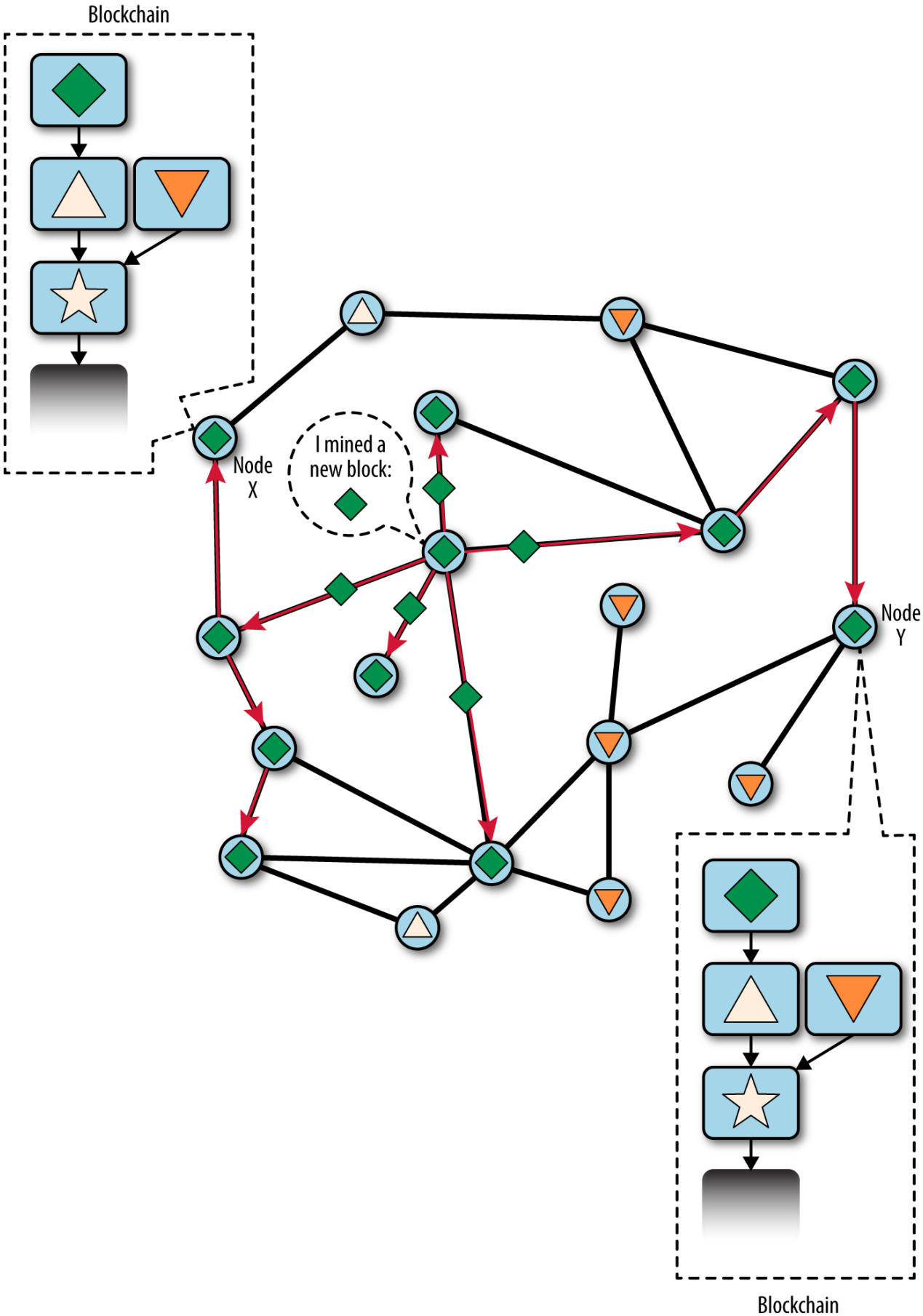


Figure 68. Visualización de un evento de bifurcación de la cadena de bloques: un nuevo bloque extiende una bifurcación, redirigiendo a la red.

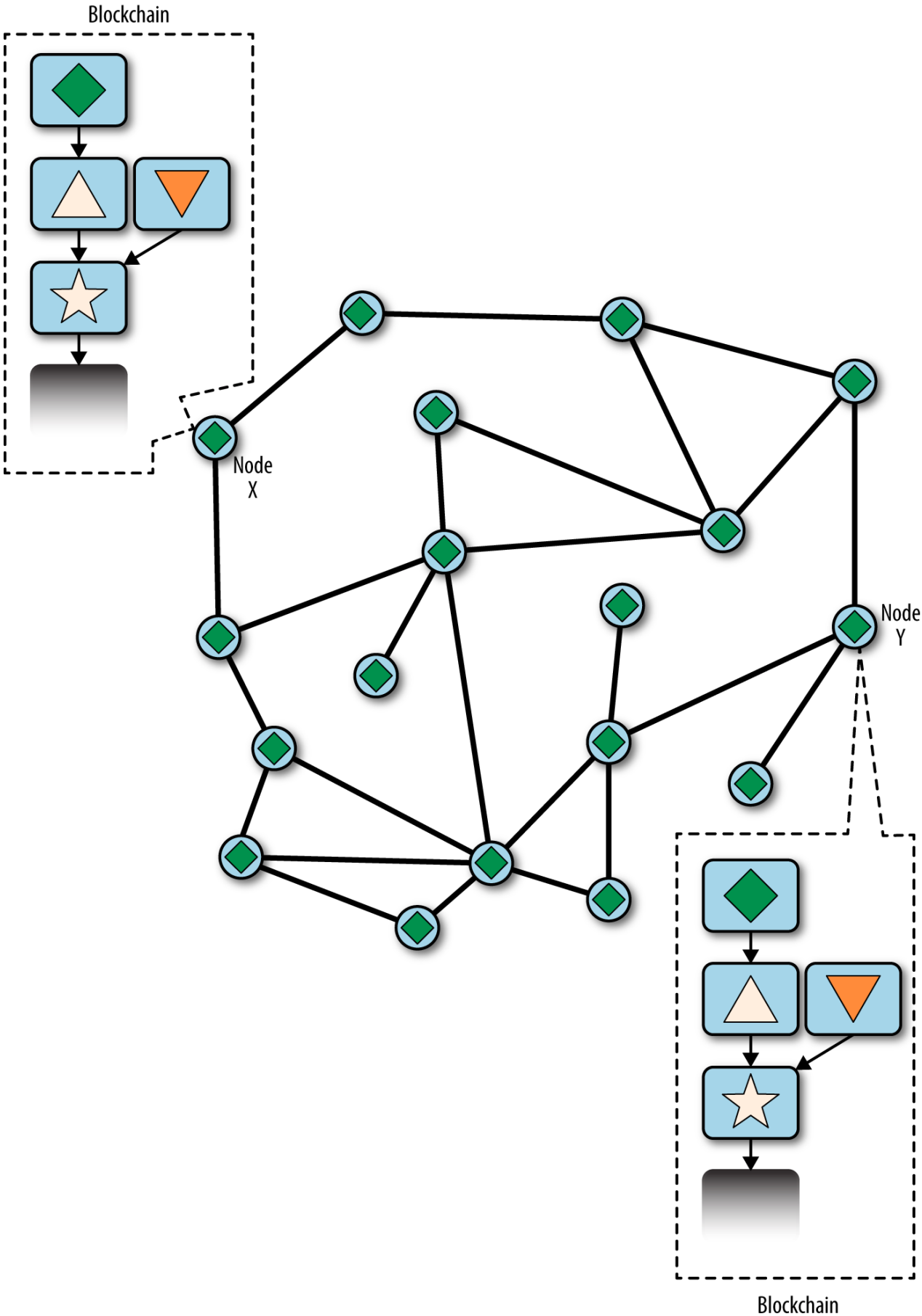


Figure 69. Visualización de un evento bifurcación de la cadena de bloques: la red reconverge en una nueva cadena más larga

Teóricamente es posible que una bifurcación se extienda a dos bloques, si los mineros encuentran dos bloques casi simultáneamente en "lados" opuestos de una bifurcación anterior. Sin embargo, la posibilidad de que eso ocurra es muy baja. Mientras que una bifurcación de un bloque puede ocurrir todos los días, una bifurcación de dos bloques ocurre como máximo una vez cada pocas semanas.

El intervalo entre bloques de 10 minutos de bitcoin es un compromiso de diseño entre tiempos de confirmación rápidos (liquidación de transacciones) y la probabilidad de una bifurcación. Un intervalo entre bloques más corto haría que las transacciones se confirmasen más rápido, pero conduciría a bifurcaciones más frecuentes en la cadena de bloques, mientras que un intervalo entre bloques más largo disminuiría el número de bifurcaciones, pero haría que la confirmación fuese más lenta.

Minería y la Carrera de Hashing

La minería de bitcoin es una industria extremadamente competitiva. La potencia de hash ha aumentado exponencialmente cada año desde la aparición de bitcoin. En algunos años, el crecimiento ha reflejado un cambio completo de tecnología, como fue en 2010 y 2011, cuando muchos mineros pasaron de usar la minería de CPU a la minería de GPU y hacia la minería de matrices de compuertas programables de campo (FPGA). En 2013, la introducción de la minería ASIC condujo a otro salto gigante en el poder de minería, al colocar la función SHA256 directamente en los chips de silicio especializados para la minería. Los primeros chips de este tipo podrían entregar más poder de minería en una sola caja que toda la red bitcoin en 2010.

La siguiente lista muestra la potencia de hash total de la red Bitcoin, durante los primeros ocho años de operación:

2009

0.5 MH / seg-8 MH / seg (16#x00D7; crecimiento)

2010

8 MH / seg-116 GH / s (14500× crecimiento)

2011

116 GH/seg–9 TH/seg (78× crecimiento)

2012

9 TH/sec–23 TH/sec (2.56#x00D7; growth)

2013

23 TH/seg–10 PH/seg (450× crecimiento)

2014

10 PH/seg–300 PH/seg (30× crecimiento)

2015

300 PH/seg-800 PH/seg (2.66× crecimiento)

2016

800 PH/seg-2.5 EH/seg (3.12× crecimiento)

En el gráfico que se muestra en Potencia de hash total, terahashes por segundo (TH/seg), podemos ver que la potencia de hash de la red bitcoin aumentó en los últimos dos años. Como puede verse, la competencia entre los mineros y el crecimiento de bitcoin ha resultado en un aumento exponencial en la potencia de hash (hashes totales por segundo en toda la red).

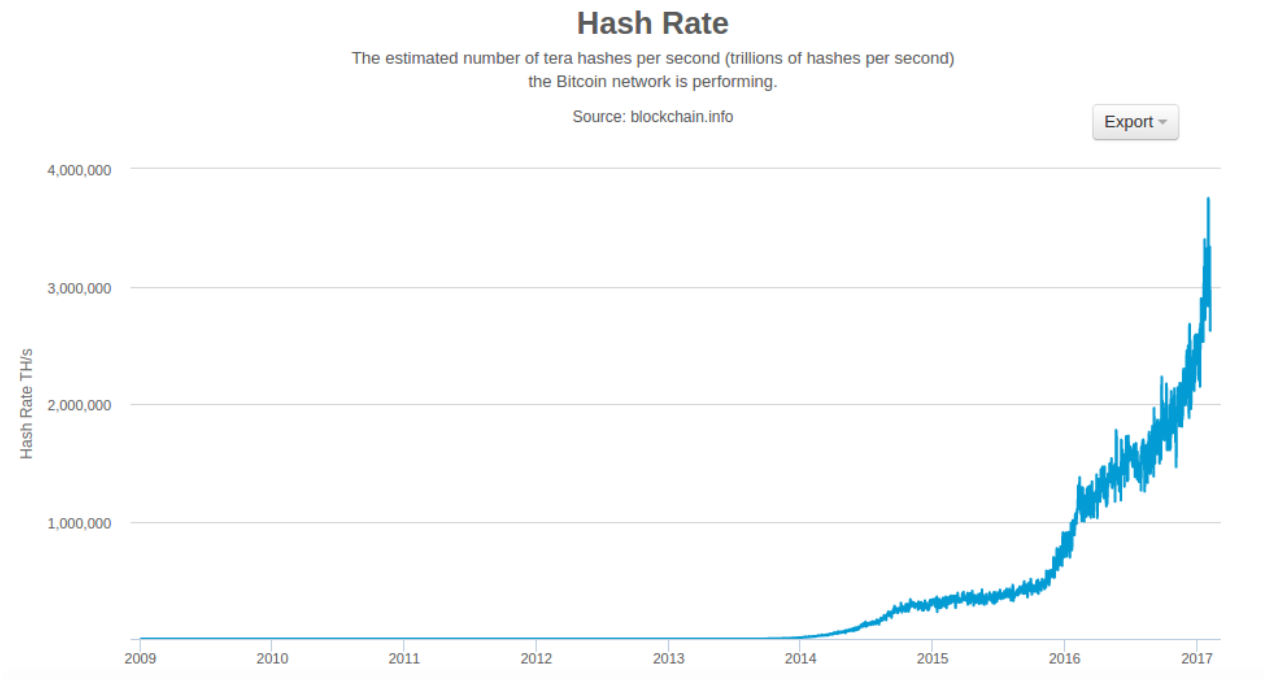


Figure 70. Potencia de hash total, terahashes por segundo (TH/seg)

A medida que la cantidad de potencia de hash aplicada a la minería de bitcoin se ha multiplicado, la dificultad ha aumentado para poder igualarla. La métrica de dificultad en el cuadro que se muestra en Medida de la dificultad de minería de bitcoin. se mide como una relación de la dificultad actual sobre la dificultad mínima (la dificultad que tuvo el primer bloque).

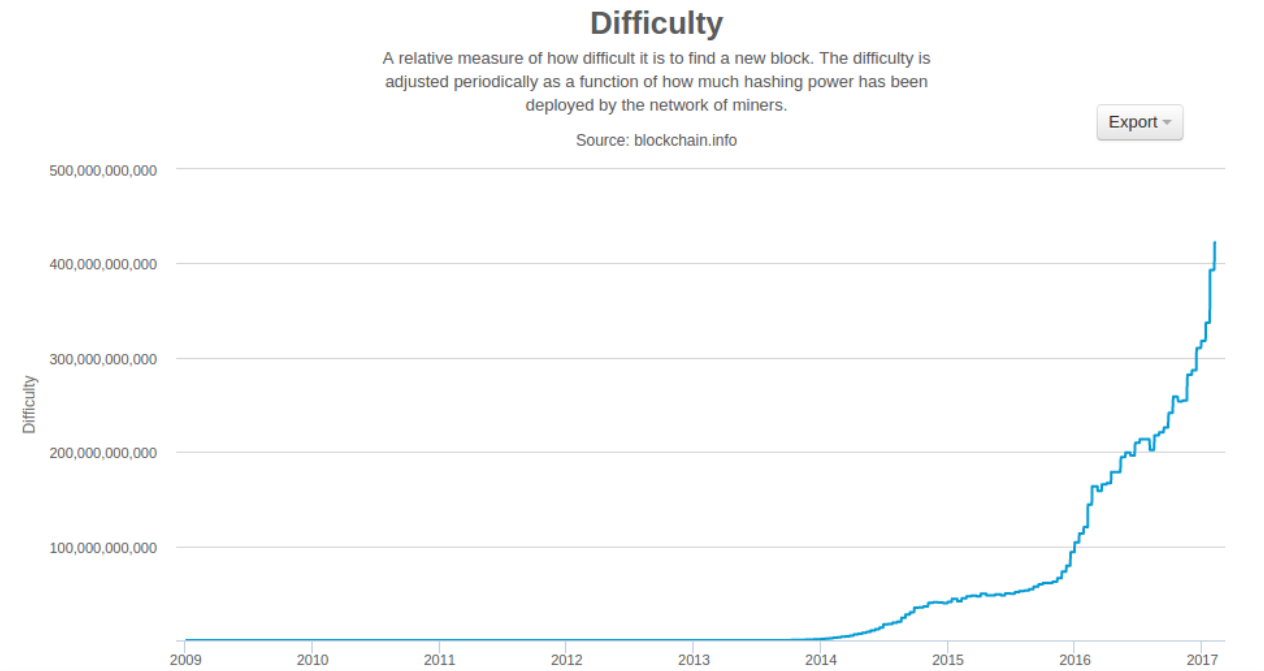


Figure 71. Medida de la dificultad de minería de bitcoin.

En los últimos dos años, los chips de minería ASIC se han vuelto cada vez más densos, acercándose a la vanguardia de la fabricación de silicio con un tamaño característico (resolución) de 16 nanómetros (nm). Actualmente, los fabricantes de ASIC tienen como objetivo superar a los fabricantes de chips de CPU de propósito general, diseñando chips con un tamaño característico de 14 nm, porque la rentabilidad de la minería está impulsando esta industria aún más rápido que la computación general. No quedan más saltos gigantes en la minería de bitcoin, porque la industria ha llegado a la vanguardia de la Ley de Moore, que estipula que la densidad informática se duplicará aproximadamente cada 18 meses. Aún así, el poder de minería de la red continúa avanzando a un ritmo exponencial a medida que la carrera por chips de mayor densidad coincide con una carrera por centros de datos de mayor densidad donde se pueden implementar miles de estos chips. Ya no se trata de la cantidad de minería que se puede hacer con un chip, sino de cuántos chips se pueden apilar dentro de una edificación, mientras se logra disipar el calor y se proporciona la energía adecuada.

La Solución del Nonce Extra

Desde 2012, la minería de bitcoins ha evolucionado para resolver una limitación fundamental en la estructura de la cabecera de bloque. En los primeros días de bitcoin, un minero podría encontrar un bloque iterando a través del nonce hasta que el hash resultante estuviera por debajo del objetivo. A medida que ha aumentado la dificultad, los mineros a menudo recorren los 4 mil millones de valores del nonce sin encontrar un bloque. Sin embargo, esto se resolvió fácilmente actualizando el sello de tiempo del bloque para tener en cuenta el tiempo transcurrido. Debido a que el sello de tiempo es parte de la cabecera, el cambio permitiría a los mineros repetir los valores del nonce nuevamente con diferentes resultados. Sin embargo, una vez que el hardware de minería superó los 4 GH/seg, este enfoque se volvió cada vez más difícil porque los valores nonce se agotaron en menos de un segundo. Conforme el hardware de minería ASIC comenzó a presionar y luego a exceder una tasa de hash de Tera-hashes por segundo, el software de minería necesitó más espacio para los valores nonce de modo de poder encontrar bloques válidos. El sello de tiempo podría extenderse un poco, pero moverla demasiado hacia el futuro provocaría que el bloque dejara de ser válido. Se necesitaba una nueva fuente de "cambios" en la cabecera de bloque. La solución fue utilizar la transacción coinbase como fuente de valores nonce adicionales. Debido a que el script coinbase puede almacenar entre 2 y 100 bytes de datos, los mineros comenzaron a usar ese espacio como espacio nonce adicional, lo que les permite explorar un rango mucho más grande de valores de cabecera de bloque para encontrar bloques válidos. La transacción de coinbase se incluye en el árbol de merkle, lo que significa que cualquier cambio en el script de coinbase hace que cambie la raíz de merkle. Ocho bytes de nonce extra, más los 4 bytes de nonce "estándar" permiten a los mineros explorar un total de 2^{96} (8 seguidos por 28 ceros) posibilidades *por segundo* sin tener que modificar el sello de tiempo. Si, en el futuro, los mineros pudieran aprovechar todas estas posibilidades, podrían modificar el sello de tiempo. También hay más espacio en el script de coinbase para la futura expansión del espacio adicional del nonce.

Pools de Minería

En este entorno altamente competitivo, los mineros individuales que trabajan solos (también conocidos como mineros en solitario) no tienen muchas oportunidad. La probabilidad de que encuentren un bloque para compensar sus costos de electricidad y hardware es tan baja que representa una apuesta, como jugar a la lotería. Incluso el sistema de minería ASIC de alta eficiencia no puede mantenerle el paso a los sistemas comerciales que apilan decenas de miles de estos chips en granjas gigantescas cercanas a las centrales hidroeléctricas. Los mineros ahora colaboran para formar grupos de minería, uniendo su potencia de hash y compartiendo la recompensa entre miles de participantes. Al participar en un grupo, los mineros obtienen una parte menor de la recompensa general, pero generalmente son recompensados todos los días, lo que reduce un factor de incertidumbre.

Veamos un ejemplo específico. Supongamos que un minero ha comprado hardware de minería con una tasa de hasheo combinada de 14.000 gigahashes por segundo (GH/s), o 14 TH/s. En 2017, este equipo costaba aproximadamente $ 2.500 USD. El hardware consume 1375 vatios (1,3 kW) de electricidad cuando funciona, 33 kW-horas al día, a un costo de $ 1 a $ 2 por día a tarifas de consumo eléctrico muy bajas. Con la dificultad actual de bitcoin, ese minero podrá extraer en solitario un bloque aproximadamente una vez cada 4 años. ¿Cómo calculamos esa probabilidad? Se basa en una tasa de hasheo de toda la red de 3 EH/seg (en 2017), y la tasa de este minero de 14 TH/seg:

- $P = (14 * 10^{12/3} * 10^{18)} * 210240 = 0,98$

i. donde 210240 es el número de bloques en cuatro años. El minero tiene una probabilidad del 98% de encontrar un bloque en cuatro años, según la tasa de hash global al comienzo del período.

Si el minero encuentra un solo bloque en ese período de tiempo, el pago de 6.25 bitcoins, a aproximadamente $1,000 por bitcoin, dará como resultado un pago único de $6,250, lo que producirá una ganancia neta de aproximadamente $750. Sin embargo, la posibilidad de encontrar un bloqueo en un período de 4 años depende de la suerte del minero. Podría encontrar dos bloques en 4 años y obtener mayores ganancias. O tal vez no encuentre ni un solo bloque durante 5 años y sufra una gran pérdida financiera. Peor aún, es probable que la dificultad del algoritmo de Prueba-de-Trabajo de bitcoin aumente significativamente durante ese período, al ritmo actual de crecimiento de la potencia de hash, lo que significa que el minero tiene, como máximo, un año para recuperarse incluso antes que el hardware sea efectivamente obsoleto y deba ser reemplazado por un hardware de minería más potente. Financieramente, esto solo tiene sentido a un costo de electricidad muy bajo (menos de 1 centavo por kW-hora) y solo a gran escala.

Las pools de minería coordinan muchos cientos o miles de mineros, mediante protocolos especializados para pools de minería. Los mineros individuales configuran sus equipos de minería para conectarse a un servidor de pool, después de haber creado una cuenta con el pool. Su hardware de minería permanece conectado al servidor del pool mientras están minando, sincronizando así sus esfuerzos con los otros mineros. Por lo tanto, los mineros de pool comparten el esfuerzo para extraer un bloque y luego comparten las recompensas.

Los bloques exitosos pagan la recompensa a una dirección bitcoin del pool, en lugar de a los mineros individuales. El servidor del pool hará periódicamente pagos a las direcciones bitcoin de los mineros, cuando su parte de la recompensa haya llegado a un cierto umbral. Normalmente, el servidor de pool cobra una tarifa de porcentaje de los beneficios por la prestación del servicio de minería en pool.

Los mineros que participan en una agrupación dividen el trabajo de buscar una solución para un bloque candidato, ganando "acciones" por su contribución de minería. El grupo de minería establece un objetivo más alto (menor dificultad) para ganar una participación, generalmente más de 1,000 veces más fácil que el objetivo de la red bitcoin. Cuando alguien en el grupo mina con éxito un bloque, el grupo obtiene la recompensa y luego se reparte entre todos los mineros en proporción al número de acciones que contribuyeron al esfuerzo.

Las agrupaciones están abiertas a cualquier minero, grande o pequeño, profesional o aficionado. Por lo tanto, un grupo tendrá algunos participantes con una sola máquina de minería pequeña, y otros con una granja llena de hardware de minería de alta gama. Algunos llevarán a cabo su minería con unas pocas decenas de kilovatios de electricidad, otros llevarán un centro de datos que consumirá un mega-vatio de energía. ¿Cómo mide un grupo de minería las contribuciones individuales, para distribuir equitativamente las recompensas, sin la posibilidad de hacer trampa? La respuesta es usar el algoritmo de Prueba-de-Trabajo de bitcoin para medir la contribución de cada minero de grupo, pero con una dificultad menor para que incluso los mineros de grupo más pequeños ganen una parte con la frecuencia suficiente para que valga la pena contribuir al grupo. Al establecer una dificultad menor para ganar acciones, el grupo mide la cantidad de trabajo realizado por cada minero. Cada vez que un minero de grupo encuentra un hash de cabecera de bloque que es menor que el objetivo del grupo, prueba que ha realizado el trabajo de hash para encontrar ese resultado. Más importante aún, el trabajo para encontrar acciones contribuye, de una manera estadísticamente mesurable, al esfuerzo general para encontrar un hash más bajo que el objetivo de la red bitcoin. Miles de mineros que intentan encontrar hashes de bajo valor eventualmente encontrarán uno lo suficientemente bajo como para satisfacer el objetivo de la red bitcoin.

Volvamos a la analogía de un juego de dados. Si los jugadores de dados están tirando los dados con el objetivo de lanzar menos de cuatro (la dificultad general de la red), un pool fijaría un objetivo más fácil, contando cuántas veces los jugadores del pool lograron tirar menos de ocho. Cuando los jugadores del pool tiran menos de ocho (el objetivo de cuota del pool), ganan cuotas, pero no ganan el juego, ya que no alcanzan el objetivo del juego (menos de cuatro). Los jugadores del pool lograrán el objetivo del pool, que es más fácil, con mucha más frecuencia, ganando cuotas muy regularmente, incluso cuando no logran el objetivo más difícil de ganar el juego. De vez en cuando, uno de los jugadores del pool lanzará un tiro de dados combinado de menos de cuatro y en ese caso, el pool gana. Entonces, las ganancias se pueden distribuir a los jugadores del pool sobre la base de las cuotas que habían conseguido. A pesar de que el objetivo de ocho-o-menos no significaba ganar, era una forma razonable de medir el número de lanzamientos de dados de los jugadores, y que de vez en cuando produce un tiro de menos-que-cuatro.

Del mismo modo, un grupo de minería establecerá un objetivo de grupo (más alto y más fácil) que garantizará que un minero del grupo, individualmente pueda encontrar hashes de cabecera de bloque que a menudo sean menores que el objetivo del grupo, ganando acciones. De vez en cuando, uno de estos intentos producirá un hash de cabecera de bloque que es menor que el objetivo de la red bitcoin, lo que lo convierte en un bloque válido, lo que le hace ganar a todo el grupo.

Pools gestionados

La mayoría de los grupos de minería están "gestionados", lo que significa que hay una empresa o individuo que ejecuta un servidor para la agrupación. El propietario del servidor de la agrupación se denomina *operador de la agrupación* y cobra a los mineros de la agrupación una tarifa porcentual de las ganancias.

El servidor de la agrupación ejecuta un software especializado y un protocolo de minería de agrupación que coordina las actividades de los mineros de esta agrupación. El servidor del grupo también está conectado a uno o más nodos completos de bitcoin y tiene acceso directo a una copia completa de la base de datos de la cadena de bloques. Esto permite al servidor de la agrupación validar bloques y transacciones en nombre de los mineros de la agrupación, liberándolos de la carga de ejecutar un nodo completo. Para los mineros de agrupaciones, esto es una consideración importante, porque un nodo completo requiere una computadora dedicada con al menos 100 a 150 GB de almacenamiento persistente (en disco) y al menos entre 2 a 4 GB de memoria (RAM). Además, el software de bitcoin que se ejecuta en el nodo completo necesita ser monitoreado, mantenido y actualizado con frecuencia. Cualquier tiempo de inactividad causado por falta de mantenimiento o falta de recursos afectará la rentabilidad del grupo minero. Para muchos mineros, la capacidad de minar sin tener que ejecutar un nodo completo es otro gran beneficio de unirse a un grupo administrado.

Los mineros de la agrupación se conectan al servidor del grupo utilizando un protocolo de minería tal como lo es Stratum (STM) o GetBlockTemplate (GBT). Un estándar anterior llamado GetWork (GWK) ha quedado obsoleto en su mayoría desde finales de 2012, porque no admite fácilmente la minería a tasas de hash superiores a 4 GH/s. Los protocolos STM y GBT crean bloques tipo *plantillas* que contienen una especie de formulario de cabecera de bloque candidato. El servidor del grupo ensambla un bloque candidato agregando transacciones y agregando una transacción coinbase (con un nonce con espacio de reserva adicional); calcula la raíz de merkle y vincula al candidato con el hash de bloque previo. La cabecera de bloque del bloque candidato se envía a cada uno de los mineros del grupo como una plantilla. Cada minero de grupo procede a minar, utilizando la plantilla del bloque, con un objetivo más alto (más fácil) que el objetivo de la red bitcoin, y envía los resultados exitosos al servidor del grupo para ganar acciones.

Agrupaciones de minería tipo "peer-to-peer" (P2Pool)

Las agrupaciones administradas crean la posibilidad de hacer trampa por el operador de la agrupación, quién podría dirigir el esfuerzo del grupo para falsificar transacciones, mediante un doble gasto o invalidar bloques (véase Ataques Contra el Consenso). Además, los servidores de un grupo centralizado representan un punto único de falla. Si el servidor del grupo quedase inactivo o se ralentizara por un ataque de denegación de servicio, los mineros del grupo no podrían operar. En 2011, para resolver estos problemas de centralización, se propuso e implementó un nuevo método de minería de grupos: P2Pool, un grupo de minería entre pares, sin un operador central.

El “P2Pool” funciona descentralizando las funciones del servidor de la agrupación, implementando un sistema paralelo similar a una pequeña cadena de bloques llamada *cadena de acciones*. Una cadena de acciones es una cadena de bloques que se ejecuta con una dificultad menor que la cadena de bloques de bitcoin. La cadena de acciones permite a los mineros del grupo colaborar en un grupo descentralizado mediante la minería de acciones en la cadena de acciones a una tasa de un bloque de acciones cada 30 segundos. Cada uno de los bloques en la cadena de acciones registra una recompensa de acciones proporcional para los mineros que contribuyen con el trabajo, llevando las acciones hacia adelante desde el bloque de acciones anterior. Cuando uno de los bloques de acciones también alcanza el objetivo de la red bitcoin, se propaga e incluye en la cadena de bloques de bitcoin, recompensando a todos los mineros que contribuyeron a todas las acciones que precedieron al bloque de acciones ganador. Esencialmente, en lugar de un servidor para el grupo que realiza un seguimiento de las acciones y recompensas de cada minero de la agrupación, la cadena de acciones permite a todos los mineros de la agrupación realizar un seguimiento de todas las acciones utilizando un mecanismo de consenso descentralizado como el mecanismo de consenso de la cadena de bloques de bitcoin.

La minería P2Pool es más compleja que la minería centralizada de grupos porque requiere que los mineros del grupo descentralizado tengan una computadora dedicada con suficiente espacio en disco, memoria y ancho de banda de Internet para admitir un nodo bitcoin completo y el software del nodo P2Pool. Los mineros de P2Pool conectan su hardware de minería a su nodo P2Pool local, que simulará las funciones del servidor de un grupo centralizado, enviando plantillas de bloques al hardware de minería. En P2Pool, los mineros de grupos individuales construyen sus propios bloques de candidatos, agregando transacciones de manera muy similar a los mineros en solitario, pero luego minan en colaboración en la cadena de acciones. P2Pool es un enfoque híbrido que tiene la ventaja de pagos mucho más granulares que la minería en solitario, pero sin dar demasiado control a un operador de grupo como los grupos centralmente gestionados.

A pesar de que P2Pool reduce la centralización del poder que poseen los administradores de agrupaciones centralizadas de minería, es posiblemente vulnerable a ataques del 51% en su propia cadena de acciones. Una adopción mucho más amplia del P2Pool no resuelve el problema de ataque del 51% para la red bitcoin en sí. Por el contrario, P2Pool hace que Bitcoin sea más robusto en general, como parte de un ecosistema de minería diversificada.

Ataques Contra el Consenso

El mecanismo de consenso de bitcoin es, al menos teóricamente, vulnerable al ataque de mineros (o agrupaciones) que intentan usar su potencia de hash para fines deshonestos o destructivos. Como vimos, el mecanismo de consenso depende de que la mayoría de los mineros actúen honestamente por interés propio. Sin embargo, si un minero o un grupo de mineros pueden acaparar una parte significativa del poder de minería, pueden atacar el mecanismo de consenso para arruinar la seguridad y la disponibilidad de la red bitcoin.

Es importante tener en cuenta que los ataques de consenso solo pueden afectar el consenso futuro, o en el mejor de los casos, el pasado más reciente (en decenas de bloques). El libro contable de bitcoin se vuelve cada vez más inmutable a medida que pasa el tiempo. Si bien, en teoría, se puede lograr una bifurcación a cualquier profundidad, en la práctica, el poder de cómputo necesario para forzar una bifurcación muy profunda es inmenso, lo que hace que los bloques viejos sean prácticamente inmutables. Los ataques de consenso tampoco afectan la seguridad de las llaves privadas como

tampoco afectan al algoritmo de firma (ECDSA). Un ataque de consenso no puede robar bitcoins, gastar bitcoins sin firmas, redirigir la red bitcoin o cambiar transacciones pasadas o registros de propiedad.Los ataques de consenso solo pueden afectar los bloques más recientes y causar interrupciones por denegación de servicio en la creación de futuros bloques.

Un escenario de ataque contra el mecanismo de consenso se llama "ataque del 51%". En este escenario, un grupo de mineros, que controlan una mayoría (51%) de la potencia de hash de la red total, se confabulan para atacar bitcoin. Con la capacidad de minar la mayoría de los bloques, los mineros atacantes pueden causar "bifurcaciones" deliberadas en la cadena de bloques y transacciones falsas mediante el doble gasto o ejecutar ataques de denegación de servicio contra transacciones o direcciones específicas. Un ataque de bifurcación/doble gasto es donde el atacante hace que los bloques previamente confirmados se invaliden al bifurcarse por debajo de ellos y volver a dirigir la red a una cadena alternativa. Con suficiente potencia, un atacante puede invalidar seis o más bloques seguidos, lo que hace que se invaliden las transacciones que se consideraron inmutables (seis confirmaciones). Téngase en cuenta que un doble gasto solo se puede hacer con las transacciones propias del atacante, para lo cual el atacante puede producir una firma válida. El doble gasto de las transacciones propias es rentable si al invalidar una transacción el atacante puede obtener un pago o producto a cambio de manera irreversible al eludir el pago en bitcoins.

Examinemos un ejemplo práctico de un ataque del 51%. En el primer capítulo, vimos una transacción entre Alice y Bob por una taza de café. Bob, el dueño del café, está dispuesto a aceptar el pago de tazas de café sin esperar confirmación (acepta una minería de un solo bloque), porque el riesgo de un doble gasto en una taza de café es bajo en comparación con la conveniencia de un servicio rápido al cliente. Esto es similar a la práctica de las cafeterías que aceptan pagos con tarjetas de crédito sin firmas por montos inferiores a $ 25, porque el riesgo de una devolución de tarjeta de crédito es bajo, mientras que el costo de retrasar la transacción para obtener una firma es comparativamente mayor. En contraste, al venderse un artículo más caro con bitcoins, se corre el riesgo de un ataque de doble gasto, donde el comprador transmite una transacción competitiva que gasta las mismas entradas (o UTXOs), cancelando el pago al comerciante. Un ataque de doble gasto puede ocurrir de dos maneras: antes de confirmar una transacción o si el atacante aprovecha una bifurcación de la cadena de bloques para deshacer varios bloques. Un ataque del 51% permite a los atacantes generar un doble gasto de sus propias transacciones en la nueva cadena, deshaciendo así la transacción correspondiente en la cadena anterior.

En nuestro ejemplo, el atacante malicioso: Mallory, va a la galería de Carol y compra una hermosa pintura tríptico que representa a Satoshi Nakamoto como Prometeo. Carol vende pinturas de "El Gran Incendio" por $ 250,000 en bitcoin a Mallory. En lugar de esperar seis o más confirmaciones de la transacción, Carol envuelve y entrega las pinturas a Mallory después de una sola confirmación. Mallory trabaja con un cómplice, Paul, que opera un gran grupo de minería, y el cómplice lanza un ataque del 51% tan pronto como la transacción de Mallory se incluye en un bloque. Paul ordena al grupo de minería que vuelva a minar la misma altura de bloque que el bloque que contiene la transacción de Mallory, reemplazando el pago de Mallory a Carol por una transacción que realiza un doble gasto de la misma entrada que el pago de Mallory. La transacción de doble gasto consume la misma UTXO y la devuelve el valor de esta salida a la misma cartera de Mallory, en lugar de pagarle a Carol, esencialmente permitiendo que Mallory conserve los bitcoins. Luego, Paul dirige al grupo de minería para minar un bloque adicional, a fin de hacer que la cadena que contiene la transacción con el doble gasto sea más larga que la cadena original (causando una bifurcación por debajo del bloque que contiene la transacción de Mallory). Cuando la bifurcación en la cadena de bloques se resuelve a favor de la nueva cadena (más larga), la transacción con el doble gasto reemplaza el pago original a Carol. A Carol ahora le faltan sus tres pinturas y tampoco tiene el pago en bitcoins. A lo largo de toda esta actividad, los participantes del grupo de minería de Paul podrían permanecer felizmente inconscientes del intento de doble gasto, porque minan con mineros automatizados y no pueden monitorear cada transacción o cada bloque.

Para protegerse contra este tipo de ataques, un comerciante que vende artículos de gran valor debe esperar al menos seis confirmaciones antes de entregar el producto al comprador. Alternativamente, el comerciante podría utilizar una cuenta de depósito en custodia multifirma, nuevamente esperando varias confirmaciones después de que la cuenta en custodia sea financiada. Cuantas más confirmaciones transcurran, más difícil será invalidar una transacción con un ataque del 51%. Para los artículos de alto valor, el pago en bitcoins seguirá siendo conveniente y eficiente incluso si el comprador tuviese que esperar 24 horas para la entrega, lo que correspondería a aproximadamente 144 confirmaciones.

Además de un ataque de doble gasto, el otro escenario para un ataque de consenso es negar el servicio a participantes específicos de bitcoin (direcciones específicas de bitcoin). Un atacante con una mayoría del poder de minería puede simplemente ignorar transacciones específicas. Si están incluidas en un bloque minado por otro minero, el atacante puede bifurcar y remover deliberadamente ese bloque, excluyendo nuevamente las transacciones específicas. Este tipo de ataque puede resultar en una denegación de servicio sostenida contra una dirección específica o un conjunto de direcciones mientras el atacante controle la mayor parte del poder de minería.

A pesar de su nombre, la posibilidad de un ataque del 51% en realidad no requiere el 51% de la potencia de hash. De hecho, este tipo de ataque se puede intentar con un porcentaje menor de la potencia de hash. El umbral del 51% es simplemente el nivel en el que el ataque es casi seguro que tenga éxito. Un ataque de consenso es esencialmente una lucha por el siguiente bloque y el grupo "más fuerte" es el que tiene más probabilidades de ganar. Con menor poder de hash, la probabilidad de éxito se reduce, debido a que otros mineros controlan la generación de algunos bloques con su potencia de minería "honesta". Una forma de verlo es que cuanta más potencia de hash tenga un atacante, mayor será la longitud de la bifurcación que pueda crear deliberadamente, mayor será el número de bloques del pasado reciente que pueda invalidar, o mayor será el número de bloques en el futuro que pueda controlar. Existen grupos de investigación de seguridad que han utilizado modelos estadísticos para afirmar que es posible llevar a cabo diversos tipos de ataques de consenso con tan solo el 30% de la potencia de hash.

El aumento masivo de la potencia de hash total ha hecho que probablemente bitcoin sea inmune a los ataques de un solo minero. No hay manera posible de que un minero en solitario pueda controlar más de un pequeño porcentaje de la potencia de minería total. Sin embargo, la centralización del control causado por los pools de minería ha introducido el riesgo de ataques con afán de lucro por parte de un operador de pool de minería. El operador del pool en un pool gestionado controla la construcción de bloques candidatos y también controla qué transacciones se incluyen. Esto le da al operador del pool la facultad de excluir transacciones o de introducir transacciones de doble gasto. Si ese abuso de poder se hace de una manera limitada y sutil, un operador de pool posiblemente podría pasar desapercibido y beneficiarse de un ataque de consenso.

Sin embargo, no todos los atacantes estarán motivados por las ganancias. Un escenario de ataque potencial es cuando un atacante intenta interrumpir la red de bitcoin sin la posibilidad de beneficiarse de dicha interrupción. Un ataque malicioso destinado a paralizar bitcoin requeriría de una inversión enorme y una planificación encubierta, pero posiblemente podría ser lanzado por un atacante bien financiado, muy probablemente patrocinado por algún estado. Alternativamente, un atacante bien financiado podría atacar el consenso de bitcoin acumulando simultáneamente hardware de minería, comprometiendo a los operadores de grupos y atacando a otros grupos con denegación de servicio. Todos estos escenarios son teóricamente posibles, pero cada vez menos prácticos a medida que la potencia de hash general de la red bitcoin continúa creciendo exponencialmente.

Sin lugar a dudas, un ataque contra el consenso realmente grave erosionaría la confianza en bitcoin a corto plazo, posiblemente causando una disminución significativa de sus precios. Sin embargo, la red y el software de bitcoin están en constante evolución, por lo que los ataques de consenso se encontrarían con contramedidas inmediatas por parte de la comunidad de bitcoin, haciendo que bitcoin sea más robusto.

Cambiando las Reglas de Consenso

Las reglas de consenso determinan la validez de las transacciones y de los bloques. Estas reglas son la base para la colaboración entre todos los nodos de bitcoin y son responsables de la convergencia de todas las perspectivas locales en una única cadena de bloques consistente en toda la red.

Si bien las reglas de consenso son invariables a corto plazo y deben ser consistentes en todos los nodos, no son invariables a largo plazo. Para evolucionar y desarrollar el sistema bitcoin, las reglas tienen que cambiar de vez en cuando para acomodar nuevas características, mejoras o correcciones de errores. Sin embargo, a diferencia del desarrollo de software tradicional, las actualizaciones a un sistema de consenso son mucho más difíciles y requieren coordinación entre todos los participantes.

Bifurcaciones Fuertes

En Bifurcaciones de la Cadena de Bloques se pudo observar cómo funciona la red bitcoin cuando esta sufre una breve divergencia, con dos partes de la red siguiendo dos ramas diferentes de la cadena de bloques por un corto tiempo. Vimos cómo este proceso ocurre de manera natural, como parte del funcionamiento normal de la red y cómo la red se redirige hacia una cadena de bloques unificada después que se han minado uno o más bloques.

Hay otro escenario en el que la red puede experimentar una divergencia en dos cadenas subsecuentes: un cambio en las reglas de consenso. Este tipo de bifurcación se denomina *bifurcación fuerte*, porque después de tal bifurcación, la red no se vuelve a dirigirse hacia una sola cadena. En cambio, las dos cadenas evolucionan independientemente. Las bifurcaciones fuertes ocurren cuando parte de la red está operando bajo un conjunto diferente de reglas de consenso que el resto de la red. Esto puede ocurrir debido a un error en el software o debido a un cambio deliberado en la implementación de las reglas de consenso.

Las bifurcaciones fuertes se pueden usar para cambiar las reglas de consenso, pero requieren de la coordinación de todos

los participantes en el sistema. Los nodos que no se actualizan a las nuevas reglas de consenso no pueden seguir participando en el mecanismo de consenso y se ven obligados a seguir una cadena separada en el momento de la bifurcación fuerte. Por lo tanto, se puede considerar que un cambio introducido por una bifurcación fuerte no es "compatible con versiones anteriores", ya que los nodos no actualizados ya no pueden procesar las nuevas reglas de consenso del sistema.

Examinemos la mecánica de una bifurcación fuerte con un ejemplo específico.

Una cadena de bloques con bifurcaciones muestra una cadena de bloques con dos bifurcaciones. En la altura de bloque igual a 4, se produce una bifurcación de un bloque. Este es el tipo de bifurcación espontánea que vimos en Bifurcaciones de la Cadena de Bloques. Cuando la minería produce al bloque 5, la red converge nuevamente en una sola cadena y se resuelve la bifurcación.

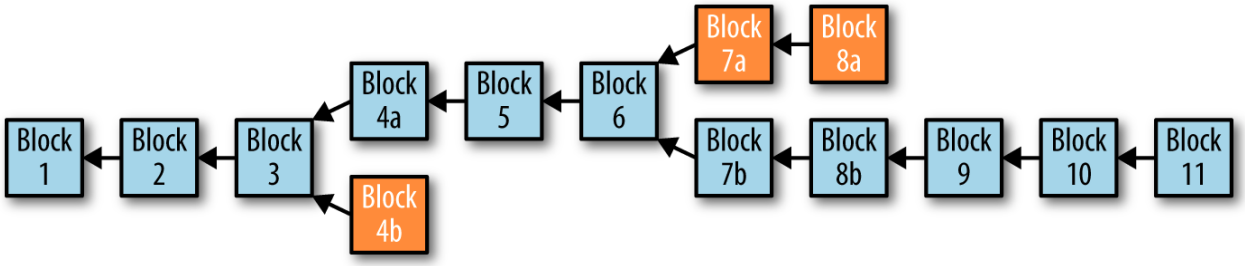


Figure 72. Una cadena de bloques con bifurcaciones

Más tarde, sin embargo, a la altura de bloque 6, se produce una bifurcación fuerte. Supongamos que se ha publicado una nueva versión del cliente con un cambio en las reglas de consenso. Comenzando en la altura de bloque 7, los mineros que ejecutan esta nueva implementación aceptarán un nuevo tipo de firma digital, a la que llamaremos firma "Smores", que no está basada en ECDSA. Inmediatamente después, un nodo que ejecuta la nueva implementación crea una transacción que contiene una firma Smores y un minero con el software actualizado mina el nuevo bloque 7b que contiene esta transacción.

Cualquier nodo o minero que no haya actualizado su software para validar las firmas Smores ahora no puede procesar el bloque 7b. Desde su perspectiva, tanto la transacción que contenía una firma Smores como el bloque 7b que contenía esa transacción no son válidos, porque los están evaluando en base a las viejas reglas de consenso. Estos nodos rechazarán la transacción y el bloque y no los propagarán. Cualquier minero que esté usando las reglas antiguas no aceptará el bloque 7b y continuará minando un bloque candidato cuyo padre es el bloque 6. De hecho, los mineros que usan las reglas viejas no pueden ni siquiera recibir el bloque 7b si todos los nodos a los que están conectados están también obedeciendo las viejas reglas y por lo tanto no estarán propagando el bloque. Llegado el momento, lograrán minar el bloque 7a, que es válido según las reglas anteriores y no contiene ninguna transacción con firmas Smores.

Las dos cadenas continuarán divididas desde este punto. Los mineros en la cadena "b" continuarán aceptando y minando transacciones que contengan firmas Smores, mientras que los mineros en la cadena "a" continuarán ignorando estas transacciones. Incluso si el bloque 8b no contuviese ninguna transacción firmada por Smores, los mineros en la cadena "a" no pueden procesarlo. Para ellos, parece ser un bloque huérfano, ya que su padre "7b" no se reconoce como un bloque válido.

Bifurcaciones Fuertes: Software, Red, Minería y Cadena

Para los desarrolladores de software, el término "bifurcación" tiene otro significado, lo que agrega confusión al término "bifurcación fuerte". En el software de código abierto, una bifurcación ocurre cuando un grupo de desarrolladores elige seguir una hoja de ruta diferente para el software y comienzan una a desarrollar una implementación diferente y competitiva de un proyecto de código abierto. Ya hemos discutido dos circunstancias que conducirán a una bifurcación fuerte: un error de software en las reglas de consenso y una modificación deliberada de las reglas de consenso. En el caso de un cambio deliberado en las reglas de consenso, una bifurcación de software precede a la bifurcación fuerte. Sin embargo, para que se produzca este tipo de bifurcación fuerte, se debe desarrollar, adoptar y lanzar una nueva implementación de software para nuevas reglas de consenso.

Los ejemplos de bifurcaciones de software que han intentado cambiar las reglas de consenso incluyen Bitcoin XT, Bitcoin Classic y, más recientemente, Bitcoin Unlimited. Sin embargo, ninguna de estas bifurcaciones de software obtuvo el resultado de una bifurcación fuerte. Si bien una bifurcación de software es una condición previa necesaria, no es en sí misma suficiente para que ocurra una bifurcación fuerte. Para que ocurra una bifurcación fuerte, la implementación

competitiva debe ser adoptada y las nuevas reglas deben ser activadas por los mineros, billeteras y nodos intermediarios. Por el contrario, existen numerosas implementaciones alternativas del Cliente Principal de Bitcoin, e incluso las bifurcaciones de software, que no cambian las reglas de consenso y a excepción de errores de software, pueden coexistir en la red e operar conjuntamente sin causar una bifurcación fuerte.

Las reglas de consenso pueden diferir de manera obvia y explícita, en torno a la validación de transacciones o bloques. Las reglas también pueden diferir en formas más sutiles, en la implementación de las reglas de consenso, en la medida que se aplican a los scripts de bitcoin o normas criptográficas primitivas, como las firmas digitales. Finalmente, las reglas de consenso pueden diferir de maneras imprevistas debido a restricciones de consenso implícitas impuestas por las limitaciones del sistema o detalles de implementación. Se vio un ejemplo de esto último en la bifurcación fuerte que no fue prevista durante la actualización de Bitcoin Core 0.7 a 0.8, que fue causada por una limitación en la implementación de “Berkeley DB” utilizada para almacenar bloques.

Conceptualmente, podemos pensar que una bifurcación fuerte se desarrolla en cuatro etapas: una bifurcación de software, una bifurcación en la red, una bifurcación de minería y una bifurcación de la cadena.

El proceso comienza cuando los desarrolladores crean una implementación alternativa del cliente, con reglas de consenso modificadas.

Cuando esta implementación bifurcada se implementa en la red, un cierto porcentaje de mineros, usuarios de billeteras y nodos intermedios pueden adoptar y ejecutar esta implementación. Una bifurcación resultante dependerá de si las nuevas reglas de consenso se aplican a los bloques, a las transacciones o algún otro aspecto del sistema. Si las nuevas reglas de consenso se refieren a las transacciones, entonces una aplicación de billetera que crea una transacción bajo las nuevas reglas puede precipitar una bifurcación de red, seguida de una bifurcación fuerte cuando la transacción se mina en un bloque. Si las nuevas reglas son referentes a los bloques, entonces el proceso de bifurcación fuerte comenzará cuando se mine un bloque bajo las nuevas reglas.

Primero, la red se bifurcará. Los nodos basados en la implementación original de las reglas de consenso rechazarán cualquier transacción y bloque que se cree bajo las nuevas reglas. Además, los nodos que siguen las reglas de consenso originales prohibirán temporalmente y se desconectarán de cualquier nodo que les envíe estas transacciones y bloques no válidos. Como resultado, la red se dividirá en dos: los nodos antiguos solo permanecerán conectados a los nodos antiguos y los nodos nuevos solo se conectarán a los nodos nuevos. Una sola transacción o bloque basado en las nuevas reglas se extenderá por la red y dará como resultado la partición en dos redes.

Una vez que un minero que usa las nuevas reglas mina un bloque, el poder de minería y la cadena también se bifurcarán. Los nuevos mineros minarán en la parte superior del nuevo bloque, mientras que los antiguos mineros trabajarán sobre una cadena separada según las viejas reglas. La red dividida hará que los mineros que operan con reglas de consenso separadas probablemente no reciban los bloques de la otra red, ya que estarán conectados a dos redes separadas.

Mineros y Dificultad Divergentes

A medida que los mineros divergen en la minería de dos cadenas diferentes, la potencia de hash también se divide entre las cadenas. El poder de minería se puede dividir en cualquier proporción entre las dos cadenas. Las nuevas reglas solo pueden ser seguidas por una minoría o por la gran mayoría del poder de minería.

Supongamos, por ejemplo, una división del 80%–20%, con la mayoría del poder de minería aplicando las nuevas reglas de consenso. Supongamos también que la bifurcación ocurre inmediatamente después de un período de reorientación.

Las dos cadenas heredarían cada una la dificultad del reciente período de reorientación. Las nuevas reglas de consenso dispondrían del 80% del poder de minería que estuvo a disposición de la red previamente. Desde la perspectiva de esta cadena, el poder de minería ha disminuido repentinamente en un 20% con respecto al período anterior. Los bloques se encontrarán en promedio cada 12,5 minutos, lo que representa la disminución del 20% en el poder de minería disponible para extender esta cadena. Esta tasa de producción de bloques continuará (salvo cualquier cambio en la potencia de hash) hasta que se minen los próximos 2016 bloques, lo que tomará aproximadamente 25.200 minutos (a 12,5 minutos por bloque), o 17 días y medio. Después de estos 17,5 días, ocurrirá un reajuste y la dificultad se calibrará (se reducirá en un 20%) para producir bloques de 10 minutos nuevamente, en función de la cantidad reducida de la potencia de hash en esta cadena.

Pero la cadena minoritaria, que ahora realiza una minería bajo las viejas reglas con solo el 20% de la potencia de hash, enfrentará una tarea mucho más difícil. En esta cadena, los bloques ahora se extraerán, en promedio cada 50 minutos. La dificultad no se ajustará en esta cadena por los próximos 2016 bloques, lo que tomará 100.800 minutos, o

aproximadamente 10 semanas para minarlos. Suponiendo una capacidad fija por bloque, esto también dará como resultado una reducción de la capacidad de transacción por un factor de 5, ya que hay menos bloques por hora disponibles para registrar las transacciones.

Bifurcaciones Fuertes por Disputa

Este es el inicio del desarrollo de software de consenso. Así como el desarrollo de código abierto cambió los métodos y productos del software y creó nuevas metodologías, nuevas herramientas y nuevas comunidades a su paso, el desarrollo de software de consenso también representa una nueva frontera en las ciencias informáticas. De los debates, experimentos y tribulaciones de la hoja de ruta del desarrollo de bitcoin, veremos surgir nuevas herramientas de desarrollo, nuevas prácticas, metodologías y comunidades de desarrollo.

Las bifurcaciones fuertes se consideran riesgosas porque obligan a una minoría a actualizarse o a permanecer en una cadena minoritaria. Muchos ven el riesgo de dividir todo el sistema en dos sistemas competitivos como un riesgo inaceptable. Como resultado, muchos desarrolladores son reacios a utilizar el mecanismo de bifurcación fuerte para implementar actualizaciones a las reglas de consenso, a menos que exista un apoyo casi unánime de toda la red. Cualquier propuesta de bifurcación fuerte que no tenga soporte casi unánime, se considera demasiado "contenciosa" para intentarla sin arriesgarse a una partición del sistema.

El tema de las bifurcaciones fuertes es muy controvertido en la comunidad de desarrolladores de bitcoin, especialmente en la medida que se relaciona con cualquier cambio propuesto a las reglas de consenso que controlan el límite máximo del tamaño del bloque. Algunos desarrolladores se oponen a cualquier forma de bifurcación fuerte, ya que las consideran demasiado arriesgadas. Otros ven en el mecanismo de bifurcación fuerte una herramienta esencial para actualizar las reglas de consenso de una manera que evite la "deuda técnica" y proporcione una separación quirúrgica con el pasado. Finalmente, algunos desarrolladores ven a las bifurcaciones fuertes como un mecanismo que debería usarse raramente, con mucha planificación anticipada y solo bajo un consenso casi unánime.

Ya hemos visto la aparición de nuevas metodologías para abordar los riesgos de las bifurcaciones fuertes. En la siguiente sección, veremos las bifurcaciones suaves y los métodos BIP-34 y BIP-9 para la señalización y activación de modificaciones por consenso.

Bifurcaciones Suaves

No todos los cambios en las reglas de consenso causan una bifurcación fuerte. Solo los cambios de consenso que son incompatibles con la versión anterior causan una bifurcación. Pero si la modificación se implementa de tal manera que un cliente no actualizado todavía pueda ver transacciones o bloques que cumplan con las reglas viejas como válidos, el cambio puede ocurrir sin que se dé una bifurcación.

El término *bifurcación suave* se introdujo para distinguir este método de actualizaciones de una "bifurcación fuerte". En la práctica, una bifurcación suave no es una bifurcación en absoluto. Una bifurcación suave es un cambio compatible con las reglas de consenso que permite a los clientes no actualizados continuar operando en consenso con las nuevas reglas.

Un aspecto de las bifurcaciones suaves que no es inmediatamente obvio, es que las actualizaciones de una bifurcación suave solo pueden usarse para restringir las reglas de consenso, no para expandirlas. Para ser compatibles con las nuevas reglas, las transacciones y los bloques creados bajo estas nuevas reglas también deben ser válidos bajo las viejas reglas, pero no al revés. Las nuevas reglas solo pueden limitar lo que es válido; de lo contrario, desencadenarían una bifurcación fuerte cuando se rechacen estos elementos según las viejas reglas.

Las bifurcaciones suaves se pueden implementar de varias maneras—el término no especifica un método en particular, sino un conjunto de métodos que tienen una cosa en común: no requieren que todos los nodos se actualicen o que sean forzados a salir del consenso si no se actualizan.

Bifurcaciones suaves que redefinieron códigos operativos NOP

Varias bifurcaciones suaves se han implementado en bitcoin, en base a la reinterpretación de los códigos de operativos “NOP”. Los Scripts Bitcoin tenían diez códigos operativos reservados para usos futuros, desde el NOP1 hasta el NOP10. Según las reglas de consenso, la presencia de estos códigos operativos en un script se interpreta como la de operadores de potencia-nula, lo que significa que no tienen ningún efecto. La ejecución continúa después del código de operación NOP como si ése código no estuviera allí.

Por lo tanto, una bifurcación suave puede modificar la semántica de un código NOP para darle un nuevo significado. Por ejemplo, la propuesta de mejoras a bitcoin BIP-65 (que versa sobre el comando CHECKLOCKTIMEVERIFY) reinterpretó el

código de operaciones NOP2. Los clientes que implementan el BIP-65 interpretan NOP2 como OP_CHECKLOCKTIMEVERIFY e imponen una regla de consenso de bloqueo temporal absoluto en las UTXOs que contienen este código de operación en sus scripts de bloqueo. Este cambio es una bifurcación suave porque una transacción que es válida bajo el BIP-65 también será válida en cualquier cliente que no esté implementando (o sea ignorante del) BIP-65. Para los clientes antiguos, el script contiene un código NOP, que se ignora.

Otras formas de actualización por bifurcación suave

La reinterpretación de los códigos de operación NOP no solo fue una opción planificada sino también un obvio mecanismo para lograr mejoras en el consenso. Recientemente, sin embargo, se introdujo otro mecanismo de bifurcación suave que no se basa en los códigos de operación NOP y que permitió un tipo muy específico de cambios en el consenso. Esto se examina con más detalle en Segregated Witness (Testigos Segregados). La mejora del testigo segregado o "Segwit" es un cambio arquitectónico en la estructura de una transacción, que mueve al script de desbloqueo (el "testigo") desde el interior de la transacción, a una estructura de datos externa (segregándola). Segwit se imaginó inicialmente como una actualización de bifurcación fuerte, ya que modificó una estructura fundamental (a la transacción misma). En noviembre de 2015, un desarrollador que trabajaba en Bitcoin Core propuso un mecanismo por el cual Segwit podría introducirse como una bifurcación suave. El mecanismo utilizado para esto es una modificación del script de bloqueo de las UTXOs creadas bajo las nuevas reglas de Segwit, de modo que los clientes viejos verían al script de bloqueo como un acertijo tonto, redimible por cualquier script de desbloqueo. Como resultado, se puede introducir el nuevo estándar Segwit sin requerir que cada nodo se actualice o se separe de la cadena principal: una bifurcación suave.

Es probable que existan otros mecanismos, aún por descubrir, mediante los cuales se pueden realizar aún más actualizaciones de una manera compatible con las versiones anteriores como una bifurcación suave.

Críticas a las Bifurcaciones Suaves

Las bifurcaciones suaves basadas en los códigos de operación NOP son, relativamente hablando, muy poco conflictivas. Los códigos de operación NOP se colocaron en las listas de Comandos Scripts de Bitcoin, con el objetivo explícito de permitir actualizaciones armoniosas en la red.

Sin embargo, a muchos desarrolladores les preocupa que otros métodos de actualización por bifurcación suave hagan concesiones inaceptables. Las críticas comunes a los cambios de bifurcación suave incluyen:

Deuda técnica

Debido a que las bifurcaciones suaves son más complejas técnicamente que una actualización por bifurcación fuerte, éstas introducen *deudas técnicas*, un término que se refiere a aumentar el costo futuro del mantenimiento del código debido a las concesiones de diseño realizadas en el pasado. La complejidad del código a su vez aumenta las probabilidades de errores y las vulnerabilidades de seguridad.

Validación flexibilizada

Los clientes no modificados ven las transacciones como válidas, sin evaluar las reglas de consenso modificadas. En efecto, los clientes no modificados no están validando las transacciones usando la gama completa de las nuevas reglas de consenso, ya que son ciegos a las nuevas reglas. Esto se aplica a las actualizaciones basadas en NOP, así como a otras actualizaciones de bifurcación suave.

Actualizaciones irreversibles

Debido a que las bifurcaciones suaves crean transacciones con restricciones de consenso adicionales, en la práctica se convierten en actualizaciones irreversibles. Si una actualización de bifurcación suave se revirtiese después que se hubiesen activado, cualquier transacción creada bajo las nuevas reglas podría resultar en una pérdida de fondos bajo las viejas reglas. Por ejemplo, si una transacción CLTV, se evalúa según las reglas anteriores, no hay restricción de bloqueo temporal y puede gastarse en cualquier momento. Por lo tanto, los críticos sostienen que una bifurcación suave que ha fallado y que tuvo que ser revertida debido a un error de software, casi con seguridad conducirá a la pérdida de fondos.

Señalizando a la Bifurcación Suave con la Versión del Bloque

Dado que las bifurcaciones suaves permiten que los clientes no actualizados continúen operando dentro del consenso, el mecanismo para "activar" una bifurcación suave se realiza a través de los mineros que anuncian el nivel de disposición: la mayoría de los mineros deben estar de acuerdo en que están preparados y dispuestos a hacer cumplir las nuevas reglas de consenso. Para coordinar sus acciones, existe un mecanismo de señalización que les permite mostrar su apoyo a un cambio a las reglas de consenso. Este mecanismo se introdujo con la activación de la mejora de bitcoin BIP-34 en marzo de

2013 y se reemplazó por la activación del BIP-9 en julio de 2016.

BIP-34: Señalización y Activación

La primera implementación de BIP-34, utilizó el campo de versión de bloque para permitir a los mineros señalar su nivel de preparación en pos del cambio de una regla de consenso específica. Antes de BIP-34, la versión de bloque fue establecida en "1" por *convenio* mas no establecida por *consenso*.

BIP-34 definió un cambio en las reglas de consenso que requería que el campo coinbase (la entrada) de la transacción coinbase contuviera la altura del bloque. Antes de BIP-34, la entrada de la transacción coinbase podía contener cualquier información arbitraria que los mineros decidieran incluir. Después de la activación de BIP-34, los bloques válidos tenían que contener un valor específico de altura de bloque al comienzo de la transacción coinbase e identificarse con un número de versión mayor o igual a "2".

Para señalar tanto el cambio como la activación de BIP-34, los mineros establecieron la versión de bloque en "2", en lugar de "1". Esto no hizo que los bloques de la versión "1" fueran inválidos de inmediato. Pero una vez activado el cambio, los bloques de la versión "1" dejarían de ser válidos y todos los bloques de la versión "2" tendrían que contener la altura de bloque al comienzo de la transacción coinbase para ser válidos.

BIP-34 definió un mecanismo de activación de dos pasos, basado en una ventana móvil de 1000 bloques. Un minero señalaría su predisposición individual para BIP-34 construyendo bloques con el número "2", como número de versión. Estrictamente hablando, estos bloques aún no tenían que cumplir con la nueva regla de consenso de incluir la altura de bloque en la transacción coinbase porque la regla de consenso aún no se había activado. Las reglas de consenso se activan en dos etapas:

- Si el 75% (750 de los 1000 bloques más recientes) están marcados con la versión "2", entonces los bloques de la versión "2" deben contener la altura de bloque en la transacción coinbase o se rechazarán como no válidos. Los bloques de la versión "1" todavía son aceptados por la red y no necesitan contener la altura de bloque. Las viejas y nuevas reglas de consenso coexisten durante este período.
- Cuando el 95% (950 de los 1000 bloques más recientes) son de la versión "2", los bloques de la versión "1" ya no se consideran válidos. Los bloques de la versión "2" son válidos solo si contienen la altura de bloque en transacción coinbase (según el umbral anterior). A partir de entonces, todos los bloques deben cumplir con las nuevas reglas de consenso, y todos los bloques válidos deben contener la altura de bloque en la transacción coinbase.

Después de una señalización y activación exitosas bajo las reglas BIP-34, este mecanismo se usó dos veces más para activar bifurcaciones suaves:

- BIP-66 La codificación estricta DER para firmas se activó mediante la señalización según el estilo BIP-34, pero con un número de versión en los bloques igual a "3", haciendo inválida la versión de bloques número "2".
- BIP-65 CHECKLOCKTIMEVERIFY se activó mediante la señalización según el estilo BIP-34 con una versión en bloque "4" e invalidando la versión "3" en bloques.

Después de la activación de BIP-65, el mecanismo de señalización y activación de BIP-34 fue retirado y reemplazado por el mecanismo de señalización de BIP-9 que se describe a continuación.

El estándar se define en BIP-34 (Block v2, Height in Coinbase).

Señalización y Activación Según BIP-9

El mecanismo utilizado por BIP-34, BIP-66 y BIP-65 logró activar tres bifurcaciones suaves. Sin embargo, fue reemplazado porque tenía varias limitaciones:

- Al usar el valor entero de la versión de bloque, solo se podía activar una bifurcación suave a la vez, por lo que se requería coordinación entre las propuestas de bifurcación suave y un acuerdo sobre su cómo decidir las prioridades y la secuencia.
- Además, debido a que la versión en el bloque se incrementó, el mecanismo no ofreció una forma directa de rechazar un cambio y luego proponer uno diferente. Si los clientes antiguos todavía se estaban ejecutando, podrían confundir la señalización de un nuevo cambio como también podrían señalizar un cambio previamente rechazado.
- Cada nuevo cambio reducía irrevocablemente las versiones de bloque disponibles para futuros cambios.

Se propuso BIP-9 para superar estos desafíos y mejorar la velocidad y la facilidad de implementar cambios futuros.

BIP-9 interpreta la versión de bloque como un campo de bits en lugar de un entero. Debido a que la versión de bloque se usó originalmente como un entero para las versiones de la 1 a la 4, solo quedarían 29 bits disponibles para usarse como campo de bits. Esto deja 29 bits que se pueden utilizar para indicar de forma independiente y simultánea la preparación para 29 propuestas diferentes.

BIP-9 también establece un tiempo máximo para la señalización y activación. De esta manera, los mineros no necesitan hacer señales para siempre. Si una propuesta no se activa dentro del período indicado como TIMEOUT (parámetro definido en la propuesta), la propuesta se considera rechazada. La propuesta se puede volver a enviar para señalizar con un bit diferente, renovando el período de activación.

Además, después de que el TIMEOUT ha expirado y una característica ha sido activada o rechazada, el bit de señalización puede reutilizarse para otra característica sin confusión. Por lo tanto, se pueden señalar hasta 29 cambios en paralelo y después del TIMEOUT los bits se pueden "reciclar" para proponer nuevos cambios.

NOTE

Si bien los bits de señalización pueden reutilizarse o reciclarse, siempre que el período de votación no se superponga, los autores de BIP-9 recomiendan que los bits se reutilicen solo cuando sea necesario; podría ocurrir un comportamiento inesperado debido a errores en el software anterior. En otras palabras, no deberíamos esperar ver la reutilización de un bit hasta que todos los 29 bits se hayan usado una vez.

Los cambios propuestos se identifican mediante una estructura de datos que contiene los siguientes campos:

nombre

Una breve descripción utilizada para distinguir diferentes propuestas. Muy a menudo, el BIP describe la propuesta, como "bipN", donde N es el número de BIP.

bit

De 0 a 28, el bit en la versión de bloque que usan los mineros para indicar la aprobación de esta propuesta.

starttime

Es el momento (basado en “Median Time Past”, o MTP) en que comienza la señalización, después de lo cual el valor del bit se interpreta como señal de preparación para la propuesta.

endtime

Es el momento (basado en MTP) después del cual el cambio se considera rechazado si no ha alcanzado el umbral de activación.

A diferencia de BIP-34, BIP-9 cuenta la señalización de activación en intervalos completos en función del recalculo de dificultad en el período de los últimos 2016 bloques. Para cada período de recalculo, si la suma de los bloques que señalan una propuesta supera el 95% (1916 de 2016), la propuesta se activará un período de recalculo más tarde.

BIP-9 ofrece un diagrama de estado para cada propuesta para ilustrar las diversas etapas y transiciones de una propuesta, como se muestra en Diagrama de transición de estado del protocolo BIP-9.

Las propuestas comienzan en el estado DEFINED, una vez que sus parámetros son conocidos (definidos) en el software bitcoin. Para los bloques con MTP posteriores de la hora de inicio, el estado de la propuesta pasa a STARTED. Si se supera el umbral de votación dentro de un período de recálculo y no se ha excedido el tiempo de espera del protocolo, el estado de la propuesta pasa a LOCKED_IN. Tras un período de recálculo posterior, la propuesta se convierte en ACTIVE. Las propuestas permanecen en el estado ACTIVE perpetuamente una vez que se alcanza ese estado. Si transcurre el tiempo de espera antes de alcanzar el umbral de votación, el estado de la propuesta cambia a FAILED, lo que indica una propuesta rechazada. Las propuestas FAILED permanecen en ese estado perpetuamente.

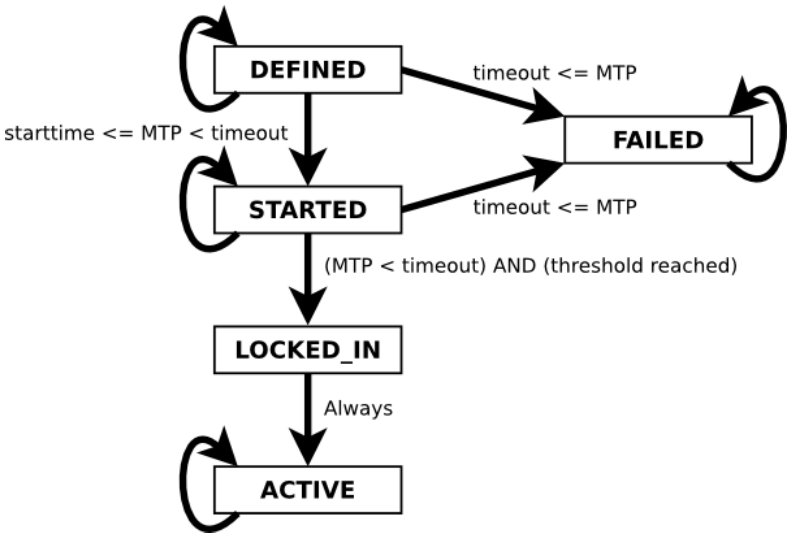


Figure 73. Diagrama de transición de estado del protocolo BIP-9

BIP-9 se implementó por primera vez para la activación de CHECKSEQUENCEVERIFY y los BIP asociados (los BIP número 68, 112 y 113). La propuesta, que se denominó "csv", se activó con éxito en julio de 2016.

El estándar se define en BIP-9 (bits de versión con tiempos de espera y retraso).

Desarrollo de Software de Consenso

El software de consenso continúa evolucionando y hay mucha discusión sobre los diversos mecanismos para cambiar las reglas de consenso. Por su propia naturaleza, bitcoin establece una barrera muy alta en lo tocante a la coordinación y el consenso para los cambios. Como sistema descentralizado, no tiene ninguna "autoridad" que pueda imponer su voluntad a los participantes de la red. El poder se distribuye entre múltiples grupos, como mineros, desarrolladores principales, desarrolladores de billeteras, casas de cambio, comerciantes y usuarios finales. Ninguna de estas agrupaciones puede tomar decisiones unilateralmente. Por ejemplo, si bien los mineros pueden cambiar teóricamente las reglas por mayoría simple (51%), están limitados por el consentimiento de los otros constituyentes. Si actúan unilateralmente, el resto de los participantes simplemente pueden negarse a seguirlos, manteniendo la actividad económica en una cadena minoritaria. Sin actividad económica (transacciones, comerciantes, billeteras, casas de cambio), los mineros acuñarían una moneda sin valor con bloques vacíos. Esta distribución en el poder significa que todos los participantes deben coordinarse, o no se pueden hacer cambios. El status quo es representado por un estado estable de este sistema con solo unos pocos cambios posibles si existe un fuerte consenso de una gran mayoría. El umbral del 95% para las bifurcaciones suaves refleja esta realidad.

Es importante reconocer que no existe una solución perfecta para el desarrollo en consenso. Tanto las bifurcaciones fuertes como las bifurcaciones suaves implican el hacer diversas concesiones. Para algunos tipos de cambios, las bifurcaciones suaves pueden ser una mejor opción; Para otros, las bifurcaciones fuertes pueden ser una mejor opción. No hay una elección perfecta; Ambas conllevan riesgos. La única característica constante del desarrollo de software en consenso es que el cambio es difícil y el consenso obliga a comprometerse.

Algunos ven esto como una debilidad de los sistemas de consenso. Con el tiempo, Ud. puede llegar a verlos como yo, como la mayor fortaleza del sistema.

Seguridad de Bitcoin

Asegurar bitcoin es un reto porque bitcoin no es una referencia abstracta a un valor, como un saldo en una cuenta bancaria. Bitcoin es muy parecido a dinero en efectivo u oro digital. Probablemente hayas escuchado la expresión, "la posesión es nueve décimas partes de la ley." Pues bien, en bitcoin, la posesión es diez décimas partes de la ley. La posesión de las llaves para desbloquear el bitcoin es equivalente a la posesión de dinero en efectivo o un trozo de metal precioso. Es posible perderlo, olvidarlo, ser robado o accidentalmente dar una cantidad incorrecta a alguien. En cada uno de estos casos, los usuarios no tienen ningún recurso, al igual que si se le cayera dinero en efectivo en una acera pública.

Sin embargo, bitcoin tiene capacidades que el efectivo, el oro y las cuentas bancarias no tienen. Una cartera bitcoin, que contiene las llaves, se puede copiar como cualquier archivo. Se puede almacenar en múltiples copias, incluso escrito en papel como copia impresa de seguridad. No se puede hacer "copia de seguridad" de las cuentas de efectivo, oro, o bancarias. Bitcoin es lo suficientemente diferente de todo lo que ha venido antes como para que debamos pensar en la seguridad de bitcoin también de una forma innovadora.

Principios de Seguridad

El principio básico de bitcoin es la descentralización y tiene importantes implicaciones para la seguridad. Un modelo centralizado, como un banco tradicional o una red de pagos, mantiene a los malos actores fuera del sistema mediante el control de acceso y la investigación de antecedentes. En comparación, un sistema descentralizado como bitcoin traslada la responsabilidad y el control a los usuarios. Dado que la seguridad de la red se basa en la Prueba-de-Trabajo, no en el control de acceso, la red puede ser abierta y no se requiere encriptación para el tráfico de bitcoin .

En una red de pago tradicional, como un sistema de tarjetas de crédito, el pago es de composición abierta, ya que contiene el identificador privado del usuario (el número de tarjeta de crédito). Después de la carga inicial, cualquier persona con acceso al identificador puede "coger" los fondos y cargarlos del propietario una y otra vez. De este modo, la red de pagos tiene que protegerse con encriptación extremo a extremo y debe asegurarse de que no haya espías o intermediarios que puedan comprometer el tráfico de pagos, en tránsito o cuando se almacena (en reposo). Si un actor malintencionado consigue acceso al sistema, puede comprometer las transacciones en curso *y* los componentes de pago que pueden utilizarse para crear nuevas transacciones. Peor aún, cuando se ven comprometidos los datos de clientes, los clientes están expuestos a robo de identidad y deben tomar medidas para evitar el uso fraudulento de las cuentas comprometidas.

Bitcoin es totalmente diferente. Una transacción bitcoin autoriza solamente un valor específico a un destinatario específico y no puede ser falsificado o modificado. No revela ninguna información privada, como la identidad de las partes, y no puede ser utilizado para autorizar pagos adicionales. Por lo tanto, una red de pago bitcoin no necesita ser encriptada o protegida contra escuchas. De hecho, puedes difundir transacciones bitcoin a través de un canal abierto al público, como WiFi o Bluetooth, sin perder seguridad.

El modelo de seguridad descentralizada de bitcoin pone mucho poder en manos de los usuarios. Con ese poder viene la responsabilidad de mantener la confidencialidad de las llaves. Para la mayoría de los usuarios eso no es fácil de hacer, sobre todo en los dispositivos informáticos de propósito general como smartphones o portátiles conectados a internet. Aunque el modelo descentralizado de bitcoin impide el tipo de exposición generalizada de las tarjetas de crédito, muchos usuarios no son capaces de asegurar adecuadamente sus llaves y pueden ser hackeados, uno por uno.

Desarrollando Sistemas Bitcoin de Forma Segura

El principio más importante para los desarrolladores de bitcoin es la descentralización. La mayoría de los desarrolladores estarán familiarizados con modelos de seguridad centralizados y podrían estar tentados a aplicar estos modelos a sus aplicaciones bitcoin, con resultados desastrosos.

La seguridad de bitcoin se basa en el control descentralizado de llaves y en la validación independiente de las transacciones por los mineros. Si quieres aprovechar la seguridad de bitcoin, es necesario que te asegures de que permaneces en el modelo de seguridad de bitcoin. En términos simples: no dejar el control de las llaves lejos de los usuarios y no llevar las transacciones fuera de la cadena de bloques.

Por ejemplo, muchas de las primeras casas de cambio de bitcoin concentraban todos los fondos de los usuarios en una sola cartera "caliente" con las llaves almacenadas en un solo servidor. Ese diseño quita el control a los usuarios y centraliza el control de las llaves en un solo sistema. Muchos de estos sistemas han sido hackeados, con consecuencias desastrosas para sus clientes.

Otro error común es llevar las transacciones "fuera de la cadena de bloques" en un esfuerzo equivocado para reducir las comisiones de transacción o para acelerar el procesamiento de transacciones. Un sistema "fuera de la cadena de bloques" registrará las transacciones en un libro contable interno, centralizado y solo se sincronizará ocasionalmente con la cadena de bloques de bitcoin. Esta práctica, sustituye la seguridad descentralizada de bitcoin con un enfoque cerrado y centralizado. Cuando las transacciones están "fuera de la cadena de bloques", los libros contables centralizados que no estén adecuadamente asegurados pueden ser falsificados, desviando fondos y agotando las reservas de manera desapercibida.

A menos que estés dispuesto a invertir fuertemente en la seguridad operacional, con múltiples capas de control de acceso, y en auditorias (como hacen los bancos tradicionales), deberías pensarlo cuidadosamente antes de llevar fondos fuera del contexto de seguridad descentralizada de bitcoin. Incluso si tienes los fondos y la disciplina para implementar un modelo de seguridad robusto, ese diseño tan solo replica el frágil modelo de las redes financieras tradicionales, plagadas por el robo de identidad, la corrupción y la malversación de fondos. Para aprovechar el modelo único de seguridad descentralizada de bitcoin, hay que evitar la tentación de arquitecturas centralizadas que podrían hacerte sentirte cómodo, pero que en última instancia, subvierten la seguridad de bitcoin.

La Raíz de la Confianza

La arquitectura de seguridad tradicional se basa en un concepto llamado *raíz de confianza*, que es un núcleo confiable que se utiliza como base para la seguridad de todo el sistema o aplicación. La arquitectura de seguridad se desarrolla alrededor de la raíz de confianza como una serie de círculos concéntricos, como las capas de una cebolla, que extiende la confianza hacia el exterior desde el centro. Cada capa se basa en su capa interna, de mayor confianza, utilizando controles de acceso, firmas digitales, cifrado y otras primitivas de seguridad. A medida que los sistemas de software se hacen más complejos, es más probable que contengan errores, que los hagan vulnerables a comprometer la seguridad. Como resultado, cuanto más complejo es un sistema de software, más difícil es de asegurar. El concepto de raíz de confianza asegura que la mayor parte de la confianza se coloca dentro de la parte menos compleja del sistema, y por tanto menos vulnerable, mientras que el software más complejo está en las capas de alrededor. Esta arquitectura de seguridad se repite a diferentes escalas, estableciendo primero una raíz de confianza en el hardware de un solo sistema, y después extendiendo esa raíz de confianza a través del sistema operativo hasta llegar a los servicios del sistema de nivel superior, y finalmente a través de muchos servidores situados en capas de círculos concéntricos de confianza decreciente.

La arquitectura de seguridad de bitcoin es diferente. En bitcoin, el sistema de consenso crea un libro contable público confiable que es completamente descentralizado. Una cadena de bloques validada correctamente utiliza el bloque génesis como la raíz de la confianza, construyendo una cadena de confianza hasta el bloque actual. Los sistemas bitcoin pueden y deben utilizar la cadena de bloques como su raíz de confianza. Al diseñar una aplicación de bitcoin compleja que consta de servicios en muchos sistemas diferentes, debes examinar cuidadosamente la arquitectura de seguridad con el fin de determinar dónde se va a colocar la confianza. En última instancia, la única cosa que debe ser de confianza expresamente es una cadena de bloques plenamente validada. Si tu aplicación de forma explícita o implícita otorga la confianza a cualquier cosa que no sea la cadena de bloques, debería ser una fuente de preocupación porque introduce vulnerabilidad. Un buen método para evaluar la arquitectura de seguridad de la aplicación es considerar cada componente individual y evaluar un escenario hipotético donde el componente esté completamente comprometido y bajo el control de un agente malicioso. Toma cada componente de tu aplicación, a su vez, y evalúa los impactos sobre la seguridad general si ese componente se ve comprometido. Si la aplicación ya no es segura cuando esos componentes están en peligro, entonces has colocado la confianza de manera inapropiada en esos componentes. Una aplicación de bitcoin sin vulnerabilidades debe ser vulnerable solamente a un compromiso del mecanismo de consenso de bitcoin, lo que significa que su raíz de confianza debe estar anclada en la parte más fuerte de la arquitectura de seguridad de bitcoin.

Los numerosos ejemplos de casas de cambio de bitcoin hackeadas sirven para subrayar este punto porque su arquitectura y diseño de seguridad falló, incluso bajo el escrutinio más informal. Estas implementaciones centralizadas habían dedicado la confianza explícitamente en numerosos componentes fuera de la cadena de bloques de bitcoin, como carteras calientes, bases de datos de contabilidad centralizadas, llaves de cifrado vulnerables, y estrategias similares.

Mejores Prácticas de Seguridad para el Usuario

Los seres humanos han utilizado los controles de seguridad físicos durante miles de años. En comparación, nuestra experiencia con la seguridad digital tiene menos de 50 años. Los sistemas operativos modernos de propósito general no son muy seguros y no son particularmente adecuados para el almacenamiento de dinero digital. Nuestros equipos están constantemente expuestos a las amenazas externas a través conexiones a internet siempre activas. Ejecutan miles de componentes de software de cientos de autores, a menudo con acceso sin restricciones a los archivos del usuario. Una sola pieza de software ilegítimo, entre los muchos miles instalados en el equipo, puede poner en peligro tus llaves y archivos, y

robar cualquier bitcoin almacenado en aplicaciones de cartera. El nivel de mantenimiento de computadoras requerido para mantener un ordenador libre de virus y troyanos está más allá del nivel de habilidad de la mayoría de los usuarios de computadoras.

A pesar de décadas de investigación y de los avances en la seguridad de la información, los activos digitales siguen siendo lamentablemente vulnerables a un adversario con determinación. Incluso los sistemas más altamente protegidos y restringidos, en las empresas de servicios financieros, agencias de inteligencia y los contratistas de defensa, son vulnerados con frecuencia. Bitcoin crea activos digitales que tienen un valor intrínseco y pueden ser robados y desviados a los nuevos propietarios al instante y de manera irrevocable. Esto crea un incentivo enorme para los piratas informáticos. Hasta ahora, los hackers tuvieron que convertir la información de identidad o los elementos de las cuentas—como tarjetas de crédito y cuentas bancarias—en valor después de haberlos comprometido. A pesar de la dificultad del tráfico y del lavado de la información financiera, hemos visto robos cada vez más intensos. Bitcoin intensifica este problema, ya que no necesita ser traficado o lavado; es valor intrínseco en un activo digital.

Afortunadamente, bitcoin también crea los incentivos para mejorar la seguridad informática. Mientras que antes el riesgo de compromiso del ordenador era vago e indirecto, bitcoin hace que estos riesgos sean claros y evidentes. Guardar bitcoin en un equipo sirve para enfocar la mente del usuario en la necesidad de mejorar la seguridad informática. Como consecuencia directa de la proliferación y la creciente adopción de bitcoin y otras monedas digitales, hemos visto una escalada en las técnicas de hacking y también en las soluciones de seguridad. En términos simples, los hackers tienen ahora un objetivo muy jugoso y los usuarios tienen un claro incentivo para defenderse.

Durante los últimos tres años, como resultado directo de la adopción de bitcoin, hemos visto una enorme innovación en el ámbito de la seguridad de la información en forma de encriptación de hardware, almacenamiento de llaves y carteras hardware, tecnología multifirma y depósito digital en garantía. En las siguientes secciones examinaremos varias mejores prácticas para hacer viable la seguridad del usuario.

Almacenamiento Físico de Bitcoin

Como la mayoría de los usuarios se sienten mucho más cómodos con la seguridad física que con la seguridad de la información, un método muy eficaz para la protección de bitcoin es convertirlos en forma física. Las llaves bitcoin no son más que números largos. Esto significa que se pueden almacenar en una forma física, como impresos en papel o grabados en una moneda de metal. Asegurar las llaves llega a ser entonces tan simple como asegurar físicamente la copia impresa de las llaves bitcoin. Un juego de llaves bitcoin que se imprime en papel se llama una "cartera de papel", y se pueden utilizar muchas herramientas gratuitas para crearlos. Personalmente mantengo la gran mayoría de mis bitcoin (99% o más) almacenados en carteras de papel, encriptadas con BIP-38, con múltiples copias guardadas en cajas fuertes. Mantener bitcoin fuera de linea se llama *almacenamiento en frío* y es una de las técnicas de seguridad más eficaces. Un sistema de almacenamiento en frío es uno donde las llaves se generan en un sistema sin conexión (nunca conectado a internet) y se almacenan también fuera de línea, ya sea en papel o en soporte digital, como un lápiz de memoria USB.

Carteras de Hardware

En el largo plazo, la seguridad de bitcoin tomará cada vez más la forma de carteras de hardware a prueba de manipulaciones. A diferencia de una computadora de escritorio o de un teléfono inteligente, una cartera de hardware bitcoin tiene un solo propósito: almacenar bitcoin de forma segura. Sin la existencia de software de propósito general que pueda comprometerse y con interfaces limitadas, las carteras de hardware pueden ofrecer un nivel de seguridad casi infalible a los usuarios no expertos. Espero ver que las carteras de hardware se conviertan en el método predominante de almacenamiento bitcoin. Para un ejemplo de este tipo de cartera de hardware, consulta Trezor.

Equilibrio del Riesgo

Aunque la mayoría de los usuarios están justamente preocupados por el robo de bitcoin, existe un riesgo aún mayor. Los archivos de datos se pierden todo el tiempo. Si contienen bitcoin, la pérdida es mucho más dolorosa. En el esfuerzo por asegurar sus carteras bitcoin, los usuarios deben tener mucho cuidado de no ir demasiado lejos y terminar perdiendo el bitcoin. En julio de 2011, un conocido proyecto de concienciación y educación de bitcoin perdió casi 7000 bitcoin. En su esfuerzo por evitar el robo, los propietarios habían implementado una compleja serie de copias de seguridad cifradas. Al final perdieron accidentalmente las llaves de cifrado, por lo que las copias de seguridad quedaron sin valor y perdieron una fortuna. Como ocultar dinero enterrándolo en el desierto, si usted asegura su bitcoin demasiado bien, podría ocurrir que no sea capaz de encontrarlo de nuevo.

Diversificación del Riesgo

¿Llevarías todo tu patrimonio neto en dinero en efectivo en tu cartera? La mayoría de la gente lo consideraría imprudente. Sin embargo, los usuarios de bitcoin a menudo mantienen todos sus bitcoin en una sola cartera. En lugar de

ello, los usuarios deben distribuir el riesgo entre múltiples y diversas carteras bitcoin. Los usuarios prudentes mantendrán solo una pequeña fracción, tal vez menos del 5%, de sus bitcoin en una cartera en línea o móvil como "dinero de bolsillo". El resto debe ser dividido entre unos pocos mecanismos de almacenamiento diferentes, tales como una cartera de escritorio y fuera de línea (almacenamiento en frío).

Multifirma y Gobernanza

Cuando una empresa o un individuo almacena grandes cantidades de bitcoin, debería considerar el uso de una dirección bitcoin multifirma. La multifirma proporciona seguridad a los fondos al exigir más de una firma para hacer un pago. Las llaves de firma deben almacenarse en diferentes ubicaciones y estar bajo el control de diferentes personas. En un entorno corporativo, por ejemplo, las llaves deben generarse de forma independiente y deben mantenerse en manos de varios ejecutivos de la empresa, para garantizar que ninguna persona de manera independiente pueda comprometer los fondos. Las direcciones multifirma también pueden ofrecer redundancia, donde una sola persona tiene varias llaves que se almacenan en ubicaciones diferentes.

Supervivencia

Una consideración importante de seguridad que a menudo se pasa por alto es la disponibilidad, especialmente en el contexto de incapacidad o muerte del titular de la llave. A los usuarios de bitcoin se les dice que deben usar contraseñas complejas y mantener sus llaves de manera segura y privada, y que no deben compartirlas con nadie. Por desgracia, esa práctica hace que sea casi imposible para la familia del usuario recuperar los fondos si éste no está disponible para desbloquearlos. En la mayoría de los casos, de hecho, las familias de los usuarios de bitcoin podrían ignorar completamente la existencia de los fondos de bitcoin.

Si tienes un montón de bitcoin, deberías considerar compartir los datos de acceso con un familiar o un abogado de confianza. Un plan de supervivencia más complejo se puede configurar con acceso multi-firma y administración patrimonial a través de un abogado especializado como un "albacea de activos digitales."

Conclusión

Bitcoin es algo completamente nuevo, sin precedentes, y de compleja tecnología. Con el tiempo vamos a desarrollar mejores herramientas y prácticas que son más fáciles de usar por los no expertos en seguridad. Por ahora, los usuarios de bitcoin pueden utilizar muchos de los consejos discutidos aquí para disfrutar de una experiencia de bitcoin segura y sin problemas.

Programación Sobre la Cadena de Bloques

Ahora vamos a desarrollar nuestra comprensión de bitcoin viéndolo como una *plataforma de protocolos*. Hoy en día, muchas personas usan el término "blockchain" (o cadena de bloques) para referirse a cualquier plataforma de protocolos que comparta los fundamentos de diseño de bitcoin. El término a menudo se usa incorrectamente y se aplica a muchas cosas que no ofrecen las características principales que sí ofrece la cadena de bloques de bitcoin.

En este capítulo vamos a revisar las características que ofrece la cadena de bloques de bitcoin, como una plataforma de protocolos. Consideraremos los fundamentos *rudimentarios*, que conforman los ladrillos de cualquier código ó aplicación en la cadena de bloques. Examinaremos varios protocolos relevantes que utilizan estos rudimentos, tales como los canales de pagos (o de estados) y los canales de pago orientados a conexión (La célebre red llamada “Lightning Network”).

Introducción

El sistema bitcoin fue diseñado tanto como una moneda descentralizada, así como también una plataforma de pagos. Sin embargo, la mayor parte de su funcionalidad, se deriva de unos ladrillos de construcción de muy bajo nivel que se pueden usar para aplicaciones mucho más variadas. Bitcoin no se creó con componentes tales como “cuentas”, “usuarios”, “saldos” y “pagos”. En lugar de esto, bitcoin utiliza un lenguaje de instrucciones de gestión transaccional (o scripts) con funciones criptográficas de bajo nivel, como se vio en <1>. Del mismo modo que los conceptos de cuentas, saldos y pagos de nivel superior pueden derivarse de estos rudimentos básicos, también pueden hacerlo muchas otras aplicaciones complejas. Por lo tanto, la cadena de bloques de bitcoin puede convertirse en una plataforma de protocolos capaz de ofrecer servicios confiables a sus aplicaciones, tales como los “contratos inteligentes”, superando con creces el propósito original de la moneda digital y de la plataforma de pagos.

Ladrillos (rudimentos)

Cuando funciona correctamente y de manera continua, el sistema bitcoin ofrece ciertas garantías, que pueden usarse como bloques de construcción o "ladrillos" para crear nuevos protocolos. Estas garantías incluyen:

Ausencia de Gastos Duplicados

La garantía más fundamental del algoritmo de consenso descentralizado de bitcoin, asegura que no será posible gastar una misma “UTXO” dos veces.

Inmutabilidad

Una vez que se registra una transacción en la cadena de bloques y se ha agregado suficiente trabajo de cómputo con los bloques subsecuentes, los datos de toda transacción se vuelven inmutables. La inmutabilidad está asegurada por la energía, ya que reescribir la cadena de bloques requiere del gasto de energía para producir una nueva Prueba-de-Trabajo. La energía requerida para manipular a una transacción y por lo tanto, el grado de inmutabilidad que ésta posee, aumentará con la cantidad de trabajo de cómputo comprometido en la cadena de los bloques que le siguen al bloque que contiene a dicha transacción.

Neutralidad

El sistema descentralizado de bitcoin propagará por toda su red a toda aquella transacción que simplemente cumpla con ser válida, sin importar el origen o el contenido de dicha transacción. Esto significa que cualquiera puede crear una transacción válida y que pagando suficientes comisiones, tendrá la certeza de que podrá transmitir esa transacción e incluirla en la cadena de bloques en cualquier momento .

Sellos de Tiempo Seguros

Las reglas de consenso rechazan cualquier bloque cuyo sello de tiempo esté en un punto muy alejado del pasado o del futuro. Esto asegura que los sellos de tiempo en los bloques sean confiables. El sello de tiempo en un bloque representa una prueba que garantiza que ninguna entrada de ninguna transacción incluida en él, ha sido previamente gastada.

Pruebas de Autorización

Las firmas digitales, validadas en una red descentralizada, ofrecen garantías de autorización. Las instrucciones que contienen un requisito de firma digital válida, no se pueden ejecutar sin la autorización del titular de la llave privada implícita en dicha instrucción.

Auditabilidad

Todas las transacciones son públicas y pueden ser auditadas. Todas las transacciones y todos los bloques se pueden

vincular de regreso en una cadena ininterrumpida, hasta el bloque génesis.

Contabilidad

En toda transacción (exceptuando a las de acuñación o transacción coinbase) el valor de las entradas debe igualar al valor neto de las salidas más las comisiones. No es posible crear o destruir valor de bitcoin en una transacción. Las salidas jamás podrán exceder a las entradas.

Ausencia de Caducidad

Una transacción de bitcoin válida no tiene fecha de caducidad. Si es válida hoy, lo será también en el mañana, siempre y cuando las entradas permanezcan sin gastar y las reglas de consenso no cambien.

Integridad

Una transacción de bitcoin firmada con el banderín SIGHASH_ALL o cuyas partes han sido firmadas por otro tipo de banderín SIGHASH no puede modificarse sin invalidar la firma, invalidando así la totalidad de la transacción.

Atomicidad de las Transacciones

Las transacciones de bitcoin son atómicas. O bien son válidas y confirmadas (o minadas) en su totalidad o no lo son. Las transacciones no se pueden confirmar parcialmente y no hay un estado intermedio para ninguna transacción. En todo momento, una transacción o bien ha sido confirmada o no.

Unidades de Valor Discretas (ó Indivisibles)

Las salidas de toda transacción son unidades de valor discretas e indivisibles. O se gastan en su totalidad, o no se gastan. No se pueden dividir ni se pueden gastar parcialmente.

Quórum de Control

Las instrucciones transaccionales de multifirma, imponen restricciones tales que exigen un quórum mínimo de autorización, predefinido en dicho esquema de multifirma. El requisito de “al menos M-de-N” se aplica mediante las reglas de consenso.

Disposición de Bloqueos Temporales / Procesos de Maduración

Cualquier cláusula script que contenga un bloqueo temporal relativo o absoluto, solo puede ejecutarse después de que su antigüedad exceda el tiempo especificado por dicha cláusula.

Replicación

El almacenamiento descentralizado típico de la cadena de bloques, asegura que cuando una transacción ha sido minada y después de suficientes confirmaciones, la información se replica en toda la red y se vuelve duradera y resistente a cortes de energía, corrupción de datos, etc.

Blindaje ante la Falsificación

Una transacción solo puede gastar salidas existentes y convalidadas. No es posible crear o falsificar valor.

Consistencia

En ausencia de un ataque de aislamiento o “particionamiento” entre mineros, los bloques que se registran en la cadena de bloques están sujetos a reorganización o desacuerdo con una probabilidad exponencialmente decreciente, en función de la profundidad a la que éstos van quedando registrados. Una vez que quedan profundamente registrados en la cadena, el cómputo y la energía necesarios para manipular un bloque hacen que el cambio sea prácticamente inviable.

Posibilidad de Registro de Estados Externos

Una transacción puede consignar un tren de datos, a través de OP_RETURN, capaz de representar una transición de estado en una máquina de estados externa a la cadena de bloques de bitcoin.

Acuñación Predecible

En total serán acuñados 21 millones de bitcoins o menos, a una tasa predecible.

Esta lista de ladrillos fundamentales del protocolo de bitcoin no es exhaustiva, y además que se agregan otras con cada nueva característica introducida en el sistema.

Los ladrillos fundacionales que nos ofrece bitcoin, son elementos de una plataforma confiable que se pueden usar para desarrollar aplicaciones. Aquí hay algunos ejemplos de las aplicaciones que existen hoy día y de los componentes básicos que estas utilizan:

Pruebas-de-Existencia (Notarios Digitales)

Inmutabilidad + Sello de tiempo + Durabilidad. El registro de la huella digital de un archivo puede perpetuarse mediante una transacción en la cadena de bloques, demostrando que existía previamente otro documento (un "Sello de Tiempo") para el momento del registro. Esta huella digital no podrá modificarse a-posteriori (Inmutabilidad) y la prueba de su existencia se almacenará de forma permanente (Durabilidad).

Colectas de Emprendimiento sin Intermediarios (Tales como Lighthouse.cash)

Consistencia + Atomicidad + Integridad. Si una entrada se firma conjuntamente con la salida (Integridad) de una transacción de recaudación de fondos, otras entradas pueden contribuir a la recaudación, pero la transacción como tal no se podrá gastar o confirmar (Atomicidad) hasta que la suma total de las entradas no financie la totalidad del objetivo (valor neto de la salida) (Consistencia).

Canales de Pago

Quórum de Control + Bloqueo Temporal + Sin Gastos Duplicados + Sin Caducidad + Resistencia a la Censura + Pruebas de Autorización. Una instrucción de firma múltiple bajo esquema (Quórum) "al menos 2-de-2" con un bloqueo de tiempo (Bloqueo Temporal), utilizada para la transacción de "liquidación" de un canal de pago, puede ser conservada (Sin Caducidad) y gastada en cualquier momento (Resistencia a la Censura) por cualquiera de las partes (Prueba de Autorización). Las dos partes pueden crear transacciones de compromiso que invaliden mediante doble gato (ya que no pueden haber Gastos Duplicados) un acuerdo previo de liquidación, utilizando un bloqueo de tiempo más corto (Bloqueo Temporal).

La Plataforma "Counterparty"

Counterparty es un protocolo que funciona como capa de servicios construida sobre bitcoin. El protocolo de Counterparty ofrece la capacidad de crear y comercializar activos y tokens (fichas) virtuales. Además, Counterparty ofrece una casa de cambio de activos descentralizada. Counterparty también está incorporando en su protocolo la capacidad de gestionar contratos inteligentes, basados en la Máquina Virtual de Ethereum (MVE o también del inglés "EVM") .

Counterparty incrusta los metadatos en las transacciones de bitcoin, utilizando la instrucción transaccional OP_RETURN o direcciones multifirma con esquemas "al menos 1-de-N" que codifican los metadatos en lugar de utilizar llaves públicas. Usando estos mecanismos, Counterparty implementa una capa de protocolos codificada en transacciones de bitcoin. La capa de protocolos adicional puede ser interpretada por cualquiera de las aplicaciones desarrolladas para detectar metadatos de Counterparty, tales como billeteras, exploradores de la cadena de bloques, o cualquier otra aplicación creada utilizando las bibliotecas de Counterparty.

Counterparty puede a su vez, utilizarse como plataforma base para otras aplicaciones y servicios. Por ejemplo, Tokenly es una capa de servicios construida sobre la capa de Counterparty, que permite a los creadores de contenidos, artistas y compañías diversas emitir tokens que den fe de cierta participación en propiedades de tipo digital o intelectual y que se pueden usar para alquilar, acceder, comerciar o comprar contenido, productos y servicios. Otras aplicaciones que aprovechan Counterparty incluyen juegos (como por ejemplo "Spells of Genesis") y proyectos de computación en malla (como "Folding Coin").

Se pueden encontrar más detalles sobre Counterparty en https://counterparty.io. El proyecto de código abierto se puede encontrar en https://github.com/CounterpartyXCP.

Canales de Pago y Canales de Estados

Los Canales de Pago son mecanismos libres de la dependencia en la confianza o "desconfiables", capaces de intercambiar transacciones de bitcoin entre dos partes, fuera de la cadena de bloques de bitcoin. Estas transacciones, que serían válidas si se liquidan en la cadena de bloques de bitcoin, se mantienen fuera de la cadena y en su lugar, actúan como *pagarés* para producir una liquidación final por lotes. Debido a que las transacciones no se difunden en la red bitcoin, se pueden intercambiar sin la latencia de confirmación habitual, lo que permite un rendimiento de transacciones extremadamente alto, una latencia baja (sub-milisegundos) y una granularidad fina (a nivel de satoshis).

En realidad, el término *canal* es una metáfora. Los canales de estados son idealizaciones virtuales representadas por el intercambio de estados entre dos partes, fuera de la cadena de bloques. No hay "canales" como tales y el mecanismo de

transporte de datos subyacente no es ningún canal. Usamos el término "canal" para representar la relación y el estado compartido entre dos partes, fuera de la cadena de bloques.

Para explicar mejor este concepto, considérese una secuencia de datos TCP (datos enviados con protocolos de transmisión controlada). Desde la perspectiva de los protocolos de nivel superior, esta secuencia no es más que una "interfaz" de internet, que conecta dos aplicaciones a través de la red de redes. Pero si se observa el tráfico real de la red, una secuencia TCP es una especie de canal virtual sobre la capa de servicios de los paquetes IP. Cada punto final del protocolo TCP transmite las secuencias y ensambla los paquetes IP, para crear la ilusión de una secuencia de bytes. Por debajo del protocolo, toda la data se conforma de paquetes desconectados. Del mismo modo, un canal de pagos es solo una serie de transacciones. Si éstas están correctamente secuenciadas y conectadas, crearán obligaciones de pago redimibles en las que se puede confiar aun cuando no exista confianza alguna en la contraparte del comercio que yace conectada al otro lado del canal.

En esta sección se abordarán varios tipos de canales de pago. Primero, examinaremos los mecanismos utilizados para construir un canal de pagos de un solo sentido (unidireccional) para asistir a un servicio de micropagos por demanda, como por ejemplo el pago por la transmisión de un vídeo. Luego, ampliaremos este mecanismo e introduciremos canales de pagos bidireccionales. Finalmente, abordaremos cómo los canales bidireccionales se pueden conectar de extremo-a-extremo para formar canales de saltos múltiples en una red orientada a conexión, inicialmente propuesta bajo el nombre de *Lightning Network*.

Los canales de pagos son parte de un concepto más amplio conocido como *canales de estados*, que representan una alteración de los estados de un sistema externo a la cadena de bloques (off-chain), y que vienen garantizados por un compromiso que eventualmente se difundirá hacia la cadena de bloques. Un canal de pago es un canal de estados donde el estado que se está alterando es el saldo referido a una moneda virtual.

Canales de Estados—Conceptos Básicos y Terminología

Un canal de estados puede establecerse entre dos partes, a través de una transacción que comprometa un estado de saldos compartido, en la cadena de bloques. A esta se le conoce como la *transacción de financiación* o *transacción de anclaje*. Esta transacción única debe difundirse a la red bitcoin y confirmarse para establecer el canal. En el ejemplo de un canal de pagos, el estado comprometido es el saldo inicial (en valor monetario) del canal.

Luego, las dos partes intercambiarán un par de transacciones firmadas, llamadas *transacciones de compromiso*, capaces de alterar el estado inicial. Estas son transacciones válidas en el sentido de que *pudieran* ser difundidas para su confirmación por cualquiera de las partes, pero aún así son mantenidas fuera de la cadena en espera de un cierre amistoso del canal. Las actualizaciones de cada estado se pueden crear a la misma velocidad a la que cada parte puede redactar, firmar y transmitir una transacción a la otra parte. En la práctica, esto significa que se pueden intercambiar miles de transacciones por segundo.

Al intercambiarse las transacciones de compromiso, las dos partes también invalidan los estados anteriores, buscando que la transacción de compromiso más actualizada sea siempre la única que pueda convalidarse. Esto evita que cualquiera de las partes haga trampa cerrando unilateralmente el canal con un estado previo ya vencido, que pueda resultarle más favorable que el estado de saldos actual. Examinaremos los diversos mecanismos reales que pueden usarse para invalidar estados previos o vencidos, en el resto de este capítulo.

Finalmente, el canal puede cerrarse de forma cooperativa, difundiendo una última *transacción de liquidación* a la cadena de bloques, o bien unilateralmente (en un cierre no amistoso), cualquiera de las partes difunde la última transacción de compromiso a la cadena de bloques. Se necesita una opción de cierre unilateral en caso de que una de las partes se desconecte inesperadamente. La transacción de liquidación representa el estado final del canal y su propósito es ser difundida en la cadena de bloques.

Durante toda la vida útil del canal, solo se deben enviar dos transacciones para ser confirmadas en la cadena de bloques: las transacciones de financiación y de liquidación. Entre estos dos estados, las dos partes pueden intercambiar cualquier cantidad de transacciones de compromiso que nunca son observadas por nadie más, ni enviadas a la cadena de bloques .

<1>ilustra un canal de pagos entre Alice y Bob, exhibiendo las transacciones de financiación, compromiso y liquidación.

Un canal de pagos entre Alice y Bob, que demuestra las transacciones de financiación, compromiso y liquidación
image::images/mbc2_1201.png ["Un canal de pagos entre Alice y Bob, que demuestra las transacciones de financiación, compromiso y liquidación"]

Ejemplo de un Canal de Pagos Simple

Para ofrecer una explicación de los canales de estados, comenzaremos con un ejemplo muy simple. Se describirá un canal unidireccional, es decir, un canal en el cual el valor fluye en una sola dirección. También se dará inicio a la explicación con la suposición ingenua de que nadie está tratando de hacer trampa, para mantener las cosas simples. Una vez explicada la idea básica del canal, serán abordados los recursos que se necesitan para hacer que dicho canal sea desconfiable para que ninguna de las partes *pueda* engañar al otro, incluso si tuviesen la intención de hacerlo.

Para este ejemplo asumiremos dos participantes: Emma y Fabián. Fabián ofrece un servicio de transmisión de video que se factura por segundo, usando un canal de micropagos. Fabián cobra 0.01 millibit (0.00001 BTC), por segundo de video, equivalente a 36 millibits (0.036 BTC) por hora de video. Emma es un usuario que compra este servicio de transmisión de video a Fabián. <1> muestra a Emma comprándole el servicio de transmisión de video a Fabián usando un canal de pago.

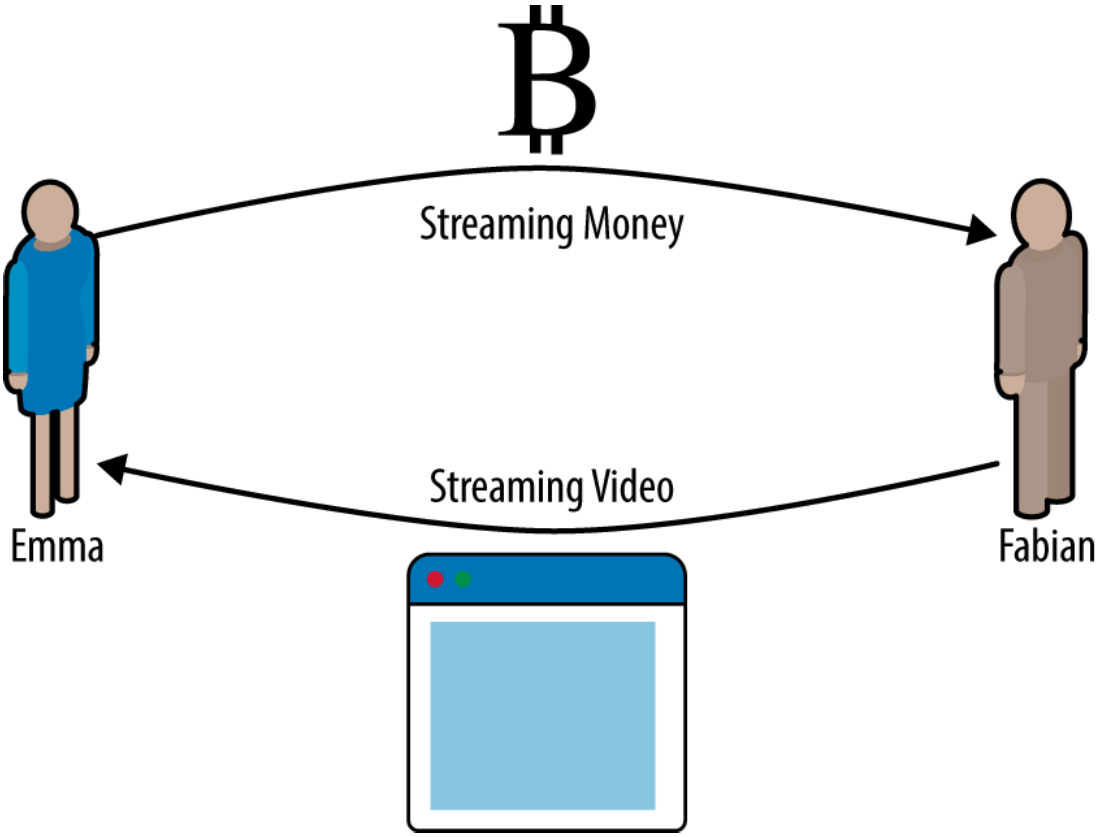


Figure 74. Emma compra vídeos “por demanda” a Fabián con un canal de pago, pagando por cada segundo de vídeo

En este ejemplo, Fabián y Emma están utilizando un software especial que maneja tanto el canal de pagos como la transmisión de video. Emma está ejecutando el software en su navegador y Fabián lo está ejecutando desde un servidor. El software incluye las funcionalidades básicas de una cartera de bitcoin y puede crear y firmar transacciones en bitcoin. Tanto el concepto como el término "canal de pagos" están completamente ocultos para los usuarios. Lo que ven es un vídeo que se está pagando por segundo.

Para configurar el canal de pagos, Emma y Fabián establecen una dirección de multifirma de “al menos 2-de-2” firmas, mientras cada uno de ellos posee una de las dos llaves. Desde la perspectiva de Emma, el software en su navegador presentará ante ella un código QR con una dirección P2SH (o del tipo “pagar a un script-hash”, que comienza con “3”), y le pide que envíe un “depósito” de hasta 1 hora de vídeo. La dirección P2SH es entonces financiada por Emma. La transacción de Emma, que ha hecho el pago a la dirección de multifirma, es la transacción de financiación o de anclaje para el canal de pagos.

Para este ejemplo, supongamos que Emma financia el canal con 36 millibits (0.036 BTC). Esto le permitirá a Emma consumir *hasta* 1 hora de transmisión de vídeo. La transacción de financiación en este caso establece la cantidad máxima de material que se puede transmitir a través de este canal, lo cual configura la *capacidad del canal*.

La transacción de financiación puede consumir una o más de las entradas de la cartera de Emma, para completar los fondos. Esta transacción creará una salida, con un valor de 36 millibits a favor de la dirección de multifirma de "al menos 2-de-2" firmas, controlada conjuntamente entre Emma y Fabián. Esta transacción pudiera tener salidas adicionales para devolver el cambio a favor de la cartera de Emma.

Una vez que se confirma la transacción de financiación, Emma puede comenzar a recibir las secuencias del vídeo. Al inicio del primer segundo, el software de Emma crea y firma una transacción de compromiso que realizará cambios del saldo del canal para acreditar unos 0.01 millibit a favor de la dirección de Fabián y devolver 35.99 millibits a Emma. La transacción firmada por Emma se gasta toda la salida de 36 millibits que fue creada por la transacción de financiación, como su única entrada y crea dos salidas: una para su reembolso y la otra para el pago de Fabián. Pero la transacción solo está parcialmente firmada—la misma requiere de ambas firmas (2-de-2), pero solo ha sido adjuntada la firma de Emma. Cuando el servidor de Fabián reciba esta transacción, se agregará la segunda firma (para cumplir con la entrada 2-de-2) y se la devuelve a Emma junto con 1 segundo de vídeo. Ahora ambas partes tienen una transacción de compromiso totalmente firmada que cualquiera puede canjear, lo que representa el saldo actualizado correcto del canal. Ninguna de las partes debería difundir esta transacción a la red.

Al comienzo del siguiente segundo, el software de Emma crea y firma otra transacción de compromiso (el compromiso Nro. 2) que consume la misma salida de “al menos 2-de-2” (P2SH) de la transacción de financiación. La segunda transacción de compromiso asigna una salida de 0.02 millibits a favor de la dirección de Fabián y una salida de 35.98 millibits a favor de la dirección de Emma. Esta nueva transacción es el pago por dos segundos acumulados de vídeo. El software de Fabián firma y devuelve la segunda transacción de compromiso, junto con otro segundo de vídeo.

De esta manera, el software de Emma continúa enviando transacciones de compromiso al servidor de Fabián a cambio de tramas adicionales de vídeo. Los estados de saldos del canal se acumulan gradualmente a favor de Fabián, ya que Emma consume progresivamente, más segundos de vídeo. Digamos que Emma mira 600 segundos (10 minutos) de vídeo, creando y firmando 600 transacciones de compromiso. La última transacción de compromiso (la Nro. 600) tendrá igualmente dos salidas, dividiendo el saldo del canal, 6 millibits a favor de Fabián y 30 millibits a favor de Emma.

Llegado el momento, Emma hace clic en "Detener" para finalizar la transmisión de vídeo. Fabián o Emma ahora pueden transmitir la transacción de estados (de compromisos) más reciente para su liquidación. Esta última transacción se convierte en la transacción de liquidación que le paga a Fabián por todo el vídeo que Emma consumió, reembolsando el resto de la transacción de financiación a Emma.

<1>ilustra el canal entre Emma y Fabián y las transacciones de compromiso que actualizan el saldo del canal.

A la larga, solo se registran dos transacciones en la cadena de bloques: la transacción de financiación que estableció el canal y la transacción de liquidación que asignó el saldo final correctamente entre los dos participantes.

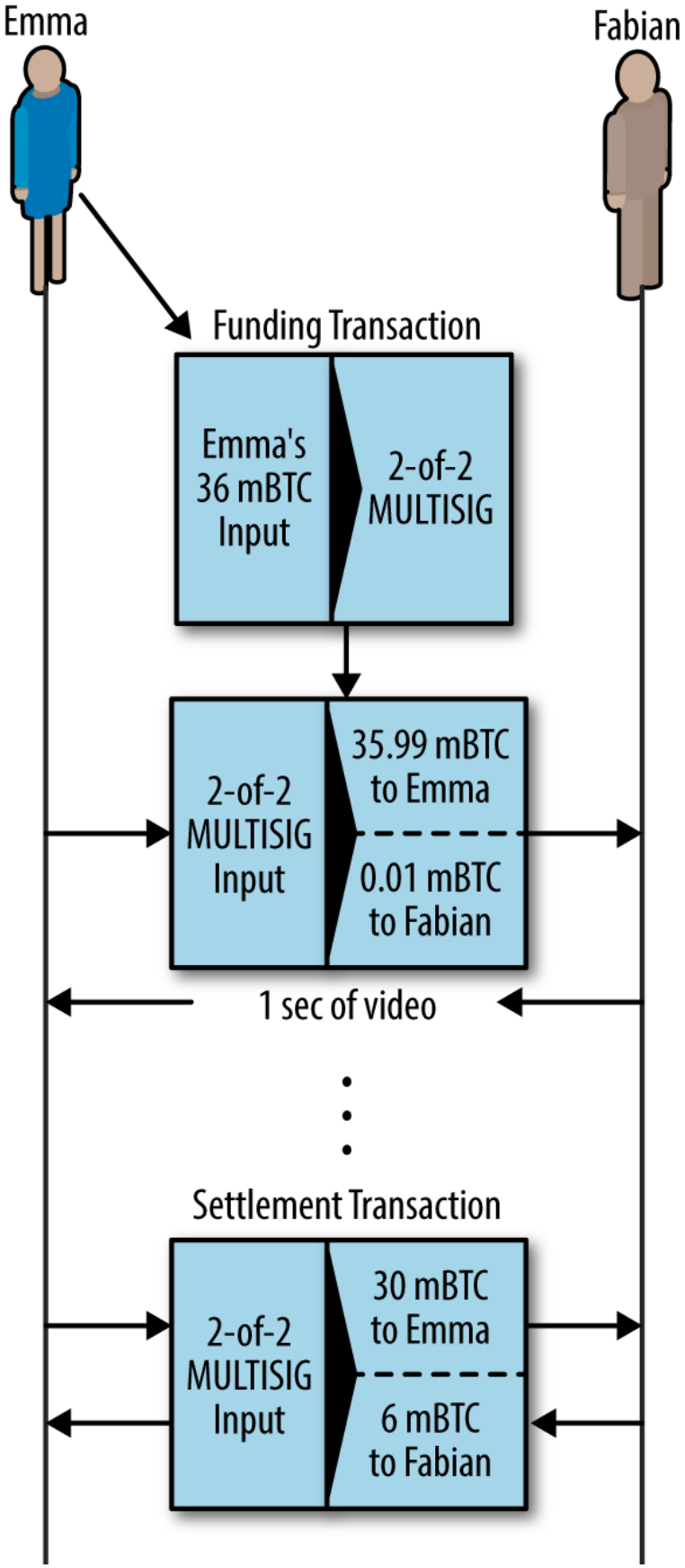


Figure 75. El canal de pagos entre Emma y Fabián, que muestra las transacciones de compromiso que han actualizado el saldo del mismo.

Rediseñando los Canal para que sean "Desconfiables"

El canal que acabamos de describir funciona, pero solo si ambas partes cooperan amistosamente, sin fallas ni intentos de hacer trampa. Veamos algunos de los escenarios que pueden arruinar este canal y veamos qué se necesita para evitarlo:

- Una vez que la transacción de financiación ha ocurrido, Emma necesita la firma de Fabián para recuperar el dinero. Si Fabián se desaparece, los fondos de Emma quedan bloqueados en un esquema de "al menos 2-de-2" firmas y

efectivamente se pierden. Este canal, tal como se ha diseñado, conduce a una pérdida de fondos si uno de los participantes se desconecta antes de que haya al menos una transacción de compromiso firmada por ambas partes.

- Mientras el canal se está ejecutando, Emma puede tomar cualquiera de las transacciones de compromiso que Fabián ha refrendado y difundir cualquiera de ellas a la cadena de bloques. ¿Por qué pagar 600 segundos de vídeo, si ella puede perfectamente difundir la transacción de compromiso Nro 1 y solo pagar 1 segundo de vídeo? El canal fracasa porque Emma puede hacer trampa al difundir cualquier compromiso previo que más le favorezca.

Ambos problemas pueden resolverse con bloqueos temporales—veamos cómo podríamos poner en práctica estos bloqueos temporales a nivel de transacciones (nLocktime).

Emma no puede arriesgarse a colocar sus fondos en una dirección de “al menos 2-de-2” firmas, a menos que tenga un reembolso garantizado. Para resolver este problema, Emma redacta tanto una transacción de financiación como una de reembolso, al mismo tiempo. Ella firma la transacción de financiación pero no se la comunica a nadie. Emma le comunica a Fabián únicamente la transacción de reembolso, (que gasta la salida de la transacción de financiación que ella mantiene en secreto), con el fin de obtener su firma.

La transacción de reembolso actúa como la primera transacción de compromiso y su bloqueo de tiempo establece un límite superior para la vida del canal. En este caso, Emma podría establecer el parámetro nLocktime en 30 días o 4320 bloques en el futuro. Todas las transacciones de compromiso subsecuentes deben tener un bloqueo de tiempo más corto, para que puedan validarse antes de que pueda difundirse la transacción de reembolso.

Ahora que Emma tiene en sus manos una transacción de reembolso completamente firmada, puede revelarle a Fabián con toda confianza, la transacción de financiación que ella ha firmado, sabiendo que en todo caso, después de que expire el bloqueo de tiempo, ella podrá difundir la transacción de reembolso, incluso si Fabián se desaparece.

Cada transacción de compromiso que las partes intercambien durante la vida del canal, de ahora en más llevará un bloqueo de tiempo. Pero el lapso de maduración de cada bloqueo, será progresivamente más corto para cada compromiso, por lo que el compromiso más reciente se debe poder difundir antes que el compromiso anterior que este nuevo estará invalidando. Debido al ajuste del parámetro nLockTime, ninguna de las partes puede propagar con éxito ninguna de las transacciones de compromiso hasta que expire su bloqueo de tiempo. Si todo sale bien, las partes cooperarán y cerrarán el canal apropiadamente con una transacción de liquidación, haciendo innecesario difundir una transacción de compromiso intermedio. De lo contrario, la transacción de compromiso más reciente será difundida por la parte más interesada, para liquidar la cuenta e invalidar todas las transacciones de compromiso anteriores.

Por ejemplo, si la transacción de compromiso Nro. 1 está bloqueada por un ajuste de 4320 bloques en el futuro, entonces la transacción de compromiso Nro. 2 debería venir bloqueada con un ajuste de no más de 4319 bloques en el futuro. La transacción de compromiso Nro. 600 se debería poder gastar no menos de 600 bloques antes de que la transacción de compromiso Nro. 1 pueda llegar a ser válida.

<1>muestra cada transacción de compromiso ajustando un bloqueo de tiempo cada vez más corto, lo que permite gastarlos antes de que los compromisos anteriores sean válidos.

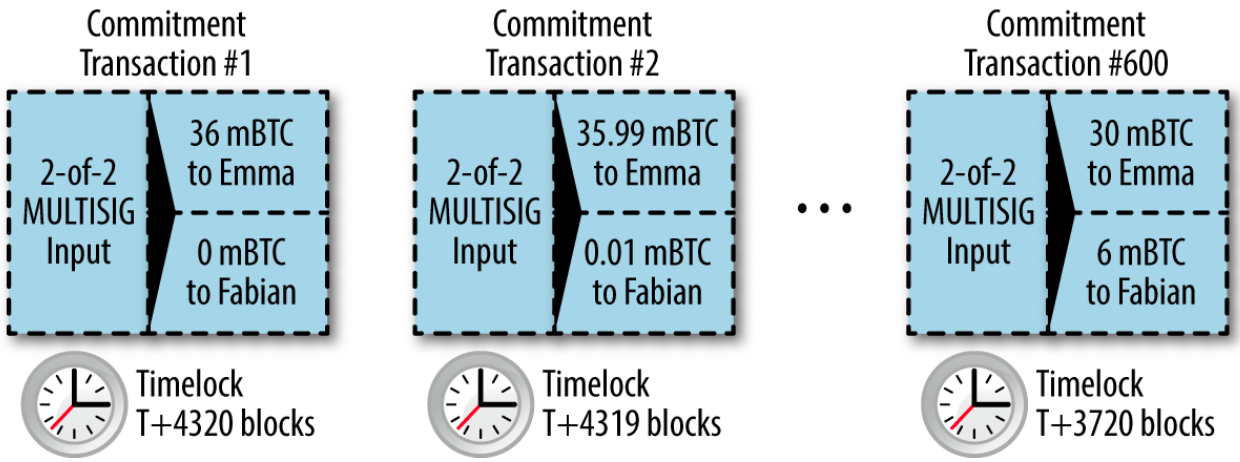


Figure 76. Cada compromiso establece un ajuste de bloqueo de tiempo cada vez más corto, lo que permite gastarlo antes de que los compromisos anteriores sean válidos

Cada transacción de compromiso subsiguiente debe tener un ajuste de maduración más corto para que pueda difundirse antes que sus predecesoras y antes que la transacción de reembolso. La capacidad de difundir un compromiso con mayor

anterioridad, garantiza que la salida de la transacción de financiación podrá gastarse y evitar que cualquier otra transacción de compromiso se difunda gastando dicha salida. Las garantías ofrecidas por la cadena de bloques de bitcoin, tales como las de impedir gastos duplicados y la capacidad de imponer bloqueos de tiempo, permiten que cada transacción de compromiso invalide a sus predecesoras.

Los canales de estados usan los bloqueos temporales para garantizar el cumplimiento de contratos inteligentes a lo largo de la línea temporal. En este ejemplo, vimos cómo la evolución temporal garantiza que la transacción de compromiso más reciente sea la única válida antes que cualquier otro compromiso anterior. Por lo tanto, la transacción de compromiso más reciente se puede difundir, gastando las entradas correspondientes, e invalidando asimismo las transacciones de compromisos anteriores. La gestión de contratos inteligentes con parámetros absolutos de bloqueos temporales protege contra el fraude de cualquiera de las partes. Esta gestión no necesita más que el ajuste del parámetro de bloqueo temporal absoluto a nivel de transacción (nLocktime). A continuación, veremos cómo los bloqueos temporales a nivel de comandos "script", del tipo CHECKLOCKTIMEVERIFY y CHECKSEQUENCEVERIFY, pueden usarse para construir canales de estados más flexibles, útiles y elegantes.

La primera forma de canal de pago unidireccional fue demostrada como una aplicación prototipo de transmisión de vídeo en 2015 por un equipo argentino de desarrolladores.

Los bloqueos temporales no son la única forma de invalidar transacciones con compromisos previas. En las siguientes secciones veremos cómo se puede usar una llave de revocación para lograr el mismo resultado. Los bloqueos temporales son efectivos pero tienen dos desventajas diferentes. Al establecerse un bloqueo temporal máximo, al momento de abrirse por primera vez el canal, se limita la vida útil del mismo. Incluso peor, esto enfrenta al diseño de canales al dilema de conseguir canales de larga duración mientras una de las partes se ve obligada a esperar mucho tiempo por un reembolso en caso de cierre prematuro. Por ejemplo, si se permite que el canal permanezca abierto durante 30 días, estableciendo el tiempo de bloqueo para el reembolso en 30 días, al desaparecerse una de las partes sin dar aviso, la otra parte debe esperar 30 días para obtener su reembolso. Cuanto más distante se concibe al punto final del canal, más distante será el reembolso.

El segundo problema es que, dado que cada transacción de compromiso posterior debe disminuir el ajuste del bloqueo temporal, existe un límite explícito en el número de transacciones de compromiso que pueden intercambiarse entre las partes. Por ejemplo, un canal de 30 días, que establece un bloqueo temporal de 4320 bloques en el futuro, solo puede acomodar 4320 transacciones de compromiso intermedias antes de que deba cerrarse. Existe un cierto peligro al establecer el intervalo entre bloqueos temporales para transacciones de compromiso, en sólo 1 bloque menos que la anterior. Al establecer el intervalo de bloqueo temporal entre las transacciones de compromiso en 1 solo bloque, el desarrollador esta imponiendo una responsabilidad muy riesgosa para los participantes del canal, ya que deben estar atentos, permanecer en línea y observando, y estar listos para transmitir la transacción de compromiso correcta en cualquier momento.

Ahora que entendemos cómo se pueden usar los bloqueos temporales para invalidar compromisos anteriores, podemos ver la diferencia entre el cierre de un canal de manera cooperativa y su cierre unilateral al difundir una transacción de compromiso. Todas las transacciones de compromiso poseen su bloqueo temporal, por lo tanto, difundir una transacción de compromiso siempre implicará esperar hasta que el bloqueo temporal haya expirado. Pero si las dos partes acuerdan amistosamente cuál es el saldo final y saben que ambas tienen transacciones de compromiso que eventualmente harán que ese saldo sea una realidad, pueden redactar una transacción de liquidación sin ningún bloqueo temporal, que refleje ese mismo estado de saldos. En un cierre cooperativo, cualquiera de las partes toma la transacción de compromiso más reciente y redacta una transacción de liquidación que es idéntica en todos los sentidos a la del referido compromiso, excepto que omite el bloqueo temporal. Ambas partes pueden firmar esta transacción de liquidación sabiendo que no hay forma de hacer trampa y obtener un resultado más favorable. Al firmar y difundir cooperativamente la transacción de liquidación, pueden cerrar el canal y recuperar sus saldos de inmediato. En el peor de los casos, si una de las partes resulta ser mezquina, o se negase a cooperar, obligaría a la otra parte a cerrar unilateralmente, difundiendo la transacción de compromiso más reciente. Pero si hacen eso, también tendrían que ponerse a esperar por sus fondos.

Compromisos con Revocación Asimétrica

Una mejor manera de manejar los estados de compromisos anteriores es revocarlos explícitamente. Sin embargo, esto no es fácil de lograr. Una característica clave de bitcoin es que una vez que una transacción es confirmada, sigue siendo válida y jamás caduca. La única forma de cancelar una transacción es generando un gasto duplicado a sus entradas, mediante otra transacción que logre confirmarse antes de que la primera sea minada. Es por esto que usamos bloqueos temporales en el ejemplo anterior del canal de pago simple, para asegurarnos de que los compromisos más recientes puedan gastarse antes de que los compromisos anteriores sean validados. Sin embargo, la secuencia de compromisos en

el tiempo crea una serie de restricciones que hacen que los canales de pago sean difíciles de usar.

Aún cuando una transacción no se puede cancelar de por sí, se puede constituir de tal manera que se haga indeseable utilizarla. La forma en que esto se logra es revelando a cada parte una *llave de revocación* que puede usarse para castigar a la contraparte si intentara hacer trampa. Este mecanismo para revocar transacciones de compromisos anteriores se propuso por vez primera, como parte de la plataforma "Lightning Network".

Para explicar el mecanismo de las llaves de revocación, construiremos un canal de pagos más complejo entre dos casas de cambio administradas por Hitesh e Irene. Hitesh e Irene administran comercios de bitcoins en India y EE. UU., respectivamente. Los clientes de la casa de cambios en India de Hitesh a menudo envían pagos a los clientes de la plataforma estadounidense de Irene y viceversa. Actualmente, estas transacciones ocurren en la cadena de bloques de bitcoin, pero esto implica pagar comisiones y esperar varios bloques para las confirmaciones. La creación de un canal de pagos entre las plataformas de intercambios reduciría significativamente los costos y aceleraría el flujo de transacciones.

Hitesh e Irene inicializan el canal redactando cooperativamente una transacción de financiación, donde cada parte financia el canal con 5 bitcoins. El estado inicial de saldos es de 5 bitcoins para Hitesh y 5 bitcoins para Irene. La transacción de financiación bloquea el estado del canal en una salida de firmas múltiples de "al menos 2-de-2", al igual que en el ejemplo de un canal simple.

La transacción de financiación puede tener una o más entradas de Hitesh (con tal que totalicen los 5 bitcoins o más), y una o más entradas de Irene (totalizando 5 bitcoins o más). Las entradas tienen que superar ligeramente la capacidad del canal para cubrir las comisiones de la transacción. La transacción tiene una salida que bloquea un total de 10 bitcoins en una dirección de firmas múltiples de "al menos 2-de-2" firmas, controlada por Hitesh e Irene. La transacción de financiación también puede tener una o más salidas que devuelven el cambio a Hitesh y a Irene, si sus entradas exceden su contribución prevista al canal. Esta es una transacción única con entradas que han sido ofrecidas y firmadas por ambas partes. La misma tiene que ser redactada en colaboración y firmada por cada parte antes de ser difundida.

Ahora, en lugar de crear una transacción de compromiso única que firmarían ambas partes, Hitesh e Irene crean dos transacciones de compromiso diferentes, es decir *asimétricas*.

Hitesh tendrá una transacción de compromiso con dos salidas. La primera salida le paga a Irene los 5 bitcoins que ella posee *inmediatamente*. La segunda salida paga a Hitesh los 5 bitcoins que le pertenecen, pero solo tras madurar un bloqueo temporal de 1000 bloques. Las salidas de la transacción se ven así:

```
Entrada Única: "2-de-2" firmas (salida única de la transacción de financiación), firmada por Irene

Salida 0 <5 bitcoin>:
<Llave Pública de Irene> CHECKSIG

Salida 1:
<1000 bloques>
    CHECKSEQUENCEVERIFY
    DROP
    <Llave Pública de Hitesh> CHECKSIG
```

Irene tendrá una transacción de compromiso diferente, con dos salidas también. La primera salida paga a Hitesh los 5 bitcoins que le corresponden de inmediato. La segunda salida le paga a Irene sus 5 bitcoins, pero solo después de expirar un bloqueo temporal de 1000 bloques. La transacción de compromiso que tiene Irene (firmada por Hitesh) se ve así:

```
Entrada Única: "2-de-2" firmas (salida única de la transacción de financiación), firmada por Hitesh

Salida 0 <5 bitcoin>:
    <Llave Pública de Hitesh> CHECKSIG

Salida 1:
<1000 bloques>
    CHECKSEQUENCEVERIFY
    DROP
<Llave Pública de Irene> CHECKSIG
```

De esta manera, cada parte va a tener una transacción de compromiso, con una entrada capaz de gastar la salida de la transacción de financiación del tipo "al menos 2-de-2" firmas múltiples. Esta entrada está firmada por cada *contraparte*. En cualquier momento, cualquiera de las partes puede firmar esta transacción (completando el segundo componente de las "2-de-2 firmas") y difundirla. Sin embargo, al difundir esta transacción de compromiso, cada parte le está pagando a su

contraparte de inmediato, mientras que aquel que difunde la transacción tiene que esperar a que madure el bloqueo temporal. Al imponer un retraso en la redención de una de las salidas, ponemos a cada parte en una ligera desventaja cuando optan por difundir unilateralmente una transacción de compromiso. Pero tan solo un retraso de tiempo no será suficiente para alentar una conducta justa.

Dos transacciones de compromiso asimétrico con pago retrasado para la parte que difunde la transacción muestra dos transacciones de compromiso asimétrico, donde la salida que le paga al tenedor del compromiso se retrasa.

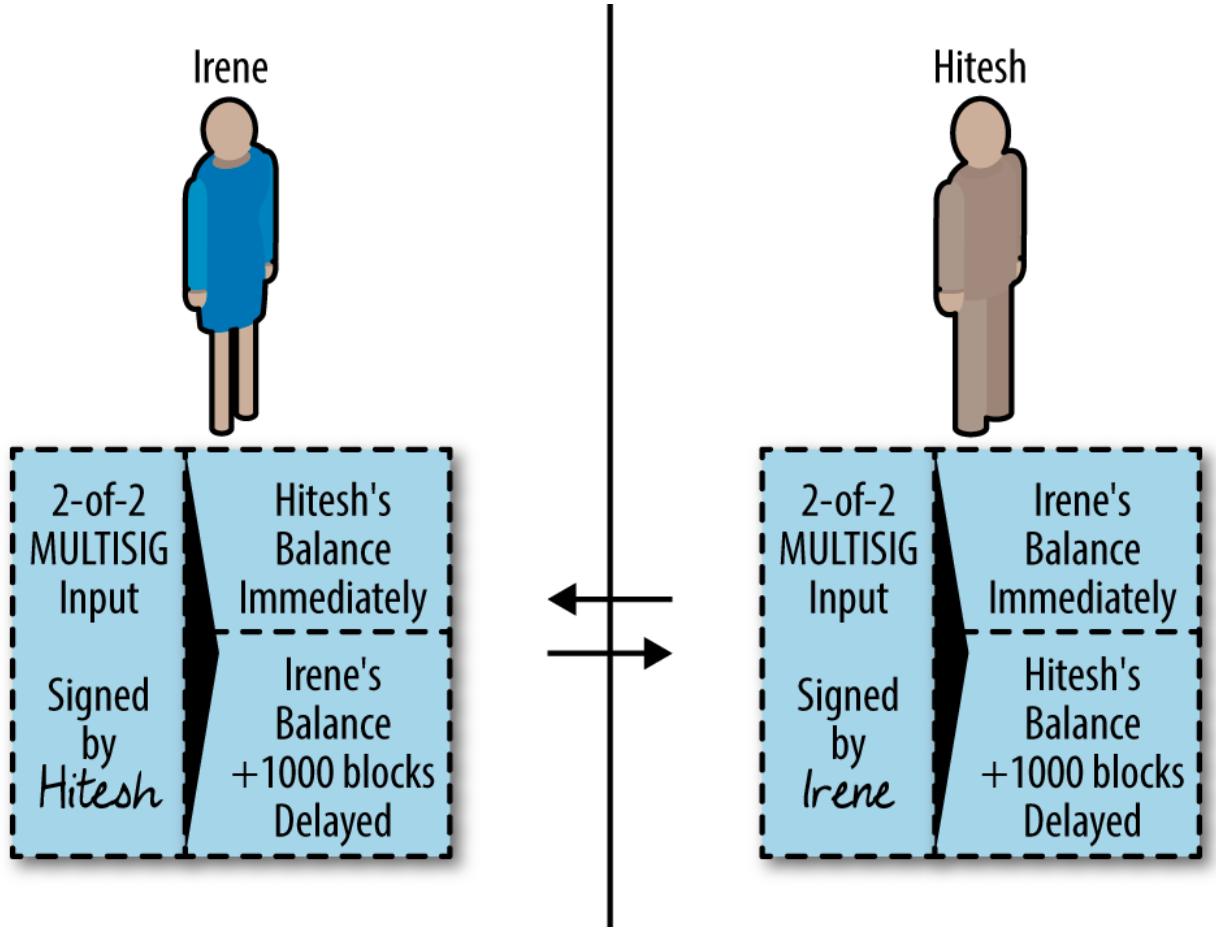


Figure 77. Dos transacciones de compromiso asimétrico con pago retrasado para la parte que difunde la transacción

A continuación será presentado el elemento final de este esquema: una llave de revocación para disuadir al tramposo de intentar difundir un compromiso vencido. La llave de revocación permite que la parte perjudicada castigue al tramposo, pues hace posible tomar todo el saldo del canal.

Las llaves de revocación se componen de dos secretos, cada uno generado independientemente por cada participante del canal. Es similar a una firma múltiple “2-de-2”, pero en este caso se usará aritmética de curva elíptica, de modo que ambas partes conozcan la llave pública de revocación, y solo accedan a la mitad de la llave secreta de revocación.

En cada ronda, ambas partes revelan su mitad del secreto de revocación a su contraparte, dándole así a la otra parte (que ahora tiene ambas mitades del secreto) los medios para gastar de inmediato la salida que tiene retrasos, como penalización si una transacción revocada alguna vez llega a difundirse.

Cada una de las transacciones de compromiso tiene una salida “retrasada”. El script de redención para esa salida permite que una de las partes logre gastarla después de 1000 bloques, *o bien* que la contraparte (y solo la contraparte) la gaste, si tienen la llave secreta de revocación completa, penalizando la difusión de un compromiso revocado.

Entonces, cuando Hitesh redacta una transacción de compromiso para que Irene la firme, hace que la segunda salida se pague a su favor después de 1000 bloques, o a favor de la correspondiente llave pública de revocación (de la cual él solo conocerá la mitad del secreto). Hitesh redacta esta transacción. Él solo revelará su mitad del secreto de revocación a Irene cuando esté listo para pasar a un nuevo estado del canal y quiera revocar este compromiso.

El script de la segunda salida se ve así:

```
Salida 0 <5 bitcoin>:
<Llave Pública de Irene> CHECKSIG

Salida 1 <5 bitcoin>:
IF
    # Salida de penalización por revocación
<Llave Pública de Revocación>
ELSE
<1000 bloques>
    CHECKSEQUENCEVERIFY
    DROP
    <Llave Pública de Hitesh>
ENDIF
CHECKSIG
```

Irene puede firmar con confianza esta transacción, ya que si se difunde, inmediatamente le pagará lo que se le debe. Hitesh retiene la transacción, pero sabe que si la difunde en un cierre de canal unilateral, tendrá que esperar 1000 bloques para recibir su pago.

Cuando el canal avanza hacia el siguiente estado, Hitesh tiene que *revocar* esta transacción de compromiso antes de que Irene acepte firmar la próxima transacción de compromiso. Para lograr esto, todo lo él que tiene que hacer es comunicar su mitad de la *llave secreta de revocación* a Irene. Una vez que Irene tiene las dos mitades de la llave secreta de revocación para este compromiso (la que ella produjo mas la que le acaba de entregar Hitesh), puede firmar el próximo compromiso con confianza. Ella sabe que si Hitesh intenta hacer trampa publicando este compromiso vencido, ella puede usar la llave de revocación para gastar a su favor la salida retrasada de Hitesh. *Si Hitesh hace trampa, Irene se queda con los fondos de AMBAS salidas*. Mientras tanto, Hitesh solo tiene la mitad del secreto para esta llave pública de revocación y no puede cobrar la segunda salida, hasta transcurridos 1000 bloques. Irene podrá cobrar dicha salida retrasada y castigar a Hitesh antes de que hayan transcurrido los 1000 bloques.

El protocolo de revocación es bilateral, lo que significa que en cada ronda, a medida que avanza el estado del canal, las dos partes intercambian nuevos compromisos, intercambian secretos de revocación por los compromisos anteriores y firman las nuevas transacciones de compromiso de cada uno. A medida que aceptan un nuevo estado, hacen que el estado anterior sea imposible de usar, dándose mutuamente los secretos de revocación necesarios para castigar cualquier trampa.

Veamos un ejemplo de cómo funciona. Uno de los clientes de Irene quiere enviar 2 bitcoins a uno de los clientes de Hitesh. Para transmitir 2 bitcoins a través del canal, Hitesh e Irene deben modificar el estado del canal para reflejar los nuevos saldos. Se comprometerán con un nuevo estado (estado número 2) donde se dividen los 10 bitcoins del canal, 7 bitcoins para Hitesh y 3 bitcoins para Irene. Para modificar el estado del canal, cada uno creará nuevas transacciones de compromiso que reflejen los nuevos saldos del mismo.

Como antes, estas transacciones de compromiso son asimétricas, de modo que la transacción de compromiso que tiene cada parte las obliga a esperar si la difunden. Es crucial que antes de firmar las nuevas transacciones de compromiso, primero se intercambien llaves de revocación para invalidar el compromiso anterior. En este caso particular, los intereses de Hitesh están alineados con el estado real del canal y, por lo tanto, no tiene ninguna razón para difundir un estado anterior. Sin embargo, para Irene, el estado número 1 la deja con un saldo más alto que el estado número 2. Cuando Irene le da a Hitesh la llave de revocación para su transacción de compromiso anterior (el estado número 1), efectivamente está revocando su capacidad para beneficiarse de regresar al canal a un estado anterior, ya que con la llave de revocación, Hitesh puede redimir ambas salidas de la transacción de compromiso anterior sin demora. Es decir, si Irene difunde el estado anterior, Hitesh puede ejercer su derecho a apoderarse de todas las salidas.

Es importante destacar que la revocación no ocurre automáticamente. Si bien Hitesh tiene la capacidad de castigar a Irene por hacer trampa, él tiene que mantenerse atento a la cadena de bloques diligentemente en busca de signos de trampa. Si ve una que una transacción de compromiso previa es difundida, tiene un lapso de tiempo correspondiente a 1000 bloques, para tomar medidas y usar la llave de revocación para frustrar el engaño de Irene y castigarla tomando el saldo completo, los 10 bitcoins.

Los compromisos revocables asimétricos con bloqueos temporales relativos (CSV) son una forma mucho mejor de implementar canales de pago y una innovación muy significativa en esta tecnología. Con este diseño, el canal puede permanecer abierto indefinidamente y puede tener miles de millones de transacciones de compromiso intermedias. En las implementaciones de pruebas de la plataforma Lightning Network, el estado de compromisos del canal se identifica mediante un índice de 48-bits, lo que permite más de 281 billones (2.8×10^{14}) de transiciones de estado en cualquier canal!

Contratos por Bloqueo Temporal y de Hash (del inglés, HTLC)

Los canales de pago pueden ampliarse aún más con un tipo especial de contrato inteligente, que permite a los participantes comprometer fondos a un secreto redimible, con un tiempo de vencimiento. Esta prestación se conoce como *Contrato por Bloqueo Temporal y de Hash*, o del inglés *HTLC* (Hash Time Lock Contract), y se usa en canales de pago bidireccionales y enrutados u orientados a conexión.

Pero primero se explicará el rasgo de bloqueo por "hash" de la prestación HTLC. Para crear un HTLC, el destinatario del pago creará primero un secreto R. Luego calculará el valor hash de este secreto H:

```
H = Hash(R)
```

Esto produce un valor hash H que se puede incluir en los script de bloqueo de alguna salida de una transacción. Quien conozca el secreto puede usarlo para redimir esa salida. El secreto R también se conoce como la *preimagen* de la función hash. La preimagen no es más que la data que se utiliza como parámetro de una función hash.

La segunda parte de un HTLC es el componente de "bloqueo temporal". Si no se revela el secreto, el pagador del HTLC puede obtener un "reembolso" después de un cierto tiempo. Esto se logra con un bloqueo de tiempo absoluto usando la instrucción transaccional CHECKLOCKTIMEVERIFY.

El código script que implementa un HTLC podría verse así:

```
IF
    # Proceso de pago a quien posea el secreto R
    HASH160 <H> EQUALVERIFY
ELSE
    # Reembolso en caso de expiración.
    <bloqueo temporal> CHECKLOCKTIMEVERIFY DROP
    <Llave Pública del Pagador> CHECKSIG
ENDIF
```

Cualquiera que conozca un secreto R, tal que cuando se le procesa con cierta función hash, devuelve el valor H, puede redimir esta salida ejecutando la primera cláusula de la estructura de control IF.

Si nadie revela este secreto para reclamar el pago del HTLC después de un cierto número de bloques, el pagador puede reclamar un reembolso utilizando la segunda cláusula de la estructura de control IF.

Esta es una implementación básica de un HTLC. Este tipo de HTLC puede ser redimido por *cualquier persona* que tenga el secreto R. Un HTLC puede tomar muchas formas diferentes con ligeras variaciones en el código script. Por ejemplo, agregar un operador transaccional CHECKSIG y una llave pública en la primera cláusula restringe la redención del hash a un destinatario pre-designado, que también debe conocer el secreto R.

Canales de Pago Enrutados (Lightning Network)

La plataforma Lightning Network es una red orientada a conexión compuesta por canales de pagos bidireccionales, conectados de extremo a extremo. Una red como esta puede permitir a cualquier participante enrutar un pago de un canal a otro sin la necesidad de confiar en ninguno de los intermediarios. Lightning Network fue por primera vez descrita por Joseph Poon y Thadeus Dryja en febrero de 2015, basándose en los conceptos de canales de pago, según se habían propuesto y elaborado por muchos otros.

"Lightning Network" se refiere a un diseño específico para una red de canales de pago enrutada, que ahora ha sido implementada por al menos cinco equipos de código abierto diferentes.Los desarrollos independientes están coordinados por un conjunto de estándares de interoperabilidad descritos en el paper del protocolo *Basics of Lightning Technology (BOLT)*.

Varios equipos han lanzado implementaciones de prueba para Lightning Network.

Lightning Network es una forma posible de implementar canales de pago enrutados. Hay varios otros diseños que apuntan a lograr objetivos similares, como Teechan y Tumblebit.

Ejemplo básico usando Lightning Network

Veamos cómo funciona esta red.

En este ejemplo, tenemos cinco participantes: Alice, Bob, Carol, Diana y Eric. Estos cinco participantes han abierto canales de pago entre ellos, en pares. Alice tiene un canal de pagos con Bob. Bob está conectado con Alice y con Carol, Carol con Bob y Diana y Diana con Carol y Eric. Para simplificar, supongamos que cada canal está financiado con 2 bitcoins por cada participante por canal, para una capacidad total de 4 bitcoins en cada canal.

Una serie de canales de pago bidireccionales vinculados para formar una Lightning Network que puede enrutar un pago desde Alice hasta Eric muestra cinco participantes en una red Lightning Network, conectados por canales de pago bidireccionales que se pueden vincular para realizar un pago de Alice a Eric (Canales de Pago Enrutados (Lightning Network)).

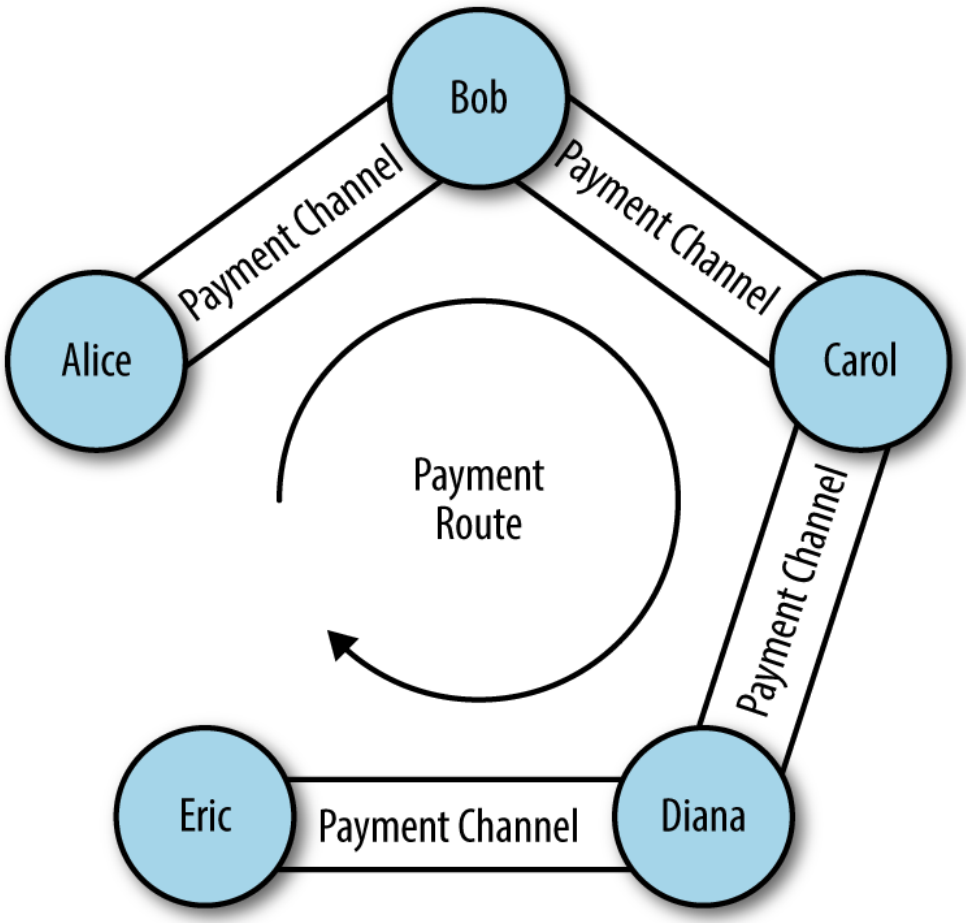


Figure 78. Una serie de canales de pago bidireccionales vinculados para formar una Lightning Network que puede enrutar un pago desde Alice hasta Eric

Alice quiere pagarle a Eric 1 bitcoin. Sin embargo, Alice no está conectada con Eric por un canal de pago. La creación de un canal de pago requiere una transacción de financiación, que debe comprometerse con la cadena de bloques de bitcoin. Alice no quiere abrir un nuevo canal de pago y comprometer más de sus fondos. ¿Hay alguna forma de pagarle a Eric, indirectamente?

Enrutamiento de pagos paso-a-paso a través de una red Lightning Network muestra el proceso paso a paso de enrutar un pago de Alice a Eric, a través de una serie de compromisos HTLC en los canales de pago que conectan a los participantes.

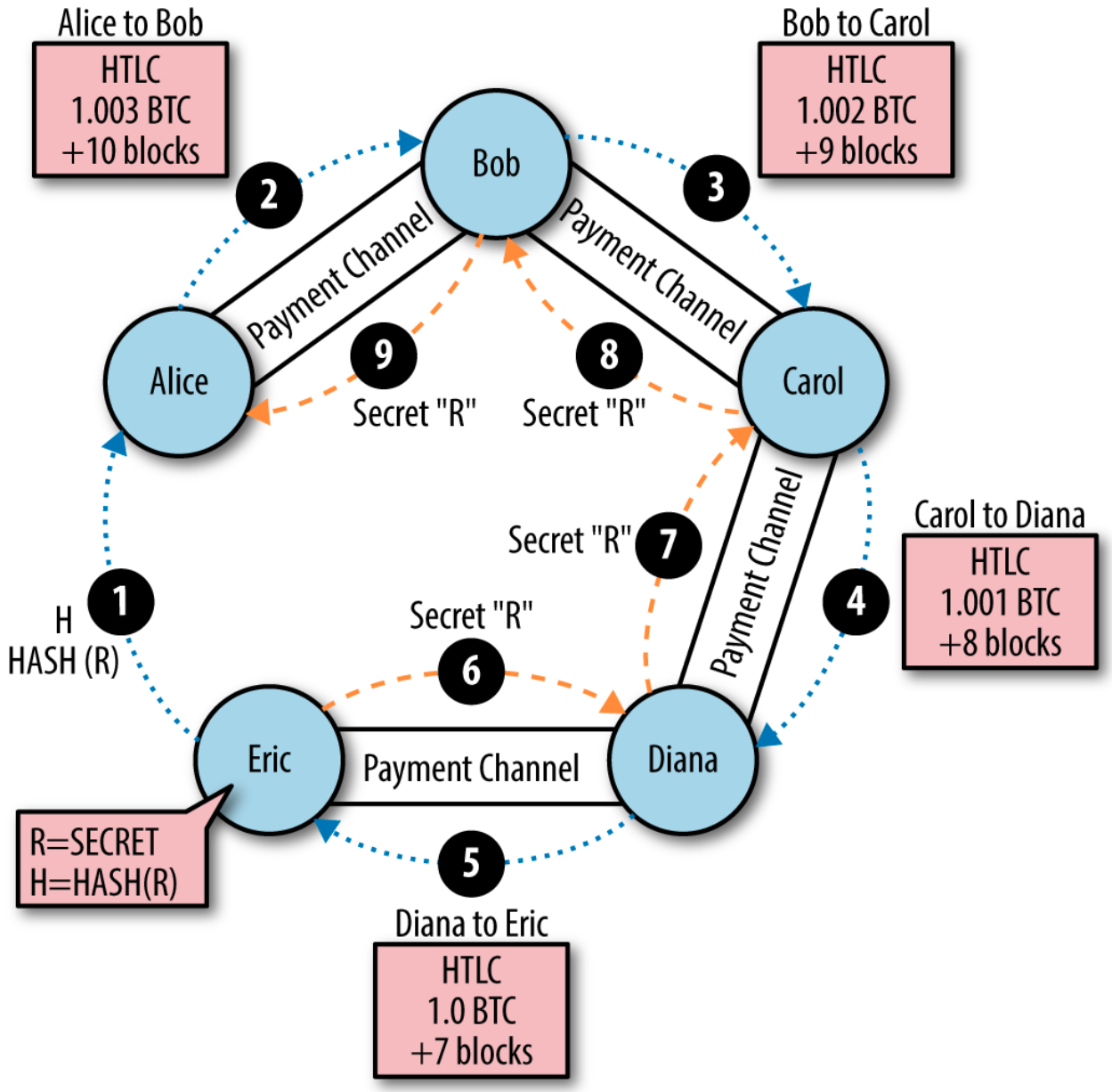


Figure 79. Enrutamiento de pagos paso-a-paso a través de una red Lightning Network

Alice está ejecutando un nodo de Lightning Network, o "LN", que es capaz de realizar un seguimiento de su canal de pagos a Bob y además puede descubrir rutas entre otros canales de pagos. El nodo LN de Alice también tiene la capacidad de conectarse a través de Internet al nodo LN de Eric. El nodo LN de Eric crea un secreto R utilizando un generador de números aleatorios. El nodo de Eric no revela este secreto a nadie. En cambio, el nodo de Eric calcula el valor hash H del secreto R y transmite este valor hash al nodo de Alice (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 1).

Ahora el nodo LN de Alice construye una ruta entre su nodo LN y el nodo LN de Eric. El algoritmo de enrutamiento utilizado se examinará con más detalle más adelante, pero por ahora supongamos que al nodo de Alice le es posible encontrar una ruta eficiente.

El nodo de Alice luego redacta un protocolo HTLC, pagadero al hash H, con un tiempo de espera de reembolso de 10 bloques (bloque actual + 10), por una cantidad de 1.003 bitcoins (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 2). El valor 0.003 adicional se usará para compensar a los nodos intermedios por su participación en esta ruta de pago. Alice ofrece este HTLC a Bob, deduciendo 1.003 bitcoins del saldo de su canal con Bob y comprometiéndolo con el HTLC. El HTLC tiene el siguiente significado: *"Alice está comprometiendo 1.003 bitcoins del saldo de su canal a Bob si Bob conoce el secreto, o a que se le reembolse a su favor este saldo si transcurren 10 bloques o más."* Los saldos del canal entre Alice y Bob son ahora expresados por transacciones de compromiso con tres salidas: un saldo de 2 bitcoins para Bob, un saldo de 0,997 bitcoins para Alice y de 1,003 bitcoins comprometidos en el contrato HTLC de Alice. El saldo de Alice se reduce por la cantidad comprometida con el contrato HTLC.

Bob ahora tiene un compromiso que promete que si puede obtener el secreto R dentro de los próximos 10 bloques, puede reclamar los 1.003 bitcoins bloqueados por Alice. Con este compromiso en la mano, el nodo de Bob redacta un nuevo contrato HTLC en su canal de pago con Carol. El HTLC de Bob compromete 1.002 bitcoins al hash H por un lapso de 9 bloques, que Carol puede redimir si averigua el secreto R (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 3). Bob sabe que si Carol puede redimir su HTLC, ella tiene que producir de algún modo a R. Si Bob obtiene R en menos de nueve bloques, puede usarlo para reclamar el saldo del HTLC de Alice. También ganará 0.001 bitcoins por comprometer el saldo de su canal durante nueve bloques. Si Carol no puede reclamar su HTLC y él no podrá reclamar el HTLC de Alice, todo vuelve a los saldos de canales anteriores y nadie está perdiendo. El balance del canal entre Bob y Carol ahora es: 2 bitcoins para Carol, 0.998 para Bob y 1.002 comprometidos por Bob en su particular HTLC.

Carol ahora tiene un compromiso que promete que si obtiene de algún modo el secreto R dentro de los próximos nueve bloques o menos, ella puede reclamar 1.002 bitcoins bloqueados por Bob. Ahora puede hacer un compromiso HTLC propio en su canal con Diana. Ella puede comprometer un HTLC de 1.001 bitcoin a favor de quien posea el secreto que produce el hash H, por un lapso de ocho bloques; compromiso que Diana puede redimir si obtiene el secreto R (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 4). Desde la perspectiva de Carol, si esto funciona, ella podría ganarse 0.001 bitcoins y si no funciona, no pierde nada. Su oferta HTLC para con Diana solo será viable si en algún momento el secreto R es revelado, en cuyo momento podrá reclamar el HTLC entre ella y Bob. Los saldos del canal entre Carol y Diana serán ahora: 2 bitcoins para Diana, 0.999 para Carol y 1.001 comprometidos por Carol para su HTLC.

Finalmente, Diana puede redactar un HTLC a Eric, comprometiendo 1 bitcoin por siete bloques a favor del hash H (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 5). Los saldos del canal entre Diana y Eric son ahora: 2 bitcoins para Eric, 1 bitcoin para Diana y 1 bitcoin comprometido por Diana en su HTLC.

Sin embargo, en este salto final de la ruta, Eric *es el dueño* del secreto R. Por lo cual él puede inmediatamente reclamar el HTLC ofrecido por Diana. Él envía así el valor R a Diana y reclama el bitcoin de ese HTLC, y lo agrega a su favor en los saldos de su canal con Diana (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 6). Los saldos de este canal son ahora: 1 bitcoin para Diana y 3 para Eric.

Ahora que Diana posee el secreto R, puede reclamar el HTLC ofrecido por Carol. Diana transmite R a Carol y agrega el valor de 1.001 bitcoins a su saldo del canal (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 7). Ahora los saldos del canal entre Carol y Diana son: 0.999 bitcoins para Carol y 3.001 para Diana. Diana ha "ganado" 0.001 bitcoin por participar en esta ruta de pago.

Volviendo hacia atrás a través de la ruta, el secreto R permite a cada participante reclamar los fondos de los HTLC pendientes. Carol reclama 1.002 de Bob, estableciendo en su canal con él los siguientes saldos: 0.998 bitcoins a Bob y 3.002 a Carol (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 8). Finalmente, Bob reclama el HTLC ofrecido por Alice (ver Enrutamiento de pagos paso-a-paso a través de una red Lightning Network paso 9). Así, los saldos de su canal con ella se actualizan a: 0.997 bitcoins para Alice y 3.003 para Bob.

Alice le ha pagado a Eric 1 bitcoin sin tener que abrir un canal con Eric. Ninguna de las partes intermedias en la ruta de pago tuvo que confiar entre sí. Por el compromiso a corto plazo de sus fondos en el canal, pueden ganar una pequeña tarifa, con el único riesgo de que se produzca un pequeño retraso en el reembolso si el canal se cerró o el pago enrutado falló.

Transporte y Enrutamiento de Lightning Network

Todas las comunicaciones entre nodos LN están encriptadas de punto a punto. Además, los nodos tienen una llave pública a largo plazo que utilizan como identificador y para autenticarse entre sí.

Siempre que un nodo desee enviar un pago a otro nodo, primero debe establecer una *ruta* a través de la red conectando canales de pago con capacidad suficiente. Los nodos anuncian la información de enrutamiento, incluidos qué canales tienen abiertos, cuánta capacidad tiene cada canal y qué comisiones cobran para enrutar los pagos. La información de enrutamiento se puede compartir de varias maneras y es probable que surjan diferentes protocolos de enrutamiento a medida que avanza la tecnología de Lightning Network. Algunas implementaciones de Lightning Network usan el protocolo IRC (el "Internet Relay Chat") como un mecanismo conveniente para que los nodos anuncien información de enrutamiento. Otra implementación de la exploración de rutas, utiliza un modelo P2P donde los nodos propagan anuncios de sus canales a sus pares, en un modelo de "inundación", similar a cómo Bitcoin propaga sus transacciones. Los planes futuros incluyen una propuesta llamada Flare, que es un modelo de enrutamiento híbrido con "vecindarios" de nodos locales y nodos de balizaje de mayor alcance.

En nuestro ejemplo anterior, el nodo de Alice usa uno de estos mecanismos de exploración de rutas para encontrar una o más rutas que conecten su nodo con el nodo de Eric. Una vez que el nodo de Alice haya establecido una ruta, ella inicializará esa ruta a través de la red, propagando una serie de instrucciones encriptadas y anidadas para conectar cada uno de los canales de pago adyacentes.

Es importante destacar que esta ruta solo es conocida por el nodo de Alice. Todos los demás participantes en la ruta de pago solo ven los nodos adyacentes. Desde la perspectiva de Carol, esto parece un pago de Bob a Diana. Carol no sabe que Bob está retransmitiendo un pago de Alice. Tampoco sabe que Diana le enviará un pago a Eric.

Esta es una característica crítica de Lightning Network, ya que garantiza la privacidad de los pagos y hace que sea muy difícil aplicar la vigilancia, la censura o las listas negras. Pero, ¿cómo establece Alice esta ruta de pago, sin revelar nada a los nodos intermediarios?

Lightning Network implementa un protocolo de enrutamiento de cebollas basado en un esquema llamado Sphinx. Este protocolo de enrutamiento garantiza que un remitente de pagos pueda construir y comunicar una ruta a través de Lightning Network de manera que:

- Los nodos intermedios pueden verificar y descifrar su porción de información de ruta y encontrar el próximo salto.
- Aparte de los saltos anteriores y siguientes, no pueden averiguar nada más sobre ningún otro nodo que forme parte de la ruta.
- No pueden identificar la longitud de la ruta de pago ni su propia posición en esa ruta.
- Cada parte de la ruta está encriptada de tal manera que un atacante a nivel de la red no podrá asociar los paquetes de diferentes partes de la ruta entre sí.
- A diferencia del protocolo Tor (un protocolo de anonimización de enrutado por cebollas en Internet), no hay "nodos terminales" (o “exit nodes”) que se puedan poner bajo vigilancia. No es necesario que los pagos se transmitan a la cadena de bloques de bitcoin; los nodos solo actualizan los saldos de sus canales.

Usando este protocolo enrutado por cebollas, Alice envuelve cada elemento de la ruta en una capa de encriptación, comenzando por el final y procediendo hacia atrás en la ruta. Ella encripta un mensaje a Eric con la llave pública de Eric. Este mensaje está envuelto en un mensaje cifrado a Diana, que identifica a Eric como el próximo destinatario. El mensaje a Diana se envuelve en un mensaje encriptado con la llave pública de Carol e identifica a Diana como el próximo destinatario. El mensaje a Carol se envuelve con la llave de Bob. Por lo tanto, Alice ha construido esta "cebolla" de mensajes multicapas encriptada. Ella le envía esto a Bob, quien solo puede descifrar y desenvolver la capa más externa. Adentro, Bob encuentra un mensaje dirigido a Carol que puede reenviar a Carol pero no que puede descifrar. Siguiendo el camino, los mensajes se reenvían, descifran, reenvían, etc., hasta llegar a Eric. Cada participante conoce solo el nodo anterior y el siguiente en cada salto.

Cada elemento de la ruta contiene información sobre el protocolo HTLC que debe extenderse al siguiente salto, la cantidad que se envía, la tarifa a incluir y el tiempo de bloqueo de CLTV (CHECKLOCKTIMEVERIFY en bloques) al vencimiento del HTLC. A medida que se propaga la información de la ruta, los nodos se comprometen con HTLC que yace en el próximo salto.

En este punto, es posible que surja la duda de cómo es posible que los nodos no conozcan la longitud de la ruta y su posición en esa ruta. Después de todo, reciben un mensaje y lo reenvían al siguiente salto. ¿No se debería acortar progresivamente el mensaje, permitiendo deducir el tamaño del camino y su posición? Para evitar esto, la ruta siempre se fija en 20 saltos y se rellena con datos aleatorios. Cada nodo ve el siguiente salto y un mensaje cifrado de longitud fija para reenviar. Solo el destinatario final ve que no hay próximo salto. Para todos los demás, parece que siempre quedan 20 saltos más.

Ventajas de Lightning Network

Una plataforma Lightning Network consiste en una tecnología de enrutamiento de segunda capa. Se puede aplicar sobre cualquier cadena de bloques que admita algunas capacidades básicas, como transacciones multifirma, bloqueos de tiempo y contratos inteligentes básicos.

Si una red Lightning Network se superpone como capa de servicios a la red bitcoin, la red bitcoin puede aumentar significativamente su capacidad, privacidad, granularidad y velocidad, sin sacrificar los principios de su operación desconfiable y sin intermediarios:

Privacidad

Los pagos de Lightning Network son mucho más privados que los pagos en la cadena de bloques de bitcoin, ya que no son públicos. Si bien los participantes en una ruta pueden ver los pagos propagados a través de sus canales, no conocen al remitente o al destinatario.

Fungibilidad

Una plataforma Lightning Network hace que sea mucho más difícil aplicar cualquier tipo de vigilancia así como listas negras en bitcoin, aumentando la fungibilidad de la moneda.

Velocidad

Las transacciones de Bitcoin que utilizan Lightning Network se liquidan en milisegundos, en lugar de minutos, ya que los HTLC se verifican sin comprometer las transacciones en un bloque de la red Bitcoin.

Granularidad

Una plataforma Lightning Network puede permitir pagos al menos tan pequeños como el límite ínfimo (o de "polvo") de bitcoin, quizás incluso más pequeños. Algunas propuestas permiten incrementos en subsatoshis.

Capacidad

Una plataforma Lightning Network aumenta la capacidad del sistema bitcoin en varios órdenes de magnitud. No existe un límite superior práctico para la cantidad de pagos por segundo que se pueden enrutar a través de una plataforma Lightning Network, ya que depende solo de la capacidad y la velocidad de cada nodo.

Operatividad Desconfiable

Una plataforma Lightning Network utiliza transacciones de bitcoin entre nodos que operan como pares sin depender en ningún tipo de confianza el uno del otro. Por lo tanto, una plataforma Lightning Network conserva los principios del sistema bitcoin, al tiempo que amplía significativamente sus parámetros operativos.

Por supuesto, como se mencionó anteriormente, el protocolo Lightning Network no es la única forma de implementar canales de pago enrutados. Otros sistemas propuestos incluyen Tumblebit y Teechan. En este momento, sin embargo, Lightning Network ya se ha implementado en la red de pruebas “testnet”. Varios equipos diferentes han desarrollado implementaciones competitivas de LN y están trabajando en pro de un estándar de interoperabilidad común (llamado BOLT). Es probable que Lightning Network sea la primera red de canales de pago enrutada que se implementará en producción.

Conclusión

Hemos examinado solo algunas de las aplicaciones emergentes que se pueden construir utilizando la cadena de bloques de bitcoin como plataforma de confianza. Estas aplicaciones amplían el alcance de bitcoin más allá de los pagos y más allá de los instrumentos financieros, para abarcar muchas otras aplicaciones donde la confianza es crítica. Al descentralizar la base de la confianza, la cadena de bloques de bitcoin funge de plataforma que generará muchas aplicaciones revolucionarias en una amplia variedad de industrias.

Appendix A: El Whitepaper (Libro Blanco) de Bitcoin por Satoshi Nakamoto

NOTE Este es el whitepaper original, reproducido exactamente como lo publicó Satoshi Nakamoto en Octubre del 2008.

Bitcoin: Una Plataforma de Efectivo Electrónico Entre Pares Iguales

Satoshi Nakamoto

satoshin@gmx.com

www.bitcoin.org

Abstract. Una versión de efectivo electrónico, verdaderamente entre iguales (peer-to-peer), posibilitaría realizar pagos en línea que vayan directamente de un lado al otro sin la mediación de una institución financiera. Las firmas digitales proporcionarían parte de la solución, pero los principales beneficios se perderían si todavía se requiriera de la participación de un tercero confiable para evitar el doble gasto. Proponemos una solución al problema del doble gasto empleando una red peer-to-peer. Las transacciones en una red a las que se les asigna un sello de tiempo mediante la aplicación de un algoritmo hash antes de insertarlas en una cadena de valores hash de una secuencia de Prueba-de-Trabajo, conformarían de esta manera un registro que no podría modificarse sin volverse a generar otra Prueba-de-Trabajo. La cadena más larga no solo constituiría una prueba de la secuencia de eventos atestiguados, sino también una prueba de que esta secuencia es resultante de emplear a la agrupación con el mayor poder de cómputo. Mientras que la mayor parte del poder de cómputo esté controlada por nodos que no están intentando atacar a la red, éstos generarán la cadena más larga y mantendrán a raya a cualquier atacante. La red en sí misma requiere una estructura mínima. Los mensajes se difunden con base a un esfuerzo optimizado, y los nodos pueden salirse y volver a conectarse a la red a voluntad, aceptando la cadena con la Prueba-de-Trabajo más larga como prueba de lo que sucedió mientras estaban desconectados.

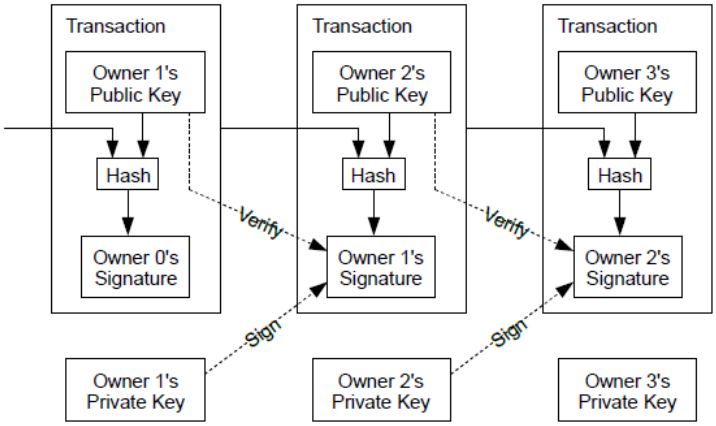
Introducción

El comercio en Internet se ha basado casi exclusivamente en instituciones financieras que actúan como terceros de confianza para procesar pagos electrónicos. Si bien el sistema funciona lo suficientemente bien para la mayoría de las transacciones, todavía sufre las debilidades inherentes de un modelo basado en la confianza. En estas circunstancias, es imposible la entrada en escena de transacciones completamente irreversibles, ya que las instituciones financieras no pueden evitar la mediación en disputas. Los gastos de tal mediación aumentan los costos de cada transacción, limitando el tamaño mínimo que puede tener una transacción en la práctica, así como también, limitando la posibilidad de pequeñas transacciones informales o más espontáneas, y se incurre en un riesgo mas costoso cuando se pierde de la capacidad de realizar pagos irreversibles cuando se paga por servicios sí son irreversibles. Con la posibilidad de reversar pagos, la dependencia en la confianza se intensifica. Los comerciantes deben estar muy atentos de sus clientes, acosándoles con el fin de obtener información de ellos, que de otro modo sería innecesaria. Un cierto nivel de fraude se acepta como un hecho inevitable de la vida. Estos costos e incertidumbres en los pagos se pueden evitar cuando el comercio se realiza en persona y mediante el uso de moneda física, pero no existe ningún mecanismo para realizar pagos a través de un canal de comunicaciones sin que sea necesaria la necesidad de confiar en un tercero.

Lo que se necesita es un sistema de pagos electrónicos basado en pruebas criptográficas, en lugar de basarse en la confianza, y que permita a las dos partes interesadas, llevar a cabo transacciones directas entre sí sin la necesidad de un tercero de confianza. Las transacciones que son computacionalmente poco prácticas para revertirse protegerían a los vendedores del fraude, y los mecanismos de custodia de rutina podrían implementarse fácilmente para proteger a los compradores. En este documento, proponemos una solución al problema del gasto duplicado utilizando un servidor distribuido con sello de tiempo entre pares de igual jerarquía, para generar pruebas computacionales del orden cronológico en que se dan las transacciones. El sistema es seguro siempre que los nodos honestos controlen colectivamente la mayor cantidad de potencia de CPU que cualquier otro grupo de nodos confabulados con los fines de un ataque.

Transacciones

Definimos una moneda electrónica como una cadena de firmas digitales. Cada propietario transfiere la moneda al siguiente o nuevo propietario, firmando digitalmente un hash de la transacción anterior y la llave pública del siguiente propietario y agregándolas al final de dicha moneda. Cualquier beneficiario puede verificar las firmas para comprobar la cadena de propietarios.

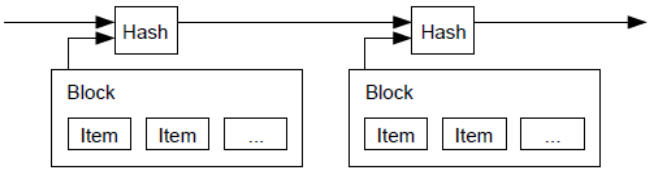


El problema, por supuesto, es que el beneficiario no puede verificar si uno de los propietarios no ha gastado dos veces la misma moneda. Una solución común es introducir una autoridad central de confianza, o casa de moneda, que verifica que cada transacción no tenga doble gasto. Después de cada transacción, la moneda debe devolverse a la casa de moneda para emitir una moneda nueva, y solo se confía en que las monedas emitidas directamente desde la casa de la moneda no se gasten dos veces. El problema con esta solución es que el destino de todo el sistema monetario depende de la compañía que administra la casa de la moneda, y cada transacción tiene que pasar por ellos, al igual que un banco.

Necesitamos una forma de que el beneficiario sepa que los propietarios previos no han firmado transacciones anteriores. Para nuestros propósitos, la transacción más temprana es la que cuenta, así que no nos preocupamos de los intentos de doble gasto posteriores. La única manera de confirmar la ausencia de una transacción es tener conocimiento de todas las transacciones. En el modelo de la casa de la moneda, esta tiene conocimiento de todas las transacciones y decide cuáles llegaron primero. Para lograr esto sin la participación de una parte de confianza, las transacciones han de ser anunciadas públicamente [1], y necesitamos un sistema para que los participantes estén de acuerdo en un único historial del orden en que fueron recibidas. El beneficiario necesita una prueba de que en el momento de la transacción la mayor parte de los nodos estaban de acuerdo en que esa fue la primera que se recibió.

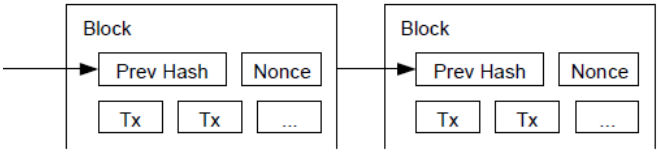
Servidor de sello de tiempo

La solución que proponemos comienza con un servidor de sello de tiempo. Un servidor de sello de tiempo trabaja tomando el hash de un bloque de elementos para sellarlos en el tiempo y notificar públicamente su hash, como un periódico o un post Usenet [2-5]. El sello de tiempo prueba que los datos han existido en el tiempo, obviamente, para entrar en el hash. Cada sello de tiempo incluye el sello de tiempo previo en su hash, formando una cadena, con cada sello de tiempo adicional reforzando a los que estaban antes.



Prueba-de-Trabajo

Para implementar un servidor de sello de tiempo distribuido entre pares de la misma jerarquía, necesitaremos emplear un sistema de Prueba-de-Trabajo similar al Hashcash de Adam Back [6], en lugar de los posts de los periódicos o las Usenet. La Prueba-de-Trabajo consiste en escanear en busca de un valor que cuando fue hasheado, tal y como se hace con SHA-256, el hash comience con cierto número de cero bits. El trabajo promedio que hace falta es exponencial y crece con el número de bits con valor cero requeridos y dicho trabajo puede verificarse ejecutando un único hash. Para nuestra red de sello de tiempo, implementamos la Prueba-de-Trabajo incrementando un nonce en un bloque de datos, hasta que se encuentre un valor que dé al hash del bloque el número de bits con valor cero requeridos. Una vez que se ha agotado el esfuerzo de CPU para satisfacer la Prueba-de-Trabajo, el bloque no se puede modificar sin rehacer el trabajo. A medida que los bloques posteriores se encadenen tras él, el trabajo para cambiar un bloque incluiría rehacer todos los bloques siguientes.



La Prueba-de-Trabajo (de sus siglas en inglés, "pow") también resuelve el problema de determinar la representación en la toma de decisiones mayoritarias. Si la mayoría se basara en que a una dirección IP, correspondiese un voto, cualquiera podría ser capaz de asignar muchas IP's. La Prueba-de-Trabajo representa esencialmente el criterio de una CPU, un voto. La decisión mayoritaria está representada por la cadena más larga, que tiene el mayor esfuerzo de Prueba-de-Trabajo invertido en ella. Si la mayoría de la potencia de la CPU está controlada por nodos honestos, la cadena honesta crecerá más rápido y superará a las cadenas de la competencia. Para modificar un bloque pasado, un atacante tendría que rehacer la Prueba-de-Trabajo del bloque y todos los bloques posteriores y luego ponerse al día y superar el trabajo de los nodos honestos. Más adelante mostraremos que la probabilidad de que un atacante más lento se recupere disminuye exponencialmente a medida que se agregan bloques posteriores.

Para compensar el incremento en la velocidad del hardware y los cambios de interés por ejecutar nodos, la dificultad de la Prueba-de-Trabajo, PoW, se determina como un promedio dinámico que hace seguimiento al número medio de bloques que se producen por hora. Si son generados demasiado rápido, la dificultad aumentará.

Red

Los pasos para ejecutar la red son los siguientes:

1. Las nuevas transacciones se transmiten a todos los nodos.
2. Cada nodo recopila nuevas transacciones en un bloque.
3. Cada nodo trabaja para encontrar una Prueba-de-Trabajo difícil para su bloque.
4. Cuando un nodo encuentra una Prueba-de-Trabajo, transmite el bloque a todos los nodos.
5. Los nodos aceptan el bloque solo si todas las transacciones en él son válidas y no están gastadas.
6. Los nodos expresan su aceptación del bloque trabajando en la creación del siguiente bloque en la cadena, utilizando el hash del bloque aceptado como el hash anterior.

Los nodos siempre consideran que la cadena más larga es la correcta y seguirá trabajando para extenderla. Si dos nodos transmiten versiones diferentes del siguiente bloque simultáneamente, algunos nodos pueden recibir uno u otro primero. En ese caso, trabajan en el primero que recibieron, pero guardan la otra rama en caso de que ésta se haga más larga. El empate se romperá cuando se encuentre la próxima prueba-de-trabajo y una de las ramas se haga la más larga; los nodos que estaban trabajando en la otra rama luego cambiarán a la más larga .

Las nuevas difusiones de transacciones no necesariamente tienen que llegar a todos los nodos. Siempre y cuando lleguen a muchos nodos, pronto entrarán en algún bloque. Las difusiones de los bloques también toleran mensajes perdidos. Si un nodo no recibe un bloque, lo solicitará cuando reciba el siguiente bloque y se dé cuenta de que perdió uno.

Incentivos

Por convención, la primera transacción en un bloque es una transacción especial que crea una nueva moneda que es propiedad del creador del bloque. Esto agrega un incentivo para que los nodos apoyen la red y proporciona una manera de distribuir nuevas monedas para que circulen, ya que no hay una autoridad central para emitirlas. La adición continua de una cantidad constante de monedas nuevas es análoga a los mineros de oro que gastan recursos para añadir más oro en circulación. En nuestro caso, es el tiempo de uso del CPU y la electricidad que se gasta.

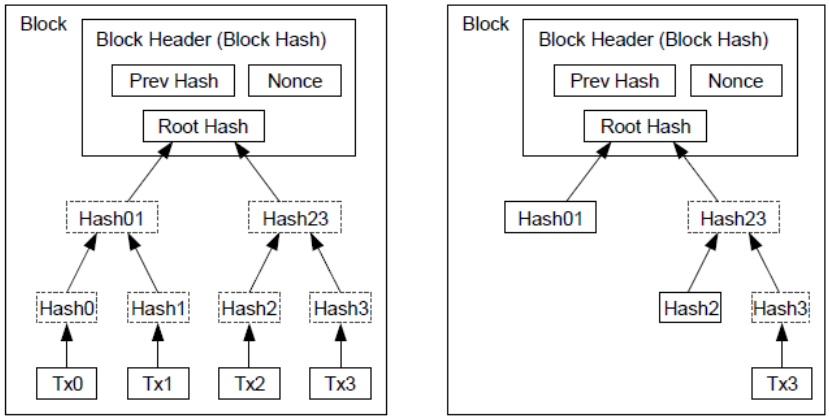
El incentivo también se puede financiar con comisiones en las transacciones. Si el valor de salida de una transacción es menor que su valor de entrada, la diferencia es una comisión de transacción que se agrega al valor de incentivo del bloque que contiene la transacción. Una vez que un número predeterminado de monedas ha entrado en circulación, el incentivo pasar a ser completamente consistente, en sólo comisiones de transacción, dejando al sistema completamente libre de inflación.

El incentivo puede ayudar a alentar a los nodos a ser honestos. Si un atacante codicioso es capaz de acaparar más potencia de CPU que todos los nodos honestos, tendría que elegir entre usarla para defraudar a las personas robando sus pagos, o usarla para generar nuevas monedas. Este atacante debería encontrar más rentable la opción de apelar jugando según las reglas, reglas que lo favorecen con muchas más monedas nuevas que las que puedan acuñar todos los demás

jugadores combinados, en lugar de socavar el sistema y la validez de su propia riqueza.

Recuperar espacio en disco

Una vez que la última transacción referente a una moneda, está enterrada bajo suficientes bloques, las transacciones gastadas antes de esta última pueden descartarse para ahorrar espacio en disco. Para facilitar esto sin corromper el hash de los bloques, las transacciones pueden agruparse en un árbol de Merkle [7] [2] [5], con solo la raíz incluida en el hash del bloque. Los bloques viejos, pueden de este modo compactarse, podando ramas del árbol. Los valores hash internos no necesitan ser almacenados.

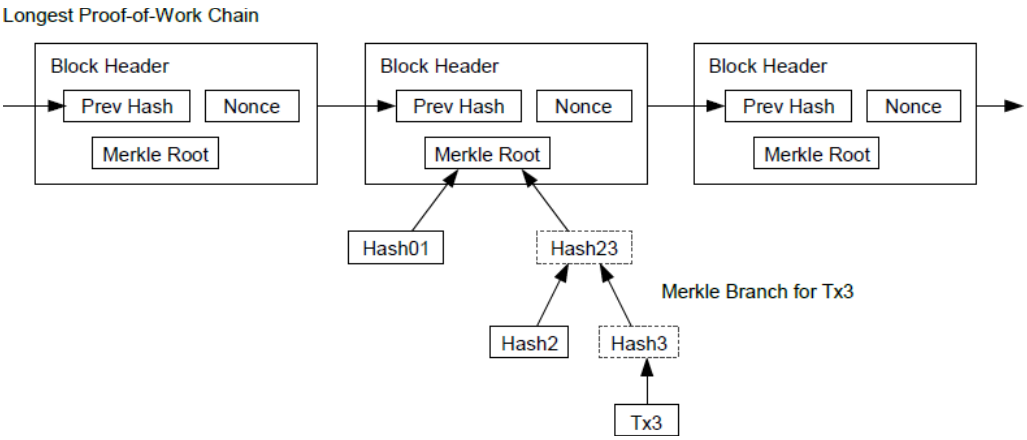


Transactions Hashed in a Merkle Tree — After Pruning Tx0-2 from the Block

Una cabecera de bloque sin transacciones tendría aproximadamente 80 bytes. Si suponemos que los bloques se generan cada 10 minutos, 80 bytes * 6 * 24 * 365 = 4.2MB por año. Con sistemas informáticos que normalmente se venden con 2 GB de RAM tal como en 2008, y tomando en cuenta la Ley de Moore que predice un crecimiento actual de 1.2 GB por año, el almacenamiento no debería ser un problema, incluso si las cabeceras de bloque deben mantenerse en la memoria.

Verificación de Pago Simplificado

Es posible verificar pagos sin ejecutar un nodo de red completo. Un usuario solo necesita conservar una copia de las cabeceras de bloque de la cadena de Prueba-de-Trabajo más larga, que puede obtener al consultar los nodos de la red hasta que esté convencido de que tiene la cadena más larga, y obtener la rama de Merkle que vincula a la transacción que refleja el pago, al bloque en que ésta posee un sello de tiempo. Este nodo no puede verificar la transacción por sí mismo, pero al vincularla a un lugar de la cadena, puede ver que un nodo de la red sí que la ha aceptado, y al ver que se han agregado más bloques después de ello, se confirma con suficiente redundancia que la red ha aceptado dicha transacción.

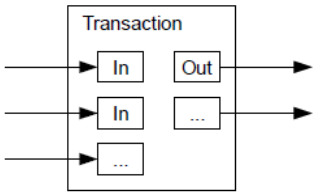


Como tal, la verificación es confiable siempre que los nodos honestos controlen la red, pero es más vulnerable si la red es dominada por un atacante. Si bien los nodos de la red pueden verificar las transacciones por sí mismos, el método simplificado puede ser engañado por las transacciones fabricadas por un atacante mientras el atacante pueda seguir dominando la red. Una estrategia para protegerse contra esto sería aceptar alertas de los nodos de la red cuando detecten un bloqueo no válido, lo que haría que el software del usuario descargue el bloque completo y las transacciones alertadas para confirmar la inconsistencia. Las empresas que reciban pagos frecuentes probablemente aún deseen ejecutar sus propios nodos completos para una seguridad más independiente y una verificación más rápida.

Combinando y Separando Valores

Aunque sería posible manejar monedas individualmente, sería difícil realizar una transacción por cada centavo en una

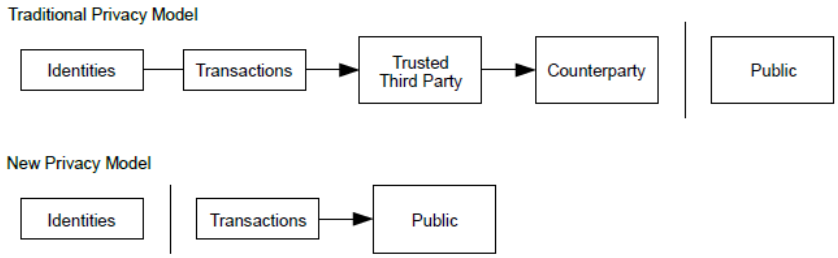
transferencia. Para permitir que el valor se divida y se combine, las transacciones contienen múltiples entradas y múltiples salidas. Comúnmente podrá haber o bien una sola entrada de una transacción previa más grande o bien múltiples entradas que combinen cantidades más pequeñas, y como máximo dos salidas: una para el pago y otra que devuelva el cambio, si corresponde, al remitente.



Cabe señalar que en ciertos despliegues, en donde una transacción depende de varias transacciones, y esas transacciones dependen a su vez de muchas más, no representa un problema aquí. Nunca será necesario extraer una copia completa e independiente de todo el historial de una transacción.

Privacidad

El modelo bancario tradicional alcanza cierto nivel de privacidad al limitarle el acceso a la información a las partes involucradas y al tercero de confianza. La necesidad de anunciar públicamente todas las transacciones excluye este método, pero la privacidad aún se puede mantener interrumpiendo el flujo de información en otro lugar: manteniendo las llaves públicas en el anonimato. El público puede ver que alguien está enviando un cierto monto a otra persona, pero sin ninguna información que vincule la transacción con nadie. Esto es similar al nivel de información publicado por las bolsas de valores, donde el tiempo y el tamaño de las transacciones individuales, es decir la "cinta", se anuncia al público, pero sin decir quiénes fueron las partes.



Como “firewall” adicional, se debería usar un nuevo par de llaves para cada transacción para evitar que ninguna de estas transacciones sea vinculada a un ningún titular en común. Algunas vinculaciones serán aún inevitables con transacciones de entradas múltiples, que necesariamente revelarán que dichas entradas fueron propiedad del mismo titular. El riesgo es que si se revela quien es el propietario de una llave, tal vinculación podría revelar otras transacciones que pertenecieron al mismo titular.

Cálculos

Consideraremos el escenario de un atacante que intenta generar una cadena alternativa más rápido que la cadena honesta. Incluso si esto se lograse, el sistema no abre a cambios arbitrarios, tales como crear valor de la nada o tomar dinero que nunca perteneció al atacante. Los nodos van a aceptar ninguna transacción inválida como forma de pago, y los nodos honestos nunca aceptarán un bloque que las contenga. Un atacante solo puede intentar cambiar una de sus propias transacciones para recuperar el dinero que gastó recientemente.

La carrera entre la cadena honesta y la cadena de un atacante puede modelarse como una Caminata Aleatoria Binomial. El evento exitoso es que la cadena honesta se extienda un bloque, lo que aumenta su ventaja en +1, y el evento de falla es que la cadena del atacante se extiende en un bloque, reduciendo la brecha en -1.

La probabilidad de que un atacante se recupere de un déficit dado es análoga al problema de la Ruina de un Jugador. Supongamos que un jugador con crédito ilimitado comienza con un déficit y juega potencialmente un número infinito de pruebas para tratar de alcanzar el equilibrio. Podemos calcular la probabilidad de que él en algún momento llegue al punto de equilibrio, o de que un atacante, en algún momento, se ponga al la par con la cadena honesta, de la siguiente manera [8]:

p = probabilidad de que un nodo honesto encuentre el siguiente bloque

q = probabilidad de que el atacante encuentre el siguiente bloque

q_z = probabilidad de que el atacante se ponga a la delantera desde una posición z bloques detrás

$$q_z = \begin{cases} 1 & if\ p \le q \\ (q/p)^z & if\ p > q \end{cases}$$

Dado nuestro supuesto de que p > q, la probabilidad cae exponencialmente a medida que aumenta el número de bloques que el atacante tiene que sobrepasar. Con las probabilidades en su contra, si no logra una embestida afortunada desde el principio, sus posibilidades se vuelven muy pequeñas a medida que se va quedando atrás.

Ahora consideramos cuánto tiempo debe esperar el destinatario de una nueva transacción antes de estar suficientemente seguros de que el remitente no puede cambiar la transacción. Asumimos que el remitente es un atacante que quiere hacer creer al destinatario que le pagó por un tiempo, para luego hacer un cambio que le pague a sí mismo el monto del pago, después de que haya pasado ese tiempo. El receptor recibirá una alerta cuando eso suceda, pero el remitente espera que ya sea demasiado tarde.

El receptor genera un nuevo par de llaves y entrega la llave pública al remitente poco antes de firmar. Esto evita que el remitente prepare una cadena de bloques con anticipación trabajando continuamente en ella hasta que tenga la suerte de adelantarse lo suficiente y luego ejecute la transacción en ese momento. Una vez que se envía la transacción, el remitente deshonesto comienza a trabajar en secreto en una cadena paralela que contiene una versión alternativa de su transacción.

El destinatario espera hasta que la transacción se haya agregado a un bloque y al menos unos “z” bloques se hayan vinculado después. No sabe la cantidad exacta de progreso que el atacante ha logrado, pero suponiendo que los bloques honestos tomaron el tiempo promedio esperado por bloque, el progreso potencial del atacante será una distribución de Poisson con el valor esperado:

$$\lambda = z\frac{q}{p}$$

Para obtener la probabilidad de que el atacante aún pueda ponerse a la par ahora, multiplicamos la densidad de Poisson por cada cantidad de progreso que pudo haber hecho por la probabilidad de que pudiera ponerse a la par desde ese punto:

$$\sum_{k=0}^{\infty} \frac{\lambda^k e^{-\lambda}}{k!} \cdot \begin{cases} (q/p)^{(z-k)} & if\ k \le z \\ 1 & if\ k > z \end{cases}$$

Reorganizando para evitar sumar la cola infinita de la distribución ...

$$1 - \sum_{k=0}^{z} \frac{\lambda^k e^{-\lambda}}{k!} \left(1 - (q/p)^{(z-k)}\right)$$

Convirtiendo a código C ...

```
#include <math.h>
double ProbabilidadExitodelAtacante(double q, int z)
{
    double p = 1.0 - q;
    double lambda = z * (q / p);
    double sum = 1.0;
    int i, k;
    for (k = 0; k <= z; k++)
    {
        double poisson = exp(-lambda);
        for (i = 1; i <= k; i++)
            poisson *= lambda / i;
        sum -= poisson * (1 - pow(q / p, z - k));
    }
    return sum;
}
```

Al ejecutar algunos resultados, podemos ver que la probabilidad disminuye exponencialmente con z.

```
q = 0.1
```

```
z = 0 P = 1.0000000
z = 1 P = 0,2045873
z = 2 P = 0.0509779
z = 3 P = 0.0131722
z = 4 P = 0.0034552
z = 5 P = 0.0009137
z = 6 P = 0.0002428
z = 7 P = 0.0000647
z = 8 P = 0.0000173
z = 9 P = 0.0000046
z = 10 P = 0.0000012
```

```
q = 0.3
z = 0 P = 1.0000000
z = 5 P = 0.1773523
z = 10 P = 0.0416605
z = 15 P = 0.0101008
z = 20 P = 0.0024804
z = 25 P = 0.0006132
z = 30 P = 0.0001522
z = 35 P = 0.0000379
z = 40 P = 0.0000095
z=45 P=0.0000024
z=50 P=0.0000006
```

Resolviendo para una P menor al 0.1% ...

```
P < 0.001
q = 0.10 z = 5
q = 0.15 z = 8
q = 0.20 z = 11
q = 0.25 z = 15
q = 0.30 z = 24
q = 0.35 z = 41
q = 0.40 z = 89
q = 0.45 z = 340
```

Conclusión

Hemos propuesto un sistema para transacciones electrónicas sin depender de la confianza. Comenzamos con el marco habitual de monedas hechas con firmas digitales, que proporciona un fuerte control de la propiedad, pero está incompleto sin una forma de evitar el gasto duplicado. Para resolver esto, propusimos una red de pares iguales utilizando la Prueba-de-Trabajo para registrar un historial público de transacciones que rápidamente se vuelve poco factible de manipular para un atacante, desde el punto de vista computacional, si los nodos honestos controlan la mayoría de la potencia de cómputo. La red es robusta en su sencillez, carente de formalidad estructural. Los nodos funcionan todos a la vez con poca coordinación. No es necesario que se identifiquen, ya que los mensajes no se enrutan a ningún lugar en particular y solo se deben entregar con base a un esfuerzo optimizado. Los nodos pueden salir y volver a unirse a la red a voluntad, aceptando la cadena de con la mejor Prueba-de-Trabajo como evidencia de lo que sucedió mientras estaban fuera. Los nodos votan con su potencia de cómputo, expresando su aceptación de los bloques válidos al trabajar en extenderlos y rechazando los bloques inválidos al negarse a trabajar en ellos. Las reglas e incentivos necesarios se pueden hacer cumplir con este mecanismo de consenso.

Referencias

[1] W. Dai, "b-money," *http://www.weidai.com/bmoney.txt*, 1998.

[2] H. Massias, X.S. Avila, and J.-J. Quisquater, "Design of a secure timestamping service with minimal trust requirements," en el 20vo Simposio sobre teoría de la información en el Benelux, mayo de 1999.

[3] S. Haber, W.S. Stornetta, "How to time-stamp a digital document," en Journal of Cryptology, vol 3, no 2, páginas 99-111, 1991.

[4] D. Bayer, S. Haber, W.S. Stornetta, "Improving the efficiency and reliability of digital time-stamping," En "Sequences II: Methods in Communication, Security and Computer Science", páginas 329-334, 1993.

[5] S. Haber, W.S. Stornetta, "Secure names for bit-strings," En: "Proceedings of the 4th ACM Conference on Computer and Communications Security", páginas 28-35, Abril de 1997.

[6] A. Back, "Hashcash - a denial of service counter-measure," *http://www.hashcash.org/papers/hashcash.pdf*, 2002.

[7] R.C. Merkle, "Protocols for public key cryptosystems," En "Proc. 1980"; Simposio de 1980 sobre seguridad y privacidad, IEEE Computer Society, páginas 122-133, abril de 1980.

[8] W. Feller, "An introduction to probability theory and its applications," 1957.

Licencia

Appendix B: Operadores, Constantes y Símbolos del Lenguaje de Script de Transacción

NOTE La fuente de las tablas y descripciones se encuentra en https://en.bitcoin.it/wiki/Script.

Empujar valor a la pila muestra los operadores que "empujan" o introducen valores a la pila.

Table 28. Empujar valor a la pila

Símbolo	Valor (hexadecimal)	Descripción
OP_0 o OP_FALSE	0x00	Un array vacío es empujado a la pila
del 1 al 75	0x01 hasta 0x4b	El comando "empuja" los próximos N bytes a la pila, donde N es de 1 a 75 bytes
OP_PUSHDATA1	0x4c	El siguiente byte del script debe contener un número N; por ende empújense los siguientes N bytes a la pila
OP_PUSHDATA2	0x4d	Los siguientes 2 bytes del script contienen al número N; por ende empújense los siguientes N bytes a la pila
OP_PUSHDATA4	0x4e	Los siguientes 4 bytes del script contienen al número N; por ende empújense los siguientes N bytes a la pila
OP_1NEGATE	0x4f	Empújese el valor "-1" a la pila
OP_RESERVED	0x50	Halt—Transacción inválida a no ser que el comando se encuentre dentro de una clausula OP_IF por ser ejecutada
OP_1 u OP_TRUE	0x51	Empujar el valor "1" a la pila
desde el OP_2 al OP_16	de 0x52 al 0x60	Para el comando OP_N, empújese el valor "N" hacia la pila, por ejemplo, OP_2 empuja el valor "2"

Control de flujo condicional muestra operadores de control de flujo condicional.

Table 29. Control de flujo condicional

Símbolo	Valor (hexadecimal)	Descripción
OP_NOP	0x61	No hacer nada
OP_VER	0x62	Halt—Transacción inválida a no ser que el comando se encuentre en una clausula OP_IF por ser ejecutada
OP_IF	0x63	Ejecutar las siguientes órdenes si el tope de la pila no es 0
OP_NOTIF	0x64	Ejecutar las siguientes órdenes si el tope de la pila es 0
OP_VERIF	0x65	Halt—Transacción inválida

OP_VERNOTIF	0x66	Halt—Transacción inválida
OP_ELSE	0x67	Ejecutar solo si las instrucciones previas no fueron ejecutadas
OP_ENDIF	0x68	Fin del bloque OP_IF, OP_NOTIF u OP_ELSE
OP_VERIFY	0x69	Comprueba la parte superior de la pila, detener y anular la transacción si no es VERDADERO
OP_RETURN	0x6A	Detener e invalidar la transacción

Operaciones de bloqueos temporales muestra operadores utilizados para bloqueos temporales.

Table 30. Operaciones de bloqueos temporales

Símbolo	Valor (hexadecimal)	Descripción
OP_CHECKLOCKTIMEVERIFY (previamente OP_NOP2)	0xb1	Marca una transacción como inválida si el valor en el tope de la pila es mayor que el valor del campo "nLockTime" de la transacción, de lo contrario, la interpretación del script continúa la ejecución tal y como si un comando OP_NOP se hubiese ejecutado. La transacción también será invalida si: 1. la pila está vacía; o 2. el valor del tope de la pila es negativo; o 3. el valor del tope de la pila es mayor o igual que 500000000 mientras que el valor del campo "nLockTime" de la transacción es menor que 500000000, o vice versa; o 4. el valor del campo de entrada "nSequence" es igual a 0xffffffff. La semántica precisa del comando se encuentra especificada en la BIP-65
OP_CHECKSEQUENCEVERIFY (previamente OP_NOP3)	0xb2	Marca la transacción como inválida si el bloqueo temporal relativo de la entrada (establecido por la BIP 0068 mediante el parámetro "nSequence") no es mas largo o igual que el valor del ítem ubicado al tope de la pila. La semántica precisa del comando se encuentra especificada en la BIP-112

Operaciones en la pila muestra operadores utilizados para manipular la pila.

Table 31. Operaciones en la pila

Símbolo	Valor (hexadecimal)	Descripción
OP_TOALTSTACK	0x6b	Saca al elemento que está en el tope de la pila principal y lo coloca al tope de una pila alternativa
OP_FROMALTSTACK	0x6c	Saca al elemento que está en el tope de una pila alternativa y lo coloca en el tope de la pila principal
OP_2DROP	0x6d	Elimina los dos elementos más altos de la pila

OP_2DUP	0x6e	Duplica los dos elementos más altos de la pila
OP_3DUP	0x6F	Duplica los tres elementos más altos de la pila
OP_2OVER	0x70	Copia a los elementos tercero y cuarto de la pila y los coloca en el tope de la misma, en el mismo orden
OP_2ROT	0x71	Mueve al quinto y sexto elementos de la pila y los coloca en la parte superior de la misma, en el mismo orden
OP_2SWAP	0x72	Intercambia de posición a los dos pares de elementos ubicados en el tope de la pila, es decir, toma al tercer y cuarto elementos de la pila y los mueve al tope en ese mismo orden
OP_IFDUP	0x73	Si el elemento ubicado en el tope de la pila no es cero (0), entonces se duplica
OP_DEPTH	0x74	Cuenta los elementos que hay la pila y empuja a la pila el número resultante
OP_DROP	0x75	Elimina el elemento que se ubica al tope de la pila
OP_DUP	0x76	Duplica el elemento que se ubica al tope de la pila
OP_NIP	0x77	Elimina el segundo elemento más alto de la pila
OP_OVER	0x78	Copia al segundo elemento más alto de la pila y lo coloca en el tope
OP_PICK	0x79	Toma el elemento ubicado al tope de la pila y lo interpreta como un número "N", y a continuación, copia el elemento enésimo desde el tope de la pila hacia abajo y lo coloca al tope de la pila
OP_ROLL	0x7a	Toma el elemento ubicado al tope de la pila y lo interpreta como un número "N", y a continuación, mueve el elemento enésimo desde el tope de la pila hacia abajo y lo coloca al tope de la pila
OP_ROT	0x7b	Toma al tercer elemento más alto de la pila y lo mueve al tope de la misma
OP_SWAP	0x7c	Intercambia de posición a los dos elementos más altos de la pila
OP_TUCK	0x7d	Lleva a cabo el comando OP_SWAP con los dos elementos más altos de la pila y luego copia el elemento que ha quedado en segundo lugar y lo coloca en el tope de la pila

Operaciones de unión de cadenas muestra operadores para cadenas.

Table 32. Operaciones de unión de cadenas

Símbolo	Valor (hexadecimal)	Descripción
OP_CAT	0x7e	Desactivada (concatena los dos elementos más altos de la pila)
OP_SUBSTR	0x7f	Desactivada (devuelve una sección de una cadena)
OP_LEFT	0x80	Desactivada (devuelve la cadena de elementos que yacen más a la izquierda)
OP_RIGHT	0x81	Desactivada (devuelve la cadena de elementos que yacen más a la derecha)
OP_SIZE	0x82	Calcula la longitud de cadena del elemento superior de la pila y devuelve el resultado en el tope de la misma

Aritmética binaria y condicional muestra operadores de aritmética binaria y de lógica booleana.

Table 33. Aritmética binaria y condicional

Símbolo	Valor (hexadecimal)	Descripción
OP_INVERT	0x83	Deshabilitado (Invierte todos los bits del elemento en el tope de la pila)
OP_AND	0x84	Deshabilitado (ejecuta un AND booleano con los dos elementos más altos de la pila)
OP_OR	0x85	Deshabilitado (ejecuta un OR booleano con los dos elementos más altos de la pila)
OP_XOR	0x86	Deshabilitado (ejecuta un XOR booleano con los dos elementos más altos de la pila)
OP_EQUAL	0x87	Empuja VERDADERO (1) al tope de la pila si los dos primeros elementos son exactamente iguales, y empuja FALSO (0) en caso contrario
OP_EQUALVERIFY	0x88	Hace lo mismo que OP_EQUAL, pero a continuación, ejecuta OP_VERIFY para detener la ejecución de la máquina virtual si no se obtuvo VERDADERO
OP_RESERVED1	0x89	Halt—Invalida la transacción a no ser que el comando se ejecute dentro de una clausula OP_IF por ser ejecutada
OP_RESERVED2	0x8a	Halt—Invalida la transacción a no ser que el comando se ejecute dentro de una clausula OP_IF por ser ejecutada

Operadores numéricos muestra operadores numéricos (aritméticos).

Table 34. Operadores numéricos

Símbolo	Valor (hexadecimal)	Descripción

OP_1ADD	0x8b	Sumar 1 al ítem superior de la pila
OP_1SUB	0x8c	Restar 1 al ítem superior de la pila
OP_2MUL	0x8d	Deshabilitado (multiplicar al ítem en el tope de la pila por el número 2)
OP_2DIV	0x8e	Deshabilitado (dividir al ítem en el tope de la pila entre el número 2)
OP_NEGATE	0x8F	Invierte el signo aritmético del elemento en el tope de la pila
OP_ABS	0x90	Cambiar el signo del elemento en el tope de la pila, cualquiera que sea, a positivo
OP_NOT	0x91	Niega lógicamente al elemento en el tope de la pila. Es decir, si el valor del elemento en el tope es 0 o 1 (posee un valor booleano), lo cambia a 1 o 0, de lo contrario devuelve 0
OP_0NOTEQUAL	0x92	Si el elemento del tope es 0 devuelve 0, en caso contrario devuelve 1
OP_ADD	0x93	Toma los dos elementos más altos de la pila, los suma, y coloca el resultado en el tope
OP_SUB	0x94	Se toman los dos elementos más altos de la pila, se le resta al valor del segundo elemento, el valor del primero, y se devuelve el resultado en el tope de la pila
OP_MUL	0x95	Deshabilitado (multiplica los dos elementos más altos de la pila)
OP_DIV	0x96	Deshabilitado (divide el segundo elemento más alto de la pila entre el primero)
OP_MOD	0x97	Deshabilitado (devuelve el residuo que resulta de la división del segundo elemento más alto de la pila entre el primero)
OP_LSHIFT	0x98	Deshabilitado (Toma e interpreta al elemento más alto de la pila como un número "N". Luego toma los N bits más a la izquierda del segundo elemento más alto de la pila y se los invierte, conservando su signo)
OP_RSHIFT	0x99	Deshabilitado (Toma e interpreta al elemento más alto de la pila como un número "N". Luego toma los N bits más a la derecha del segundo elemento más alto de la pila y se los invierte, conservando su signo)
OP_BOOLAND	0x9a	Ejecuta la operación booleana AND con los dos elementos más altos de la pila

OP_BOOLOR	0x9b	Ejecuta la operación booleana OR con los dos elementos más altos de la pila
OP_NUMEQUAL	0x9c	Devuelve VERDADERO si los dos elementos superiores de la pila son dos números iguales
OP_NUMEQUALVERIFY	0x9d	Hace lo mismo que NUMEQUAL, y después ejecuta OP_VERIFY para detener la ejecución si no es VERDADERO
OP_NUMNOTEQUAL	0x9e	Devuelve VERDADERO si los dos elementos superiores de la pila no son números iguales
OP_LESSTHAN	0x9f	Devuelve VERDADERO si el valor del segundo elemento es menor que el del elemento superior
OP_GREATERTHAN	0xa0	Devuelve VERDADERO si el valor del segundo elemento es mayor que el del elemento superior
OP_LESSTHANOREQUAL	0xa1	Devuelve VERDADERO si el segundo elemento es menor o igual que el elemento superior
OP_GREATERTHANOREQUAL	0xa2	Devuelve VERDADERO si el segundo elemento es mayor o igual que el elemento superior
OP_MIN	0xa3	Devuelve el menor de los dos elementos superiores
OP_MAX	0xa4	Devuelve el mayor de los dos elementos superiores
OP_WITHIN	0xa5	Devuelve VERDADERO si el tercer elemento está entre el segundo elemento (o igual) y el primer elemento

Operaciones criptográficas y de cálculos de hash muestra operadores de funciones criptográficas.

Table 35. Operaciones criptográficas y de cálculos de hash

Símbolo	Valor (hexadecimal)	Descripción
OP_RIPEMD160	0xa6	Devuelve el hash RIPEMD160 del elemento superior
OP_SHA1	0xa7	Devuelve el hash SHA1 del elemento superior
OP_SHA256	0xa8	Devuelve el hash SHA256 del elemento superior
OP_HASH160	0xa9	Devuelve el hash RIPEMD160(SHA256(x)) del elemento superior x
OP_HASH256	0xaa	Devuelve el hash SHA256(SHA256(x)) del elemento superior x

OP_CODESEPARATOR	0xab	Marca el comienzo de los datos a ser firmados
OP_CHECKSIG	0xac	Extrae del tope de la pila una llave pública y una firma y verifica la firma con el hash de los datos de la transacción; devuelve VERDADERO si el resultado es congruente
OP_CHECKSIGVERIFY	0xad	Hace lo mismo que CHECKSIG, pero después ejecuta OP_VERIFY para detener la ejecución si no se obtiene VERDADERO
OP_CHECKMULTISIG	0xae	Ejecuta CHECKSIG por cada par de datos suministrados respectivamente como firma y llave pública. Todos los datos deben ser congruentes. Un error de código en la implementación de este comando elimina un valor extra, lo que puede remediarse usando como prefijo un elemento OP_0
OP_CHECKMULTISIGVERIFY	0xaf	Hace lo mismo que CHECKMULTISIG, pero después ejecuta OP_VERIFY para detener la ejecución si no se obtiene VERDADERO

No-operadores muestra símbolos no-operacionales.

Table 36. No-operadores

Símbolo	Valor (hexadecimal)	Descripción
desde OP_NOP1 al OP_NOP10	de 0xb0 a 0xb9	No hace nada, comando ignorado

Códigos OP reservados para el uso del intérprete interno muestra códigos operacionales reservados para su uso por parte del intérprete de scripts interno.

Table 37. Códigos OP reservados para el uso del intérprete interno

Símbolo	Valor (hexadecimal)	Descripción
OP_SMALLDATA	0xf9	Representa un campo pequeño de datos
OP_SMALLINTEGER	0xfa	Representa un campo de entero pequeño
OP_PUBKEYS	0xfb	Representa campos de llave pública
OP_PUBKEYHASH	0xfd	Representa un campo de hash de llave pública
OP_PUBKEY	0xfe	Representa un campo de llave pública
OP_INVALIDOPCODE	0xff	Representa cualquier código OP que no esté asignado actualmente

Appendix C: Propuestas de Mejora para Bitcoin

Las propuestas de mejora de bitcoin (o del inglés, las "BIP") son documentos de diseño para proporcionar información a la comunidad de bitcoin, o para proponer una nueva característica para bitcoin o sus procesos o entorno.

Según la BIP-01 *BIP Propósito y Directrices*, hay tres tipos de BIPs:

***Standard* BIP**

Describe cualquier cambio que afecte a la mayoría o a todas las implementaciones de Bitcoin, tales como un cambio en el protocolo de red, un cambio en las reglas de validación de bloques o transacciones, o cualquier cambio o adición que afecte a la interoperabilidad de las aplicaciones que usan bitcoin.

***Informational* BIP**

Describe un problema de diseño de bitcoin, o proporciona directrices generales o información a la comunidad bitcoin, pero no propone una nueva característica. Los Informational_ BIP no representan necesariamente un consenso o recomendación de la comunidad bitcoin, por lo que los usuarios y los ejecutores pueden ignorarlos o seguir sus recomendaciones.

***Process* BIP**

Las BIPs de procesos, describen un proceso de bitcoin, o proponen un cambio a (o un evento en) un proceso. Las BIP de procesos son como las BIP estándar, pero aplican a otras áreas distintas al protocolo de bitcoin en sí. Podrían proponer una implementación, pero no a la base de códigos de bitcoin; a menudo requieren del consenso de la comunidad; y a diferencia de las BIP informativas, son más que recomendaciones, y los usuarios generalmente no son libres de ignorarlas. Algunos ejemplos incluyen procedimientos, pautas, cambios en el proceso de toma de decisiones y cambios en las herramientas o el entorno utilizados en el desarrollo de bitcoin. Cualquier “meta-BIP” es también considerado una BIP de procesos.

Las BIP se registran en un repositorio versionado en GitHub: https://github.com/bitcoin/bips. Captura de BIPs muestra una instantánea de las BIP en abril de 2017. Consúltese el repositorio autorizado para obtener información actualizada sobre las BIP existentes y sus contenidos.

Table 38. Captura de BIPs

Leyenda:	BIP #	Título de la Propuesta	Proponente	Tipo de Propuesta	Estatus de la Propuesta
BIP-1	Propósito y directrices de BIP	Amir Taaki	Procesos	Reemplazado	BIP-2
Proceso BIP, revisado	Luke Dashjr	Procesos	Activo	BIP-8]	Bits de versión c bloqueo garantizado
Shaolin Fry	Informativo	Borrador	BIP-9	Bits de versión con tiempo de espera y retraso	Pieter Wuille, Peter Todd, Greg Maxwell , Rusty Russell
Informativo	Final	BIP-10	Distribución de transacciones de varios signos	Alan Reiner	Informativo
Retirado	BIP-11	Transacciones estándar M-de-N	Gavin Andresen	Estándar	Final
BIP-12	OP_EVAL	Gavin Andresen	Estándar	Retirado	BIP-13
Formato de direcciones para el pago-al-hash-de-un-script (pay-	Gavin Andresen	Estándar	Final	BIP-14	Versión de Protocolo y Agen de Usuario

to-script-hash)					
Amir Taaki, Patrick Strateman	Estándar	Final	BIP-15	Los Alias	Amir Taaki
Estándar	Diferido	BIP-16	Pagar al Hash de un Script	Gavin Andresen	Estándar
Final	BIP-17	OP_CHECKHASHVERIFY (CHV)	Luke Dashjr	Estándar	Retirado
BIP-18	hashScriptCheck	Luke Dashjr	Estándar	Propuesto	BIP-19
Transacciones estándar M-de-N (Low SigOp)	Luke Dashjr	Estándar	Borrador	BIP-20	Esquema URI
Luke Dashjr	Estándar	Reemplazado	BIP-21	Esquema URI	Nils Schneider, Matt Corallo
Estándar	Final	BIP-22]	getblocktemplate - Fundamentos	Luke Dashjr	Estándar
Final	BIP-23	getblocktemplate - Minería Agrupada	Luke Dashjr	Estándar	Final
BIP-30	Transacciones duplicadas	Pieter Wuille	Estándar	Final	BIP-31
Mensaje Pong	Mike Hearn	Estándar	Final	BIP-32	Carteras Jerárquico-Deterministas
Pieter Wuille	Informativo	Final	BIP-33	Nodos Estratificados	Amir Taaki
Estándar	Borrador	BIP-34	Bloque v2, Altura en Coinbase	Gavin Andresen	Estándar
Final	BIP-35	mensaje de mempool	Jeff Garzik	Estándar	Final
BIP-36	Servicios a la Medida del Cliente	Stefan Thomas	Estándar	Borrador	BIP-37
Filtrado de Conexiones Bloom	Mike Hearn, Matt Corallo	Estándar	Final	BIP-39	Código mnemónico para generar llaves deterministas
Marek Palatinus, Pavol Rusnak, Aaron Voisine , Sean Bowe	Estándar	Propuesto	BIP-40	Protocolo de conectividad de Stratum	Marek Palatinus
Estándar	Número de BIP asignado	BIP-41	Protocolo de minería de Stratum	Marek Palatinus	Estándar
Número de BIP asignado	BIP-42	Un suministro monetario finito para Bitcoin	Pieter Wuille	Estándar	Borrador
BIP-43	Indicador de propósito para definir carteras deterministas	Marek Palatinus, Pavol Rusnak	Informativo	Borrador	BIP-44

Jerarquía de múltiples cuentas para carteras deterministas	Marek Palatinus, Pavol Rusnak	Estándar	Propuesta	BIP-45	Estructura para carteras deterministas P2SH de firmas múltiples
Manuel Araoz, Ryan X. Charles, Matias Alejo Garcia	Estándar	Propuesto	BIP-47	Códigos de pago reutilizables para carteras jerárquico deterministas	Justus Ranvier
Informativo	Borrador	BIP-49	Esquema de derivación para cuentas basadas en P2WPKH-anidadas-en-P2SH	Daniel Weigl	Informativo
Borrador	BIP-50	Cadena Bifurcada de Marzo de 2013 Post-Mortem	Gavin Andresen	Informativo	Final
BIP-60	Mensaje de "versión" de longitud fija (campo de transacciones de retransmisión)	Amir Taaki	Estándar	Borrador	BIP-61
Rechazo de mensajes P2P	Gavin Andresen	Estándar	Final	BIP-62	Manejo de la maleabilidad
Pieter Wuille	Estándar	Retirado	BIP-63	Direcciones ocultas	Peter Todd
Estándar	Número BIP asignado	BIP-64	mensaje getutxo	Mike Hearn	Estándar
Borrador	BIP-65	OP_CHECKLOCKTIMEVERIFY	Peter Todd	Estándar	Final
BIP-66	Firmas estrictamente en DER	Pieter Wuille	Estándar	Final	BIP-67
Direcciones determinísticas de Pago-al-Hash-de-un-Script (P2SH) de multi-firma mediante la clasificación de llaves públicas	Thomas Kerin, Jean-Pierre Rupp, Ruben de Vries	Estándar	Propuesta	BIP-68	Bloqueo tempora relativo utilizand números de secuencia impuestos por consenso
Mark Friedenbach, BtcDrak , Nicolas Dorier, kinoshitajona	Estándar	Final	BIP-69	Indexación lexicográfica de entradas y salidas de transacciones	Kristov Atlas
Informativo	Propuesto	BIP-70	Protocolo de pagos	Gavin Andresen, Mike Hearn	Estándar
Final	BIP-71	Protocolos de Pagos del tipo MIME	Gavin Andresen	Estándar	Final

BIP-72	bitcoin: extensiones uri para el Protocolo de Pagos	Gavin Andresen	Estándar	Final	BIP-73
Utilice el encabezado "Aceptar" para respuestas en negociaciones con las URL de Solicitudes de Pagos	Stephen Par	Estándar	Final	BIP-74	Permitir un valo cero para OP_RETURN en e Protocolo de Pag
Toby Padilla	Estándar	Borrador	BIP-75	Intercambio de Direcciones Fuera de Banda usando Encriptación del Protocolo de Pagos	Justin Newton, Matt David , Aar Voisine, James MacWhyte
Estándar	Proyecto	BIP-80	Jerarquías para Carteras Multi Firmas Deterministas en Grupos de Votación No-Coloreados	Justus Ranvier, Jimmy Song	Informativo
Diferido	BIP-81	Jerarquías para Carteras Multi Firmas Deterministas en Grupos de Votación Coloreados	Justus Ranvier, Jimmy Song	Informativo	Diferido
BIP-83	Árboles de Llaves Deterministas con Jerarquías Dinámicas	Eric Lombrozo	Estándar	Borrador	BIP-90
Implementaciones enterradas	Suhas Daftuar	Informativo	Borrador	BIP-99	Motivación y despliegue de cambios de regla de consenso (bifurcaciones [suaves / fuertes]
Jorge Timón	Informativo	Borrador	BIP-101	Aumentar el tamaño máximo de bloque	Gavin Andresen
Estándar	Retirado	BIP-102	El tamaño del bloque aumenta a 2 MB	Jeff Garzik	Estándar
Borrador	BIP-103	El tamaño del bloque en seguimiento al crecimiento tecnológico	Pieter Wuille	Estándar	Borrador
BIP-104	'Block75' - Tamaño de bloque máximo como dificultad	t.khan	Estándar	Borrador	BIP-105

Algoritmo de redefinición del tamaño de bloque basado en consenso	BtcDrak	Estándar	Borrador	BIP-106	Tamaño de la Máxima Capacidad del Bloque de Bitcoi Controlado Dinámicamente
Upal Chakraborty	Estándar	Borrador	BIP-107	Límite dinámico en el tamaño del bloque	Washington Y. Sanchez
Estándar	Borrador	BIP-109	Límite de tamaño de dos millones de bytes con límites sigop y sighash	Gavin Andresen	Estándar
Rechazado	BIP-111	Bit de servicio NODE_BLOOM	Matt Corallo, Peter Todd	Estándar	Propuesto
BIP-112	CHECKSEQUENCEVERIFY	BtcDrak, Mark Friedenbach, Eric Lombrozo	Estándar	Final	BIP-113
Mediana de tiempo pasado como punto final para los cálculos de bloqueos temporales	Thomas Kerin, Mark Friedenbach	Estándar	Final	BIP-114	Árbol de Sintaxis Abstracto Merkelizado
Johnson Lau	Estándar	Borrador	BIP-120	Prueba de Pago	Kalle Rosenbaum
Estándar	Borrador	BIP-121	Esquema URI de Prueba de Pago	Kalle Rosenbaum	Estándar
Borrador	BIP-122	Esquema URI para referencias / exploración de la cadena de bloques	Marco Pontello	Estándar	Borrador
BIP-123	Clasificación de BIP's	Eric Lombrozo	Proceso	Activo	BIP-124
Plantillas de Scripts Jerárquico Deterministas	Eric Lombrozo, William Swanson	Informativo	Borrador	BIP-125	Señalización de Opción de Ingres para el Reemplaz Integro por Comisiones
David A. Harding , Peter Todd	Estándar	Propuesto	BIP-126	Mejores Prácticas para Transacciones con Scripts de Entrada Heterogéneos	Kristov Atlas
Informativo	Borrador	BIP-130	mensaje de envío de cabeceras "sendheaders"	Suhas Daftuar	Estándar
Propuesto	BIP-131	Especificación para "Transacción de Coalición" (entradas tipo tarjeta libre o	Chris Priest	Estándar	Borrador

		comodín)			
BIP-132	Proceso de aceptación de una BIP basada en comités	Andy Chase	Proceso	Retirado	BIP-133
mensaje del filtro de tarifas	Alex Morcos	Estándar	Borrador	BIP-134	Transacciones flexibles
Tom Zander	Estándar	Borrador	BIP-140	TXID Normalizado	Christian Decker
Estándar	Borrador	BIP-141	Testigo Segregado (Capa de consenso)	Eric Lombrozo, Johnson Lau, Pieter Wuille	Estándar
Borrador	BIP-142	Formato de Direcciones para el Testigo Segregado	Johnson Lau	Estándar	Diferido
BIP-143	Verificación de Firmas de Transacciones para el Programa de Testigos de la Versión 0	Johnson Lau, Pieter Wuille	Estándar	Borrador	BIP-144
Testigo Segregado (Servicios de Pares)	Eric Lombrozo, Pieter Wuille	Estándar	Borrador	BIP-145	Actualizaciones d “getblocktemplat para Testigos Segregados
Luke Dashjr	Estándar	Borrador	BIP-146	Manejo de la maleabilidad de la codificación de la firma	Johnson Lau, Pieter Wuille
Estándar	Borrador	BIP-147	Manejo de la maleabilidad ante elementos de pila superfluos	Johnson Lau	Estándar
Borrador	BIP-148	Activación obligatoria del despliegue “segwit”	Shaolin Fry	Estándar	Borrador
BIP-150	Autenticación entre Pares	Jonas Schnelli	Estándar	Borrador	BIP-151
Cifrado de Comunicación Entre Pares	Jonas Schnelli	Estándar	Borrador	BIP-152	Retransmisión d Bloque Compact
Matt Corallo	Estándar	Borrador	BIP-171	Aplicación para monitoreo del tipo de cambio entre monedas	Luke Dashjr
Estándar	Borrador	BIP-180	Tamaño/Peso del Bloque: prueba de fraude	Luke Dashjr	Estándar

Borrador	BIP-199	Transacciones Contractuales por Bloqueo Temporal y Hash	Sean Bowe, Daira Hopwood	Estándar	Borrador

Appendix D: Bitcore

Bitcore es un conjunto de herramientas proporcionadas por BitPay. Su objetivo es proporcionar herramientas fáciles de usar para los desarrolladores de Bitcoin. Casi todo el código de Bitcore está escrito en JavaScript. Hay algunos módulos escritos específicamente para NodeJS. Finalmente, el módulo "node" de Bitcore incluye el código C++ de Bitcoin Core. Por favor vea https://bitcore.io para más información.

Lista de características de Bitcore

- Nodo completo de Bitcoin (bitcore-nodo)
- Explorador de bloques (insight)
- Funcionalidades de cartera, para bloques y para transacciones (bitcore-lib)
- Comunicación directa con la red de pares P2P de Bitcoin (bitcore-p2p)
- Generación entrópica de semilla mnemonica (bitcore-mnemonic)
- Protocolo de pago (bitcore-payment-protocol)
- Verificación y firma de mensajes (bitcore-message)
- Esquema Integrado de Cifrado por curva elíptica (bitcore-ecies)
- Servicio de Cartera (bitcore-wallet-service)
- Aplicación Cliente de Cartera (bitcore-wallet-client)
- Integración directa de servicios con la aplicación principal Bitcoin Core (bitcore-node)

Ejemplos de Bibliotecas Bitcore

Prerequisitos

- NodeJS >= 4.x

Si ha de usarse NodeJS y el nodo REPL:

```
$ npm install -g bitcore-lib bitcore-p2p
```

Ejemplos de billetera usando bitcore-lib

Creando una nueva dirección bitcoin con llave privada asociada:

```
> bitcore = require('bitcore-lib')
> privateKey = new bitcore.PrivateKey()
> address = privateKey.toAddress().toString()
```

Creando una llave privada jerárquico determinista y su dirección:

```
> hdPrivateKey = bitcore.HDPrivateKey()
> hdPublicKey = bitcore.HDPublicKey(hdPrivateKey)
> hdAddress = new bitcore.Address(hdPublicKey.publicKey).toString()
```

Creando y firmando una transacción desde una UTXO:

```
> utxo = {
  txId: la id de una transacción contentiva de una salida sin gastar,
  outputIndex: el índice de la referida salida, por ejemplo 0,
  address: addressOfUtxo,
  script: bitcore.Script.buildPublicKeyHashOut(addressOfUtxo).toString(),
  satoshis: cantidad enviada a la address ó dirección
}
> fee = 3000 //calculado adecuadamente según las condiciones de la red
> tx = new bitcore.Transaction()
        .from(utxo)
        .to(address, 35000)
        .fee(fee)
        .enableRBF()
```

```
        .sign(privateKeyOfUtxo)
```

Reemplazo de la última transacción en el tanque de memoria (replace-by-fee):

```
> rbfTx = new Transaction()
           .from(utxo)
           .to(address, 35000)
           .fee(fee*2)
           .enableRBF()
           .sign(privateKeyOfUtxo);
> tx.serialize();
> rbfTx.serialize();
```

Transmisión de una transacción a la red Bitcoin (observación: únicamente difúndanse transacciones válidas; hacer referencia a https://bitnodes.21.co/nodes para un listado de pares o "peers"):

1. Cópiese el siguiente código en un archivo llamado *broadcast.js*.
2. Las variables tx y rbfTx son las salidas de tx.serialize() y rbfTx.serialize(), respectivamente.
3. Con el fin de realizar un reemplazo-por-comisiones, el nodo peer debe reconocer la opción bitcoind: mempoolreplace y tenerla ajustada con el valor 1.
4. Ejecútese mediante comando node, el archivo *broadcast.js*:

```
var p2p = require('bitcore-p2p');
var bitcore = require('bitcore-lib');
var tx = new bitcore.Transaction('la salida que se obtuvo de la función de serialización');
var rbfTx = new bitcore.Transaction('la salida que se obtuvo de la función de serialización');
var host = 'dirección ip'; //úsese un nodo "peer" válido y activo en el puerto tcp 8333
var peer = new p2p.Peer({host: host});
var messages = new p2p.Messages();
peer.on('ready', function() {
  var txs = [messages.Transaction(tx), messages.Transaction(rbfTx)];
  var index = 0;
  var interval = setInterval(function() {
    peer.sendMessage(txs[index++]);
    console.log('tx: ' + index + ' sent');
    if (index === txs.length) {
      clearInterval(interval);
      console.log('disconnecting from peer: ' + host);
      peer.disconnect();
    }
  }, 2000);
});
peer.connect();
```

Appendix E: pycoin, ku, y tx

La siguiente biblioteca de Python, library pycoin, que fue originalmente escrita y mantenida por Richard Kiss, es una biblioteca basada en Python que da soporte a la manipulación tanto de llaves como de transacciones de bitcoin, e incluso, da soporta al lenguaje script con el suficiente nivel como para permitir el manejo de transacciones fuera del estándar.

La biblioteca pycoin puede ejecutarse tanto con Python 2 (versión 2.7.x) como con Python 3 (versión 3.3 o posteriores) y viene con algunas herramientas prácticas para la línea de comandos, ku y tx.

Utilidad para llaves (KU)

La utilidad de la línea de comandos ku (utilidad para llaves o del inglés "key utility") es una "Navaja Suiza" para la manipulación de llaves. Da soporta a llaves de acuerdo con la BIP-32, formato WIF, y direcciones diversas (tanto para bitcoin como para monedas alternativas). Los siguientes son algunos ejemplos.

Creación de una llave según la BIP-32 utilizando las fuentes de entropía por defecto de GPG y */dev/random*:

```
$ ku create

input           : create
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJBsY7VqXUG7hipgdWaU
                    m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version  : xpub661MyMwAqRbcFpYYiuvZpKjKhnJDZYAkWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
                    DGcpfT56AMFeo8M8KPkFMfLUtvwjwb6WPv8rY65L2q8Hz
tree depth      : 0
fingerprint     : 9d9c6092
parent f'print  : 00000000
child index     : 0
chain code      : 80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key     : yes
secret exponent : 112471538590155650688604752840386134637231974546906847202389294096567806844862
 hex            : f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif             : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
 uncompressed   : 5KhoEavGNNH4GHKoy2Ptu4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x   : 764606382405464783648433974782784681018771177678734621270215603682901140160 34
public pair y   : 598078796574697741020401202982722077309212917366332477370774067536768257777 01
 x as hex       : a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 y as hex       : 843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
y parity        : odd
key pair as sec : 03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 uncompressed   : 04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
                    843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160         : 9d9c609247174ae323acfc96c852753fe3c8819d
 uncompressed   : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
 uncompressed   : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM
```

Creación de una llave acorde con la BIP-32 a partir de una frase de contraseña:

WARNING La contrasena en este ejemplo es muy facil de adivinar.

```
$ ku P:foo

input           : P:foo
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K31AgNK5pyVvW23gHnkBq2wh5aEk6g1s496M8ZMjxncCKZKgb5j
                    ZoY5eSJMJ2Vbyvi2hbmQnCuHBujZ2WXGTux1X2k9Krdtq
public version  : xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtS
                    VYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
tree depth      : 0
fingerprint     : 5d353a2e
parent f'print  : 00000000
child index     : 0
chain code      : 5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc
private key     : yes
secret exponent : 65825730547097305716057160437970790220123864299761908948746835886007793998275
 hex            : 91880b0e3017ba586b735fe7d04f1790f3c46b818a2151fb2def5f14dd2fd9c3
wif             : L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
 uncompressed   : 5JvNzA5vXDoKYJdw8SwwLHxUxaWvn9mDea6k1vRPCX7KLUVWa7W
public pair x   : 818219827193811040617773492691304190244936166509935893945534043477743931681 91
public pair y   : 589942180696054242783207032506897801547850995092776917231263250512004590382 90
```

```
 x as hex       : b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 y as hex       : 826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
y parity        : even
key pair as sec : 02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
 uncompressed   : 04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f
                    826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52
hash160         : 5d353a2ecdb262477172852d57a3f11de0c19286
 uncompressed   : e5bd3a7e6cb62b4c820e51200fb1c148d79e67da
Bitcoin address : 19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
 uncompressed   : 1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT
```

Recuperar la información en formato JSON:

```
$ ku P:foo -P -j
```


```
{
   "y_parity": "even",
   "public_pair_y_hex": "826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",
   "private_key": "no",
   "parent_fingerprint": "00000000",
   "tree_depth": "0",
   "network": "Bitcoin",
   "btc_address_uncompressed": "1MwkRkogzBRMehBntgcq2aJhXCXStJTXHT",
   "key_pair_as_sec_uncompressed":
"04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2
983544de52",
   "public_pair_x_hex": "b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f",
   "wallet_key":
"xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy",
   "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
   "child_index": "0",
   "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
   "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
   "fingerprint": "5d353a2e",
   "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
   "input": "P:foo",
   "public_pair_x": "81821982719381104061777349269130419024493616650993589394553404347774393168191",
   "public_pair_y": "58994218069605424278320703250689780154785099509277691723126325051200459038290",
   "key_pair_as_sec": "02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}
```


Llave pública BIP32:

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy
```

Generar una sub-llave:

```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6iogWKxx6mefEw4M8EroLgKj
```

Sub-llave reforzada:

```
$ ku -w -s3/2H P:foo
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8iDkBN45k67UKsJUXM1JfRCdn1
```

WIF

```
$ ku -W P:foo
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

Una Dirección

```
$ ku -a P:foo
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

Generar un montón de sub-llaves

```
$ ku P:foo -s 0/0-5 -w
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3bhXxKWB89Ggn2dzLcoJsuEdRK
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVej5QHckc5Adtwxa28ACmANi9XhCrRvtFqQcUxt8rUgFz3souMiDdWxJDZnQxzx
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxan6srXUPBtj3PTxQFkZJAiwoUpmvtrxKZu4zfsnr3pqyy2vthpkwuoVq
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtFXdiEY8UsRNJfqK6DAdyZXoMvtaLHyWQx3FS4A9zw
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKdT84Mm7Wtc2xmzFuKg7iYf7XFHKkSsaYKWKJbR54bnyAD9GzjUYbAYTtN4ruo
```

Generar las direcciones correspondientes:

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkjdHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

Generar los WIFs correspondientes:

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQGxLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuvFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

Compruébese que esto funcione, eligiendo una cadena BIP32 (la correspondiente a la sub-llave 0/3):

```
$ ku -W xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
$ ku -a xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

En efecto, luce familiar.

Del exponente secreto:

```
$ ku 1

input           : 1
network         : Bitcoin
secret exponent : 1
 hex            : 1
wif             : KwDiBf89QgGbjEhKnhXJuH7LrciVrZi3qYjgd9M7rFU73sVHnoWn
 uncompressed   : 5HpHagT65TZzG1PH3CSu63k8DbpvD8s5ip4nEB3kEsreAnchuDf
public pair x   : 55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y   : 32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex       : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex       : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity        : even
key pair as sec : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed   : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                    483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed   : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
 uncompressed   : 1EHNa6Q4Jz2uvNExL497mE43ikXhwF6kZm
```

Versión Litecoin:

```
$ ku -nL 1

input           : 1
network         : Litecoin
secret exponent : 1
 hex            : 1
wif             : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdwUwyfRDeGZm76aUjV
 uncompressed   : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x   : 55066263022277343669578718895168534326250603453777594175500187360389116729240
```

```
public pair y    : 32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity         : even
key pair as sec  : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                     483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6sUkLKoqDEVUnJ
 uncompressed    : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

WIF para Dogecoin:

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot
```

De un par público (en la red de pruebas):

```
$ ku -nT 550662630222773436695787188951685343262506034537775941755001873603 89116729240,even

input                  : 550662630222773436695787188951685343262506034537775941755001873603
                           89116729240,even
network                : Bitcoin testnet
public pair x          : 55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y          : 32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex              : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex              : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity               : even
key pair as sec        : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed          : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                           483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160                : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed          : 91b24bf9f5288532960ac687abb035127b1d28a5
Bitcoin testnet address : mrCDrCybB6J1vRfbwM5hemdJz73FwDBC8r
 uncompressed          : mtoKs9V381UAhUia3d7Vb9GNak8Qvmcsme
```

Del hash160:

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6

input           : 751e76e8199196d454941c45d1b3a323f1433bd6
network         : Bitcoin
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

("", startref="pycoin library"Cómo dirección de Dogecoin:

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6

input            : 751e76e8199196d454941c45d1b3a323f1433bd6
network          : Dogecoin
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

Utilidad de Transacciones (TX)

La utilidad de la linea de comandos tx desplegará las transacciones en un formato amigable al ojo humano, rastreará transacciones base a partir de transacciones en caché de pycoin o a partir de servicios web (los sitios: blockchain.info, blockcypher.com, blockr.io y chain.so, forman actualmente parte del soporte del servicio), combinará transacciones, añadirá o eliminará entradas o salidas, y permitirá firmar transacciones.

Los siguientes son algunos ejemplos.

Encontrar a la famosa "transacción de la pizza"

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache to cache transactions fetched via web
services
warning: no service providers found for get_tx; consider setting environment variable PYCOIN_BTC_PROVIDERS
```

```
usage: tx [-h] [-t TRANSACTION_VERSION] [-l LOCK_TIME] [-n NETWORK] [-a]
          [-i address] [-f path-to-private-keys] [-g GPG_ARGUMENT]
          [--remove-tx-in tx_in_index_to_delete]
          [--remove-tx-out tx_out_index_to_delete] [-F transaction-fee] [-u]
          [-b BITCOIND_URL] [-o ruta-al-archivo-de-salida]
          argument [argument ...]
tx: error: can't find Tx with id 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
```

Huy! No tenemos servicios web configurados para ubicar la transacción buscada. Hagámoslo ahora:

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_BTC_PROVIDERS="block.io blockchain.info blockexplorer.com"
$ export PYCOIN_CACHE_DIR PYCOIN_BTC_PROVIDERS
```

Esto no se hace automáticamente así que una herramienta de la linea de comandos no dejara escapar información privada acerca de las transacciones en las que estas interesado a un sitio web de terceros. Si no te importa, tu puedes agregar estas lineas en tu *.profile*.

Intentémoslo de nuevo:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
Versión: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (válido en cualquier momento)
Input:
  0:                          (unknown) from 1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total output 10000000.00000 mBTC
incluyendo salidas no gastadas en el volcado hexadecimal ya que la transacción no esta completamente firmada
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda874931999c90f87f0
1ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff01
0010a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

** can't validate transaction as source transactions missing
```

La linea final aparece porque para validar las firmas de las transacciones, técnicamente necesitamos las transacciones fuente. Así que agregamos -a para incrementar las transacciones con información fuente:

```
$ tx -a 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: transaction fees recommendations casually calculated and estimates may be incorrect
warning: transaction fee lower than (casually calculated) expected value of 0.1 mBTC, transaction might not propagate
Versión: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes
TxIn count: 1; TxOut count: 1
Lock time: 0 (válido en cualquier momento)
Input:
  0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk de 1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0
10000000.00000 mBTC sig ok
Output:
  0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC
Total input  10000000.00000 mBTC
Total output 10000000.00000 mBTC
Total fees        0.00000 mBTC

010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda874931999c90f87f0
1ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff01
0010a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

all incoming transaction values validated
```

Ahora, echaremos un vistazo a las salidas sin gastar para una dirección específica (UTXO). En el bloque #1, se observa una transacción coinbase a favor de 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX. Utilizaremos el comando fetch_unspent para encontrar todas las monedas asociadas a esta dirección:

```
$ fetch_unspent 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX
a3a6f902a51a2cbebede144e48a88c05e608c2cce28024041a5b9874013a1e2a/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/333000
cea36d008badf5c7866894b191d3239de9582d89b6b452b596f1f1b76347f8cb/31/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/10000
065ef6b1463f552f675622a5d1fd2c08d6324b4402049f68e767a719e2049e8d/86/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/10000
a66dddd42f9f2491d3c336ce5527d45cc5c2163aaed3158f81dc054447f447a2/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
```

```
/10000
ffd901679de65d4398de90cefe68d2c3ef073c41f7e8dbec2fb5cd75fe71dfe7/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/100
d658ab87cc053b8dbcfd4aa2717fd23cc3edfe90ec75351fadd6a0f7993b461d/5/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/911
36ebe0ca3237002acb12e1474a3859bde0ac84b419ec4ae373e63363ebef731c/1/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/100000
fd87f9adebb17f4ebb1673da76ff48ad29e64b7afa02fda0f2c14e43d220fe24/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac/1
dfdf0b375a987f17056e5e919ee6eadd87dad36c09c4016d4a03cea15e5c05e3/1/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/1337
cb2679bfd0a557b2dc0d8a6116822f3fcbe281ca3f3e18d3855aa7ea378fa373/0/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/1337
d6be34ccf6edddc3cf69842dce99fe503bf632ba2c2adb0f95c63f6706ae0c52/1/76a914119b098e2e980a229e139a9ed01a469e518e6f2688ac
/2000000
    0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0
/410496b538e853519c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141781e622947211 66bf621e73a82c
bf2342c858eeac/5000000000
```

Appendix F: Comandos del "Bitcoin Explorer" (bx)

El Bitcoin Explorer (bx) es una herramienta de la linea de comandos que ofrece una variedad de instrucciones para la administración de llaves y la construcción de transacciones. Forma parte de la biblioteca de bitcoin llamada "libbitcoin".

```
Como usar los comandos: bx COMANDO [--help]

Información: Los comandos "bx" son:

address-decode
address-embed
address-encode
address-validate
base16-decode
base16-encode
base58-decode
base58-encode
base58check-decode
base58check-encode
base64-decode
base64-encode
bitcoin160
bitcoin256
btc-to-satoshi
ec-add
ec-add-secrets
ec-multiply
ec-multiply-secrets
ec-new
ec-to-address
ec-to-public
ec-to-wif
fetch-balance
fetch-header
fetch-height
fetch-history
fetch-stealth
fetch-tx
fetch-tx-index
hd-new
hd-private
hd-public
hd-to-address
hd-to-ec
hd-to-public
hd-to-wif
help
input-set
input-sign
input-validate
message-sign
message-validate
mnemonic-decode
mnemonic-encode
ripemd160
satoshi-to-btc
script-decode
script-encode
script-to-address
seed
send-tx
send-tx-node
send-tx-p2p
settings
sha160
sha256
sha512
stealth-decode
stealth-encode
stealth-public
stealth-secret
stealth-shared
tx-decode
tx-encode
uri-decode
uri-encode
validate-tx
watch-address
wif-to-ec
```

```
wif-to-public
wrap-decode
wrap-encode
```

Para mayor información, consúltese: Bitcoin Explorer homepage y Bitcoin Explorer user documentation.

Ejemplos de Uso de Comandos del "bx"

Veamos algunos ejemplos del uso de los comandos del Bitcoin Explorer para experimentar un poco con llaves y direcciones.

Vamos a generar un valor de "semilla" aleatoria, utilizando el comando seed, el cual utiliza el generador de números aleatorios de nuestro sistema operativo. Luego vamos a procesar esta semilla por el comando ec-new para generar una nueva llave privada. Finalmente guardaremos el resultado que se obtiene como data estándar, en el archivo *private_key*:

```
$ bx seed | bx ec-new > private_key
$ cat private_key
73096ed11ab9f1db6135857958ece7d73ea7c30862145bcc4bbc7649075de474
```

A continuación, generaremos la llave pública a partir de ésa llave privada utilizando el comando ec-to-public. Procesaremos el archivo *private_key* como dato de entrada estándar y guardaremos la data de salida estándar que nos arrojará el comando, en un nuevo archivo que llamaremos *public_key*:

```
$ bx ec-to-public < private_key > public_key
$ cat public_key
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500
```

Podemos re-formatear la data de public_key en la forma de una address o dirección bitcoin, utilizando el comando ec-to-address. Procesaremos la data del archivo *public_key* como dato de entrada estándar:

```
$ bx ec-to-address < public_key
17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG
```

Las llaves generadas de este modo producen una cartera "tipo-0" no-determinista. Esto significa que cada llave es generada a partir de una semilla independiente. Los comandos del Bitcoin Explorer también pueden generar llaves de manera determinista, de acuerdo con el BIP-32. En este caso, una llave "maestra" es creada a partir de una semilla y a partir de allí es extendida determinísticamente para producir un árbol de sub-llaves, resultando en una cartera determinista "tipo-2".

En primer lugar, utilizaremos los comandos seed y hd-new para generar una llave maestra, que será utilizada como la base para derivar una jerarquía de llaves:

```
$ bx seed > seed
$ cat seed
eb68ee9f3df6bd4441a9feadec179ff1

$ bx hd-new < seed > master
$ cat master
xprv9s21ZrQH143K2BEhMYpNQoUvAgiEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBqkYoWWMCRwzfuJk8RwF1SVEpAQ
```

Ahora utilizaremos el comando hd-private para generar una llave para una "cuenta" fortalecida y una secuencia de dos llaves privadas dentro de esta cuenta:

```
$ bx hd-private --hard < master > account
$ cat account
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwcgS8PYbgoR2NWHeLyvg4DhoEE68A1n

$ bx hd-private --index 0 < account
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVEsuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F3DWPkX7Tp1V1u97HKG1FJwUpU

$ bx hd-private --index 1 < account
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdjsuMVERadbgDbziCRJru9n6tzEWrASVpEdrZrFidt1RDfn4yA3
```

A continuación, utilizaremos el comando hd-public para generar la secuencia correspondiente de dos llaves públicas:

```
$ bx hd-public --index 0 < account
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz5wzaSW4FiGrseNDR4LKqTy

$ bx hd-public --index 1 < account
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQDhohEcDFTEYMvYzwRD7Juf8
```

Las llaves públicas también pueden ser derivadas de sus correspondientes llaves privadas, utilizando el comando hd-to-public:

```
$ bx hd-private --index 0 < account | bx hd-to-public
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz5wzaSW4FiGrseNDR4LKqTy

$ bx hd-private --index 1 < account | bx hd-to-public
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQDhohEcDFTEYMvYzwRD7Juf8
```

Podemos generar un número prácticamente ilimitado de llaves en una cadena determinística, todas derivadas de una misma semilla. Esta técnica es usada en muchas aplicaciones de cartera para generar llaves a las que es posible hacer copias de respaldo con un único valor de semilla. Esto es más fácil que tener que guardar copias de la cartera con todas sus llaves generadas aleatoriamente cada vez que una nueva llave es creada.

La semilla se puede codificar usando el comando mnemonic-encode:

```
$ bx hd-mnemonic < semilla > words
adore repeat vision worst especially veil inch woman cast recall dwell appreciate
```

A continuación, la semilla puede ser decodificada utilizando el comando mnemonic-decode:

```
$ bx mnemonic-decode < words
eb68ee9f3df6bd4441a9feadec179ff1
```

La codificación Mnemónica puede hacer a la semilla más fácil de almacenar e incluso de recordar.

---

1. "Bitcoin: A Peer-to-Peer Electronic Cash System," Satoshi Nakamoto (https://bitcoin.org/bitcoin.pdf).

Last updated 2020-11-26 16:42:19 -0300