Maitriser Bitcoin

2022-03-07

# Préface

# Écrire le livre Bitcoin

Je suis tombé sur bitcoin pour la première fois à la mi-2011. Ma réaction immédiate a été plus ou moins "Pfft ! Nerd money !" et je l'ai ignoré pendant encore six mois, n'en saisissant pas l'importance. C'est une réaction que j'ai vue se répéter chez bon nombre des personnes les plus intelligentes que je connaisse, ce qui me réconforte un peu. La deuxième fois que je suis tombé sur bitcoin, lors d'une discussion sur une liste de diffusion, j'ai décidé de lire le livre blanc écrit par Satoshi Nakamoto pour étudier la source faisant autorité et voir de quoi il s'agissait. Je me souviens encore du moment où j'ai fini de lire ces neuf pages, quand j'ai réalisé que le bitcoin n'était pas simplement une monnaie numérique, mais un réseau de confiance qui pouvait aussi fournir la base de bien plus que de simples devises. La prise de conscience que "ce n'est pas de l'argent, c'est un réseau de confiance décentralisé", m'a lancé dans un voyage de quatre mois pour dévorer toutes les informations sur le bitcoin que j'ai pu trouver. Je suis devenu obsédé et captivé, passant 12 heures ou plus par jour collé à un écran, lisant, écrivant, codant et apprenant autant que possible. Je suis sorti de cet état de fugue, plus de 20 livres en moins à cause du manque de repas réguliers, déterminé à me consacrer au travail sur le bitcoin.

Deux ans plus tard, après avoir créé un certain nombre de petites start-ups pour explorer divers services et produits liés au bitcoin, j'ai décidé qu'il était temps d'écrire mon premier livre. Bitcoin était le sujet qui m'avait poussé dans une frénésie de créativité et consommé mes pensées; c'était la technologie la plus excitante que j'avais rencontrée depuis Internet. Il était maintenant temps de partager ma passion pour cette technologie incroyable avec un public plus large.

# Public visé

Ce livre est principalement destiné aux codeurs. Si vous pouvez utiliser un langage de programmation, ce livre vous apprendra comment fonctionnent les monnaies cryptographiques, comment les utiliser et comment développer des logiciels qui fonctionnent avec elles. Les premiers chapitres conviennent également comme une introduction approfondie au bitcoin pour les non-codeurs, tel ceux qui veulent comprendre le fonctionnement interne du bitcoin et des crypto-monnaies.

# Les conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

*Italique*

Indique de nouveaux termes, URL, adresses e-mail, noms de fichiers et extensions de fichiers.

Largeur constante

Utilisé pour les listes de programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que des noms de variables ou de fonctions, des bases de données, des types de données, des variables d'environnement, des instructions et des mots-clés.

**Largeur constante en gras**

Affiche les commandes ou tout autre texte qui doit être saisi littéralement par l'utilisateur.

*Largeur constante en italique* : Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

Cette icône signifie un conseil ou une suggestion.

Cette icône signifie une note générale.

Cette icône indique un avertissement ou une mise en garde.

# Exemples de codes

Les exemples sont illustrés en Python, C++, et en utilisant la ligne de commande d'un système d'exploitation de type Unix tel que Linux ou macOS. Tous les extraits de code sont disponibles dans le référentiel GitHub (<https://github.com/bitcoinbook/bitcoinbook>) dans le sous-répertoire *code* du référentiel principal. Fourchez le code du livre, essayez les exemples de code ou soumettez des corrections via GitHub.

Tous les extraits de code peuvent être répliqués sur la plupart des systèmes d'exploitation avec une installation minimale de compilateurs et d'interpréteurs pour les langages correspondants. Si nécessaire, nous fournissons des instructions d'installation de base et des exemples étape par étape de la sortie de ces instructions.

Certains des extraits de code et des sorties de code ont été reformatés pour l'impression. Dans tous ces cas, les lignes ont été séparées par un caractère barre oblique inverse (\), suivi d'un caractère de saut de ligne. Lors de la transcription des exemples, supprimez ces deux caractères et rejoignez les lignes à nouveau et vous devriez voir des résultats identiques comme indiqué dans l'exemple.

Tous les extraits de code utilisent des valeurs réelles et des calculs dans la mesure du possible, afin que vous puissiez construire d'exemple en exemple et voir les mêmes résultats dans n'importe quel code que vous écrivez pour calculer les mêmes valeurs. Par exemple, les clés privées et les clés et adresses publiques correspondantes sont toutes réelles. Les exemples de transactions, de blocs et de références de chaîne de blocs ont tous été introduits dans la chaîne de blocs bitcoin réelle et font partie du grand livre public, vous pouvez donc les consulter sur n'importe quel système bitcoin.

# Adresses et transactions Bitcoin dans ce livre

Les adresses bitcoin, les transactions, les clés, les codes QR et les données de chaîne de blocs utilisées dans ce livre sont, pour la plupart, réelles. Cela signifie que vous pouvez parcourir la chaîne de blocs, regarder les transactions proposées en exemple, les récupérer avec vos propres scripts ou programmes, etc.

Cependant, notez que les clés privées utilisées pour construire les adresses sont soit imprimées dans ce livre, soit ont été "radiées". Cela signifie que si vous envoyez de l'argent à l'une de ces adresses, l'argent sera perdu pour toujours ou, dans certains cas, tous ceux qui peuvent lire ce livre peuvent le prendre en utilisant les clés privées imprimées dans celui-ci.

N'ENVOYEZ PAS D'ARGENT AUX ADRESSES DANS CE LIVRE. Votre argent sera pris par un autre lecteur, ou perdu à jamais.

# Publication par O'Reilly Media

Ce livre est dérivé sous licence du livre "Mastering Bitcoin 2nd Edition", par O'Reilly Media.

# Droit d'auteur et licence de livre

Copyright (c) 2014 Andreas M. Antonopoulos LLC

Ce travail est sous licence Creative Commons Attribution-ShareAlike 4.0 International License. Pour voir une copie de cette licence, visitez

<http://creativecommons.org/licenses/by-sa/4.0/>.

Pour fournir une attribution, veuillez créer un lien vers le référentiel du livre :

<https://github.com/bitcoinbook/bitcoinbook>

et inclure cette licence ou une licence ouverte équivalente, pour satisfaire aux exigences ShareAlike.

Cette licence conforme à la "Culture libre" a été approuvée par mon éditeur O'Reilly Media (<http://oreilly.com>), qui comprend la valeur du mouvement de source ouverte. O'Reilly Media n'est pas seulement le meilleur éditeur de livres techniques au monde, mais aussi un fervent partisan d'une culture ouverte et du partage des connaissances.

Merci O'Reilly !

# Contacter l'auteur

Vous pouvez me contacter, Andreas M. Antonopoulos, sur mon site personnel : <https://aantonop.com/>

Des informations sur *Maîtriser Bitcoin (Mastering Bitcoin en anglais)* ainsi que l'Édition Ouverte et les traductions sont disponibles sur : <https://bitcoinbook.info/>

Suivez-moi sur Twitter: <https://twitter.com/aantonop>

Retrouvez plus de 500 vidéos pédagogiques sur ma chaîne YouTube : <https://youtube.com/aantonop>

Un grand merci à tous mes mécènes qui soutiennent mon travail par des dons mensuels. Vous pouvez suivre ma page Patreon ici : <https://patreon.com/aantonop>

# Remerciements

Ce livre représente les efforts et les contributions de nombreuses personnes. Je suis reconnaissant pour toute l'aide que j'ai reçue d'amis, de collègues et même de parfaits inconnus, qui m'ont rejoint dans cet effort pour écrire le livre technique définitif sur les crypto-monnaies et le bitcoin.

Il est impossible de faire une distinction entre la technologie bitcoin et la communauté bitcoin, et ce livre est autant un produit de cette communauté qu'un livre sur la technologie. Mon travail sur ce livre a été encouragé, applaudi, soutenu et récompensé par toute la communauté bitcoin du tout début jusqu'à la toute fin. Plus que tout, ce livre m'a permis de faire partie d'une merveilleuse communauté pendant deux ans et je ne vous remercierai jamais assez de m'avoir accepté dans cette communauté. Il y a beaucoup trop de personnes à mentionner par leur nom - des personnes que j'ai rencontrées lors de conférences, d'événements, de séminaires, de rencontres, de pizzas et de petits rassemblements privés, ainsi que beaucoup qui ont communiqué avec moi par Twitter, sur reddit, sur bitcointalk. org, et sur GitHub qui ont eu un impact sur ce livre. Chaque idée, analogie, question, réponse et explication que vous trouverez dans ce livre a été à un moment donné inspirée, testée ou améliorée grâce à mes interactions avec la communauté. Merci à tous pour votre soutien; sans vous ce livre n'aurait pas vu le jour. Je suis éternellement reconnaissant.

Le voyage pour devenir auteur commence bien avant le premier livre, bien sûr. Ma première langue (et ma scolarité) était le grec, j'ai donc dû suivre un cours de rattrapage en anglais écrit lors de ma première année d'université. Je dois remercier Diana Kordas, mon professeur d'écriture d'anglais, qui m'a aidé à développer ma confiance et mes compétences cette année-là. Plus tard, en tant que professionnel, j'ai développé mes compétences en rédaction technique sur le thème des centres de données, en écrivant pour le magazine *Network World*. Je dois remercier John Dix et John Gallant, qui m'ont donné mon premier travail d'écriture en tant que chroniqueur à *Network World* et à mon éditeur Michael Cooney et mon collègue Johna Till Johnson qui ont édité mes chroniques et les ont rendues aptes à la publication. Écrire 500 mots par semaine pendant quatre ans m'a donné suffisamment d'expérience pour éventuellement envisager de devenir auteur.

Merci également à ceux qui m'ont soutenu lorsque j'ai soumis ma proposition de livre à O'Reilly, en fournissant des références et en examinant la proposition. Plus précisément, merci à John Gallant, Gregory Ness, Richard Stiennon, Joel Snyder, Adam B. Levine, Sandra Gittlen, John Dix, Johna Till Johnson, Roger Ver et Jon Matonis. Remerciements particuliers à Richard Kagan et Tymon Mattoszko, qui ont révisé les premières versions de la proposition et à Matthew Taylor, qui a révisé la proposition.

Merci à Cricket Liu, auteur du titre O'Reilly *DNS et BIND*, qui m'a présenté O'Reilly. Merci également à Michael Loukides et Allyson MacDonald de O'Reilly, qui ont travaillé pendant des mois pour aider à la réalisation de ce livre. Allyson a été particulièrement patiente lorsque les délais ont été manqués et les livrables retardés alors que la vie intervenait dans notre échéancier. Pour la deuxième édition, je remercie Timothy McGovern d'avoir guidé le processus, Kim Cofer d'avoir patiemment édité et Rebecca Panzer d'avoir illustré de nombreux nouveaux diagrammes.

Les premières ébauches des premiers chapitres ont été les plus difficiles, car le bitcoin est un sujet difficile à démêler. Chaque fois que je tirais sur un fil de la technologie bitcoin, je devais tirer sur le tout. Je me suis retrouvé coincé à plusieurs reprises et un peu découragé alors que je luttais pour rendre le sujet facile à comprendre et créer un récit autour d'un sujet technique aussi dense. Finalement, j'ai décidé de raconter l'histoire du bitcoin à travers les histoires des personnes utilisant le bitcoin et tout le livre est devenu beaucoup plus facile à écrire. Je dois remercier mon ami et mentor, Richard Kagan, qui m'a aidé à démêler l'histoire et à surmonter les moments de blocage de l'écrivain. Je remercie Pamela Morgan, qui a révisé les premières ébauches de chaque chapitre de la première et de la deuxième édition du livre, et a posé les questions difficiles pour les améliorer. Merci également aux développeurs du groupe San Francisco Bitcoin Developers Meetup ainsi qu'à Taariq Lewis et Denise Terry pour avoir aidé à tester le premier matériel. Merci également à Andrew Naugler pour la conception infographique.

Pendant le développement du livre, j'ai rendu les premières ébauches disponibles sur GitHub et j'ai invité les commentaires du public. Plus d'une centaine de commentaires, suggestions, corrections et contributions ont été soumis en réponse. Ces contributions sont explicitement reconnues, avec mes remerciements, dans <<github\_contrib> >. Surtout, mes sincères remerciements à mes éditeurs bénévoles de GitHub, Ming T. Nguyen (1ère édition) et Will Binns (2e édition), qui ont travaillé sans relâche pour organiser, gérer et résoudre les demandes d'extraction, publier des rapports et effectuer des corrections de bogues sur GitHub.

Une fois le livre rédigé, il est passé par plusieurs séries d'examens techniques. Merci à Cricket Liu et Lorne Lantz pour leur examen approfondi, leurs commentaires et leur soutien.

Plusieurs développeurs de bitcoins ont fourni des échantillons de code, des critiques, des commentaires et des encouragements. Merci à Amir Taaki et Eric Voskuil pour les extraits de code par exemple et de nombreux excellents commentaires ; Chris Kleeschulte pour sa contribution à l'annexe Bitcore ; Vitalik Buterin et Richard Kiss pour leur aide avec les mathématiques des courbes elliptiques et leurs contributions au code ; Gavin Andresen pour les corrections, les commentaires et les encouragements ; Michalis Kargakis pour les commentaires, les contributions et la rédaction de btcd ; et Robin Inge pour les soumissions d'errata améliorant la deuxième impression. Dans la deuxième édition, j'ai de nouveau reçu beaucoup d'aide de la part de nombreux développeurs de Bitcoin Core, dont Eric Lombrozo qui a démystifié le Témoin Séparé, Luke Dashjr qui a aidé à améliorer le chapitre sur les transactions, Johnson Lau qui a revu le Témoin Séparé et d'autres chapitres, et bien d'autres. Je dois remercier Joseph Poon, Tadge Dryja et Olaoluwa Osuntokun qui ont expliqué le Lightning Network, revu mes écrits et répondu aux questions lorsque je suis resté bloqué.

Je dois mon amour des mots et des livres à ma mère, Theresa, qui m'a élevé dans une maison avec des livres tapissant tous les murs. Ma mère m'a également acheté mon premier ordinateur en 1982, bien qu'elle se décrive comme une technophobe. Mon père, Menelaos, un ingénieur civil qui vient de publier son premier livre à 80 ans, est celui qui m'a enseigné la pensée logique et analytique et l'amour de la science et de l'ingénierie.

Merci à tous de m'avoir soutenu tout au long de ce parcours.

Unresolved directive in for\_translation\_mastering-bitcoin\_preface-open\_fr\_CA.asciidoc - include::github\_contrib.asciidoc[]

# Préface

# Écrire le livre sur Bitcoin

Je suis tombé sur bitcoin pour la première fois à la mi-2011. Ma réaction immédiate a été plus ou moins "Pfft ! Nerd money !" et je l'ai ignoré pendant encore six mois, n'en saisissant pas l'importance. C'est une réaction que j'ai vue se répéter chez bon nombre des personnes les plus intelligentes que je connaisse, ce qui me réconforte un peu. La deuxième fois que je suis tombé sur bitcoin, lors d'une discussion sur une liste de diffusion, j'ai décidé de lire le livre blanc écrit par Satoshi Nakamoto pour étudier la source faisant autorité et voir de quoi il s'agissait. Je me souviens encore du moment où j'ai fini de lire ces neuf pages, quand j'ai réalisé que le bitcoin n'était pas simplement une monnaie numérique, mais un réseau de confiance qui pouvait aussi fournir la base de bien plus que de simples devises. La prise de conscience que "ce n'est pas de l'argent, c'est un réseau de confiance décentralisé", m'a lancé dans un voyage de quatre mois pour dévorer toutes les informations sur le bitcoin que j'ai pu trouver. Je suis devenu obsédé et captivé, passant 12 heures ou plus par jour collé à un écran, lisant, écrivant, codant et apprenant autant que possible. Je suis sorti de cet état de fugue, plus de 20 livres en moins à cause du manque de repas réguliers, déterminé à me consacrer au travail sur le bitcoin.

Deux ans plus tard, après avoir créé un certain nombre de petites entreprises en démarrage (start-ups) pour explorer divers services et produits liés au bitcoin, j'ai décidé qu'il était temps d'écrire mon premier livre. Bitcoin était le sujet qui m'avait poussé dans une frénésie de créativité et consommé mes pensées; c'était la technologie la plus excitante que j'avais rencontrée depuis Internet. Il était maintenant temps de partager ma passion pour cette technologie incroyable avec un public plus large.

# Public visé

Ce livre est principalement destiné aux codeurs. Si vous pouvez utiliser un langage de programmation, ce livre vous apprendra comment fonctionnent les monnaies cryptographiques, comment les utiliser et comment développer des logiciels qui fonctionnent avec elles. Les premiers chapitres conviennent également comme une introduction approfondie au bitcoin pour les non-codeurs, tel ceux qui veulent comprendre le fonctionnement interne du bitcoin et des cryptomonnaies.

# Pourquoi y a-t-il des insectes sur la couverture ?

La fourmi coupeuse de feuilles est une espèce qui présente un comportement très complexe dans un super-organisme de colonie, mais chaque fourmi individuelle fonctionne selon un ensemble de règles simples dictées par l'interaction sociale et l’échange d'odeurs chimiques (phéromones). Par Wikipedia : "À côté des humains, les fourmis coupeuses de feuilles forment les sociétés animales les plus grandes et les plus complexes sur Terre." Les fourmis coupeuses de feuilles ne mangent pas réellement les feuilles, mais les utilisent plutôt pour cultiver un champignon, qui est la principale source de nourriture de la colonie. Imaginez, ces fourmis cultivent !

Bien que les fourmis forment une société basée sur les castes et aient une reine pour produire une progéniture, il n'y a pas d'autorité centrale ou de chef dans une colonie de fourmis. Le comportement hautement intelligent et sophistiqué présenté par une colonie de plusieurs millions de membres est une propriété émergente de l'interaction des individus dans un réseau social.

La nature démontre que les systèmes décentralisés peuvent être résilients et peuvent produire une complexité émergente et une sophistication incroyable sans avoir besoin d'une autorité centrale, d'une hiérarchie ou de parties complexes.

Bitcoin est un réseau de confiance décentralisé hautement sophistiqué qui peut prendre en charge une myriade de processus financiers. Pourtant, chaque nœud du réseau Bitcoin suit quelques règles mathématiques simples. L'interaction entre de nombreux nœuds est ce qui conduit à l'émergence du comportement sophistiqué, et non une complexité inhérente ou une confiance dans un seul nœud. Comme une colonie de fourmis, le réseau Bitcoin est un réseau résilient de nœuds simples suivant des règles simples qui, ensemble, peuvent faire des choses incroyables sans aucune coordination centrale.

# Conventions utilisées dans ce livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

*Italique*

Indique de nouveaux termes, URL, adresses e-mail, noms de fichiers et extensions de fichiers.

Largeur constante

Utilisé pour les listes de programmes, ainsi que dans les paragraphes pour faire référence à des éléments de programme tels que des noms de variables ou de fonctions, des bases de données, des types de données, des variables d'environnement, des instructions et des mots-clés.

**Largeur constante en gras**

Affiche les commandes ou tout autre texte qui doit être saisi littéralement par l'utilisateur.

*Largeur constante en italique*

Affiche le texte qui doit être remplacé par des valeurs fournies par l'utilisateur ou par des valeurs déterminées par le contexte.

Cette icône signifie un conseil ou une suggestion.

Cette icône signifie une note générale.

Cette icône indique un avertissement ou une mise en garde.

# Exemples de codes

Les exemples sont illustrés en Python, C++, et en utilisant la ligne de commande d'un système d'exploitation de type Unix tel que Linux ou macOS. Tous les extraits de code sont disponibles dans le référentiel GitHub (<https://github.com/bitcoinbook/bitcoinbook>) dans le sous-répertoire *code* du référentiel principal. Fourchez le code du livre, essayez les exemples de code ou soumettez des corrections via GitHub.

Tous les extraits de code peuvent être répliqués sur la plupart des systèmes d'exploitation avec une installation minimale de compilateurs et d'interpréteurs pour les langages correspondants. Si nécessaire, nous fournissons des instructions d'installation de base et des exemples étape par étape de la sortie de ces instructions.

Certains des extraits de code et des sorties de code ont été reformatés pour l'impression. Dans tous ces cas, les lignes ont été séparées par un caractère barre oblique inverse (\), suivi d'un caractère de saut de ligne. Lors de la transcription des exemples, supprimez ces deux caractères et rejoignez les lignes à nouveau et vous devriez voir des résultats identiques comme indiqué dans l'exemple.

Tous les extraits de code utilisent des valeurs réelles et des calculs lorsque cela est possible, de sorte que vous pouvez construire d'exemple en exemple et voir les mêmes résultats dans n'importe quel code que vous écrivez pour calculer les mêmes valeurs. Par exemple, les clés privées et les clés et adresses publiques correspondantes sont toutes réelles. Les exemples de transactions, de blocs et de références de chaîne de blocs ont tous été introduits dans la chaîne de blocs Bitcoin réelle et font partie du grand livre public, vous pouvez donc les consulter sur n'importe quel système Bitcoin.

# Utilisation des exemples de code

Ce livre est là pour vous aider à faire votre travail. En général, si un exemple de code est proposé avec ce livre, vous pouvez l'utiliser dans vos programmes et votre documentation. Vous n'avez pas besoin de nous contacter pour obtenir une autorisation, sauf si vous reproduisez une partie importante du code. Par exemple, écrire un programme qui utilise plusieurs morceaux de code de ce livre ne nécessite pas d'autorisation. La vente ou la distribution d'un CD-ROM d'exemples tirés des livres d'O'Reilly nécessite une autorisation. Répondre à une question en citant ce livre et en citant un exemple de code ne nécessite pas d'autorisation. L'incorporation d'une quantité importante d'exemples de code de ce livre dans la documentation de votre produit nécessite une autorisation.

Nous apprécions, mais n'exigeons pas, l'attribution. Une attribution comprend généralement le titre, l'auteur, l'éditeur et l'ISBN. Par exemple : « *Mastering Bitcoin* par Andreas M. Antonopoulos (O'Reilly). Copyright 2017 Andreas M. Antonopoulos, 978-1-491-95438-6.

Certaines éditions de ce livre sont proposées sous une licence de source libre, comme [CC-BY-NC](https://creativecommons.org/licenses/by-nc/4.0/), auquel cas les termes de cette licence s'appliquent.

Si vous pensez que votre utilisation d'exemples de code ne respecte pas l'utilisation équitable ou l'autorisation donnée ci-dessus, n'hésitez pas à nous contacter à permissions@oreilly.com.

# Adresses et transactions Bitcoin dans ce livre

("codes QR", "avertissements et mises en garde")Les adresses Bitcoin, les transactions, les clés, les codes QR et les données de chaîne de blocs utilisées dans ce livre sont, pour la plupart, réelles. Cela signifie que vous pouvez parcourir la chaîne de blocs, regarder les transactions proposées en exemple, les récupérer avec vos propres scripts ou programmes, etc.

Cependant, notez que les clés privées utilisées pour construire les adresses sont soit imprimées dans ce livre, soit ont été "radiées". Cela signifie que si vous envoyez de l'argent à l'une de ces adresses, l'argent sera perdu pour toujours ou, dans certains cas, tous ceux qui peuvent lire ce livre peuvent le prendre en utilisant les clés privées imprimées dans celui-ci.

N'ENVOYEZ PAS D'ARGENT AUX ADRESSES DANS CE LIVRE. Votre argent sera pris par un autre lecteur, ou perdu à jamais.

# O’Reilly Safari

Safari (anciennement Safari Books Online) est une plate-forme de formation et de référence basée sur l'adhésion pour les entreprises, le gouvernement, les éducateurs et les particuliers.

Les membres ont accès à des milliers de livres, de vidéos de formation, de parcours d'apprentissage, de didacticiels interactifs et de listes de lecture sélectionnées par plus de 250 éditeurs, dont 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 et Course Technology, entre autres.

Pour plus d'informations, rendez-vous sur https://oreilly.com/safari.

# Comment nous contacter

Veuillez adresser vos commentaires et questions concernant ce livre à l'éditeur :

O’Reilly Media, Inc.

1005 Gravenstein Highway North

Sebastopol, CA 95472

800-998-9938 (aux États-Unis ou au Canada)

707-829-0515 (international ou local)

707-829-0104 (fax)

Pour commenter ou poser des questions techniques sur ce livre, envoyez un courriel à l'adresse suivante bookquestions@oreilly.com.

Pour plus d'informations sur nos livres, cours, conférences et actualités, consultez notre site Web au lien : https://www.oreilly.com[].

Retrouvez-nous sur Facebook : <https://facebook.com/oreilly>

Suivez-nous sur Twitter : <https://twitter.com/oreillymedia>

Regardez-nous sur YouTube : <https://www.youtube.com/oreillymedia>

# Contacter l'auteur

Vous pouvez me contacter, Andreas M. Antonopoulos, sur mon site personnel : <https://aantonop.com/>

Des informations sur *Maîtriser Bitcoin (Mastering Bitcoin en anglais)* ainsi que l'Édition Ouverte et les traductions sont disponibles sur : <https://bitcoinbook.info/>

Suivez-moi sur Facebook: <https://facebook.com/AndreasMAntonopoulos>

Suivez-moi sur Twitter: <https://twitter.com/aantonop>

Suivez-moi sur LinkedIn : <https://linkedin.com/company/aantonop>

Un grand merci à tous mes mécènes qui soutiennent mon travail par des dons mensuels. Vous pouvez suivre ma page Patreon ici : <https://patreon.com/aantonop>

# Remerciements

Ce livre représente les efforts et les contributions de nombreuses personnes. Je suis reconnaissant pour toute l'aide que j'ai reçue d'amis, de collègues et même de parfaits inconnus, qui m'ont rejoint dans cet effort pour écrire le livre technique définitif sur les crypto-monnaies et le bitcoin.

Il est impossible de faire une distinction entre la technologie bitcoin et la communauté bitcoin, et ce livre est autant un produit de cette communauté qu'un livre sur la technologie. Mon travail sur ce livre a été encouragé, applaudi, soutenu et récompensé par toute la communauté bitcoin du tout début jusqu'à la toute fin. Plus que tout, ce livre m'a permis de faire partie d'une merveilleuse communauté pendant deux ans et je ne vous remercierai jamais assez de m'avoir accepté dans cette communauté. Il y a beaucoup trop de personnes à mentionner par leur nom - des personnes que j'ai rencontrées lors de conférences, d'événements, de séminaires, de rencontres, de pizzas et de petits rassemblements privés, ainsi que beaucoup qui ont communiqué avec moi par Twitter, sur reddit, sur bitcointalk. org, et sur GitHub qui ont eu un impact sur ce livre. Chaque idée, analogie, question, réponse et explication que vous trouverez dans ce livre a été à un moment donné inspirée, testée ou améliorée grâce à mes interactions avec la communauté. Merci à tous pour votre soutien; sans vous ce livre n'aurait pas vu le jour. Je suis éternellement reconnaissant.

Le voyage pour devenir auteur commence bien avant le premier livre, bien sûr. Ma première langue (et ma scolarité) était le grec, j'ai donc dû suivre un cours de rattrapage en anglais écrit lors de ma première année d'université. Je dois remercier Diana Kordas, mon professeur d'écriture d'anglais, qui m'a aidé à développer ma confiance et mes compétences cette année-là. Plus tard, en tant que professionnel, j'ai développé mes compétences en rédaction technique sur le thème des centres de données, en écrivant pour le magazine *Network World*. Je dois remercier John Dix et John Gallant, qui m'ont donné mon premier travail d'écriture en tant que chroniqueur à *Network World* et à mon éditeur Michael Cooney et mon collègue Johna Till Johnson qui ont édité mes chroniques et les ont rendues aptes à la publication. Écrire 500 mots par semaine pendant quatre ans m'a donné suffisamment d'expérience pour éventuellement envisager de devenir auteur.

Merci également à ceux qui m'ont soutenu lorsque j'ai soumis ma proposition de livre à O'Reilly, en fournissant des références et en examinant la proposition. Plus précisément, merci à John Gallant, Gregory Ness, Richard Stiennon, Joel Snyder, Adam B. Levine, Sandra Gittlen, John Dix, Johna Till Johnson, Roger Ver et Jon Matonis. Remerciements particuliers à Richard Kagan et Tymon Mattoszko, qui ont révisé les premières versions de la proposition et à Matthew Taylor, qui a révisé la proposition.

Merci à Cricket Liu, auteur du titre O'Reilly *DNS et BIND*, qui m'a présenté O'Reilly. Merci également à Michael Loukides et Allyson MacDonald de O'Reilly, qui ont travaillé pendant des mois pour aider à la réalisation de ce livre. Allyson a été particulièrement patiente lorsque les délais ont été manqués et les livrables retardés alors que la vie intervenait dans notre échéancier. Pour la deuxième édition, je remercie Timothy McGovern d'avoir guidé le processus, Kim Cofer d'avoir patiemment édité et Rebecca Panzer d'avoir illustré de nombreux nouveaux diagrammes.

Les premières ébauches des premiers chapitres ont été les plus difficiles, car le bitcoin est un sujet difficile à démêler. Chaque fois que je tirais sur un fil de la technologie bitcoin, je devais tirer sur le tout. Je me suis retrouvé coincé à plusieurs reprises et un peu découragé alors que je luttais pour rendre le sujet facile à comprendre et créer un récit autour d'un sujet technique aussi dense. Finalement, j'ai décidé de raconter l'histoire du bitcoin à travers les histoires des personnes utilisant le bitcoin et tout le livre est devenu beaucoup plus facile à écrire. Je dois remercier mon ami et mentor, Richard Kagan, qui m'a aidé à démêler l'histoire et à surmonter les moments de blocage de l'écrivain. Je remercie Pamela Morgan, qui a révisé les premières ébauches de chaque chapitre de la première et de la deuxième édition du livre, et a posé les questions difficiles pour les améliorer. Merci également aux développeurs du groupe San Francisco Bitcoin Developers Meetup ainsi qu'à Taariq Lewis et Denise Terry pour avoir aidé à tester le premier matériel. Merci également à Andrew Naugler pour la conception infographique.

Pendant le développement du livre, j'ai rendu les premières ébauches disponibles sur GitHub et j'ai invité les commentaires du public. Plus d'une centaine de commentaires, suggestions, corrections et contributions ont été soumis en réponse. Ces contributions sont explicitement reconnues, avec mes remerciements, dans &lt<github\_contrib>>. Surtout, mes sincères remerciements à mes éditeurs bénévoles de GitHub, Ming T. Nguyen (1ère édition) et Will Binns (2e édition), qui ont travaillé sans relâche pour organiser, gérer et résoudre les demandes d'extraction, publier des rapports et effectuer des corrections de bogues sur GitHub.

Une fois le livre rédigé, il est passé par plusieurs séries d'examens techniques. Merci à Cricket Liu et Lorne Lantz pour leur examen approfondi, leurs commentaires et leur soutien.

Plusieurs développeurs de bitcoins ont fourni des échantillons de code, des critiques, des commentaires et des encouragements. Merci à Amir Taaki et Eric Voskuil pour les extraits de code par exemple et de nombreux excellents commentaires ; Chris Kleeschulte pour sa contribution à l'annexe Bitcore ; Vitalik Buterin et Richard Kiss pour leur aide avec les mathématiques des courbes elliptiques et leurs contributions au code ; Gavin Andresen pour les corrections, les commentaires et les encouragements ; Michalis Kargakis pour les commentaires, les contributions et la rédaction de btcd ; et Robin Inge pour les soumissions d'errata améliorant la deuxième impression. Dans la deuxième édition, j'ai de nouveau reçu beaucoup d'aide de la part de nombreux développeurs de Bitcoin Core, dont Eric Lombrozo qui a démystifié le Témoin Séparé, Luke Dashjr qui a aidé à améliorer le chapitre sur les transactions, Johnson Lau qui a revu le Témoin Séparé et d'autres chapitres, et bien d'autres. Je dois remercier Joseph Poon, Tadge Dryja et Olaoluwa Osuntokun qui ont expliqué Lightning Network, revu mes écrits et répondu aux questions lorsque je suis resté bloqué.

Je dois mon amour des mots et des livres à ma mère, Theresa, qui m'a élevé dans une maison avec des livres tapissant tous les murs. Ma mère m'a également acheté mon premier ordinateur en 1982, bien qu'elle se décrive comme une technophobe. Mon père, Menelaos, un ingénieur civil qui vient de publier son premier livre à 80 ans, est celui qui m'a enseigné la pensée logique et analytique et l'amour de la science et de l'ingénierie.

Merci à tous de m'avoir soutenu tout au long de ce parcours.

## Version préliminaire (contributions GitHub)

De nombreux contributeurs ont proposé des commentaires, des corrections et des ajouts à la première version préliminaire sur GitHub. Merci à tous pour vos contributions à ce livre.

Voici une liste des contributeurs notables de GitHub :

Unresolved directive in for\_translation\_mastering-bitcoin\_preface\_fr\_CA.asciidoc - include::github\_contrib.asciidoc[]

# Glossaire rapide

Ce glossaire rapide contient de nombreux termes utilisés en relation avec le bitcoin. Ces termes sont utilisés tout au long du livre, alors mettez-les en signet pour une référence rapide.

adresse

Une adresse Bitcoin ressemble à 1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV. Il se compose d'une suite de lettres et de chiffres. Il s'agit en réalité d'une version codée en base58check d'un résultat de hachage à clé publique de 160 bits. Tout comme vous demandez aux autres d'envoyer un courriel à l’adresse de celle-ci, vous demanderez aux autres de vous envoyer des bitcoins à l'une de vos adresses Bitcoin.

bip

Bitcoin Improvement Proposals (ou Propositions d’améliorations du Bitcoin). Un ensemble de propositions que les membres de la communauté Bitcoin ont soumises pour améliorer Bitcoin. Par exemple, BIP-21 est une proposition visant à améliorer le schéma d'identificateur de ressource uniforme (URI) bitcoin.

bitcoin

Le nom de l'unité monétaire (unité de cryptomonnaie), du réseau et du logiciel.

bloc

Regroupement de transactions, marqué d'un horodatage et d'une empreinte digitale du bloc précédent. L'en-tête du bloc est haché (c.-à-d., on obtient un résultat de hachage) pour produire une preuve de travail, validant ainsi les transactions. Les blocs valides sont ajoutés à la chaîne de blocs principale par consensus du réseau.

blockchain (ou chaîne de blocs)

Une liste de blocs validés, chacun lié à son prédécesseur jusqu'au bloc de genèse.

récompense de bloc (alias récompense coinbase (ou unité de monnaie primaire du bloc)

Un montant inclus dans chaque nouveau bloc comme récompense par le réseau au mineur qui a trouvé la solution Proof-of-Work (ou Preuve de travail)). Environ tous les quatre ans, ou plus précisément tous les 210 000 blocs, la récompense de bloc globale est réduite de moitié. Il est actuellement de 6,25 BTC par bloc.

Byzantine Generals Problem (ou Problème des généraux Byzantins)

Un système informatique fiable doit pouvoir faire face à la défaillance d'un ou plusieurs de ses composants. Un composant défaillant peut présenter un type de comportement souvent négligé, à savoir l'envoi d'informations contradictoires à différentes parties du système. Le problème de faire face à ce type d'échec est exprimé abstraitement comme le Byzantine Generals Problem (ou Problème des généraux byzantins) (voir <https://fr.wikipedia.org/wiki/Probl%C3%A8me_des_g%C3%A9n%C3%A9raux_byzantins>).

bloc candidat

Un bloc qu'un mineur essaie toujours d'exploiter. Ce n'est pas encore un bloc valide, car il ne contient pas de Proof-of-Work (ou Preuve de travail) valide.

coinbase (ou unité de monnaie primaire du bloc) (alias données coinbase)

Un champ spécial utilisé comme seule entrée pour les transactions coinbase. Le champ de données coinbase permet de réclamer la récompense de bloc et fournit jusqu'à 100 octets pour les données arbitraires. À ne pas confondre avec la "transaction coinbase" ou la "récompense coinbase".

transaction coinbase

La première transaction d'un bloc; toujours créé par un mineur. Le coinbase est unique dans le bloc. À ne pas confondre avec coinbase (données coinbase) ou "récompense coinbase".

stockage à froid

Fait référence à la conservation d'une réserve de bitcoins hors ligne. Le stockage à froid est réalisé lorsque les clés privées bitcoin sont créées et stockées dans un environnement hors ligne sécurisé. Le stockage à froid est important pour quiconque possède des avoirs en bitcoins. Les ordinateurs en ligne sont vulnérables aux pirates et ne doivent pas être utilisés pour stocker une quantité importante de bitcoins. (note du traducteur: l’auteur est un peu paranoïde; les noeuds sont en ligne. Si les noeuds sont insécures, c’est la fin de ce système et de ce livre.) ;)

confirmations

Une fois qu'une transaction est incluse dans un bloc, elle a une confirmation. Dès qu'un autre bloc est miné sur la même chaîne de blocs, la transaction a deux confirmations, et ainsi de suite. Six confirmations ou plus sont considérées comme une preuve suffisante qu'une transaction ne peut pas être annulée.

consensus

Lorsque plusieurs nœuds, généralement la plupart des nœuds du réseau, ont tous les mêmes blocs dans leur meilleure chaîne de blocs validée localement. À ne pas confondre avec les règles de consensus.

règles de consensus

Les règles de validation de bloc que les nœuds complets suivent pour rester en consensus avec les autres nœuds. A ne pas confondre avec le consensus.

difficulté

Un paramètre à l'échelle du réseau qui contrôle la quantité de calculs nécessaires pour produire une preuve de travail (ou le Proof-of-Work).

reciblage de difficulté

Un recalcul à l'échelle du réseau de la difficulté qui se produit une fois tous les 2 016 blocs et prend en compte la puissance de hachage des 2 016 blocs précédents.

cible de difficulté

Une difficulté pour le calcul dans tout le réseau et qui permet de solutionner et valider l’ajout d’un bloc environ toutes les 10 minutes.

double dépense

La double dépense est le résultat d'avoir réussi à dépenser de l'argent plus d'une fois. Bitcoin protège contre les doubles dépenses en vérifiant chaque transaction ajoutée à la chaîne de blocs pour s'assurer que les entrées de la transaction n'avaient pas déjà été dépensées auparavant.

ECDSA

Elliptic Curve Digital Signature Algorithm ou ECDSA est un algorithme cryptographique utilisé par bitcoin pour garantir que les fonds ne peuvent être dépensés que par leurs propriétaires légitimes.

nombre aléatoire explicite supplémentaire

Au fur et à mesure que la difficulté augmentait, les mineurs parcouraient souvent les 4 milliards de nombres aléatoires explicites (nonce) sans trouver de bloc. Étant donné que le script coinbase peut stocker entre 2 et 100 octets de données, les mineurs ont commencé à utiliser cet espace comme espace pour un nombre aléatoire explicite supplémentaire, leur permettant d'explorer une gamme beaucoup plus large de valeurs d'en-tête de bloc pour trouver des blocs valides.

frais

L'expéditeur d'une transaction inclut souvent une valeur de frais au réseau pour le traitement de la transaction demandée. La plupart des transactions nécessitent des frais minimum de 0,5 mBTC.

fourche

La fourche, également connue sous le nom de fourche accidentelle, se produit lorsque deux blocs ou plus ont la même hauteur de bloc, forçant la chaîne de blocs à se dédoubler. Cela se produit généralement lorsque deux mineurs ou plus trouvent des blocs presque en même temps. Peut également se produire dans le cadre d'une attaque.

bloc de genèse

Le premier bloc de la chaîne de blocs, utilisé pour initialiser la cryptomonnaie (ou cybermonnaie).

halving (ou réduction de moitié)

Un événement de réduction de moitié se produit lorsque la récompense de bloc est réduite de moitié, ce qui se produit environ tous les quatre ans (ou précisément tous les 210 000 blocs). Bitcoin a déjà connu trois événements de réduction de moitié : en 2012 (de 50 à 25 BTC), en 2016 (de 25 à 12,5 BTC) et en 2020 (de 12,5 à 6,25 BTC).

fourche dure

La fourche dure, également connu sous le nom de Hard-Forking Change, est une divergence permanente dans la chaîne de blocs, se produit généralement lorsque les nœuds non mis à niveau ne peuvent pas valider les blocs créés par les nœuds mis à niveau qui suivent les nouvelles règles de consensus. À ne pas confondre avec une fourche, une fourche molle, une fourche de code ou une fourche Git, mais tout de même un nouvel embranchement d’une arborescence de données (avec une source parent unique).

portefeuille matériel (hardware)

Un portefeuille matériel est un type spécial de portefeuille bitcoin qui stocke les clés privées de l'utilisateur dans un périphérique matériel sécurisé.

résultat de hachage

Une empreinte numérique d'une entrée binaire.

hashlocks (ou serrure par empreinte numérique)

Un hashlock (ou serrure par empreinte numérique) est un type de contrainte qui limite les dépenses d'une sortie jusqu'à ce qu'une donnée spécifique soit révélée publiquement. Les hashlocks ont la propriété utile qu'une fois qu'un hashlock est ouvert publiquement, tout autre hashlock sécurisé à l'aide de la même clé peut également être ouvert. Cela permet de créer plusieurs sorties qui sont toutes encombrées par le même hashlock et qui deviennent toutes dépensables en même temps.

Protocole HD

Le protocole de création et de transfert de clés Hierarchical Deterministic (HD) (BIP-32), qui permet de créer des clés enfants à partir de clés parents dans une hiérarchie.

Portefeuille HD

Portefeuilles utilisant le protocole de création et de transfert de clés Hierarchical Deterministic (HD Protocol) (BIP-32).

Portefeuille d’amorçage HD

Le portefeuille d’amorçage HD ou la valeur d’amorçage est une valeur potentiellement courte utilisée comme valeur racine pour générer la clé privée principale et le code de chaîne principal pour un portefeuille HD.

HTLC

Un contrat Hashed TimeLock ou HTLC est une classe de paiements qui utilisent des hashlocks (ou serrures par empreintes numériques) et des timelocks (ou serrures par horodatage) pour exiger que le destinataire d'un paiement reconnaisse avoir reçu le paiement avant une date limite en générant une preuve cryptographique de paiement ou renonce à la possibilité de réclamer le paiement, le retournant au payeur.

KYC

Know Your Customer (ou Connaître son client) (KYC) est le processus d'une entreprise qui identifie et vérifie l'identité de ses clients. Le terme est également utilisé pour désigner la réglementation bancaire qui régit ces activités.

LevelDB

LevelDB est un moyen de stockage clé-valeur open source sur disque. LevelDB est une bibliothèque légère à usage unique pour la persistance avec des liaisons sur de nombreuses plates-formes.

Lightning Networks

Lightning Network est une implémentation de Hashed Timelock Contracts (HTLC) avec des canaux de paiement bidirectionnels qui permettent aux paiements d'être acheminés en toute sécurité sur plusieurs canaux de paiement pair à pair. Cela permet la formation d'un réseau où n'importe quel pair sur le réseau peut payer n'importe quel autre pair même s'ils n'ont pas directement de canal ouvert entre eux.

Locktime (ou valeur de relâche de transaction)

Locktime, ou plus techniquement nLockTime, est la partie d'une transaction qui indique la première heure où le premier bloc auquel cette transaction peut être ajoutée à la chaîne de blocs. C’est une valeur horodatée servant de date ultérieur pour une transaction; comme un chèque postdaté.

mempool

Le bitcoin Mempool (bassin de mémoire) est une collection de toutes les données de transaction dans un bloc qui ont été vérifiées par les nœuds Bitcoin, mais qui ne sont pas encore confirmées.

racine de merkle

Le nœud racine d'un arbre Merkle, un descendant de toutes les paires hachées de l'arbre. Les en-têtes de bloc doivent inclure une racine merkle valide issue de toutes les transactions de ce bloc.

arbre de merkle

Un arbre construit en hachant des données appariées (les feuilles), puis en appariant et en hachant les résultats jusqu'à ce qu'il ne reste qu'un seul résultat de hachage, la racine de Merkle. En bitcoin, les feuilles sont presque toujours des transactions d'un seul bloc.

mineur

Un nœud de réseau qui trouve une preuve de travail valide pour les nouveaux blocs, par hachage répété.

récompense minière

Les récompense que les mineurs reçoivent en échange de la sécurité fournie par l'exploitation minière, comprend les nouvelles pièces bitcoin créées avec chaque nouveau bloc, également appelées récompense de bloc ou récompense coinbase, et les frais de transaction de toutes les transactions incluses dans le bloc.

multisignature

La multisignature (multisig) fait référence à l'exigence d'un nombre minimum (M) de clés (N) pour autoriser une transaction M-sur-N.

réseau

Un réseau pair à pair qui propage les transactions et les bloques à chaque nœud Bitcoin du réseau.

nonce (ou nombre aléatoire explicite)

Le "nonce" (ou nombre aléatoire explicite) dans un bloc bitcoin est un champ de 32 bits (4 octets) dont la valeur est définie de manière à ce que le résultat de hachage du bloc contienne une série de zéros non significatifs. Les autres champs ne peuvent pas être modifiés, car ils ont une signification définie.

transactions hors chaîne

Une transaction hors chaîne est le mouvement de valeur en dehors de la chaîne de blocs, alors qu'une transaction en chaîne&#x2014 est simplement appelée *une transaction*&#x2014. La transaction hors chaîne modifie la chaîne de blocs et dépend de cette dernière pour déterminer sa validité. Une transaction hors chaîne de blocs s'appuie sur d'autres méthodes pour enregistrer et valider la transaction.

opcode

Codes d'opération du langage de script bitcoin qui poussent des données ou exécutent des fonctions dans un script pubkey ou un script de signature.

Protocole Open Assets (ou Actifs ouverts)

Le protocole Open Assets (ou Actifs ouverts) est un protocole simple et puissant construit au-dessus de la chaîne de blocs Bitcoin. Il permet l'émission et le transfert d'actifs créés par l'utilisateur.

OP\_RETURN

Un opcode (ou code d’opération) utilisé dans l'une des sorties d'une transaction OP\_RETURN. A ne pas confondre avec la transaction OP\_RETURN.

Transaction OP\_RETURN

Un type de transaction qui ajoute des données arbitraires à un script pubkey prouvé que les nœuds complets n'ont pas à stocker dans leur base de données UTXO. A ne pas confondre avec l'opcode OP\_RETURN.

bloc orphelin

Les blocs dont le bloc parent n'a pas été traité par le nœud local, ils ne peuvent donc pas encore être entièrement validés. À ne pas confondre avec le bloc obsolète.

orphan transactions

Transactions qui ne peuvent pas entrer dans le bassin de mémoire (Mempool) en raison d'une ou plusieurs transactions d'entrée manquantes.

extrant

L’Output (le produit sorti ou l’extrant), la sortie de transaction ou TxOut est une sortie dans une transaction qui contient deux champs : un champ de valeur pour transférer zéro ou plusieurs satoshis et un script pubkey pour indiquer quelles conditions doivent être remplies pour que ces satoshis soient dépensés davantage.

P2PKH

Les transactions qui paient une adresse Bitcoin contiennent des scripts P2PKH ou Pay To PubKey Hash. Une sortie verrouillée par un script P2PKH peut être déverrouillée (dépensée) en présentant une clé publique et une signature numérique créée par la clé privée correspondante.

P2SH

P2SH ou Pay-to-Script-Hash est un nouveau type de transaction puissant qui simplifie grandement l'utilisation de scripts de transaction complexes. Avec P2SH le script complexe qui détaille les conditions de dépense de la sortie (redeem script) n'est pas présenté dans le script de verrouillage. Au lieu de cela, seul un résultat de hachage de celui-ci se trouve dans le script de verrouillage.

Adresse P2SH

Les adresses P2SH sont des encodages Base58Check du résultat de hachage de 20 octets d'un script. Ils utilisent le préfixe de version "5", ce qui donne des adresses encodées en Base58Check qui commencent par un "3". Les adresses P2SH cachent toute la complexité, de sorte que la personne effectuant un paiement ne voit pas le script.

P2WPKH

La signature d'un P2WPKH (Pay-to-Witness-Public-Key-Hash) contient les mêmes informations qu'une dépense P2PKH, mais se trouve dans le champ témoin au lieu du champ scriptSig. Le scriptPubKey est également modifié.

P2WSH

La différence entre P2SH et P2WSH (Pay-to-Witness-Script-Hash) concerne le changement d'emplacement de la preuve cryptographique du champ scriptSig au champ témoin et le scriptPubKey qui est également modifié.

portefeuille papier

Dans le sens le plus spécifique, un portefeuille papier est un document contenant toutes les données nécessaires pour générer un nombre quelconque de clés privées bitcoin, formant un portefeuille de clés. Cependant, les gens utilisent souvent le terme pour désigner tout moyen de stocker des bitcoins hors ligne en tant que document physique. Cette deuxième définition comprend également les clés papier et les codes échangeables.

phrase de passe

Une phrase de passe est une chaîne de caractères facultative créée par l'utilisateur qui sert de facteur de sécurité supplémentaire protégeant la valeur d’amorçage, même lorsque la valeur d’amorçage est compromise par un voleur. Il peut également être utilisé comme une forme de pot de miel, où une phrase de passe choisie (une adresse bitcoin, un URL ou autre) mène à un portefeuille avec une petite quantité de fonds utilisée pour distraire un attaquant du "vrai" portefeuille qui contient la majorité des fonds.

canaux de paiement

Un canal de micropaiement ou canal de paiement est une classe de techniques conçues pour permettre aux utilisateurs d'effectuer plusieurs transactions bitcoin sans engager toutes les transactions dans la chaîne de blocs Bitcoin. Dans un canal de paiement typique, seules deux transactions sont ajoutées à la chaîne de blocs, mais un nombre illimité ou presque illimité de paiements peut être effectué entre les participants.

exploitation minière groupée

L'exploitation minière en groupe est une approche d'exploitation minière dans laquelle plusieurs clients générateurs contribuent à la génération d'un bloc, puis divisent la récompense du bloc en fonction de la puissance de traitement apportée.

Proof-of-Stake (ou Preuve d’enjeu)

La preuve d’enjeu (PoS) est une méthode par laquelle un réseau de chaîne de blocs de cryptomonnaie vise à atteindre un consensus distribué. La preuve d’enjeu demande aux utilisateurs de prouver qu'ils sont propriétaires d'un certain montant de devise (leur « enjeu » ou « participation » dans la devise).

Proof-of-Work (ou Preuve de travail)

Une donnée qui nécessite des calculs importants pour être trouvée. Dans le bitcoin, les mineurs doivent trouver une solution numérique à l'algorithme SHA256 qui répond à une cible à l'échelle du réseau, la cible de difficulté.

RIPEMD-160

RIPEMD-160 est une fonction de hachage cryptographique 160 bits. RIPEMD-160 est une version renforcée de RIPEMD avec un résultat de hachage de 160 bits, et devrait être sécurisé pour les dix prochaines années ou plus.

satoshi

Un satoshi est la plus petite dénomination de bitcoin pouvant être enregistrée sur la blockchain. Il équivaut à 0,00000001 bitcoin et porte le nom du créateur du bitcoin, Satoshi Nakamoto.

Satoshi Nakamoto

Satoshi Nakamoto est le nom utilisé par la ou les personnes qui ont conçu le bitcoin et créé son implémentation de référence originale, Bitcoin Core. Dans le cadre de la mise en œuvre, ils ont également conçu la première base de données chaîne de blocs. Dans le processus, ils ont été les premiers à résoudre le problème de la double dépense pour la monnaie numérique. Leur véritable identité reste inconnue.

Script

Bitcoin utilise un système de script pour les transactions. Comme Forth, Script est simple, basé sur une pile et traité de gauche à droite. Il n'est délibérément pas complet de Turing et sans boucles.

ScriptPubKey (alias script pubkey): : ScriptPubKey ou script pubkey, est un script inclus dans les sorties qui définit les conditions qui doivent être remplies pour que ces satoshis soient dépensés. Les données permettant de remplir les conditions peuvent être fournies dans un script de signature.

ScriptSig (ou Script de signature): : ScriptSig ou script de signature, sont les données générées par un dépensier qui sont presque toujours utilisées comme variables pour satisfaire un script pubkey.

clé secrète (ou clé privée): : Le numéro secret qui déverrouille le bitcoin envoyé à l'adresse correspondante. Une clé secrète ressemble à ceci :

+

5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh

Témoin Séparé

Le témoin séparé est une mise à niveau du protocole Bitcoin dans laquelle les données de signature ("témoin") sont séparées des données de l'expéditeur/destinataire pour optimiser davantage la structure des transactions. Le témoin Séparé a été implémenté comme un embranchement convergent ; un changement qui rend techniquement les règles du protocole de bitcoin plus restrictives.

SHA

Le Secure Hash Algorithm ou SHA est une famille de fonctions de hachage cryptographiques publiées par le National Institute of Standards and Technology (NIST).

Simplified Payment Verification (SPV ou Vérification simplifiée des paiements): : SPV ou vérification simplifiée des paiements est une méthode permettant de vérifier que des transactions particulières ont été incluses dans un bloc, sans télécharger le bloc entier. Cette méthode de vérification est souvent utilisée par les clients Bitcoin légers.

embranchement convergent

L’embranchement convergent ou le Soft-Forking Change est un fourche temporaire dans la chaîne de blocs qui se produit généralement lorsque les mineurs utilisant des nœuds non mis à niveau ne suivent pas une nouvelle règle de consensus que leurs nœuds ne connaissent pas. À ne pas confondre avec fourche, fourche dure, fourche logicielle ou fourche de Git.

bloc périmé

Un bloc valide qui a été extrait avec succès mais qui n'est pas inclus dans la branche principale actuelle (avec la plupart des preuves de travail cumulatives), car un autre bloc valide qui a été extrait à la même hauteur a vu sa chaîne étendue en premier. Le mineur d'un bloc périmé ne reçoit pas la récompense du bloc ni les frais de transaction de ce bloc. À ne pas confondre avec le bloc orphelin ou le bloc candidat.

timelocks (ou serrures horaires)

Un timelock (ou serrure horaire) est un type de contrainte qui limite la dépense de certains bitcoins jusqu'à une heure future ou une hauteur de bloc spécifiée. Les timelocks figurent en bonne place dans de nombreux contrats bitcoin, y compris les canaux de paiement et les contrats de timelock hachés.

transaction

En termes simples, un transfert de bitcoin d'une adresse à une autre. Plus précisément, une transaction est une structure de données signée exprimant un transfert de valeur. Les transactions sont transmises sur le réseau Bitcoin, collectées par les mineurs et incluses dans des blocs, rendus permanents sur la chaîne de blocs.

bassin de transactions

Une collection non ordonnée de transactions qui ne sont pas en blocs dans la chaîne principale, mais pour lesquelles nous avons des transactions d'entrée.

Complétude de Turing

Un langage de programmation est appelé "Turing complet" s'il peut exécuter n'importe quel programme qu'une machine de Turing peut exécuter, avec suffisamment de temps et de mémoire.

unspent transaction output (UTXO ou sortie de transaction non dépensée)

UTXO est une sortie de transaction non dépensée qui peut être dépensée comme entrée dans une nouvelle transaction.

portefeuille

Logiciel qui contient toutes vos adresses Bitcoin et clés secrètes. Utilisez-le pour envoyer, recevoir et stocker vos bitcoins.

Wallet Import Format (WIF ou Format d’importation de portefeuille)

WIF ou Wallet Import Format (format d’importation de portefeuille) est un format d'échange de données conçu pour permettre l'exportation et l'importation d'une seule clé privée avec un indicateur indiquant s'il utilise ou non une clé publique compressée.

Certaines définitions fournies ont été extraites sous une licence CC-BY de [bitcoin Wiki](https://en.bitcoin.it/wiki/Main_Page) ou d'autres documentations à source libre.

# Introduction

# Qu'est-ce que Bitcoin ?

Bitcoin est un ensemble de concepts et de technologies qui constituent la base d'un écosystème de monnaie numérique. Les unités monétaires appelées bitcoin sont utilisées pour stocker et transmettre de la valeur entre les participants au réseau Bitcoin. Les utilisateurs de Bitcoin communiquent entre eux en utilisant le protocole Bitcoin principalement via Internet, bien que d'autres réseaux de transport puissent également être utilisés. La pile de protocole Bitcoin, disponible en tant que logiciel à source libre, peut être exécutée sur une large gamme d'appareils informatiques, y compris les ordinateurs portables et les smartphones, ce qui rend la technologie facilement accessible.

Remarquez comment l'unité monétaire s'appelle "bitcoin" avec un petit *b*, et le système s'appelle "Bitcoin", avec un *B* majuscule.

Les utilisateurs peuvent transférer des bitcoins sur le réseau pour faire à peu près tout ce qui peut être fait avec des devises conventionnelles, y compris acheter et vendre des biens, envoyer de l'argent à des personnes ou à des organisations ou accorder un crédit. Le bitcoin peut être acheté, vendu et échangé contre d'autres devises dans des bureaux de change spécialisés. Le bitcoin est en un sens la forme d'argent idéale pour Internet car il est rapide, sécurisé et sans frontières.

Contrairement aux monnaies traditionnelles, le bitcoin est entièrement virtuel. Il n'y a pas de pièces physiques ni même de pièces numériques en soi. Les pièces sont impliquées dans des transactions qui transfèrent de la valeur de l'expéditeur au destinataire. Les utilisateurs de Bitcoin possèdent des clés qui leur permettent de prouver la propriété du bitcoin dans le réseau Bitcoin. Avec ces clés, ils peuvent signer des transactions pour débloquer la valeur et la dépenser en la transférant à un nouveau propriétaire. Les clés sont souvent stockées dans un portefeuille numérique sur l'ordinateur ou le smartphone de chaque utilisateur. La possession de la clé qui peut signer une transaction est la seule condition préalable pour dépenser du bitcoin, mettant le contrôle entièrement entre les mains de chaque utilisateur.

Bitcoin est un système distribué pair à pair. En tant que tel, il n'y a pas de serveur "central" ou de point de contrôle. Les unités de bitcoin sont créés par un processus appelé "minage", qui implique une compétition pour trouver des solutions à un problème mathématique lors du traitement des transactions Bitcoin. Tout participant au réseau Bitcoin (c'est-à-dire toute personne utilisant un appareil exécutant la pile complète du protocole Bitcoin) peut opérer en tant que mineur, en utilisant la puissance de traitement de son ordinateur pour vérifier et enregistrer les transactions. Toutes les 10 minutes, en moyenne, un mineur de Bitcoin peut valider les transactions et est récompensé par un tout nouveau bitcoin. Essentiellement, l'extraction de Bitcoin décentralise les fonctions d'émission et de compensation de devises d'une banque centrale et remplace le besoin de cette dernière.

Le protocole Bitcoin comprend des algorithmes intégrés qui régulent la fonction de minage sur le réseau. La difficulté de la tâche de traitement que les mineurs doivent effectuer est ajustée dynamiquement de sorte qu'en moyenne, quelqu'un réussisse toutes les 10 minutes, quel que soit le nombre de mineurs (et la quantité de traitement) en compétition à tout moment. Le protocole réduit également de moitié le taux de création de nouveaux bitcoins tous les 4 ans et limite le nombre total de bitcoins qui seront créés à un total fixe juste en dessous de 21 millions de pièces. Le résultat est que le nombre de bitcoins en circulation suit de près une courbe facilement prévisible qui approche les 21 millions d'ici 2140. En raison du taux d'émission décroissant du bitcoin, sur le long terme, la monnaie Bitcoin est déflationniste. De plus, le bitcoin ne peut pas être gonflé en "imprimant" de la nouvelle monnaie au-delà du taux d'émission prévu.

Dans les coulisses, Bitcoin est aussi le nom du protocole, un réseau pair à pair et une innovation informatique distribuée. La monnaie bitcoin n'est vraiment que la première application de cette invention. Bitcoin représente l'aboutissement de décennies de recherche en cryptographie et en systèmes distribués et comprend quatre innovations clés réunies dans une combinaison unique et puissante. Bitcoin se compose de :

* Un réseau pair à pair décentralisé (le protocole Bitcoin)
* Un registre public des transactions (la chaîne de blocs)
* Un ensemble de règles pour la validation indépendante des transactions et l'émission de devises (règles de consensus)
* Un mécanisme pour parvenir à un consensus mondial décentralisé sur la chaîne de blocs valide (algorithme de Proof-of-Work (ou Preuve de travail))

En tant que développeur, je vois Bitcoin comme un Internet de l'argent, un réseau pour propager la valeur et sécuriser la propriété des actifs numériques via un calcul distribué. Il y a beaucoup plus à Bitcoin qu'il n'y paraît à première vue.

Dans ce chapitre, nous commencerons par expliquer certains des concepts et termes principaux, et obtenir le logiciel nécessaire et utiliser Bitcoin pour des transactions simples. Dans les chapitres suivants, nous commencerons à déballer les couches de technologie qui rendent Bitcoin possible et examinerons le fonctionnement interne du réseau et du protocole Bitcoin.

L'émergence d'une monnaie numérique viable est étroitement liée aux développements de la cryptographie. Cela n'est pas surprenant si l'on considère les défis fondamentaux liés à l'utilisation de bits pour représenter la valeur qui peut être échangée contre des biens et des services. Trois questions fondamentales pour quiconque accepte de l'argent numérique sont :

1. Puis-je avoir la certitude que l'argent est authentique et non contrefait ?
2. Puis-je avoir confiance que l'argent numérique ne peut être dépensé qu'une seule fois (connu sous le nom de problème de « double dépense ») ?
3. Puis-je être sûr que personne d'autre ne pourra prétendre que cet argent lui appartient et pas à moi ?

Les émetteurs de papier-monnaie luttent constamment contre le problème de la contrefaçon en utilisant des papiers et des technologies d'impression de plus en plus sophistiqués. L'argent physique résout facilement le problème des doubles dépenses, car le même billet papier ne peut pas se trouver à deux endroits à la fois. Bien sûr, l'argent conventionnel est aussi souvent stocké et transmis numériquement. Dans ces cas, les problèmes de contrefaçon et de double dépense sont traités en compensant toutes les transactions électroniques par des autorités centrales qui ont une vue globale de la monnaie en circulation. Pour la monnaie numérique, qui ne peut tirer profit des encres ésotériques ou des bandes holographiques, la cryptographie fournit la base pour faire confiance à la légitimité de la revendication de valeur d'un utilisateur. Plus précisément, les signatures numériques cryptographiques permettent à un utilisateur de signer un actif numérique ou une transaction prouvant la propriété de cet actif. Avec l'architecture appropriée, les signatures numériques peuvent également être utilisées pour résoudre le problème des doubles dépenses.

Lorsque la cryptographie a commencé à devenir plus largement disponible et comprise à la fin des années 1980, de nombreux chercheurs ont commencé à essayer d'utiliser la cryptographie pour créer des monnaies numériques. Ces premiers projets de monnaie numérique émettaient de la monnaie numérique, généralement adossée à une monnaie nationale ou à un métal précieux comme l'or.

Bien que ces anciennes monnaies numériques fonctionnaient, elles étaient centralisées et, par conséquent, faciles à attaquer par les gouvernements et les pirates. Les premières monnaies numériques utilisaient une chambre de compensation centrale pour régler toutes les transactions à intervalles réguliers, tout comme un système bancaire traditionnel. Malheureusement, dans la plupart des cas, ces monnaies numériques naissantes ont été ciblées par des gouvernements inquiets et ont finalement été dissoutes. Certains ont échoué dans des crashs spectaculaires lorsque la société mère a été liquidée brutalement. Pour être robuste contre l'intervention d'antagonistes, qu'il s'agisse de gouvernements légitimes ou d'éléments criminels, une monnaie numérique *décentralisée* était nécessaire pour éviter un seul point d'attaque. Bitcoin est un tel système, décentralisé par conception, et libre de toute autorité centrale ou point de contrôle qui peut être attaqué ou corrompu.

# Histoire de Bitcoin

Bitcoin a été inventé en 2008 avec la publication d'un article intitulé "Bitcoin : A Peer-to-Peer Electronic Cash System,"[[1]](#footnote-62) écrit sous l'alias de Satoshi Nakamoto (voir [???](#satoshi_whitepaper)). Nakamoto a combiné plusieurs inventions antérieures telles que b-money et HashCash pour créer un système de paiement électronique entièrement décentralisé qui ne repose pas sur une autorité centrale pour l'émission de devises ou le règlement et la validation des transactions. L'innovation clé consistait à utiliser un système de calcul distribué (appelé algorithme de « preuve de travail ») pour effectuer une « élection » globale toutes les 10 minutes, permettant au réseau décentralisé d'arriver à un *consensus* sur l'état des transactions. Cela résout élégamment le problème de la double dépense où une seule unité monétaire peut être dépensée deux fois. Auparavant, le problème des doubles dépenses était une faiblesse de la monnaie numérique et était résolu en compensant toutes les transactions via une chambre de compensation centrale.

Le réseau Bitcoin a démarré en 2009, sur la base d'une référence d’implémentation publiée par Nakamoto et révisée depuis par de nombreux autres programmeurs. La mise en œuvre de l'algorithme Proof-of-Work (ou Preuve de travail ou minage) qui assure la sécurité et la résilience de Bitcoin a augmenté de manière exponentielle et dépasse désormais la puissance de traitement combinée des meilleurs supercalculateurs du monde. La valeur marchande totale de Bitcoin a parfois dépassé 1 billion de dollars américains, selon le taux de change bitcoin-dollar. La transaction la plus importante traitée jusqu'à présent par le réseau était de 1,1 milliard de dollars américains, transmise instantanément et traitée pour des frais de seulement 0,68 $.

Satoshi Nakamoto s'est retiré du public en avril 2011, laissant la responsabilité de développer le code et le réseau à un groupe prospère de bénévoles. L'identité de la personne ou des personnes derrière Bitcoin est encore inconnue. Cependant, ni Satoshi Nakamoto ni personne d'autre n'exerce de contrôle individuel sur le système Bitcoin, qui fonctionne sur la base de principes mathématiques totalement transparents, d'un code à source libre et d'un consensus entre les participants. L'invention elle-même est révolutionnaire et a déjà engendré une nouvelle science dans les domaines de l'informatique distribuée, de l'économie et de l'économétrie.

L'invention de Satoshi Nakamoto est également une solution pratique et nouvelle à un problème d'informatique distribuée, connu sous le nom de "Problème des généraux byzantins". Brièvement, le problème consiste à essayer de s'entendre sur une ligne de conduite ou sur l'état d'un système en échangeant des informations sur un réseau peu fiable et potentiellement compromis. La solution de Satoshi Nakamoto, qui utilise le concept de preuve de travail pour parvenir à un consensus *sans autorité centrale de confiance*, représente une percée dans l'informatique distribuée et a une large applicabilité au-delà de la monnaie. Il peut être utilisé pour parvenir à un consensus sur les réseaux décentralisés afin de prouver l'équité des élections, des loteries, des registres d'actifs, de la notarisation numérique, etc.

# Utilisations, utilisateurs et histoires de Bitcoin

Bitcoin est une innovation dans l'ancienne technologie de l'argent. À la base, l'argent facilite simplement l'échange de valeur entre les personnes. Par conséquent, afin de bien comprendre Bitcoin et ses utilisations, nous l'examinerons du point de vue des personnes qui l'utilisent. Chacune des personnes et de leurs histoires, telles qu'énumérées ici, illustre un ou plusieurs cas d'utilisation spécifiques. Nous les verrons tout au long du livre :

Vente au détail de faible valeur en Amérique du Nord

Alice vit dans la région de la baie de Californie du Nord. Elle a entendu parler de Bitcoin par ses amis technophiles et veut commencer à l'utiliser. Nous suivrons son histoire au fur et à mesure qu'elle découvre le Bitcoin, en acquiert, puis dépense une partie de son bitcoin pour acheter une tasse de café au Bob's Cafe à Palo Alto. Cette histoire nous présentera le logiciel, les échanges et les transactions de base du point de vue d'un consommateur de détail.

Vente au détail de grande valeur en Amérique du Nord

Carol est propriétaire d'une galerie d'art à San Francisco. Elle vend des peintures chères pour Bitcoin. Cette histoire présentera les risques d'une attaque de consensus "51%" pour les détaillants d'articles de grande valeur.

Services contractuels extraterritorial

Bob, le propriétaire d'un café à Palo Alto, est en train de créer un nouveau site Web. Il a passé un contrat avec un développeur Web indien, Gopesh, qui vit à Bangalore, en Inde. Gopesh a accepté d'être payé en bitcoin. Cette histoire examinera l'utilisation de Bitcoin pour l'externalisation, les services contractuels et les virements internationaux.

Commerce en ligne

Gabriel est un jeune adolescent entreprenant à Rio de Janeiro, qui dirige une petite boutique en ligne qui vend des t-shirts, des tasses à café et des autocollants de marque Bitcoin. Gabriel est trop jeune pour avoir un compte bancaire, mais ses parents encouragent son esprit d'entreprise.

Dons caritatifs

Eugenia est directrice d'une association caritative pour enfants aux Philippines. Récemment, elle a découvert Bitcoin et souhaite l'utiliser pour atteindre un tout nouveau groupe de donateurs étrangers et nationaux afin de collecter des fonds pour son association caritative. Elle étudie également les moyens d'utiliser Bitcoin pour distribuer rapidement des fonds aux zones qui en ont besoin. Cette histoire montrera l'utilisation de Bitcoin pour la collecte de fonds mondiale à travers les devises et les frontières et l'utilisation d'un registre ouvert pour la transparence dans les organisations caritatives.

Importation/Exportation

Mohammed est un importateur d'électronique à Dubaï. Il essaie d'utiliser Bitcoin pour acheter des appareils électroniques aux États-Unis et en Chine pour les importer aux Émirats arabes unis afin d'accélérer le processus de paiement des importations. Cette histoire montrera comment Bitcoin peut être utilisé pour d'importants paiements internationaux interentreprises liés à des biens physiques.

Miner des bitcoins

Jing est un étudiant en génie informatique à Shanghai. Il a construit une plate-forme «minière» pour exploiter le bitcoin en utilisant ses compétences en ingénierie pour compléter ses revenus. Cette histoire examinera la base "industrielle" de Bitcoin : l'équipement spécialisé utilisé pour sécuriser le réseau Bitcoin et émettre de nouvelles devises.

Chacune de ces histoires est basée sur de vraies personnes et de vraies industries utilisant actuellement Bitcoin pour créer de nouveaux marchés, de nouvelles industries et des solutions innovantes aux problèmes économiques mondiaux.

# Mise en route

Bitcoin est un protocole accessible à l'aide d'une application cliente qui parle le protocole. Un "portefeuille Bitcoin" est l'interface utilisateur la plus courante du système Bitcoin, tout comme un navigateur Web est l'interface utilisateur la plus courante pour le protocole HTTP. Il existe de nombreuses implémentations et marques de portefeuilles Bitcoin, tout comme il existe de nombreuses marques de navigateurs Web (par exemple, Chrome, Safari, Firefox et Internet Explorer). Et tout comme nous avons tous nos navigateurs préférés (Mozilla Firefox, Yé !) et nos méchants (Internet Explorer, Beurk !), les portefeuilles Bitcoin varient en termes de qualité, de performances, de sécurité, de confidentialité et de fiabilité. Il existe également une implémentation de référence du protocole Bitcoin qui inclut un portefeuille, connu sous le nom de "Satoshi Client" ou "Bitcoin Core", qui est dérivé de l'implémentation originale écrite par Satoshi Nakamoto.

## Choisir un portefeuille Bitcoin

Les portefeuilles Bitcoin sont l'une des applications les plus activement développées dans l'écosystème Bitcoin. La concurrence est intense et, bien qu'un nouveau portefeuille soit probablement en cours de développement, plusieurs portefeuilles de l'année dernière ne sont plus activement maintenus. De nombreux portefeuilles se concentrent sur des plates-formes spécifiques ou des utilisations spécifiques et certains sont plus adaptés aux débutants tandis que d'autres sont remplis de fonctionnalités pour les utilisateurs avancés. Le choix d'un portefeuille est très subjectif et dépend de l'utilisation et de l'expertise de l'utilisateur. Par conséquent, il serait inutile de recommander une marque ou un portefeuille spécifique. Cependant, nous pouvons classer les portefeuilles Bitcoin en fonction de leur plate-forme et de leur fonction et fournir des éclaircissements sur tous les différents types de portefeuilles qui existent. Mieux encore, déplacer des clés ou des valeurs d’amorçages (les "seeds") entre les portefeuilles Bitcoin est relativement facile, il vaut donc la peine d'essayer plusieurs portefeuilles différents jusqu'à ce que vous en trouviez un qui corresponde à vos besoins.

Les portefeuilles Bitcoin peuvent être classés comme suit, selon la plateforme :

Portefeuille de bureau

Un portefeuille de bureau a été le premier type de portefeuille Bitcoin créé comme implémentation de référence et de nombreux utilisateurs utilisent des portefeuilles de bureau pour les fonctionnalités, l'autonomie et le contrôle qu'ils offrent. L'exécution sur des systèmes d'exploitation à usage général tels que Windows et Mac OS présente cependant certains inconvénients en matière de sécurité, car ces plates-formes sont souvent non sécurisées et mal configurées.

Portefeuille mobile

Un portefeuille mobile est le type le plus courant de portefeuille Bitcoin. Fonctionnant sur des systèmes d'exploitation de téléphones intelligents tels qu'Apple iOS et Android, ces portefeuilles sont souvent un excellent choix pour les nouveaux utilisateurs. Beaucoup sont conçus pour être simples et faciles à utiliser, mais il existe également des portefeuilles mobiles complets pour les utilisateurs expérimentés.

Portefeuille Web

Les portefeuilles Web sont accessibles via un navigateur Web et stockent le portefeuille de l'utilisateur sur un serveur appartenant à un tiers. Ceci est similaire au webmail en ce sens qu'il repose entièrement sur un serveur tiers. Certains de ces services fonctionnent à l'aide d'un code côté client exécuté dans le navigateur web de l'utilisateur, qui garde le contrôle des clés Bitcoin entre les mains de l'utilisateur. La plupart, cependant, présentent un compromis en prenant le contrôle des clés Bitcoin des utilisateurs en échange d'une facilité d'utilisation. Il est déconseillé de stocker de grandes quantités de bitcoins sur des systèmes tiers.

Portefeuille matériel (hardware)

Les portefeuilles matériels sont des appareils qui exploitent un portefeuille Bitcoin autonome sécurisé sur du matériel à usage spécial. Ils se connectent généralement à un ordinateur de bureau ou à un appareil mobile via un câble USB ou une communication en champ proche (NFC) et fonctionnent avec un navigateur Web ou un logiciel associé. En gérant toutes les opérations liées au Bitcoin sur le matériel spécialisé, ces portefeuilles sont considérés comme très sécurisés et adaptés au stockage de grandes quantités de Bitcoin.

Une autre façon de catégoriser les portefeuilles bitcoin est leur degré d'autonomie et la façon dont ils interagissent avec le réseau Bitcoin :

Client de nœud complet

Un client complet, ou "nœud complet", est un client qui stocke l'historique complet des transactions Bitcoin (chaque transaction par chaque utilisateur, toujours), gère les portefeuilles des utilisateurs et peuvent initier des transactions directement sur le réseau Bitcoin. Un nœud complet gère tous les aspects du protocole et peut valider indépendamment l'ensemble de la chaîne de blocs et toute transaction. Un client à nœud complet consomme des ressources informatiques importantes (par exemple, plus de 125 Go de disque, 2 Go de RAM) mais offre une autonomie complète et une vérification indépendante des transactions.

Client léger

Un client léger, également connu sous le nom de client de vérification de paiement simplifié (SPV ou simplified-payment-verification), se connecte aux nœuds complets Bitcoin (mentionnés précédemment) pour l'accès aux informations de transaction Bitcoin, mais stocke le portefeuille de l'utilisateur localement et crée, valide et transmet indépendamment les transactions. Les clients légers interagissent directement avec le réseau Bitcoin, sans intermédiaire.

Client API tiers

Un client API tiers est un client qui interagit avec Bitcoin via un système tiers d'interfaces de programmation d'applications (API), plutôt qu'en se connectant directement au réseau Bitcoin. Le portefeuille peut être stocké par l'utilisateur ou par des serveurs tiers, mais toutes les transactions passent par un tiers.

En combinant ces catégorisations, de nombreux portefeuilles Bitcoin se répartissent en quelques groupes, les trois plus courants étant le client complet de bureau, le portefeuille léger mobile et le portefeuille Web tiers. Les frontières entre les différentes catégories sont souvent floues, car de nombreux portefeuilles fonctionnent sur plusieurs plates-formes et peuvent interagir avec le réseau de différentes manières.

Pour les besoins de ce livre, nous démontrerons l'utilisation d'une variété de clients Bitcoin téléchargeables, de l'implémentation de référence (Bitcoin Core) aux portefeuilles mobiles et Web. Certains des exemples nécessiteront l'utilisation de Bitcoin Core, qui, en plus d'être un client complet, expose également les API au portefeuille, au réseau et aux services de transaction. Si vous envisagez d'explorer les interfaces de programmation dans le système Bitcoin, vous devrez exécuter Bitcoin Core ou l'un des clients alternatifs.

## Démarrage rapide

Alice, que nous avons présenté dans [Utilisations, utilisateurs et histoires de Bitcoin](#user-stories), n'est pas une utilisatrice technique et n'a entendu parler de Bitcoin que récemment par son ami Joe. Lors d'une fête, Joe explique à nouveau avec enthousiasme Bitcoin à tout le monde et propose une démonstration. Intriguée, Alice demande comment elle peut démarrer avec Bitcoin. Joe dit qu'un portefeuille mobile est ce qu'il y a de mieux pour les nouveaux utilisateurs et il recommande quelques-uns de ses portefeuilles préférés. Alice télécharge "Bluewallet" (disponible pour iOS et Android) et l'installe sur son téléphone.

Quand Alice exécute son application de portefeuille pour la première fois, elle choisit l'option de créer un nouveau portefeuille Bitcoin et prend un moment **loin de Joe et de toutes les autres parties** pour écrire une phrase mnémonique secrète *dans l'ordre* sur un morceau de papier. Comme expliqué par le portefeuille mobile et par Joe plus tôt, la phrase mnémotechnique permet à Alice de restaurer son portefeuille au cas où elle perdrait son appareil mobile et lui donne accès à ses fonds sur un autre appareil. Après avoir créé son portefeuille et sécurisé sa phrase mnémonique, Alice peut appuyer sur son portefeuille pour voir son montant en bitcoins, l'historique des transactions, ainsi que deux boutons qui lui permettent de *recevoir* ou *envoyer* des bitcoins, illustrés dans [figure\_title](#bluewallet-welcome).

## Mots Mnémoniques

Un portefeuille Bitcoin moderne fournira une *phrase mnémonique* (aussi parfois appelée "valeur d’amorçage" ou "phrase de récupération") qu'Alice pourra sauvegarder. La phrase mnémonique se compose de 12 à 24 mots anglais, sélectionnés au hasard par le logiciel, et utilisés comme base pour les clés générées par le portefeuille. La phrase mnémotechnique peut être utilisée par Alice pour restaurer toutes les transactions et tous les fonds de son portefeuille en cas d'événement tel qu'un appareil mobile perdu, un bogue logiciel ou une corruption de la mémoire.

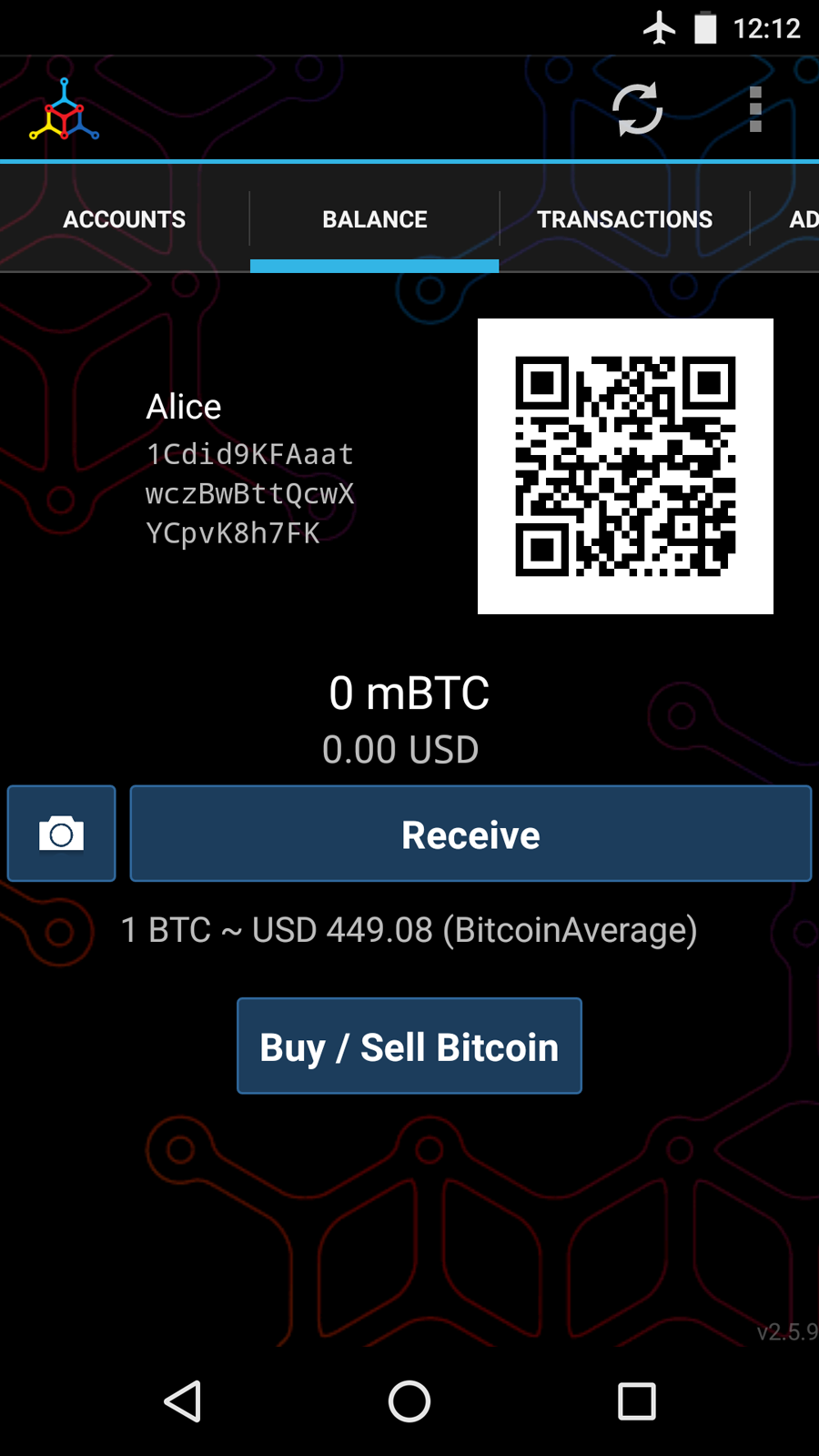
Le terme correct pour ces mots de sauvegarde est "phrase mnémonique". Nous évitons d'utiliser le terme "valeur d’amorçage" pour désigner une phrase mnémotechnique, car même si son utilisation est la même, son usage initial est différent.

## Stocker le mnémonique en toute sécurité

Alice doit faire attention à stocker la phrase mnémonique d'une manière qui équilibre la nécessité d'empêcher le vol et la perte accidentelle. Si elle ne le protège pas suffisamment, son mnémonique risque de se faire voler. Si elle le protège trop, son mnémonique risque d'être définitivement perdu. La méthode recommandée pour équilibrer correctement ces risques est d'écrire deux copies de la phrase mnémonique sur papier, avec chacun des mots numérotés selon l'ordre.

Une fois qu'Alice a enregistré la phrase mnémotechnique, elle doit prévoir de stocker chaque copie dans un endroit sécurisé séparé, tel qu'un tiroir de bureau verrouillé ou un coffre-fort ignifuge.

N'essayez jamais un système de sécurité "bricolage" qui s'écarte de quelque manière que ce soit de la recommandation des meilleures pratiques dans <<mnemonic-storage> >. Ne coupez pas votre mnémonique en deux, ne faites pas de captures d'écran, ne stockez pas sur des clés USB, des e-mails ou des lecteurs cloud, ne le cryptez pas ou n'essayez aucune autre méthode non standard. Vous ferez pencher la balance de manière à risquer une perte permanente ou un vol. De nombreuses personnes ont perdu de l'argent, non pas à cause d'un vol, mais parce qu'elles ont essayé une solution non standard sans avoir l'expertise pour équilibrer les risques encourus. La recommandation des meilleures pratiques est soigneusement équilibrée par des experts et adaptée à la grande majorité des utilisateurs.



Alice utilise l'écran de réception sur son portefeuille Bitcoin mobile Bluewallet et affiche son adresse à Bob au format de code QR

La vue principale du portefeuille affiche le montant en bitcoins, l'historique des transactions et les boutons *Receive* et *Send*. En outre, de nombreux portefeuilles offrent la possibilité d'acheter des Bitcoins directement via un échange ou un service similaire où vous pouvez offrir de l'argent fiduciaire en échange de cryptomonnaie, ce qui est fait par [simpara\_title](#bitcoin_price) et vendre à l'utilisateur du portefeuille à ce prix ou au-dessus. Le bouton *Buy Bitcoin* permettrait à Alice d'acheter des Bitcoins de cette manière.

Alice est maintenant prête à commencer à utiliser son nouveau portefeuille Bitcoin. Son application de portefeuille a généré aléatoirement une clé privée (décrite plus en détail dans [???](#private_keys)) qui sera utilisé pour dériver des adresses Bitcoin qui dirigent vers son portefeuille. À ce stade, ses adresses Bitcoin ne sont pas connues du réseau Bitcoin ou "enregistrées" auprès d'une quelconque partie du système Bitcoin. Ses adresses Bitcoin sont simplement des nombres aléatoires qui correspondent à sa clé privée qu'elle peut utiliser pour contrôler l'accès aux fonds. Les adresses sont générées indépendamment par son portefeuille sans référence ni inscription à aucun service. En fait, dans la plupart des portefeuilles, il n'y a aucune association entre une adresse Bitcoin et toute information d'identification externe, y compris l'identité de l'utilisateur. Jusqu'au moment où une adresse est référencée comme destinataire de la valeur dans une transaction publiée sur le grand livre bitcoin, l'adresse Bitcoin fait simplement partie du grand nombre d'adresses possibles valides en bitcoin. Ce n'est qu'une fois qu'une adresse a été associée à une transaction qu'elle fait partie des adresses connues du réseau.

Alice utilise le bouton *Receive*, qui affiche un code QR avec une adresse Bitcoin. Le code QR est le carré avec un motif de points noirs et blancs, servant de forme de code-barres contenant les mêmes informations dans un format pouvant être scanné par l'appareil photo du smartphone de Joe. À côté du code QR du portefeuille se trouve l'adresse Bitcoin qu'il encode, et Alice peut choisir d'envoyer manuellement son adresse à Joe en la copiant dans son presse-papiers d'un simple toucher. Il convient de noter que lorsque vous recevez des fonds sur un nouveau portefeuille mobile pour la première fois, de nombreux portefeuilles revérifieront souvent que vous avez bien sécurisé votre phrase mnémonique. Cela peut aller d'une simple invitation à demander à l'utilisateur de ressaisir manuellement la phrase.

Les adresses Bitcoin commencent par 1, 3 ou bc1. Comme les adresses e-mail, elles peuvent être partagées avec d'autres utilisateurs de bitcoins qui peuvent les utiliser pour envoyer des bitcoins directement dans votre portefeuille. Il n'y a rien de sensible, du point de vue de la sécurité, à propos de l'adresse Bitcoin. Il peut être affiché n'importe où sans risquer la sécurité du compte. Contrairement aux adresses e-mail, vous pouvez créer de nouvelles adresses aussi souvent que vous le souhaitez, qui dirigeront toutes les fonds vers votre portefeuille. En fait, de nombreux portefeuilles modernes créent automatiquement une nouvelle adresse pour chaque transaction afin de maximiser la confidentialité. Un portefeuille est simplement une collection d'adresses et les clés qui débloquent les fonds à l'intérieur.

==== Obtenir son premier Bitcoin

Il existe plusieurs façons pour Alice d'acquérir des bitcoins :

* Elle peut échanger une partie de sa monnaie nationale (par exemple USD) à un échange de crypto-monnaie
* Elle peut en acheter à un ami ou à une connaissance d'un Meetup Bitcoin, en échange d'argent
* Elle peut trouver un *guichet Bitcoin* dans sa région, qui agit comme un distributeur automatique, vendant des bitcoins contre de l'argent
* Elle peut offrir ses compétences ou un produit qu'elle vend et accepte le paiement en bitcoin
* Elle peut demander à son employeur ou à ses clients de la payer en bitcoin

Toutes ces méthodes présentent des degrés de difficulté variables et nombre d'entre elles impliquent le paiement de frais. Certaines institutions financières exigeront également qu'Alice fournisse des documents d'identification pour se conformer aux réglementations bancaires locales/aux pratiques de lutte contre le blanchiment d'argent (AML), un processus connu sous le nom de Know Your Customer (KYC). Cependant, avec toutes ces méthodes, Alice pourra recevoir des bitcoins.

L'un des avantages du bitcoin par rapport aux autres systèmes de paiement est que, lorsqu'il est utilisé correctement, il offre aux utilisateurs beaucoup plus d'intimité. Acquérir, détenir et dépenser des bitcoins ne vous oblige pas à divulguer des informations sensibles et personnellement identifiables à des tiers. Cependant, lorsque le bitcoin touche les systèmes traditionnels, tels que les échanges de devises, les réglementations nationales et internationales s'appliquent souvent. Afin d'échanger des bitcoins contre votre monnaie nationale, il vous sera souvent demandé de fournir une preuve d'identité et des informations bancaires. Les utilisateurs doivent savoir qu'une fois qu'une adresse Bitcoin est associée à une identité, toutes les transactions Bitcoin associées sont également faciles à identifier et à suivre. C'est l'une des raisons pour lesquelles de nombreux utilisateurs choisissent de maintenir des comptes d'échange dédiés non liés à leurs portefeuilles.

Alice a été initiée au bitcoin par un ami, elle a donc un moyen facile d'acquérir son premier bitcoin. Ensuite, nous verrons comment elle achète des bitcoins à son ami Joe et comment Joe envoie les bitcoins dans son portefeuille.

==== Trouver le prix actuel du Bitcoin

Avant qu'Alice puisse acheter du bitcoin à Joe, ils doivent s'entendre sur le *taux de change* entre le bitcoin et le dollar américain . Cela soulève une question courante pour ceux qui découvrent le bitcoin : "Qui fixe le prix du bitcoin ?" La réponse courte est que le prix est fixé par les marchés.

Le bitcoin, comme la plupart des autres devises, a un *cours du change flottant*. Cela signifie que la valeur du bitcoin vis-à-vis de toute autre devise fluctue en fonction de l'offre et de la demande sur les différents marchés où il est négocié. Par exemple, le « prix » du bitcoin en dollars américains est calculé sur chaque marché en fonction de la transaction la plus récente de bitcoin et de dollars américains. En tant que tel, le prix a tendance à fluctuer minutieusement plusieurs fois par seconde. Un service de tarification regroupera les prix de plusieurs marchés et calculera une moyenne pondérée en fonction du volume représentant le taux de change général du marché d'une paire de devises (par exemple, BTC/USD).

Il existe des centaines d'applications et de sites Web qui peuvent fournir le taux actuel du marché. Voici quelques-uns des plus populaires :

[Bitcoin Average](https://bitcoinaverage.com/)

Un site qui fournit une vue simple de la moyenne pondérée en fonction du volume pour chaque devise.

[CoinCap](https://coincap.io/)

Un service répertoriant la capitalisation boursière et les taux de change de centaines de crypto-monnaies, dont le bitcoin.

[Chicago Mercantile Exchange Bitcoin Reference Rate](https://bit.ly/cmebrr)

Un taux de référence qui peut être utilisé comme référence institutionnelle et contractuelle, fourni dans le cadre des flux de données d'investissement par le CME.

En plus de ces différents sites et applications, la plupart des portefeuilles bitcoin convertiront automatiquement les montants entre le bitcoin et d'autres devises. Joe utilisera son portefeuille pour convertir automatiquement le prix avant d'envoyer des bitcoins à Alice.

==== Envoi et réception de Bitcoin

Alice a décidé d'échanger 10 dollars américains contre des bitcoins, afin de ne pas risquer trop d'argent sur cette nouvelle technologie. Elle donne à Joe 10 $ en espèces, ouvre son application de portefeuille mobile Bluewallet et sélectionne Recevoir. Cela affiche un code QR avec la première adresse Bitcoin d'Alice.

Joe sélectionne ensuite *Envoyer* sur son portefeuille de smartphone Bluewallet et un écran contenant les entrées suivantes s'affiche :

* Le montant à envoyer, en bitcoin (BTC) ou sa devise locale (USD)
* Une adresse Bitcoin de destination
* Une note de transaction (description)
* Des frais de transaction

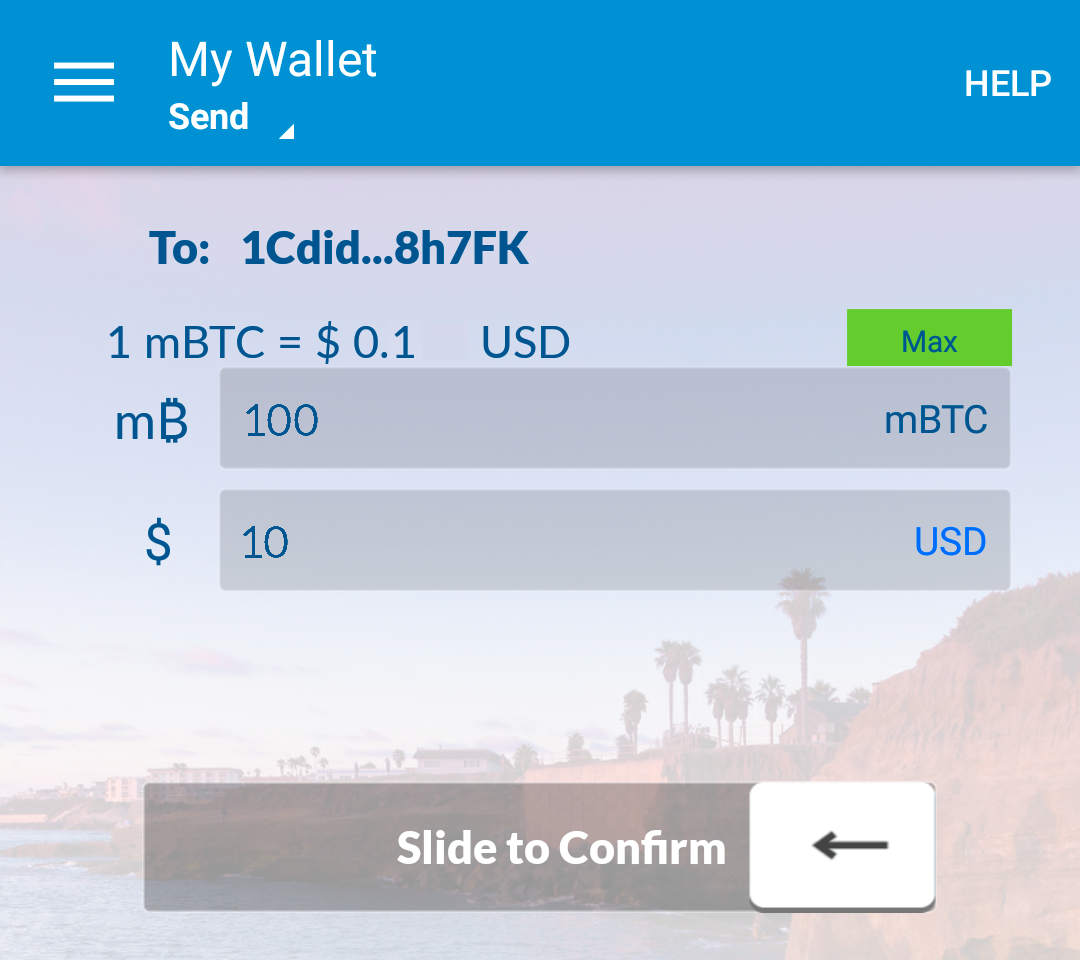
Dans le champ de saisie de l'adresse Bitcoin, il y a un petit bouton *Scan*. Cela permet à Joe de scanner le code QR avec l'appareil photo de son smartphone afin de ne pas avoir à taper l'adresse Bitcoin d'Alice, qui est assez longue et difficile à taper. Joe appuie sur le bouton *Scan* et il active la caméra du smartphone, scannant le code QR affiché sur le smartphone d'Alice.

Joe a maintenant l'adresse Bitcoin d'Alice définie comme destinataire. Joe saisit le montant de 10 dollars américains et son portefeuille le convertit en accédant au taux de change le plus récent à partir d'un service en ligne. Le taux de change à l'époque est de 100 dollars américains par bitcoin, donc 10 dollars américains valent 0,10 bitcoin (BTC) ou 100 millibitcoin (mBTC) comme le montre la capture d'écran du portefeuille de Joe (voir [figure\_title](#bluewallet-mobile-send)).

Dans la saisie de la note/description de la transaction, Joe saisit « Alice ». Il peut utiliser ce champ pour ajouter des informations concernant sa transaction pour référence future. Cette fonction est pour sa tenue de dossiers uniquement. La note de transaction sera stockée dans son portefeuille et seul Joe pourra la voir. Il ne sera pas envoyé à Alice, ni stocké sur la blockchain.

Il sélectionne également des frais de transaction pour sa transaction. Plus les frais de transaction sont élevés, plus vite sa transaction sera confirmée (incluse dans un bloc par un mineur). Il sélectionne les frais de transaction minimum possibles à ce moment-là (0 satoshis/octet).

Le prix du bitcoin a beaucoup changé au fil du temps, et une quantité incroyable depuis la rédaction de la première édition de ce livre. En mars 2021, une personne aurait besoin d'environ 54 000 USD pour acheter un bitcoin entier. De nombreux exemples dans ce livre font référence à des transactions passées réelles, lorsque le prix du bitcoin était beaucoup plus bas et que des transactions sans frais étaient encore possibles. Pensez à quel point Joe aurait été un ami généreux s'il avait conclu le même accord avec Alice aujourd'hui !



Écran d'envoi du portefeuille bitcoin mobile Bluewallet

À l'aide de Bluewallet, Joe vérifie soigneusement qu'il a saisi le bon montant, car il est sur le point de transmettre de l'argent et les erreurs sont irréversibles. Par souci de simplicité, nous supposerons que Joe ne paie aucun frais de transaction. L'objet et la fixation des frais de transaction sont traités dans les chapitres suivants. Après avoir revérifié l'adresse et le montant, il appuie sur *Envoyer* pour transmettre la transaction. Le portefeuille bitcoin mobile de Joe construit une transaction qui attribue 0,10 BTC à l'adresse fournie par Alice, puise les fonds dans le portefeuille de Joe et signe la transaction avec les clés privées de Joe. Cela indique au réseau Bitcoin que Joe a autorisé un transfert de valeur vers la nouvelle adresse d'Alice. Comme la transaction est transmise via le protocole pair à pair, elle se propage rapidement sur le réseau Bitcoin. En moins d'une seconde, la plupart des nœuds bien connectés du réseau reçoivent la transaction et voient l'adresse d'Alice pour la première fois.

Pendant ce temps, le portefeuille d'Alice "écoute" constamment les transactions publiées sur le réseau Bitcoin, à la recherche de celles qui correspondent aux adresses qu'il contient. Quelques secondes après que le portefeuille de Joe ait transmis la transaction, le portefeuille d'Alice indiquera qu'il reçoit 0,10 BTC.

Chaque bitcoin peut être subdivisé en 100 millions d'unités, chacune appelée "satoshi" (singulier) ou "satoshis" (pluriel). Nommé en l'honneur du créateur du bitcoin, le Satoshi est la plus petite unité de bitcoin, équivalente à 0,00000001 BTC.

**Confirmations**

Au début, le portefeuille d'Alice affichera la transaction de Joe comme "Non confirmée". Cela signifie que la transaction a été propagée au réseau mais n'a pas encore été enregistrée dans le registre des transactions bitcoin, connu sous le nom de blockchain ou la chaîne de blocs. Pour être confirmée, une transaction doit être incluse dans un bloc et ajoutée à la chaîne de blocs, ce qui se produit toutes les 10 minutes en moyenne. En termes financiers traditionnels, cela s'appelle *liquidation (ou clearing)*. Ce livre couvrira en détail la propagation, la validation et la liquidation (ou la confirmation) des transactions bitcoin.

Alice est maintenant l'heureuse propriétaire de 0,10 BTC qu'elle peut dépenser. Dans le chapitre suivant, nous examinerons son premier achat avec Bitcoin et examinerons plus en détail les technologies sous-jacentes de transaction et de propagation.

== Comment fonctionne Bitcoin

=== Transactions, blocs, minage et la chaîne de blocs

Le système Bitcoin, contrairement aux systèmes bancaires et de paiement traditionnels, est basé sur la confiance décentralisée. Au lieu d'une autorité centrale de confiance, dans le bitcoin, la confiance est obtenue en tant que propriété émergente à partir des interactions des différents participants au système Bitcoin. Dans ce chapitre, nous examinerons le bitcoin d'un niveau élevé en suivant une seule transaction via le système Bitcoin et en observant qu'elle devient "de confiance" et acceptée par le mécanisme de consensus distribué bitcoin et est finalement enregistrée sur la chaîne de blocs, le grand livre distribué de toutes transactions. Les chapitres suivants approfondiront la technologie derrière les transactions, le réseau et l'exploitation minière.

==== Présentation de Bitcoin

Dans le schéma d'ensemble présenté dans [figure\_title](#bitcoin-overview), nous voyons que le système Bitcoin se compose d'utilisateurs avec des portefeuilles contenant des clés, des transactions qui se propagent à travers le réseau et des mineurs qui produisent (par le calcul concurrentiel) le consensus de la chaîne de blocs, qui est le registre faisant autorité de toutes les transactions.

Chaque exemple de ce chapitre est basé sur une transaction réelle effectuée sur le réseau Bitcoin, simulant les interactions entre les utilisateurs (Joe, Alice, Bob et Gopesh) en envoyant des fonds à partir d'un portefeuille à un autre. Lors du suivi d'une transaction via le réseau Bitcoin jusqu'à la chaîne de blocs, nous utiliserons un site *explorateur de chaîne de blocs* pour visualiser chaque étape. Un explorateur de chaîne de blocs est une application Web qui fonctionne comme un moteur de recherche bitcoin, en ce sens qu'il vous permet de rechercher des adresses, des transactions et des blocs et de voir les relations et les flux entre eux.



Présentation de Bitcoin

Les explorateurs de blockchain populaires incluent :

* [Explorateur BlockCypher](https://live.blockcypher.com)
* [Blockchain.com](https://blockchain.com)
* [BitPay Insight](https://insight.bitpay.com)
* [Blockstream Explorer](https://blockstream.info)

Chacun d'eux a une fonction de recherche qui peut prendre une adresse Bitcoin, un hachage de transaction, un numéro de bloc ou un hachage de bloc et récupérer les informations correspondantes du réseau Bitcoin. Avec chaque exemple de transaction ou de bloc, nous fournirons une URL afin que vous puissiez la rechercher vous-même et l'étudier en détail.

==== Acheter une tasse de café

Alice, présentée dans le chapitre précédent, est une nouvelle utilisatrice qui vient d'acquérir son premier bitcoin. Dans [simpara\_title](#getting_first_bitcoin), Alice a rencontré son ami Joe pour échanger de l'argent contre du bitcoin. La transaction créée par Joe a financé le portefeuille d'Alice avec 0,10 BTC. Alice va maintenant faire sa première transaction de vente au détail en achetant une tasse de café au café de Bob à Palo Alto, en Californie.

Bob's Cafe a récemment commencé à accepter les paiements en bitcoin en ajoutant une option bitcoin à son système de point de vente. Les prix au Bob's Cafe sont indiqués dans la devise locale (dollars américains), mais à la caisse, les clients ont la possibilité de payer en dollars ou en bitcoins. Alice passe sa commande pour une tasse de café et Bob l'inscrit dans le registre, comme il le fait pour toutes les transactions. Le système de point de vente convertit automatiquement le prix total des dollars américains en bitcoins au taux du marché en vigueur et affiche le prix dans les deux devises :

Total:  
1,50 USD  
0,015 BTC

Bob dit : "C'est un dollar cinquante, ou quinze millibits."

Le système de point de vente de Bob créera également automatiquement un code QR spécial contenant une *demande de paiement* (voir [figure\_title](#payment-request-QR)).

Contrairement à un code QR qui contient simplement une adresse Bitcoin de destination, une demande de paiement est une URL encodée QR qui contient une adresse de destination, un montant de paiement et une description générique telle que "Bob's Cafe". Cela permet à une application de portefeuille bitcoin de pré-remplir les informations utilisées pour envoyer le paiement tout en affichant une description lisible par l'utilisateur à l'utilisateur. Vous pouvez scanner le code QR avec une application de portefeuille bitcoin pour voir ce qu'Alice verrait.



Code QR de la demande de paiement

Essayez de scanner ceci avec votre portefeuille pour voir l'adresse et le montant, mais N'ENVOYEZ PAS D'ARGENT.

**Le QR code de demande de paiement encode l'URL suivante, définie dans BIP-21 :**

bitcoin:1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA?  
amount=0.015&  
label=Bob%27s%20Cafe&  
message=Acheter%20chez%20Bob%27s%20Café  
  
Composants de l&#39;URL  
  
Une adresse Bitcoin : &quot;1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA&quot;  
Le montant du paiement: &quot;0,015&quot;  
Une étiquette pour l&#39;adresse du destinataire : « Bob&#39;s Cafe »  
Une description du paiement : &quot;Acheter au Bob&#39;s Cafe&quot;

Alice utilise son smartphone pour scanner le code-barres affiché. Son smartphone affiche un paiement de 0,0150 BTC à Bob&#39;s Cafe et elle sélectionne Envoyer pour autoriser le paiement. En quelques secondes (environ le même temps qu'une autorisation de carte de crédit), Bob voit la transaction sur le registre, complétant la transaction.

Dans les sections suivantes, nous examinerons cette transaction plus en détail. Nous verrons comment le portefeuille d'Alice l'a construit, comment il s'est propagé sur le réseau, comment il a été vérifié et enfin, comment Bob peut dépenser ce montant lors de transactions ultérieures.

Le réseau Bitcoin peut effectuer des transactions en valeurs fractionnaires, par exemple à partir de millibitcoin (1/1000e de bitcoin ) jusqu'à 1/100 000 000e de bitcoin, connu sous le nom de satoshi. Tout au long de ce livre, nous utiliserons le terme "bitcoin" pour désigner toute quantité de monnaie bitcoin, de la plus petite unité (1 satoshi) au nombre total (21 000 000) de tous les bitcoins qui seront extraits.

Vous pouvez examiner la transaction d'Alice avec Bob's Cafe sur la blockchain en utilisant un site d'exploration de blocs ([example\_title](#view_alice_transaction)):

https://www.blockchain.com/btc/tx/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2

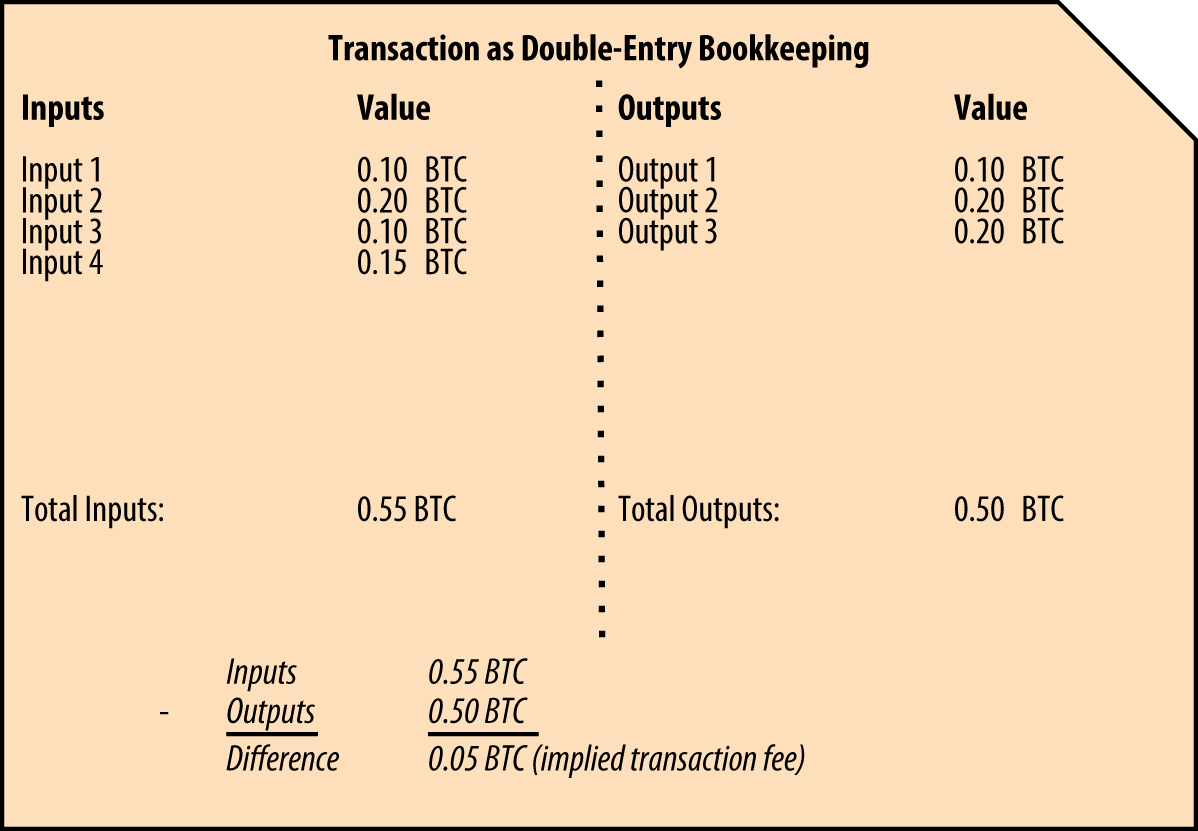
=== Transactions en bitcoins

En termes simples, une transaction indique au réseau que le propriétaire d'une valeur bitcoin a autorisé le transfert de cette valeur à un autre propriétaire. Le nouveau propriétaire peut désormais dépenser le bitcoin en créant une autre transaction qui autorise le transfert à un autre propriétaire, et ainsi de suite, dans une chaîne de propriété.

==== Entrées et sorties de transactions

Les transactions sont comme des lignes dans un livre comptable en partie double. Chaque transaction contient une ou plusieurs "entrées", qui sont comme des débits sur un compte bitcoin. De l'autre côté de la transaction, il y a une ou plusieurs "sorties", qui sont comme des crédits ajoutés à un compte bitcoin. Les entrées et les sorties (débits et crédits) ne totalisent pas nécessairement le même montant. Au lieu de cela, les sorties s'additionnent légèrement moins que les entrées et la différence représente des *frais de transaction* implicites, qui sont un petit paiement perçu par le mineur qui inclut la transaction dans le grand livre. Une transaction bitcoin s'affiche sous la forme d'une écriture comptable dans [figure\_title](#transaction-double-entry).

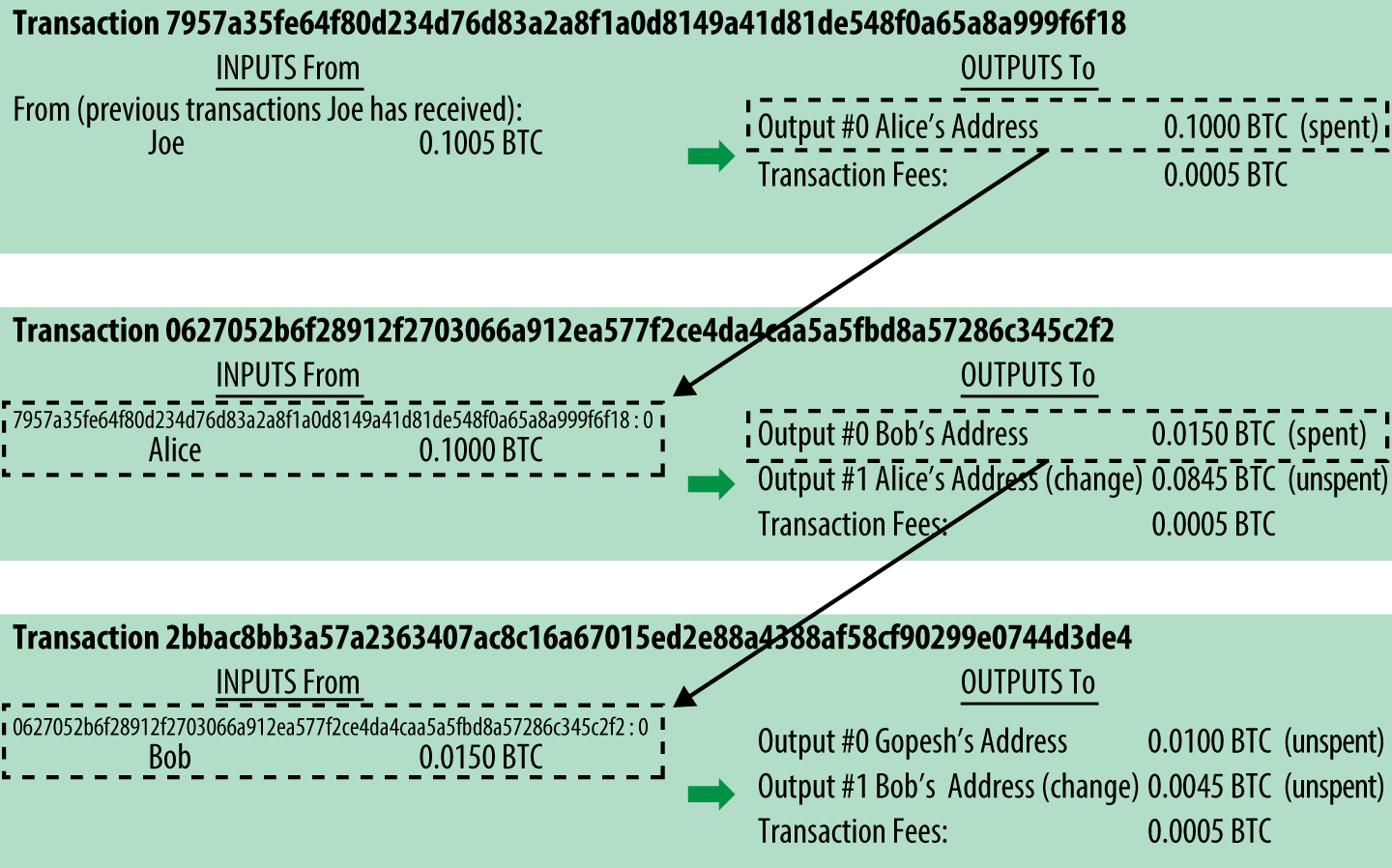
La transaction contient également une preuve de propriété pour chaque montant de bitcoin (entrées) dont la valeur est dépensée, sous la forme d'une signature numérique du propriétaire, qui peut être validée indépendamment par n'importe qui. En termes de bitcoin, "dépenser" consiste à signer une transaction qui transfère la valeur d'une transaction précédente à un nouveau propriétaire identifié par une adresse Bitcoin.



Transaction comptable en double parties

==== Chaînes de transaction

Le paiement d'Alice à Bob's Cafe utilise la sortie d'une transaction précédente comme entrée. Dans le chapitre précédent, Alice a reçu des bitcoins de son ami Joe en échange d'argent. Cette transaction a créé une valeur bitcoin verrouillée par la clé d'Alice. Sa nouvelle transaction avec Bob's Cafe fait référence à la transaction précédente comme entrée et crée de nouvelles sorties pour payer la tasse de café et recevoir la monnaie. Les transactions forment une chaîne, où les entrées de la dernière transaction correspondent aux sorties des transactions précédentes. La clé d'Alice fournit la signature qui déverrouille ces sorties de transaction précédentes, prouvant ainsi au réseau Bitcoin qu'elle possède les fonds. Elle joint le paiement du café à l'adresse de Bob, "encombrant" ainsi cette sortie avec l'exigence que Bob produise une signature afin de dépenser ce montant. Cela représente un transfert de valeur entre Alice et Bob. Cette chaîne de transactions, de Joe à Alice à Bob, est illustrée dans [figure\_title](#blockchain-mnemonic).



Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante

==== Faire du change (ou de la petite monnaie)

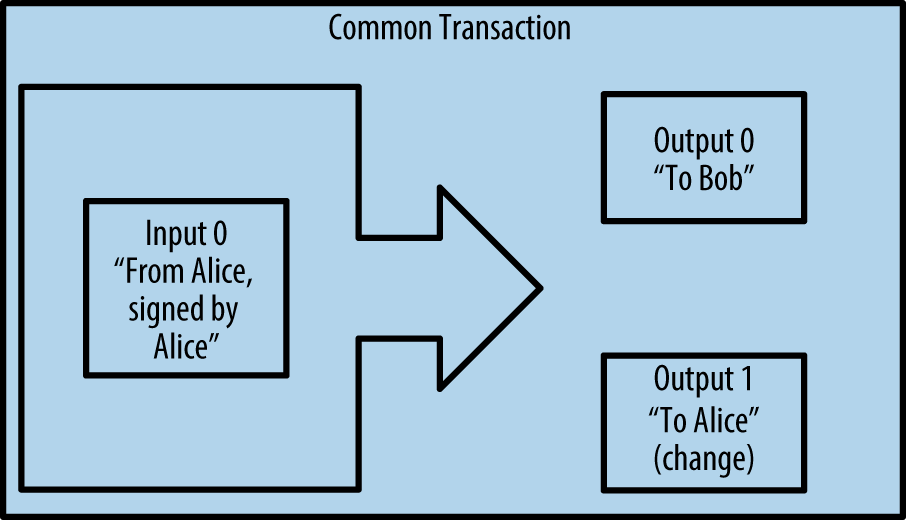
De nombreuses transactions bitcoin incluront des sorties qui référencent à la fois une adresse du nouveau propriétaire et une adresse du propriétaire actuel, appelée *adresse de change*. En effet, les entrées de transaction, comme les billets de banque, ne peuvent pas être divisées. Si vous achetez un article de 5 dollars américains dans un magasin mais que vous utilisez un billet de 20 dollars américains pour payer l'article, vous vous attendez à recevoir 15 dollars américains en monnaie. Le même concept s'applique aux entrées de transaction bitcoin. Si vous avez acheté un article qui coûte 5 bitcoins mais que vous n'aviez qu'une entrée de 20 bitcoins à utiliser, votre portefeuille créerait une seule transaction qui enverrait deux sorties, une sortie de 5 bitcoins au propriétaire du magasin et une sortie de 15 bitcoins à vous-même comme changement (moins les frais de transaction applicables). Il est important de noter que l'adresse de change ne doit pas nécessairement être la même que celle de l'entrée et, pour des raisons de confidentialité, il s'agit souvent d'une nouvelle adresse du portefeuille du propriétaire.

Différents portefeuilles peuvent utiliser différentes stratégies lors de l'agrégation des entrées pour effectuer un paiement demandé par l'utilisateur. Ils peuvent regrouper de nombreux petits intrants ou en utiliser un égal ou supérieur au paiement souhaité. À moins que le portefeuille ne puisse agréger les entrées de manière à correspondre exactement au paiement souhaité plus les frais de transaction, le portefeuille devra générer du change. C'est très similaire à la façon dont les gens gèrent l'argent liquide. Si vous utilisez toujours le plus gros billet dans votre poche, vous vous retrouverez avec une poche pleine de monnaie. Si vous n'utilisez que la petite monnaie, vous n'aurez toujours que de gros billets. Les gens trouvent inconsciemment un équilibre entre ces deux extrêmes, et les développeurs de portefeuilles Bitcoin s'efforcent de programmer cet équilibre.

En résumé, les *transactions* déplacent des valeurs des *entrées de transaction* aux *sorties de transaction*. Une entrée est une référence à la sortie d'une transaction précédente, indiquant d'où provient la valeur. Une transaction comprend généralement une sortie qui dirige une valeur spécifique vers l'adresse Bitcoin d'un nouveau propriétaire et une sortie de modification vers le propriétaire d'origine. Les sorties d'une transaction peuvent être utilisées comme entrées dans une nouvelle transaction, créant ainsi une chaîne de propriété lorsque la valeur est déplacée d'un propriétaire à l'autre (voir [figure\_title](#blockchain-mnemonic)).

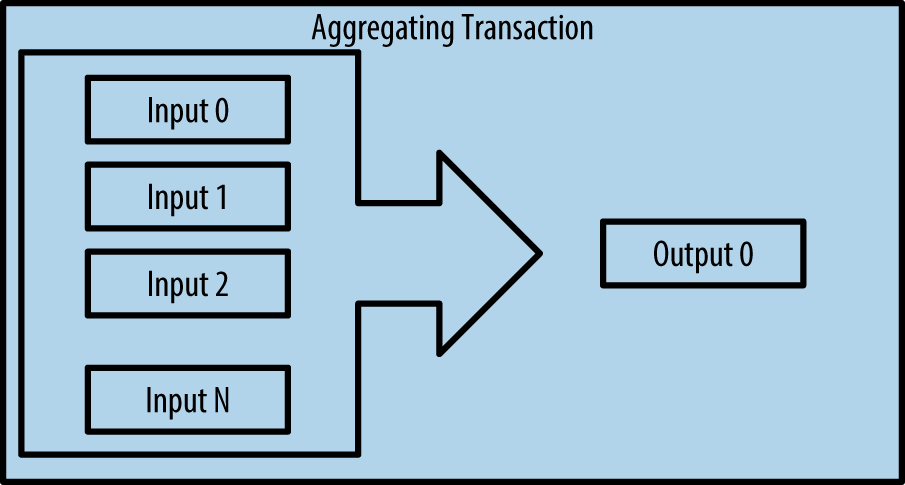
==== Formulaires de transaction communs

La forme de transaction la plus courante est un simple paiement d'une adresse à une autre, qui comprend souvent une « monnaie » rendue au propriétaire d'origine. Ce type de transaction a une entrée et deux sorties et est affiché dans [figure\_title](#transaction-common).



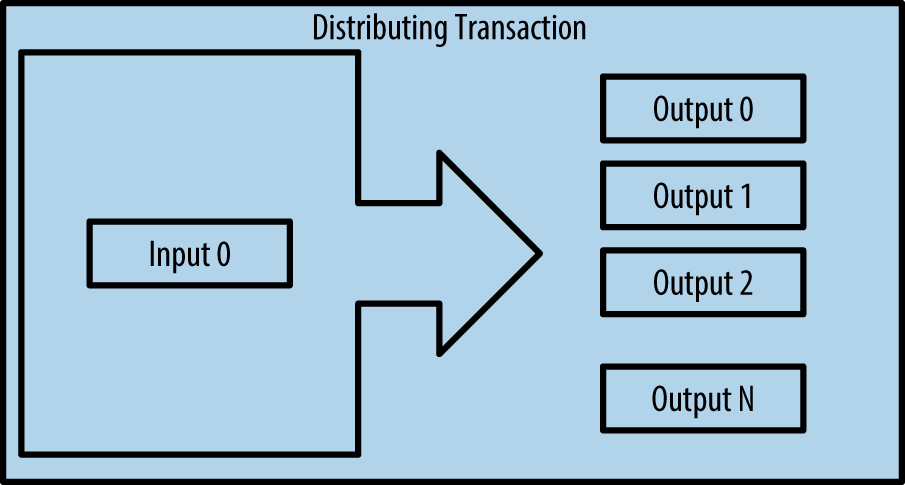
Transaction la plus courante

Une autre forme courante de transaction est celle qui agrège plusieurs entrées en une seule sortie (voir <<transaction-aggregating> >). Cela représente l'équivalent réel de l'échange d'une pile de pièces et de billets de banque contre un seul billet plus gros. Des transactions comme celles-ci sont parfois générées par des applications de portefeuille pour nettoyer de nombreux petits montants reçus en monnaie de paiement.



Transaction agrégeant des fonds

Enfin, une autre forme de transaction que l'on voit souvent sur le grand livre bitcoin est une transaction par lots, qui distribue une entrée à plusieurs sorties représentant plusieurs destinataires, une technique appelée "transactions par lot" (voir [figure\_title](#transaction-distributing)). Étant donné que ce type de transaction est utile pour économiser sur les frais de transaction, il est couramment utilisé par les entités commerciales pour distribuer des fonds, par exemple lorsqu'une entreprise traite les paiements de paie à plusieurs employés ou lorsqu'un échange de bitcoins traite les retraits de plusieurs clients en un seul. transaction.



Transaction distribuant des fonds

=== Construction d'une transaction

L'application de portefeuille d'Alice contient toute la logique pour sélectionner les entrées et les sorties appropriées pour construire une transaction à la spécification d'Alice. Alice n'a qu'à spécifier une destination et un montant, et le reste se passe dans l'application portefeuille sans qu'elle ne voie les détails. Il est important de noter qu'une application de portefeuille peut créer des transactions même si elle est complètement hors ligne. Comme écrire un chèque à la maison et l'envoyer plus tard à la banque dans une enveloppe, la transaction n'a pas besoin d'être construite et signée tout en étant connectée au réseau Bitcoin.

==== Obtenir les bonnes entrées

L'application de portefeuille d'Alice devra d'abord trouver des entrées qui peuvent payer le montant qu'elle veut envoyer à Bob. La plupart des portefeuilles gardent une trace de toutes les sorties disponibles appartenant aux adresses du portefeuille. Par conséquent, le portefeuille d'Alice contiendrait une copie de la sortie de transaction de la transaction de Joe, qui a été créée en échange d'argent (voir [simpara\_title](#getting_first_bitcoin)). Une application de portefeuille bitcoin qui s'exécute en tant que client à nœud complet contient en fait une copie de chaque sortie non dépensée de chaque transaction dans la chaîne de blocs. Cela permet à un portefeuille de construire des entrées de transaction ainsi que de vérifier rapidement les transactions entrantes comme ayant des entrées correctes. Cependant, comme un client à nœud complet occupe beaucoup d'espace disque, la plupart des portefeuilles d'utilisateurs exécutent des clients "légers" qui ne suivent que les sorties non dépensées de l'utilisateur.

Si l'application de portefeuille ne conserve pas une copie des sorties de transaction non dépensées, elle peut interroger le réseau Bitcoin pour récupérer ces informations à l'aide d'une variété d'API disponibles par différents fournisseurs ou en demandant à un nœud complet via l'aide d'un appel d'interface de programmation d'application (API). [example\_title](#example_2-2) montre une requête API, construite comme une commande HTTP GET vers une URL spécifique. Cette URL renverra toutes les sorties de transaction non dépensées pour une adresse, donnant à toute application les informations dont elle a besoin pour construire des entrées de transaction pour les dépenses. Nous utilisons le simple client HTTP en ligne de commande *cURL* pour récupérer la réponse.

$ 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 réponse en [example\_title](#example_2-2) montre une sortie non dépensée (une qui n'a pas encore été échangée) sous la propriété de l'adresse d'Alice 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK. La réponse inclut la référence à la transaction dans laquelle cette sortie non dépensée est contenue (le paiement de Joe) et sa valeur en satoshis, à 10 millions, équivalent à 0,10 bitcoin. Avec ces informations, l'application de portefeuille d'Alice peut construire une transaction pour transférer cette valeur aux nouvelles adresses de propriétaire.

Voir la [transaction de Joe à Alice](https://www.blockchain.com/btc/tx/7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18).

Comme vous pouvez le voir, le portefeuille d'Alice contient suffisamment de bitcoins en une seule sortie non dépensée pour payer la tasse de café. Si cela n'avait pas été le cas, l'application de portefeuille d'Alice aurait peut-être dû "fouiller" dans une pile de petites sorties non dépensées, comme prendre des pièces dans un sac à main jusqu'à ce qu'elle puisse en trouver assez pour payer le café. Dans les deux cas, il peut être nécessaire de récupérer de la monnaie, ce que nous verrons dans la section suivante, car l'application de portefeuille crée les sorties de transaction (paiements).

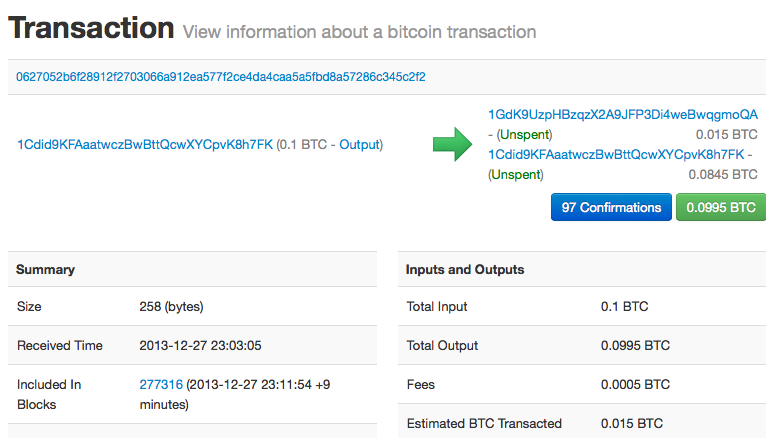
==== Création des sorties

Une sortie de transaction est créée sous la forme d'un script qui crée une charge sur la valeur et ne peut être rachetée que par l'introduction d'une solution au script. En termes plus simples, la sortie de transaction d'Alice contiendra un script qui dit quelque chose comme "Cette sortie est payable à quiconque peut présenter une signature à partir de la clé correspondant à l'adresse de Bob." Étant donné que seul Bob possède le portefeuille avec les clés correspondant à cette adresse, seul le portefeuille de Bob peut présenter une telle signature pour racheter cette sortie. Alice « encombrera » donc la valeur de sortie avec une demande de signature de Bob.

Cette transaction comprendra également une deuxième sortie, car les fonds d'Alice se présentent sous la forme d'une sortie à 0,10 BTC, trop d'argent pour la tasse de café à 0,015 BTC. Alice aura besoin de 0,085 BTC en monnaie. Le paiement de monnaie d'Alice est créé par le portefeuille d'Alice en tant que sortie dans la même transaction que le paiement à Bob. Essentiellement, le portefeuille d'Alice divise ses fonds en deux paiements : un à Bob et un à elle-même. Elle peut ensuite utiliser (dépenser) la sortie de monnaie dans une transaction ultérieure.

Enfin, pour que la transaction soit traitée par le réseau en temps opportun, l'application de portefeuille d'Alice ajoutera une somme modique. Ce n'est pas explicite dans la transaction ; il est impliqué par la différence entre les entrées et les sorties. Si au lieu de prendre 0,085 en monnaie, Alice ne crée que 0,0845 comme deuxième sortie, il restera 0,0005 BTC (un demi-millibitcoin). Le 0,10 BTC de l'entrée n'est pas entièrement dépensé avec les deux sorties, car elles totaliseront moins de 0,10. La différence qui en résulte est le *frais de transaction* qui est perçu par le mineur en tant que frais de validation et d'inclusion de la transaction dans un bloc à enregistrer sur la chaîne de blocs.

La transaction résultante peut être vue à l'aide d'une application Web d'exploration de chaîne de blocs, comme indiqué dans [figure\_title](#transaction-alice).



Transaction d'Alice au Bob's Cafe

Voir la [transaction d'Alice à Bob's Cafe](https://www.blockchain.com/btc/tx/0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2).

==== Ajout de la transaction au grand livre

La transaction créée par l'application du portefeuille d'Alice fait 258 octets et contient tout le nécessaire pour confirmer la propriété des fonds et attribuer à de nouveaux propriétaires. Désormais, la transaction doit être transmise au réseau Bitcoin où elle fera partie de la chaîne de blocs. Dans la section suivante, nous verrons comment une transaction devient partie intégrante d'un nouveau bloc et comment le bloc est « miné ». Enfin, nous verrons comment le nouveau bloc, une fois ajouté à la chaîne de blocs, est de plus en plus approuvé par le réseau au fur et à mesure que de nouveaux blocs sont ajoutés.

===== Transmission de la transaction

La transaction contient toutes les informations nécessaires au traitement, et ce peu importe comment et où elle est transmise au réseau Bitcoin. Le réseau Bitcoin est un réseau pair à pair, chaque client Bitcoin participant en se connectant à plusieurs autres clients Bitcoin. Le but du réseau Bitcoin est de propager les transactions et les blocages à tous les participants.

===== Comment ça se propage

Tout système, tel qu'un serveur, une application de bureau ou un portefeuille, qui participe au réseau Bitcoin en "parlant", le protocole Bitcoin est appelé un *nœud Bitcoin*. L'application de portefeuille d'Alice peut envoyer la nouvelle transaction à n'importe quel nœud Bitcoin auquel elle est connectée via n'importe quel type de connexion : filaire, Wi-Fi, mobile, etc. Son portefeuille Bitcoin n'a pas besoin d'être connecté directement au portefeuille Bitcoin de Bob et elle n'a pas à nécessairement utiliser la connexion Internet offerte par le café, bien que ces deux options soient également possibles. Tout nœud Bitcoin qui reçoit une transaction valide qu'il n'a pas vue auparavant la transmettra immédiatement à tous les autres nœuds auxquels il est connecté , une technique de propagation connue sous le nom d'*inondation*. Ainsi, la transaction se propage (en inondant ou distribuant) rapidement à travers le réseau pair à pair, atteignant un grand pourcentage de nœuds en quelques secondes.

===== Le point de vue de Bob

Si l'application de portefeuille bitcoin de Bob est directement connectée à l'application de portefeuille d'Alice, l'application de portefeuille de Bob peut être le premier nœud à recevoir la transaction. Cependant, même si le portefeuille d'Alice envoie la transaction via d'autres nœuds, elle atteindra le portefeuille de Bob en quelques secondes. Le portefeuille de Bob identifiera immédiatement la transaction d'Alice comme un paiement entrant car il contient des sorties remboursables par les clés de Bob. L'application de portefeuille de Bob peut également vérifier de manière indépendante que la transaction est bien formée, utilise des sorties précédemment non dépensées et contient des frais de transaction suffisants pour être inclus dans le bloc suivant. À ce stade, Bob peut supposer, avec peu de risques, que la transaction sera bientôt incluse dans un bloc et confirmée.

Une idée fausse courante à propos des transactions bitcoin est qu'elles doivent être "confirmées" en attendant 10 minutes pour un nouveau bloc, ou jusqu'à 60 minutes pour six confirmations complètes. Bien que les confirmations garantissent que la transaction a été acceptée par l'ensemble du réseau, un tel délai n'est pas nécessaire pour les articles de petite valeur comme une tasse de café. Un commerçant peut accepter une transaction valide de petite valeur sans confirmation, sans plus de risque qu'un paiement par carte de crédit effectué sans pièce d'identité ni signature, comme les commerçants l'acceptent couramment aujourd'hui.("", startref="Tconstruct02" )

=== Extraction de bitcoins

La transaction d'Alice est maintenant propagé sur le réseau Bitcoin. Il ne fait pas partie de la *chaîne de blocs* tant qu'il n'est pas vérifié et inclus dans un bloc par un processus appelé *minage*. Voir [simpara\_title](#mining) pour une explication détaillée.

Le système de confiance Bitcoin est basé sur le calcul. Les transactions sont regroupées en *blocs*, qui nécessitent une énorme quantité de calculs pour prouver, mais seulement une petite quantité de calculs pour vérifier comme prouvé. Le processus de minage sert à deux fins dans le bitcoin :

* Les nœuds de minage valident toutes les transactions en se référant au *consensus des règles de bitcoin*. Par conséquent, le minage assure la sécurité des transactions bitcoin en rejetant les transactions invalides ou malformées.
* L'exploitation minière crée de nouveaux bitcoins dans chaque bloc, presque comme une banque centrale imprimant de la nouvelle monnaie. La quantité de bitcoin créée par bloc est limitée et diminue avec le temps, suivant un calendrier d'émission fixe.

L'exploitation minière atteint un bon équilibre entre le coût et la récompense. L'exploitation minière utilise l'électricité pour résoudre un problème mathématique. Un mineur qui réussit recevra une *récompense* sous la forme de nouveaux bitcoins et de frais de transaction. Cependant, la récompense ne sera perçue que si le mineur a correctement validé toutes les transactions, à la satisfaction des règles de *consensus*. Cet équilibre délicat assure la sécurité du bitcoin sans autorité centrale.

Une bonne façon de décrire l'exploitation minière est comme un jeu compétitif géant de sudoku qui se réinitialise chaque fois que quelqu'un trouve une solution et dont la difficulté s'ajuste automatiquement de sorte qu'il faut environ 10 minutes pour trouver une solution. Imaginez un puzzle sudoku géant, de plusieurs milliers de lignes et de colonnes. Si je vous montre un puzzle terminé, vous pouvez le vérifier assez rapidement. Cependant, si le puzzle a quelques cases remplies et que les autres sont vides, cela demande beaucoup de travail à résoudre ! La difficulté du sudoku peut être ajustée en modifiant sa taille (plus ou moins de lignes et de colonnes), mais elle peut toujours être vérifiée assez facilement même si elle est très grande. Le "casse-tête" utilisé dans le bitcoin est basé sur un hachage cryptographique et présente des caractéristiques similaires : il est asymétriquement difficile à résoudre mais facile à vérifier, et sa difficulté peut être ajustée.

In [Utilisations, utilisateurs et histoires de Bitcoin](#user-stories), nous avons présenté Jing est un entrepreneur à Shanghai. Jing dirige une *ferme minière*, qui est une entreprise qui gère des milliers d'ordinateurs miniers spécialisés, en compétition pour la récompense. Toutes les 10 minutes environ, les ordinateurs miniers de Jing rivalisent avec des milliers de systèmes similaires dans une course mondiale pour trouver une solution à un bloc de transactions. Trouver une telle solution, la soi-disant *Proof-of-Work (ou Preuve de travail)* (PoW), nécessite des quadrillions d'opérations de hachage par seconde sur l'ensemble du réseau Bitcoin. L'algorithme de preuve de travail implique de hacher à plusieurs reprises l'en-tête du bloc et un nombre aléatoire avec l'algorithme cryptographique SHA256 jusqu'à ce qu'une solution correspondant à un modèle prédéterminé émerge. Le premier mineur à trouver une telle solution remporte la compétition et publie ce bloc dans la chaîne de blocs.

Jing a commencé l'exploitation minière en 2010 en utilisant un ordinateur de bureau très rapide pour trouver une preuve de travail appropriée pour les nouveaux blocs. Au fur et à mesure que de plus en plus de mineurs rejoignaient le réseau Bitcoin, la difficulté du problème augmentait rapidement. Bientôt, Jing et d'autres mineurs sont passés à du matériel plus spécialisé, avec des unités de traitement graphique (GPU) dédiées de haute de gamme, souvent utilisées dans les ordinateurs de bureau ou les consoles de jeu. Au moment d'écrire ces lignes, la difficulté est si élevée qu'il n'est rentable de miner qu'avec des circuits intégrés à application spécifique (ASIC), essentiellement des centaines d'algorithmes de minage imprimé en matériel, fonctionnant en parallèle sur une seule puce de silicium. La société de Jing participe également à un *groupe ou bassin de mineurs*, qui, tout comme un groupe de loterie, permet à plusieurs participants de partager leurs efforts et leurs récompenses. La société de Jing gère désormais un entrepôt contenant des milliers de mineurs ASIC pour extraire du bitcoin 24 heures sur 24. L'entreprise paie ses frais d'électricité en vendant le bitcoin qu'elle est capable de générer à partir de l'exploitation minière, créant ainsi des revenus à partir des bénéfices.

=== Transactions minières en blocs

De nouvelles transactions affluent constamment sur le réseau à partir des portefeuilles des utilisateurs et d'autres applications. Comme ceux-ci sont vus par les nœuds du réseau Bitcoin, ils sont ajoutés à un bassin temporaire de transactions non vérifiées maintenues par chaque nœud. Lorsque les mineurs construisent un nouveau bloc, ils ajoutent des transactions non vérifiées de ce bassin au nouveau bloc, puis tentent de prouver la validité de ce nouveau bloc, avec l'algorithme de minage (la preuve de travail). Le processus d'extraction est expliqué en détail dans [simpara\_title](#mining).

Les transactions sont ajoutées au nouveau bloc, classées par ordre de priorité par les transactions les plus payantes en premier et quelques autres critères. Chaque mineur commence le processus d'extraction d'un nouveau bloc de transactions dès qu'il reçoit le bloc précédent du réseau, sachant qu'il a perdu le tour de compétition précédent. Ils créent immédiatement un nouveau bloc, le remplissent avec les transactions et l'empreinte numérique du bloc précédent, et commencent à calculer la preuve de travail pour le nouveau bloc. Chaque mineur inclut une transaction spéciale dans son bloc, une transaction qui paie à sa propre adresse Bitcoin la récompense du bloc (actuellement 6,25 bitcoins nouvellement créés) plus la somme des frais de transaction de toutes les transactions incluses dans le bloc. S'ils trouvent une solution qui rend ce bloc valide, ils "gagnent" cette récompense car leur bloc réussi est ajouté à la chaîne de blocs mondiale et la transaction de récompense qu'ils ont incluse devient dépensable. Jing, qui participe à un groupe de minage, a mis en place son logiciel pour créer de nouveaux blocs qui attribuent la récompense à une adresse de pool. À partir de là, une part de la récompense est distribuée à Jing et aux autres mineurs proportionnellement à la quantité de travail qu'ils ont apportée lors du dernier tour.

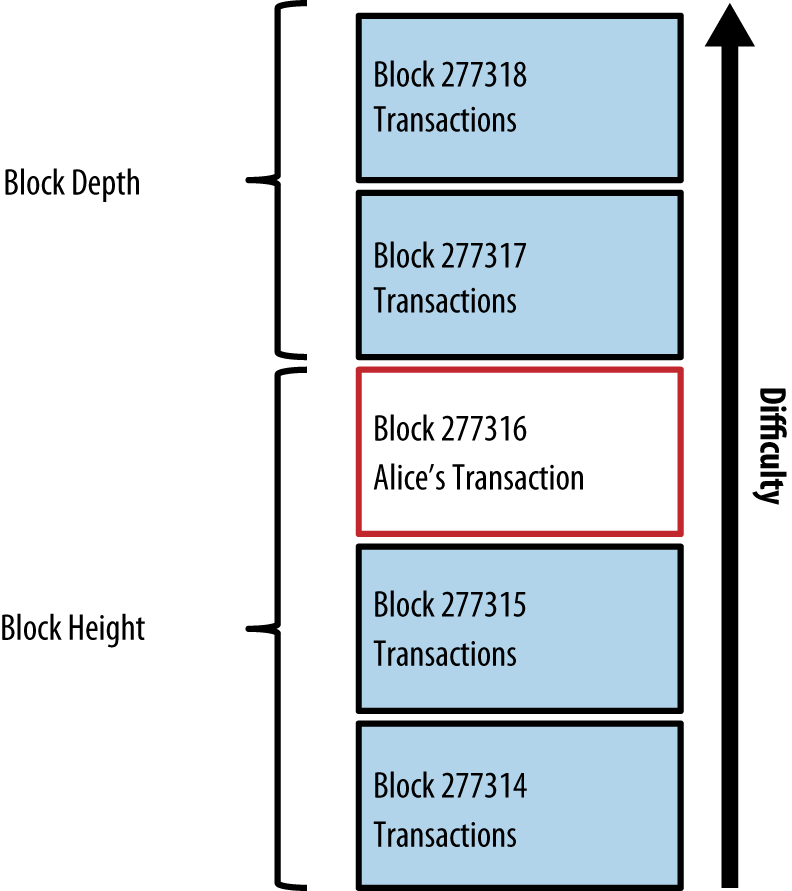
La transaction d'Alice a été récupérée par le réseau et incluse dans le groupe de transactions non vérifiées. Une fois validé par le logiciel de minage, il a été inclus dans un nouveau bloc, appelé *bloc candidat*, généré par le pool de minage de Jing. Tous les mineurs participant à ce pool de minage commencent immédiatement à calculer la preuve de travail pour le bloc candidat. Environ cinq minutes après la première transmission de la transaction par le portefeuille d'Alice, l'un des mineurs ASIC de Jing a trouvé une solution pour le bloc candidat et l'a annoncée au réseau. Une fois que les autres mineurs ont validé le bloc gagnant, ils ont commencé la course pour générer le bloc suivant.

Le bloc gagnant de Jing est devenu une partie de la blockchain en tant que bloc # 277316, contenant 419 transactions, y compris la transaction d'Alice. Le bloc contenant la transaction d'Alice est compté comme une "confirmation" de cette transaction.

Vous pouvez voir le bloc qui inclut [Transaction d’Alice](https://www.blockchain.com/btc/block/277316).

Environ 20 minutes plus tard, un nouveau bloc, #277317, est miné par un autre mineur. Parce que ce nouveau bloc est construit au-dessus du bloc #277316 qui contenait la transaction d'Alice, il a ajouté encore plus de calculs à la chaîne de blocs, renforçant ainsi la confiance dans ces transactions. Chaque bloc miné en plus de celui contenant la transaction compte comme une confirmation supplémentaire pour la transaction d'Alice. Au fur et à mesure que les blocs s'empilent les uns sur les autres, il devient exponentiellement plus difficile d'inverser la transaction, ce qui la rend de plus en plus fiable par le réseau.

Dans le diagramme en [figure\_title](#block-alice1), nous pouvons voir le bloc #277316, qui contient la transaction d'Alice. En dessous se trouvent 277 316 blocs (y compris le bloc #0), liés les uns aux autres dans une chaîne de blocs (blockchain) jusqu'au bloc #0, connu sous le nom de *bloc d’origine*. Au fil du temps, à mesure que la "hauteur" des blocs augmente, la difficulté de calcul de chaque bloc et de la chaîne dans son ensemble augmente également. Les blocs extraits après celui qui contient la transaction d'Alice agissent comme une assurance supplémentaire, car ils accumulent plus de calculs dans une chaîne de plus en plus longue. Par convention, tout bloc avec plus de six confirmations est considéré comme irrévocable, car il faudrait une immense quantité de calculs pour invalider et recalculer six blocs. Nous examinerons plus en détail le processus d'exploitation minière et la manière dont il renforce la confiance dans [simpara\_title](#mining).



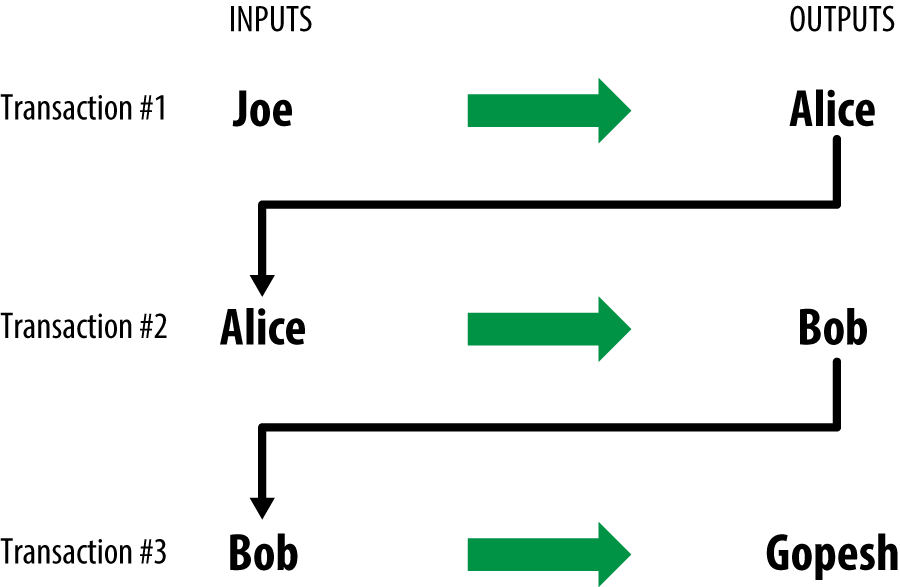
Transaction d'Alice incluse dans le bloc #277316

=== Dépenser la transaction

Maintenant que la transaction d'Alice a été intégrée à la chaîne de blocs dans le cadre d’un bloc, il fait partie du grand livre distribué de Bitcoin et est visible par toutes les applications Bitcoin. Chaque client Bitcoin peut indépendamment vérifier que la transaction est valide et utilisable. Les clients du nœud complet peuvent suivre la source des fonds à partir du moment où les bitcoins ont été générés pour la première fois dans un bloc, progressivement d'une transaction à l'autre, jusqu'à ce qu'ils atteignent l'adresse de Bob. Les clients légers peuvent faire ce qu'on appelle une vérification de paiement simplifiée (voir [simpara\_title](#spv_nodes)) en confirmant que la transaction est dans la chaîne de blocs et qu'elle a plusieurs blocs extraits après elle, fournissant ainsi l'assurance que les mineurs l'ont acceptée comme valide.

Bob peut maintenant dépenser le résultat de cette transaction et d'autres transactions. Par exemple, Bob peut payer un entrepreneur ou un fournisseur en transférant la valeur du paiement de la tasse de café d'Alice à ces nouveaux propriétaires. Très probablement, le logiciel Bitcoin de Bob regroupera de nombreux petits paiements en un paiement plus important, concentrant peut-être tous les revenus Bitcoin de la journée en une seule transaction. Cela regrouperait les différents paiements en un seul résultat (et une seule adresse). Pour un diagramme d'une transaction d'agrégation, voir [figure\_title](#transaction-aggregating).

Au fur et à mesure que Bob dépense les paiements reçus d'Alice et d'autres clients, il étend la chaîne des transactions. Supposons que Bob paie son concepteur Web Gopesh à Bangalore pour une nouvelle page Web. Maintenant, la chaîne de transactions ressemblera à [figure\_title](#block-alice2).



La transaction d'Alice dans le cadre d'une chaîne de transactions de Joe à Gopesh, où la sortie d'une transaction est utilisée comme entrée de la transaction suivante

Dans ce chapitre, nous avons vu comment les transactions construisent une chaîne qui déplace la valeur d'un propriétaire à l'autre. Nous avons également suivi la transaction d'Alice, à partir du moment où elle a été créée dans son portefeuille, via le réseau Bitcoin et jusqu'aux mineurs qui l'ont enregistrée sur la chaîne de blocs. Dans le reste de ce livre, nous examinerons les technologies spécifiques derrière les portefeuilles, les adresses, les signatures, les transactions, le réseau et enfin le minage.

== Bitcoin Core : La référence d'implémentation

Bitcoin est un projet *source libre* et le code source est disponible sous une licence ouverte (MIT), téléchargeable et utilisable à n'importe quelle fin. Source libre signifie plus que simplement libre d'utilisation. Cela signifie également que le bitcoin est développé par une communauté ouverte de bénévoles. Au début, cette communauté se composait uniquement de Satoshi Nakamoto. En 2016, le code source de Bitcoin comptait plus de 400 contributeurs avec une douzaine de développeurs travaillant sur le code presque à temps plein et plusieurs dizaines d'autres à temps partiel. Tout le monde peut contribuer au code, y compris vous !

("Bitcoin Core", "implémentation de référence")Lorsque bitcoin a été créé par Satoshi Nakamoto, le logiciel était en fait terminé avant le livre blanc reproduit dans [???](#satoshi_whitepaper). Satoshi voulait s'assurer que cela fonctionnait avant d'écrire à ce sujet. Cette première implémentation, alors simplement connue sous le nom de "Bitcoin" ou "client Satoshi", a été fortement modifiée et améliorée. Il a évolué vers ce qu'on appelle *Bitcoin Core*, pour le différencier des autres implémentations compatibles. Bitcoin Core est la *mise en œuvre de référence* du système Bitcoin, ce qui signifie qu'il s'agit de la référence faisant autorité sur la manière dont chaque partie de la technologie doit être mise en œuvre. Bitcoin Core implémente tous les aspects du bitcoin, y compris les portefeuilles, un moteur de validation des transactions et des blocs et un nœud de réseau complet dans le réseau Bitcoin pair à pair.

Même si Bitcoin Core inclut une implémentation de référence d'un portefeuille, celui-ci n'est pas destiné à être utilisé comme portefeuille de production pour les utilisateurs ou pour les applications. Il est conseillé aux développeurs d'applications de créer des portefeuilles en utilisant des normes modernes telles que BIP-39 et BIP-32 (voir [simpara\_title](#mnemonic_code_words) et [simpara\_title](#hd_wallets)). BIP signifie *Bitcoin Improvement Proposal*.

[variablelist\_title](#bitcoin_core_architecture) montre l'architecture de Bitcoin Core.

image

images/mbc2\_0301.png["Architecture de base Bitcoin"]

=== Environnement de développement Bitcoin

Si vous êtes un développeur, vous souhaiterez configurer un environnement de développement avec tous les outils, bibliothèques et logiciels de support pour le développement d’applications bitcoin. Dans ce chapitre très technique, nous allons parcourir ce processus étape par étape. Si le matériel devient trop dense (et que vous n'êtes pas en train de mettre en place un environnement de développement), n'hésitez pas à passer au chapitre suivant, qui est moins technique.

=== Compilation de Bitcoin Core à partir du code source

Le code source de Bitcoin Core peut être téléchargé sous forme d'archive ou en clonant le référentiel source faisant autorité à partir de GitHub. Sur la [page de téléchargement de Bitcoin Core](https://bitcoincore.org/bin/), sélectionnez la version la plus récente et téléchargez l'archive compressée du code source, par exemple, bitcoin-0.15.0.2.tar.gz. Vous pouvez également utiliser la ligne de commande git pour créer une copie locale du code source à partir de la [page GitHub de bitcoin](https://github.com/bitcoin/bitcoin).

Dans de nombreux exemples de ce chapitre, nous utiliserons l'interface de de commande du système d'exploitation ( également appelé "shell" ou "interface système" ou "coque logicielle"), accessible via une application "terminal". L’interface système affichera une invite; vous tapez une commande; et l’interface système répond avec du texte et une nouvelle invite pour votre prochaine commande. L'invite peut sembler différente sur votre système, mais dans les exemples suivants, elle est indiquée par un symbole $. Dans les exemples, lorsque vous voyez du texte après un symbole $, ne tapez pas le symbole $ mais tapez la commande qui le suit immédiatement, puis appuyez sur Entrée pour exécuter la commande. Dans les exemples, les lignes sous chaque commande sont les réponses du système d'exploitation à cette commande. Lorsque vous verrez le prochain préfixe $, vous saurez qu'il s'agit d'une nouvelle commande et vous devrez répéter le processus.

Dans cet exemple, nous utilisons la commande git pour créer une copie locale (un "clone") du code source :

$ 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.  
$

Git est le système de contrôle de version distribué le plus largement utilisé et un élément essentiel de la boîte à outils de tout développeur de logiciels. Vous devrez peut-être installer la commande git, ou une interface utilisateur graphique pour git, sur votre système d'exploitation si vous ne l'avez pas déjà.

Une fois l'opération de clonage git terminée, vous aurez une copie locale complète du référentiel de code source dans le répertoire *bitcoin*. Accédez à ce répertoire en tapant \*\*cd bitcoin\*\* à l'invite :

$ cd bitcoin

==== Sélection d'une version de Bitcoin Core Par défaut, la copie locale sera synchronisée avec le code le plus récent, qui peut être une version instable ou bêta de bitcoin. Avant de compiler le code, sélectionnez une version spécifique en consultant une version avec la commande Git *tag*. Cela synchronisera la copie locale avec un instantané (snapshot) spécifique du référentiel de code identifié par une balise de mot-clé (le *tag*.) Les balises sont utilisées par les développeurs pour marquer des versions spécifiques du code par numéro de version. Tout d'abord, pour trouver les balises disponibles, nous utilisons la commande 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 liste des balises montre toutes les versions publiées de bitcoin. Par convention, les *release candidates* (ou *versions d’évaluations*), qui sont destinées aux tests, ont le suffixe "rc". Les versions stables pouvant être exécutées sur des systèmes de production n'ont pas de suffixe. Dans la liste précédente, sélectionnez la version la plus élevée, qui au moment de la rédaction était v0.15.0. Pour synchroniser le code local avec cette version, utilisez la commande 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

Vous pouvez confirmer que vous avez la version souhaitée "extraite" en lançant la commande git status :

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

==== Configuration de la construction de Bitcoin Core

Le code source inclut la documentation, qui peut être trouvée dans un certain nombre de fichiers. Passez en revue la documentation principale située dans *README.md* dans le répertoire *bitcoin* en tapant \*\*more README.md\*\* à l'invite et en utilisant la barre d'espace pour passer à la page suivante. Dans ce chapitre, nous allons construire le client Bitcoin en ligne de commande, également connu sous le nom de bitcoind sous Linux. Passez en revue les instructions pour compiler le client de ligne de commande bitcoind sur votre plate-forme en tapant \*\*more doc/build-unix.md\*\*. Des instructions alternatives pour macOS et Windows peuvent être trouvées dans le répertoire *doc*, comme *build-osx.md* ou *build-windows.md*, respectivement.

Examinez attentivement les prérequis de construction, qui se trouvent dans la première partie de la documentation de construction. Ce sont des bibliothèques qui doivent être présentes sur votre système avant de pouvoir commencer à compiler Bitcoin. Si ces conditions préalables sont manquantes, le processus de génération échouera avec une erreur. Si cela se produit parce que vous avez manqué une condition préalable, vous pouvez l'installer, puis reprendre le processus de génération là où vous l'avez laissé. En supposant que les prérequis sont installés, vous démarrez le processus de génération en générant un ensemble de scripts de génération à l'aide du 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'  
...

Le script *autogen.sh* crée un ensemble de scripts de configuration automatique qui interrogeront votre système pour découvrir les paramètres corrects et s'assurer que vous disposez de toutes les bibliothèques nécessaires pour compiler le code. Le plus important d'entre eux est le script configure qui offre un certain nombre d'options différentes pour personnaliser le processus de construction. Tapez \*\*./configure --help\*\* pour voir les différentes options:

$ ./configure --help  
`configure' configures Bitcoin Core 0.15.0 to adapt to many kinds of systems.  
  
Usage: ./configure [OPTION]... [VAR=VALUE]...  
  
...  
Optional Features:  
 --disable-option-checking ignore unrecognized --enable/--with options  
 --disable-FEATURE do not include FEATURE (same as --enable-FEATURE=no)  
 --enable-FEATURE[=ARG] include FEATURE [ARG=yes]  
  
 --enable-wallet enable wallet (default is yes)  
  
 --with-gui[=no|qt4|qt5|auto]  
...

Le script configure vous permet d'activer ou de désactiver certaines fonctionnalités de bitcoind grâce à l'utilisation des drapeaux --enable-FEATURE et --disable-FEATURE, où FEATURE est remplacé par le nom de la fonctionnalité, comme indiqué dans la sortie d'aide. Dans ce chapitre, nous allons construire le client bitcoind avec toutes les fonctionnalités par défaut. Nous n'utiliserons pas les indicateurs de configuration, mais vous devriez les consulter pour comprendre quelles fonctionnalités facultatives font partie du client. Si vous êtes dans un environnement universitaire, les restrictions du laboratoire informatique peuvent vous obliger à installer des applications dans votre répertoire personnel (par exemple, en utilisant --prefix=$HOME).

Voici quelques options utiles qui remplacent le comportement par défaut du script configure:

--prefix=$HOME

Cela remplace l'emplacement d'installation par défaut (qui est

/usr/local/

) pour l'exécutable résultant. Utilisez

$HOME

pour tout mettre dans votre répertoire personnel, ou un chemin différent.

--disable-wallet

Ceci est utilisé pour désactiver l'implémentation du portefeuille de référence.

--with-incompatible-bdb

Si vous construisez un portefeuille, autorisez l'utilisation d'une version incompatible de la bibliothèque Berkeley DB.

--with-gui=no

Ne construisez pas l'interface utilisateur graphique, qui nécessite la bibliothèque Qt. Cela construit uniquement le bitcoin du serveur et de la ligne de commande.

Ensuite, exécutez le script configure pour découvrir automatiquement toutes les bibliothèques nécessaires et créer un script de construction personnalisé pour votre système :

$ ./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  
...  
[de nombreuses pages de tests de configuration suivent]  
...  
$

Si tout s'est bien passé, la commande configure se terminera par la création des scripts de construction personnalisés qui nous permettront de compiler bitcoind. S'il y a des bibliothèques manquantes ou des erreurs, la commande configure se terminera par une erreur au lieu de créer les scripts de construction. Si une erreur se produit, elle est probablement due à une bibliothèque manquante ou incompatible. Relisez à nouveau la documentation de build et assurez-vous d'installer les prérequis manquants. Ensuite, exécutez à nouveau configure et voyez si cela corrige l'erreur.

==== Construire les exécutables Bitcoin Core

Ensuite, vous allez compiler le code source, un processus qui peut prendre jusqu'à une heure, en fonction de la vitesse de votre processeur et de la mémoire disponible. Pendant le processus de compilation, vous devriez voir une sortie toutes les quelques secondes ou toutes les quelques minutes, ou une erreur si quelque chose ne va pas. Si une erreur se produit ou si le processus de compilation est interrompu, il peut être repris à tout moment en tapant à nouveau make. Tapez \*\*make\*\* pour lancer la compilation de l'application exécutable :

$ 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  
  
[... de nombreux autres messages de compilation suivent ...]  
  
$

Sur un système rapide avec plusieurs processeurs, vous souhaiterez peut-être définir le nombre de tâches de compilation parallèles. Par exemple, make -j 2 utilisera deux cœurs s'ils sont disponibles. Si tout se passe bien, Bitcoin Core est maintenant compilé. Vous devez exécuter la suite de tests unitaires avec make check pour vous assurer que les bibliothèques liées ne sont pas cassées de manière évidente. La dernière étape consiste à installer les différents exécutables sur votre système à l'aide de la commande make install. Vous serez peut-être invité à saisir votre mot de passe utilisateur, car cette étape nécessite des privilèges d'administrateur:

$ 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  
...  
$

L'installation par défaut de bitcoind le place dans */usr/local/bin*. Vous pouvez confirmer que Bitcoin Core est correctement installé en demandant au système le chemin des exécutables, comme suit:

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

=== Exécution d'un nœud Bitcoin Core

Le réseau pair à pair de Bitcoin est composé de "nœuds" de réseau, gérés principalement par des bénévoles et certaines des entreprises qui créent des applications bitcoin. Ceux qui exécutent des nœuds Bitcoin ont une vue directe et faisant autorité sur la chaîne de blocs Bitcoin, avec une copie locale de toutes les transactions, validées indépendamment par leur propre système. En exécutant un nœud, vous n'avez pas besoin de faire appel à un tiers pour valider une transaction. De plus, en exécutant un nœud Bitcoin, vous contribuez au réseau Bitcoin en le rendant plus robuste et sécuritaire.

Cependant, l'exécution d'un nœud nécessite un système connecté en permanence avec suffisamment de ressources pour traiter toutes les transactions bitcoin. Selon que vous choisissez d'indexer toutes les transactions et de conserver une copie complète de la chaîne de blocs, vous pouvez également avoir besoin de beaucoup d'espace disque et de mémoire RAM. Au début de 2021, un nœud à index complet a besoin de 2 Go de RAM et d'un minimum de 360 Go d'espace disque (voir <https://www.blockchain.com/charts/blocks-size>). Les nœuds Bitcoin transmettent et reçoivent également des transactions et des blocs Bitcoin, consommant de la bande passante Internet. Si votre connexion Internet est limitée, a un faible plafond de données ou est mesurée (facturée au gigabit), vous ne devriez probablement pas exécuter un nœud Bitcoin, ou l'exécuter d'une manière qui limite sa bande passante (voir [example\_title](#constrained_resources)).

Bitcoin Core conserve une copie complète de la chaîne de blocs par défaut, avec chaque transaction qui s'est déjà produite sur le Bitcoin réseau depuis sa création en 2009. Cet ensemble de données a une taille de dizaines de gigaoctets et est téléchargé de manière incrémentielle sur plusieurs jours ou semaines, en fonction de la vitesse de votre CPU et de votre connexion Internet. Bitcoin Core ne pourra pas traiter les transactions ou mettre à jour les soldes des comptes tant que l'ensemble de données complet de la chaîne de blocs n'aura pas été téléchargé. Assurez-vous que vous disposez de suffisamment d'espace disque, de bande passante et de temps pour effectuer la synchronisation initiale. Vous pouvez configurer Bitcoin Core pour réduire la taille de la chaîne de blocs en supprimant les anciens blocs (voir [example\_title](#constrained_resources)), mais il téléchargera tout de même l'intégralité de l'ensemble de données avant de les supprimer.

Malgré ces besoins en ressources, des milliers de bénévoles gèrent des nœuds Bitcoin. Certains fonctionnent sur des systèmes aussi simples qu'un Raspberry Pi (un ordinateur à 35$ US de la taille d'un paquet de cartes). De nombreux volontaires exécutent également des nœuds Bitcoin sur des serveurs loués, généralement une variante de Linux. Une instance de *Virtual Private Server* (VPS ou serveur virtuel privé) ou *Cloud Computing Server* (ou serveur en infonuagique) peut être utilisée pour exécuter un nœud Bitcoin. Ces serveurs peuvent être loués pour 25$ à 50$ US par mois auprès de divers fournisseurs.

Pourquoi voudriez-vous exécuter un nœud ? Voici quelques-unes des raisons les plus courantes :

* Si vous développez un logiciel bitcoin et que vous devez vous fier à un nœud Bitcoin pour un accès programmable (API) au réseau et à la blockchain.
* Si vous créez des applications qui doivent valider les transactions selon les règles de consensus de bitcoin. En règle générale, les éditeurs de logiciels Bitcoin exécutent plusieurs nœuds.
* Si vous souhaitez prendre en charge le bitcoin. L'exécution d'un nœud rend le réseau plus robuste et capable de servir plus de portefeuilles, plus d'utilisateurs et plus de transactions.
* Si vous ne souhaitez pas vous fier à un tiers pour traiter ou valider vos transactions.

Si vous lisez ce livre et êtes intéressé par le développement d'un logiciel Bitcoin, vous devriez exécuter votre propre nœud.

==== Configuration du nœud Bitcoin Core

( "sécurité", "mots de passe")Bitcoin Core recherchera un fichier de configuration dans son répertoire de données à chaque démarrage. Dans cette section, nous examinerons les différentes options de configuration et créerons un fichier de configuration. Pour localiser le fichier de configuration, exécutez bitcoind -printtoconsole dans votre terminal et recherchez les deux premières lignes.

$ 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  
...  
[beaucoup plus de sortie de débogage]  
...

Vous pouvez appuyer sur Ctrl-C pour fermer le nœud une fois que vous avez déterminé l'emplacement du fichier de configuration. Habituellement, le fichier de configuration se trouve dans le répertoire de données *.bitcoin* sous le répertoire personnel de votre utilisateur. Il n'est pas créé automatiquement, mais vous pouvez créer un fichier de configuration de démarrage en copiant et collant à partir de l'exemple [example\_title](#full_index_node) ci-dessous. Vous pouvez créer ou modifier le fichier de configuration dans votre éditeur préféré.

Bitcoin Core propose plus de 100 options de configuration qui modifient le comportement du nœud de réseau, le stockage de la chaîne de blocs et de nombreux autres aspects de son fonctionnement. Pour voir une liste de ces options, exécutez 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)  
...  
[beaucoup plus d&#39;options]  
...  
  
 -rpcthreads=<n>  
 Set the number of threads to service RPC calls (default: 4)

Voici quelques-unes des options les plus importantes que vous pouvez définir dans le fichier de configuration, ou en tant que paramètres de ligne de commande pour bitcoind :

alertnotify

Exécute une commande ou un script spécifié pour envoyer des alertes d'urgence au propriétaire de ce nœud, généralement par courriel.

conf

Un emplacement alternatif pour le fichier de configuration. Cela n'a de sens qu'en tant que paramètre de ligne de commande pour bitcoind, car il ne peut pas se trouver dans le fichier de configuration auquel il se réfère.

datadir

Sélectionnez le répertoire et le système de fichiers dans lesquels vous placer toutes les données de la chaîne de blocs. Par défaut, il s'agit d’un sous-répertoire de *.bitcoin* de votre répertoire personnel. Assurez-vous que ce système de fichiers dispose de plusieurs gigaoctets d'espace libre.

prune

Réduisez les besoins en espace disque à ce nombre de mégaoctets, en supprimant les anciens blocs. Utilisez ceci sur un nœud à ressources limitées qui ne peut pas contenir la chaîne de blocs complète.

txindex

Maintient un index de toutes les transactions. Cela signifie une copie complète de la chaîne de blocs. Cela vous permet de récupérer par programmation toute transaction par identifiant (ID).

dbcache

La taille du cache UTXO. La valeur par défaut est 450 Mio. Augmentez cette valeur sur le matériel haut de gamme et réduisez la taille sur le matériel bas de gamme pour économiser de la mémoire au détriment des entrées-sorties des disques lents.

maxconnections

Définissez le nombre maximal de nœuds à partir desquels accepter les connexions. Le réduire par défaut réduira votre consommation de bande passante. À utiliser si vous avez un plafond de données ou si vous payez au gigaoctet.

maxmempool

Limite le bassin de mémoire de transaction à ce nombre de mégaoctets. Utilisez-le pour réduire l'utilisation de la mémoire sur les nœuds à mémoire limitée.

maxreceivebuffer/maxsendbuffer

Limite la mémoire tampon par connexion à ce nombre de multiples de 1000 octets. À utiliser sur les nœuds à mémoire limitée.

minrelaytxfee

Définissez le taux de frais minimum pour la transaction que vous relayerez. En dessous de cette valeur, la transaction est traitée non standard, rejetée du bassin de transactions et non relayée.

Par défaut, Bitcoin Core construit une base de données contenant *uniquement* les transactions liées au portefeuille de l'utilisateur. Si vous voulez pouvoir accéder à *n'importe quelle* transaction avec des commandes comme getrawtransaction (voir [simpara\_title](#exploring_and_decoding_transanctions)), vous devez configurer Bitcoin Core pour créer un index de transaction complet, ce qui peut être réalisé avec l'option txindex. Définissez txindex=1 dans le fichier de configuration de Bitcoin Core. Si vous ne définissez pas cette option au début et que vous la réglez ensuite sur l'indexation complète, vous devez redémarrer bitcoind avec l'option -reindex et attendre qu'elle reconstruise l'index.

[example\_title](#full_index_node) montre comment vous pouvez combiner les options précédentes, avec un nœud entièrement indexé, fonctionnant comme un backend API pour une application bitcoin.

alertnotify=myemailscript.sh &quot;Alerte : %s&quot;  
datadir=/lotsofspace/bitcoin  
index\_tx=1

[example\_title](#constrained_resources) montre un nœud à ressources limitées s'exécutant sur un serveur plus petit.

alertnotify=myemailscript.sh &quot;Alerte : %s&quot;  
maxconnexions=15  
prune=5000  
dbcache=150  
maxmempool=150  
maxreceivebuffer=2500  
maxsendbuffer=500

Une fois que vous avez modifié le fichier de configuration et défini les options qui représentent le mieux vos besoins, vous pouvez tester bitcoind avec cette configuration. Exécutez Bitcoin Core avec l'option printtoconsole pour s'exécuter au premier plan avec une sortie vers la console :

$ bitcoind -printtoconsole  
  
Bitcoin version v0.15.0  
InitParameterInteraction: parameter interaction: -whitelistforcerelay=1 -> setting -whitelistrelay=1  
Assuming ancestors of block 0000000000000000003b9ce759c2a087d52abc4266f8f4ebd6d768b89defa50a 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  
  
[... plus de messages de démarrage ...]

Vous pouvez appuyer sur Ctrl-C pour interrompre le processus une fois que vous êtes convaincu qu'il charge les paramètres correctement et s'exécute comme prévu.

Pour exécuter Bitcoin Core en arrière-plan en tant que processus, démarrez-le avec l'option daemon, en tant que bitcoind -daemon.

Pour surveiller la progression et l'état d'exécution de votre nœud Bitcoin, utilisez la commande 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,  
 [...]  
}

Cela montre un nœud avec une hauteur de blockchain de 0 blocs et 83999 en-têtes. Le nœud récupère actuellement les en-têtes de bloc de la meilleure chaîne et continue ensuite à télécharger les blocs complets.

Une fois que vous êtes satisfait des options de configuration que vous avez sélectionnées, vous devez ajouter *bitcoin* aux scripts de démarrage de votre système d'exploitation, afin qu'il s'exécute en continu et au redémarrage du système d'exploitation. Vous trouverez un certain nombre d'exemples de scripts de démarrage pour différents systèmes d'exploitation dans le répertoire source de bitcoin sous *contrib/init* et un fichier *README.md* indiquant quel système utilise quel script.("", startref="BNcore03")

=== Interface de programmation d'application (API) de base Bitcoin

Le client Bitcoin Core implémente une interface JSON-RPC accessible également à l'aide de l'assistant de ligne de commande bitcoin-cli. La ligne de commande nous permet d'expérimenter de manière interactive les fonctionnalités qui sont également disponibles par programmation via l'API. Pour commencer, appelez la commande help pour voir une liste des commandes RPC bitcoin disponibles :

$ 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 "passphrase" timeout  
walletpassphrasechange "oldpassphrase" "newpassphrase"

Chacune de ces commandes peut prendre un certain nombre de paramètres. Pour obtenir une aide supplémentaire, une description détaillée et des informations sur les paramètres, ajoutez le nom de la commande après help. Par exemple, pour voir l'aide sur la commande getblockhash RPC :

$ bitcoin-cli help getblockhash  
getblockhash height  
  
Returns hash of block in best-block-chain at height provided.  
  
Arguments:  
1. height (numeric, required) The height index  
  
Result:  
"hash" (string) The block hash  
  
Examples:  
> 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/

À la fin des informations d'aide, vous verrez deux exemples de la commande RPC, utilisant l'assistant bitcoin-cli ou le client HTTP curl. Ces exemples montrent comment vous pouvez appeler la commande. Copiez le premier exemple et voyez le résultat :

$ bitcoin-cli getblockhash 1000  
00000000c937983704a73af28acdec37b049d214adbda81d7e2a3dd146f6ed09

Le résultat est un hachage de bloc, qui est décrit plus en détail dans les chapitres suivants. Mais pour l'instant, cette commande devrait renvoyer le même résultat sur votre système, démontrant que votre nœud Bitcoin Core est en cours d'exécution, accepte les commandes et dispose d'informations sur le bloc 1000 à vous renvoyer.

Dans les sections suivantes, nous allons démontrer quelques commandes RPC très utiles et leur sortie attendue.

==== Obtenir des informations sur l'état du client Bitcoin Core

Bitcoin Core fournit des rapports d'état sur différents modules via l'interface JSON-RPC. Les commandes les plus importantes incluent getblockchaininfo, getmempoolinfo, getnetworkinfo et getwalletinfo.

La commande getblockchaininfo RPC de Bitcoin a été introduite plus tôt. La commande getnetworkinfo affiche des informations de base sur l'état du nœud de réseau Bitcoin. Utilisez bitcoin-cli pour l'exécuter :

$ bitcoin-cli getnetworkinfo

"version": 150000,  
 "subversion": "/Satoshi:0.15.0/",  
 "protocolversion": 70015,  
 "localservices": "000000000000000d",  
 "localrelay": true,  
 "timeoffset": 0,  
 "networkactive": true,  
 "connections": 8,  
 "networks": [  
 ...  
 detailed information about all networks (ipv4, ipv6 or onion)  
 ...  
 ],  
 "relayfee": 0.00001000,  
 "incrementalfee": 0.00001000,  
 "localaddresses": [  
 ],  
 "warnings": ""  
}

Les données sont renvoyées en JavaScript Object Notation (JSON), un format qui peut facilement être "consommé" par tous les langages de programmation mais qui est également assez lisible par l'homme. Parmi ces données, nous voyons les numéros de version du client logiciel bitcoin (150000) et du protocole Bitcoin (70015). On y voit le nombre actuel de connexions (8) et diverses informations sur le réseau Bitcoin et les paramètres liés à ce client.

Il faudra un certain temps, peut-être plus d'une journée, pour que le client bitcoind "rattrape" la hauteur actuelle de la chaîne de blocs lorsqu'il télécharge des blocs d'autres clients Bitcoin. Vous pouvez vérifier sa progression en utilisant getblockchaininfo pour voir le nombre de blocs connus.

==== Explorer et décoder les transactions

Commandes : getrawtransaction, decoderawtransaction

Dans [simpara\_title](#cup_of_coffee), Alice a acheté une tasse de café au Bob's Cafe. Sa transaction a été enregistrée sur la blockchain avec l'identifiant de transaction (txid) 0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2. Utilisons l'API pour récupérer et examiner cette transaction en transmettant l'identifiant de transaction en tant que paramètre :

$ bitcoin-cli getrawtransaction 0627052b6f28912f2703066a912ea577f2ce4da4caa5a↵ 5fbd8a57286c345c2f2 0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd734d2804fe65fa35779000↵ 000008b483045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4↵ ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813014↵ 10484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc54123363767↵ 89d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adfffffffff0260e3160000000↵ 0001976a914ab68025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef8000000000001976a9↵ 147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000000

Un identifiant de transaction ne fait pas autorité tant qu'une transaction n'a pas été confirmée. L'absence d'un hachage de transaction dans la blockchain ne signifie pas que la transaction n'a pas été traitée. C'est ce qu'on appelle la "malléabilité des transactions", car les hachages de transaction peuvent être modifiés avant la confirmation dans un bloc. Après confirmation, le txid est immuable et fait autorité.

La commande getrawtransaction renvoie une transaction sérialisée en notation hexadécimale. Pour décoder cela, nous utilisons la commande decoderawtransaction, en passant les données hexadécimales en paramètre. Vous pouvez copier l'hex renvoyé par getrawtransaction et le coller en tant que paramètre dans decoderawtransaction :

$ bitcoin-cli decoderawtransaction 0100000001186f9f998a5aa6f048e51dd8419a14d8↵ a0f1a8a2836dd734d2804fe65fa35779000000008b483045022100884d142d86652a3f47ba474↵ 6ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298↵ cad530a863ea8f53982c09db8f6e381301410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fd↵ e0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa↵ 336a8d752adfffffffff0260e31600000000001976a914ab68025513c3dbd2f7b92a94e0581f5↵ 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" ] } } ] }

Le décodage de la transaction montre tous les composants de cette transaction, y compris les entrées et les sorties de la transaction. Dans ce cas, nous voyons que la transaction qui a crédité notre nouvelle adresse de 15 millibits a utilisé une entrée et généré deux sorties. L'entrée de cette transaction était la sortie d'une transaction précédemment confirmée (indiquée par txid commençant par 7957a35fe). Les deux sorties correspondent au crédit de 15 millibits et une sortie avec retour à l'expéditeur.

Nous pouvons explorer davantage la chaîne de blocs en examinant la transaction précédente référencée par son txid dans cette transaction en utilisant les mêmes commandes (par exemple, getrawtransaction). En sautant d'une transaction à l'autre, nous pouvons suivre une chaîne de transactions à mesure que les pièces sont transmises d'adresse de propriétaire à adresse de propriétaire.

==== Explorer les blocs

Commandes : getblock, getblockhash

L'exploration de blocs est similaire à l'exploration de transactions. Cependant, les blocs peuvent être référencés soit par la *hauteur* du bloc (*height*), soit par le résultat de *hachage* (ou *hash*) du bloc. Tout d'abord, trouvons un bloc par sa hauteur. Dans [simpara\_title](#cup_of_coffee), nous avons vu que la transaction d'Alice était incluse dans le bloc 277316.

Nous utilisons la commande getblockhash, qui prend la hauteur du bloc comme paramètre et renvoie le hachage pour ce bloc :

$ bitcoin-cli getblockhash 277316 0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

Maintenant que nous savons dans quel bloc la transaction d'Alice était incluse, nous pouvons interroger ce bloc. Nous utilisons la commande getblock avec le hachage de bloc comme paramètre :

$ bitcoin-cli getblock 0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b3↵ 1b2cc7bdc4

{ "hash": "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4", "confirmations": 37371, "size": 218629, "height": 277316, "version": 2, "merkleroot": "c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e", "tx": [ "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f", "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe", "04905ff987ddd4cfe603b03cfb7ca50ee81d89d1f8f5f265c38f763eea4a21fd", "32467aab5d04f51940075055c2f20bbd1195727c961431bf0aff8443f9710f81", "561c5216944e21fa29dd12aaa1a45e3397f9c0d888359cb05e1f79fe73da37bd", [... des centaines de transactions ...] "78b300b2a1d2d9449b58db7bc71c3884d6e0579617e0da4991b9734cef7ab23a", "6c87130ec283ab4c2c493b190c20de4b28ff3caf72d16ffa1ce3e96f2069aca9", "6f423dbc3636ef193fd8898dfdf7621dcade1bbe509e963ffbff91f696d81a62", "802ba8b2adabc5796a9471f25b02ae6aeee2439c679a5c33c4bbcee97e081196", "eaaf6a048588d9ad4d1c092539bd571dd8af30635c152a3b0e8b611e67d1a1af", "e67abc6bd5e2cac169821afc51b207127f42b92a841e976f9b752157879ba8bd", "d38985a6a1bfd35037cb7776b2dc86797abbb7a06630f5d03df2785d50d5a2ac", "45ea0a3f6016d2bb90ab92c34a7aac9767671a8a84b9bcce6c019e60197c134b", "c098445d748ced5f178ef2ff96f2758cbec9eb32cb0fc65db313bcac1d3bc98f" ], "time": 1388185914, "mediantime": 1388183675, "nonce": 924591752, "bits": "1903a30c", "difficulty": 1180923195.258026, "chainwork": "000000000000000000000000000000000000000000000934695e92aaf53afa1a", "previousblockhash": "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569", "nextblockhash": "000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7" }

Le bloc contient 419 transactions et la 64ème transaction répertoriée (0627052b...) est le paiement du café d'Alice. L'entrée + hauteur + nous indique qu'il s'agit du 277316e bloc de la blockchain.

==== Utilisation de l'interface de programmation de Bitcoin Core

L'assistant bitcoin-cli est très utile pour explorer la bibliothèue (API) Bitcoin Core et les fonctions de test. Mais tout l'intérêt d'une interface de programmation d'application est d'accéder aux fonctions par programmation. Dans cette section, nous montrerons comment accéder à Bitcoin Core à partir d'un autre programme.

L'API de Bitcoin Core est une interface JSON-RPC. JSON signifie JavaScript Object Notation et c'est un moyen très pratique de présenter des données que les humains et les programmes peuvent facilement lire. RPC signifie Remote Procedure Call (ou appel de procédure à distance), ce qui signifie que nous appelons des procédures (fonctions) distantes (sur le nœud Bitcoin Core) via un protocole réseau. Dans ce cas, le protocole réseau est HTTP ou HTTPS (pour les connexions cryptées).

Lorsque nous avons utilisé la commande bitcoin-cli pour obtenir de l'aide sur une commande, elle nous a montré un exemple d'utilisation de curl, le client HTTP polyvalent en ligne de commande pour construire l'un de ces appels 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/

Cette commande montre que curl soumet une requête HTTP à l'hôte local (127.0.0.1), se connecte au port bitcoin par défaut (8332) et soumet une requête jsonrpc pour la méthode getblockchaininfo en utilisant l'encodage text/plain.

Vous remarquerez peut-être que curl demandera que les informations d'identification soient envoyées avec la demande. Bitcoin Core créera un mot de passe aléatoire à chaque démarrage et le placera dans le répertoire de données sous le nom .cookie. L'assistant bitcoin-cli peut lire ce fichier de mots de passe étant donné le répertoire de données. De même, vous pouvez copier le mot de passe et le transmettre à curl (ou à tout wrapper Bitcoin Core RPC de niveau supérieur). Alternativement, vous pouvez créer un mot de passe statique avec le script d'assistance fourni dans *./share/rpcauth/rpcauth.py* dans le répertoire source de Bitcoin Core.

Si vous implémentez un appel JSON-RPC dans votre propre programme, vous pouvez utiliser une bibliothèque HTTP générique pour construire l'appel, similaire à ce qui est montré dans l'exemple curl précédent.

Cependant, il existe des bibliothèques dans la plupart des langages de programmation qui "enveloppent" l'API Bitcoin Core d'une manière qui rend cela beaucoup plus simple. Nous utiliserons la bibliothèque python-bitcoinlib pour simplifier l'accès à l'API. N'oubliez pas que cela nécessite que vous disposiez d'une instance Bitcoin Core en cours d'exécution, qui sera utilisée pour effectuer des appels JSON-RPC.

Le script Python dans [example\_title](#rpc_example) effectue un simple appel getblockchaininfo et imprime le paramètre blocks à partir des données renvoyées par Bitcoin Core (nœud complet requis).

from bitcoin.rpc import RawProxy  
  
# Create a connection to local Bitcoin Core node  
p = RawProxy()  
  
# Run the getinfo command, store the resulting data in info  
info = p.getinfo()  
  
# Retrieve the 'blocks' element from the info  
print(info['blocks'])

Son exécution nous donne le résultat suivant:

$ python rpc\_example.py  
394075

Il nous indique que notre nœud local Bitcoin Core a 394075 blocs dans sa chaîne de blocs. Ce n'est pas un résultat spectaculaire, mais cela démontre l'utilisation de base de la bibliothèque en tant qu'interface simplifiée avec l'API JSON-RPC de Bitcoin Core.

Ensuite, utilisons les appels getrawtransaction et decodetransaction pour récupérer les détails du paiement du café d'Alice. Dans [example\_title](#rpc_transaction), nous récupérons la transaction d'Alice et listons les sorties de la transaction. Pour chaque sortie, nous affichons l'adresse et la valeur du destinataire. Pour rappel, la transaction d'Alice avait une sortie pour payer Bob's Cafe et une sortie pour rendre la monnaie à Alice.

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'])

En exécutant ce code, nous obtenons :

$ python rpc\_transaction.py  
([u'1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA'], Decimal('0.01500000'))  
([u'1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK'], Decimal('0.08450000'))

Les deux exemples précédents sont assez simples. Vous n'avez pas vraiment besoin d'un programme pour les exécuter ; vous pouvez tout aussi bien utiliser l'assistant bitcoin-cli. L'exemple suivant, cependant, nécessite plusieurs centaines d'appels RPC et démontre plus clairement l'utilisation d'une interface de programmation.

Dans [example\_title](#rpc_block), nous récupérons d'abord le bloc 277316, puis récupérons chacune des 419 transactions en référence à chaque identifiant de transaction. Ensuite, nous parcourons chacune des sorties de la transaction et additionnons la valeur.

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)

En exécutant ce code, nous obtenons :

$ python rpc\_block.py  
  
('Total value in block: ', Decimal('10322.07722534'))

Notre exemple de code calcule que la valeur totale des transactions dans ce bloc est de 10 322,07722534 BTC (dont 25 BTC de récompense et 0,0909 BTC de frais). Comparez cela au montant rapporté par un site d'exploration de blocs en recherchant le hachage ou la hauteur du bloc. Certains explorateurs de blocs rapportent la valeur totale hors récompense et hors frais. Voyez si vous pouvez repérer la différence.

=== Clients, bibliothèques et boîtes à outils alternatifs

Il existe de nombreux clients alternatifs, bibliothèques, boîtes à outils et même des implémentations de nœuds complets dans l'écosystème bitcoin. Ceux-ci sont implémentés dans une variété de langages de programmation, offrant aux programmeurs des interfaces natives dans leur langue préférée.

Les sections suivantes répertorient certaines des meilleures bibliothèques, clients et kits d'outils, organisés par langages de programmation.

==== C/C https://github.com/bitcoin/bitcoin[Bitcoin Core]:: L&#39;implémentation de référence de bitcoin https://github.com/libbitcoin/libbitcoin-system[libbitcoin]:: Boîte à outils de développement C multiplateforme, nœud et bibliothèque de consensus [bitcoin explorer](https://github.com/libbitcoin/libbitcoin-explorer):: L'outil de ligne de commande de Libbitcoin [picocoin](https://github.com/jgarzik/picocoin):: Bibliothèque cliente légère en langage C pour bitcoin par Jeff Garzik

==== JavaScript [bcoin](https://bcoin.io/):: Une implémentation modulaire et évolutive de nœud complet avec API [Bitcore](https://bitcore.io/):: Noeud complet, API et bibliothèque par Bitpay [BitcoinJS](https://github.com/bitcoinjs/bitcoinjs-lib):: Une pure bibliothèque Bitcoin en JavaScript pour node.js et les navigateurs

==== Java [bitcoinj](https://bitcoinj.github.io) :: Une bibliothèque client Java à nœud complet

====PHP [bitwasp/bitcoin](https://github.com/bit-wasp/bitcoin-php):: Une bibliothèque Bitcoin en PHP et des projets associés

====Python [python-bitcoinlib](https://github.com/petertodd/python-bitcoinlib):: Une bibliothèque Bitcoin en Python dont une bibliothèque de consensus et un nœud par Peter Todd [pycoin](https://github.com/richardkiss/pycoin):: Une bibliothèque Bitcoin en Python par Richard Kiss [pybitcointools](https://github.com/primal100/pybitcointools):: Une fourche archivé de la bibliothèque Bitcoin en Python par Vitalik Buterin

==== Ruby [bitcoin-client](https://github.com/sinisterchipmunk/bitcoin-client):: Un wrapper de bibliothèque Ruby pour l'API JSON-RPC

==== Go [btcd](https://github.com/btcsuite/btcd):: Un client Bitcoin à nœud complet en langage Go

==== Rust [rust-bitcoin](https://github.com/rust-bitcoin/rust-bitcoin):: Bibliothèque Bitcoin Rust pour la sérialisation, l'analyse et les appels d'API

==== C# [NBitcoin](https://github.com/MetacoSA/NBitcoin):: Bibliothèque Bitcoin complète pour le framework .NET

==== Objective-C [CoreBitcoin](https://github.com/oleganza/CoreBitcoin) :: Boîte à outils Bitcoin pour ObjC et Swift

De nombreuses autres bibliothèques existent dans une variété d'autres langages de programmation et d'autres sont créées tout le temps.

== Clés, Adresses

Vous avez peut-être entendu dire que le bitcoin est basé sur la *cryptographie*, qui est une branche des mathématiques largement utilisée en sécurité informatique. La cryptographie signifie "écriture secrète" en grec, mais la science de la cryptographie englobe plus que la simple écriture secrète, appelée cryptage. La cryptographie peut également être utilisée pour prouver la connaissance d'un secret sans révéler ce secret (signature numérique), ou prouver l'authenticité des données (empreinte numérique). Ces types de preuves cryptographiques sont les outils mathématiques essentiels au bitcoin et largement utilisés dans les applications bitcoin. Ironiquement, le cryptage n'est pas une partie importante du bitcoin, car ses communications et ses données de transaction ne sont pas cryptées et ne doivent être chiffrés pour protéger les fonds. Dans ce chapitre, nous présenterons une partie de la cryptographie utilisée dans le bitcoin pour contrôler la propriété des fonds, sous la forme de clés, d'adresses et de portefeuilles.

=== Présentation

 La propriété du bitcoin est établie par le biais de *clés numériques*, *adresses Bitcoin* et *signatures numériques*. Les clés numériques ne sont pas réellement stockées dans le réseau, mais sont plutôt créées et stockées par les utilisateurs dans un fichier, ou une simple base de données, appelée *wallet*. Les clés numériques du portefeuille d'un utilisateur sont totalement indépendantes du protocole Bitcoin et peuvent être générées et gérées par le logiciel du portefeuille de l'utilisateur sans référence à la blockchain ni accès à Internet. Les clés permettent de nombreuses propriétés intéressantes du bitcoin, notamment la confiance et le contrôle décentralisés, l'attestation de propriété et le modèle de sécurité à l'épreuve de la cryptographie.

La plupart des transactions bitcoin nécessitent une signature numérique valide à inclure dans la blockchain, qui ne peut être générée qu'avec une clé secrète ; par conséquent, toute personne possédant une copie de cette clé a le contrôle du bitcoin. La signature numérique utilisée pour dépenser des fonds est également appelée *témoin*, un terme utilisé en cryptographie. Les données témoins dans une transaction bitcoin témoignent de la véritable propriété des fonds dépensés.

Les clés sont fournies par paires constituées d'une clé privée (secrète) et une clé publique. Considérez la clé publique comme similaire à un numéro de compte bancaire et la clé privée comme similaire au code PIN secret, ou à la signature d'un chèque, qui permet de contrôler le compte. Ces clés numériques sont très rarement vues par les utilisateurs de bitcoin. Pour la plupart, ils sont stockés dans le fichier du portefeuille et gérés par le logiciel du portefeuille bitcoin.

Dans la partie paiement d'une transaction bitcoin, la clé publique du destinataire est représentée par son empreinte numérique, appelée *adresse Bitcoin*, qui est utilisée de la même manière que le nom du bénéficiaire sur un chèque (c'est-à-dire "Payer à l'ordre de") . Dans la plupart des cas, une adresse Bitcoin est générée à partir et correspond à une clé publique. Cependant, toutes les adresses Bitcoin ne représentent pas des clés publiques ; ils peuvent également représenter d'autres bénéficiaires tels que des scripts, comme nous le verrons plus loin dans ce chapitre. De cette façon, les adresses Bitcoin font abstraction du destinataire des fonds, ce qui rend les destinations de transaction flexibles, similaires aux chèques papier : un instrument de paiement unique qui peut être utilisé pour payer sur les comptes des personnes, payer sur les comptes de l'entreprise, payer des factures ou payer en espèces. L'adresse Bitcoin est la seule représentation des clés que les utilisateurs verront régulièrement, car c'est la partie qu'ils doivent partager avec le monde.

Tout d'abord, nous présenterons la cryptographie et expliquerons les mathématiques utilisées dans le bitcoin. Ensuite, nous verrons comment les clés sont générées, stockées et gérées. Nous passerons en revue les différents formats de codage utilisés pour représenter les clés privées et publiques, les adresses et les adresses de script. Enfin, nous nous pencherons sur l'utilisation avancée des clés et des adresses : adresses personnalisées, multisignatures, script et portefeuilles papier.

==== Cryptographie à clé publique et crypto-monnaie

La cryptographie à clé publique a été inventée dans les années 1970 et constitue une base mathématique pour informatique et sécurité de l'information.

Depuis l'invention de la cryptographie à clé publique, plusieurs fonctions mathématiques appropriées, telles que l'exponentiation des nombres premiers et la multiplication des courbes elliptiques, ont été découvertes. Ces fonctions mathématiques sont pratiquement irréversibles, ce qui signifie qu'elles sont faciles à calculer dans un sens et impossibles à calculer dans le sens opposé. Sur la base de ces fonctions mathématiques, la cryptographie permet la création de secrets numériques et de signatures numériques infalsifiables. Bitcoin utilise la multiplication de courbes elliptiques comme base de sa cryptographie.

En bitcoin, nous utilisons la cryptographie à clé publique pour créer une paire de clés qui contrôle l'accès au bitcoin. La paire de clés se compose d'une clé privée et - dérivée de celle-ci - d'une clé publique unique. La clé publique est utilisée pour recevoir des fonds et la clé privée est utilisée pour signer des transactions afin de dépenser les fonds.

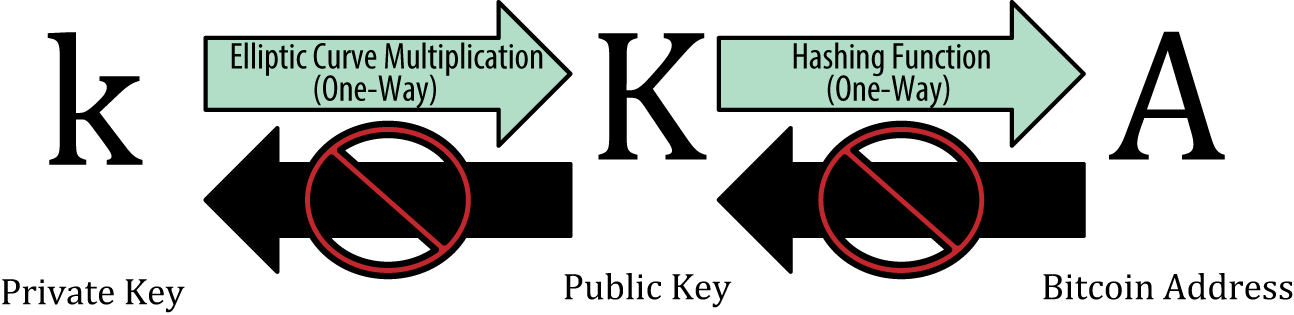
Il existe une relation mathématique entre la clé publique et la clé privée qui permet d'utiliser la clé privée pour générer des signatures sur les messages. Ces signatures peuvent être validées par rapport à la clé publique sans révéler la clé privée.

Lorsqu'il dépense des bitcoins, le propriétaire actuel de bitcoins présente sa clé publique et une signature (différente à chaque fois, mais créée à partir de la même clé privée) dans une transaction pour dépenser ces bitcoins. Grâce à la présentation de la clé publique et de la signature, tous les membres du réseau Bitcoin peuvent vérifier et accepter la transaction comme valide, confirmant que la personne transférant le bitcoin en était propriétaire au moment du transfert.

Dans la plupart des implémentations de portefeuille, les clés privées et publiques sont stockées ensemble sous la forme d'une *paire de clés* pour plus de commodité. Cependant, la clé publique peut être calculée à partir de la clé privée, de sorte qu'il est également possible de ne stocker que la clé privée.

==== Clés privées et publiques

Un portefeuille bitcoin contient une collection de paires de clés, chacune composée d'une clé privée et d'une clé publique. La clé privée (k) est un nombre, généralement choisi au hasard. A partir de la clé privée, nous utilisons la multiplication par courbe elliptique, une fonction cryptographique à sens unique, pour générer une clé publique (K). A partir de la clé publique (K), nous utilisons une fonction de hachage cryptographique unidirectionnelle pour générer une adresse Bitcoin (A). Dans cette section, nous commencerons par générer la clé privée, examinerons les mathématiques de la courbe elliptique utilisées pour la transformer en clé publique, et enfin, générerons une adresse Bitcoin à partir de la clé publique. La relation entre la clé privée, la clé publique et l'adresse Bitcoin est indiquée dans <<k\_to\_K\_to\_A> >.



Clé privée, clé publique et adresse Bitcoin

Pourquoi la cryptographie asymétrique est-elle utilisée dans le bitcoin ? Il n'est pas utilisé pour "crypter" (rendre secrètes) les transactions. Au contraire, la propriété utile de la cryptographie asymétrique est la capacité à générer des *signatures numériques*. Une clé privée peut être appliquée à l'empreinte numérique d'une transaction pour produire une signature numérique. Cette signature ne peut être produite que par une personne connaissant la clé privée. Cependant, toute personne ayant accès à la clé publique et à l'empreinte digitale de la transaction peut les utiliser pour *vérifier* la signature. Cette propriété utile de la cryptographie asymétrique permet à quiconque de vérifier chaque signature sur chaque transaction, tout en garantissant que seuls les propriétaires de clés privées peuvent produire des signatures valides.

==== Clés privées

Une clé privée est simplement un nombre, choisi au hasard . La propriété et le contrôle de la clé privée sont à la base du contrôle de l'utilisateur sur tous les fonds associés à l'adresse Bitcoin correspondante. La clé privée est utilisée pour créer des signatures nécessaires pour dépenser des bitcoins en prouvant la propriété des fonds utilisés dans une transaction. La clé privée doit rester secrète à tout moment, car la révéler à des tiers équivaut à leur donner le contrôle sur le bitcoin sécurisé par cette clé. La clé privée doit également être sauvegardée et protégée contre toute perte accidentelle, car si elle est perdue, elle ne peut pas être récupérée et les fonds qu'elle garantit sont également perdus à jamais.

La clé privée bitcoin n'est qu'un nombre. Vous pouvez choisir vos clés privées au hasard en utilisant simplement une pièce de monnaie, un crayon et du papier : lancez une pièce 256 fois et vous avez les chiffres binaires d'une clé privée aléatoire que vous pouvez utiliser dans un portefeuille bitcoin. La clé publique peut alors être générée à partir de la clé privée.

===== Générer une clé privée à partir d'un nombre aléatoire

La première et la plus importante étape de la génération de clés consiste à trouver une source sécurisée d'entropie ou de caractère aléatoire. La création d'une clé bitcoin est essentiellement la même que "Choisissez un nombre entre 1 et 2 ^ 256 ^". La méthode exacte que vous utilisez pour choisir ce nombre n'a pas d'importance tant qu'elle n'est pas prévisible ou reproductible. Le logiciel Bitcoin utilise les générateurs de nombres aléatoires du système d'exploitation sous-jacent pour produire 256 bits d'entropie (aléatoire). Habituellement, le générateur de nombres aléatoires du système d'exploitation est initialisé par une source humaine d'aléatoire, c'est pourquoi il peut vous être demandé de remuer votre souris pendant quelques secondes.

Plus précisément, la clé privée peut être n'importe quel nombre compris entre 0 et n - 1 inclus, où n est une constante (n = 1,1578 \* 1077, légèrement inférieure à 2256) définie comme l'ordre de la courbe elliptique utilisée dans le bitcoin (voir <<elliptic\_curve> >). Pour créer une telle clé, nous choisissons au hasard un nombre de 256 bits et vérifions qu'il est inférieur à n. En termes de programmation, cela est généralement réalisé en alimentant une plus grande chaîne de bits aléatoires, collectés à partir d'une source aléatoire sécurisée par cryptographie, dans l'algorithme de hachage SHA256, qui produira commodément un nombre de 256 bits. Si le résultat est inférieur à n, nous avons une clé privée appropriée. Sinon, nous réessayons simplement avec un autre nombre aléatoire.

N'écrivez pas votre propre code pour créer un nombre aléatoire ou utilisez un nombre aléatoire "simple" générateur offert par votre langage de programmation. Utilisez un générateur de nombres pseudo-aléatoires cryptographiquement sécurisé (CSPRNG) avec une graine provenant d'une source d'entropie suffisante. Étudiez la documentation de la bibliothèque de générateurs de nombres aléatoires que vous choisissez pour vous assurer qu'elle est cryptographiquement sécurisée. La mise en œuvre correcte du CSPRNG est essentielle à la sécurité des clés.

Ce qui suit est une clé privée générée aléatoirement (k) affichée au format hexadécimal (256 bits affichés sous la forme de 64 chiffres hexadécimaux, chacun de 4 bits) :

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

La taille de l'espace de clé privée de bitcoin, (2 ^ 256 ^) est un nombre insondable. Il est d'environ 1077 en décimal. À titre de comparaison, on estime que l'univers visible contient 1080 atomes.

Pour générer une nouvelle clé avec le client Bitcoin Core (voir <<ch03\_bitcoin\_client> >), utilisez la commande getnewaddress. Pour des raisons de sécurité, il affiche uniquement l'adresse, pas la clé privée. Pour demander à bitcoind d'exposer la clé privée, utilisez la commande dumpprivkey. La commande dumpprivkey affiche la clé privée dans un format de somme de contrôle Base58 appelé *Wallet Import Format* (WIF), que nous examinerons plus en détail dans <<priv\_formats> >. Voici un exemple de génération et d'affichage d'une clé privée à l'aide de ces deux commandes :

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

La commande dumpprivkey ouvre le portefeuille et extrait la clé privée qui a été générée par la commande getnewaddress. Il n'est pas possible pour bitcoind de connaître la clé privée à partir de l'adresse à moins qu'elles ne soient toutes deux stockées dans le portefeuille.

La commande dumpprivkey ne génère pas de clé privée à partir d'une adresse, car cela est impossible. La commande révèle simplement la clé privée qui est déjà connue du portefeuille et qui a été générée par la commande getnewaddress.

Vous pouvez également utiliser l'outil de ligne de commande Bitcoin Explorer (voir <<appdx\_bx> >) pour générer et afficher des clés privées avec les commandes seed, ec-new et ec-to-wif :

$ bx semence | bx ec-nouveau | bx ec vers wif  
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

==== Clés publiques

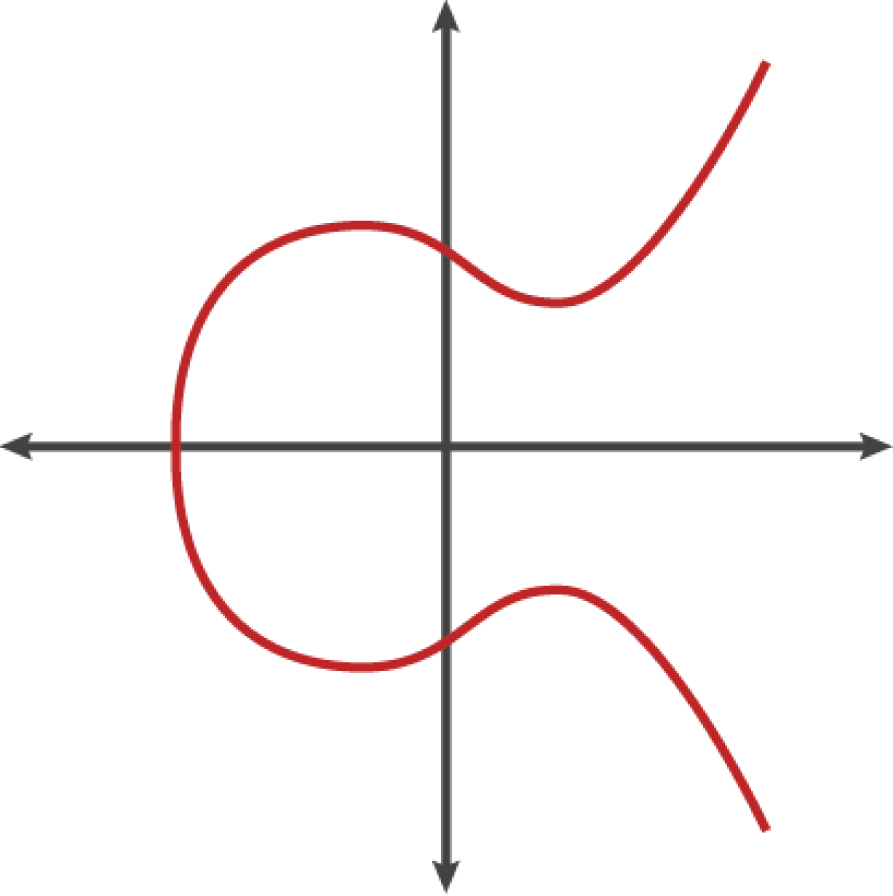
La clé publique est calculée à partir de la clé privée en utilisant la multiplication par courbe elliptique, qui est irréversible : *K* = *k* \* *G*, où *k* est la clé privée, *G* est un point constant appelé *point générateur* et *K* est la clé publique résultante. L'opération inverse, connue sous le nom de "trouver le logarithme discret" - calculer *k* si vous connaissez *K* - est aussi difficile que d'essayer toutes les valeurs possibles de *k*, c'est-à-dire une recherche par force brute. Avant de montrer comment générer une clé publique à partir d'une clé privée, examinons un peu plus en détail la cryptographie à courbe elliptique.

La multiplication de courbe elliptique est un type de fonction que les cryptographes appellent une fonction « à sens unique » : elle est facile à faire dans un sens (multiplication) et impossible à faire dans le sens inverse (« division », ou trouver le logarithme discret). Le propriétaire de la clé privée peut facilement créer la clé publique, puis la partager avec le monde entier, sachant que personne ne peut inverser la fonction et calculer la clé privée à partir de la clé publique. Cette astuce mathématique devient la base de signatures numériques infalsifiables et sécurisées qui prouvent la propriété des fonds bitcoin.

==== La cryptographie à courbe elliptique expliquée

La cryptographie à courbe elliptique est un type de cryptographie asymétrique ou à clé publique basée sur le problème du logarithme discret exprimé par addition et multiplication sur les points d'une courbe elliptique.

<<ecc-curve> > est un exemple de courbe elliptique, similaire à celle utilisée par le bitcoin.



Une courbe elliptique

Bitcoin utilise une courbe elliptique spécifique et un ensemble de constantes mathématiques, telles que définies dans une norme appelée secp256k1, établie par le National Institute of Standards and Technology (NIST). La courbe secp256k1 est définie par la fonction suivante, qui produit une courbe elliptique :

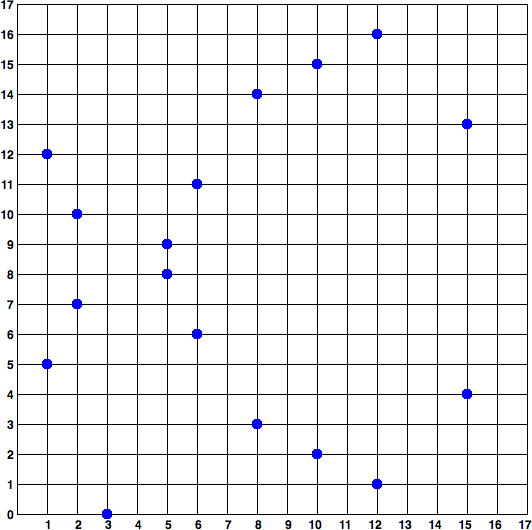
\begin{equation} {y^2 = (x^3 + 7)}~\text{over}~(\mathbb{F}\_p) \end{equation}

ou

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

Le *mod p* (modulo nombre premier p) indique que cette courbe est sur un corps fini d'ordre premier *p*, également écrit comme latexmath :[\( \mathbb{F}\_p \)], où p = 2256 – 2 32 – 29 – 28 – 27 – 26 – 24 – 1, un très grand nombre premier.

Parce que cette courbe est définie sur un champ fini d'ordre premier au lieu de sur les nombres réels, elle ressemble à un motif de points dispersés en deux dimensions, ce qui la rend difficile à visualiser. Cependant, le calcul est identique à celui d'une courbe elliptique sur des nombres réels. A titre d'exemple, <<ecc-over-F17-math> > montre la même courbe elliptique sur un champ fini beaucoup plus petit d'ordre premier 17, montrant un motif de points sur une grille. La courbe elliptique du bitcoin secp256k1 peut être considérée comme un motif beaucoup plus complexe de points sur une grille insondable.



Cryptographie sur courbe elliptique : visualisation d'une courbe elliptique sur F(p), avec p=17

Ainsi, par exemple, ce qui suit est un point P de coordonnées (x,y) qui est un point sur la courbe secp256k1 :

P = (55066263022277343669578718895168534326250603453777594175500187360389116729240, 32670510020733184471273335757337482424)

<<example\_4\_1> > montre comment vous pouvez vérifier cela vous-même en utilisant Python :

Python 3.4.0 (par défaut, 30 mars 2014, 19:23:13)  
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.38)] sur darwin  
Tapez &quot;aide&quot;, &quot;copyright&quot;, &quot;crédits&quot; ou &quot;licence&quot; pour plus d&#39;informations.  
&gt;&gt;&gt; p = 115792089237316195423570985008687907853269984665640564039457584007908834671663  
&gt;&gt;&gt; x = 55066263022277343669578718895168534326250603453777594175500187360389116729240  
&gt;&gt;&gt; y = 32670510020758816978083085130507043184471273380659243275938904335757337482424  
&gt;&gt;&gt; (x \*\* 3 + 7 - y\*\*2) % p  
0

Dans les mathématiques des courbes elliptiques, il existe un point appelé le "point à l'infini", qui correspond à peu près au rôle de zéro en plus. Sur les ordinateurs, il est parfois représenté par x = y = 0 (ce qui ne satisfait pas l'équation de la courbe elliptique, mais c'est un cas séparé facile qui peut être vérifié).

Il existe également un opérateur +, appelé "addition", qui possède certaines propriétés similaires à l'addition traditionnelle de nombres réels apprise par les élèves du primaire. Étant donné deux points P1 et P2 sur la courbe elliptique, il existe un troisième point P3 = P1 + P2, également sur la courbe elliptique.

Géométriquement, ce troisième point P3 est calculé en traçant une ligne entre P1 et P2. Cette ligne coupera la courbe elliptique exactement à un endroit supplémentaire. Appelons ce point P3' = (x, y). Réfléchissez ensuite sur l'axe des x pour obtenir P3 = (x, –y).

Il y a quelques cas particuliers qui expliquent la nécessité du "point à l'infini".

Si P1 et P2 sont le même point, la ligne "entre" P1 et P2 doit s'étendre pour être la tangente sur la courbe en ce point P1. Cette tangente coupera la courbe en exactement un nouveau point. Vous pouvez utiliser des techniques de calcul pour déterminer la pente de la ligne tangente. Ces techniques fonctionnent curieusement, même si nous restreignons notre intérêt aux points de la courbe à deux coordonnées entières !

Dans certains cas (par exemple, si P1 et P2 ont les mêmes valeurs x mais des valeurs y différentes), la ligne entre P1 et P2 sera exactement verticale, auquel cas P~3 ~ = "point à l'infini."

Si P1 est le "point à l'infini", alors P1 + P2 = P2. De même, si P2 est le point à l'infini, alors P1 + P2 = P1. Cela montre comment le point à l'infini joue le rôle de zéro.

Il s'avère que + est associatif, ce qui signifie que (A + B) + C = A + (B + C). Cela signifie que nous pouvons écrire A pass :[] B pass :[] C sans parenthèses et sans ambiguïté.

Maintenant que nous avons défini l'addition, nous pouvons définir la multiplication de la manière standard qui étend l'addition. Pour un point P sur la courbe elliptique, si k est un nombre entier, alors kP = P + P + P + …​ + P (k fois). Notez que k est parfois appelé un "exposant" dans ce cas.

==== Génération d'une clé publique

En partant d'une clé privée sous la forme d'un nombre généré aléatoirement *k*, nous la multiplions par un point prédéterminé sur la courbe appelé *point générateur* *G* pour produire un autre point ailleurs sur la courbe, qui est la clé publique correspondante *K*. Le point générateur est spécifié dans le cadre de la norme secp256k1 et est toujours le même pour toutes les clés en bitcoin :

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

où *k* est la clé privée, *G* est le point générateur et *K* est la clé publique résultante, un point sur la courbe. Comme le point générateur est toujours le même pour tous les utilisateurs de bitcoins, une clé privée *k* multipliée par *G* donnera toujours la même clé publique *K*. La relation entre *k* et *K* est fixe, mais ne peut être calculée que dans un sens, de *k* vers *K*. C'est pourquoi une adresse Bitcoin (dérivée de *K*) peut être partagée avec n'importe qui et ne révèle pas la clé privée de l'utilisateur (*k*).

Une clé privée peut être convertie en clé publique, mais une clé publique ne peut pas être reconvertie en clé privée car le calcul ne fonctionne que dans un sens.

En implémentant la multiplication par courbe elliptique, nous prenons la clé privée *k* générée précédemment et la multiplions avec le point générateur G pour trouver la clé publique *K* :

K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD \* G

La clé publique *K* est définie comme un point K = (x,y) :

K = (x, y)  
  
où,  
  
x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A  
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

Pour visualiser la multiplication d'un point avec un nombre entier, nous utiliserons la courbe elliptique plus simple sur des nombres réels - rappelez-vous, le calcul est le même. Notre objectif est de trouver le multiple *kG* du point générateur *G*, ce qui revient à ajouter *G* à lui-même, *k* fois de suite. Dans les courbes elliptiques, l'ajout d'un point à lui-même équivaut à tracer une ligne tangente sur le point et à trouver à nouveau l'endroit où il coupe la courbe, puis à refléter ce point sur l'axe des x.

<<ecc\_illustrated> > montre le processus de dérivation de *G*, *2G*, *4G* et *8G* en tant qu'opération géométrique sur la courbe.

Bitcoin utilise la [bibliothèque C optimisée secp256k1](https://github.com/bitcoin-core/secp256k1) pour effectuer les calculs de la courbe elliptique.



Cryptographie sur courbe elliptique : visualisation de la multiplication d'un point G par un entier k sur une courbe elliptique

=== Adresses Bitcoin

Une adresse Bitcoin est une chaîne de chiffres et de caractères qui peut être partagée avec toute personne souhaitant vous envoyer de l'argent. Les adresses produites à partir de clés publiques consistent en une chaîne de chiffres et de lettres commençant par le chiffre "1". Voici un exemple d'adresse Bitcoin :

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

L'adresse Bitcoin est ce qui apparaît le plus souvent dans une transaction en tant que "destinataire" des fonds. Si nous comparons une transaction bitcoin à un chèque papier, l'adresse Bitcoin est le bénéficiaire, c'est ce que nous écrivons sur la ligne après "Payer à l'ordre de". Sur un chèque papier, ce bénéficiaire peut parfois être le nom d'un titulaire de compte bancaire, mais peut également inclure des sociétés, des institutions ou même des espèces. Étant donné que les chèques papier n'ont pas besoin de spécifier un compte, mais utilisent plutôt un nom abstrait en tant que destinataire des fonds, ce sont des instruments de paiement très flexibles. Les transactions Bitcoin utilisent une abstraction similaire, l'adresse Bitcoin, pour les rendre très flexibles. Une adresse Bitcoin peut représenter le propriétaire d'une paire de clés privée/publique, ou elle peut représenter autre chose, comme un script de paiement, comme nous le verrons dans <<p2sh> >. Pour l'instant, examinons le cas simple, une adresse Bitcoin qui représente et est dérivée d'une clé publique.

L'adresse Bitcoin est dérivée de la clé publique grâce à l'utilisation d'un hachage cryptographique unidirectionnel. Un "algorithme de hachage" ou simplement "algorithme de hachage" est une fonction à sens unique qui produit une empreinte digitale ou "hachage" d'une entrée de taille arbitraire. Les fonctions de hachage cryptographiques sont largement utilisées dans le bitcoin : dans les adresses Bitcoin, dans les adresses de script et dans l'algorithme de minage Proof-of-Work. Les algorithmes utilisés pour créer une adresse Bitcoin à partir d'une clé publique sont le Secure Hash Algorithm (SHA) et le RACE Integrity Primitives Evaluation Message Digest (RIPEMD), plus précisément SHA256 et RIPEMD160.

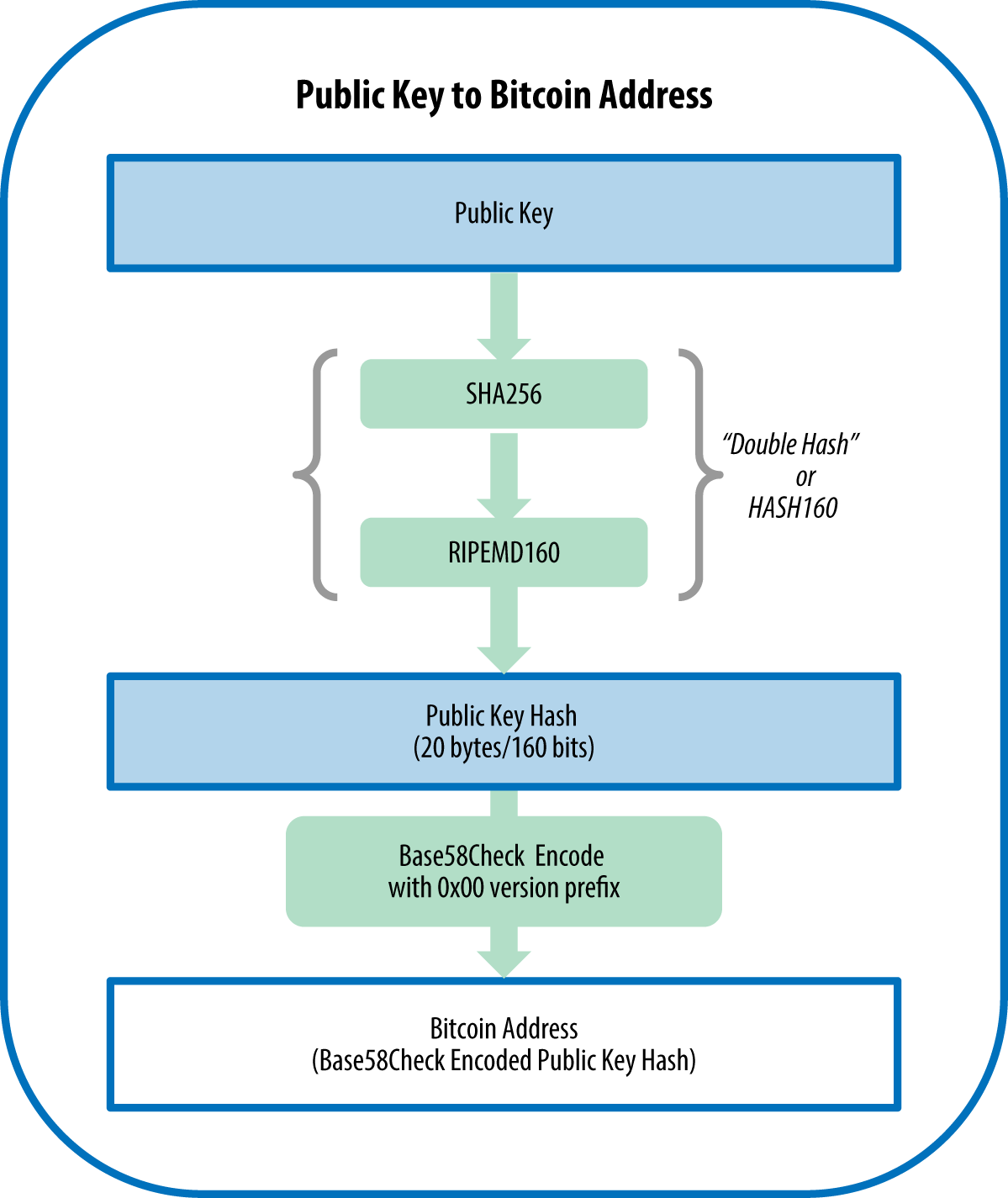
En commençant par la clé publique *K*, nous calculons le hachage SHA256, puis calculons le hachage RIPEMD160 du résultat, produisant un nombre de 160 bits (20 octets) :

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

où *K* est la clé publique et *A* est l'adresse Bitcoin résultante.

Une adresse Bitcoin n'est *pas* la même chose qu'une clé publique. Les adresses Bitcoin sont dérivées d'une clé publique à l'aide d'une fonction unidirectionnelle.

Les adresses Bitcoin sont presque toujours encodées en "Base58Check" (voir <<base58> >), qui utilise 58 caractères (un système de numérotation Base58) et une somme de contrôle pour faciliter la lisibilité humaine, éviter toute ambiguïté et protéger contre les erreurs de transcription et de saisie d'adresse. Base58Check est également utilisé de nombreuses autres manières dans Bitcoin, chaque fois qu'un utilisateur a besoin de lire et de transcrire correctement un numéro, tel qu'une adresse Bitcoin, une clé privée, une clé cryptée ou un hachage de script. Dans la section suivante, nous examinerons les mécanismes d'encodage et de décodage Base58Check et les représentations qui en résultent. <<pubkey\_to\_address> > illustre la conversion d'une clé publique en une adresse Bitcoin.



Clé publique vers adresse Bitcoin : conversion d'une clé publique en adresse Bitcoin

==== Encodage Base58 et Base58Check

Afin de représenter les nombres longs de manière compacte, en utilisant moins de symboles, de nombreux systèmes informatiques utilisent des représentations alphanumériques mixtes avec une base (ou base) supérieure à 10. Par exemple, alors que le système décimal traditionnel utilise les 10 chiffres de 0 à 9, le système hexadécimal utilise 16, avec les lettres A à F comme six symboles supplémentaires. Un nombre représenté au format hexadécimal est plus court que la représentation décimale équivalente. Encore plus compacte, la représentation Base64 utilise 26 lettres minuscules, 26 lettres majuscules, 10 chiffres et 2 caractères supplémentaires tels que "``&quot; et &quot;/&quot; pour transmettre des données binaires sur des supports textuels tels que le courrier électronique. Base64 est le plus couramment utilisé pour ajouter des pièces jointes binaires aux e-mails. Base58 est un format de codage binaire basé sur du texte développé pour être utilisé dans le bitcoin et utilisé dans de nombreuses autres crypto-monnaies. Il offre un équilibre entre représentation compacte, lisibilité et détection et prévention des erreurs. Base58 est un sous-ensemble de Base64, utilisant des lettres et des chiffres majuscules et minuscules, mais omettant certains caractères qui sont fréquemment confondus et qui peuvent apparaître identiques lorsqu&#39;ils sont affichés dans certaines polices. Plus précisément, Base58 est Base64 sans le 0 (chiffre zéro), O (o majuscule), l (L inférieur), I (i majuscule) et les symboles &quot;``" et "/". Ou, plus simplement, c'est un ensemble de lettres minuscules et majuscules et de chiffres sans les quatre (0, O, l, I) que nous venons de mentionner. <<base58alphabet> > affiche l'alphabet Base58 complet.

Alphabet Base58 de .Bitcoin

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

Pour ajouter une sécurité supplémentaire contre les fautes de frappe ou les erreurs de transcription, Base58Check est un format d'encodage Base58, fréquemment utilisé dans le bitcoin, qui possède un code de vérification des erreurs intégré. La somme de contrôle est constituée de quatre octets supplémentaires ajoutés à la fin des données en cours de codage. La somme de contrôle est dérivée du hachage des données codées et peut donc être utilisée pour détecter et prévenir les erreurs de transcription et de frappe. Lorsqu'il est présenté avec le code Base58Check, le logiciel de décodage calcule la somme de contrôle des données et la compare à la somme de contrôle incluse dans le code. Si les deux ne correspondent pas, une erreur a été introduite et les données Base58Check ne sont pas valides. Cela empêche qu'une adresse Bitcoin mal saisie soit acceptée par le logiciel du portefeuille comme destination valide, une erreur qui entraînerait autrement une perte de fonds.

Pour convertir des données (un nombre) dans un format Base58Check, nous ajoutons d'abord un préfixe aux données, appelé "octet de version", qui sert à identifier facilement le type de données encodées. Par exemple, dans le cas d'une adresse Bitcoin, le préfixe est zéro (0x00 en hexadécimal), alors que le préfixe utilisé lors de l'encodage d'une clé privée est 128 (0x80 en hexadécimal). Une liste des préfixes de version courants est affichée dans <<base58check\_versions> >.

Ensuite, nous calculons la somme de contrôle "double-SHA", ce qui signifie que nous appliquons l'algorithme de hachage SHA256 deux fois sur le résultat précédent (préfixe et données):

somme de contrôle = SHA256(SHA256(préfixe+données))

À partir du hachage de 32 octets résultant (hash-of-a-hash), nous ne prenons que les quatre premiers octets. Ces quatre octets servent de code de contrôle d'erreur ou de somme de contrôle. La somme de contrôle est concaténée (ajoutée) à la fin.

Le résultat est composé de trois éléments : un préfixe, les données et une somme de contrôle. Ce résultat est encodé en utilisant l'alphabet Base58 décrit précédemment. <<base58check\_encoding> > illustre le processus d'encodage Base58Check.

Encodage .Base58Check : un format Base58, versionné et à somme de contrôle pour encoder sans ambiguïté les données bitcoin image::images/mbc2\_0406.png["Base58CheckEncoding"]

En bitcoin, la plupart des données présentées à l'utilisateur sont encodées en Base58Check pour les rendre compactes, faciles à lire et faciles à détecter les erreurs. Le préfixe de version dans l'encodage Base58Check est utilisé pour créer des formats faciles à distinguer qui, lorsqu'ils sont encodés en Base58, contiennent des caractères spécifiques au début de la charge utile encodée en Base58Check. Ces caractères permettent aux humains d'identifier facilement le type de données codées et comment les utiliser. C'est ce qui différencie, par exemple, une adresse Bitcoin encodée en Base58Check qui commence par un 1 d'une clé privée WIF encodée en Base58Check qui commence par un 5. Quelques exemples de préfixes de version et les caractères Base58 résultants sont affichés dans <<base58check\_versions> >.

Base58Check préfixe de version et exemples de résultats encodés

|  |  |  |
| --- | --- | --- |
| Type | Préfixe de version (hex) | Préfixe de résultat Base58 |
| Adresse Bitcoin | 0x00 | 1 |
| Adresse de hachage Pay-to-Script | 0x05 | 3 |
| Adresse de test Bitcoin | 0x6F | m ou n |
| Clé privée WIF | 0x80 | 5, K ou L |
| Clé privée cryptée BIP-38 | 0x0142 | 6P |
| Clé publique étendue BIP-32 | 0x0488B21E | xpub |

==== Formats de clé

Les clés privées et publiques peuvent être représentées dans un certain nombre de formats différents. Ces représentations codent toutes le même nombre, même si elles semblent différentes. Ces formats sont principalement utilisés pour faciliter la lecture et la transcription des clés sans introduire d'erreurs.

===== Formats de clé privée

La clé privée peut être représentée dans un certain nombre de formats différents, qui correspondent tous au même nombre de 256 bits. <<table\_4-2> > montre trois formats courants utilisés pour représenter les clés privées. Différents formats sont utilisés dans différentes circonstances. Les formats binaires hexadécimaux et bruts sont utilisés en interne dans les logiciels et rarement montrés aux utilisateurs. Le WIF est utilisé pour l'import/export de clés entre portefeuilles et souvent utilisé dans les représentations de code QR (code-barres) des clés privées.

Représentations de clés privées (formats d'encodage)

|  |  |  |
| --- | --- | --- |
| Type | Préfixe | Description |
| Brut | Aucun | 32 octets |
| Hex | Aucun | 64 chiffres hexadécimaux |
| WIF | 5 | Encodage Base58Check : Base58 avec préfixe de version 0x80 et somme de contrôle de 4 octets |
| WIF compressé | K ou L | Comme ci-dessus, avec le suffixe ajouté 0x01 avant l'encodage |

<<table\_4-3> > affiche la clé privée générée dans ces trois formats.

Exemple : même clé, différents formats

|  |  |
| --- | --- |
| Formater | Clé privée |
| Hex | 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd |
| WIF | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |
| WIF compressé | KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ |

Toutes ces représentations sont des manières différentes de montrer le même numéro, la même clé privée. Ils semblent différents, mais n'importe quel format peut facilement être converti en n'importe quel autre format. Notez que le "binaire brut" n'est pas affiché dans <<table\_4-3> > car tout encodage à afficher ici ne serait, par définition, pas une donnée binaire brute.

Nous utilisons la commande wif-to-ec de Bitcoin Explorer (voir <<appdx\_bx> >) pour montrer que les deux clés WIF représentent la même clé privée :

$ bx wif-to-ec 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn  
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd  
  
$ bx wif-to-ec KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ  
1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

===== Décoder à partir de Base58Check

Les commandes de Bitcoin Explorer (voir <<appdx\_bx> >) facilitent l'écriture de scripts shell et de "tubes" de ligne de commande qui manipulent les clés, les adresses et les transactions bitcoin. Vous pouvez utiliser Bitcoin Explorer pour décoder le format Base58Check sur la ligne de commande.

Nous utilisons la commande base58check-decode pour décoder la clé non compressée :

$ bx base58vérifier-décoder 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn  
emballage  
{  
 somme de contrôle 4286807748  
 charge utile 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd  
 édition 128  
}

Le résultat contient la clé comme charge utile, le préfixe de version WIF 128 et une somme de contrôle.

Notez que la "charge utile" de la clé compressée est ajoutée avec le suffixe 01, signalant que la clé publique dérivée doit être compressée :

$ bx base58check-decode KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ  
emballage  
{  
 somme de contrôle 2339607926  
 charge utile 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01  
 édition 128  
}

===== Encoder de l'hex vers Base58Check

Pour encoder en Base58Check (l'inverse de la commande précédente), nous utilisons la commande base58check-encode de Bitcoin Explorer (voir <<appdx\_bx> >) et indiquez la clé privée hexadécimale, suivie du préfixe de version WIF 128 :

bx base58check-encoder 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd --version 128  
5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

===== Encoder de l'hex (clé compressée) à Base58Check

Pour encoder dans Base58Check comme une clé privée "compressée" (voir <<comp\_priv> >), nous ajoutons le suffixe 01 à la clé hexadécimale puis encodons comme dans la section précédente :

$ bx base58check-encoder 1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd01 --version 128  
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Le format compressé WIF résultant commence par un "K". Cela indique que la clé privée à l'intérieur a un suffixe de "01" et sera utilisée pour produire des clés publiques compressées uniquement (voir <<comp\_pub> >).

===== Formats de clé publique

Les clés publiques sont également présentées de différentes manières, généralement sous forme de clés publiques *compressées* ou *non compressées*.

Comme nous l'avons vu précédemment, la clé publique est un point de la courbe elliptique constitué d'un couple de coordonnées (x,y). Il est généralement présenté avec le préfixe 04 suivi de deux nombres de 256 bits : un pour la coordonnée *x* du point, l'autre pour la coordonnée *y*. Le préfixe 04 est utilisé pour distinguer les clés publiques non compressées des clés publiques compressées qui commencent par un 02 ou un 03.

Voici la clé publique générée par la clé privée que nous avons créée précédemment, indiquée par les coordonnées x et y :

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A  
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

Voici la même clé publique affichée sous la forme d'un nombre de 520 bits (130 chiffres hexadécimaux) avec le préfixe 04 suivi de x puis des coordonnées y, sous la forme 04 x y :

K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A↵ 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

===== Clés publiques compressées

Des clés publiques compressées ont été introduites dans le bitcoin pour réduire la taille des transactions et économiser de l'espace disque sur les nœuds qui stockent la base de données de la blockchain Bitcoin. La plupart des transactions incluent la clé publique, qui est nécessaire pour valider les informations d'identification du propriétaire et dépenser le bitcoin. Chaque clé publique nécessite 520 bits (préfixe + x + y), ce qui, multiplié par plusieurs centaines de transactions par bloc, soit des dizaines de milliers de transactions par jour, ajoute une quantité importante de données à la blockchain.

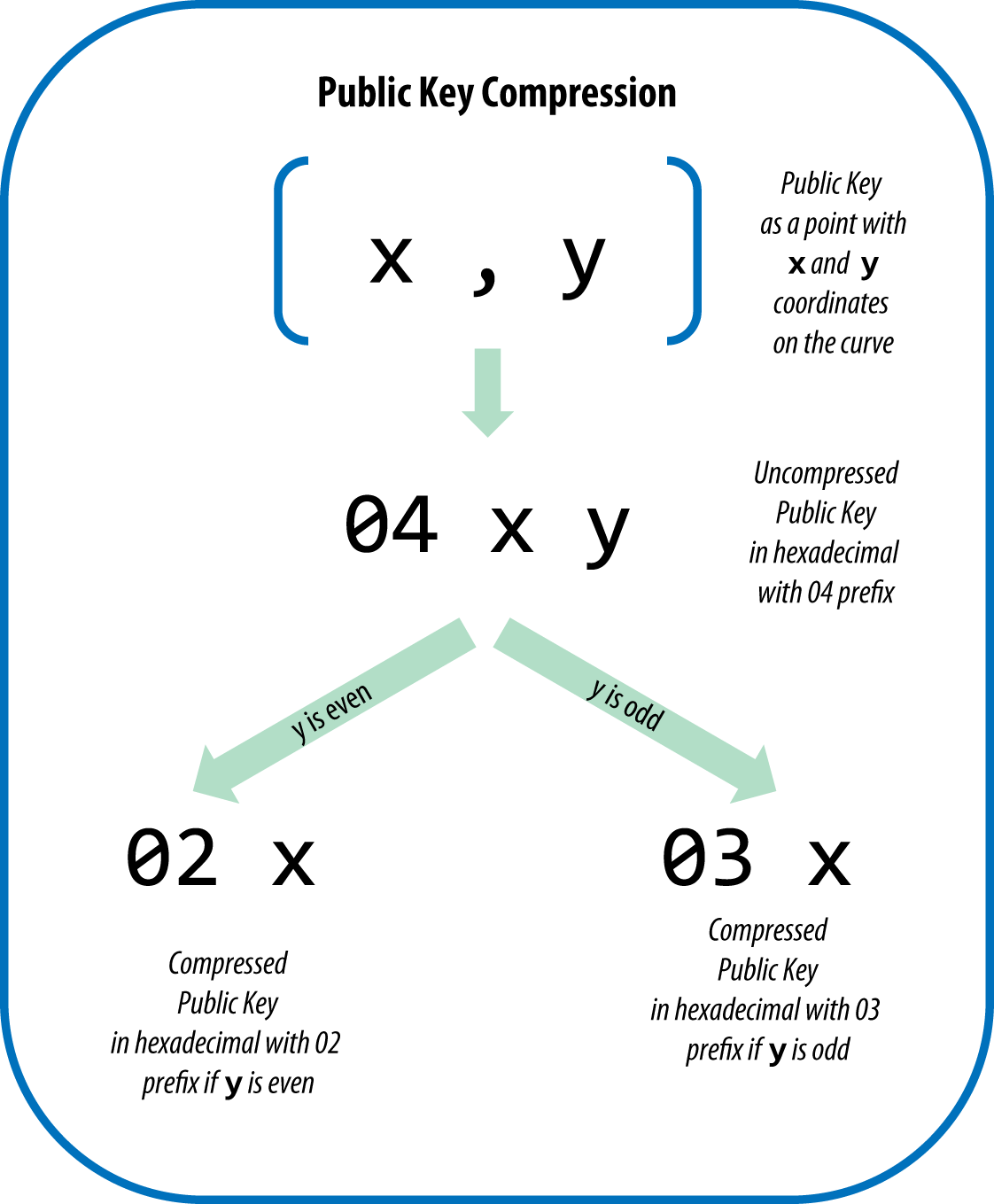
Comme nous l'avons vu dans la section <<pubkey> >, une clé publique est un point (x,y) sur une courbe elliptique. Parce que la courbe exprime une fonction mathématique, un point sur la courbe représente une solution à l'équation et, par conséquent, si nous connaissons la coordonnée *x*, nous pouvons calculer la coordonnée *y* en résolvant l'équation y ^ 2 ^ mod p = (x ^ 3 ^ + 7) mod p. Cela nous permet de stocker uniquement la coordonnée *x* du point de clé publique, en omettant la coordonnée *y* et en réduisant la taille de la clé et l'espace requis pour la stocker de 256 bits. Une réduction de près de 50 % de la taille de chaque transaction représente une grande quantité de données enregistrées au fil du temps !

Alors que les clés publiques non compressées ont un préfixe de 04, les clés publiques compressées commencent par un préfixe 02 ou 03. Regardons pourquoi il y a deux préfixes possibles : parce que le côté gauche de l'équation est *y*2, la solution pour *y* est une racine carrée, qui peut avoir une valeur positive ou négative. Visuellement, cela signifie que la coordonnée *y* résultante peut être au-dessus ou au-dessous de l'axe des x. Comme vous pouvez le voir sur le graphique de la courbe elliptique en <<ecc-curve> >, la courbe est symétrique, c'est-à-dire qu'elle est réfléchie comme un miroir par l'axe des abscisses. Ainsi, bien que nous puissions omettre la coordonnée *y*, nous devons stocker le *sign* de *y* (positif ou négatif) ; ou en d'autres termes, nous devons nous rappeler si c'était au-dessus ou au-dessous de l'axe des x parce que chacune de ces options représente un point différent et une clé publique différente. Lors du calcul de la courbe elliptique en arithmétique binaire sur le corps fini d'ordre premier p, la coordonnée *y* est paire ou impaire, ce qui correspond au signe positif/négatif comme expliqué précédemment. Ainsi, pour distinguer les deux valeurs possibles de *y*, on stocke une clé publique compressée avec le préfixe 02 si le *y* est pair, et 03 s'il est impair, permettant au logiciel de déduire correctement la coordonnée *y* de la *x* coordonne et décompresse la clé publique aux coordonnées complètes du point. La compression de clé publique est illustrée dans <<pubkey\_compression> >.

Voici la même clé publique générée précédemment, présentée sous la forme d'une clé publique compressée stockée sur 264 bits (66 chiffres hexadécimaux) avec le préfixe 03 indiquant que la coordonnée *y* est impaire :

K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

Cette clé publique compressée correspond à la même clé privée, c'est-à-dire qu'elle est générée à partir de la même clé privée. Cependant, il semble différent de la clé publique non compressée. Plus important encore, si nous convertissons cette clé publique compressée en une adresse Bitcoin à l'aide de la fonction de double hachage (RIPEMD160(SHA256(K))), cela produira une adresse Bitcoin *différente*. Cela peut prêter à confusion, car cela signifie qu'une seule clé privée peut produire une clé publique exprimée dans deux formats différents (compressé et non compressé) qui produisent deux adresses Bitcoin différentes. Cependant, la clé privée est identique pour les deux adresses Bitcoin.



Compression de clé publique

Les clés publiques compressées deviennent progressivement la valeur par défaut pour les clients Bitcoin, ce qui a un impact significatif sur la réduction de la taille des transactions et donc de la blockchain. Cependant, tous les clients ne prennent pas encore en charge les clés publiques compressées. Les clients plus récents qui prennent en charge les clés publiques compressées doivent comptabiliser les transactions des clients plus anciens qui ne prennent pas en charge les clés publiques compressées. Ceci est particulièrement important lorsqu'une application de portefeuille importe des clés privées à partir d'une autre application de portefeuille bitcoin, car le nouveau portefeuille doit analyser la blockchain pour trouver les transactions correspondant à ces clés importées. Quelles adresses Bitcoin le portefeuille Bitcoin doit-il rechercher ? Les adresses Bitcoin produites par des clés publiques non compressées, ou les adresses Bitcoin produites par des clés publiques compressées ? Les deux sont des adresses Bitcoin valides et peuvent être signées par la clé privée, mais ce sont des adresses différentes !

Pour résoudre ce problème, lorsque des clés privées sont exportées depuis un portefeuille, le WIF utilisé pour les représenter est implémenté différemment dans les nouveaux portefeuilles Bitcoin, pour indiquer que ces clés privées ont été utilisées pour produire des clés publiques *compressées* et donc des adresses Bitcoin *compressées*. Cela permet au portefeuille importateur de faire la distinction entre les clés privées provenant de portefeuilles plus anciens ou plus récents et de rechercher dans la blockchain des transactions avec des adresses Bitcoin correspondant respectivement aux clés publiques non compressées ou compressées. Voyons comment cela fonctionne plus en détail, dans la section suivante.

===== Clés privées compressées

Ironiquement, le terme "clé privée compressée" est un abus de langage, car lorsqu'une clé privée est exportée au format WIF compressé, elle est en fait un octet *plus long* qu'un clé privée "non compressée". C'est parce que la clé privée a un suffixe d'un octet ajouté (affiché comme 01 en hexadécimal dans <<table\_4-4> >), ce qui signifie que la clé privée provient d'un portefeuille plus récent et ne doit être utilisée que pour produire des clés publiques compressées. Les clés privées ne sont pas elles-mêmes compressées et ne peuvent pas être compressées. Le terme "clé privée compressée" signifie en réalité "clé privée à partir de laquelle seules les clés publiques compressées doivent être dérivées", tandis que "clé privée non compressée" signifie en réalité "clé privée à partir de laquelle seules les clés publiques non compressées doivent être dérivées". Vous devez uniquement faire référence au format d'exportation en tant que "WIF-compressé" ou "WIF" et ne pas faire référence à la clé privée elle-même en tant que "compressée" pour éviter toute confusion supplémentaire.

<<table\_4-4> > affiche la même clé, encodée aux formats WIF et WIF compressé.

Exemple : même clé, différents formats

|  |  |
| --- | --- |
| Formater | Clé privée |
| Hex | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD |
| WIF | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |
| Hex-compressé | 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD01 |
| WIF compressé | KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ |

Notez que le format de clé privée compressée en hexadécimal a un octet supplémentaire à la fin (01 en hexadécimal). Bien que le préfixe de version Base58Check soit le même (0x80) pour les formats WIF et WIF compressés, l'ajout d'un octet à la fin du numéro fait passer le premier caractère de l'encodage Base58 de 5 à *K* ou *L* . Considérez cela comme l'équivalent Base58 de la différence de codage décimal entre le nombre 100 et le nombre 99. Alors que 100 est un chiffre plus long que 99, il a également un préfixe de 1 au lieu d'un préfixe de 9. Lorsque la longueur change, il affecte le préfixe. En Base58, le préfixe 5 se transforme en *K* ou *L* lorsque la longueur du nombre augmente d'un octet.

N'oubliez pas que ces formats ne sont *pas* utilisés de manière interchangeable. Dans un portefeuille plus récent qui implémente des clés publiques compressées, les clés privées ne seront exportées qu'au format WIF compressé (avec un préfixe *K* ou *L*). Si le portefeuille est une implémentation plus ancienne et n'utilise pas de clés publiques compressées, les clés privées ne seront exportées qu'au format WIF (avec un préfixe 5). Le but ici est de signaler au portefeuille qui importe ces clés privées s'il doit rechercher dans la blockchain des clés et des adresses publiques compressées ou non.

Si un portefeuille bitcoin est capable d'implémenter des clés publiques compressées, il les utilisera dans toutes les transactions. Les clés privées du portefeuille seront utilisées pour dériver les points de clé publique sur la courbe, qui seront compressés. Les clés publiques compressées seront utilisées pour produire des adresses Bitcoin et celles-ci seront utilisées dans les transactions. Lors de l'exportation de clés privées à partir d'un nouveau portefeuille qui implémente des clés publiques compressées, le WIF est modifié, avec l'ajout d'un suffixe d'un octet 01 à la clé privée. La clé privée encodée en Base58Check qui en résulte est appelée "WIF compressé" et commence par la lettre *K* ou *L*, au lieu de commencer par "5" comme c'est le cas avec les clés encodées en WIF (non compressées) des portefeuilles plus anciens.

"Clés privées compressées" est un terme impropre ! Ils ne sont pas compressés ; au lieu de cela, WIF-compressé signifie que les clés ne doivent être utilisées que pour dériver des clés publiques compressées et leurs adresses Bitcoin correspondantes. Ironiquement, une clé privée codée "compressée en WIF" est plus longue d'un octet car elle a le suffixe 01 ajouté pour la distinguer d'une clé "non compressée".

=== Implémentation des clés et des adresses dans Cpass :[++]

Regardons le processus complet de création d'une adresse Bitcoin, d'une clé privée, à une clé publique (un point sur la courbe elliptique), à une adresse à double hachage, et enfin, l'encodage Base58Check. Le code C++ dans <<addr\_example> > montre le processus complet étape par étape, de la clé privée à l'adresse Bitcoin encodée en Base58Check. L'exemple de code utilise la bibliothèque libbitcoin introduite dans <<alt\_libraries> > pour certaines fonctions d'assistance.

#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;  
}

Le code utilise une clé privée prédéfinie pour produire la même adresse Bitcoin à chaque exécution, comme indiqué dans <<addr\_example\_run> >.

# Compiler le code addr.cpp  
$ g++ -o addr addr.cpp -std=c++11 $(pkg-config --cflags --libs libbitcoin)  
# Lancer l&#39;exécutable addr  
$ ./adresse  
Clé publique : 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa  
Adresse : 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK

Le code dans <<addr\_example\_run> > produit une adresse Bitcoin (1PRTT...) à partir d'une clé publique *compressée* (voir <<comp\_pub> >). Si vous utilisiez la clé publique non compressée à la place, cela produirait une adresse Bitcoin différente (14K1y...).

=== Implémenter des clés et des adresses en Python

La bibliothèque bitcoin la plus complète en Python est <https://github.com/vbuterin/> pybitcointools[pybitcointools] par Vitalik Buterin. Dans <<key-to-address\_script> >, nous utilisons la bibliothèque pybitcointools (importée en tant que "bitcoin") pour générer et afficher des clés et des adresses dans différents formats.

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  
if (public\_key\_y % 2) == 0:  
 compressed\_prefix = '02'  
else:  
 compressed\_prefix = '03'  
hex\_compressed\_public\_key = compressed\_prefix + bitcoin.encode(public\_key\_x, 16)  
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)

<<key-to-address\_script\_run> > affiche la sortie de l'exécution de ce code.

$ python key-to-address-ecc-example.py  
La clé privée (hex) est :  
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa6  
La clé privée (décimal) est :  
 26563230048437957592232553826663696440606756685920117476832299673293013768870  
La clé privée (WIF) est :  
 5JG9hT3beGTJuUAmCQEmNaxAuMacCTfXuw1R3FCXig23RQHMr4K  
La clé privée compressée (hex) est :  
 3aba4162c7251c891207b747840551a71939b0de081f85c4e44cf7c13e41daa601  
La clé privée (compressée en WIF) est :  
 KyBsPXxTuVD82av65KZkrGrWi5qLMah5SdNq6uftawDbgKa2wv6S  
Les coordonnées de la clé publique (x,y) sont :  
 (41637322786646325214887832269588396900663353932545912953362782457239403430124L,  
 16388935128781238405526710466724741593761085120864331449066658622400339362166L)  
La clé publique (hex) est :  
 045c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec↵  
243bcefdd4347074d44bd7356d6a53c495737dd96295e2a9374bf5f02ebfc176  
La clé publique compressée (hex) est :  
 025c0de3b9c8ab18dd04e3511243ec2952002dbfadc864b9628910169d9b9b00ec  
L&#39;adresse Bitcoin (b58check) est :  
 1thMirt546nngXqyPEz532S8fLwbozud8  
L&#39;adresse Bitcoin compressée (b58check) est :  
 14cxpo3MBCYYWCgF74SWTdcmxipnGUsPw3

<<ec\_math> > est un autre exemple, utilisant la bibliothèque Python ECDSA pour les mathématiques de la courbe elliptique et sans utiliser de bibliothèques bitcoin spécialisées.

import ecdsa  
import os  
from ecdsa.util import string\_to\_number, number\_to\_string  
  
# secp256k1, http://www.oid-info.com/get/1.3.132.0.10  
\_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2FL  
\_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L  
\_b = 0x0000000000000000000000000000000000000000000000000000000000000007L  
\_a = 0x0000000000000000000000000000000000000000000000000000000000000000L  
\_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L  
\_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L  
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():  
 convert\_to\_int = lambda array: int("".join(array).encode("hex"), 16)  
   
 # Collect 256 bits of random data from the OS's cryptographically secure random generator  
 byte\_array = os.urandom(32)  
   
 return convert\_to\_int(byte\_array)  
  
def get\_point\_pubkey(point):  
 if point.y() & 1:  
 key = '03' + '%064x' % point.x()  
 else:  
 key = '02' + '%064x' % point.x()  
 return key.decode('hex')  
  
def get\_point\_pubkey\_uncompressed(point):  
 key = '04' + \  
 '%064x' % point.x() + \  
 '%064x' % point.y()  
 return key.decode('hex')  
  
  
# Generate a new private key.  
secret = random\_secret()  
print "Secret: ", secret  
  
# Get the public key point.  
point = secret \* generator  
print "EC point:", point  
  
print "BTC public key:", get\_point\_pubkey(point).encode("hex")  
  
# 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

<<ec\_math\_run> > affiche la sortie produite en exécutant ce script.

<<ec\_math> > utilise os.urandom, qui reflète un générateur de nombres aléatoires cryptographiquement sécurisé (CSRNG) fourni par le système d'exploitation sous-jacent. Attention : Selon le système d'exploitation, os.urandom peut *ne pas* être implémenté avec une sécurité suffisante ou correctement amorcé et peut *ne pas* être approprié pour générer des clés bitcoin de qualité production.

# Installer le gestionnaire de packages Python PIP  
$ sudo apt-get install python-pip  
# Installer la bibliothèque Python ECDSA  
$ sudo pip installer ecdsa  
# Exécuter le script  
$ python ec-math.py  
Secret : 38090835015954358862481132628887443905906204995912378278060168703580660294000  
POINT EC: (70048853531867179489857750497606966272382583471322935454624595540007269312627, 10526743479589329920209527365803935736021686045542353380)  
Clé publique BTC : 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873

=== Clés et adresses avancées

Dans les sections suivantes, nous examinerons les formes avancées de clés et d'adresses, telles que les clés privées cryptées, les adresses de script et multisignatures, les adresses personnalisées , et portefeuilles en papier.

==== Pay-to-Script Hash (P2SH) et adresses multisig

("Pay-to-Script-Hash (P2SH)", "adresses multisig et")Comme nous le savons, les adresses Bitcoin traditionnelles commencent par le chiffre "1" et sont dérivées de la clé publique, qui est dérivé de la clé privée. Bien que n'importe qui puisse envoyer des bitcoins à une adresse "1", ce bitcoin ne peut être dépensé qu'en présentant la signature de la clé privée et la clé publique correspondantes.

Les adresses Bitcoin qui commencent par le chiffre "3" sont des adresses de hachage de pay-to-script (P2SH), parfois appelées à tort multisignature ou adresses multisignatures. Ils désignent le bénéficiaire d'une transaction bitcoin comme le hachage d'un script, au lieu du propriétaire d'une clé publique. La fonctionnalité a été introduite en janvier 2012 avec BIP-16 (voir <<appdxbitcoinimpproposals> >), et est largement adopté car il offre la possibilité d'ajouter des fonctionnalités à l'adresse elle-même. Contrairement aux transactions qui "envoient" des fonds aux adresses Bitcoin traditionnelles "1", également connues sous le nom de hachage de clé publique (P2PKH), les fonds envoyés aux adresses "3" nécessitent quelque chose de plus que la présentation d'une clé publique et une signature de clé privée comme preuve de propriété. Les exigences sont désignées au moment de la création de l'adresse, dans le script, et toutes les entrées de cette adresse seront encombrées des mêmes exigences.

Une adresse P2SH est créée à partir d'un script de transaction, qui définit qui peut dépenser une sortie de transaction (pour plus de détails, voir <<p2sh> >). L'encodage d'une adresse P2SH consiste à utiliser la même fonction de double hachage que celle utilisée lors de la création d'une adresse Bitcoin, appliquée uniquement sur le script à la place de la clé publique :

hachage de script = RIPEMD160(SHA256(script))

Le "hachage de script" résultant est encodé avec Base58Check avec un préfixe de version de 5, ce qui donne une adresse encodée commençant par un 3. Un exemple d'adresse P2SH est 3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM, qui peut être dérivée à l'aide des commandes Bitcoin Explorer script-encode, sha256, ripemd160 et base58check-encode (voir <<appdx\_bx> >) comme suit :

$ écho \  
&#39;DUP HASH160 [89abcdefabbaabbaabbaabbaabbaabbaabbaabba] EQUALVERIFY CHECKSIG&#39; &gt; script  
$ bx script-encoder &lt; script | bx sha256 | bx mûrmd160 \  
| bx base58check-encode --version 5  
3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM

P2SH n'est pas nécessairement la même chose qu'une transaction standard multisignature. Une adresse P2SH représente *le plus souvent* un script multi-signatures, mais elle peut également représenter un script codant d'autres types de transactions.

===== Adresses multisignatures et P2SH

Actuellement, l'implémentation la plus courante de la fonction P2SH est le script d'adresse multi-signature. Comme son nom l'indique, le script sous-jacent nécessite un nombre minimum de signatures pour prouver la propriété et donc dépenser des fonds. La fonctionnalité de multi-signature bitcoin est conçue pour exiger M signatures (également appelées «seuil») à partir d'un total de N clés, appelées multisig M-of-N, où M est égal ou inférieur à N. Par exemple , Bob le propriétaire du café de <<ch01\_intro\_what\_is\_bitcoin> > pourrait utiliser une adresse multisignature nécessitant 1 sur 2 signatures d'une clé lui appartenant et d'une clé appartenant à son conjoint, garantissant que l'un ou l'autre pourrait signer pour passer une sortie de transaction verrouillée à cette adresse. Cela ressemblerait à un «compte conjoint» tel qu'il est mis en œuvre dans les banques traditionnelles où l'un ou l'autre des conjoints peut dépenser avec une seule signature. Ou Gopesh, le concepteur Web payé par Bob pour créer un site Web, pourrait avoir une adresse multisignature 2 sur 3 pour son entreprise qui garantit qu'aucun fonds ne peut être dépensé à moins qu'au moins deux des partenaires commerciaux ne signent une transaction.

Nous explorerons comment créer des transactions qui dépensent des fonds à partir d'adresses P2SH (et multi-signatures) dans <<transactions> >.

==== Adresses personnalisées

Les adresses personnalisées sont des adresses Bitcoin valides qui contiennent des messages lisibles par l'homme. Par exemple, 1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33 est une adresse valide qui contient les lettres formant le mot "Love" comme les quatre premières lettres Base58. Les adresses personnalisées nécessitent de générer et de tester des milliards de clés privées candidates, jusqu'à ce qu'une adresse Bitcoin avec le modèle souhaité soit trouvée. Bien qu'il y ait quelques optimisations dans l'algorithme de génération de vanité, le processus consiste essentiellement à choisir une clé privée au hasard, à dériver la clé publique, à dériver l'adresse Bitcoin et à vérifier si elle correspond au modèle de vanité souhaité, en répétant des milliards de fois jusqu'à ce qu'un correspondance est trouvée.

Une fois qu'une adresse personnalisée correspondant au modèle souhaité est trouvée, la clé privée dont elle est dérivée peut être utilisée par le propriétaire pour dépenser des bitcoins exactement de la même manière que n'importe quelle autre adresse. Les adresses personnalisées ne sont ni moins ni plus sécurisées que toute autre adresse. Ils dépendent de la même cryptographie à courbe elliptique (ECC) et SHA que toute autre adresse. Vous ne pouvez pas plus facilement trouver la clé privée d'une adresse commençant par un modèle de vanité que vous ne le pouvez de n'importe quelle autre adresse.

Dans <<ch01\_intro\_what\_is\_bitcoin> >, nous avons présenté Eugenia, directrice d'une association caritative pour enfants opérant aux Philippines. Disons qu'Eugenia organise une collecte de fonds en bitcoins et souhaite utiliser une adresse vanity Bitcoin pour faire connaître la collecte de fonds. Eugenia créera une adresse de vanité qui commence par "1Kids" pour promouvoir la collecte de fonds caritative pour les enfants. Voyons comment cette adresse personnalisée sera créée et ce que cela signifie pour la sécurité de l'association caritative d'Eugenia.

===== Génération d'adresses personnalisées

Il est important de réaliser qu'une adresse Bitcoin est simplement un nombre représenté par des symboles dans l'alphabet Base58. La recherche d'un modèle comme "1Kids" peut être considérée comme la recherche d'une adresse comprise entre 1Kids1111111111111111111111111111 et 1Kidszzzzzzzzzzzzzzzzzzzzzzzzzz. Il y a environ 58 ^ 29 ^ (environ 1,4 \* 10 ^ 51 ^) adresses dans cette plage, toutes commençant par "1Kids". <<table\_4-11> > affiche la plage d'adresses qui ont le préfixe 1Kids.

La plage d'adresses personnalisées commençant par "1Kids"

|  |  |
| --- | --- |
| **Depuis** | 1Enfants111111111111111111111111111 |
|  | 1Enfants111111111111111111111111112 |
|  | 1Enfants111111111111111111111111113 |
|  | ... |
| **À** | 1Enfantszzzzzzzzzzzzzzzzzzzzzzzzzzz |

Regardons le modèle "1Kids" comme un nombre et voyons à quelle fréquence nous pourrions trouver ce modèle dans une adresse Bitcoin (voir <<table\_4-12> >). Un ordinateur de bureau moyen, sans aucun matériel spécialisé, peut rechercher environ 100 000 clés par seconde.

La fréquence d'un modèle de vanité (1KidsCharity) et le temps de recherche moyen sur un PC de bureau

|  |  |  |  |
| --- | --- | --- | --- |
| Longueur | Motif | Fréquence | Temps de recherche moyen |
| 1 | 1K | 1 sur 58 touches | < 1 millisecondes |
| 2 | 1Ki | 1 sur 3 364 | 50 millisecondes |
| 3 | 1Enfant | 1 sur 195 000 | < 2 secondes |
| 4 | 1Enfants | 1 sur 11 millions | 1 minute |
| 5 | 1EnfantsC | 1 sur 656 millions | 1 heure |
| 6 | 1EnfantsCh | 1 sur 38 milliards | 2 jours |
| 7 | 1KidsCha | 1 sur 2,2 trillions | 3–4 mois |
| 8 | 1KidsChar | 1 sur 128 billions | 13–18 ans |
| 9 | 1EnfantsChari | 1 sur 7 quadrillions | 800 ans |
| 10 | 1KidsCharit | 1 sur 400 quadrillions | 46 000 ans |
| 11 | 1KidsCharité | 1 sur 23 quintillions | 2,5 millions d'années |

Comme vous pouvez le voir, Eugenia ne créera pas de sitôt l'adresse personnalisée "1KidsCharity", même si elle avait accès à plusieurs milliers d'ordinateurs. Chaque caractère supplémentaire augmente la difficulté d'un facteur de 58. Les modèles de plus de sept caractères sont généralement trouvés par du matériel spécialisé, tel que des ordinateurs de bureau personnalisés avec plusieurs GPU. Ce sont souvent des "plates-formes" de minage de bitcoin réutilisées qui ne sont plus rentables pour le minage de bitcoin mais peuvent être utilisées pour trouver des adresses personnalisées. Les recherches de vanité sur les systèmes GPU sont de plusieurs ordres de grandeur plus rapides que sur un processeur à usage général.

Une autre façon de trouver une adresse de vanité consiste à sous-traiter le travail à un pool de mineurs de vanité, comme le pool sur [Vanity Pool](https://vanitypool.appspot.com). Un pool de ce type est un service qui permet à ceux qui disposent de matériel GPU de gagner des bitcoins en recherchant des adresses personnalisées pour les autres. Pour un petit paiement (0,01 bitcoin ou environ 5 $ au moment d'écrire ces lignes), Eugenia peut externaliser la recherche d'une adresse personnalisée à sept caractères et obtenir des résultats en quelques heures au lieu d'avoir à exécuter une recherche de CPU pendant des mois.

Générer une adresse personnalisée est un exercice de force brute : essayez une clé aléatoire, vérifiez l'adresse résultante pour voir si elle correspond au modèle souhaité, répétez jusqu'à ce que vous réussissiez. <<vanity\_miner\_code> > montre un exemple de "vanity miner", un programme conçu pour trouver des adresses personnalisées, écrit en C++. L'exemple utilise la bibliothèque libbitcoin, que nous avons introduite dans <<alt\_libraries> >.

Mineur d'adresse .Vanity

#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() % std::numeric\_limits<uint8\_t>::max();  
 // 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;  
}

<<vanity\_miner\_code> > utilise std::random\_device. Selon l'implémentation, cela peut refléter un CSRNG fourni par le système d'exploitation sous-jacent. Dans le cas d'un système d'exploitation de type Unix tel que Linux, il tire de /dev/urandom. Le générateur de nombres aléatoires utilisé ici est à des fins de démonstration, et il n'est *pas* approprié pour générer des clés bitcoin de qualité production car il n'est pas implémenté avec une sécurité suffisante.

L'exemple de code doit être compilé à l'aide d'un compilateur C++ et lié à la bibliothèque libbitcoin (qui doit d'abord être installée sur ce système). Pour exécuter l'exemple, lancez l'exécutable vanity-miner sans paramètres (voir <<vanity\_miner\_run> >) et il tentera de trouver une adresse personnalisée commençant par "1kid".

# Compiler le code avec g++  
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)  
# Exécutez l&#39;exemple  
$ ./vanité-mineur  
Adresse de vanité trouvée! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT  
Secret : 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f  
# Exécutez-le à nouveau pour un résultat différent  
$ ./vanité-mineur  
Adresse de vanité trouvée! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn  
Secret : 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623  
# Utilisez &quot;time&quot; pour voir combien de temps il faut pour trouver un résultat  
$ time ./vanity-mineur  
Adresse de vanité trouvée! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM  
Secret : 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349  
  
réel 0m8.868s  
utilisateur 0m8.828s  
système 0m0.035s

L'exemple de code prendra quelques secondes pour trouver une correspondance pour le modèle à trois caractères "kid", comme nous pouvons le voir lorsque nous utilisons la commande Unix time pour mesurer le temps d'exécution. Modifiez le modèle search dans le code source et voyez combien de temps cela prend pour les modèles à quatre ou cinq caractères !

===== Sécurité de l'adresse personnalisée

Les adresses personnalisées peuvent être utilisées pour améliorer *et* pour contourner les mesures de sécurité ; ils sont vraiment une épée à double tranchant. Utilisée pour améliorer la sécurité, une adresse distinctive rend plus difficile pour les adversaires de substituer leur propre adresse et de tromper vos clients en les payant à votre place. Malheureusement, les adresses personnalisées permettent également à quiconque de créer une adresse qui *ressemble* à n'importe quelle adresse aléatoire, ou même à une autre adresse personnalisée, trompant ainsi vos clients.

Eugenia pourrait annoncer une adresse générée aléatoirement (par exemple, 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy) à laquelle les gens peuvent envoyer leurs dons. Ou, elle pourrait générer une adresse de vanité qui commence par 1Kids, pour la rendre plus distinctive.

Dans les deux cas, l'un des risques d'utiliser une seule adresse fixe (plutôt qu'une adresse dynamique distincte par donateur) est qu'un voleur puisse s'infiltrer dans votre site Web et le remplacer par sa propre adresse, détournant ainsi les dons vers lui. Si vous avez annoncé votre adresse de don à plusieurs endroits différents, vos utilisateurs peuvent inspecter visuellement l'adresse avant d'effectuer un paiement pour s'assurer qu'il s'agit de la même adresse qu'ils ont vue sur votre site Web, sur votre e-mail et sur votre dépliant. Dans le cas d'une adresse aléatoire comme 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy, l'utilisateur moyen inspectera peut-être les premiers caractères "1J7mdg" et sera satisfait que l'adresse corresponde. À l'aide d'un générateur d'adresses personnalisées, une personne ayant l'intention de voler en substituant une adresse d'apparence similaire peut rapidement générer des adresses qui correspondent aux premiers caractères, comme indiqué dans <<table\_4-13> >.

Génération d'adresses personnalisées pour correspondre à une adresse aléatoire

|  |  |
| --- | --- |
| **Adresse aléatoire originale** | 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy |
| **Vanité (correspondance à 4 caractères)** | 1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy |
| **Vanité (correspondance à 5 caractères)** | 1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n |
| **Vanité (correspondance à 6 caractères)** | 1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX |

Alors, une adresse personnalisée augmente-t-elle la sécurité ? Si Eugenia génère l'adresse personnalisée 1Kids33q44erFfpeXrmDSz7zEqG2FesZEN, les utilisateurs sont susceptibles de regarder le mot de modèle personnalisé *et quelques caractères au-delà*, par exemple en remarquant la partie "1Kids33" de l'adresse. Cela obligerait un attaquant à générer une adresse de vanité correspondant à au moins six caractères (deux de plus), en déployant un effort 3 364 fois (58 × 58) supérieur à l'effort qu'Eugenia a dépensé pour sa vanité à 4 caractères. Essentiellement, l'effort qu'Eugenia déploie (ou paie une vanité pour) "pousse" l'attaquant à devoir produire une vanité de modèle plus longue. Si Eugenia paie un pool pour générer une adresse de vanité à 8 caractères, l'attaquant serait poussé dans le domaine de 10 caractères, ce qui est irréalisable sur un ordinateur personnel et coûteux même avec une plate-forme d'extraction de vanité personnalisée ou un pool de vanité. Ce qui est abordable pour Eugenia devient inabordable pour l'attaquant, surtout si la récompense potentielle de la fraude n'est pas assez élevée pour couvrir le coût de la génération d'adresse personnalisée.

==== Portefeuilles en papier

Les paper wallets sont des clés privées bitcoin imprimées sur papier. Souvent, le portefeuille papier comprend également l'adresse Bitcoin correspondante pour plus de commodité, mais ce n'est pas nécessaire car elle peut être dérivée de la clé privée.

Les portefeuilles en papier sont une technologie OBSOLÈTE et sont dangereux pour la plupart des utilisateurs. Il existe de nombreux pièges subtils impliqués dans leur génération, notamment la possibilité que le code de génération soit compromis par une "porte dérobée". Des centaines de bitcoins ont été volés de cette façon. Les portefeuilles en papier sont présentés ici à titre informatif uniquement et ne doivent pas être utilisés pour stocker des bitcoins. Utilisez une phrase mnémonique BIP-39 pour sauvegarder vos clés. Utilisez un portefeuille matériel pour stocker des clés et signer des transactions. N'UTILISEZ PAS DE PORTEFEUILLES EN PAPIER.

Les portefeuilles en papier se présentent sous de nombreuses formes, tailles et conceptions, mais à un niveau très basique, il ne s'agit que d'une clé et d'une adresse imprimées sur papier. <<table\_4-14> > montre la forme la plus simple d'un portefeuille en papier.

La forme la plus simple d'un portefeuille papier - une impression de l'adresse Bitcoin et de la clé privée

|  |  |
| --- | --- |
| Adresse publique | Clé privée (WIF) |
| 1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x | 5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn |

Les portefeuilles en papier sont disponibles dans de nombreux modèles et tailles, avec de nombreuses fonctionnalités différentes. <<paper\_wallet\_simple> > montre un exemple de portefeuille en papier.



Un exemple de portefeuille en papier simple

Certains sont destinés à être offerts en cadeau et ont des thèmes saisonniers, tels que les thèmes de Noël et du Nouvel An. D'autres sont conçus pour être stockés dans un coffre-fort de banque ou un coffre-fort avec la clé privée cachée d'une manière ou d'une autre, soit avec des autocollants opaques à gratter, soit pliés et scellés avec une feuille adhésive inviolable.

D'autres conceptions comportent des copies supplémentaires de la clé et de l'adresse, sous la forme de talons détachables similaires aux talons de ticket, vous permettant de stocker plusieurs copies pour vous protéger contre les incendies, les inondations ou d'autres catastrophes naturelles.



Un exemple de portefeuille papier avec des copies supplémentaires des clés sur un "stub" de sauvegarde

== Wallets

The word "wallet" is used to describe a few different things in bitcoin.

At a high level, a wallet is an application that serves as the primary user interface. The wallet controls access to a user’s money, managing keys and addresses, tracking the balance, and creating and signing transactions.

More narrowly, from a programmer’s perspective, the word "wallet" refers to the data structure used to store and manage a user’s keys.

In this chapter we will look at the second meaning, where wallets are containers for private keys, usually implemented as structured files or simple databases.

=== Wallet Technology Overview

In this section we summarize the various technologies used to construct user-friendly, secure, and flexible bitcoin wallets.

A common misconception about bitcoin is that bitcoin wallets contain bitcoin. In fact, the wallet contains only keys. The "coins" are recorded in the blockchain on the Bitcoin network. Users control the coins on the network by signing transactions with the keys in their wallets. In a sense, a bitcoin wallet is a *keychain*.

Bitcoin wallets contain keys, not coins. Each user has a wallet containing keys. Wallets are really keychains containing pairs of private/public keys (see [???](#private_public_keys)). Users sign transactions with the keys, thereby proving they own the transaction outputs (their coins). The coins are stored on the blockchain in the form of transaction outputs (often noted as vout or txout).

There are two primary types of wallets, distinguished by whether the keys they contain are related to each other or not.

The first type is a *nondeterministic wallet*, where each key is independently generated from a random number. The keys are not related to each other. This type of wallet is also known as a JBOK wallet from the phrase "Just a Bunch Of Keys."

The second type of wallet is a *deterministic wallet*, where all the keys are derived from a single master key, known as the *seed*. All the keys in this type of wallet are related to each other and can be generated again if one has the original seed. There are a number of different *key derivation* methods used in deterministic wallets. The most commonly used derivation method uses a tree-like structure and is known as a *hierarchical deterministic* or *HD* wallet.

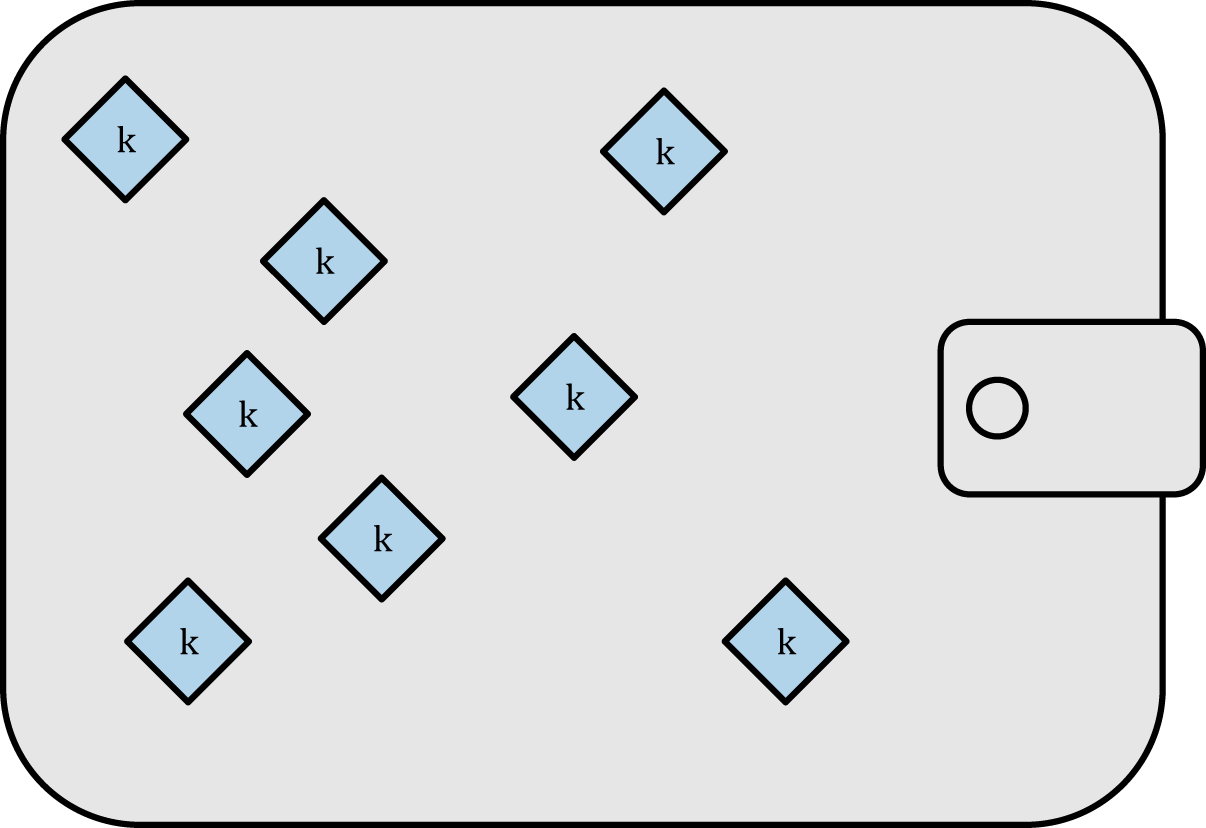
Deterministic wallets are initialized from a random sequence (entropy). To make these easier to use, random sequences are encoded as English words, also known as *mnemonic code words*.

The next few sections introduce each of these technologies at a high level.

==== Nondeterministic (Random) Wallets

In the first bitcoin wallet (now called Bitcoin Core), wallets were collections of randomly generated private keys. For example, the original Bitcoin Core client pregenerates 100 random private keys when first started and generates more keys as needed, using each key only once. Such wallets are being replaced with deterministic wallets because they are cumbersome to manage, back up, and import. The disadvantage of random keys is that if you generate many of them you must keep copies of all of them, meaning that the wallet must be backed up frequently. Each key must be backed up, or the funds it controls are irrevocably lost if the wallet becomes inaccessible. This conflicts directly with the principle of avoiding address reuse, by using each Bitcoin address for only one transaction. Address reuse reduces privacy by associating multiple transactions and addresses with each other. A Type-0 nondeterministic wallet is a poor choice of wallet, especially if you want to avoid address reuse because it means managing many keys, which creates the need for frequent backups. Although the Bitcoin Core client includes a Type-0 wallet, using this wallet is discouraged by developers of Bitcoin Core. [figure\_title](#Type0_wallet) shows a nondeterministic wallet, containing a loose collection of random keys.

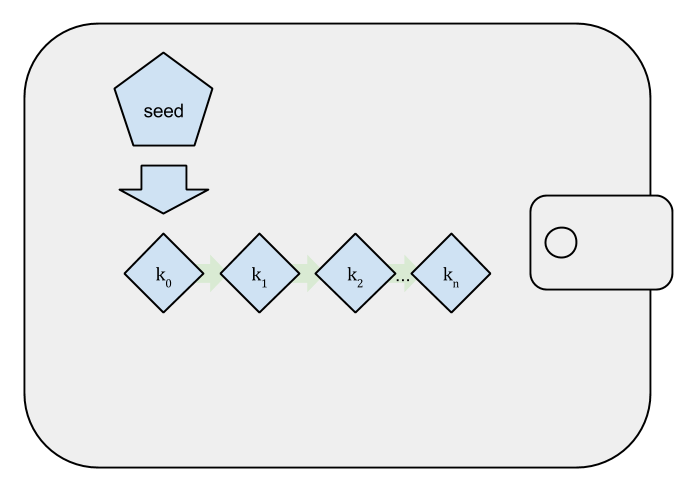
The use of nondeterministic wallets is discouraged for anything other than simple tests. They are simply too cumbersome to back up and use. Instead, use an industry-standard–based *HD wallet* with a *mnemonic* random sequence (entropy, or "initial seed") for backup.



Type-0 nondeterministic (random) wallet: a collection of randomly generated keys

==== Deterministic (Seeded) Wallets

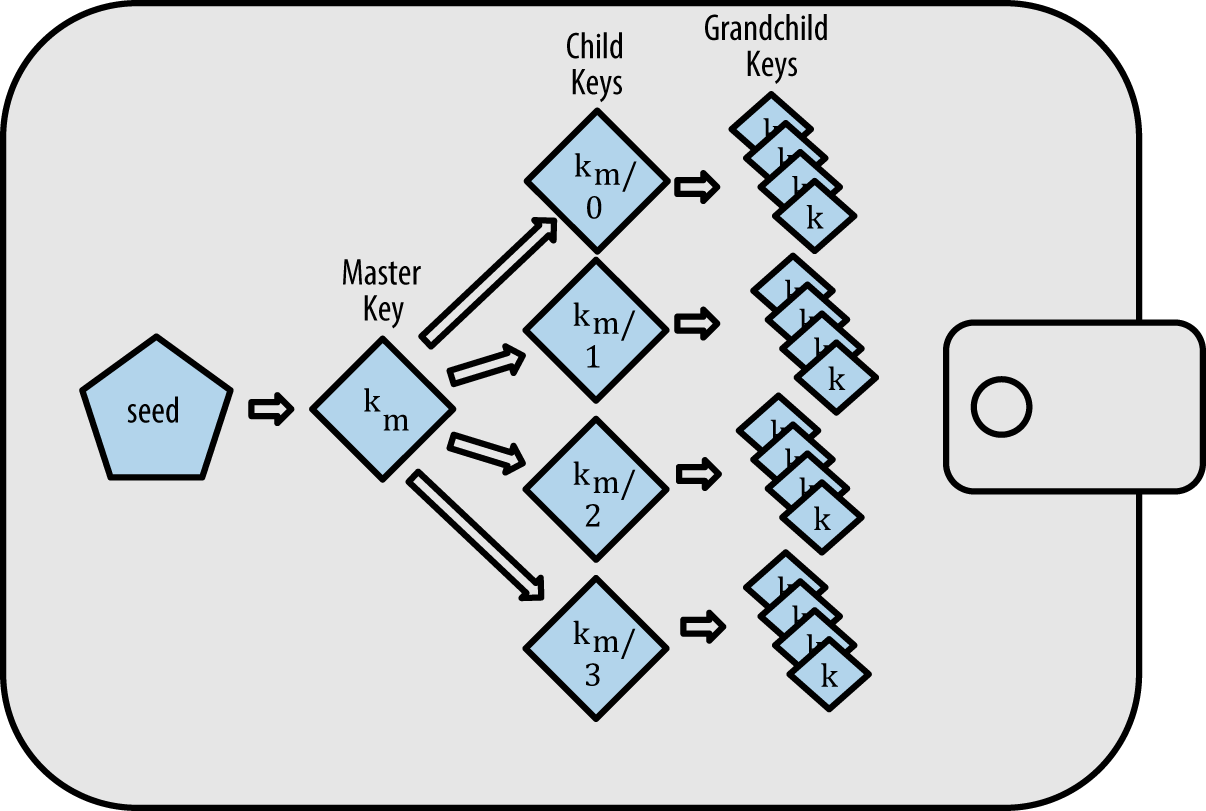
Deterministic, or "seeded," wallets are wallets that contain private keys that are all derived from a common seed, through the use of a one-way hash function. The seed is a randomly generated number that is combined with other data, such as an index number or "chain code" (see [simpara\_title](#hd_wallets)) to derive the private keys. In a deterministic wallet, the seed is sufficient to recover all the derived keys, and therefore a single backup at creation time is sufficient. The seed is also sufficient for a wallet export or import, allowing for easy migration of all the user’s keys between different wallet implementations. [figure\_title](#Type1_wallet) shows a logical diagram of a deterministic wallet.



Type-1 deterministic (seeded) wallet: a deterministic sequence of keys derived from a seed

==== HD Wallets (BIP-32/BIP-44)

Deterministic wallets were developed to make it easy to derive many keys from a single "seed". The most advanced form of deterministic wallets is the HD wallet defined by the BIP-32 standard. HD wallets contain keys derived in a tree structure, such that a parent key can derive a sequence of children keys, each of which can derive a sequence of grandchildren keys, and so on, to an infinite depth. This tree structure is illustrated in [figure\_title](#Type2_wallet).



Type-2 HD wallet: a tree of keys generated from a single seed

HD wallets offer two major advantages over random (nondeterministic) keys. First, the tree structure can be used to express additional organizational meaning, such as when a specific branch of subkeys is used to receive incoming payments and a different branch is used to receive change from outgoing payments. Branches of keys can also be used in corporate settings, allocating different branches to departments, subsidiaries, specific functions, or accounting categories.

The second advantage of HD wallets is that users can create a sequence of public keys without having access to the corresponding private keys. This allows HD wallets to be used on an insecure server or in a receive-only capacity, issuing a different public key for each transaction. The public keys do not need to be preloaded or derived in advance, yet the server doesn’t have the private keys that can spend the funds.

==== Seeds and Mnemonic Codes (BIP-39)

HD wallets are a very powerful mechanism for managing many keys and addresses. They are even more useful if they are combined with a standardized way of creating seeds from a sequence of English words that are easy to transcribe, export, and import across wallets. This is known as a *mnemonic* and the standard is defined by BIP-39. Today, most bitcoin wallets (as well as wallets for other cryptocurrencies) use this standard and can import and export seeds for backup and recovery using interoperable mnemonics.

Let’s look at this from a practical perspective. Which of the following seeds is easier to transcribe, record on paper, read without error, export, and import into another wallet?

**A seed for a deterministic wallet, in hex**

0C1E24E5917779D297E14D45F14E1A1A

**A seed for a deterministic wallet, from a 12-word mnemonic**

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

==== Wallet Best Practices

As bitcoin wallet technology has matured, certain common industry standards have emerged that make bitcoin wallets broadly interoperable, easy to use, secure, and flexible. These common standards are:

* Mnemonic code words, based on BIP-39
* HD wallets, based on BIP-32
* Multipurpose HD wallet structure, based on BIP-43
* Multicurrency and multiaccount wallets, based on BIP-44

These standards may change or may become obsolete by future developments, but for now they form a set of interlocking technologies that have become the de facto wallet standard for bitcoin.

The standards have been adopted by a broad range of software and hardware bitcoin wallets, making all these wallets interoperable. A user can export a mnemonic generated on one of these wallets and import it in another wallet, recovering all transactions, keys, and addresses.

Some example of software wallets supporting these standards include (listed alphabetically) Bluewallet, Breadwallet, Copay, and Multibit HD. Examples of hardware wallets supporting these standards include (listed alphabetically) KeepKey, Ledger, and Trezor.

The following sections examine each of these technologies in detail.

If you are implementing a bitcoin wallet, it should be built as a HD wallet, with a seed derived from, and encoded as, a mnemonic code for backup, following the BIP-32, BIP-39, BIP-43, and BIP-44 standards, as described in the following sections.

==== Using a Bitcoin Wallet

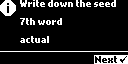
In [Utilisations, utilisateurs et histoires de Bitcoin](#user-stories) we introduced Gabriel, an enterprising young teenager in Rio de Janeiro, who is running a simple web store that sells bitcoin-branded t-shirts, coffee mugs, and stickers.

Gabriel uses a Trezor bitcoin hardware wallet ([figure\_title](#a_trezor_device)) to securely manage his bitcoin. The Trezor is a simple USB device with two buttons that stores keys (in the form of an HD wallet) and signs transactions. Trezor wallets implement all the industry standards discussed in this chapter, so Gabriel is not reliant on any proprietary technology or single vendor solution.



A Trezor device: a bitcoin HD wallet in hardware

When Gabriel used the Trezor for the first time, the device generated a random sequence (entropy), the associated mnemonic and derived a seed from a built-in hardware random number generator. During this initialization phase, the wallet displayed a numbered sequence of words, one by one, on the screen (see [figure\_title](#trezor_mnemonic_display)).



Trezor displaying one of the mnemonic words

By writing down this mnemonic, Gabriel created a backup (see [table\_title](#mnemonic_paper_backup)) that can be used for recovery in the case of loss or damage to the Trezor device. This mnemonic can be used for recovery in a new Trezor or in any one of the many compatible software or hardware wallets. Note that the sequence of words is important, so mnemonic paper backups have numbered spaces for each word. Gabriel had to carefully record each word in the numbered space to preserve the correct sequence.

Gabriel’s paper backup of the mnemonic

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

A 12-word mnemonic is shown in [table\_title](#mnemonic_paper_backup), for simplicity. In fact, most hardware wallets generate a more secure 24-word mnemonic. The mnemonic is used in exactly the same way, regardless of length.

For the first implementation of his web store, Gabriel uses a single Bitcoin address, generated on his Trezor device. This single address is used by all customers for all orders. As we will see, this approach has some drawbacks and can be improved upon with an HD wallet.

=== Wallet Technology Details

Let’s now examine each of the important industry standards that are used by many bitcoin wallets in detail.

==== Mnemonic Code Words (BIP-39)

Mnemonic code words are word sequences that represent (encode) a random number used as a seed to derive a deterministic wallet. The sequence of words is sufficient to re-create the seed and from there re-create the wallet and all the derived keys. A wallet application that implements deterministic wallets with mnemonic words will show the user a sequence of 12 to 24 words when first creating a wallet. That sequence of words is the wallet backup and can be used to recover and re-create all the keys in the same or any compatible wallet application. Mnemonic words make it easier for users to back up wallets because they are easy to read and correctly transcribe, as compared to a random sequence of numbers.

Mnemonic words are often confused with "brainwallets." They are not the same. The primary difference is that a brainwallet consists of words chosen by the user, whereas mnemonic words are created randomly by the wallet and presented to the user. This important difference makes mnemonic words much more secure, because humans are very poor sources of randomness.

Mnemonic codes are defined in BIP-39 (see [simpara\_title](#appdxbitcoinimpproposals)). Note that BIP-39 is one implementation of a mnemonic code standard. There is a different standard, with a different set of words, used by the Electrum wallet and predating BIP-39. BIP-39 was proposed by the company behind the Trezor hardware wallet and is incompatible with Electrum’s implementation. However, BIP-39 has now achieved broad industry support across dozens of interoperable implementations and should be considered the de facto industry standard.

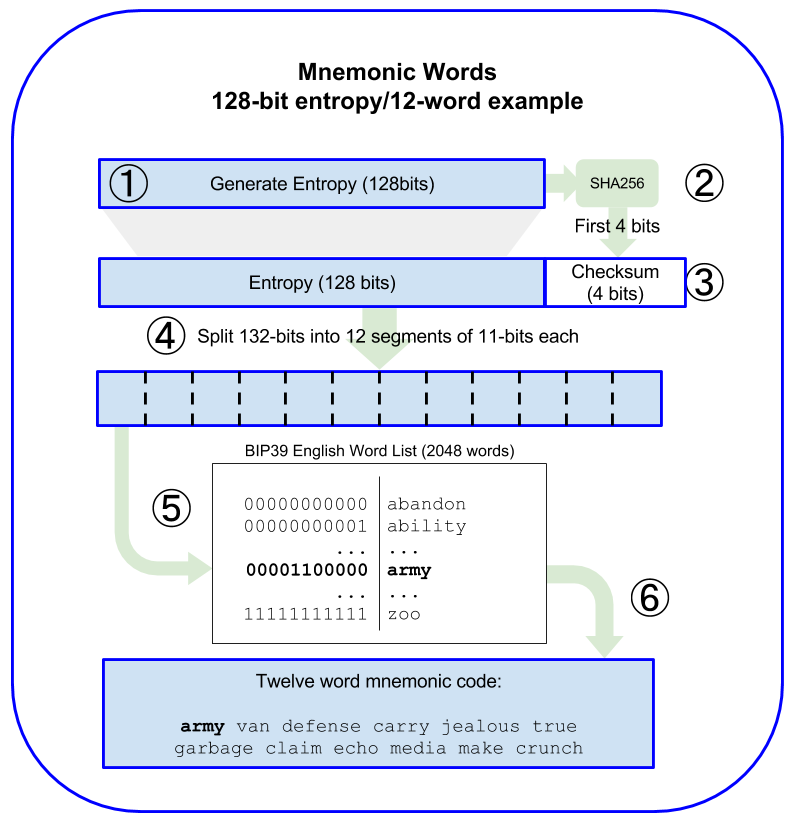
BIP-39 defines the creation of a mnemonic code and seed, which we describe here in nine steps. For clarity, the process is split into two parts: steps 1 through 6 are shown in [simpara\_title](#generating_mnemonic_words) and steps 7 through 9 are shown in [simpara\_title](#mnemonic_to_seed).

===== Generating mnemonic words

Mnemonic words are generated automatically by the wallet using the standardized process defined in BIP-39. The wallet starts from a source of entropy, adds a checksum, and then maps the entropy to a word list:

1. Create a random sequence (entropy) of 128 to 256 bits.
2. Create a checksum of the random sequence by taking the first (entropy-length/32) bits of its SHA256 hash.
3. Add the checksum to the end of the random sequence.
4. Split the result into 11-bit length segments.
5. Map each 11-bit value to a word from the predefined dictionary of 2048 words.
6. The mnemonic code is the sequence of words.

[figure\_title](#generating_entropy_and_encoding) shows how entropy is used to generate mnemonic words.



Generating entropy and encoding as mnemonic words

[table\_title](#table_4-5) shows the relationship between the size of the entropy data and the length of mnemonic codes in words.

Mnemonic codes: entropy and word length

| Entropy (bits) | Checksum (bits) | Entropy **+** checksum (bits) | Mnemonic length (words) |
| --- | --- | --- | --- |
| 128 | 4 | 132 | 12 |
| 160 | 5 | 165 | 15 |
| 192 | 6 | 198 | 18 |
| 224 | 7 | 231 | 21 |
| 256 | 8 | 264 | 24 |

===== From mnemonic to seed

The mnemonic words represent entropy with a length of 128 to 256 bits. The entropy is then used to derive a longer (512-bit) seed through the use of the key-stretching function PBKDF2. The seed produced is then used to build a deterministic wallet and derive its keys.

The key-stretching function takes two parameters: the mnemonic and a *salt*. The purpose of a salt in a key-stretching function is to make it difficult to build a lookup table enabling a brute-force attack. In the BIP-39 standard, the salt has another purpose—it allows the introduction of a passphrase that serves as an additional security factor protecting the seed, as we will describe in more detail in [simpara\_title](#mnemonic_passphrase).

The process described in steps 7 through 9 continues from the process described previously in [simpara\_title](#generating_mnemonic_words):

The first parameter to the PBKDF2 key-stretching function is the

mnemonic

produced from step 6.

The second parameter to the PBKDF2 key-stretching function is a

salt

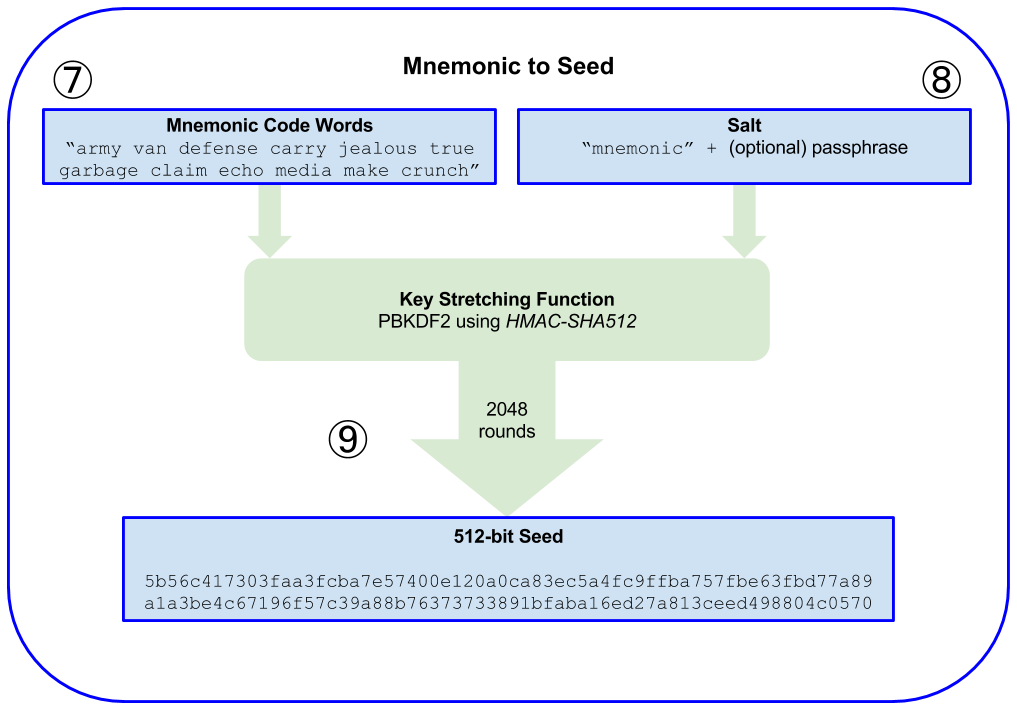
. The salt is composed of the string constant "

mnemonic

" concatenated with an optional user-supplied passphrase string.

PBKDF2 stretches the mnemonic and salt parameters using 2048 rounds of hashing with the HMAC-SHA512 algorithm, producing a 512-bit value as its final output. That 512-bit value is the seed.

[figure\_title](#fig_5_7) shows how a mnemonic is used to generate a seed.



From mnemonic to seed

The key-stretching function, with its 2048 rounds of hashing, is a very effective protection against brute-force attacks against the mnemonic or the passphrase. It makes it extremely costly (in computation) to try more than a few thousand passphrase and mnemonic combinations, while the number of possible derived seeds is vast (2512).

Tables #mnemonic\_128\_no\_pass, #mnemonic\_128\_w\_pass, and #mnemonic\_256\_no\_pass show some examples of mnemonic codes and the seeds they produce (either with or without a passphrase).

128-bit entropy mnemonic code, no passphrase, resulting seed

|  |  |
| --- | --- |
| **Entropy input (128 bits)** | 0c1e24e5917779d297e14d45f14e1a1a |
| **Mnemonic (12 words)** | army van defense carry jealous true garbage claim echo media make crunch |
| **Passphrase** | (none) |
| **Seed (512 bits)** | 5b56c417303faa3fcba7e57400e120a0ca83ec5a4fc9ffba757fbe63fbd77a89a1a3be4c67196f57c39 a88b76373733891bfaba16ed27a813ceed498804c0570 |

128-bit entropy mnemonic code, with passphrase, resulting seed

|  |  |
| --- | --- |
| **Entropy input (128 bits)** | 0c1e24e5917779d297e14d45f14e1a1a |
| **Mnemonic (12 words)** | army van defense carry jealous true garbage claim echo media make crunch |
| **Passphrase** | SuperDuperSecret |
| **Seed (512 bits)** | 3b5df16df2157104cfdd22830162a5e170c0161653e3afe6c88defeefb0818c793dbb28ab3ab091897d0 715861dc8a18358f80b79d49acf64142ae57037d1d54 |

256-bit entropy mnemonic code, no passphrase, resulting seed

|  |  |
| --- | --- |
| **Entropy input (256 bits)** | 2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c |
| **Mnemonic (24 words)** | cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige |
| **Passphrase** | (none) |
| **Seed (512 bits)** | 3269bce2674acbd188d4f120072b13b088a0ecf87c6e4cae41657a0bb78f5315b33b3a04356e53d062e5 5f1e0deaa082df8d487381379df848a6ad7e98798404 |

Many wallets do not allow for the creation of wallets with more than a 12 word mnemonic phrase. You will notice from the tables above that despite the unique lengths of entropy input, the seed size remains the same (512 bits). From a security perspective, the amount of entropy actually used for the production of HD wallets is roughly 128 bits, which equals 12 words. Providing more than 12 words produces additional entropy which is unnecessary, and this *unused* entropy is not used for the derivation of the seed in the way that one might initially suspect. From a usability perspective, 12 words is also easier to write down, back up, and store.

===== Optional passphrase in BIP-39

The BIP-39 standard allows the use of an optional passphrase in the derivation of the seed. If no passphrase is used, the mnemonic is stretched with a salt consisting of the constant string "mnemonic", producing a specific 512-bit seed from any given mnemonic. If a passphrase is used, the stretching function produces a *different* seed from that same mnemonic. In fact, given a single mnemonic, every possible passphrase leads to a different seed. Essentially, there is no "wrong" passphrase. All passphrases are valid and they all lead to different seeds, forming a vast set of possible uninitialized wallets. The set of possible wallets is so large (2512) that there is no practical possibility of brute-forcing or accidentally guessing one that is in use.

There are no "wrong" passphrases in BIP-39. Every passphrase leads to some wallet, which unless previously used will be empty.

The optional passphrase creates two important features:

* A second factor (something memorized) that makes a mnemonic useless on its own, protecting mnemonic backups from compromise by a thief.
* A form of plausible deniability or "duress wallet," where a chosen passphrase leads to a wallet with a small amount of funds used to distract an attacker from the "real" wallet that contains the majority of funds.

However, it is important to note that the use of a passphrase also introduces the risk of loss:

* If the wallet owner is incapacitated or dead and no one else knows the passphrase, the mnemonic code is useless and all the funds stored in the wallet are lost forever.
* Conversely, if the owner backs up the passphrase in the same place as the mnemonic code, it defeats the purpose of a second factor.

While passphrases are very useful, they should only be used in combination with a carefully planned process for backup and recovery, considering the possibility of surviving the owner and allowing his or her family to recover the cryptocurrency estate.

===== Working with mnemonic codes

BIP-39 is implemented as a library in many different programming languages:

[python-mnemonic](https://github.com/trezor/python-mnemonic)

The reference implementation of the standard by the SatoshiLabs team that proposed BIP-39, in Python

[bitcoinjs/bip39](https://github.com/bitcoinjs/bip39)

An implementation of BIP-39, as part of the popular bitcoinJS framework, in JavaScript

[libbitcoin/mnemonic](https://github.com/libbitcoin/libbitcoin/blob/master/src/wallet/mnemonic.cpp)

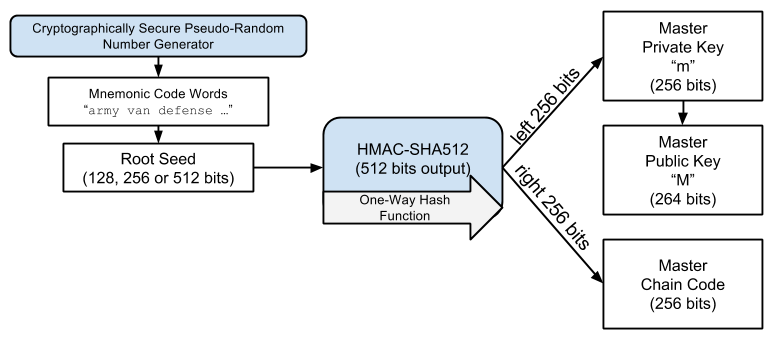
An implementation of BIP-39, as part of the popular Libbitcoin framework, in C++

==== Creating an HD Wallet from the Seed

HD wallets are created from a single *root seed*, which is a 128-, 256-, or 512-bit random number. Most commonly, this seed is generated from a *mnemonic* as detailed in the previous section.

Every key in the HD wallet is deterministically derived from this root seed, which makes it possible to re-create the entire HD wallet from that seed in any compatible HD wallet. This makes it easy to back up, restore, export, and import HD wallets containing thousands or even millions of keys by simply transferring only the mnemonic that the root seed is derived from.

The process of creating the master keys and master chain code for an HD wallet is shown in [figure\_title](#HDWalletFromSeed).



Creating master keys and chain code from a root seed

The root seed is input into the HMAC-SHA512 algorithm and the resulting hash is used to create a *master private key* (m) and a *master chain code* (c).

The master private key (m) then generates a corresponding master public key (M) using the normal elliptic curve multiplication process m \* G that we saw in [simpara\_title](#pubkey).

The chain code (c) is used to introduce entropy in the function that creates child keys from parent keys, as we will see in the next section.

===== Private child key derivation

HD wallets use a *child key derivation* (CKD) function to derive child keys from parent keys.

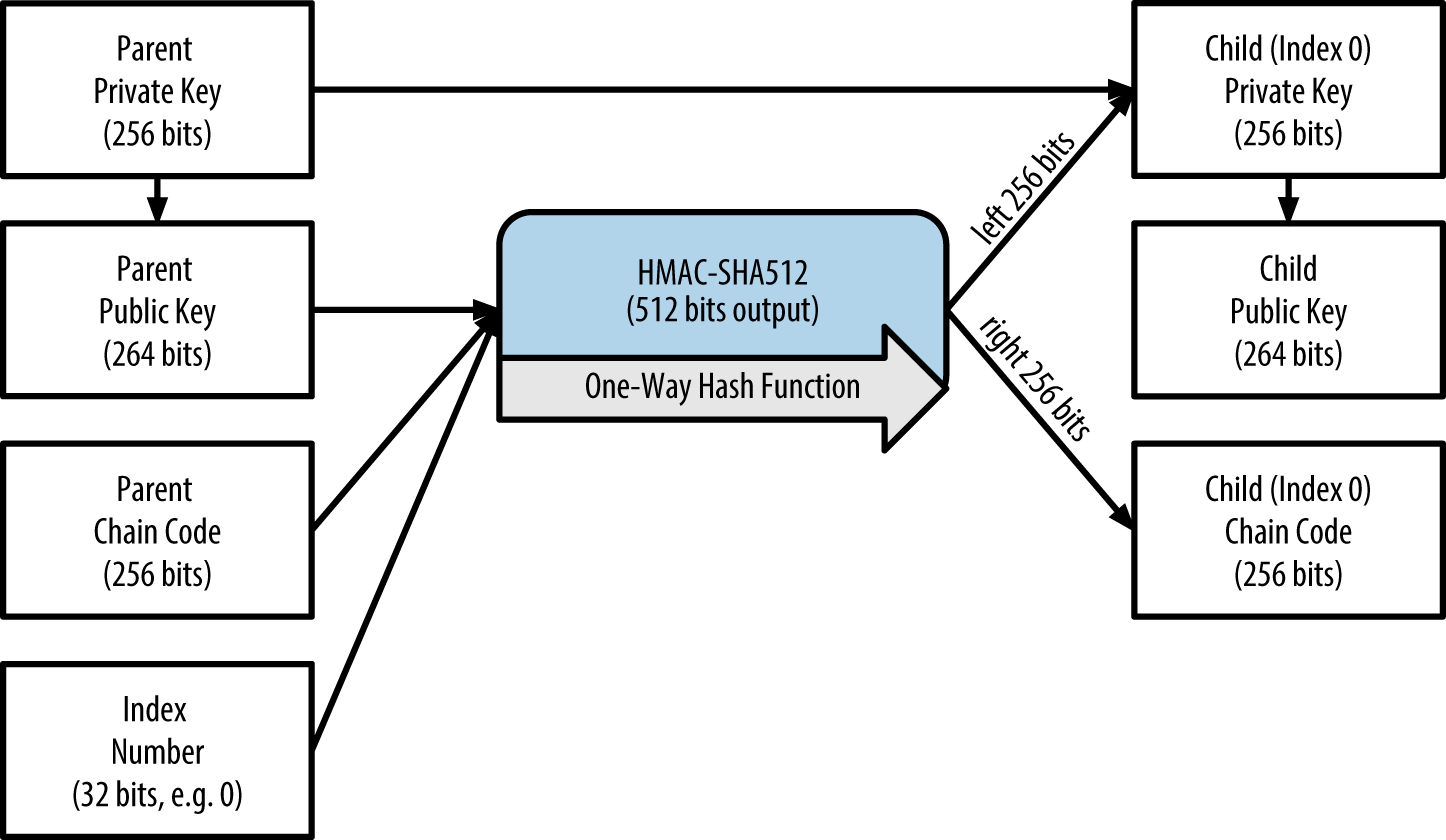
The child key derivation functions are based on a one-way hash function that combines:

* A parent private or public key (ECDSA compressed key)
* A seed called a chain code (256 bits)
* An index number (32 bits)

The chain code is used to introduce deterministic random data to the process, so that knowing the index and a child key is not sufficient to derive other child keys. Knowing a child key does not make it possible to find its siblings, unless you also have the chain code. The initial chain code seed (at the root of the tree) is made from the seed, while subsequent child chain codes are derived from each parent chain code.

These three items (parent key, chain code, and index) are combined and hashed to generate children keys, as follows.

The parent public key, chain code, and the index number are combined and hashed with the HMAC-SHA512 algorithm to produce a 512-bit hash. This 512-bit hash is split into two 256-bit halves. The right-half 256 bits of the hash output become the chain code for the child. The left-half 256 bits of the hash are added to the parent key to produce the child private key. In [figure\_title](#CKDpriv), we see this illustrated with the index set to 0 to produce the "zero" (first by index) child of the parent.



Extending a parent private key to create a child private key

Changing the index allows us to extend the parent and create the other children in the sequence, e.g., Child 0, Child 1, Child 2, etc. Each parent key can have 2,147,483,647 (231) children (231 is half of the entire 232 range available because the other half is reserved for a special type of derivation we will talk about later in this chapter).

Repeating the process one level down the tree, each child can in turn become a parent and create its own children, in an infinite number of generations.

===== Using derived child keys

Child private keys are indistinguishable from nondeterministic (random) keys. Because the derivation function is a one-way function, the child key cannot be used to find the parent key. The child key also cannot be used to find any siblings. If you have the nth child, you cannot find its siblings, such as the n–1 child or the n+1 child, or any other children that are part of the sequence. Only the parent key and chain code can derive all the children. Without the child chain code, the child key cannot be used to derive any grandchildren either. You need both the child private key and the child chain code to start a new branch and derive grandchildren.

So what can the child private key be used for on its own? It can be used to make a public key and a Bitcoin address. Then, it can be used to sign transactions to spend anything paid to that address.

A child private key, the corresponding public key, and the Bitcoin address are all indistinguishable from keys and addresses created randomly. The fact that they are part of a sequence is not visible outside of the HD wallet function that created them. Once created, they operate exactly as "normal" keys.

===== Extended keys

As we saw earlier, the key derivation function can be used to create children at any level of the tree, based on the three inputs: a key, a chain code, and the index of the desired child. The two essential ingredients are the key and chain code, and combined these are called an *extended key*. The term "extended key" could also be thought of as "extensible key" because such a key can be used to derive children.

Extended keys are stored and represented simply as the concatenation of the 256-bit key and 256-bit chain code into a 512-bit sequence. There are two types of extended keys. An extended private key is the combination of a private key and chain code and can be used to derive child private keys (and from them, child public keys). An extended public key is a public key and chain code, which can be used to create child public keys (*public only*), as described in [simpara\_title](#public_key_derivation).

Think of an extended key as the root of a branch in the tree structure of the HD wallet. With the root of the branch, you can derive the rest of the branch. The extended private key can create a complete branch, whereas the extended public key can *only* create a branch of public keys.

An extended key consists of a private or public key and chain code. An extended key can create children, generating its own branch in the tree structure. Sharing an extended key gives access to the entire branch.

Extended keys are encoded using Base58Check, to easily export and import between different BIP-32–compatible wallets. The Base58Check coding for extended keys uses a special version number that results in the prefix "xprv" and "xpub" when encoded in Base58 characters to make them easily recognizable. Because the extended key is 512 or 513 bits, it is also much longer than other Base58Check-encoded strings we have seen previously.

Here’s an example of an extended *private* key, encoded in Base58Check:

xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMKUga5biW6Hx4tws2six3b9c

Here’s the corresponding extended *public* key, encoded in Base58Check:

xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMstweyLXhRgPxdp14sk9tJPW9

===== Public child key derivation

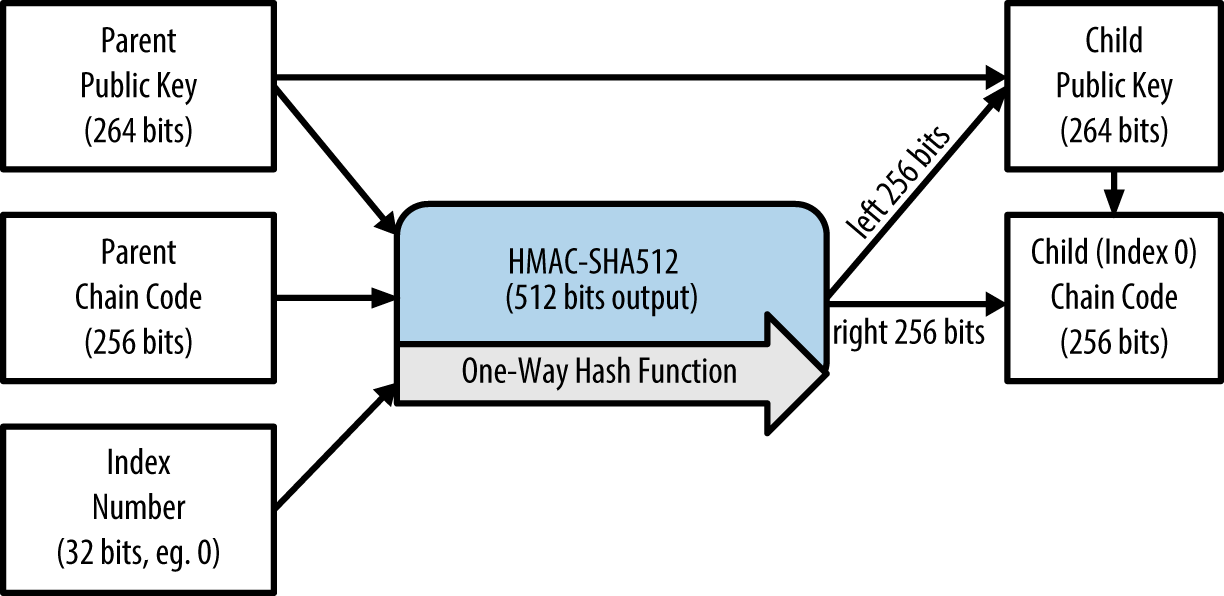
As mentioned previously, a very useful characteristic of HD wallets is the ability to derive public child keys from public parent keys, *without* having the private keys. This gives us two ways to derive a child public key: either from the child private key, or directly from the parent public key.

An extended public key can be used, therefore, to derive all of the *public* keys (and only the public keys) in that branch of the HD wallet structure.

This shortcut can be used to create very secure public key–only deployments where a server or application has a copy of an extended public key and no private keys whatsoever. That kind of deployment can produce an infinite number of public keys and Bitcoin addresses, but cannot spend any of the money sent to those addresses. Meanwhile, on another, more secure server, the extended private key can derive all the corresponding private keys to sign transactions and spend the money.

One common application of this solution is to install an extended public key on a web server that serves an ecommerce application. The web server can use the public key derivation function to create a new Bitcoin address for every transaction (e.g., for a customer shopping cart). The web server will not have any private keys that would be vulnerable to theft. Without HD wallets, the only way to do this is to generate thousands of Bitcoin addresses on a separate secure server and then preload them on the ecommerce server. That approach is cumbersome and requires constant maintenance to ensure that the ecommerce server doesn’t "run out" of addresses.

Another common application of this solution is for cold-storage or hardware wallets. In that scenario, the extended private key can be stored on a paper wallet or hardware device (such as a Trezor hardware wallet), while the extended public key can be kept online. The user can create "receive" addresses at will, while the private keys are safely stored offline. To spend the funds, the user can use the extended private key on an offline signing Bitcoin client or sign transactions on the hardware wallet device (e.g., Trezor). [figure\_title](#CKDpub) illustrates the mechanism for extending a parent public key to derive child public keys.

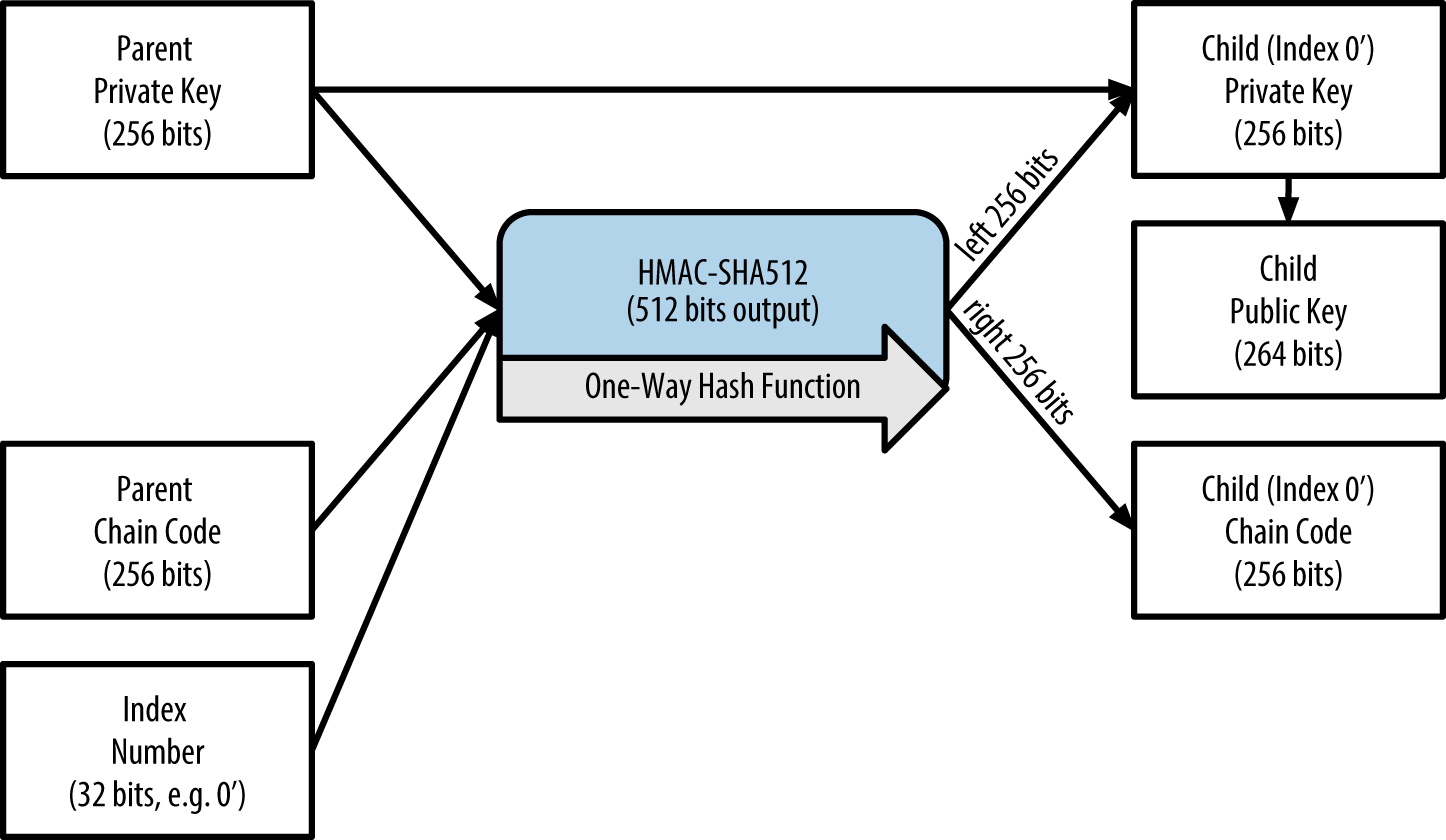


Extending a parent public key to create a child public key

===== Hardened child key derivation

The ability to derive a branch of public keys from an xpub is very useful, but it comes with a potential risk. Access to an xpub does not give access to child private keys. However, because the xpub contains the chain code, if a child private key is known, or somehow leaked, it can be used with the chain code to derive all the other child private keys. A single leaked child private key, together with a parent chain code, reveals all the private keys of all the children. Worse, the child private key together with a parent chain code can be used to deduce the parent private key.

To counter this risk, HD wallets use an alternative derivation function called *hardened derivation*, which "breaks" the relationship between parent public key and child chain code. The hardened derivation function uses the parent private key to derive the child chain code, instead of the parent public key. This creates a "firewall" in the parent/child sequence, with a chain code that cannot be used to compromise a parent or sibling private key. The hardened derivation function looks almost identical to the normal child private key derivation, except that the parent private key is used as input to the hash function, instead of the parent public key, as shown in the diagram in [figure\_title](#CKDprime).



Hardened derivation of a child key; omits the parent public key

When the hardened private derivation function is used, the resulting child private key and chain code are completely different from what would result from the normal derivation function. The resulting "branch" of keys can be used to produce extended public keys that are not vulnerable, because the chain code they contain cannot be exploited to reveal any private keys. Hardened derivation is therefore used to create a "gap" in the tree above the level where extended public keys are used.

In simple terms, if you want to use the convenience of an xpub to derive branches of public keys, without exposing yourself to the risk of a leaked chain code, you should derive it from a hardened parent key, rather than a normal (non-hardened) parent key. As a best practice, the level-1 children of the master keys are always derived through the hardened derivation, to prevent compromise of the master keys.

===== Index numbers for normal and hardened derivation

The index number used in the derivation function is a 32-bit integer. To easily distinguish between keys derived through the normal derivation function versus keys derived through hardened derivation, this index number is split into two ranges. Index numbers between 0 and 231–1 (0x0 to 0x7FFFFFFF) are used *only* for normal derivation. Index numbers between 231 and 232–1 (0x80000000 to 0xFFFFFFFF) are used *only* for hardened derivation. Therefore, if the index number is less than 231, the child is normal, whereas if the index number is equal or above 231, the child is hardened.

To make the index number easier to read and display, the index number for hardened children is displayed starting from zero, but with a prime symbol. The first normal child key is therefore displayed as 0, whereas the first hardened child (index 0x80000000) is displayed as 0&#x27;. In sequence then, the second hardened key would have index 0x80000001 and would be displayed as 1&#x27;, and so on. When you see an HD wallet index i&#x27;, that means 231+i.

===== HD wallet key identifier (path)

Keys in an HD wallet are identified using a "path" naming convention, with each level of the tree separated by a slash (/) character (see [table\_title](#table_4-8)). Private keys derived from the master private key start with "m." Public keys derived from the master public key start with "M." Therefore, the first child private key of the master private key is m/0. The first child public key is M/0. The second grandchild of the first child is m/0/1, and so on.

The "ancestry" of a key is read from right to left, until you reach the master key from which it was derived. For example, identifier m/x/y/z describes the private key that is the z-th child of the parent private key m/x/y, which is the y-th child of the parent private key m/x, which is the x-th child of the parent master private key m.

HD wallet path examples

| HD path | Key described |
| --- | --- |
| m/0 | The first (0) child private key from the master private key (m) |
| m/0/0 | The first (0) child private key from the first child (m/0) |
| m/0'/0 | The first (0) normal child from the first *hardened* child (m/0') |
| m/1/0 | The first (0) child private key from the second child (m/1) |
| M/23/17/0/0 | The first (0) child public key from the first child (M/23/17/0) from the 18th child (M/23/17) from the 24th child (M/23) |

===== Navigating the HD wallet tree structure

The HD wallet tree structure offers tremendous flexibility. Each parent extended key can have 4 billion children: 2 billion normal children and 2 billion hardened children. Each of those children can have another 4 billion children, and so on. The tree can be as deep as you want, with an infinite number of generations. With all that flexibility, however, it becomes quite difficult to navigate this infinite tree. It is especially difficult to transfer HD wallets between implementations, because the possibilities for internal organization into branches and subbranches are endless.

Two BIPs offer a solution to this complexity by creating some proposed standards for the structure of HD wallet trees. BIP-43 proposes the use of the first hardened child index as a special identifier that signifies the "purpose" of the tree structure. Based on BIP-43, an HD wallet should use only one level-1 branch of the tree, with the index number identifying the structure and namespace of the rest of the tree by defining its purpose. For example, an HD wallet using only branch m/i&#x27;/ is intended to signify a specific purpose and that purpose is identified by index number "i."

Extending that specification, BIP-44 proposes a multiaccount structure as "purpose" number 44' under BIP-43. All HD wallets following the BIP-44 structure are identified by the fact that they only used one branch of the tree: m/44'/.

BIP-44 specifies the structure as consisting of five predefined tree levels:

m / purpose' / coin\_type' / account' / change / address\_index

The first-level "purpose" is always set to 44'. The second-level "coin\_type" specifies the type of cryptocurrency coin, allowing for multicurrency HD wallets where each currency has its own subtree under the second level. There are three currencies defined for now: Bitcoin is m/44'/0', Bitcoin Testnet is m/44&#x27;/1&#x27;, and Litecoin is m/44&#x27;/2&#x27;.

The third level of the tree is "account," which allows users to subdivide their wallets into separate logical subaccounts, for accounting or organizational purposes. For example, an HD wallet might contain two bitcoin "accounts": m/44&#x27;/0&#x27;/0&#x27; and m/44&#x27;/0&#x27;/1&#x27;. Each account is the root of its own subtree.

On the fourth level, "change," an HD wallet has two subtrees, one for creating receiving addresses and one for creating change addresses. Note that whereas the previous levels used hardened derivation, this level uses normal derivation. This is to allow this level of the tree to export extended public keys for use in a nonsecured environment. Usable addresses are derived by the HD wallet as children of the fourth level, making the fifth level of the tree the "address\_index." For example, the third receiving address for bitcoin payments in the primary account would be M/44&#x27;/0&#x27;/0&#x27;/0/2. [table\_title](#table_4-9) shows a few more examples.

BIP-44 HD wallet structure examples

| HD path | Key described |
| --- | --- |
| M/44&#x27;/0&#x27;/0&#x27;/0/2 | The third receiving public key for the primary bitcoin account |
| M/44&#x27;/0&#x27;/3&#x27;/1/14 | The fifteenth change-address public key for the fourth bitcoin account |
| m/44&#x27;/2&#x27;/0&#x27;/0/1 | The second private key in the Litecoin main account, for signing transactions |

==== Using an Extended Public Key on a Web Store

Let’s see how HD wallets are used by continuing our story with Gabriel’s web store.

Gabriel first set up his web store as a hobby, based on a simple hosted Wordpress page. His store was quite basic with only a few pages and an order form with a single Bitcoin address.

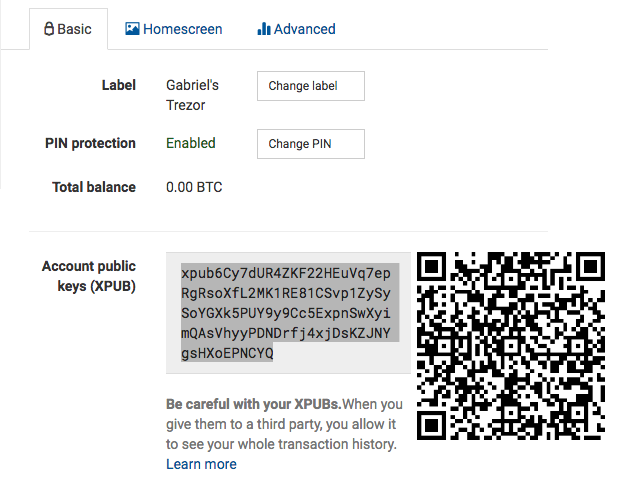
Gabriel used the first Bitcoin address generated by his Trezor device as the main Bitcoin address for his store. This way, all incoming payments would be paid to an address controlled by his Trezor hardware wallet.

Customers would submit an order using the form and send payment to Gabriel’s published Bitcoin address, triggering an email with the order details for Gabriel to process. With just a few orders each week, this system worked well enough.

However, the little web store became quite successful and attracted many orders from the local community. Soon, Gabriel was overwhelmed. With all the orders paying the same address, it became difficult to correctly match orders and transactions, especially when multiple orders for the same amount came in close together.

Gabriel’s HD wallet offers a much better solution through the ability to derive public child keys without knowing the private keys. Gabriel can load an extended public key (xpub) on his website, which can be used to derive a unique address for every customer order. Gabriel can spend the funds from his Trezor, but the xpub loaded on the website can only generate addresses and receive funds. This feature of HD wallets is a great security feature. Gabriel’s website does not contain any private keys and therefore does not need high levels of security.

To export the xpub, Gabriel uses the Trezor Suite desktop app in conjunction with the Trezor hardware wallet. The Trezor device must be plugged in for the public keys to be exported. Note that hardware wallets will never export private keys—those always remain on the device. [figure\_title](#export_xpub) shows what Gabriel sees in Trezor Suite when exporting the xpub.



Exporting an xpub from a Trezor hardware wallet

Gabriel copies the xpub to his web store’s bitcoin shop software. He uses *BTCPay Server*, which is an open source web-store for a variety of web hosting and content platforms. BTCPay Server uses the xpub to generate a unique address for every purchase.

===== Account Discovery and Management

Gabriel’s business is flourishing. He has provided his extended public key (xpub) to *BTCPay Server*, which is generating unique addresses for customers to his website. Every time a customer to Gabriel’s website clicks on the "Checkout" button with a specified payment modality (in this case, bitcoin), *BTCPay Server* generates a new address for that customer. More specifically, *BTCPay Server* iterates on the *address\_index* tree to create a new address to display to the customer, as defined by BIP-44. If the customer decides to switch payment methods or abandon the transaction entirely, this Bitcoin address goes unused and will not be used for another customer right away.

At a single moment in time, Gabriel’s website may have a large volume of outstanding addresses for customers making purchases, some of which may go unused and eventually expire. Once these addresses expire, *BTCPay Server* will go back to reuse these addresses to fill the gap in *address\_index*, but it becomes clear how there can be gaps between the *address\_index* leaves of the hierarchical deterministic tree where the money is actually located.

Let’s say that Gabriel is interested in viewing his total amount of bitcoin earned on a watch-only wallet (one that allows you to view transaction history, but not spend funds) that is separate from BTCPay Server but also conforms to the BIP-44 standard. How should this separate wallet go about searching for funds in this vast hierarchical tree, and when should it stop looking? Most wallets will typically follow an iterative process that utilizes a predefined limit, known as the *gap limit*. If, while searching for used addresses, the wallet doesn’t find used addresses in a row beyond this limit number, it will stop searching the address chain. The default gap limit is typically set to 20. This is detailed in [BIP-44](https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki).

Gap limits explain the phenomenon whereby the importing of a wallet may show an incorrect or zero balance. The funds are not lost, but rather, the wallet importing function has not traversed enough leaves to fully detect funds. Many wallets allow this default gap limit to be changed, and Gabriel may need to increase this limit to allow his wallet to fully import his transaction history.

== Transactions

=== Introduction

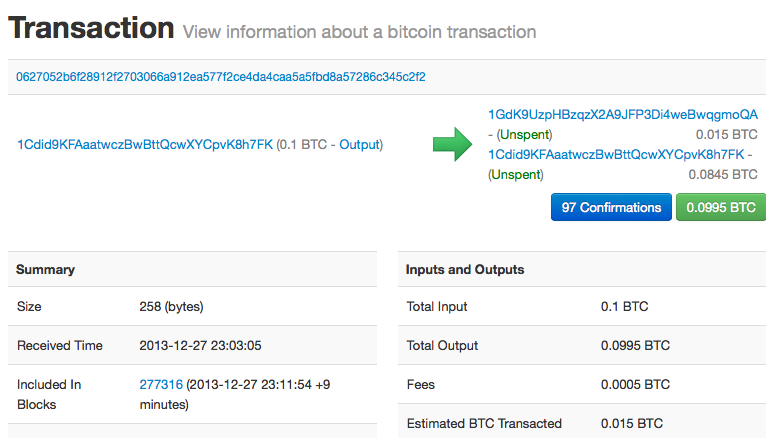
Transactions are the most important part of the Bitcoin system. Everything else in bitcoin is designed to ensure that transactions can be created, propagated on the network, validated, and finally added to the global ledger of transactions (the blockchain). Transactions are data structures that encode the transfer of value between participants in the Bitcoin system. Each transaction is a public entry in bitcoin’s blockchain, the global double-entry bookkeeping ledger.

In this chapter we will examine all the various forms of transactions, what they contain, how to create them, how they are verified, and how they become part of the permanent record of all transactions. When we use the term "wallet" in this chapter, we are referring to the software that constructs transactions, not just the database of keys.

=== Transactions in Detail

In [simpara\_title](#ch02_bitcoin_overview), we looked at the transaction Alice used to pay for coffee at Bob’s coffee shop using a block explorer ([figure\_title](#alices_transactions_to_bobs_cafe)).

The block explorer application shows a transaction from Alice’s "address" to Bob’s "address." This is a much simplified view of what is contained in a transaction. In fact, as we will see in this chapter, much of the information shown is constructed by the block explorer and is not actually in the transaction.



Alice’s transaction to Bob’s Cafe

==== Transactions—Behind the Scenes

Behind the scenes, an actual transaction looks very different from a transaction provided by a typical block explorer. In fact, most of the high-level constructs we see in the various bitcoin application user interfaces *do not actually exist* in the Bitcoin system.

We can use Bitcoin Core’s command-line interface (getrawtransaction and decoderawtransaction) to retrieve Alice’s "raw" transaction, decode it, and see what it contains. The result looks like this:

**Alice’s transaction decoded**

{  
 "version": 1,  
 "locktime": 0,  
 "vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "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",  
 }  
 ]  
}

You may notice a few things about this transaction, mostly the things that are missing! Where is Alice’s address? Where is Bob’s address? Where is the 0.1 input "sent" by Alice? In bitcoin, there are no coins, no senders, no recipients, no balances, no accounts, and no addresses. All those things are constructed at a higher level for the benefit of the user, to make things easier to understand.

You may also notice a lot of strange and indecipherable fields and hexadecimal strings. Don’t worry, we will explain each field shown here in detail in this chapter.

=== Transaction Outputs and Inputs

The fundamental building block of a bitcoin transaction is a *transaction output*. Transaction outputs are indivisible chunks of bitcoin currency, recorded on the blockchain, and recognized as valid by the entire network. Bitcoin full nodes track all available and spendable outputs, known as *unspent transaction outputs*, or *UTXO*. The collection of all UTXO is known as the *UTXO set* and currently numbers in the millions of UTXO. The UTXO set grows as new UTXO is created and shrinks when UTXO is consumed. Every transaction represents a change (state transition) in the UTXO set.

When we say that a user’s wallet has "received" bitcoin, what we mean is that the wallet has detected on the blockchain an UTXO that can be spent with one of the keys controlled by that wallet. Thus, a user’s bitcoin "balance" is the sum of all UTXO that user’s wallet can spend and which may be scattered among hundreds of transactions and hundreds of blocks. The concept of a balance is created by the wallet application. The wallet calculates the user’s balance by scanning the blockchain and aggregating the value of any UTXO the wallet can spend with the keys it controls. Most wallets maintain a database or use a database service to store a quick reference set of all the UTXO they can spend with the keys they control.

[figure\_title](#utxo-stxo) displays the blockchain in three different moments, as the transaction chain from Joe to Gopesh is being built. Notice how each transaction spends an UTXO that was created in a previous transaction, turning it into a *spent transaction output*, or *STXO*. Since transaction #1 (from Joe to Alice) spends a single UTXO (from Joe) and creates a single UTXO (to Alice), it doesn’t modify the UTXO set size. On the other hand, transactions #2 and #3 both create change outputs to the sender, spending a single UTXO and creating two UTXO (the payment and the change output). Therefore, each of them increases the UTXO set size by 1.

Transaction chain from Joe to Gopesh being built on the blockchain

A transaction output can have an arbitrary (integer) value denominated as a multiple of satoshis. Just as dollars can be divided down to two decimal places as cents, bitcoin can be divided down to eight decimal places as satoshis. Although an output can have any arbitrary value, once created it is indivisible. This is an important characteristic of outputs that needs to be emphasized: outputs are *discrete* and *indivisible* units of value, denominated in integer satoshis. An unspent output can only be consumed in its entirety by a transaction.

If an UTXO is larger than the desired value of a transaction, it must still be consumed in its entirety and change must be generated in the transaction. In other words, if you have an UTXO worth 20 bitcoin and want to pay only 1 bitcoin, your transaction must consume the entire 20-bitcoin UTXO and produce two outputs: one paying 1 bitcoin to your desired recipient and another paying 19 bitcoin in change back to your wallet. As a result of the indivisible nature of transaction outputs, most bitcoin transactions will have to generate change.

Imagine a shopper buying a $1.50 beverage, reaching into her wallet and trying to find a combination of coins and bank notes to cover the $1.50 cost. The shopper will choose exact change if available e.g. a dollar bill and two quarters (a quarter is $0.25), or a combination of smaller denominations (six quarters), or if necessary, a larger unit such as a $5 note. If she hands too much money, say $5, to the shop owner, she will expect $3.50 change, which she will return to her wallet and have available for future transactions.

Similarly, a bitcoin transaction must be created from a user’s UTXO in whatever denominations that user has available. Users cannot cut an UTXO in half any more than they can cut a dollar bill in half and use it as currency. The user’s wallet application will typically select from the user’s available UTXO to compose an amount greater than or equal to the desired transaction amount.

As with real life, the bitcoin application can use several strategies to satisfy the purchase amount: combining several smaller units, finding exact change, or using a single unit larger than the transaction value and making change. All of this complex assembly of spendable UTXO is done by the user’s wallet automatically and is invisible to users. It is only relevant if you are programmatically constructing raw transactions from UTXO.

A transaction consumes previously recorded unspent transaction outputs and creates new transaction outputs that can be consumed by a future transaction. This way, chunks of bitcoin value move forward from owner to owner in a chain of transactions consuming and creating UTXO.

The exception to the output and input chain is a special type of transaction called the *coinbase* transaction, which is the first transaction in each block. This transaction is placed there by the "winning" miner and creates brand-new bitcoin payable to that miner as a reward for mining. This special coinbase transaction does not consume UTXO; instead, it has a special type of input called the "coinbase." This is how bitcoin’s money supply is created during the mining process, as we will see in [simpara\_title](#mining).

What comes first? Inputs or outputs, the chicken or the egg? Strictly speaking, outputs come first because coinbase transactions, which generate new bitcoin, have no inputs and create outputs from nothing.

==== Transaction Outputs

Every bitcoin transaction creates outputs, which are recorded on the bitcoin ledger. Almost all of these outputs, with one exception (see [simpara\_title](#op_return)) create spendable chunks of bitcoin called UTXO, which are then recognized by the whole network and available for the owner to spend in a future transaction.

UTXO are tracked by every full-node Bitcoin client in the UTXO set. New transactions consume (spend) one or more of these outputs from the UTXO set.

Transaction outputs consist of two parts:

* An amount of bitcoin, denominated in *satoshis*, the smallest bitcoin unit
* A cryptographic puzzle that determines the conditions required to spend the output

The cryptographic puzzle is also known as a *locking script*, a *witness script*, or a scriptPubKey.

The transaction scripting language, used in the locking script mentioned previously, is discussed in detail in [simpara\_title](#tx_script).

Now, let’s look at Alice’s transaction (shown previously in [simpara\_title](#transactions_behind_the_scenes)) and see if we can identify the outputs. In the JSON encoding, the outputs are in an array (list) named 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",  
 }  
]

As you can see, the transaction contains two outputs. Each output is defined by a value and a cryptographic puzzle. In the encoding shown by Bitcoin Core, the value is shown in bitcoin, but in the transaction itself it is recorded as an integer denominated in satoshis. The second part of each output is the cryptographic puzzle that sets the conditions for spending. Bitcoin Core shows this as scriptPubKey and shows us a human-readable representation of the script.

The topic of locking and unlocking UTXO will be discussed later, in [simpara\_title](#tx_lock_unlock). The scripting language that is used for the script in scriptPubKey is discussed in [simpara\_title](#tx_script). But before we delve into those topics, we need to understand the overall structure of transaction inputs and outputs.

===== Transaction serialization—outputs

When transactions are transmitted over the network or exchanged between applications, they are *serialized*. Serialization is the process of converting the internal representation of a data structure into a format that can be transmitted one byte at a time, also known as a byte stream. Serialization is most commonly used for encoding data structures for transmission over a network or for storage in a file. The serialization format of a transaction output is shown in [table\_title](#tx_out_structure).

Transaction output serialization

| Size | Field | Description |
| --- | --- | --- |
| 8 bytes (little-endian) | Amount | Bitcoin value in satoshis (10-8 bitcoin) |
| 1–9 bytes (VarInt) | Locking-Script Size | Locking-Script length in bytes, to follow |
| Variable | Locking-Script | A script defining the conditions needed to spend the output |

Most bitcoin libraries and frameworks do not store transactions internally as byte-streams, as that would require complex parsing every time you needed to access a single field. For convenience and readability, bitcoin libraries store transactions internally in data structures (usually object-oriented structures).

The process of converting from the byte-stream representation of a transaction to a library’s internal representation data structure is called *deserialization* or *transaction parsing*. The process of converting back to a byte-stream for transmission over the network, for hashing, or for storage on disk is called *serialization*. Most bitcoin libraries have built-in functions for transaction serialization and deserialization.

See if you can manually decode Alice’s transaction from the serialized hexadecimal form, finding some of the elements we saw previously. The section containing the two outputs is highlighted in [example\_title](#example_6_1) to help you:

0100000001186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73 4d2804fe65fa35779000000008b483045022100884d142d86652a3f47 ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039 ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813 01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84 16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1 7b4a10fa336a8d752adfffffffff02**60e31600000000001976a914ab6** **8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000** **1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac** 00000000

Here are some hints:

* There are two outputs in the highlighted section, each serialized as shown in [table\_title](#tx_out_structure).
* The value of 0.015 bitcoin is 1,500,000 satoshis. That’s 16 e3 60 in hexadecimal.
* In the serialized transaction, the value 16 e3 60 is encoded in little-endian (least-significant-byte-first) byte order, so it looks like 60 e3 16.
* The scriptPubKey length is 25 bytes, which is 19 in hexadecimal.

==== Transaction Inputs

Transaction inputs identify (by reference) which UTXO will be consumed and provide proof of ownership through an unlocking script.

To build a transaction, a wallet selects from the UTXO it controls, UTXO with enough value to make the requested payment. Sometimes one UTXO is enough, other times more than one is needed. For each UTXO that will be consumed to make this payment, the wallet creates one input pointing to the UTXO and unlocks it with an unlocking script.

Let’s look at the components of an input in greater detail. The first part of an input is a pointer to an UTXO by reference to the transaction hash and an output index, which identifies the specific UTXO in that transaction. The second part is an unlocking script, which the wallet constructs in order to satisfy the spending conditions set in the UTXO. Most often, the unlocking script is a digital signature and public key proving ownership of the bitcoin. However, not all unlocking scripts contain signatures. The third part is a sequence number, which will be discussed later.

Consider our example in [simpara\_title](#transactions_behind_the_scenes). The transaction inputs are an array (list) called vin:

**The transaction inputs in Alice’s transaction**

"vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "sequence": 4294967295  
 }  
]

As you can see, there is only one input in the list (because one UTXO contained sufficient value to make this payment). The input contains four elements:

* A transaction ID, referencing the transaction that contains the UTXO being spent
* An output index (vout), identifying which UTXO from that transaction is referenced (first one is zero)
* A scriptSig, which satisfies the conditions placed on the UTXO, unlocking it for spending
* A sequence number (to be discussed later)

In Alice’s transaction, the input points to the transaction ID:

7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18

and output index 0 (i.e., the first UTXO created by that transaction). The unlocking script is constructed by Alice’s wallet by first retrieving the referenced UTXO, examining its locking script, and then using it to build the necessary unlocking script to satisfy it.

Looking just at the input you may have noticed that we don’t know anything about this UTXO, other than a reference to the parent transaction containing it. We don’t know its value (amount in satoshi), and we don’t know the locking script that sets the conditions for spending it. To find this information, we must retrieve the referenced UTXO by retrieving the parent transaction that contains it. Notice that because the value of the input is not explicitly stated, we must also use the referenced UTXO in order to calculate the fees that will be paid in this transaction (see [simpara\_title](#tx_fees)).

It’s not just Alice’s wallet that needs to retrieve UTXO referenced in the inputs. Once this transaction is broadcast to the network, every validating node will also need to retrieve the UTXO referenced in the transaction inputs in order to validate the transaction.

Transactions on their own seem incomplete because they lack context. They reference UTXO in their inputs but without retrieving that UTXO we cannot know the value of the inputs or their locking conditions. When writing bitcoin software, anytime you decode a transaction with the intent of validating it or counting the fees or checking the unlocking script, your code will first have to retrieve the referenced UTXO from the blockchain in order to build the context implied but not present in the UTXO references of the inputs. For example, to calculate the amount paid in fees, you must know the sum of the values of inputs and outputs. But without retrieving the UTXO referenced in the inputs, you do not know their value. So a seemingly simple operation like counting fees in a single transaction in fact involves multiple steps and data from multiple transactions.

We can use the same sequence of commands with Bitcoin Core as we used when retrieving Alice’s transaction (getrawtransaction and decoderawtransaction). With that we can get the UTXO referenced in the input from Alice’s transaction and take a look:

**UTXO from the previous transaction, referenced in the input from Alice’s transaction**

"vout": [  
 {  
 "value": 0.10000000,  
 "scriptPubKey": "OP\_DUP OP\_HASH160 7f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a8 OP\_EQUALVERIFY OP\_CHECKSIG"  
 }  
 ]

We see that this UTXO has a value of 0.1 BTC and that it has a locking script (scriptPubKey) that contains "OP\_DUP OP\_HASH160…​".

To fully understand Alice’s transaction we had to retrieve the previous transaction referenced as input. A function that retrieves previous transactions and unspent transaction outputs is very common and exists in almost every bitcoin library and API.

===== Transaction serialization—inputs

When transactions are serialized for transmission on the network, their inputs are encoded into a byte stream as shown in [table\_title](#tx_in_structure).

Transaction input serialization

| Size | Field | Description |
| --- | --- | --- |
| 32 bytes | Transaction Hash | Pointer to the transaction containing the UTXO to be spent |
| 4 bytes | Output Index | The index number of the UTXO to be spent; first one is 0 |
| 1–9 bytes (VarInt) | Unlocking-Script Size | Unlocking-Script length in bytes, to follow |
| Variable | Unlocking-Script | A script that fulfills the conditions of the UTXO locking script |
| 4 bytes | Sequence Number | Used for locktime or disabled (0xFFFFFFFF) |

As with the outputs, let’s see if we can find the inputs from Alice’s transaction in the serialized format. First, the inputs decoded:

"vin": [  
 {  
 "txid": "7957a35fe64f80d234d76d83a2a8f1a0d8149a41d81de548f0a65a8a999f6f18",  
 "vout": 0,  
 "scriptSig" : "3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813[ALL] 0484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade8416ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc17b4a10fa336a8d752adf",  
 "sequence": 4294967295  
 }  
],

Now, let’s see if we can identify these fields in the serialized hex encoding in [example\_title](#example_6_2):

0100000001**186f9f998a5aa6f048e51dd8419a14d8a0f1a8a2836dd73** **4d2804fe65fa35779000000008b483045022100884d142d86652a3f47** **ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039** **ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813** **01410484ecc0d46f1918b30928fa0e4ed99f16a0fb4fde0735e7ade84** **16ab9fe423cc5412336376789d172787ec3457eee41c04f4938de5cc1** **7b4a10fa336a8d752adfffffffff**0260e31600000000001976a914ab6 8025513c3dbd2f7b92a94e0581f5d50f654e788acd0ef800000000000 1976a9147f9b1a7fb68d60c536c2fd8aeaa53a8f3cc025a888ac00000 000

Hints:

* The transaction ID is serialized in reversed byte order, so it starts with (hex) 18 and ends with 79
* The output index is a 4-byte group of zeros, easy to identify
* The length of the scriptSig is 139 bytes, or 8b in hex
* The sequence number is set to FFFFFFFF, again easy to identify

ScriptSig is a specific type of unlocking script that when serialized for transmission on the network, inputs are encoded into a byte stream as shown in [table\_title](#scriptsig_in_structure). The serialization of the signature field is detailed in [simpara\_title](#seralization_of_signatures_der). The signature field also includes a Signature Hash Type (SIGHASH), which is detailed in [simpara\_title](#sighash_types).

ScriptSig input serialization

| Size | Field | Description |
| --- | --- | --- |
| 1–9 bytes (VarInt) | Signature Size | Signature length in bytes, to follow |
| Variable | Signature | A signature that is produced by the user’s wallet from his or her private key, which includes a SIGHASH |
| 1–9 bytes (VarInt) | Public Key Size | Public key length in bytes, to follow |
| Variable | Public Key | The public key, unhashed |

==== Transaction Fees

Most transactions include transaction fees, which compensate the bitcoin miners for securing the network. Fees also serve as a security mechanism themselves, by making it economically infeasible for attackers to flood the network with transactions. Mining and the fees and rewards collected by miners are discussed in more detail in [simpara\_title](#mining).

This section examines how transaction fees are included in a typical transaction. Most wallets calculate and include transaction fees automatically. However, if you are constructing transactions programmatically, or using a command-line interface, you must manually account for and include these fees.

Transaction fees serve as an incentive to include (mine) a transaction into the next block and also as a disincentive against abuse of the system by imposing a small cost on every transaction. Transaction fees are collected by the miner who mines the block that records the transaction on the blockchain.

Transaction fees are calculated based on the size of the transaction in kilobytes, not the value of the transaction in bitcoin. Overall, transaction fees are set based on market forces within the Bitcoin network. Miners prioritize transactions based on many different criteria, including fees, and might even process transactions for free under certain circumstances. Transaction fees affect the processing priority, meaning that a transaction with sufficient fees is likely to be included in the next block mined, whereas a transaction with insufficient or no fees might be delayed, processed on a best-effort basis after a few blocks, or not processed at all. Transaction fees are not mandatory, and transactions without fees might be processed eventually; however, including transaction fees encourages priority processing.

Over time, the way transaction fees are calculated and the effect they have on transaction prioritization has evolved. At first, transaction fees were fixed and constant across the network. Gradually, the fee structure relaxed and may be influenced by market forces, based on network capacity and transaction volume. Since at least the beginning of 2016, capacity limits in bitcoin have created competition between transactions, resulting in higher fees and effectively making free transactions a thing of the past. Zero fee or very low fee transactions rarely get mined and sometimes will not even be propagated across the network.

In Bitcoin Core, fee relay policies are set by the minrelaytxfee option. The current default minrelaytxfee is 0.00001 bitcoin or a hundredth of a millibitcoin per kilobyte. Therefore, by default, transactions with a fee less than 0.00001 bitcoin are treated as free and are only relayed if there is space in the mempool; otherwise, they are dropped. Bitcoin nodes can override the default fee relay policy by adjusting the value of minrelaytxfee.

Any bitcoin service that creates transactions, including wallets, exchanges, retail applications, etc., *must* implement dynamic fees. Dynamic fees can be implemented through a third-party fee estimation service or with a built-in fee estimation algorithm. If you’re unsure, begin with a third-party service and as you gain experience design and implement your own algorithm if you wish to remove the third-party dependency.

Fee estimation algorithms calculate the appropriate fee, based on capacity and the fees offered by "competing" transactions. These algorithms range from simplistic (average or median fee in the last block) to sophisticated (statistical analysis). They estimate the necessary fee (in satoshis per byte) that will give a transaction a high probability of being selected and included within a certain number of blocks. Most services offer users the option of choosing high, medium, or low priority fees. High priority means users pay higher fees but the transaction is likely to be included in the next block. Medium and low priority means users pay lower transaction fees but the transactions may take much longer to confirm.

Many wallet applications use third-party services for fee calculations. One popular service is [*https://bitcoinfees.earn.com/*](https://bitcoinfees.earn.com/), which provides an API and a visual chart showing the fee in satoshi/byte for different priorities.

Static fees are no longer viable on the Bitcoin network. Wallets that set static fees will produce a poor user experience as transactions will often get "stuck" and remain unconfirmed. Users who don’t understand bitcoin transactions and fees are dismayed by "stuck" transactions because they think they’ve lost their money.

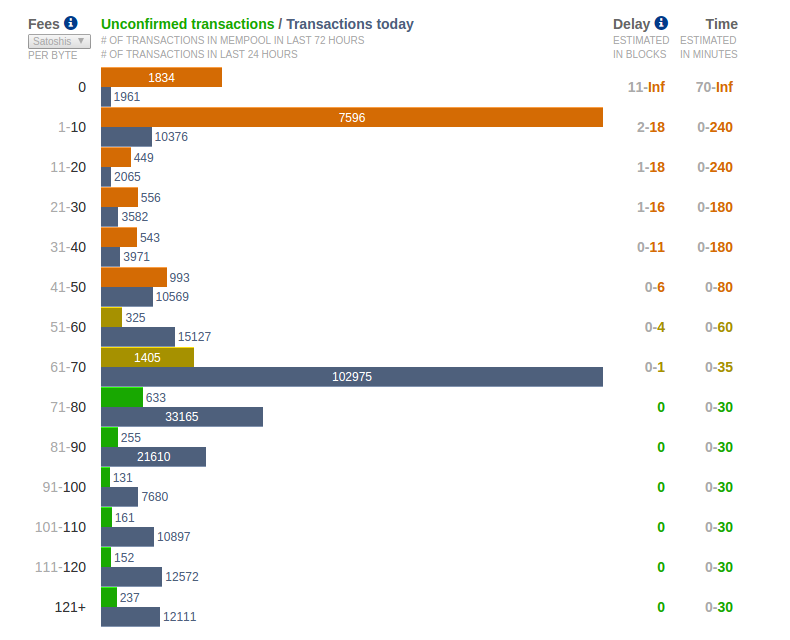
The chart in [figure\_title](#bitcoinfeesearncom) shows the real-time estimate of fees in 10 satoshi/byte increments and the expected confirmation time (in minutes and number of blocks) for transactions with fees in each range. For each fee range (e.g., 61–70 satoshi/byte), two horizontal bars show the number of unconfirmed transactions (1405) and total number of transactions in the past 24 hours (102,975), with fees in that range. Based on the graph, the recommended high-priority fee at this time was 80 satoshi/byte, a fee likely to result in the transaction being mined in the very next block (zero block delay). For perspective, the median transaction size is 226 bytes, so the recommended fee for this transaction size would be 18,080 satoshis (0.00018080 BTC).

The fee estimation data can be retrieved via a simple HTTP REST API, at <https://bitcoinfees.earn.com/api/v1/fees/recommended>. For example, on the command line using the curl command:

**Using the fee estimation API**

$ curl https://bitcoinfees.earn.com/api/v1/fees/recommended  
  
{"fastestFee":80,"halfHourFee":80,"hourFee":60}

The API returns a JSON object with the current fee estimate for fastest confirmation (fastestFee), confirmation within three blocks (halfHourFee) and six blocks (hourFee), in satoshi per byte.



Fee estimation service bitcoinfees.earn.com

==== Adding Fees to Transactions

The data structure of transactions does not have a field for fees. Instead, fees are implied as the difference between the sum of inputs and the sum of outputs. Any excess amount that remains after all outputs have been deducted from all inputs is the fee that is collected by the miners:

**Transaction fees are implied, as the excess of inputs minus outputs:**

Fees = Sum(Inputs) – Sum(Outputs)

This is a somewhat confusing element of transactions and an important point to understand, because if you are constructing your own transactions you must ensure you do not inadvertently include a very large fee by underspending the inputs. That means that you must account for all inputs, if necessary by creating change, or you will end up giving the miners a very big tip!

For example, if you consume a 20-bitcoin UTXO to make a 1-bitcoin payment, you must include a 19-bitcoin change output back to your wallet. Otherwise, the 19-bitcoin "leftover" will be counted as a transaction fee and will be collected by the miner who mines your transaction in a block. Although you will receive priority processing and make a miner very happy, this is probably not what you intended.

If you forget to add a change output in a manually constructed transaction, you will be paying the change as a transaction fee. Saying "Keep the change!" to the miner might not be what you really intended.

Let’s see how this works in practice, by looking at Alice’s coffee purchase again. Alice wants to spend 0.015 bitcoin to pay for coffee. To ensure this transaction is processed promptly, she will want to include a transaction fee, say 0.0005. That will mean that the total cost of the transaction will be 0.0155. Her wallet must therefore source a set of UTXO that adds up to 0.0155 bitcoin or more and, if necessary, create change. Let’s say her wallet has a 0.1-bitcoin UTXO available. It will therefore need to consume this UTXO, create one output to Bob’s Cafe for 0.015, and a second output with 0.0845 bitcoin in change back to her own wallet, leaving 0.0005 bitcoin unallocated, as an implicit fee for the transaction.

Now let’s look at a different scenario. Eugenia, our children’s charity director in the Philippines, has completed a fundraiser to purchase schoolbooks for the children. She received several thousand small donations from people all around the world, totaling 50 bitcoin, so her wallet is full of very small payments (UTXO). Now she wants to purchase hundreds of schoolbooks from a local publisher, paying in bitcoin.

As Eugenia’s wallet application tries to construct a single larger payment transaction, it must source from the available UTXO set, which is composed of many smaller amounts. That means that the resulting transaction will source from more than a hundred small-value UTXO as inputs and only one output, paying the book publisher. A transaction with that many inputs will be larger than one kilobyte, perhaps several kilobytes in size. As a result, it will require a much higher fee than the median-sized transaction.

Eugenia’s wallet application will calculate the appropriate fee by measuring the size of the transaction and multiplying that by the per-kilobyte fee. Many wallets will overpay fees for larger transactions to ensure the transaction is processed promptly. The higher fee is not because Eugenia is spending more money, but because her transaction is more complex and larger in size—​the fee is independent of the transaction’s bitcoin value.

=== Transaction Scripts and Script Language

The bitcoin transaction script language, called *Script*, is a Forth-like reverse-polish notation stack-based execution language. If that sounds like gibberish, you probably haven’t studied 1960s programming languages, but that’s ok—we will explain it all in this chapter. Both the locking script placed on an UTXO and the unlocking script are written in this scripting language. When a transaction is validated, the unlocking script in each input is executed alongside the corresponding locking script to see if it satisfies the spending condition.

Script is a very simple language that was designed to be limited in scope and executable on a range of hardware, perhaps as simple as an embedded device. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. For its use in validating programmable money, this is a deliberate security feature.

Today, most transactions processed through the Bitcoin network have the form "Payment to Bob’s Bitcoin address" and are based on a script called a Pay-to-Public-Key-Hash script. However, bitcoin transactions are not limited to the "Payment to Bob’s Bitcoin address" script. In fact, locking scripts can be written to express a vast variety of complex conditions. In order to understand these more complex scripts, we must first understand the basics of transaction scripts and script language.

In this section, we will demonstrate the basic components of the bitcoin transaction scripting language and show how it can be used to express simple conditions for spending and how those conditions can be satisfied by unlocking scripts.

Bitcoin transaction validation is not based on a static pattern, but instead is achieved through the execution of a scripting language. This language allows for a nearly infinite variety of conditions to be expressed. This is how bitcoin gets the power of "programmable money".

==== Turing Incompleteness

The bitcoin transaction script language contains many operators, but is deliberately limited in one important way—​there are no loops or complex flow control capabilities other than conditional flow control. This ensures that the language is not *Turing Complete*, meaning that scripts have limited complexity and predictable execution times. Script is not a general-purpose language. These limitations ensure that the language cannot be used to create an infinite loop or other form of "logic bomb" that could be embedded in a transaction in a way that causes a denial-of-service attack against the Bitcoin network. Remember, every transaction is validated by every full node on the Bitcoin network. A limited language prevents the transaction validation mechanism from being used as a vulnerability.

==== Stateless Verification

The bitcoin transaction script language is stateless, in that there is no state prior to execution of the script, or state saved after execution of the script. Therefore, all the information needed to execute a script is contained within the script. A script will predictably execute the same way on any system. If your system verifies a script, you can be sure that every other system in the Bitcoin network will also verify the script, meaning that a valid transaction is valid for everyone and everyone knows this. This predictability of outcomes is an essential benefit of the Bitcoin system.

==== Script Construction (Lock + Unlock)

Bitcoin’s transaction validation engine relies on two types of scripts to validate transactions: a locking script and an unlocking script.

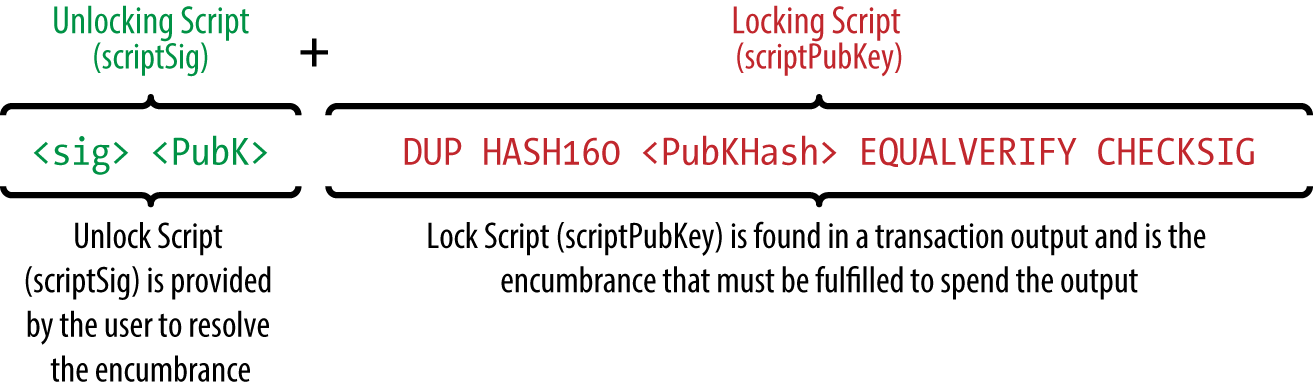
A locking script is a spending condition placed on an output: it specifies the conditions that must be met to spend the output in the future. Historically, the locking script was called a *scriptPubKey*, because it usually contained a public key or Bitcoin address (public key hash). In this book we refer to it as a "locking script" to acknowledge the much broader range of possibilities of this scripting technology. In most bitcoin applications, what we refer to as a locking script will appear in the source code as scriptPubKey. You will also see the locking script referred to as a *witness script* (see [simpara\_title](#segwit)) or more generally as a *cryptographic puzzle*. These terms all mean the same thing, at different levels of abstraction.

An unlocking script is a script that "solves," or satisfies, the conditions placed on an output by a locking script and allows the output to be spent. Unlocking scripts are part of every transaction input. Most of the time they contain a digital signature produced by the user’s wallet from his or her private key. Historically, the unlocking script was called *scriptSig*, because it usually contained a digital signature. In most bitcoin applications, the source code refers to the unlocking script as scriptSig. You will also see the unlocking script referred to as a *witness* (see [simpara\_title](#segwit)). In this book, we refer to it as an "unlocking script" to acknowledge the much broader range of locking script requirements, because not all unlocking scripts must contain signatures.

Every bitcoin validating node will validate transactions by executing the locking and unlocking scripts together. Each input contains an unlocking script and refers to a previously existing UTXO. The validation software will copy the unlocking script, retrieve the UTXO referenced by the input, and copy the locking script from that UTXO. The unlocking and locking script are then executed in sequence. The input is valid if the unlocking script satisfies the locking script conditions (see [simpara\_title](#script_exec)). All the inputs are validated independently, as part of the overall validation of the transaction.

Note that the UTXO is permanently recorded in the blockchain, and therefore is invariable and is unaffected by failed attempts to spend it by reference in a new transaction. Only a valid transaction that correctly satisfies the conditions of the output results in the output being considered as "spent" and removed from the set of unspent transaction outputs (UTXO set).

[figure\_title](#scriptSig_and_scriptPubKey) is an example of the unlocking and locking scripts for the most common type of bitcoin transaction (a payment to a public key hash), showing the combined script resulting from the concatenation of the unlocking and locking scripts prior to script validation.



Combining scriptSig and scriptPubKey to evaluate a transaction script

===== The script execution stack

Bitcoin’s scripting language is called a stack-based language because it uses a data structure called a *stack*. A stack is a very simple data structure that can be visualized as a stack of cards. A stack allows two operations: push and pop. Push adds an item on top of the stack. Pop removes the top item from the stack. Operations on a stack can only act on the topmost item on the stack. A stack data structure is also called a Last-In-First-Out, or "LIFO" queue.

The scripting language executes the script by processing each item from left to right. Numbers (data constants) are pushed onto the stack. Operators push or pop one or more parameters from the stack, act on them, and might push a result onto the stack. For example, OP\_ADD will pop two items from the stack, add them, and push the resulting sum onto the stack.

Conditional operators evaluate a condition, producing a boolean result of TRUE or FALSE. For example, OP\_EQUAL pops two items from the stack and pushes TRUE (TRUE is represented by the number 1) if they are equal or FALSE (represented by zero) if they are not equal. Bitcoin transaction scripts usually contain a conditional operator, so that they can produce the TRUE result that signifies a valid transaction.

===== A simple script

Now let’s apply what we’ve learned about scripts and stacks to some simple examples.

In [figure\_title](#simplemath_script), the script 2 3 OP\_ADD 5 OP\_EQUAL demonstrates the arithmetic addition operator OP\_ADD, adding two numbers and putting the result on the stack, followed by the conditional operator OP\_EQUAL, which checks that the resulting sum is equal to 5. For brevity, the OP\_ prefix is omitted in the step-by-step example. For more details on the available script operators and functions, see [simpara\_title](#tx_script_ops).

Although most locking scripts refer to a public key hash (essentially, a Bitcoin address), thereby requiring proof of ownership to spend the funds, the script does not have to be that complex. Any combination of locking and unlocking scripts that results in a TRUE value is valid. The simple arithmetic we used as an example of the scripting language is also a valid locking script that can be used to lock a transaction output.

Use part of the arithmetic example script as the locking script:

3 OP\_ADD 5 OP\_EQUAL

which can be satisfied by a transaction containing an input with the unlocking script:

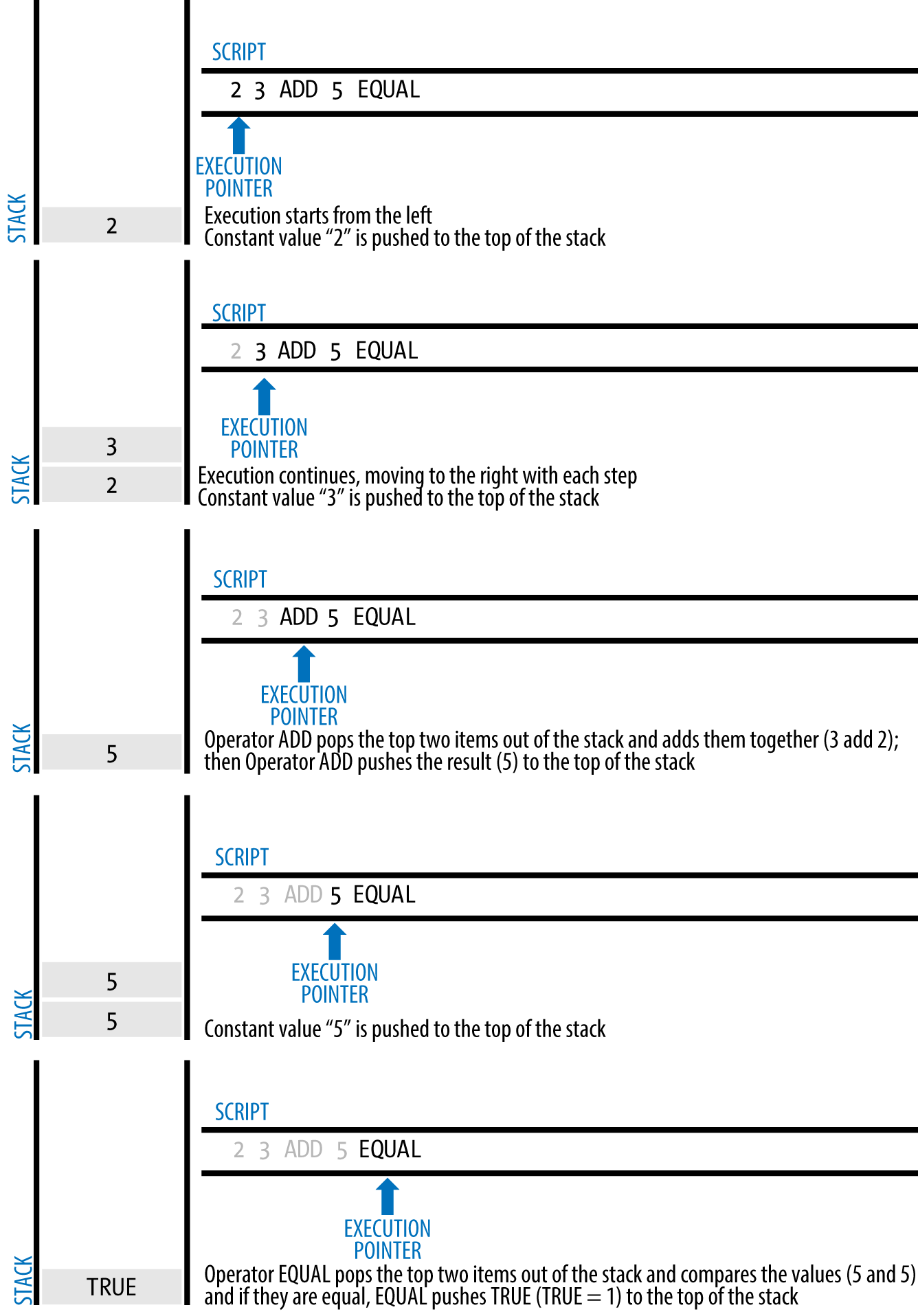
2

The validation software combines the locking and unlocking scripts and the resulting script is:

2 3 OP\_ADD 5 OP\_EQUAL

As we saw in the step-by-step example in [figure\_title](#simplemath_script), when this script is executed, the result is OP\_TRUE, making the transaction valid. Not only is this a valid transaction output locking script, but the resulting UTXO could be spent by anyone with the arithmetic skills to know that the number 2 satisfies the script.

Transactions are valid if the top result on the stack is TRUE (noted as &#x7b;0x01&#x7d;), any other nonzero value, not OP\_0, or if the stack is empty after script execution. Transactions are invalid if the top value on the stack is FALSE (a zero-length empty value, noted as &#x7b;&#x7d;) or if script execution is halted explicitly by an operator, such as OP\_VERIFY, OP\_RETURN, or a conditional terminator such as OP\_ENDIF. See [simpara\_title](#tx_script_ops) for details.



Bitcoin’s script validation doing simple math

The following is a slightly more complex script, which calculates 2 + 7 - 3 + 1. Notice that when the script contains several operators in a row, the stack allows the results of one operator to be acted upon by the next operator:

2 7 OP\_ADD 3 OP\_SUB 1 OP\_ADD 7 OP\_EQUAL

Try validating the preceding script yourself using pencil and paper. When the script execution ends, you should be left with the value TRUE on the stack.

===== Separate execution of unlocking and locking scripts

In the original Bitcoin client, the unlocking and locking scripts were concatenated and executed in sequence. For security reasons, this was changed in 2010, because of a vulnerability that allowed a malformed unlocking script to push data onto the stack and corrupt the locking script. In the current implementation, the scripts are executed separately with the stack transferred between the two executions, as described next.

First, the unlocking script is executed, using the stack execution engine. If the unlocking script is executed without errors (e.g., it has no "dangling" pointers left over), the main stack is copied and the locking script is executed. If the result of executing the locking script with the stack data copied from the unlocking script is "TRUE," the unlocking script has succeeded in resolving the conditions imposed by the locking script and, therefore, the input is a valid authorization to spend the UTXO. If any result other than "TRUE" remains after execution of the combined script, the input is invalid because it has failed to satisfy the spending conditions placed on the UTXO.

==== Pay-to-Public-Key-Hash (P2PKH)

The vast majority of transactions processed on the Bitcoin network spend outputs locked with a Pay-to-Public-Key-Hash or "P2PKH" script. These outputs contain a locking script that locks the output to a public key hash, more commonly known as a Bitcoin address. An output locked by a P2PKH script can be unlocked (spent) by presenting a public key and a digital signature created by the corresponding private key (see [simpara\_title](#digital_sigs)).

For example, let’s look at Alice’s payment to Bob’s Cafe again. Alice made a payment of 0.015 bitcoin to the cafe’s Bitcoin address. That transaction output would have a locking script of the form:

OP\_DUP OP\_HASH160 <Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

The Cafe Public Key Hash is equivalent to the Bitcoin address of the cafe, without the Base58Check encoding. Most applications would show the *public key hash* in hexadecimal encoding and not the familiar Bitcoin address Base58Check format that begins with a "1."

The preceding locking script can be satisfied with an unlocking script of the form:

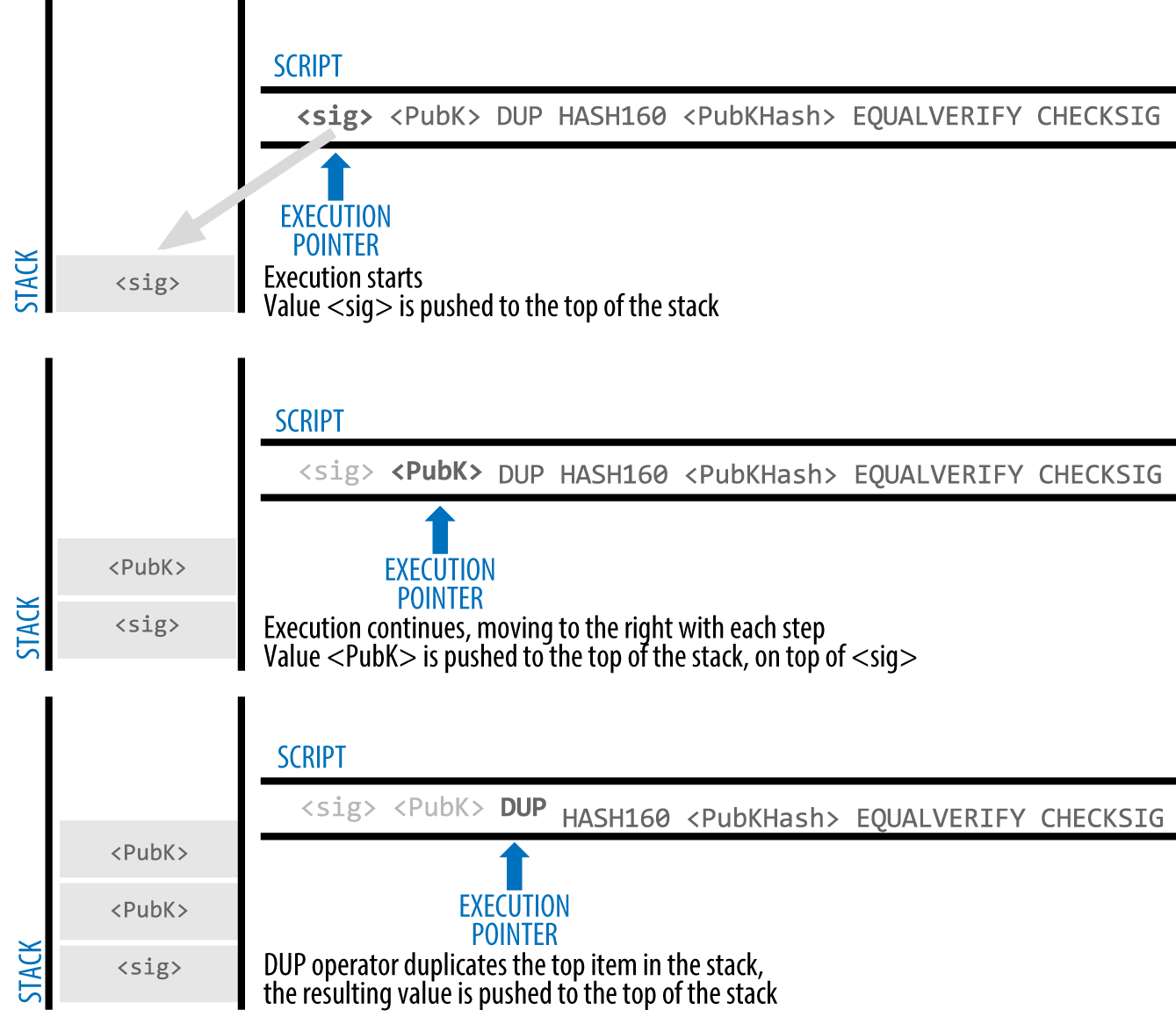
<Cafe Signature> <Cafe Public Key>

The two scripts together would form the following combined validation script:

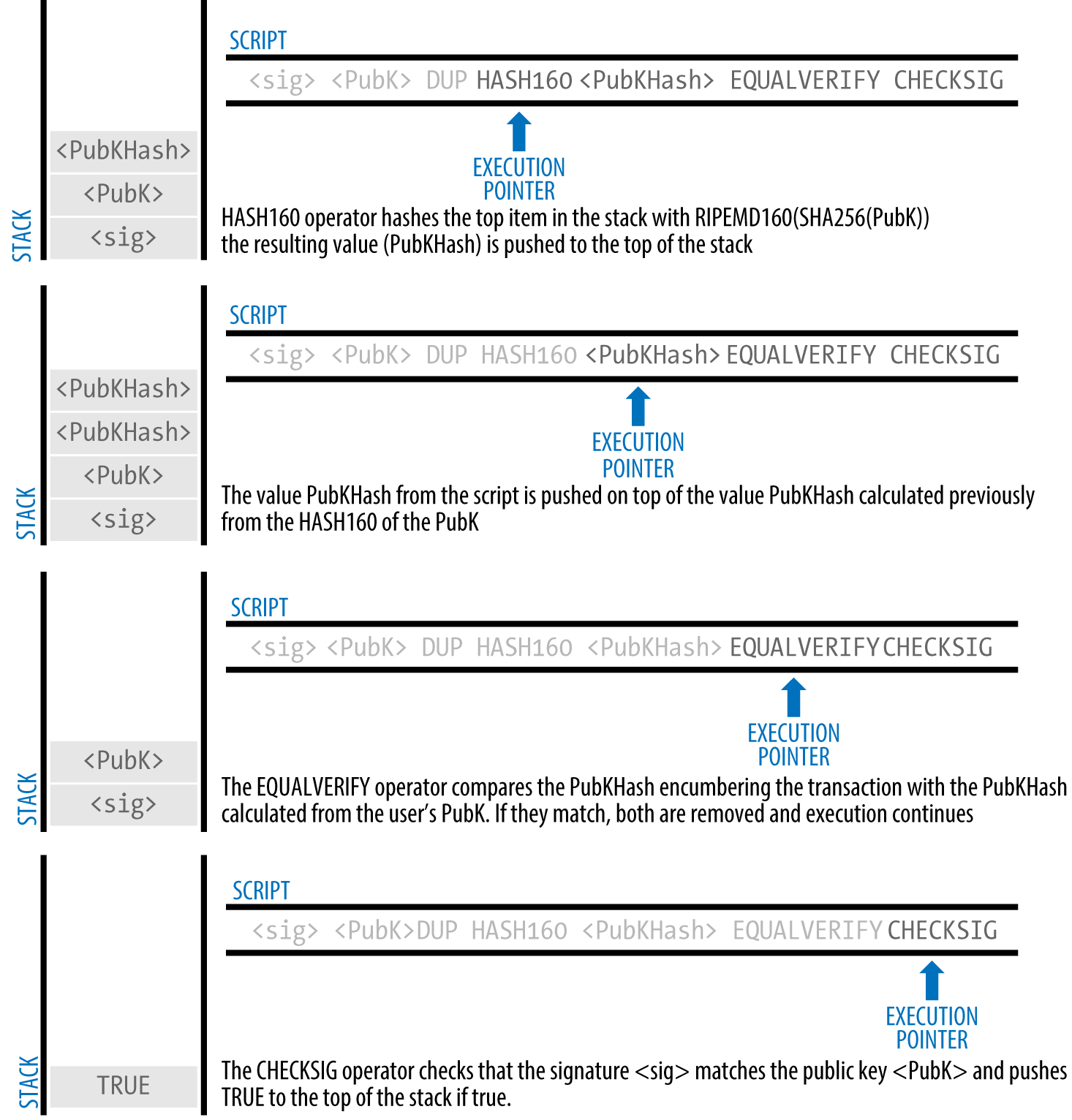
<Cafe Signature> <Cafe Public Key> OP\_DUP OP\_HASH160  
<Cafe Public Key Hash> OP\_EQUALVERIFY OP\_CHECKSIG

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In other words, the result will be TRUE if the unlocking script has a valid signature from the cafe’s private key that corresponds to the public key hash set as an encumbrance.

Figures #P2PubKHash1 and #P2PubKHash2 show (in two parts) a step-by-step execution of the combined script, which will prove this is a valid transaction.



Evaluating a script for a P2PKH transaction (part 1 of 2)



Evaluating a script for a P2PKH transaction (part 2 of 2)

=== Digital Signatures (ECDSA)

So far, we have not delved into any detail about "digital signatures." In this section we look at how digital signatures work and how they can present proof of ownership of a private key without revealing that private key.

The digital signature algorithm used in bitcoin is the *Elliptic Curve Digital Signature Algorithm*, or *ECDSA*. ECDSA is the algorithm used for digital signatures based on elliptic curve private/public key pairs, as described in [???](#elliptic_curve). ECDSA is used by the script functions OP\_CHECKSIG, OP\_CHECKSIGVERIFY, OP\_CHECKMULTISIG, and OP\_CHECKMULTISIGVERIFY. Any time you see those in a locking script, the unlocking script must contain an ECDSA signature.

A digital signature serves three purposes in bitcoin. First, the signature proves that the owner of the private key, who is by implication the owner of the funds, has *authorized* the spending of those funds. Secondly, the proof of authorization is *undeniable* (nonrepudiation). Thirdly, the signature proves that the transaction (or specific parts of the transaction) have not and *cannot be modified* by anyone after it has been signed.

Note that each transaction input is signed independently. This is critical, as neither the signatures nor the inputs have to belong to or be applied by the same "owners." In fact, a specific transaction scheme called "CoinJoin" uses this fact to create multi-party transactions for privacy.

Each transaction input and any signature it may contain is *completely* independent of any other input or signature. Multiple parties can collaborate to construct transactions and sign only one input each.

A digital signature is a mathematical scheme for demonstrating the authenticity of a digital message or documents. A valid digital signature gives a recipient reason to believe that the message was created by a known sender (authentication), that the sender cannot deny having sent the message (nonrepudiation), and that the message was not altered in transit (integrity).

*Source:* [*https://en.wikipedia.org/wiki/Digital\_signature*](https://en.wikipedia.org/wiki/Digital_signature)

==== How Digital Signatures Work

A digital signature is a *mathematical scheme* that consists of two parts. The first part is an algorithm for creating a signature, using a private key (the signing key), from a message (the transaction). The second part is an algorithm that allows anyone to verify the signature, given also the message and a public key.

===== Creating a digital signature

In bitcoin’s implementation of the ECDSA algorithm, the "message" being signed is the transaction, or more accurately a hash of a specific subset of the data in the transaction (see [simpara\_title](#sighash_types)). The signing key is the user’s private key. The result is the signature:

$\(Sig = F\_{sig}(F\_{hash}(m), dA)\)$

where:

* *dA* is the signing private key
* *m* is the transaction (or parts of it)
* *Fhash* is the hashing function
* *Fsig* is the signing algorithm
* *Sig* is the resulting signature

More details on the mathematics of ECDSA can be found in [simpara\_title](#ecdsa_math).

The function *Fsig* produces a signature Sig that is composed of two values, commonly referred to as R and S:

Sig = (R, S)

Now that the two values R and S have been calculated, they are serialized into a byte-stream using an international standard encoding scheme called the *Distinguished Encoding Rules*, or *DER*.

===== Serialization of signatures (DER)

Let’s look at the transaction Alice created again. In the transaction input there is an unlocking script that contains the following DER-encoded signature from Alice’s wallet:

3045022100884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb02204b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e381301

That signature is a serialized byte-stream of the R and S values produced by Alice’s wallet to prove she owns the private key authorized to spend that output. The serialization format consists of nine elements as follows:

* 0x30—indicating the start of a DER sequence
* 0x45—the length of the sequence (69 bytes)
* 0x02—an integer value follows
* 0x21—the length of the integer (33 bytes)
* R—00884d142d86652a3f47ba4746ec719bbfbd040a570b1deccbb6498c75c4ae24cb
* 0x02—another integer follows
* 0x20—the length of the integer (32 bytes)
* S—4b9f039ff08df09cbe9f6addac960298cad530a863ea8f53982c09db8f6e3813
* A suffix (0x01) indicating the type of hash used (SIGHASH\_ALL)

See if you can decode Alice’s serialized (DER-encoded) signature using this list. The important numbers are R and S; the rest of the data is part of the DER encoding scheme.

==== Verifying the Signature

To verify the signature, one must have the signature (R and S), the serialized transaction, and the public key (that corresponds to the private key used to create the signature). Essentially, verification of a signature means "Only the owner of the private key that generated this public key could have produced this signature on this transaction."

The signature verification algorithm takes the message (a hash of the transaction or parts of it), the signer’s public key and the signature (R and S values), and returns TRUE if the signature is valid for this message and public key.

==== Signature Hash Types (SIGHASH)

Digital signatures are applied to messages, which in the case of bitcoin, are the transactions themselves. The signature implies a *commitment* by the signer to specific transaction data. In the simplest form, the signature applies to the entire transaction, thereby committing all the inputs, outputs, and other transaction fields. However, a signature can commit to only a subset of the data in a transaction, which is useful for a number of scenarios as we will see in this section.

Bitcoin signatures have a way of indicating which part of a transaction’s data is included in the hash signed by the private key using a SIGHASH flag. The SIGHASH flag is a single byte that is appended to the signature. Every signature has a SIGHASH flag and the flag can be different from input to input. A transaction with three signed inputs may have three signatures with different SIGHASH flags, each signature signing (committing) different parts of the transaction.

Remember, each input may contain a signature in its unlocking script. As a result, a transaction that contains several inputs may have signatures with different SIGHASH flags that commit different parts of the transaction in each of the inputs. Note also that bitcoin transactions may contain inputs from different "owners," who may sign only one input in a partially constructed (and invalid) transaction, collaborating with others to gather all the necessary signatures to make a valid transaction. Many of the SIGHASH flag types only make sense if you think of multiple participants collaborating outside the Bitcoin network and updating a partially signed transaction.

There are three SIGHASH flags: ALL, NONE, and SINGLE, as shown in [table\_title](#sighash_types_and_their).

SIGHASH types and their meanings

| SIGHASH flag | Value | Description |
| --- | --- | --- |
| ALL | 0x01 | Signature applies to all inputs and outputs |
| NONE | 0x02 | Signature applies to all inputs, none of the outputs |
| SINGLE | 0x03 | Signature applies to all inputs but only the one output with the same index number as the signed input |

In addition, there is a modifier flag SIGHASH\_ANYONECANPAY, which can be combined with each of the preceding flags. When ANYONECANPAY is set, only one input is signed, leaving the rest (and their sequence numbers) open for modification. The ANYONECANPAY has the value 0x80 and is applied by bitwise OR, resulting in the combined flags as shown in [table\_title](#sighash_types_with_modifiers).

SIGHASH types with modifiers and their meanings

| SIGHASH flag | Value | Description |
| --- | --- | --- |
| ALL|ANYONECANPAY | 0x81 | Signature applies to one input and all outputs |
| NONE|ANYONECANPAY | 0x82 | Signature applies to one input, none of the outputs |
| SINGLE|ANYONECANPAY | 0x83 | Signature applies to one input and the output with the same index number |

These flag combinations are summarized in [figure\_title](#sighash_combinations).

Summary of different sighash combinations

The way SIGHASH flags are applied during signing and verification is that a copy of the transaction is made and certain fields within are truncated (set to zero length and emptied). The resulting transaction is serialized. The SIGHASH flag is added to the end of the serialized transaction and the result is hashed. The hash itself is the "message" that is signed. Depending on which SIGHASH flag is used, different parts of the transaction are truncated. The resulting hash depends on different subsets of the data in the transaction. By including the SIGHASH as the last step before hashing, the signature commits the SIGHASH type as well, so it can’t be changed (e.g., by a miner).

All SIGHASH types sign the transaction nLocktime field (see [simpara\_title](#transaction_locktime_nlocktime)). In addition, the SIGHASH type itself is appended to the transaction before it is signed, so that it can’t be modified once signed.

In the example of Alice’s transaction (see the list in [simpara\_title](#seralization_of_signatures_der)), we saw that the last part of the DER-encoded signature was 01, which is the SIGHASH\_ALL flag. This locks the transaction data, so Alice’s signature is committing the state of all inputs and outputs. This is the most common signature form.

Let’s look at some of the other SIGHASH types and how they can be used in practice:

ALL|ANYONECANPAY

This construction can be used to make a "crowdfunding”-style transaction. Someone attempting to raise funds can construct a transaction with a single output. The single output pays the "goal" amount to the fundraiser. Such a transaction is obviously not valid, as it has no inputs. However, others can now amend it by adding an input of their own, as a donation. They sign their own input with ALL|ANYONECANPAY. Unless enough inputs are gathered to reach the value of the output, the transaction is invalid. Each donation is a "pledge," which cannot be collected by the fundraiser until the entire goal amount is raised.

NONE

This construction can be used to create a "bearer check" or "blank check" of a specific amount. It commits to the input, but allows the output locking script to be changed. Anyone can write their own Bitcoin address into the output locking script and redeem the transaction. However, the output value itself is locked by the signature.

NONE|ANYONECANPAY

This construction can be used to build a "dust collector." Users who have tiny UTXO in their wallets can’t spend these because the cost in fees exceeds the value of the dust. With this type of signature, the dust UTXO can be donated for anyone to aggregate and spend whenever they want.

There are some proposals to modify or expand the SIGHASH system. One such proposal is *Bitmask Sighash Modes* by Blockstream’s Glenn Willen, as part of the Elements project. This aims to create a flexible replacement for SIGHASH types that allows "arbitrary, miner-rewritable bitmasks of inputs and outputs" that can express "more complex contractual precommitment schemes, such as signed offers with change in a distributed asset exchange."

You will not see SIGHASH flags presented as an option in a user’s wallet application. With few exceptions, wallets construct P2PKH scripts and sign with SIGHASH\_ALL flags. To use a different SIGHASH flag, you would have to write software to construct and sign transactions. More importantly, SIGHASH flags can be used by special-purpose bitcoin applications that enable novel uses.

==== ECDSA Math

As mentioned previously, signatures are created by a mathematical function *Fsig* that produces a signature composed of two values *R* and *S*. In this section we look at the function *Fsig* in more detail.

The signature algorithm first generates an *ephemeral* (temporary) private public key pair. This temporary key pair is used in the calculation of the *R* and *S* values, after a transformation involving the signing private key and the transaction hash.

The temporary key pair is based on a random number *k*, which is used as the temporary private key. From *k*, we generate the corresponding temporary public key *P* (calculated as *P = k\*G*, in the same way bitcoin public keys are derived; see [simpara\_title](#pubkey)). The *R* value of the digital signature is then the x coordinate of the ephemeral public key *P*.

From there, the algorithm calculates the *S* value of the signature, such that:

*S* = *k*-1 (*Hash*(*m*) + *dA* \* *R*) *mod n*

where:

* *k* is the ephemeral private key
* *R* is the x coordinate of the ephemeral public key
* *dA* is the signing private key
* *m* is the transaction data
* *n* is the prime order of the elliptic curve

Verification is the inverse of the signature generation function, using the *R*, *S* values and the public key to calculate a value *P*, which is a point on the elliptic curve (the ephemeral public key used in signature creation):

*P* = *S*-1 \* *Hash*(*m*) \* *G* + *S*-1 \* *R* \* *Qa*

where:

* *R* and *S* are the signature values
* *Qa* is Alice’s public key
* *m* is the transaction data that was signed
* *G* is the elliptic curve generator point

If the x coordinate of the calculated point *P* is equal to *R*, then the verifier can conclude that the signature is valid.

Note that in verifying the signature, the private key is neither known nor revealed.

ECDSA is necessarily a fairly complicated piece of math; a full explanation is beyond the scope of this book. A number of great guides online take you through it step by step: search for "ECDSA explained" or try this one: <https://bit.ly/2r0HhGB>.

==== The Importance of Randomness in Signatures

As we saw in [simpara\_title](#ecdsa_math), the signature generation algorithm uses a random key *k*, as the basis for an ephemeral private/public key pair. The value of *k* is not important, *as long as it is random*. If the same value *k* is used to produce two signatures on different messages (transactions), then the signing *private key* can be calculated by anyone. Reuse of the same value for *k* in a signature algorithm leads to exposure of the private key!

If the same value *k* is used in the signing algorithm on two different transactions, the private key can be calculated and exposed to the world!

This is not just a theoretical possibility. We have seen this issue lead to exposure of private keys in a few different implementations of transaction-signing algorithms in bitcoin. People have had funds stolen because of inadvertent reuse of a *k* value. The most common reason for reuse of a *k* value is an improperly initialized random-number generator.

To avoid this vulnerability, the industry best practice is to not generate *k* with a random-number generator seeded with entropy, but instead to use a deterministic-random process seeded with the transaction data itself. This ensures that each transaction produces a different *k*. The industry-standard algorithm for deterministic initialization of *k* is defined in [RFC 6979](https://tools.ietf.org/html/rfc6979), published by the Internet Engineering Task Force.

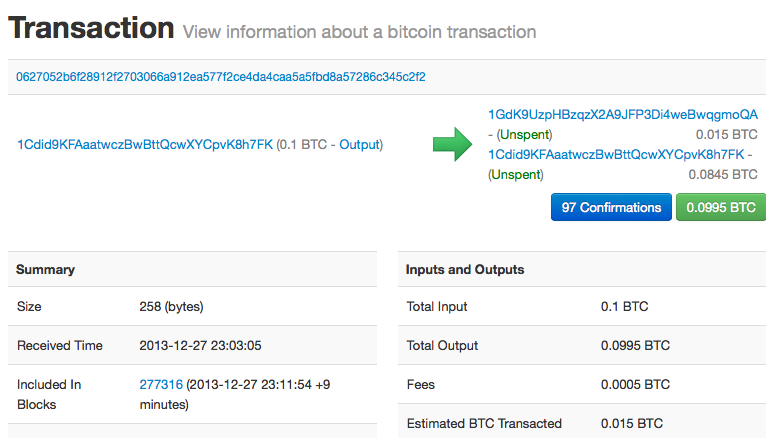
If you are implementing an algorithm to sign transactions in bitcoin, you *must* use RFC 6979 or a similarly deterministic-random algorithm to ensure you generate a different *k* for each transaction.

=== Bitcoin Addresses, Balances, and Other Abstractions

We began this chapter with the discovery that transactions look very different "behind the scenes" than how they are presented in wallets, blockchain explorers, and other user-facing applications. Many of the simplistic and familiar concepts from the earlier chapters, such as Bitcoin addresses and balances, seem to be absent from the transaction structure. We saw that transactions don’t contain Bitcoin addresses, per se, but instead operate through scripts that lock and unlock discrete values of bitcoin. Balances are not present anywhere in this system and yet every wallet application prominently displays the balance of the user’s wallet.

Now that we have explored what is actually included in a bitcoin transaction, we can examine how the higher-level abstractions are derived from the seemingly primitive components of the transaction.

Let’s look again at how Alice’s transaction was presented on a popular block explorer ([figure\_title](#alice_transaction_to_bobs_cafe)).

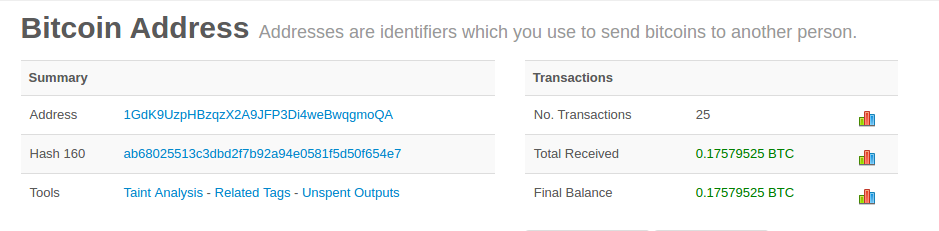


Alice’s transaction to Bob’s Cafe

On the left side of the transaction, the blockchain explorer shows Alice’s Bitcoin address as the "sender." In fact, this information is not in the transaction itself. When the blockchain explorer references the transaction it also references the previous transaction associated with the input and extracts the first output from that older transaction. Within that output is a locking script that locks the UTXO to Alice’s public key hash (a P2PKH script). The blockchain explorer extracted the public key hash and encoded it using Base58Check encoding to produce and display the Bitcoin address that represents that public key.

Similarly, on the right side, the blockchain explorer shows the two outputs; the first to Bob’s Bitcoin address and the second to Alice’s Bitcoin address (as change). Once again, to create these Bitcoin addresses, the blockchain explorer extracted the locking script from each output, recognized it as a P2PKH script, and extracted the public-key-hash from within. Finally, the blockchain explorer reencoded each public key hash with Base58Check to produce and display the Bitcoin addresses.

If you were to click on Bob’s Bitcoin address, the blockchain explorer would show you the view in [figure\_title](#the_balance_of_bobs_bitcoin_address).



The balance of Bob’s Bitcoin address

The blockchain explorer displays the balance of Bob’s Bitcoin address. But nowhere in the Bitcoin system is there a concept of a "balance." Rather, the values displayed here are constructed by the blockchain explorer as follows.

To construct the "Total Received" amount, the blockchain explorer first will decode the Base58Check encoding of the Bitcoin address to retrieve the 160-bit hash of Bob’s public key that is encoded within the address. Then, the blockchain explorer will search through the database of transactions, looking for outputs with P2PKH locking scripts that contain Bob’s public key hash. By summing up the value of all the outputs, the blockchain explorer can produce the total value received.

Constructing the current balance (displayed as "Final Balance") requires a bit more work. The blockchain explorer keeps a separate database of the outputs that are currently unspent, the UTXO set. To maintain this database, the blockchain explorer must monitor the Bitcoin network, add newly created UTXO, and remove spent UTXO, in real time, as they appear in unconfirmed transactions. This is a complicated process that depends on keeping track of transactions as they propagate, as well as maintaining consensus with the Bitcoin network to ensure that the correct chain is followed. Sometimes, the blockchain explorer goes out of sync and its perspective of the UTXO set is incomplete or incorrect.

From the UTXO set, the blockchain explorer sums up the value of all unspent outputs referencing Bob’s public key hash and produces the "Final Balance" number shown to the user.

In order to produce this one image, with these two "balances," the blockchain explorer has to index and search through dozens, hundreds, or even hundreds of thousands of transactions.

In summary, the information presented to users through wallet applications, blockchain explorers, and other bitcoin user interfaces is often composed of higher-level abstractions that are derived by searching many different transactions, inspecting their content, and manipulating the data contained within them. By presenting this simplistic view of bitcoin transactions that resemble bank checks from one sender to one recipient, these applications have to abstract a lot of underlying details. They mostly focus on the common types of transactions: P2PKH with SIGHASH\_ALL signatures on every input. Thus, while bitcoin applications can present more than 80% of all transactions in an easy-to-read manner, they are sometimes stumped by transactions that deviate from the norm. Transactions that contain more complex locking scripts, or different SIGHASH flags, or many inputs and outputs, demonstrate the simplicity and weakness of these abstractions.

Every day, hundreds of transactions that do not contain P2PKH outputs are confirmed on the blockchain. The blockchain explorers often present these with red warning messages saying they cannot decode an address.

As we will see in the next chapter, these are not necessarily strange transactions. They are transactions that contain more complex locking scripts than the common P2PKH. We will learn how to decode and understand more complex scripts and the applications they support next.

== Advanced Transactions and Scripting

=== Introduction

In the previous chapter, we introduced the basic elements of bitcoin transactions and looked at the most common type of transaction script, the P2PKH script. In this chapter we will look at more advanced scripting and how we can use it to build transactions with complex conditions.

First, we will look at *multisignature* scripts. Next, we will examine the second most common transaction script, *Pay-to-Script-Hash*, which opens up a whole world of complex scripts. Then, we will examine new script operators that add a time dimension to bitcoin, through *timelocks*. Finally, we will look at *Segregated Witness*, an architectural change to the structure of transactions.

=== Multisignature

Multisignature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to unlock the funds. This is also known as an M-of-N scheme, where N is the total number of keys and M is the threshold of signatures required for validation. For example, a 2-of-3 multisignature is one where three public keys are listed as potential signers and at least two of those must be used to create signatures for a valid transaction to spend the funds.

At this time, *standard* multisignature scripts are limited to at most 3 listed public keys, meaning you can do anything from a 1-of-1 to a 3-of-3 multisignature or any combination within that range. The limitation to 3 listed keys might be lifted by the time this book is published, so check the IsStandard() function to see what is currently accepted by the network. Note that the limit of 3 keys applies only to standard (also known as "bare") multisignature scripts, not to multisignature scripts wrapped in a Pay-to-Script-Hash (P2SH) script. P2SH multisignature scripts are limited to 15 keys, allowing for up to 15-of-15 multisignature. This limitation is also imposed by the IsStandard() function. We will learn about P2SH in [simpara\_title](#p2sh).

The general form of a locking script setting an M-of-N multisignature condition is:

M <Public Key 1> <Public Key 2> ... <Public Key N> N CHECKMULTISIG

where N is the total number of listed public keys and M is the threshold of required signatures to spend the output.

A locking script setting a 2-of-3 multisignature condition looks like this:

2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

The preceding locking script can be satisfied with an unlocking script containing any combination of two signatures from the private keys corresponding to the three listed public keys:

<Signature B> <Signature C>

The two scripts together would form the combined validation script:

<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

When executed, this combined script will evaluate to TRUE if, and only if, the unlocking script matches the conditions set by the locking script. In this case, the condition is whether the unlocking script has a valid signature from the two private keys that correspond to two of the three public keys set as an encumbrance.

===== A bug in CHECKMULTISIG execution

There is a bug in CHECKMULTISIG's execution that requires a slight workaround. When CHECKMULTISIG executes, it should consume M+N+2 items on the stack as parameters. However, due to the bug, CHECKMULTISIG will pop an extra value or one value more than expected.

Let’s look at this in greater detail using the previous validation example:

<Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

First, CHECKMULTISIG pops the top item, which is N (in this example "3"). Then it pops N items, which are the public keys that can sign. In this example, public keys A, B, and C. Then, it pops one item, which is M, the quorum (how many signatures are needed). Here M = 2. At this point, CHECKMULTISIG should pop the final M items, which are the signatures, and see if they are valid. However, unfortunately, a bug in the implementation causes CHECKMULTISIG to pop one more item (M+1 total) than it should. The extra item is disregarded when checking the signatures so it has no direct effect on CHECKMULTISIG itself. However, an extra value must be present because if it is not present, when CHECKMULTISIG attempts to pop on an empty stack, it will cause a stack error and script failure (marking the transaction as invalid). Because the extra item is disregarded it can be anything, but customarily 0 is used.

Because this bug became part of the consensus rules, it must now be replicated forever. Therefore the correct script validation would look like this:

0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3 CHECKMULTISIG

Thus the unlocking script actually used in multisig is not:

<Signature B> <Signature C>

but instead it is:

0 <Signature B> <Signature C>

From now on, if you see a multisig unlocking script, you should expect to see an extra 0 in the beginning, whose only purpose is as a workaround to a bug that accidentally became a consensus rule.

=== Pay-to-Script-Hash (P2SH)

Pay-to-Script-Hash (P2SH) was introduced in 2012 as a powerful new type of transaction that greatly simplifies the use of complex transaction scripts. To explain the need for P2SH, let’s look at a practical example.

In [Introduction](#ch01_intro_what_is_bitcoin) we introduced Mohammed, an electronics importer based in Dubai. Mohammed’s company uses bitcoin’s multisignature feature extensively for its corporate accounts. Multisignature scripts are one of the most common uses of bitcoin’s advanced scripting capabilities and are a very powerful feature. Mohammed’s company uses a multisignature script for all customer payments, known in accounting terms as "accounts receivable," or AR. With the multisignature scheme, any payments made by customers are locked in such a way that they require at least two signatures to release, from Mohammed and one of his partners or from his attorney who has a backup key. A multisignature scheme like that offers corporate governance controls and protects against theft, embezzlement, or loss.

The resulting script is quite long and looks like this:

2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 CHECKMULTISIG

Although multisignature scripts are a powerful feature, they are cumbersome to use. Given the preceding script, Mohammed would have to communicate this script to every customer prior to payment. Each customer would have to use special bitcoin wallet software with the ability to create custom transaction scripts, and each customer would have to understand how to create a transaction using custom scripts. Furthermore, the resulting transaction would be about five times larger than a simple payment transaction, because this script contains very long public keys. The burden of that extra-large transaction would be borne by the customer in the form of fees. Finally, a large transaction script like this would be carried in the UTXO set in RAM in every full node, until it was spent. All of these issues make using complex locking scripts difficult in practice.

P2SH was developed to resolve these practical difficulties and to make the use of complex scripts as easy as a payment to a Bitcoin address. With P2SH payments, the complex locking script is replaced with its digital fingerprint, a cryptographic hash. When a transaction attempting to spend the UTXO is presented later, it must contain the script that matches the hash, in addition to the unlocking script. In simple terms, P2SH means "pay to a script matching this hash, a script that will be presented later when this output is spent."

In P2SH transactions, the locking script that is replaced by a hash is referred to as the *redeem script* because it is presented to the system at redemption time rather than as a locking script. [table\_title](#without_p2sh) shows the script without P2SH and [table\_title](#with_p2sh) shows the same script encoded with P2SH.

Complex script without P2SH

|  |  |
| --- | --- |
| Locking Script | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG |
| Unlocking Script | 0 Sig1 Sig2 |

Complex script as P2SH

|  |  |
| --- | --- |
| Redeem Script | 2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 CHECKMULTISIG |
| Locking Script | HASH160 <20-byte hash of redeem script> EQUAL |
| Unlocking Script | 0 Sig1 Sig2 <redeem script> |

As you can see from the tables, with P2SH the complex script that details the conditions for spending the output (redeem script) is not presented in the locking script. Instead, only a hash of it is in the locking script and the redeem script itself is presented later, as part of the unlocking script when the output is spent. This shifts the burden in fees and complexity from the sender (who creates the transaction) to the recipient (who unlocks and spends the transaction).

Let’s look at Mohammed’s company, the complex multisignature script, and the resulting P2SH scripts.

First, the multisignature script that Mohammed’s company uses for all incoming payments from customers:

2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public Key> <Attorney Public Key> 5 CHECKMULTISIG

If the placeholders are replaced by actual public keys (shown here as 520-bit numbers starting with 04) you can see that this script becomes very long:

2  


This entire script can instead be represented by a 20-byte cryptographic hash, by first applying the SHA256 hashing algorithm and then applying the RIPEMD160 algorithm on the result.

We use libbitcoin-explorer (bx) on the command-line to produce the script hash, as follows:

echo \  
2 \  
[04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \  
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \  
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \  
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \  
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \  
5 CHECKMULTISIG \  
| bx script-encode | bx sha256 | bx ripemd160  
54c557e07dde5bb6cb791c7a540e0a4796f5e97e

The series of commands above first encodes Mohammed’s multisig redeem script as a serialized hex-encoded bitcoin Script. The next bx command calculates the SHA256 hash of that. The next bx command hashes again with RIPEMD160, producing the final script-hash:

The 20-byte hash of Mohammed’s redeem script is:

54c557e07dde5bb6cb791c7a540e0a4796f5e97e

A P2SH transaction locks the output to this hash instead of the longer redeem script, using the locking script:

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

which, as you can see, is much shorter. Instead of "pay to this 5-key multisignature script," the P2SH equivalent transaction is "pay to a script with this hash." A customer making a payment to Mohammed’s company need only include this much shorter locking script in his payment. When Mohammed and his partners want to spend this UTXO, they must present the original redeem script (the one whose hash locked the UTXO) and the signatures necessary to unlock it, like this:

<Sig1> <Sig2> <2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG>

The two scripts are combined in two stages. First, the redeem script is checked against the locking script to make sure the hash matches:

<2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG> HASH160 <redeem scriptHash> EQUAL

If the redeem script hash matches, the unlocking script is executed on its own, to unlock the redeem script:

<Sig1> <Sig2> 2 PK1 PK2 PK3 PK4 PK5 5 CHECKMULTISIG

Almost all the scripts described in this chapter can only be implemented as P2SH scripts. For example, a 2 of 5 standard multisignature locking script cannot be used directly in the locking script of an UTXO, as IsStandard() would invalidate the transaction. To conform, a P2SH locking script can be used instead, as seen above. A transaction that then includes a P2SH unlocking script can be used to redeem this UTXO and will be valid so long as it does not contain more than 15 public keys.

Remember, because of policy set forth by the IsStandard() function at the time of this writing, standard multisignature scripts are limited to at most 3 listed public keys, while P2SH scripts are limited to at most 15 listed public keys. Standard multisignature scripts can invalidate transactions by way of their locking *or* unlocking script, while P2SH scripts can invalidate transactions by way of their unlocking script *only*. This is because there is no way for IsStandard() to tell if a hash of a redeem script in a locking script includes more signatures than the currently imposed size limitation, so it can only observe the unlocking scripts in transaction inputs.

==== P2SH Addresses

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP-13. P2SH addresses are Base58Check encodings of the 20-byte hash of a script, just like Bitcoin addresses are Base58Check encodings of the 20-byte hash of a public key. P2SH addresses use the version prefix "5," which results in Base58Check-encoded addresses that start with a "3."

For example, Mohammed’s complex script, hashed and Base58Check-encoded as a P2SH address, becomes 39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw. We can confirm that with the bx command:

echo \  
'54c557e07dde5bb6cb791c7a540e0a4796f5e97e'\  
 | bx address-encode -v 5  
39RF6JqABiHdYHkfChV6USGMe6Nsr66Gzw

Now, Mohammed can give this "address" to his customers and they can use almost any bitcoin wallet to make a simple payment, as if it were a Bitcoin address. The 3 prefix gives them a hint that this is a special type of address, one corresponding to a script instead of a public key, but otherwise it works in exactly the same way as a payment to a Bitcoin address.

P2SH addresses hide all of the complexity, so that the person making a payment does not see the script.

==== Benefits of P2SH

The P2SH feature offers the following benefits compared to the direct use of complex scripts in locking outputs:

* Complex scripts are replaced by shorter fingerprints in the transaction output, making the transaction smaller.
* Scripts can be coded as an address, so the sender and the sender’s wallet don’t need complex engineering to implement P2SH.
* P2SH shifts the burden of constructing the script to the recipient, not the sender.
* P2SH shifts the burden in data storage for the long script from the output (which additionally to being stored on the blockchain is in the UTXO set) to the input (only stored on the blockchain).
* P2SH shifts the burden in data storage for the long script from the present time (payment) to a future time (when it is spent).
* P2SH shifts the higher transaction fee costs of a long script from the sender to the recipient, who has to include the long redeem script to spend it.

==== Redeem Script and Validation

Prior to version 0.9.2 of the Bitcoin Core client, Pay-to-Script-Hash was limited to the standard types of bitcoin transaction scripts, by the IsStandard() function. That means that the redeem script presented in the spending transaction could only be one of the standard types: P2PK, P2PKH, or multisig.

As of version 0.9.2 of the Bitcoin Core client, P2SH transactions can contain any valid script, making the P2SH standard much more flexible and allowing for experimentation with many novel and complex types of transactions.

You are not able to put a P2SH inside a P2SH redeem script, because the P2SH specification is not recursive. Also, while it is technically possible to include RETURN (see [simpara\_title](#op_return)) in a redeem script, as nothing in the rules prevents you from doing so, it is of no practical use because executing RETURN during validation will cause the transaction to be marked invalid.

Note that because the redeem script is not presented to the network until you attempt to spend a P2SH output, if you lock an output with the hash of an invalid redeem script it will be processed regardless. The UTXO will be successfully locked. However, you will not be able to spend it because the spending transaction, which includes the redeem script, will not be accepted because it is an invalid script. This creates a risk, because you can lock bitcoin in a P2SH that cannot be spent later. The network will accept the P2SH locking script even if it corresponds to an invalid redeem script, because the script hash gives no indication of the script it represents.

P2SH locking scripts contain the hash of a redeem script, which gives no clues as to the content of the redeem script itself. The P2SH transaction will be considered valid and accepted even if the redeem script is invalid. You might accidentally lock bitcoin in such a way that it cannot later be spent.

=== Data Recording Output (RETURN)

Bitcoin’s distributed and timestamped ledger, the blockchain, has potential uses far beyond payments. Many developers have tried to use the transaction scripting language to take advantage of the security and resilience of the system for applications such as digital notary services, stock certificates, and smart contracts. Early attempts to use bitcoin’s script language for these purposes involved creating transaction outputs that recorded data on the blockchain; for example, to record a digital fingerprint of a file in such a way that anyone could establish proof-of-existence of that file on a specific date by reference to that transaction.

The use of bitcoin’s blockchain to store data unrelated to bitcoin payments is a controversial subject. Many developers consider such use abusive and want to discourage it. Others view it as a demonstration of the powerful capabilities of blockchain technology and want to encourage such experimentation. Those who object to the inclusion of nonpayment data argue that it causes "blockchain bloat," burdening those running full Bitcoin nodes with carrying the cost of disk storage for data that the blockchain was not intended to carry. Moreover, such transactions create UTXO that cannot be spent, using the destination Bitcoin address as a freeform 20-byte field. Because the address is used for data, it doesn’t correspond to a private key and the resulting UTXO can *never* be spent; it’s a fake payment. These transactions that can never be spent are therefore never removed from the UTXO set and cause the size of the UTXO database to forever increase, or "bloat."

In version 0.9 of the Bitcoin Core client, a compromise was reached with the introduction of the RETURN operator. RETURN allows developers to add 80 bytes of nonpayment data to a transaction output. However, unlike the use of "fake" UTXO, the RETURN operator creates an explicitly *provably unspendable* output, which does not need to be stored in the UTXO set. RETURN outputs are recorded on the blockchain, so they consume disk space and contribute to the increase in the blockchain’s size, but they are not stored in the UTXO set and therefore do not bloat the UTXO memory pool and burden full nodes with the cost of more expensive RAM.

RETURN scripts look like this:

RETURN <data>

The data portion is limited to 80 bytes and most often represents a hash, such as the output from the SHA256 algorithm (32 bytes). Many applications put a prefix in front of the data to help identify the application. For example, the [Proof of Existence](https://proofofexistence.com) digital notarization service uses the 8-byte prefix DOCPROOF, which is ASCII encoded as 44 4f 43 50 52 4f 4f 46 in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to RETURN that could possibly be used to "spend" a RETURN output. The whole point of RETURN is that you can’t spend the money locked in that output, and therefore it does not need to be held in the UTXO set as potentially spendable—RETURN is *provably unspendable*. RETURN is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If a RETURN is referenced as an input in a transaction, the script validation engine will halt the execution of the validation script and mark the transaction as invalid. The execution of RETURN essentially causes the script to "RETURN" with a FALSE and halt. Thus, if you accidentally reference a RETURN output as an input in a transaction, that transaction is invalid.

A standard transaction (one that conforms to the IsStandard() checks) can have only one RETURN output. However, a single RETURN output can be combined in a transaction with outputs of any other type.

Two new command-line options have been added in Bitcoin Core as of version 0.10. The option datacarrier controls relay and mining of RETURN transactions, with the default set to "1" to allow them. The option datacarriersize takes a numeric argument specifying the maximum size in bytes of the RETURN script, 83 bytes by default, which, allows for a maximum of 80 bytes of RETURN data plus one byte of RETURN opcode and two bytes of PUSHDATA opcode.

RETURN was initially proposed with a limit of 80 bytes, but the limit was reduced to 40 bytes when the feature was released. In February 2015, in version 0.10 of Bitcoin Core, the limit was raised back to 80 bytes. Nodes may choose not to relay or mine RETURN, or only relay and mine RETURN containing less than 80 bytes of data.

=== Timelocks

Timelocks are restrictions on transactions or outputs that only allow spending after a point in time. Bitcoin has had a transaction-level timelock feature from the beginning. It is implemented by the nLocktime field in a transaction. Two new timelock features were introduced in late 2015 and mid-2016 that offer UTXO-level timelocks. These are CHECKLOCKTIMEVERIFY and CHECKSEQUENCEVERIFY.

Timelocks are useful for postdating transactions and locking funds to a date in the future. More importantly, timelocks extend bitcoin scripting into the dimension of time, opening the door for complex multistep smart contracts.

==== Transaction Locktime (nLocktime)

From the beginning, bitcoin has had a transaction-level timelock feature. Transaction locktime is a transaction-level setting (a field in the transaction data structure) that defines the earliest time that a transaction is valid and can be relayed on the network or added to the blockchain. Locktime is also known as nLocktime from the variable name used in the Bitcoin Core codebase. It is set to zero in most transactions to indicate immediate propagation and execution. If nLocktime is nonzero and below 500 million, it is interpreted as a block height, meaning the transaction is not valid and is not relayed or included in the blockchain prior to the specified block height. If it is greater than or equal to 500 million, it is interpreted as a Unix Epoch timestamp (seconds since Jan-1-1970) and the transaction is not valid prior to the specified time. Transactions with nLocktime specifying a future block or time must be held by the originating system and transmitted to the Bitcoin network only after they become valid. If a transaction is transmitted to the network before the specified nLocktime, the transaction will be rejected by the first node as invalid and will not be relayed to other nodes. The use of nLocktime is equivalent to postdating a paper check.

===== Transaction locktime limitations

nLocktime has the limitation that while it makes it possible to spend some outputs in the future, it does not make it impossible to spend them until that time. Let’s explain that with the following example.

Alice signs a transaction spending one of her outputs to Bob’s address, and sets the transaction nLocktime to 3 months in the future. Alice sends that transaction to Bob to hold. With this transaction Alice and Bob know that:

* Bob cannot transmit the transaction to redeem the funds until 3 months have elapsed.
* Bob may transmit the transaction after 3 months.

However:

* Alice can create another transaction, double-spending the same inputs without a locktime. Thus, Alice can spend the same UTXO before the 3 months have elapsed.
* Bob has no guarantee that Alice won’t do that.

It is important to understand the limitations of transaction nLocktime. The only guarantee is that Bob will not be able to redeem it before 3 months have elapsed. There is no guarantee that Bob will get the funds. To achieve such a guarantee, the timelock restriction must be placed on the UTXO itself and be part of the locking script, rather than on the transaction. This is achieved by the next form of timelock, called Check Lock Time Verify.

==== Check Lock Time Verify (CLTV)

In December 2015, a new form of timelock was introduced to bitcoin as a soft fork upgrade. Based on a specification in BIP-65, a new script operator called *CHECKLOCKTIMEVERIFY* (*CLTV*) was added to the scripting language. CLTV is a per-output timelock, rather than a per-transaction timelock as is the case with nLocktime. This allows for much greater flexibility in the way timelocks are applied.

In simple terms, by adding the CLTV opcode in the redeem script of an output it restricts the output, so that it can only be spent after the specified time has elapsed.

While nLocktime is a transaction-level timelock, CLTV is an output-based timelock.

CLTV doesn’t replace nLocktime, but rather restricts specific UTXO such that they can only be spent in a future transaction with nLocktime set to a greater or equal value.

The CLTV opcode takes one parameter as input, expressed as a number in the same format as nLocktime (either a block height or Unix epoch time). As indicated by the VERIFY suffix, CLTV is the type of opcode that halts execution of the script if the outcome is FALSE. If it results in TRUE, execution continues.

In order to lock an output with CLTV, you insert it into the redeem script of the output in the transaction that creates the output. For example, if Alice is paying Bob’s address, the output would normally contain a P2PKH script like this:

DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG

To lock it to a time, say 3 months from now, the transaction would be a P2SH transaction with a redeem script like this:

<now + 3 months> CHECKLOCKTIMEVERIFY DROP DUP HASH160 <Bob's Public Key Hash> EQUALVERIFY CHECKSIG

where <now {plus} 3 months> is a block height or time value estimated 3 months from the time the transaction is mined: current block height + 12,960 (blocks) or current Unix epoch time + 7,760,000 (seconds). For now, don’t worry about the DROP opcode that follows CHECKLOCKTIMEVERIFY; it will be explained shortly.

When Bob tries to spend this UTXO, he constructs a transaction that references the UTXO as an input. He uses his signature and public key in the unlocking script of that input and sets the transaction nLocktime to be equal to or greater than the timelock in the CHECKLOCKTIMEVERIFY Alice set. Bob then broadcasts the transaction on the Bitcoin network.

Bob’s transaction is evaluated as follows. If the CHECKLOCKTIMEVERIFY parameter Alice set is less than or equal to the spending transaction’s nLocktime, script execution continues (acts as if a “no operation” or NOP opcode was executed). Otherwise, script execution halts and the transaction is deemed invalid.

More precisely, CHECKLOCKTIMEVERIFY fails and halts execution, marking the transaction invalid if (source: BIP-65):

1. the stack is empty; or
2. the top item on the stack is less than 0; or
3. the timelock type (height versus timestamp) of the top stack item and the nLocktime field are not the same; or
4. the top stack item is greater than the transaction’s nLocktime field; or
5. the nSequence field of the input is 0xffffffff.

CLTV and nLocktime use the same format to describe timelocks, either a block height or the time elapsed in seconds since Unix epoch. Critically, when used together, the format of nLocktime must match that of CLTV in the outputs—they must both reference either block height or time in seconds.

After execution, if CLTV is satisfied, the time parameter that preceded it remains as the top item on the stack and may need to be dropped, with DROP, for correct execution of subsequent script opcodes. You will often see CHECKLOCKTIMEVERIFY followed by DROP in scripts for this reason.

By using nLocktime in conjunction with CLTV, the scenario described in [simpara\_title](#locktime_limitations) changes. Alice can no longer spend the money (because it’s locked with Bob’s key) and Bob cannot spend it before the 3-month locktime has expired.

By introducing timelock functionality directly into the scripting language, CLTV allows us to develop some very interesting complex scripts.

The standard is defined in [BIP-65 (CHECKLOCKTIMEVERIFY)](https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki).

==== Relative Timelocks

nLocktime and CLTV are both *absolute timelocks* in that they specify an absolute point in time. The next two timelock features we will examine are *relative timelocks* in that they specify, as a condition of spending an output, an elapsed time from the confirmation of the output in the blockchain.

Relative timelocks are useful because they allow a chain of two or more interdependent transactions to be held off chain, while imposing a time constraint on one transaction that is dependent on the elapsed time from the confirmation of a previous transaction. In other words, the clock doesn’t start counting until the UTXO is recorded on the blockchain. This functionality is especially useful in bidirectional state channels and Lightning Networks, as we will see in [simpara\_title](#state_channels).

Relative timelocks, like absolute timelocks, are implemented with both a transaction-level feature and a script-level opcode. The transaction-level relative timelock is implemented as a consensus rule on the value of nSequence, a transaction field that is set in every transaction input. Script-level relative timelocks are implemented with the CHECKSEQUENCEVERIFY (CSV) opcode.

Relative timelocks are implemented according to the specifications in [BIP-68, Relative lock-time using consensus-enforced sequence numbers](https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki) and [BIP-112, CHECKSEQUENCEVERIFY](https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki).

BIP-68 and BIP-112 were activated in May 2016 as a soft fork upgrade to the consensus rules.

==== Relative Timelocks with nSequence

Relative timelocks can be set on each input of a transaction, by setting the nSequence field in each input.

===== Original meaning of nSequence

The nSequence field was originally intended (but never properly implemented) to allow modification of transactions in the mempool. In that use, a transaction containing inputs with nSequence value below 232 - 1 (0xFFFFFFFF) indicated a transaction that was not yet "finalized." Such a transaction would be held in the mempool until it was replaced by another transaction spending the same inputs with a higher nSequence value. Once a transaction was received whose inputs had an nSequence value of 0xFFFFFFFF it would be considered "finalized" and mined.

The original meaning of nSequence was never properly implemented and the value of nSequence is customarily set to 0xFFFFFFFF in transactions that do not utilize timelocks. For transactions with nLocktime or CHECKLOCKTIMEVERIFY, the nSequence value must be set to less than 231 for the timelock guards to have an effect, as explained below.

===== nSequence as a consensus-enforced relative timelock

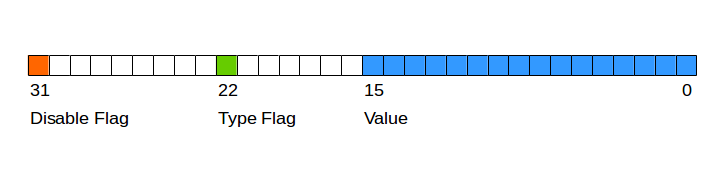
Since the activation of BIP-68, new consensus rules apply for any transaction containing an input whose nSequence value is less than 231 (bit 1<<31 is not set). Programmatically, that means that if the most significant bit (bit 1<<31) is not set, it is a flag that means "relative locktime." Otherwise (bit 1<<31 set), the nSequence value is reserved for other uses such as enabling CHECKLOCKTIMEVERIFY, nLocktime, Opt-In-Replace-By-Fee, and other future developments.

Transaction inputs with nSequence values less than 231 are interpreted as having a relative timelock. Such a transaction is only valid once the input has aged by the relative timelock amount. For example, a transaction with one input with an nSequence relative timelock of 30 blocks is only valid when at least 30 blocks have elapsed from the time the UTXO referenced in the input was mined. Since nSequence is a per-input field, a transaction may contain any number of timelocked inputs, all of which must have sufficiently aged for the transaction to be valid. A transaction can include both timelocked inputs (nSequence < 231) and inputs without a relative timelock (nSequence >= 231).

The nSequence value is specified in either blocks or seconds, but in a slightly different format than we saw used in nLocktime. A type-flag is used to differentiate between values counting blocks and values counting time in seconds. The type-flag is set in the 23rd least-significant bit (i.e., value 1<<22). If the type-flag is set, then the nSequence value is interpreted as a multiple of 512 seconds. If the type-flag is not set, the nSequence value is interpreted as a number of blocks.

When interpreting nSequence as a relative timelock, only the 16 least significant bits are considered. Once the flags (bits 32 and 23) are evaluated, the nSequence value is usually "masked" with a 16-bit mask (e.g., nSequence & 0x0000FFFF).

[figure\_title](#bip_68_def_of_nseq) shows the binary layout of the nSequence value, as defined by BIP-68.



BIP-68 definition of nSequence encoding (Source: BIP-68)

Relative timelocks based on consensus enforcement of the nSequence value are defined in BIP-68.

The standard is defined in [BIP-68, Relative lock-time using consensus-enforced sequence numbers](https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki).

==== Relative Timelocks with CSV

Just like CLTV and nLocktime, there is a script opcode for relative timelocks that leverages the nSequence value in scripts. That opcode is CHECKSEQUENCEVERIFY, commonly referred to as CSV for short.

The CSV opcode when evaluated in an UTXO’s redeem script allows spending only in a transaction whose input nSequence value is greater than or equal to the CSV parameter. Essentially, this restricts spending the UTXO until a certain number of blocks or seconds have elapsed relative to the time the UTXO was mined.

As with CLTV, the value in CSV must match the format in the corresponding nSequence value. If CSV is specified in terms of blocks, then so must nSequence. If CSV is specified in terms of seconds, then so must nSequence.

Relative timelocks with CSV are especially useful when several (chained) transactions are created and signed, but not propagated, when they’re kept "off-chain." A child transaction cannot be used until the parent transaction has been propagated, mined, and aged by the time specified in the relative timelock. One application of this use case can be seen in [simpara\_title](#state_channels) and [simpara\_title](#lightning_network).

CSV is defined in detail in [BIP-112, CHECKSEQUENCEVERIFY](https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki).

==== Median-Time-Past

As part of the activation of relative timelocks, there was also a change in the way "time" is calculated for timelocks (both absolute and relative). In bitcoin there is a subtle, but very significant, difference between wall time and consensus time. Bitcoin is a decentralized network, which means that each participant has his or her own perspective of time. Events on the network do not occur instantaneously everywhere. Network latency must be factored into the perspective of each node. Eventually everything is synchronized to create a common ledger. Bitcoin reaches consensus every 10 minutes about the state of the ledger as it existed in the *past*.

The timestamps set in block headers are set by the miners. There is a certain degree of latitude allowed by the consensus rules to account for differences in clock accuracy between decentralized nodes. However, this creates an unfortunate incentive for miners to lie about the time in a block so as to earn extra fees by including timelocked transactions that are not yet mature. See the following section for more information.

To remove the incentive to lie and strengthen the security of timelocks, a BIP was proposed and activated at the same time as the BIPs for relative timelocks. This is BIP-113, which defines a new consensus measurement of time called *Median-Time-Past*.

Median-Time-Past is calculated by taking the timestamps of the last 11 blocks and finding the median. That median time then becomes consensus time and is used for all timelock calculations. By taking the midpoint from approximately two hours in the past, the influence of any one block’s timestamp is reduced. By incorporating 11 blocks, no single miner can influence the timestamps in order to gain fees from transactions with a timelock that hasn’t yet matured.

Median-Time-Past changes the implementation of time calculations for nLocktime, CLTV, nSequence, and CSV. The consensus time calculated by Median-Time-Past is always approximately one hour behind wall clock time. If you create timelock transactions, you should account for it when estimating the desired value to encode in nLocktime, nSequence, CLTV, and CSV.

Median-Time-Past is specified in [BIP-113](https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki).

==== Timelock Defense Against Fee Sniping

Fee-sniping is a theoretical attack scenario, where miners attempting to rewrite past blocks "snipe" higher-fee transactions from future blocks to maximize their profitability.

For example, let’s say the highest block in existence is block #100,000. If instead of attempting to mine block #100,001 to extend the chain, some miners attempt to remine #100,000. These miners can choose to include any valid transaction (that hasn’t been mined yet) in their candidate block #100,000. They don’t have to remine the block with the same transactions. In fact, they have the incentive to select the most profitable (highest fee per kB) transactions to include in their block. They can include any transactions that were in the "old" block #100,000, as well as any transactions from the current mempool. Essentially they have the option to pull transactions from the "present" into the rewritten "past" when they re-create block #100,000.

Today, this attack is not very lucrative, because block reward is much higher than total fees per block. But at some point in the future, transaction fees will be the majority of the mining reward (or even the entirety of the mining reward). At that time, this scenario becomes inevitable.

To prevent "fee sniping," when Bitcoin Core creates transactions, it uses nLocktime to limit them to the "next block," by default. In our scenario, Bitcoin Core would set nLocktime to 100,001 on any transaction it created. Under normal circumstances, this nLocktime has no effect—the transactions could only be included in block #100,001 anyway; it’s the next block.

But under a blockchain fork/double-spend attack, the miners would not be able to pull high-fee transactions from the mempool, because all those transactions would be timelocked to block #100,001. They can only remine #100,000 with whatever transactions were valid at that time, essentially gaining no new fees.

To achieve this, Bitcoin Core sets the nLocktime on all new transactions to <current block # + 1> and sets the nSequence on all the inputs to 0xFFFFFFFE to enable nLocktime.

=== Scripts with Flow Control (Conditional Clauses)

One of the more powerful features of Bitcoin Script is flow control, also known as conditional clauses. You are probably familiar with flow control in various programming languages that use the construct IF...THEN...ELSE. Bitcoin conditional clauses look a bit different, but are essentially the same construct.

At a basic level, bitcoin conditional opcodes allow us to construct a redeem script that has two ways of being unlocked, depending on a TRUE/FALSE outcome of evaluating a logical condition. For example, if x is TRUE, the redeem script is A and the ELSE redeem script is B.

Additionally, bitcoin conditional expressions can be "nested" indefinitely, meaning that a conditional clause can contain another within it, which contains another, etc. Bitcoin Script flow control can be used to construct very complex scripts with hundreds or even thousands of possible execution paths. There is no limit to nesting, but consensus rules impose a limit on the maximum size, in bytes, of a script.

Bitcoin implements flow control using the IF, ELSE, ENDIF, and NOTIF opcodes. Additionally, conditional expressions can contain boolean operators such as BOOLAND, BOOLOR, and NOT.

At first glance, you may find the bitcoin’s flow control scripts confusing. That is because Bitcoin Script is a stack language. The same way that 1 {plus} 1 looks "backward" when expressed as 1 1 ADD, flow control clauses in bitcoin also look "backward."

In most traditional (procedural) programming languages, flow control looks like this:

**Pseudocode of flow control in most programming languages**

if (condition):  
 code to run when condition is true  
else:  
 code to run when condition is false  
code to run in either case

In a stack-based language like Bitcoin Script, the logical condition comes before the IF, which makes it look "backward," like this:

**Bitcoin Script flow control**

condition  
IF  
 code to run when condition is true  
ELSE  
 code to run when condition is false  
ENDIF  
code to run in either case

When reading Bitcoin Script, remember that the condition being evaluated comes *before* the IF opcode.

==== Conditional Clauses with VERIFY Opcodes

Another form of conditional in Bitcoin Script is any opcode that ends in VERIFY. The VERIFY suffix means that if the condition evaluated is not TRUE, execution of the script terminates immediately and the transaction is deemed invalid.

Unlike an IF clause, which offers alternative execution paths, the VERIFY suffix acts as a *guard clause*, continuing only if a precondition is met.

For example, the following script requires Bob’s signature and a pre-image (secret) that produces a specific hash. Both conditions must be satisfied to unlock it:

**A redeem script with an EQUALVERIFY guard clause.**

HASH160 <expected hash> EQUALVERIFY <Bob's Pubkey> CHECKSIG

To redeem this, Bob must construct an unlocking script that presents a valid pre-image and a signature:

**An unlocking script to satisfy the above redeem script**

<Bob's Sig> <hash pre-image>

Without presenting the pre-image, Bob can’t get to the part of the script that checks for his signature.

This script can be written with an IF instead:

**A redeem script with an IF guard clause**

HASH160 <expected hash> EQUAL  
IF  
 <Bob's Pubkey> CHECKSIG  
ENDIF

Bob’s unlocking script is identical:

**An unlocking script to satisfy the above redeem script**

<Bob's Sig> <hash pre-image>

The script with IF does the same thing as using an opcode with a VERIFY suffix; they both operate as guard clauses. However, the VERIFY construction is more efficient, using two fewer opcodes.

So, when do we use VERIFY and when do we use IF? If all we are trying to do is to attach a precondition (guard clause), then VERIFY is better. If, however, we want to have more than one execution path (flow control), then we need an IF...ELSE flow control clause.

An opcode such as EQUAL will push the result (TRUE/FALSE) onto the stack, leaving it there for evaluation by subsequent opcodes. In contrast, the opcode EQUALVERIFY suffix does not leave anything on the stack. Opcodes that end in VERIFY do not leave the result on the stack.

==== Using Flow Control in Scripts

A very common use for flow control in Bitcoin Script is to construct a redeem script that offers multiple execution paths, each a different way of redeeming the UTXO.

Let’s look at a simple example, where we have two signers, Alice and Bob, and either one is able to redeem. With multisig, this would be expressed as a 1-of-2 multisig script. For the sake of demonstration, we will do the same thing with an IF clause:

IF  
 <Alice's Pubkey> CHECKSIG  
ELSE  
 <Bob's Pubkey> CHECKSIG  
ENDIF

Looking at this redeem script, you may be wondering: "Where is the condition? There is nothing preceding the IF clause!"

The condition is not part of the redeem script. Instead, the condition will be offered in the unlocking script, allowing Alice and Bob to "choose" which execution path they want.

Alice redeems this with the unlocking script:

<Alice's Sig> 1

The 1 at the end serves as the condition (TRUE) that will make the IF clause execute the first redemption path for which Alice has a signature.

For Bob to redeem this, he would have to choose the second execution path by giving a FALSE value to the IF clause:

<Bob's Sig> 0

Bob’s unlocking script puts a 0 on the stack, causing the IF clause to execute the second (ELSE) script, which requires Bob’s signature.

Since IF clauses can be nested, we can create a "maze" of execution paths. The unlocking script can provide a "map" selecting which execution path is actually executed:

IF  
 script A  
ELSE  
 IF  
 script B  
 ELSE  
 script C  
 ENDIF  
ENDIF

In this scenario, there are three execution paths (script A, script B, and script C). The unlocking script provides a path in the form of a sequence of TRUE or FALSE values. To select path script B, for example, the unlocking script must end in 1 0 (TRUE, FALSE). These values will be pushed onto the stack, so that the second value (FALSE) ends up at the top of the stack. The outer IF clause pops the FALSE value and executes the first ELSE clause. Then the TRUE value moves to the top of the stack and is evaluated by the inner (nested) IF, selecting the B execution path.

Using this construct, we can build redeem scripts with tens or hundreds of execution paths, each offering a different way to redeem the UTXO. To spend, we construct an unlocking script that navigates the execution path by putting the appropriate TRUE and FALSE values on the stack at each flow control point.

=== Complex Script Example

In this section we combine many of the concepts from this chapter into a single example.

Our example uses the story of Mohammed, the company owner in Dubai who is operating an import/export business.

In this example, Mohammed wishes to construct a company capital account with flexible rules. The scheme he creates requires different levels of authorization depending on timelocks. The participants in the multisig scheme are Mohammed, his two partners Saeed and Zaira, and their company lawyer Abdul. The three partners make decisions based on a majority rule, so two of the three must agree. However, in the case of a problem with their keys, they want their lawyer to be able to recover the funds with one of the three partner signatures. Finally, if all partners are unavailable or incapacitated for a while, they want the lawyer to be able to manage the account directly.

Here’s the redeem script that Mohammed designs to achieve this (line number prefix as XX):

**Variable Multi-Signature with Timelock**

01 IF  
02 IF  
03 2  
04 ELSE  
05 <30 days> CHECKSEQUENCEVERIFY DROP  
06 <Abdul the Lawyer's Pubkey> CHECKSIGVERIFY  
07 1  
08 ENDIF  
09 <Mohammed's Pubkey> <Saeed's Pubkey> <Zaira's Pubkey> 3 CHECKMULTISIG  
10 ELSE  
11 <90 days> CHECKSEQUENCEVERIFY DROP  
12 <Abdul the Lawyer's Pubkey> CHECKSIG  
13 ENDIF

Mohammed’s script implements three execution paths using nested IF...ELSE flow control clauses.

In the first execution path, this script operates as a simple 2-of-3 multisig with the three partners. This execution path consists of lines 3 and 9. Line 3 sets the quorum of the multisig to 2 (2-of-3). This execution path can be selected by putting TRUE TRUE at the end of the unlocking script:

**Unlocking script for the first execution path (2-of-3 multisig)**

0 <Mohammed's Sig> <Zaira's Sig> TRUE TRUE

The 0 at the beginning of this unlocking script is because of a bug in CHECKMULTISIG that pops an extra value from the stack. The extra value is disregarded by the CHECKMULTISIG, but it must be present or the script fails. Pushing 0 (customarily) is a workaround to the bug, as described in [simpara\_title](#multisig_bug).

The second execution path can only be used after 30 days have elapsed from the creation of the UTXO. At that time, it requires the signature of Abdul the lawyer and one of the three partners (a 1-of-3 multisig). This is achieved by line 7, which sets the quorum for the multisig to 1. To select this execution path, the unlocking script would end in FALSE TRUE:

**Unlocking script for the second execution path (Lawyer + 1-of-3)**

0 <Abdul the Lawyer's Sig> <Saeed's Sig> FALSE TRUE

Why FALSE TRUE? Isn’t that backward? Because the two values are pushed on to the stack, with FALSE pushed first, then TRUE pushed second. TRUE is therefore popped *first* by the first IF opcode.

Finally, the third execution path allows Abdul the lawyer to spend the funds alone, but only after 90 days. To select this execution path, the unlocking script has to end in FALSE:

**Unlocking script for the third execution path (Lawyer only)**

<Abdul the Lawyer's Sig> FALSE

Try running the script on paper to see how it behaves on the stack.

A few more things to consider when reading this example. See if you can find the answers:

* Why can’t the lawyer redeem the third execution path at any time by selecting it with FALSE on the unlocking script?
* How many execution paths can be used 5, 35, and 105 days, respectively, after the UTXO is mined?
* Are the funds lost if the lawyer loses his key? Does your answer change if 91 days have elapsed?
* How do the partners "reset" the clock every 29 or 89 days to prevent the lawyer from accessing the funds?
* Why do some CHECKSIG opcodes in this script have the VERIFY suffix while others don’t?

=== Segregated Witness

Segregated Witness (segwit) is an upgrade to the bitcoin consensus rules and network protocol, proposed and implemented as a BIP-9 soft-fork that was activated on bitcoin’s mainnet on August 1st, 2017.

In cryptography, the term "witness" is used to describe a solution to a cryptographic puzzle. In bitcoin terms, the witness satisfies a cryptographic condition placed on an unspent transaction output (UTXO).

In the context of bitcoin, a digital signature is *one type of witness*, but a witness is more broadly any solution that can satisfy the conditions imposed on an UTXO and unlock that UTXO for spending. The term “witness” is a more general term for an “unlocking script” or “scriptSig.”

Before segwit’s introduction, every input in a transaction was followed by the witness data that unlocked it. The witness data was embedded in the transaction as part of each input. The term *segregated witness*, or *segwit* for short, simply means separating the signature or unlocking script of a specific output. Think "separate scriptSig," or “separate signature” in the simplest form.

Segregated Witness therefore is an architectural change to bitcoin that aims to move the witness data from the scriptSig (unlocking script) field of a transaction into a separate *witness* data structure that accompanies a transaction. Clients may request transaction data with or without the accompanying witness data.

In this section we will look at some of the benefits of Segregated Witness, describe the mechanism used to deploy and implement this architecture change, and demonstrate the use of Segregated Witness in transactions and addresses.

Segregated Witness is defined by the following BIPs:

[BIP-141](https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki)

The main definition of Segregated Witness.

[BIP-143](https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki)

Transaction Signature Verification for Version 0 Witness Program

[BIP-144](https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki)

Peer Services—New network messages and serialization formats

[BIP-145](https://github.com/bitcoin/bips/blob/master/bip-0145.mediawiki)

getblocktemplate Updates for Segregated Witness (for mining)

[BIP-173](https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki)

Base32 address format for native v0-16 witness outputs

==== Why Segregated Witness?

Segregated Witness is an architectural change that has several effects on the scalability, security, economic incentives, and performance of bitcoin:

Transaction Malleability

By moving the witness outside the transaction data, the transaction hash used as an identifier no longer includes the witness data. Since the witness data is the only part of the transaction that can be modified by a third party (see [simpara\_title](#segwit_txid)), removing it also removes the opportunity for transaction malleability attacks. With Segregated Witness, transaction hashes become immutable by anyone other than the creator of the transaction, which greatly improves the implementation of many other protocols that rely on advanced bitcoin transaction construction, such as payment channels, chained transactions, and lightning networks.

Script Versioning

With the introduction of Segregated Witness scripts, every locking script is preceded by a *script version* number, similar to how transactions and blocks have version numbers. The addition of a script version number allows the scripting language to be upgraded in a backward-compatible way (i.e., using soft fork upgrades) to introduce new script operands, syntax, or semantics. The ability to upgrade the scripting language in a nondisruptive way will greatly accelerate the rate of innovation in bitcoin.

Network and Storage Scaling

The witness data is often a big contributor to the total size of a transaction. More complex scripts such as those used for multisig or payment channels are very large. In some cases these scripts account for the majority (more than 75%) of the data in a transaction. By moving the witness data outside the transaction data, Segregated Witness improves bitcoin’s scalability. Nodes can prune the witness data after validating the signatures, or ignore it altogether when doing simplified payment verification. The witness data doesn’t need to be transmitted to all nodes and does not need to be stored on disk by all nodes.

Signature Verification Optimization

Segregated Witness upgrades the signature functions (CHECKSIG, CHECKMULTISIG, etc.) to reduce the algorithm’s computational complexity. Before segwit, the algorithm used to produce a signature required a number of hash operations that was proportional to the size of the transaction. Data-hashing computations increased in O(n2) with respect to the number of signature operations, introducing a substantial computational burden on all nodes verifying the signature. With segwit, the algorithm is changed to reduce the complexity to O(n).

Offline Signing Improvement

Segregated Witness signatures incorporate the value (amount) referenced by each input in the hash that is signed. Previously, an offline signing device, such as a hardware wallet, would have to verify the amount of each input before signing a transaction. This was usually accomplished by streaming a large amount of data about the previous transactions referenced as inputs. Since the amount is now part of the commitment hash that is signed, an offline device does not need the previous transactions. If the amounts do not match (are misrepresented by a compromised online system), the signature will be invalid.

==== How Segregated Witness Works

At first glance, Segregated Witness appears to be a change to how transactions are constructed and therefore a transaction-level feature, but it is not. Rather, Segregated Witness is a change to how individual UTXO are spent and therefore is a per-output feature.

A transaction can spend Segregated Witness outputs or traditional (inline-witness) outputs or both. Therefore, it does not make much sense to refer to a transaction as a “Segregated Witness transaction.” Rather we should refer to specific transaction outputs as “Segregated Witness outputs."

When a transaction spends an UTXO, it must provide a witness. In a traditional UTXO, the locking script requires that witness data be provided *inline* in the input part of the transaction that spends the UTXO. A Segregated Witness UTXO, however, specifies a locking script that can be satisfied with witness data outside of the input (segregated).

==== Soft Fork (Backward Compatibility)

Segregated Witness is a significant change to the way outputs and transactions are architected. Such a change would normally require a simultaneous change in every Bitcoin node and wallet to change the consensus rules—what is known as a hard fork. Instead, segregated witness is introduced with a much less disruptive change, which is backward compatible, known as a soft fork. This type of upgrade allows non-upgraded software to ignore the changes and continue to operate without any disruption.

Segregated Witness outputs are constructed so that older systems that are not segwit-aware can still validate them. To an old wallet or node, a Segregated Witness output looks like an output that *anyone can spend*. Such outputs can be spent with an empty signature, therefore the fact that there is no signature inside the transaction (it is segregated) does not invalidate the transaction. Newer wallets and mining nodes, however, see the Segregated Witness output and expect to find a valid witness for it in the transaction’s witness data.

==== Segregated Witness Output and Transaction Examples

Let’s look at some of our example transactions and see how they would change with Segregated Witness. We’ll first look at how a Pay-to-Public-Key-Hash (P2PKH) payment is transformed with the Segregated Witness program. Then, we’ll look at the Segregated Witness equivalent for Pay-to-Script-Hash (P2SH) scripts. Finally, we’ll look at how both of the preceding Segregated Witness programs can be embedded inside a P2SH script.

===== Pay-to-Witness-Public-Key-Hash (P2WPKH)

In [simpara\_title](#cup_of_coffee), Alice created a transaction to pay Bob for a cup of coffee. That transaction created a P2PKH output with a value of 0.015 BTC that was spendable by Bob. The output’s script looks like this:

**Example P2PKH output script**

DUP HASH160 ab68025513c3dbd2f7b92a94e0581f5d50f654e7 EQUALVERIFY CHECKSIG

With Segregated Witness, Alice would create a Pay-to-Witness-Public-Key-Hash (P2WPKH) script, which looks like this:

**Example P2WPKH output script**

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

As you can see, a Segregated Witness output’s locking script is much simpler than a traditional output. It consists of two values that are pushed on to the script evaluation stack. To an old (nonsegwit-aware) Bitcoin client, the two pushes would look like an output that anyone can spend and does not require a signature (or rather, can be spent with an empty signature). To a newer, segwit-aware client, the first number (0) is interpreted as a version number (the *witness version*) and the second part (20 bytes) is the equivalent of a locking script known as a *witness program*. The 20-byte witness program is simply the hash of the public key, as in a P2PKH script.

Now, let’s look at the corresponding transaction that Bob uses to spend this output. For the original script (nonsegwit), Bob’s transaction would have to include a signature within the transaction input:

**Decoded transaction showing a P2PKH output being spent with a signature**

[...]  
“Vin” : [  
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",  
"vout": 0,  
 "scriptSig": “<Bob’s scriptSig>”,  
]  
[...]

However, to spend the Segregated Witness output, the transaction has no signature in the input part. Instead, Bob’s transaction has an empty scriptSig in the transaction data (the first part of a transaction, which includes the input part) and includes his signature in the witness data (the second part of a transaction, which is separated from the transaction data):

**Decoded transaction showing a P2WPKH output being spent with separate witness data**

[...]  
“Vin” : [  
"txid": "0627052b6f28912f2703066a912ea577f2ce4da4caa5a5fbd8a57286c345c2f2",  
"vout": 0,  
 "scriptSig": “”,  
]  
[...]  
“witness”: “<Bob’s witness data>”  
[...]

===== Wallet construction of P2WPKH

It is extremely important to note that P2WPKH should only be created by the payee (recipient) and not converted by the sender from a known public key, P2PKH script, or address. The receiver has no way of knowing if the sender’s wallet has the ability to construct segwit transactions and spend P2WPKH outputs.

Additionally, P2WPKH outputs must be constructed from the hash of a *compressed* public key. Uncompressed public keys are nonstandard in segwit and may be explicitly disabled by a future soft fork. If the hash used in the P2WPKH came from an uncompressed public key, it may be unspendable and you may lose funds. P2WPKH outputs should be created by the payee’s wallet by deriving a compressed public key from their private key.

P2WPKH should be constructed by the payee (recipient) by converting a compressed public key to a P2WPKH hash. You should never transform a P2PKH script, Bitcoin address, or uncompressed public key to a P2WPKH witness script.

===== Pay-to-Witness-Script-Hash (P2WSH)

The second type of witness program corresponds to a Pay-to-Script-Hash (P2SH) script. We saw this type of script in [simpara\_title](#p2sh). In that example, P2SH was used by Mohammed’s company to express a multisignature script. Payments to Mohammed’s company were encoded with a locking script like this:

**Example P2SH output script**

HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e EQUAL

This P2SH script references the hash of a *redeem script* that defines a 2-of-5 multisignature requirement to spend funds. To spend this output, Mohammed’s company would present the redeem script (whose hash matches the script hash in the P2SH output) and the signatures necessary to satisfy that redeem script, all inside the transaction input:

**Decoded transaction showing a P2SH output being spent**

[...]  
“Vin” : [  
"txid": "abcdef12345...",  
"vout": 0,  
 "scriptSig": “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”,  
]

Now, let’s look at how this entire example would be upgraded to segwit. If Mohammed’s customers were using a segwit-compatible wallet, they would make a payment, creating a Pay-to-Witness-Script-Hash (P2WSH) output that would look like this:

**Example P2WSH output script**

0 a9b7b38d972cabc7961dbfbcb841ad4508d133c47ba87457b4a0e8aae86dbb89

Again, as with the example of P2WPKH, you can see that the Segregated Witness equivalent script is a lot simpler and omits the various script operands that you see in P2SH scripts. Instead, the Segregated Witness program consists of two values pushed to the stack: a witness version (0) and the 32-byte SHA256 hash of the redeem script.

Mohammed’s company can spend the P2WSH output by presenting the correct redeem script and sufficient signatures to satisfy it. Both the redeem script and the signatures would be segregated *outside* the spending transaction data as part of the witness data. Within the transaction input, Mohammed’s wallet would put an empty scriptSig:

**Decoded transaction showing a P2WSH output being spent with separate witness data**

[...]  
“Vin” : [  
"txid": "abcdef12345...",  
"vout": 0,  
 "scriptSig": “”,  
]  
[...]  
“witness”: “<SigA> <SigB> <2 PubA PubB PubC PubD PubE 5 CHECKMULTISIG>”  
[...]

While P2SH uses the 20-byte RIPEMD160(SHA256(script)) hash, the P2WSH witness program uses a 32-byte SHA256(script) hash. This difference in the selection of the hashing algorithm is deliberate and provides stronger security to P2WSH (128 bits of security in P2WSH versus 80 bits of security in P2SH). It is also used to differentiate between the two types of witness programs (P2WPKH and P2WSH) by using the length of the hash (see below).

===== Differentiating between P2WPKH and P2WSH

In the previous two sections, we demonstrated two types of witness programs: [simpara\_title](#p2wpkh) and [simpara\_title](#p2wsh). Both types of witness programs consist of a single byte version number followed by a longer hash. They look very similar, but are interpreted very differently: one is interpreted as a public key hash, which is satisfied by a signature and the other as a script hash, which is satisfied by a redeem script. The critical difference between them is the length of the hash:

* The public key hash in P2WPKH is 20 bytes
* The script hash in P2WSH is 32 bytes

This is the one difference that allows a wallet to differentiate between the two types of witness programs. By looking at the length of the hash, a wallet can determine what type of witness program it is, P2WPKH or P2WSH.

==== Upgrading to Segregated Witness

As we can see from the previous examples, upgrading to Segregated Witness is a two-step process. First, wallets must create special segwit type outputs. Then, these outputs can be spent by wallets that know how to construct Segregated Witness transactions. In the examples, Alice’s wallet was segwit-aware and able to create special outputs with Segregated Witness scripts. Bob’s wallet is also segwit-aware and able to spend those outputs. What may not be obvious from the example is that in practice, Alice’s wallet needs to *know* that Bob uses a segwit-aware wallet and can spend these outputs. Otherwise, if Bob’s wallet is not upgraded and Alice tries to make segwit payments to Bob, Bob’s wallet will not be able to detect these payments.

For P2WPKH and P2WSH payment types, both the sender and the recipient wallets need to be upgraded to be able to use segwit. Furthermore, the sender’s wallet needs to know that the recipient’s wallet is segwit-aware.

Segregated Witness will not be implemented simultaneously across the entire network. Rather, Segregated Witness is implemented as a backward-compatible upgrade, where *old and new clients can coexist*. Wallet developers will independently upgrade wallet software to add segwit capabilities. The P2WPKH and P2WSH payment types are used when both sender and recipient are segwit-aware. The traditional P2PKH and P2SH will continue to work for non-upgraded wallets. That leaves two important scenarios, which are addressed in the next section:

* Ability of a sender’s wallet that is not segwit-aware to make a payment to a recipient’s wallet that can process segwit transactions
* Ability of a sender’s wallet that is segwit-aware to recognize and distinguish between recipients that are segwit-aware and ones that are not, by their *addresses*.

===== Embedding Segregated Witness inside P2SH

Let’s assume, for example, that Alice’s wallet is not upgraded to segwit, but Bob’s wallet is upgraded and can handle segwit transactions. Alice and Bob can use "old" non-segwit transactions. But Bob would likely want to use segwit to reduce transaction fees, taking advantage of the discount that applies to witness data.

In this case Bob’s wallet can construct a P2SH address that contains a segwit script inside it. Alice’s wallet sees this as a "normal" P2SH address and can make payments to it without any knowledge of segwit. Bob’s wallet can then spend this payment with a segwit transaction, taking full advantage of segwit and reducing transaction fees.

Both forms of witness scripts, P2WPKH and P2WSH, can be embedded in a P2SH address. The first is noted as P2SH(P2WPKH) and the second is noted as P2SH(P2WSH).

===== Pay-to-Witness-Public-Key-Hash inside Pay-to-Script-Hash

The first form of witness script we will examine is P2SH(P2WPKH). This is a Pay-to-Witness-Public-Key-Hash witness program, embedded inside a Pay-to-Script-Hash script, so that it can be used by a wallet that is not aware of segwit.

Bob’s wallet constructs a P2WPKH witness program with Bob’s public key. This witness program is then hashed and the resulting hash is encoded as a P2SH script. The P2SH script is converted to a Bitcoin address, one that starts with a "3," as we saw in the [simpara\_title](#p2sh) section.

Bob’s wallet starts with the P2WPKH witness program we saw earlier:

**Bob’s P2WPKH witness program**

0 ab68025513c3dbd2f7b92a94e0581f5d50f654e7

The P2WPKH witness program consists of the witness version and Bob’s 20-byte public key hash.

Bob’s wallet then hashes the preceding witness program, first with SHA256, then with RIPEMD160, producing another 20-byte hash.

Let’s use bx on the command-line to replicate that:

**HASH160 of the P2WPKH witness program**

echo \  
'0 [ab68025513c3dbd2f7b92a94e0581f5d50f654e7]'\  
 | bx script-encode | bx sha256 | bx ripemd160  
3e0547268b3b19288b3adef9719ec8659f4b2b0b

Next, the redeem script hash is converted to a Bitcoin address. Let’s use bx on the command-line again:

**P2SH address**

echo \  
'3e0547268b3b19288b3adef9719ec8659f4b2b0b' \  
| bx address-encode -v 5  
37Lx99uaGn5avKBxiW26HjedQE3LrDCZru

Now, Bob can display this address for customers to pay for their coffee. Alice’s wallet can make a payment to 37Lx99uaGn5avKBxiW26HjedQE3LrDCZru, just as it would to any other Bitcoin address.

To pay Bob, Alice’s wallet would lock the output with a P2SH script:

HASH160 3e0547268b3b19288b3adef9719ec8659f4b2b0b EQUAL

Even though Alice’s wallet has no support for segwit, the payment it creates can be spent by Bob with a segwit transaction.

===== Pay-to-Witness-Script-Hash inside Pay-to-Script-Hash

Similarly, a P2WSH witness program for a multisig script or other complicated script can be embedded inside a P2SH script and address, making it possible for any wallet to make payments that are segwit compatible.

As we saw in [simpara\_title](#p2wsh), Mohammed’s company is using Segregated Witness payments with multisignature scripts. To make it possible for any client to pay his company, regardless of whether their wallets are upgraded for segwit, Mohammed’s wallet can embed the P2WSH witness program inside a P2SH script.

First, Mohammed’s wallet hashes the redeem script with SHA256 (just once). Let’s use bx to do that on the command-line:

**Mohammed’s wallet creates a P2WSH witness program**

echo \  
2 \ [04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984D83F1F50C900A24DD47F569FD4193AF5DE762C587] \  
[04A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49] \  
[047E63248B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC10F1E8E8F3020DECDBC3C0DD389D9977965] \  
[0421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5] \  
[043752580AFA1ECED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D137AAB59E0B000EB7ED238F4D800] \  
5 CHECKMULTISIG \  
| bx script-encode | bx sha256  
9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

Next, the hashed redeem script is turned into a P2WSH witness program:

0 9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73

Then, the witness program itself is hashed with SHA256 and RIPEMD160, producing a new 20-byte hash, as used in traditional P2SH. Let’s use bx on the command-line to do that:

**The HASH160 of the P2WSH witness program**

echo \  
'0 [9592d601848d04b172905e0ddb0adde59f1590f1e553ffc81ddc4b0ed927dd73]'\  
 | bx script-encode | bx sha256 | bx ripemd160  
86762607e8fe87c0c37740cddee880988b9455b2

Next, the wallet constructs a P2SH Bitcoin address from this hash. Again, we use bx to calculate on the command-line:

**P2SH Bitcoin address**

echo \  
'86762607e8fe87c0c37740cddee880988b9455b2'\  
 | bx address-encode -v 5  
3Dwz1MXhM6EfFoJChHCxh1jWHb8GQqRenG

Now, Mohammed’s clients can make payments to this address without any need to support segwit. To send a payment to Mohammed, a wallet would lock the output with the following P2SH script:

**P2SH script used to lock payments to Mohammed’s multisig**

HASH160 86762607e8fe87c0c37740cddee880988b9455b2 EQUAL

Mohammed’s company can then construct segwit transactions to spend these payments, taking advantage of segwit features including lower transaction fees.

===== Segregated Witness addresses

Even after segwit activation, it will take some time until most wallets are upgraded. At first, segwit will be embedded in P2SH, as we saw in the previous section, to ease compatibility between segwit-aware and unaware wallets.

However, once wallets are broadly supporting segwit, it makes sense to encode witness scripts directly in a native address format designed for segwit, rather than embed it in P2SH.

The native segwit address format is defined in BIP-173:

[BIP-173](https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki)

Base32 address format for native v0-16 witness outputs

BIP-173 only encodes witness (P2WPKH and P2WSH) scripts. It is not compatible with non-segwit P2PKH or P2SH scripts. BIP-173 is a checksummed Base32 encoding, as compared to the Base58 encoding of a "traditional" Bitcoin address. BIP-173 addresses are also called *bech32* addresses, pronounced "beh-ch thirty two", alluding to the use of a "BCH" error detection algorithm and 32-character encoding set.

BIP-173 addresses use 32 lower-case-only alphanumeric character set, carefully selected to reduce errors from misreading or mistyping. By choosing a lower-case-only character set, bech32 is easier to read, speak, and 45% more efficient to encode in QR codes.

The BCH error detection algorithm is a vast improvement over the previous checksum algorithm (from Base58Check), allowing not only detection but also *correction* of errors. Address-input interfaces (such as text-fields in forms) can detect and highlight which character was most likely mistyped when they detect an error.

From the BIP-173 specification, here are some examples of bech32 addresses:

Mainnet P2WPKH

bc1qw508d6qejxtdg4y5r3zarvary0c5xw7kv8f3t4

Testnet P2WPKH

tb1qw508d6qejxtdg4y5r3zarvary0c5xw7kxpjzsx

Mainnet P2WSH

bc1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3qccfmv3

Testnet P2WSH

tb1qrp33g0q5c5txsp9arysrx4k6zdkfs4nce4xj0gdcccefvpysxf3q0sl5k7

As you can see in these examples, a segwit bech32 string is up to 90 characters long and consists of three parts:

The human readable part

This prefix "bc" or "tb" identifying mainnet or testnet

The separator

The digit "1", which is not part of the 32-character encoding set and can only appear in this position as a separator

The data part

A minimum of 6 alphanumeric characters, the checksum encoded witness script

At this time, only a few wallets accept or produce native segwit bech32 addresses, but as segwit adoption increases, you will see these more and more often.

[table\_title](#segwit_addresses) shows bitcoin non-segwit (legacy) and segwit addresses.

Bitcoin non-segwit (legacy) and segwit addresses

|  |  |  |
| --- | --- | --- |
| Type | Encoding | Prefix |
| Legacy P2PKH Address | Base58 | 1 |
| Legacy Testnet P2PKH Address | Base58 | m or n |
| Legacy P2SH Address | Base58 | 3 |
| Legacy Testnet P2SH Address | Base58 | 2 |
| Nested (embedded) Segwit P2SH(P2WPKH) Address | Base58 | 3 |
| Nested (embedded) Segwit P2SH(P2WSH) Address | Base58 | 3 |
| Native Segwit P2WPKH Address | Bech32 | bc1 |
| Native Segwit Testnet P2WPKH Address | Bech32 | tb1 |
| Native Segwit P2WSH Address | Bech32 | bc1 |
| Native Segwit Testnet P2WSH Address | Bech32 | tb1 |

===== Transaction identifiers

One of the greatest benefits of Segregated Witness is that it eliminates third-party transaction malleability.

Before segwit, transactions could have their signatures subtly modified by third parties, changing their transaction ID (hash) without changing any fundamental properties (inputs, outputs, amounts). This created opportunities for denial-of-service attacks as well as attacks against poorly written wallet software that assumed unconfirmed transaction hashes were immutable.

With the introduction of Segregated Witness, transactions have two identifiers, txid and wtxid. The traditional transaction ID txid is the double-SHA256 hash of the serialized transaction, without the witness data. A transaction wtxid is the double-SHA256 hash of the new serialization format of the transaction with witness data.

The traditional txid is calculated in exactly the same way as with a nonsegwit transaction. However, since a pure segwit transaction (a transaction that only contains segwit inputs) has empty scriptSigs in every input, there is no part of the transaction that can be modified by a third party. Therefore, in a pure segwit transaction, the txid is immutable by a third party, even when the transaction is unconfirmed.

The wtxid is like an "extended" ID, in that the hash also incorporates the witness data. If a transaction is transmitted without witness data, then the wtxid and txid are identical. Note that since the wtxid includes witness data (signatures) and since witness data may be malleable, the wtxid should be considered malleable until the transaction is confirmed. Only the txid of a pure segwit transaction can be considered immutable by third parties.

Segregated Witness transactions have two IDs: txid and wtxid. The txid is the hash of the transaction without the witness data and the wtxid is the hash inclusive of witness data. Only pure segwit transactions (transactions that only contain segwit inputs) have a txid that is not susceptible to third-party transaction malleability.

==== Segregated Witness' New Signing Algorithm

Segregated Witness modifies the semantics of the four signature verification functions (CHECKSIG, CHECKSIGVERIFY, CHECKMULTISIG, and CHECKMULTISIGVERIFY), changing the way a transaction commitment hash is calculated.

Signatures in bitcoin transactions are applied on a *commitment hash*, which is calculated from the transaction data, locking specific parts of the data indicating the signer’s commitment to those values. For example, in a simple SIGHASH\_ALL type signature, the commitment hash includes all inputs and outputs.

Unfortunately, the way the commitment hash was calculated introduced the possibility that a node verifying the signature can be forced to perform a significant number of hash computations. Specifically, the hash operations increase in O(n2) with respect to the number of signature operations in the transaction. An attacker could therefore create a transaction with a very large number of signature operations, causing the entire Bitcoin network to have to perform hundreds or thousands of hash operations to verify the transaction.

Segwit represented an opportunity to address this problem by changing the way the commitment hash is calculated. For segwit version 0 witness programs, signature verification occurs using an improved commitment hash algorithm as specified in BIP-143.

The new algorithm achieves two important goals. Firstly, the number of hash operations increases by a much more gradual O(n) to the number of signature operations, reducing the opportunity to create denial-of-service attacks with overly complex transactions. Secondly, the commitment hash now also includes the value (amounts) of each input as part of the commitment. This means that a signer can commit to a specific input value without needing to "fetch" and check the previous transaction referenced by the input. In the case of offline devices, such as hardware wallets, this greatly simplifies the communication between the host and the hardware wallet, removing the need to stream previous transactions for validation. A hardware wallet can accept the input value "as stated" by an untrusted host. Since the signature is invalid if that input value is not correct, the hardware wallet doesn’t need to validate the value before signing the input.

==== Economic Incentives for Segregated Witness

Bitcoin mining nodes and full nodes incur costs for the resources used to support the Bitcoin network and the blockchain. As the volume of bitcoin transactions increases, so does the cost of resources (CPU, network bandwidth, disk space, memory). Miners are compensated for these costs through fees that are proportional to the size (in bytes) of each transaction. Nonmining full nodes are not compensated, so they incur these costs because they have a need to run an authoritative fully validating full-index node, perhaps because they use the node to operate a bitcoin business.

Without transaction fees, the growth in bitcoin data would arguably increase dramatically. Fees are intended to align the needs of bitcoin users with the burden their transactions impose on the network, through a market-based price discovery mechanism.

The calculation of fees based on transaction size treats all the data in the transaction as equal in cost. But from the perspective of full nodes and miners, some parts of a transaction carry much higher costs. Every transaction added to the Bitcoin network affects the consumption of four resources on nodes:

Disk Space

Every transaction is stored in the blockchain, adding to the total size of the blockchain. The blockchain is stored on disk, but the storage can be optimized by “pruning” (deleting) older transactions.

CPU

Every transaction must be validated, which requires CPU time.

Bandwidth

Every transaction is transmitted (through flood propagation) across the network at least once. Without any optimization in the block propagation protocol, transactions are transmitted again as part of a block, doubling the impact on network capacity.

Memory

Nodes that validate transactions keep the UTXO index or the entire UTXO set in memory to speed up validation. Because memory is at least one order of magnitude more expensive than disk, growth of the UTXO set contributes disproportionately to the cost of running a node.

As you can see from the list, not every part of a transaction has an equal impact on the cost of running a node or on the ability of bitcoin to scale to support more transactions. The most expensive part of a transaction are the newly created outputs, as they are added to the in-memory UTXO set. By comparison, signatures (aka witness data) add the least burden to the network and the cost of running a node, because witness data are only validated once and then never used again. Furthermore, immediately after receiving a new transaction and validating witness data, nodes can discard that witness data. If fees are calculated on transaction size, without discriminating between these two types of data, then the market incentives of fees are not aligned with the actual costs imposed by a transaction. In fact, the current fee structure actually encourages the opposite behavior, because witness data is the largest part of a transaction.

The incentives created by fees matter because they affect the behavior of wallets. All wallets must implement some strategy for assembling transactions that takes into consideration a number of factors, such as privacy (reducing address reuse), fragmentation (making lots of loose change), and fees. If the fees are overwhelmingly motivating wallets to use as few inputs as possible in transactions, this can lead to UTXO picking and change address strategies that inadvertently bloat the UTXO set.

Transactions consume UTXO in their inputs and create new UTXO with their outputs. A transaction, therefore, that has more inputs than outputs will result in a decrease in the UTXO set, whereas a transaction that has more outputs than inputs will result in an increase in the UTXO set. Let’s consider the *difference* between inputs and outputs and call that the “Net-new-UTXO.” That’s an important metric, as it tells us what impact a transaction will have on the most expensive network-wide resource, the in-memory UTXO set. A transaction with positive Net-new-UTXO adds to that burden. A transaction with a negative Net-new-UTXO reduces the burden. We would therefore want to encourage transactions that are either negative Net-new-UTXO or neutral with zero Net-new-UTXO.

Let’s look at an example of what incentives are created by the transaction fee calculation, with and without Segregated Witness. We will look at two different transactions. Transaction A is a 3-input, 2-output transaction, which has a Net-new-UTXO metric of –1, meaning it consumes one more UTXO than it creates, reducing the UTXO set by one. Transaction B is a 2-input, 3-output transaction, which has a Net-new-UTXO metric of 1, meaning it adds one UTXO to the UTXO set, imposing additional cost on the entire Bitcoin network. Both transactions use multisignature (2-of-3) scripts to demonstrate how complex scripts increase the impact of segregated witness on fees. Let’s assume a transaction feerate of 30 satoshi per byte and a 75% fee discount on witness data:

Without Segregated Witness

Transaction A fee: 28,590 satoshi

Transaction B fee: 20,760 satoshi

With Segregated Witness

Transaction A fee: 12,255 satoshi

Transaction B fee: 10,425 satoshi

Both transactions are less expensive when segregated witness is implemented. Comparing the costs between the two transactions, we see that before Segregated Witness, the transaction with the positive Net-new-UTXO has significant cost-savings. With Segregated Witness, the cost difference shrinks significantly in absolute as well as relative terms. While it would require inputs to become cheaper than outputs to incentivize UTXO set consolidation, this discount reduces the incentive to create new UTXO in order to avoid using more inputs.

Segregated Witness therefore has two main effects on the fees paid by bitcoin users. Firstly, segwit reduces the overall cost of transactions by discounting witness data and increasing the capacity of the Bitcoin blockchain. Secondly, segwit’s discount on witness data partially mitigates a misalignment of incentives that may have inadvertently created more bloat in the UTXO set.

== The Bitcoin Network

=== Peer-to-Peer Network Architecture

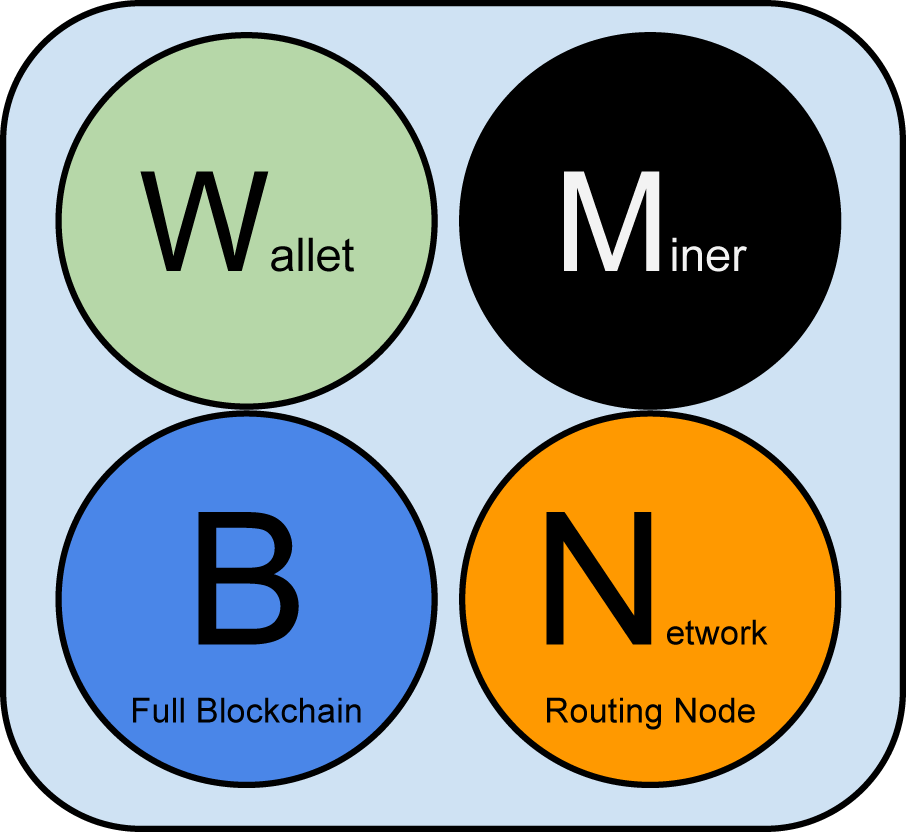
Bitcoin is structured as a peer-to-peer network architecture on top of the internet. The term peer-to-peer, or P2P, means that the computers that participate in the network are peers to each other, that they are all equal, that there are no "special" nodes, and that all nodes share the burden of providing network services. The network nodes interconnect in a mesh network with a "flat" topology. There is no server, no centralized service, and no hierarchy within the network. Nodes in a P2P network both provide and consume services at the same time with reciprocity acting as the incentive for participation. P2P networks are inherently resilient, decentralized, and open. A preeminent example of a P2P network architecture was the early internet itself, where nodes on the IP network were equal. Today’s internet architecture is more hierarchical, but the Internet Protocol still retains its flat-topology essence. Beyond bitcoin, the largest and most successful application of P2P technologies is file sharing, with Napster as the pioneer and BitTorrent as the most recent evolution of the architecture.

Bitcoin’s P2P network architecture is much more than a topology choice. Bitcoin is a P2P digital cash system by design, and the network architecture is both a reflection and a foundation of that core characteristic. Decentralization of control is a core design principle that can only be achieved and maintained by a flat, decentralized P2P consensus network.

The term "Bitcoin network" refers to the collection of nodes running the bitcoin P2P protocol. In addition to the bitcoin P2P protocol, there are other protocols such as Stratum that are used for mining and lightweight or mobile wallets. These additional protocols are provided by gateway routing servers that access the Bitcoin network using the bitcoin P2P protocol and then extend that network to nodes running other protocols. For example, Stratum servers connect Stratum mining nodes via the Stratum protocol to the main Bitcoin network and bridge the Stratum protocol to the bitcoin P2P protocol. We use the term "extended Bitcoin network" to refer to the overall network that includes the bitcoin P2P protocol, pool-mining protocols, the Stratum protocol, and any other related protocols connecting the components of the Bitcoin system.

=== Node Types and Roles

Although nodes in the bitcoin P2P network are equal, they may take on different roles depending on the functionality they are supporting. A Bitcoin node is a collection of functions: routing, the blockchain database, mining, and wallet services. A full node with all four of these functions is shown in [figure\_title](#full_node_reference).



A Bitcoin network node with all four functions: wallet, miner, full blockchain database, and network routing

All nodes include the routing function to participate in the network and might include other functionality. All nodes validate and propagate transactions and blocks, and discover and maintain connections to peers. In the full-node example in [figure\_title](#full_node_reference), the routing function is indicated by a circle named "Network Routing Node" or with the letter "N."

Some nodes, called full nodes, also maintain a complete and up-to-date copy of the blockchain. Full nodes can autonomously and authoritatively verify any transaction without external reference. Some nodes maintain only a subset of the blockchain and verify transactions using a method called *simplified payment verification*, or SPV. These nodes are known as SPV nodes or lightweight nodes. In the full-node example in the figure, the full-node blockchain database function is indicated by a circle called "Full Blockchain" or the letter "B." In [figure\_title](#bitcoin_network), SPV nodes are drawn without the "B" circle, showing that they do not have a full copy of the blockchain.

Mining nodes compete to create new blocks by running specialized hardware to solve the Proof-of-Work algorithm. Some mining nodes are also full nodes, maintaining a full copy of the blockchain, while others are lightweight nodes participating in pool mining and depending on a pool server to maintain a full node. The mining function is shown in the full node as a circle called "Miner" or the letter "M."

User wallets might be part of a full node, as is usually the case with desktop Bitcoin clients. Increasingly, many user wallets, especially those running on resource-constrained devices such as smartphones, are SPV nodes. The wallet function is shown in [figure\_title](#full_node_reference) as a circle called "Wallet" or the letter "W."

In addition to the main node types on the bitcoin P2P protocol, there are servers and nodes running other protocols, such as specialized mining pool protocols and lightweight client-access protocols.

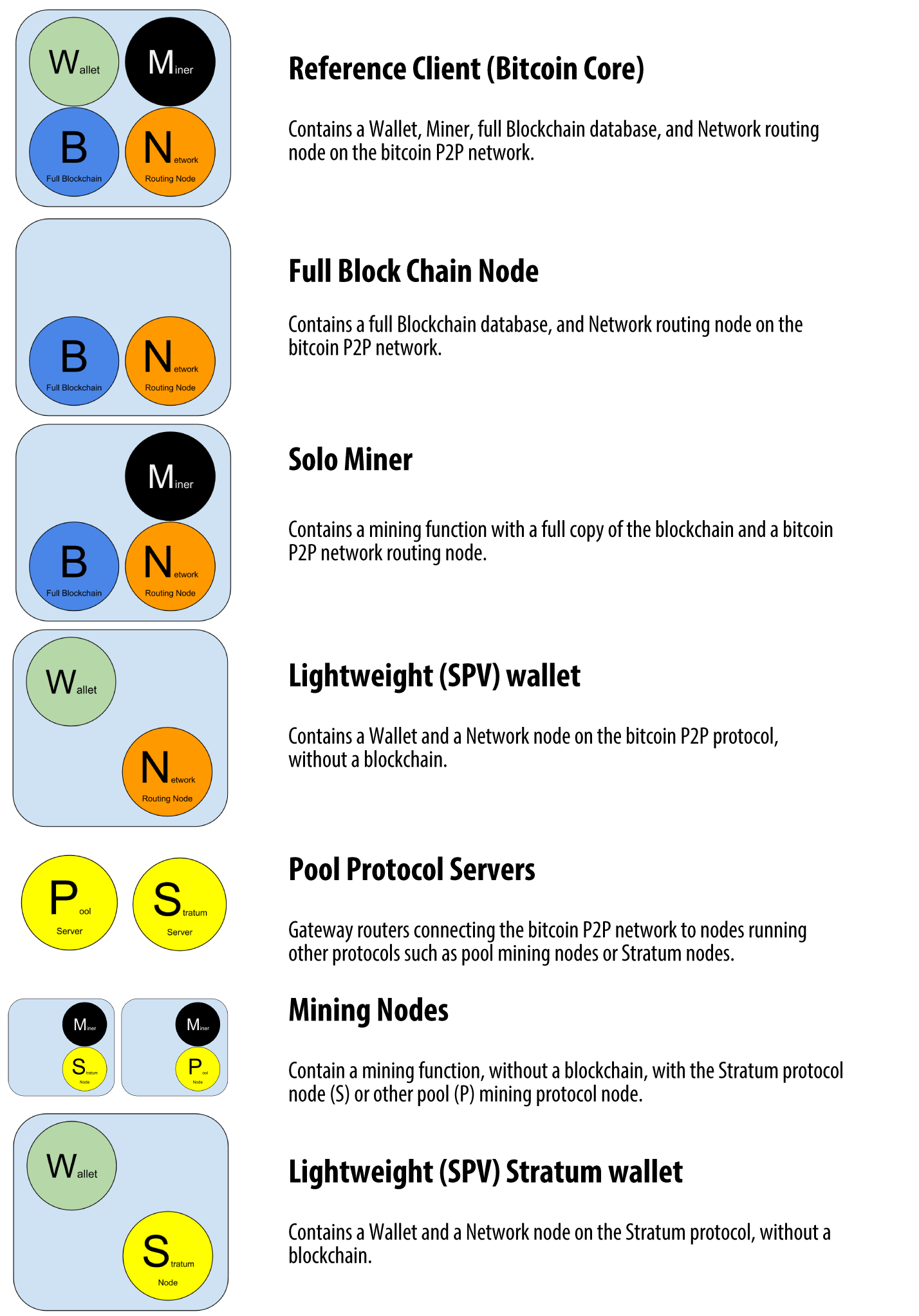
[figure\_title](#node_type_ledgend) shows the most common node types on the extended Bitcoin network.

=== The Extended Bitcoin Network

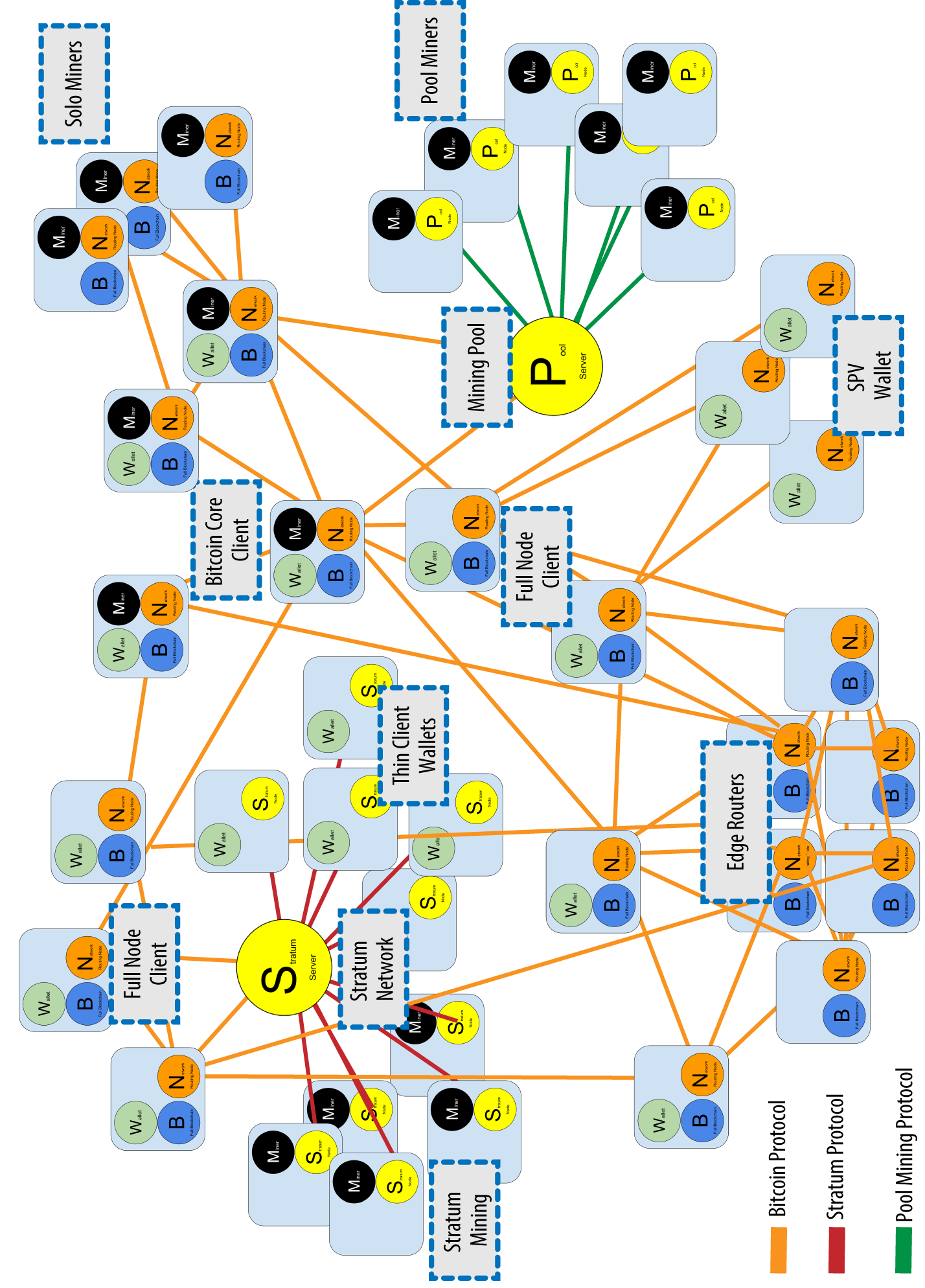
The main Bitcoin network, running the bitcoin P2P protocol, consists of between 5,000 and 8,000 listening nodes running various versions of the bitcoin reference client (Bitcoin Core) and a few hundred nodes running various other implementations of the bitcoin P2P protocol, such as Bitcoin Classic, Bitcoin Unlimited, BitcoinJ, Libbitcoin, btcd, and bcoin. A small percentage of the nodes on the bitcoin P2P network are also mining nodes, competing in the mining process, validating transactions, and creating new blocks. Various large companies interface with the Bitcoin network by running full-node clients based on the Bitcoin Core client, with full copies of the blockchain and a network node, but without mining or wallet functions. These nodes act as network edge routers, allowing various other services (exchanges, wallets, block explorers, merchant payment processing) to be built on top.

The extended Bitcoin network includes the network running the bitcoin P2P protocol, described earlier, as well as nodes running specialized protocols. Attached to the main bitcoin P2P network are a number of pool servers and protocol gateways that connect nodes running other protocols. These other protocol nodes are mostly pool mining nodes (see [simpara\_title](#mining)) and lightweight wallet clients, which do not carry a full copy of the blockchain.

[figure\_title](#bitcoin_network) shows the extended Bitcoin network with the various types of nodes, gateway servers, edge routers, and wallet clients and the various protocols they use to connect to each other.



Different types of nodes on the extended Bitcoin network



The extended Bitcoin network showing various node types, gateways, and protocols

=== Bitcoin Relay Networks

While the bitcoin P2P network serves the general needs of a broad variety of node types, it exhibits too high network latency for the specialized needs of bitcoin mining nodes.

Bitcoin miners are engaged in a time-sensitive competition to solve the Proof-of-Work problem and extend the blockchain (see [simpara\_title](#mining)). While participating in this competition, bitcoin miners must minimize the time between the propagation of a winning block and the beginning of the next round of competition. In mining, network latency is directly related to profit margins.

A *Bitcoin Relay Network* is a network that attempts to minimize the latency in the transmission of blocks between miners. The original [Bitcoin Relay Network](https://www.bitcoinrelaynetwork.org) was created by core developer Matt Corallo in 2015 to enable fast synchronization of blocks between miners with very low latency. The network consisted of several specialized nodes hosted on the Amazon Web Services infrastructure around the world and served to connect the majority of miners and mining pools.

The original Bitcoin Relay Network was replaced in 2016 with the introduction of the *Fast Internet Bitcoin Relay Engine* or [*FIBRE*](https://bitcoinfibre.org), also created by core developer Matt Corallo. FIBRE is a UDP-based relay network that relays blocks within a network of nodes. FIBRE implements *compact block* optimization to further reduce the amount of data transmitted and the network latency.

Relay networks are not replacements for bitcoin’s P2P network. Instead they are overlay networks that provide additional connectivity between nodes with specialized needs. Like freeways are not replacements for rural roads, but rather shortcuts between two points with heavy traffic, you still need small roads to connect to the freeways.

=== Network Discovery

When a new node boots up, it must discover other Bitcoin nodes on the network in order to participate. To start this process, a new node must discover at least one existing node on the network and connect to it. The geographic location of other nodes is irrelevant; the Bitcoin network topology is not geographically defined. Therefore, any existing Bitcoin nodes can be selected at random.

To connect to a known peer, nodes establish a TCP connection, usually to port 8333 (the port generally known as the one used by bitcoin), or an alternative port if one is provided. Upon establishing a connection, the node will start a "handshake" (see [figure\_title](#network_handshake)) by transmitting a version message, which contains basic identifying information, including:

nVersion

The bitcoin P2P protocol version the client "speaks" (e.g., 70002)

nLocalServices

A list of local services supported by the node, currently just NODE\_NETWORK

nTime

The current time

addrYou

The IP address of the remote node as seen from this node

addrMe

The IP address of the local node, as discovered by the local node

subver

A sub-version showing the type of software running on this node (e.g., /Satoshi:0.9.2.1/)

BestHeight

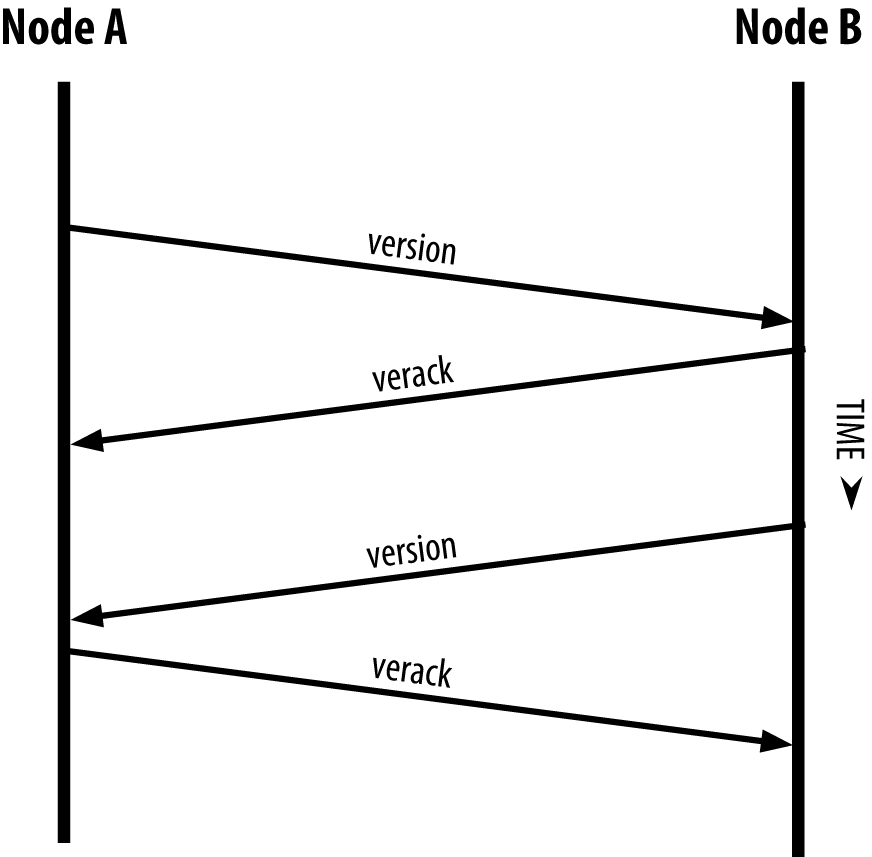
The block height of this node’s blockchain

(See [GitHub](https://bit.ly/1qlsC7w) for an example of the version network message.)

The version message is always the first message sent by any peer to another peer. The local peer receiving a version message will examine the remote peer’s reported nVersion and decide if the remote peer is compatible. If the remote peer is compatible, the local peer will acknowledge the version message and establish a connection by sending a verack message.

How does a new node find peers? The first method is to query DNS using a number of "DNS seeds," which are DNS servers that provide a list of IP addresses of Bitcoin nodes. Some of those DNS seeds provide a static list of IP addresses of stable bitcoin listening nodes. Some of the DNS seeds are custom implementations of BIND (Berkeley Internet Name Daemon) that return a random subset from a list of Bitcoin node addresses collected by a crawler or a long-running Bitcoin node. The Bitcoin Core client contains the names of nine different DNS seeds. The diversity of ownership and diversity of implementation of the different DNS seeds offers a high level of reliability for the initial bootstrapping process. In the Bitcoin Core client, the option to use the DNS seeds is controlled by the option switch -dnsseed (set to 1 by default, to use the DNS seed).

Alternatively, a bootstrapping node that knows nothing of the network must be given the IP address of at least one Bitcoin node, after which it can establish connections through further introductions. The command-line argument -seednode can be used to connect to one node just for introductions using it as a seed. After the initial seed node is used to form introductions, the client will disconnect from it and use the newly discovered peers.



The initial handshake between peers

Once one or more connections are established, the new node will send an addr message containing its own IP address to its neighbors. The neighbors will, in turn, forward the addr message to their neighbors, ensuring that the newly connected node becomes well known and better connected. Additionally, the newly connected node can send getaddr to the neighbors, asking them to return a list of IP addresses of other peers. That way, a node can find peers to connect to and advertise its existence on the network for other nodes to find it. [figure\_title](#address_propagation) shows the address discovery protocol.



Address propagation and discovery

A node must connect to a few different peers in order to establish diverse paths into the Bitcoin network. Paths are not persistent—nodes come and go—and so the node must continue to discover new nodes as it loses old connections as well as assist other nodes when they bootstrap. Only one connection is needed to bootstrap, because the first node can offer introductions to its peer nodes and those peers can offer further introductions. It’s also unnecessary and wasteful of network resources to connect to more than a handful of nodes. After bootstrapping, a node will remember its most recent successful peer connections, so that if it is rebooted it can quickly reestablish connections with its former peer network. If none of the former peers respond to its connection request, the node can use the seed nodes to bootstrap again.

On a node running the Bitcoin Core client, you can list the peer connections with the command 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  
 }  
]

To override the automatic management of peers and to specify a list of IP addresses, users can provide the option -connect=<IPAddress> and specify one or more IP addresses. If this option is used, the node will only connect to the selected IP addresses, instead of discovering and maintaining the peer connections automatically.

If there is no traffic on a connection, nodes will periodically send a message to maintain the connection. If a node has not communicated on a connection for more than 90 minutes, it is assumed to be disconnected and a new peer will be sought. Thus, the network dynamically adjusts to transient nodes and network problems, and can organically grow and shrink as needed without any central control.

=== Full Nodes

Full nodes are nodes that maintain a full blockchain with all transactions. More accurately, they probably should be called "full blockchain nodes." In the early years of bitcoin, all nodes were full nodes and currently the Bitcoin Core client is a full blockchain node. In the past two years, however, new forms of Bitcoin clients have been introduced that do not maintain a full blockchain but run as lightweight clients. We’ll examine these in more detail in the next section.

Full blockchain nodes maintain a complete and up-to-date copy of the Bitcoin blockchain with all the transactions, which they independently build and verify, starting with the very first block (genesis block) and building up to the latest known block in the network. A full blockchain node can independently and authoritatively verify any transaction without recourse or reliance on any other node or source of information. The full blockchain node relies on the network to receive updates about new blocks of transactions, which it then verifies and incorporates into its local copy of the blockchain.

Running a full blockchain node gives you the pure bitcoin experience: independent verification of all transactions without the need to rely on, or trust, any other systems. It’s easy to tell if you’re running a full node because it requires more than one hundred gigabytes of persistent storage (disk space) to store the full blockchain. If you need a lot of disk and it takes two to three days to sync to the network, you are running a full node. That is the price of complete independence and freedom from central authority.

There are a few alternative implementations of full blockchain Bitcoin clients, built using different programming languages and software architectures. However, the most common implementation is the reference client Bitcoin Core, also known as the Satoshi client. More than 75% of the nodes on the Bitcoin network run various versions of Bitcoin Core. It is identified as "Satoshi" in the sub-version string sent in the version message and shown by the command getpeerinfo as we saw earlier; for example, /Satoshi:0.8.6/.

=== Exchanging "Inventory"

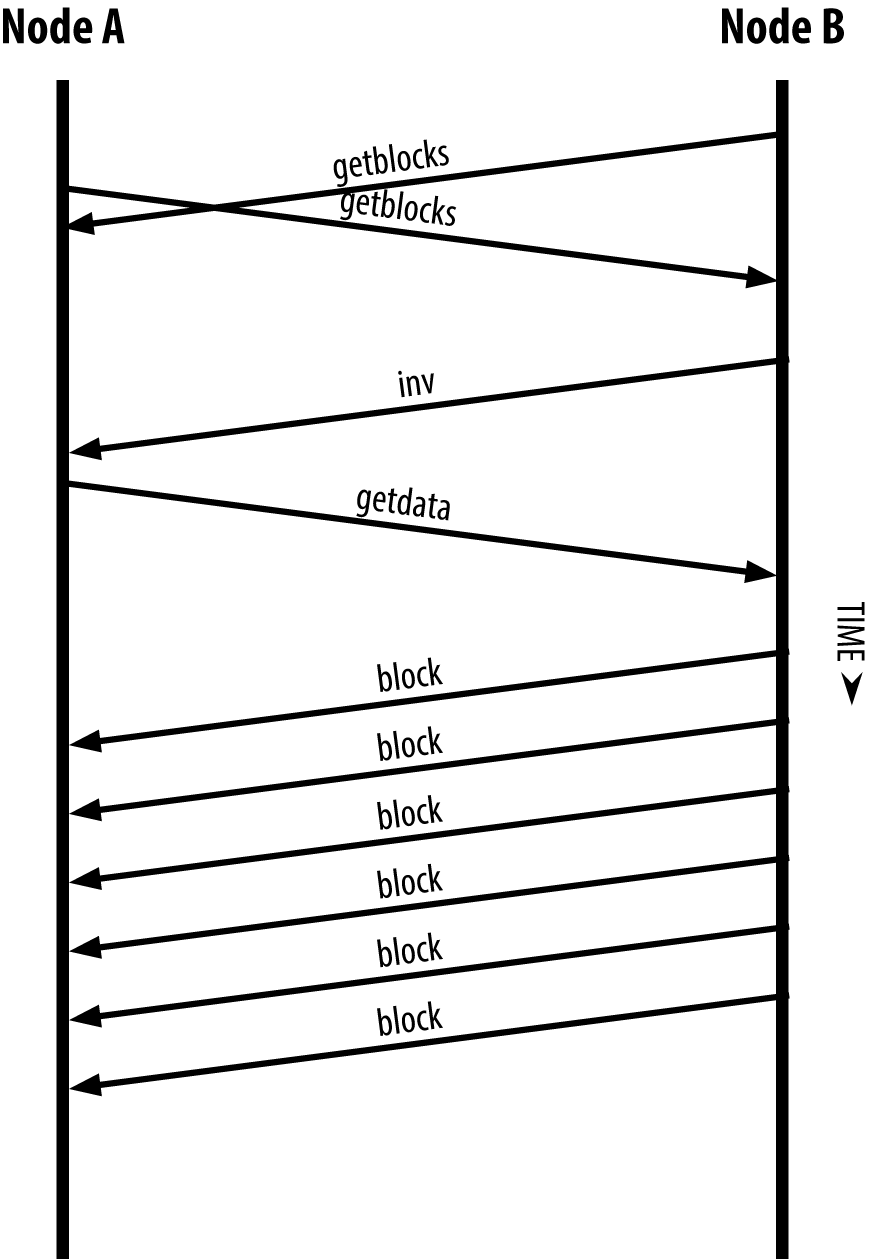
The first thing a full node will do once it connects to peers is try to construct a complete blockchain. If it is a brand-new node and has no blockchain at all, it only knows one block, the genesis block, which is statically embedded in the client software. Starting with block #0 (the genesis block), the new node will have to download hundreds of thousands of blocks to synchronize with the network and reestablish the full blockchain.

The process of syncing the blockchain starts with the version message, because that contains BestHeight, a node’s current blockchain height (number of blocks). A node will see the version messages from its peers, know how many blocks they each have, and be able to compare to how many blocks it has in its own blockchain. Peered nodes will exchange a getblocks message that contains the hash (fingerprint) of the top block on their local blockchain. One of the peers will be able to identify the received hash as belonging to a block that is not at the top, but rather belongs to an older block, thus deducing that its own local blockchain is longer than its peer’s.

The peer that has the longer blockchain has more blocks than the other node and can identify which blocks the other node needs in order to "catch up." It will identify the first 500 blocks to share and transmit their hashes using an inv (inventory) message. The node missing these blocks will then retrieve them, by issuing a series of getdata messages requesting the full block data and identifying the requested blocks using the hashes from the inv message.

Let’s assume, for example, that a node only has the genesis block. It will then receive an inv message from its peers containing the hashes of the next 500 blocks in the chain. It will start requesting blocks from all of its connected peers, spreading the load and ensuring that it doesn’t overwhelm any peer with requests. The node keeps track of how many blocks are "in transit" per peer connection, meaning blocks that it has requested but not received, checking that it does not exceed a limit (MAX\_BLOCKS\_IN\_TRANSIT\_PER\_PEER). This way, if it needs a lot of blocks, it will only request new ones as previous requests are fulfilled, allowing the peers to control the pace of updates and not overwhelm the network. As each block is received, it is added to the blockchain, as we will see in [simpara\_title](#blockchain). As the local blockchain is gradually built up, more blocks are requested and received, and the process continues until the node catches up to the rest of the network.

This process of comparing the local blockchain with the peers and retrieving any missing blocks happens any time a node goes offline for any period of time. Whether a node has been offline for a few minutes and is missing a few blocks, or a month and is missing a few thousand blocks, it starts by sending getblocks, gets an inv response, and starts downloading the missing blocks. [figure\_title](#inventory_synchronization) shows the inventory and block propagation protocol.



Node synchronizing the blockchain by retrieving blocks from a peer

=== Simplified Payment Verification (SPV) Nodes

Not all nodes have the ability to store the full blockchain. Many Bitcoin clients are designed to run on space- and power-constrained devices, such as smartphones, tablets, or embedded systems. For such devices, a *simplified payment verification* (SPV) method is used to allow them to operate without storing the full blockchain. These types of clients are called SPV clients or lightweight clients. As bitcoin adoption surges, the SPV node is becoming the most common form of Bitcoin node, especially for bitcoin wallets.

SPV nodes download only the block headers and do not download the transactions included in each block. The resulting chain of blocks, without transactions, is 1,000 times smaller than the full blockchain. SPV nodes cannot construct a full picture of all the UTXOs that are available for spending because they do not know about all the transactions on the network. SPV nodes verify transactions using a slightly different method that relies on peers to provide partial views of relevant parts of the blockchain on demand.

As an analogy, a full node is like a tourist in a strange city, equipped with a detailed map of every street and every address. By comparison, an SPV node is like a tourist in a strange city asking random strangers for turn-by-turn directions while knowing only one main avenue. Although both tourists can verify the existence of a street by visiting it, the tourist without a map doesn’t know what lies down any of the side streets and doesn’t know what other streets exist. Positioned in front of 23 Church Street, the tourist without a map cannot know if there are a dozen other "23 Church Street" addresses in the city and whether this is the right one. The mapless tourist’s best chance is to ask enough people and hope some of them are not trying to mug him.

SPV verifies transactions by reference to their *depth* in the blockchain instead of their *height*. Whereas a full blockchain node will construct a fully verified chain of thousands of blocks and transactions reaching down the blockchain (back in time) all the way to the genesis block, an SPV node will verify the chain of all blocks (but not all transactions) and link that chain to the transaction of interest.

For example, when examining a transaction in block 300,000, a full node links all 300,000 blocks down to the genesis block and builds a full database of UTXO, establishing the validity of the transaction by confirming that the UTXO remains unspent. An SPV node cannot validate whether the UTXO is unspent. Instead, the SPV node will establish a link between the transaction and the block that contains it, using a *merkle path* (see [simpara\_title](#merkle_trees)). Then, the SPV node waits until it sees the six blocks 300,001 through 300,006 piled on top of the block containing the transaction and verifies it by establishing its depth under blocks 300,006 to 300,001. The fact that other nodes on the network accepted block 300,000 and then did the necessary work to produce six more blocks on top of it is proof, by proxy, that the transaction was not a double-spend.

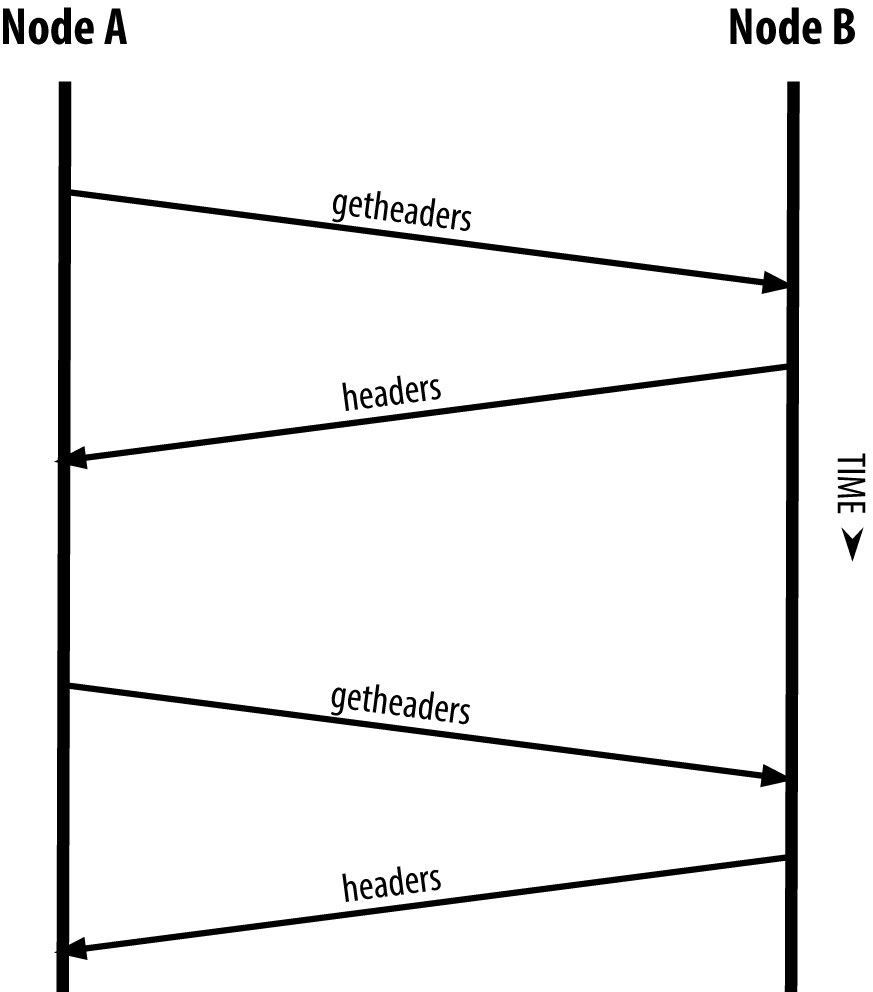
An SPV node cannot be persuaded that a transaction exists in a block when the transaction does not in fact exist. The SPV node establishes the existence of a transaction in a block by requesting a merkle path proof and by validating the Proof-of-Work in the chain of blocks. However, a transaction’s existence can be "hidden" from an SPV node. An SPV node can definitely prove that a transaction exists but cannot verify that a transaction, such as a double-spend of the same UTXO, doesn’t exist because it doesn’t have a record of all transactions. This vulnerability can be used in a denial-of-service attack or for a double-spending attack against SPV nodes. To defend against this, an SPV node needs to connect randomly to several nodes, to increase the probability that it is in contact with at least one honest node. This need to randomly connect means that SPV nodes also are vulnerable to network partitioning attacks or Sybil attacks, where they are connected to fake nodes or fake networks and do not have access to honest nodes or the real Bitcoin network.

For most practical purposes, well-connected SPV nodes are secure enough, striking a balance between resource needs, practicality, and security. For infallible security, however, nothing beats running a full blockchain node.

A full blockchain node verifies a transaction by checking the entire chain of thousands of blocks below it in order to guarantee that the UTXO is not spent, whereas an SPV node checks how deep the block is buried by a handful of blocks above it.

To get the block headers, SPV nodes use a getheaders message instead of getblocks. The responding peer will send up to 2,000 block headers using a single headers message. The process is otherwise the same as that used by a full node to retrieve full blocks. SPV nodes also set a filter on the connection to peers, to filter the stream of future blocks and transactions sent by the peers. Any transactions of interest are retrieved using a getdata request. The peer generates a tx message containing the transactions, in response. [figure\_title](#spv_synchronization) shows the synchronization of block headers.

Because SPV nodes need to retrieve specific transactions in order to selectively verify them, they also create a privacy risk. Unlike full blockchain nodes, which collect all transactions within each block, the SPV node’s requests for specific data can inadvertently reveal the addresses in their wallet. For example, a third party monitoring a network could keep track of all the transactions requested by a wallet on an SPV node and use those to associate Bitcoin addresses with the user of that wallet, destroying the user’s privacy.



SPV node synchronizing the block headers

Shortly after the introduction of SPV/lightweight nodes, bitcoin developers added a feature called *bloom filters* to address the privacy risks of SPV nodes. Bloom filters allow SPV nodes to receive a subset of the transactions without revealing precisely which addresses they are interested in, through a filtering mechanism that uses probabilities rather than fixed patterns.

=== Bloom Filters

A bloom filter is a probabilistic search filter that offers an efficient way to express a search pattern while protecting privacy. They are used by SPV nodes to ask their peers for transactions matching a specific pattern, without revealing exactly which addresses, keys, or transactions they are searching for.

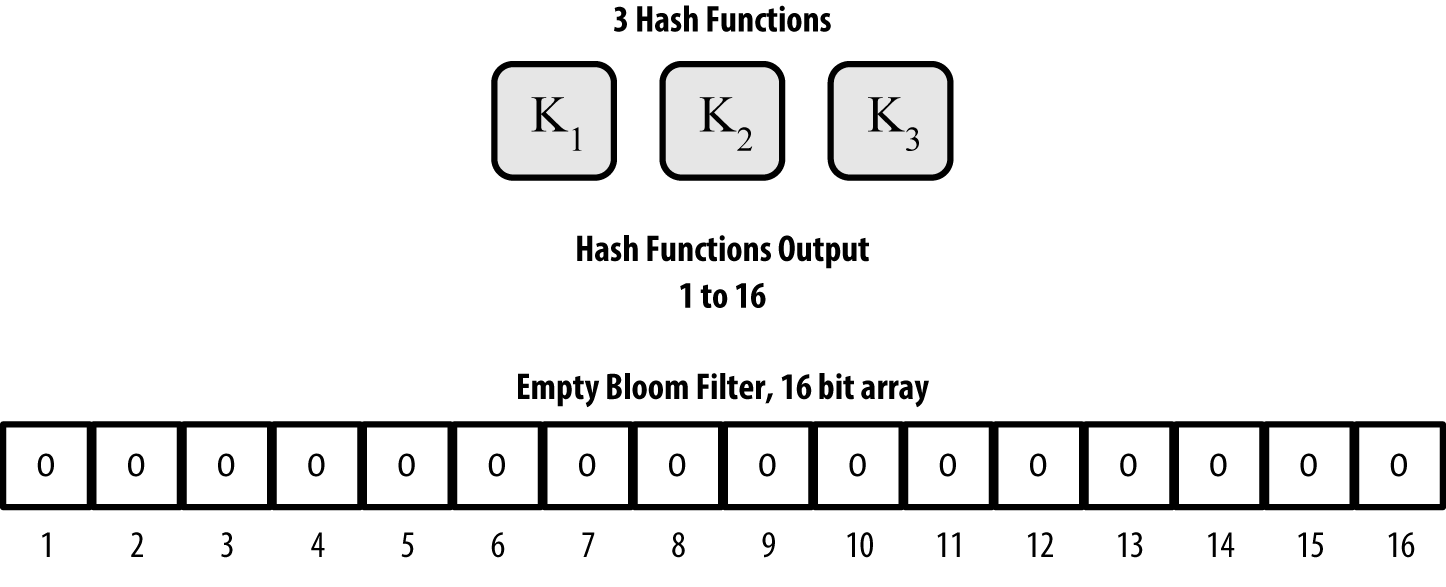
In our previous analogy, a tourist without a map is asking for directions to a specific address, "23 Church St." If she asks strangers for directions to this street, she inadvertently reveals her destination. A bloom filter is like asking, "Are there any streets in this neighborhood whose name ends in R-C-H?" A question like that reveals slightly less about the desired destination than asking for "23 Church St." Using this technique, a tourist could specify the desired address in more detail such as "ending in U-R-C-H" or less detail as "ending in H." By varying the precision of the search, the tourist reveals more or less information, at the expense of getting more or less specific results. If she asks a less specific pattern, she gets a lot more possible addresses and better privacy, but many of the results are irrelevant. If she asks for a very specific pattern, she gets fewer results but loses privacy.

Bloom filters serve this function by allowing an SPV node to specify a search pattern for transactions that can be tuned toward precision or privacy. A more specific bloom filter will produce accurate results, but at the expense of revealing what patterns the SPV node is interested in, thus revealing the addresses owned by the user’s wallet. A less specific bloom filter will produce more data about more transactions, many irrelevant to the node, but will allow the node to maintain better privacy.

==== How Bloom Filters Work

Bloom filters are implemented as a variable-size array of N binary digits (a bit field) and a variable number of M hash functions. The hash functions are designed to always produce an output that is between 1 and N, corresponding to the array of binary digits. The hash functions are generated deterministically, so that any node implementing a bloom filter will always use the same hash functions and get the same results for a specific input. By choosing different length (N) bloom filters and a different number (M) of hash functions, the bloom filter can be tuned, varying the level of accuracy and therefore privacy.

In [figure\_title](#bloom1), we use a very small array of 16 bits and a set of three hash functions to demonstrate how bloom filters work.

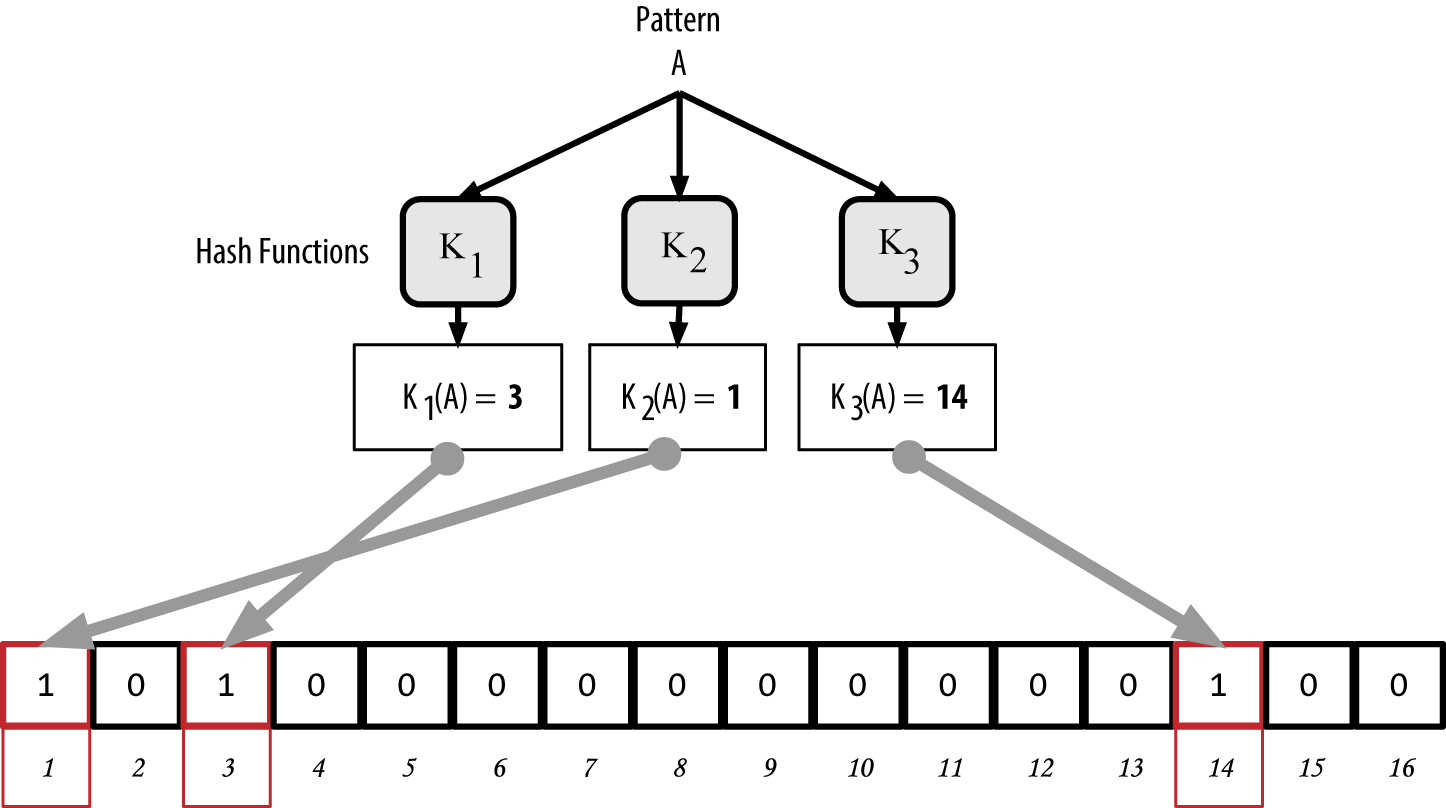


An example of a simplistic bloom filter, with a 16-bit field and three hash functions

The bloom filter is initialized so that the array of bits is all zeros. To add a pattern to the bloom filter, the pattern is hashed by each hash function in turn. Applying the first hash function to the input results in a number between 1 and N. The corresponding bit in the array (indexed from 1 to N) is found and set to 1, thereby recording the output of the hash function. Then, the next hash function is used to set another bit and so on. Once all M hash functions have been applied, the search pattern will be "recorded" in the bloom filter as M bits that have been changed from 0 to 1.

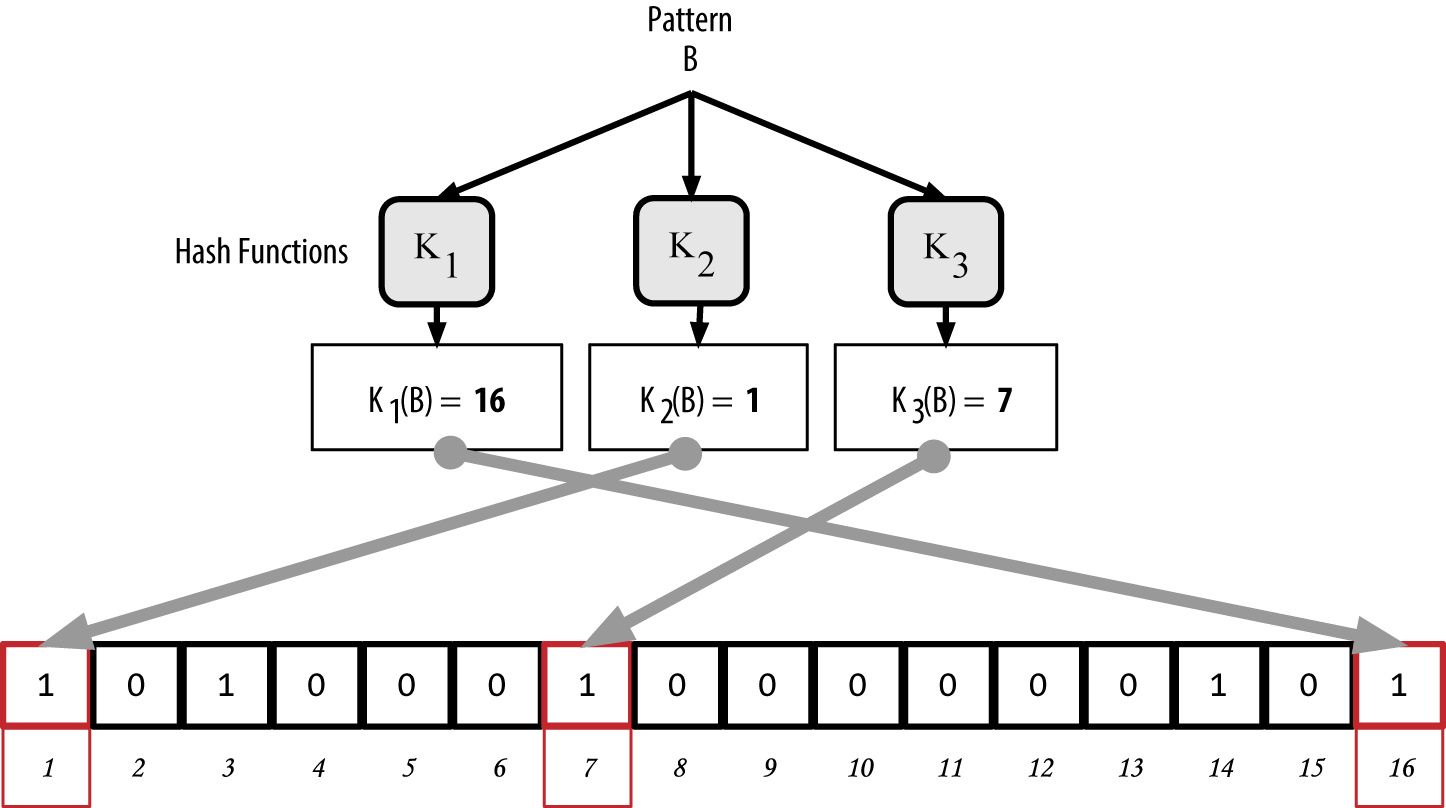
[figure\_title](#bloom2) is an example of adding a pattern "A" to the simple bloom filter shown in [figure\_title](#bloom1).

Adding a second pattern is as simple as repeating this process. The pattern is hashed by each hash function in turn and the result is recorded by setting the bits to 1. Note that as a bloom filter is filled with more patterns, a hash function result might coincide with a bit that is already set to 1, in which case the bit is not changed. In essence, as more patterns record on overlapping bits, the bloom filter starts to become saturated with more bits set to 1 and the accuracy of the filter decreases. This is why the filter is a probabilistic data structure—it gets less accurate as more patterns are added. The accuracy depends on the number of patterns added versus the size of the bit array (N) and number of hash functions (M). A larger bit array and more hash functions can record more patterns with higher accuracy. A smaller bit array or fewer hash functions will record fewer patterns and produce less accuracy.



Adding a pattern "A" to our simple bloom filter

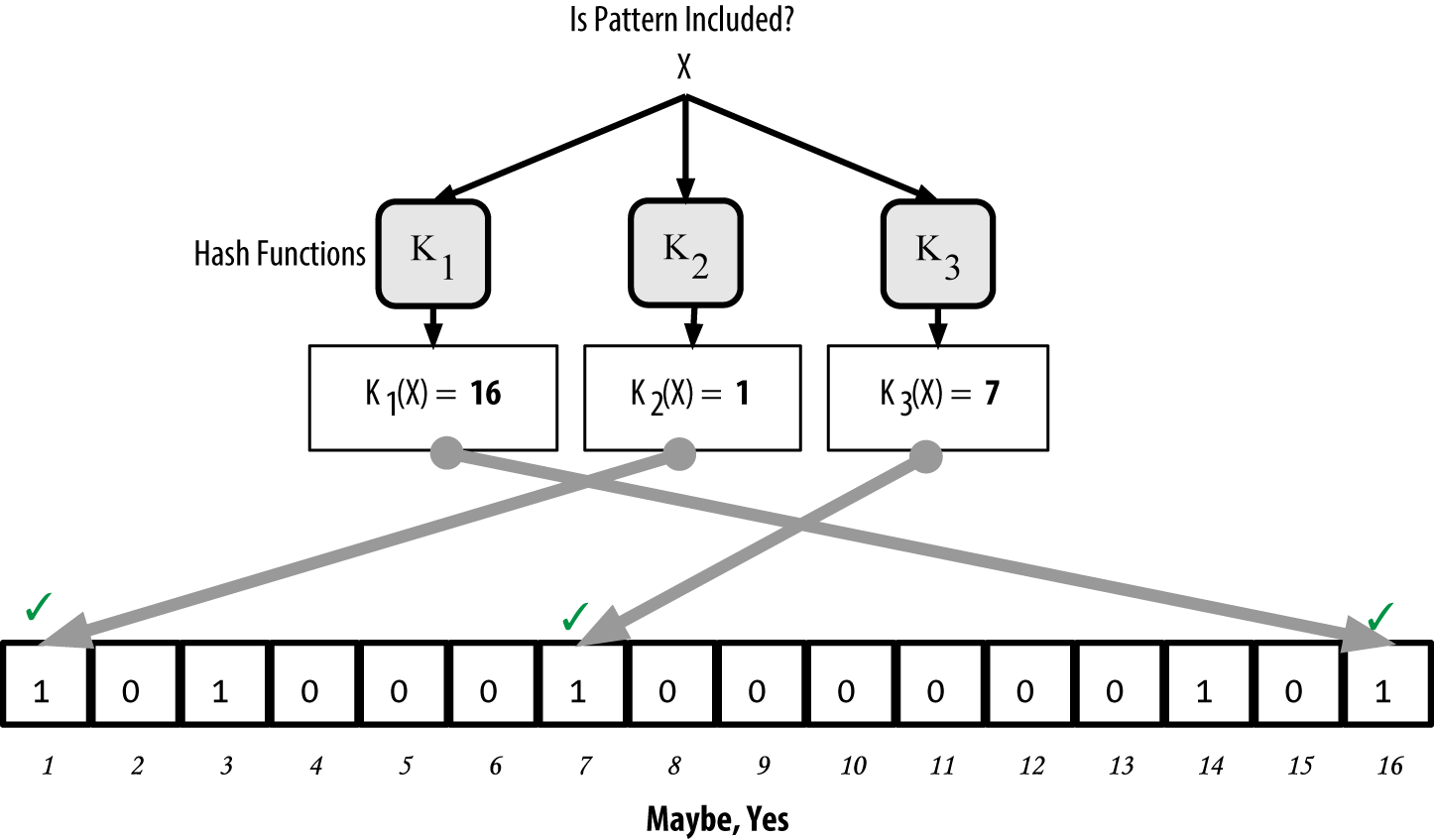
[figure\_title](#bloom3) is an example of adding a second pattern "B" to the simple bloom filter.



Adding a second pattern "B" to our simple bloom filter

To test if a pattern is part of a bloom filter, the pattern is hashed by each hash function and the resulting bit pattern is tested against the bit array. If all the bits indexed by the hash functions are set to 1, then the pattern is *probably* recorded in the bloom filter. Because the bits may be set because of overlap from multiple patterns, the answer is not certain, but is rather probabilistic. In simple terms, a bloom filter positive match is a "Maybe, Yes."

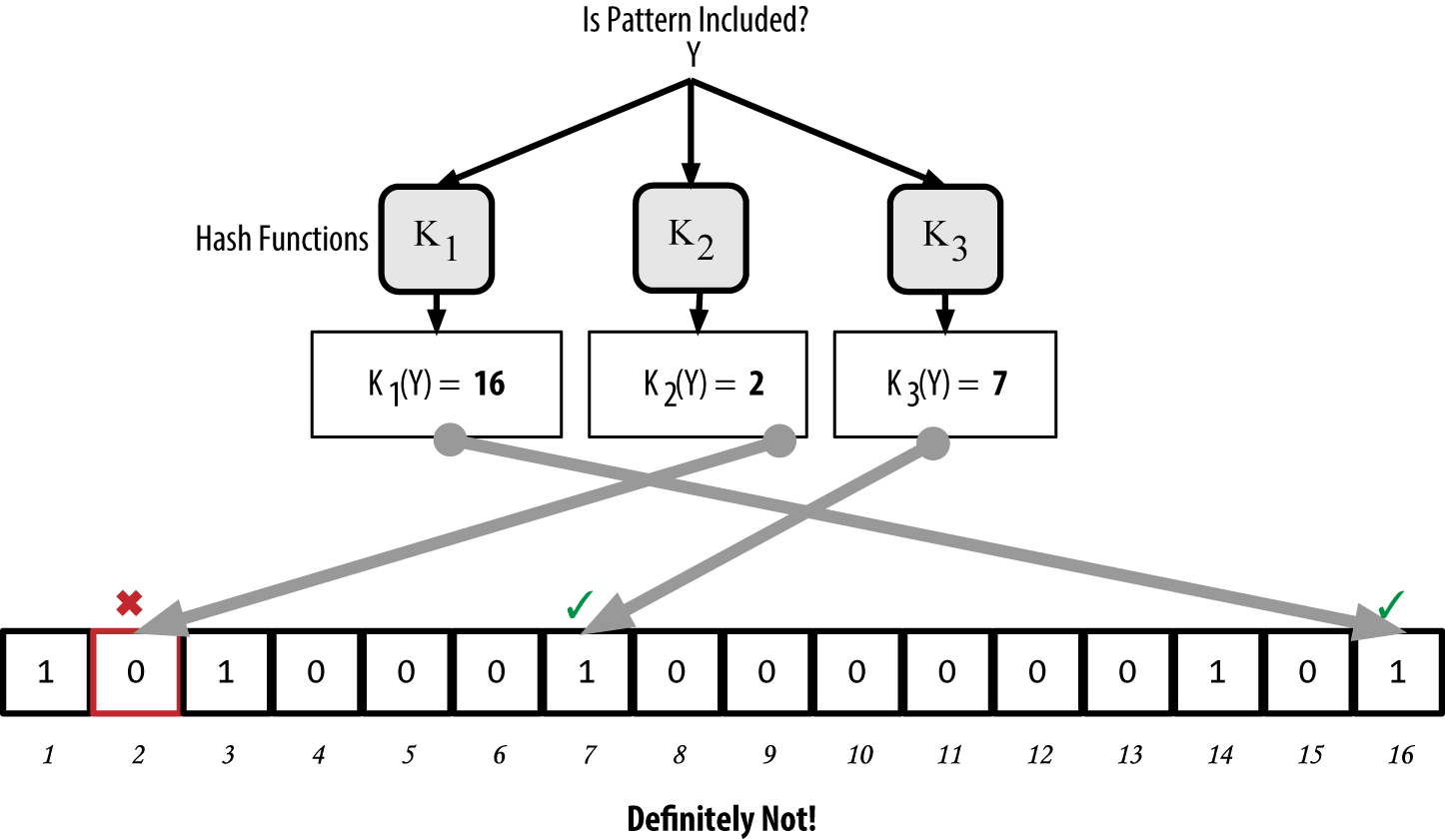
[figure\_title](#bloom4) is an example of testing the existence of pattern "X" in the simple bloom filter. The corresponding bits are set to 1, so the pattern is probably a match.



Testing the existence of pattern "X" in the bloom filter. The result is a probabilistic positive match, meaning "Maybe."

On the contrary, if a pattern is tested against the bloom filter and any one of the bits is set to 0, this proves that the pattern was not recorded in the bloom filter. A negative result is not a probability, it is a certainty. In simple terms, a negative match on a bloom filter is a "Definitely Not!"

[figure\_title](#bloom5) is an example of testing the existence of pattern "Y" in the simple bloom filter. One of the corresponding bits is set to 0, so the pattern is definitely not a match.



Testing the existence of pattern "Y" in the bloom filter. The result is a definitive negative match, meaning "Definitely Not!"

=== How SPV Nodes Use Bloom Filters

Bloom filters are used to filter the transactions (and blocks containing them) that an SPV node receives from its peers, selecting only transactions of interest to the SPV node without revealing which addresses or keys it is interested in.

An SPV node will initialize a bloom filter as "empty"; in that state the bloom filter will not match any patterns. The SPV node will then make a list of all the addresses, keys, and hashes that it is interested in. It will do this by extracting the public key hash and script hash and transaction IDs from any UTXO controlled by its wallet. The SPV node then adds each of these to the bloom filter, so that the bloom filter will "match" if these patterns are present in a transaction, without revealing the patterns themselves.

The SPV node will then send a filterload message to the peer, containing the bloom filter to use on the connection. On the peer, bloom filters are checked against each incoming transaction. The full node checks several parts of the transaction against the bloom filter, looking for a match including:

* The transaction ID
* The data components from the locking scripts of each of the transaction outputs (every key and hash in the script)
* Each of the transaction inputs
* Each of the input signature data components (or witness scripts)

By checking against all these components, bloom filters can be used to match public key hashes, scripts, OP\_RETURN values, public keys in signatures, or any future component of a smart contract or complex script.

After a filter is established, the peer will then test each transaction’s output against the bloom filter. Only transactions that match the filter are sent to the node.

In response to a getdata message from the node, peers will send a merkleblock message that contains only block headers for blocks matching the filter and a merkle path (see [simpara\_title](#merkle_trees)) for each matching transaction. The peer will then also send tx messages containing the transactions matched by the filter.

As the full node sends transactions to the SPV node, the SPV node discards any false positives and uses the correctly matched transactions to update its UTXO set and wallet balance. As it updates its own view of the UTXO set, it also modifies the bloom filter to match any future transactions referencing the UTXO it just found. The full node then uses the new bloom filter to match new transactions and the whole process repeats.

The node setting the bloom filter can interactively add patterns to the filter by sending a filteradd message. To clear the bloom filter, the node can send a filterclear message. Because it is not possible to remove a pattern from a bloom filter, a node has to clear and resend a new bloom filter if a pattern is no longer desired.

The network protocol and bloom filter mechanism for SPV nodes is defined in [BIP-37 (Peer Services)](https://bit.ly/1x6qCiO).

=== SPV Nodes and Privacy

Nodes that implement SPV have weaker privacy than a full node. A full node receives all transactions and therefore reveals no information about whether it is using some address in its wallet. An SPV node receives a filtered list of transactions related to the addresses that are in its wallet. As a result, it reduces the privacy of the owner.

Bloom filters are a way to reduce the loss of privacy. Without them, an SPV node would have to explicitly list the addresses it was interested in, creating a serious breach of privacy. However, even with bloom filters, an adversary monitoring the traffic of an SPV client or connected to it directly as a node in the P2P network can collect enough information over time to learn the addresses in the wallet of the SPV client.

=== Encrypted and Authenticated Connections

Most new users of bitcoin assume that the network communications of a Bitcoin node are encrypted. In fact, the original implementation of bitcoin communicates entirely in the clear. While this is not a major privacy concern for full nodes, it is a big problem for SPV nodes.

As a way to increase the privacy and security of the bitcoin P2P network, there are two solutions that provide encryption of the communications: *Tor Transport* and *P2P Authentication and Encryption* with BIP-150/151.

==== Tor Transport

Tor, which stands for *The Onion Routing network*, is a software project and network that offers encryption and encapsulation of data through randomized network paths that offer anonymity, untraceability and privacy.

Bitcoin Core offers several configuration options that allow you to run a Bitcoin node with its traffic transported over the Tor network. In addition, Bitcoin Core can also offer a Tor hidden service allowing other Tor nodes to connect to your node directly over Tor.

As of Bitcoin Core version 0.12, a node will offer a hidden Tor service automatically if it is able to connect to a local Tor service. If you have Tor installed and the Bitcoin Core process runs as a user with adequate permissions to access the Tor authentication cookie, it should work automatically. Use the debug flag to turn on Bitcoin Core’s debugging for the Tor service like this:

$ bitcoind --daemon --debug=tor

You should see "tor: ADD\_ONION successful" in the logs, indicating that Bitcoin Core has added a hidden service to the Tor network.

You can find more instructions on running Bitcoin Core as a Tor hidden service in the Bitcoin Core documentation (*docs/tor.md*) and various online tutorials.

==== Peer-to-Peer Authentication and Encryption

Two Bitcoin Improvement Proposals, BIP-150 and BIP-151, add support for P2P authentication and encryption in the bitcoin P2P network. These two BIPs define optional services that may be offered by compatible Bitcoin nodes. BIP-151 enables negotiated encryption for all communications between two nodes that support BIP-151. BIP-150 offers optional peer authentication that allows nodes to authenticate each other’s identity using ECDSA and private keys. BIP-150 requires that prior to authentication the two nodes have established encrypted communications as per BIP-151.

As of February 2021, BIP-150 and BIP-151 are not implemented in Bitcoin Core. However, the two proposals have been implemented by at least one alternative Bitcoin client named bcoin.

BIP-150 and BIP-151 allow users to run SPV clients that connect to a trusted full node, using encryption and authentication to protect the privacy of the SPV client.

Additionally, authentication can be used to create networks of trusted Bitcoin nodes and prevent Man-in-the-Middle attacks. Finally, P2P encryption, if deployed broadly, would strengthen the resistance of bitcoin to traffic analysis and privacy-eroding surveillance, especially in totalitarian countries where internet use is heavily controlled and monitored.

The standard is defined in [BIP-150 (Peer Authentication)](https://github.com/bitcoin/bips/blob/master/bip-0150.mediawiki) and [BIP-151 (Peer-to-Peer Communication Encryption)](https://github.com/bitcoin/bips/blob/master/bip-0151.mediawiki).

=== Transaction Pools

Almost every node on the Bitcoin network maintains a temporary list of unconfirmed transactions called the *memory pool*, *mempool*, or *transaction pool*. Nodes use this pool to keep track of transactions that are known to the network but are not yet included in the blockchain. For example, a wallet node will use the transaction pool to track incoming payments to the user’s wallet that have been received on the network but are not yet confirmed.

As transactions are received and verified, they are added to the transaction pool and relayed to the neighboring nodes to propagate on the network.

Some node implementations also maintain a separate pool of orphaned transactions. If a transaction’s inputs refer to a transaction that is not yet known, such as a missing parent, the orphan transaction will be stored temporarily in the orphan pool until the parent transaction arrives.

When a transaction is added to the transaction pool, the orphan pool is checked for any orphans that reference this transaction’s outputs (its children). Any matching orphans are then validated. If valid, they are removed from the orphan pool and added to the transaction pool, completing the chain that started with the parent transaction. In light of the newly added transaction, which is no longer an orphan, the process is repeated recursively looking for any further descendants, until no more descendants are found. Through this process, the arrival of a parent transaction triggers a cascade reconstruction of an entire chain of interdependent transactions by re-uniting the orphans with their parents all the way down the chain.

Both the transaction pool and orphan pool (where implemented) are stored in local memory and are not saved on persistent storage; rather, they are dynamically populated from incoming network messages. When a node starts, both pools are empty and are gradually populated with new transactions received on the network.

Some implementations of the Bitcoin client also maintain an UTXO database or pool, which is the set of all unspent outputs on the blockchain. Bitcoin Core users will find it in the chainstate/ folder of their client’s data directory. Although the name "UTXO pool" sounds similar to the transaction pool, it represents a different set of data. Unlike the transaction and orphan pools, the UTXO pool is not initialized empty but instead contains millions of entries of unspent transaction outputs, everything that is unspent from all the way back to the genesis block. The UTXO pool may be housed in local memory or as an indexed database table on persistent storage.

Whereas the transaction and orphan pools represent a single node’s local perspective and might vary significantly from node to node depending upon when the node was started or restarted, the UTXO pool represents the emergent consensus of the network and therefore will vary little between nodes. Furthermore, the transaction and orphan pools only contain unconfirmed transactions, while the UTXO pool only contains confirmed outputs.

== The Blockchain

=== Introduction

The blockchain data structure is an ordered, back-linked list of blocks of transactions. The blockchain can be stored as a flat file, or in a simple database. The Bitcoin Core client stores the blockchain metadata using Google’s LevelDB database. Blocks are linked "back," each referring to the previous block in the chain. The blockchain is often visualized as a vertical stack, with blocks layered on top of each other and the first block serving as the foundation of the stack. The visualization of blocks stacked on top of each other results in the use of terms such as "height" to refer to the distance from the first block, and "top" or "tip" to refer to the most recently added block.

Each block within the blockchain is identified by a hash, generated using the SHA256 cryptographic hash algorithm on the header of the block. Each block also references a previous block, known as the *parent* block, through the "previous block hash" field in the block header. In other words, each block contains the hash of its parent inside its own header. The sequence of hashes linking each block to its parent creates a chain going back all the way to the first block ever created, known as the *genesis block*.

Although a block has just one parent, it can temporarily have multiple children. Each of the children refers to the same block as its parent and contains the same (parent) hash in the "previous block hash" field. Multiple children arise during a blockchain "fork," a temporary situation that occurs when different blocks are discovered almost simultaneously by different miners (see [simpara\_title](#forks)). Eventually, only one child block becomes part of the blockchain and the "fork" is resolved. Even though a block may have more than one child, each block can have only one parent. This is because a block has one single "previous block hash" field referencing its single parent.

The "previous block hash" field is inside the block header and thereby affects the *current* block’s hash. The child’s own identity changes if the parent’s identity changes. When the parent is modified in any way, the parent’s hash changes. The parent’s changed hash necessitates a change in the "previous block hash" pointer of the child. This in turn causes the child’s hash to change, which requires a change in the pointer of the grandchild, which in turn changes the grandchild, and so on. This cascade effect ensures that once a block has many generations following it, it cannot be changed without forcing a recalculation of all subsequent blocks. Because such a recalculation would require enormous computation (and therefore energy consumption), the existence of a long chain of blocks makes the blockchain’s deep history immutable, which is a key feature of bitcoin’s security.

One way to think about the blockchain is like layers in a geological formation, or glacier core sample. The surface layers might change with the seasons, or even be blown away before they have time to settle. But once you go a few inches deep, geological layers become more and more stable. By the time you look a few hundred feet down, you are looking at a snapshot of the past that has remained undisturbed for millions of years. In the blockchain, the most recent few blocks might be revised if there is a chain recalculation due to a fork. The top six blocks are like a few inches of topsoil. But once you go more deeply into the blockchain, beyond six blocks, blocks are less and less likely to change. After 100 blocks back, there is so much stability that the coinbase transaction—the transaction containing newly mined bitcoin—can be spent. A few thousand blocks back (a month) and the blockchain is settled history, for all practical purposes. While the protocol always allows a chain to be undone by a longer chain and while the possibility of any block being reversed always exists, the probability of such an event decreases as time passes until it becomes infinitesimal.

=== Structure of a Block

A block is a container data structure that aggregates transactions for inclusion in the public ledger, the blockchain. The block is made of a header, containing metadata, followed by a long list of transactions that make up the bulk of its size. The block header is 80 bytes, whereas the average transaction is at least 400 bytes and the average block contains more than 1900 transactions. A complete block, with all transactions, is therefore 10,000 times larger than the block header. [table\_title](#block_structure1) describes the structure of a block.

The structure of a block

| Size | Field | Description |
| --- | --- | --- |
| 4 bytes | Block Size | The size of the block, in bytes, following this field |
| 80 bytes | Block Header | Several fields form the block header |
| 1–9 bytes (VarInt) | Transaction Counter | How many transactions follow |
| Variable | Transactions | The transactions recorded in this block |

=== Block Header

The block header consists of three sets of block metadata. First, there is a reference to a previous block hash, which connects this block to the previous block in the blockchain. The second set of metadata, namely the *difficulty*, *timestamp*, and *nonce*, relate to the mining competition, as detailed in [simpara\_title](#mining). The third piece of metadata is the merkle tree root, a data structure used to efficiently summarize all the transactions in the block. [table\_title](#block_header_structure_ch09) describes the structure of a block header.

The structure of the block header

| Size | Field | Description |
| --- | --- | --- |
| 4 bytes | Version | A version number to track software/protocol upgrades |
| 32 bytes | Previous Block Hash | A reference to the hash of the previous (parent) block in the chain |
| 32 bytes | Merkle Root | A hash of the root of the merkle tree of this block’s transactions |
| 4 bytes | Timestamp | The approximate creation time of this block (in seconds elapsed since Unix Epoch) |
| 4 bytes | Difficulty Target | The Proof-of-Work algorithm difficulty target for this block |
| 4 bytes | Nonce | A counter used for the Proof-of-Work algorithm |

The nonce, difficulty target, and timestamp are used in the mining process and will be discussed in more detail in [simpara\_title](#mining).

=== Block Identifiers: Block Header Hash and Block Height

The primary identifier of a block is its cryptographic hash, a digital fingerprint, made by hashing the block header twice through the SHA256 algorithm. The resulting 32-byte hash is called the *block hash* but is more accurately the *block header hash*, because only the block header is used to compute it. For example, 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f is the block hash of the first bitcoin block ever created. The block hash identifies a block uniquely and unambiguously and can be independently derived by any node by simply double hashing the block header with the SHA256 algorithm.

Note that the block hash is not actually included inside the block’s data structure, neither when the block is transmitted on the network, nor when it is stored on a node’s persistence storage as part of the blockchain. Instead, the block’s hash is computed by each node as the block is received from the network. The block hash might be stored in a separate database table as part of the block’s metadata, to facilitate indexing and faster retrieval of blocks from disk.

A second way to identify a block is by its position in the blockchain, called the block height. The first block ever created is at block height 0 (zero) and is the same block that was previously referenced by the following block hash 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. A block can thus be identified in two ways: by referencing the block hash or by referencing the block height. Each subsequent block added "on top" of that first block is one position "higher" in the blockchain, like boxes stacked one on top of the other.

In addition, the term *current block height* indicates the size of the blockchain in blocks at any given time. For example, the current block height on March 1, 2021 was approximately 672,722, meaning there were 672,722 blocks stacked on top of the first block created in January 2009.

Unlike the block hash, the block height is not a unique identifier. Although a single block will always have a specific and invariant block height, the reverse is not true—the block height does not always identify a single block. Two or more blocks might have the same block height, competing for the same position in the blockchain. This scenario is discussed in detail in the section [simpara\_title](#forks). The block height is also not a part of the block’s data structure; it is not stored within the block. Each node dynamically identifies a block’s position (height) in the blockchain when it is received from the Bitcoin network. The block height might also be stored as metadata in an indexed database table for faster retrieval.

A block’s *block hash* always identifies a single block uniquely. A block also always has a specific *block height*. However, it is not always the case that a specific block height can identify a single block. Rather, two or more blocks might compete for a single position in the blockchain.

=== The Genesis Block

The first block in the blockchain is called the genesis block and was created in 2009. It is the common ancestor of all the blocks in the blockchain, meaning that if you start at any block and follow the chain backward in time, you will eventually arrive at the genesis block.

Every node always starts with a blockchain of at least one block because the genesis block is statically encoded within the Bitcoin client software, such that it cannot be altered. Every node always "knows" the genesis block’s hash and structure, the fixed time it was created, and even the single transaction within. Thus, every node has the starting point for the blockchain, a secure "root" from which to build a trusted blockchain.

See the statically encoded genesis block inside the Bitcoin Core client, in [*chainparams.cpp*](https://bit.ly/1x6rcwP).

The following identifier hash belongs to the genesis block:

000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

You can search for that block hash in any block explorer website, such as *Blockchain.com*, and you will find a page describing the contents of this block, with a URL containing that hash:

<https://www.blockchain.com/btc/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f>

Using the Bitcoin Core reference client on the command line:

$ 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"  
}

The genesis block contains a hidden message within it. The coinbase transaction input contains the text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks." This message was intended to offer proof of the earliest date this block was created, by referencing the headline of the British newspaper *The Times*. It also serves as a tongue-in-cheek reminder of the importance of an independent monetary system, with bitcoin’s launch occurring at the same time as an unprecedented worldwide monetary crisis. The message was embedded in the first block by Satoshi Nakamoto, bitcoin’s creator.

=== Linking Blocks in the Blockchain

Bitcoin full nodes maintain a local copy of the blockchain, starting at the genesis block. The local copy of the blockchain is constantly updated as new blocks are found and used to extend the chain. As a node receives incoming blocks from the network, it will validate these blocks and then link them to the existing blockchain. To establish a link, a node will examine the incoming block header and look for the "previous block hash."

Let’s assume, for example, that a node has 277,314 blocks in the local copy of the blockchain. The last block the node knows about is block 277,314, with a block header hash of:

00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249

The Bitcoin node then receives a new block from the network, which it parses as follows:

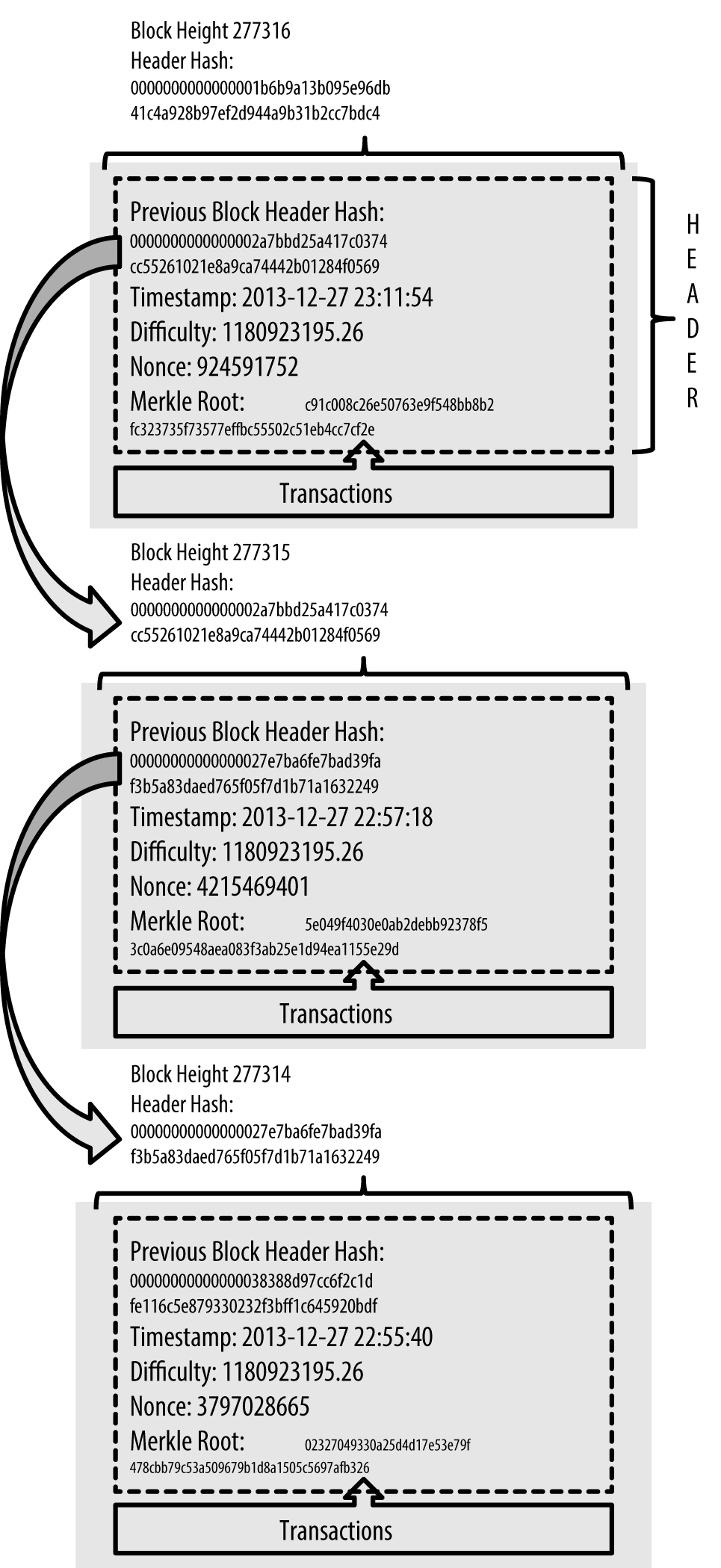
{  
 "size" : 43560,  
 "version" : 2,  
 "previousblockhash" :  
 "00000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",  
 "merkleroot" :  
 "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",  
 "time" : 1388185038,  
 "difficulty" : 1180923195.25802612,  
 "nonce" : 4215469401,  
 "tx" : [  
 "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",  
  
 #[... many more transactions omitted ...]  
  
 "05cfd38f6ae6aa83674cc99e4d75a1458c165b7ab84725eda41d018a09176634"  
 ]  
}

Looking at this new block, the node finds the previousblockhash field, which contains the hash of its parent block. It is a hash known to the node, that of the last block on the chain at height 277,314. Therefore, this new block is a child of the last block on the chain and extends the existing blockchain. The node adds this new block to the end of the chain, making the blockchain longer with a new height of 277,315. [figure\_title](#chain_of_blocks) shows the chain of three blocks, linked by references in the previousblockhash field.

=== Merkle Trees

Each block in the Bitcoin blockchain contains a summary of all the transactions in the block using a *merkle tree*.

A *merkle tree*, also known as a *binary hash tree*, is a data structure used for efficiently summarizing and verifying the integrity of large sets of data. Merkle trees are binary trees containing cryptographic hashes. The term "tree" is used in computer science to describe a branching data structure, but these trees are usually displayed upside down with the "root" at the top and the "leaves" at the bottom of a diagram, as you will see in the examples that follow.



Blocks linked in a chain by reference to the previous block header hash

Merkle trees are used in bitcoin to summarize all the transactions in a block, producing an overall digital fingerprint of the entire set of transactions, providing a very efficient process to verify whether a transaction is included in a block. A merkle tree is constructed by recursively hashing pairs of nodes until there is only one hash, called the *root*, or *merkle root*. The cryptographic hash algorithm used in bitcoin’s merkle trees is SHA256 applied twice, also known as double-SHA256.

When N data elements are hashed and summarized in a merkle tree, you can check to see if any one data element is included in the tree with at most 2\*log~2~(N) calculations, making this a very efficient data structure.

The merkle tree is constructed bottom-up. In the following example, we start with four transactions, A, B, C, and D, which form the *leaves* of the merkle tree, as shown in [figure\_title](#simple_merkle). The transactions are not stored in the merkle tree; rather, their data is hashed and the resulting hash is stored in each leaf node as HA, HB, HC, and HD:

H

A

= SHA256(SHA256(Transaction A))

Consecutive pairs of leaf nodes are then summarized in a parent node, by concatenating the two hashes and hashing them together. For example, to construct the parent node HAB, the two 32-byte hashes of the children are concatenated to create a 64-byte string. That string is then double-hashed to produce the parent node’s hash:

H

AB

= SHA256(SHA256(H

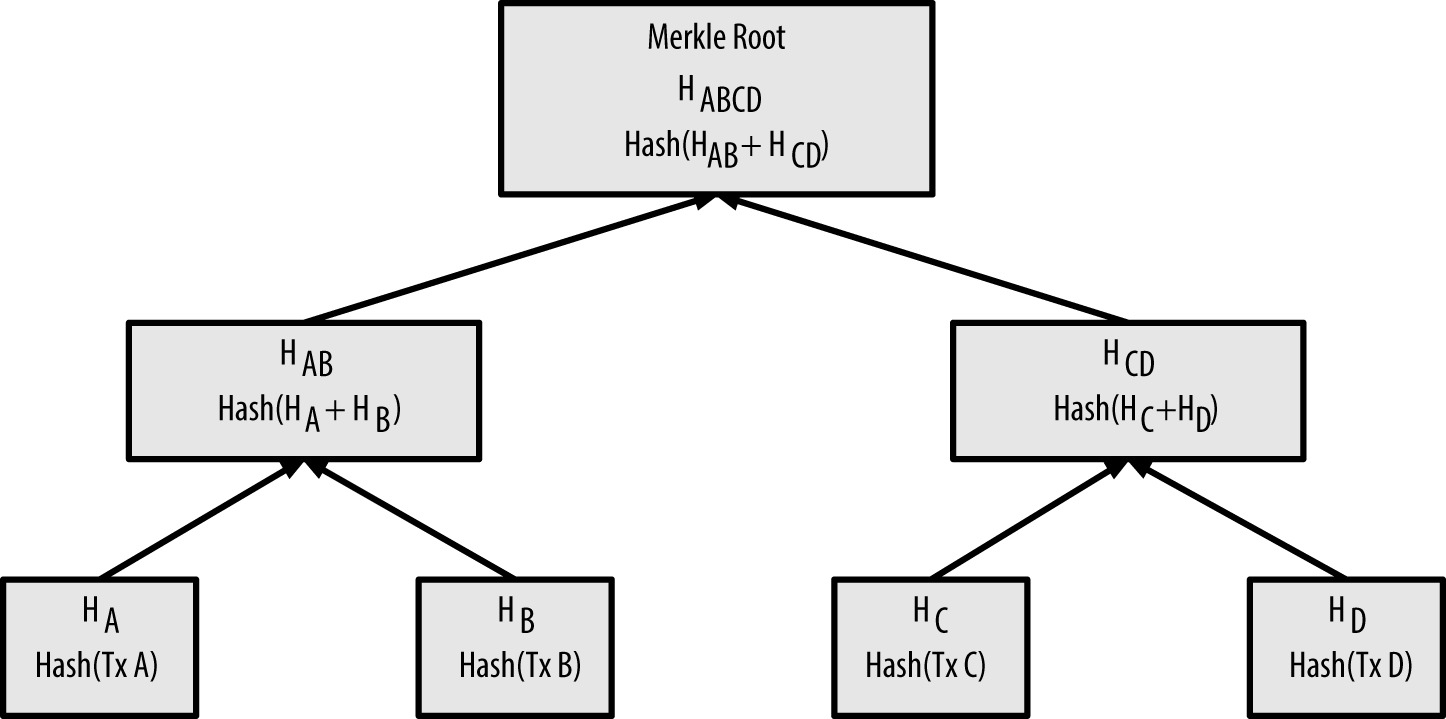
A

+ H

B

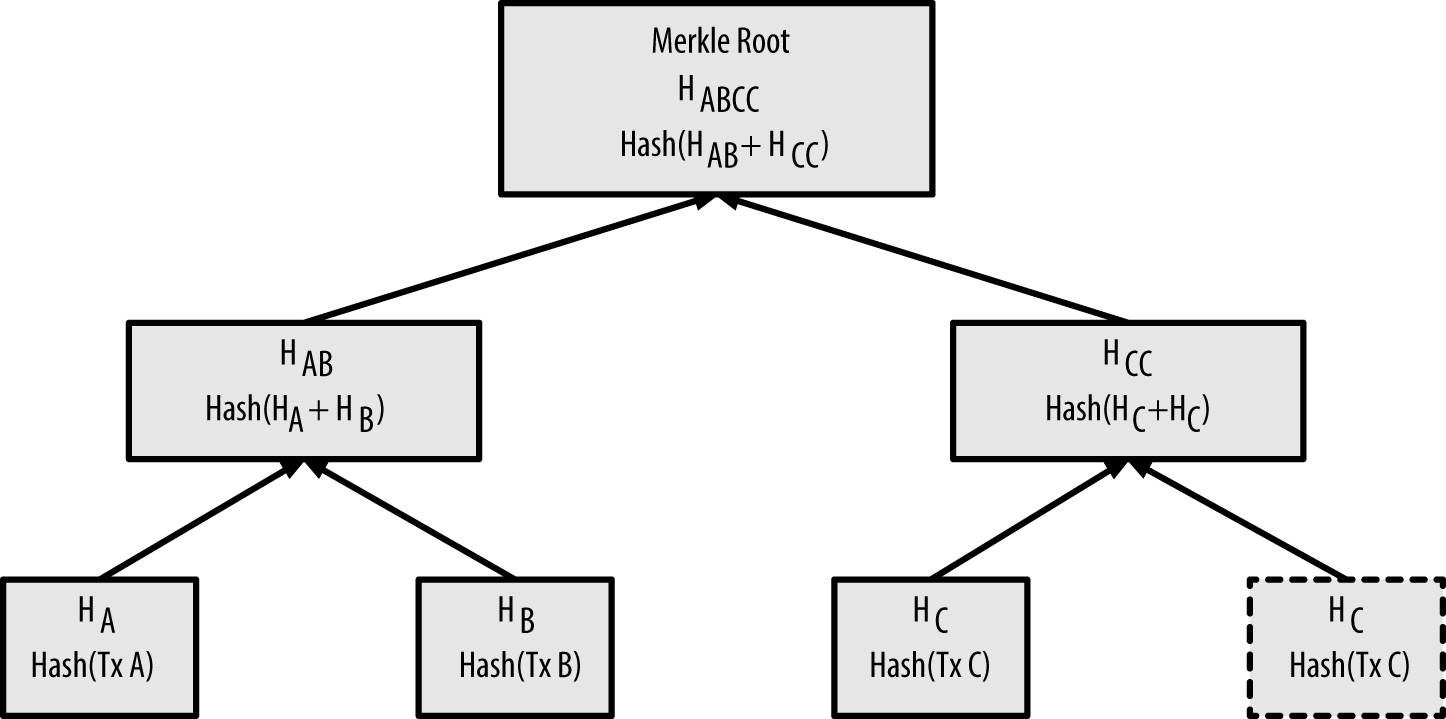
))

The process continues until there is only one node at the top, the node known as the merkle root. That 32-byte hash is stored in the block header and summarizes all the data in all four transactions. [figure\_title](#simple_merkle) shows how the root is calculated by pair-wise hashes of the nodes.



Calculating the nodes in a merkle tree

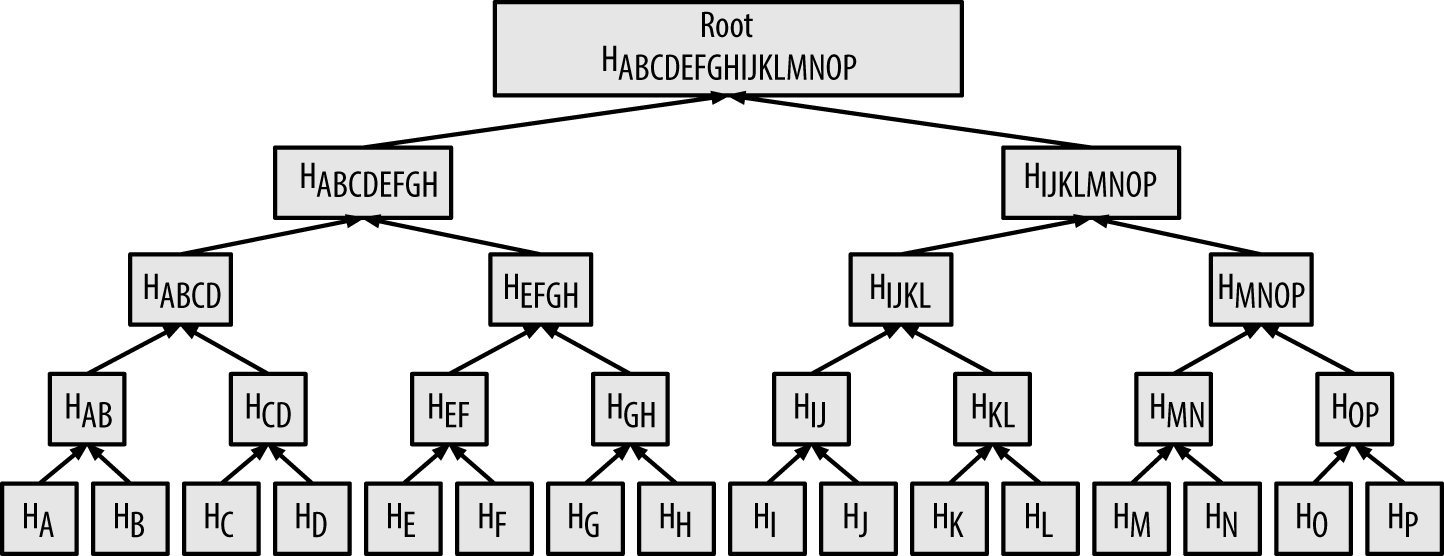
Because the merkle tree is a binary tree, it needs an even number of leaf nodes. If there is an odd number of transactions to summarize, the last transaction hash will be duplicated to create an even number of leaf nodes, also known as a *balanced tree*. This is shown in [figure\_title](#merkle_tree_odd), where transaction C is duplicated.



Duplicating one data element achieves an even number of data elements

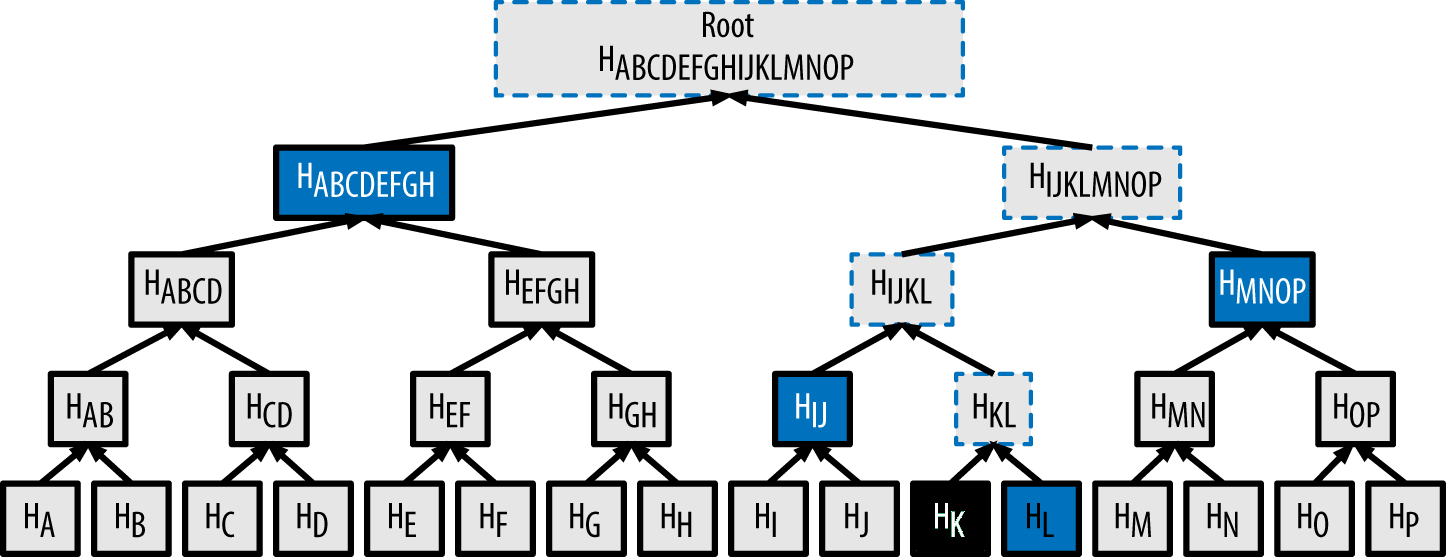
The same method for constructing a tree from four transactions can be generalized to construct trees of any size. In bitcoin it is common to have several hundred to more than a thousand transactions in a single block, which are summarized in exactly the same way, producing just 32 bytes of data as the single merkle root. In [figure\_title](#merkle_tree_large), you will see a tree built from 16 transactions. Note that although the root looks bigger than the leaf nodes in the diagram, it is the exact same size, just 32 bytes. Whether there is one transaction or a hundred thousand transactions in the block, the merkle root always summarizes them into 32 bytes.

To prove that a specific transaction is included in a block, a node only needs to produce log~2~(N) 32-byte hashes, constituting an *authentication path* or *merkle path* connecting the specific transaction to the root of the tree. This is especially important as the number of transactions increases, because the base-2 logarithm of the number of transactions increases much more slowly. This allows Bitcoin nodes to efficiently produce paths of 10 or 12 hashes (320–384 bytes), which can provide proof of a single transaction out of more than a thousand transactions in a megabyte-sized block.



A merkle tree summarizing many data elements

In [figure\_title](#merkle_tree_path), a node can prove that a transaction K is included in the block by producing a merkle path that is only four 32-byte hashes long (128 bytes total). The path consists of the four hashes (shown with a shaded background in [figure\_title](#merkle_tree_path)) HL, HIJ, HMNOP, and HABCDEFGH. With those four hashes provided as an authentication path, any node can prove that HK (with a black background at the bottom of the diagram) is included in the merkle root by computing four additional pair-wise hashes HKL, HIJKL, HIJKLMNOP, and the merkle tree root (outlined in a dashed line in the diagram).



A merkle path used to prove inclusion of a data element

The code in [example\_title](#merkle_example) demonstrates the process of creating a merkle tree from the leaf-node hashes up to the root, using the libbitcoin library for some helper functions.

#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;  
}

[example\_title](#merkle_example_run) shows the result of compiling and running the merkle code.

# Compile the merkle.cpp code  
$ g++ -o merkle merkle.cpp $(pkg-config --cflags --libs libbitcoin)  
# Run the merkle executable  
$ ./merkle  
Current merkle hash list:  
 32650049a0418e4380db0af81788635d8b65424d397170b8499cdc28c4d27006  
 30861db96905c8dc8b99398ca1cd5bd5b84ac3264a4e1b3e65afa1bcee7540c4  
  
Current merkle hash list:  
 d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3  
  
Result: d47780c084bad3830bcdaf6eace035e4c6cbf646d103795d22104fb105014ba3

The efficiency of merkle trees becomes obvious as the scale increases. [table\_title](#block_structure2) shows the amount of data that needs to be exchanged as a merkle path to prove that a transaction is part of a block.

Merkle tree efficiency

| Number of transactions | Approx. size of block | Path size (hashes) | Path size (bytes) |
| --- | --- | --- | --- |
| 16 transactions | 4 kilobytes | 4 hashes | 128 bytes |
| 512 transactions | 128 kilobytes | 9 hashes | 288 bytes |
| 2048 transactions | 512 kilobytes | 11 hashes | 352 bytes |
| 65,535 transactions | 16 megabytes | 16 hashes | 512 bytes |

As you can see from the table, while the block size increases rapidly, from 4 KB with 16 transactions to a block size of 16 MB to fit 65,535 transactions, the merkle path required to prove the inclusion of a transaction increases much more slowly, from 128 bytes to only 512 bytes. With merkle trees, a node can download just the block headers (80 bytes per block) and still be able to identify a transaction’s inclusion in a block by retrieving a small merkle path from a full node, without storing or transmitting the vast majority of the blockchain, which might be several gigabytes in size. Nodes that do not maintain a full blockchain, called simplified payment verification (SPV) nodes, use merkle paths to verify transactions without downloading full blocks.

=== Merkle Trees and Simplified Payment Verification (SPV)

Merkle trees are used extensively by SPV nodes. SPV nodes don’t have all transactions and do not download full blocks, just block headers. In order to verify that a transaction is included in a block, without having to download all the transactions in the block, they use an authentication path, or merkle path.

Consider, for example, an SPV node that is interested in incoming payments to an address contained in its wallet. The SPV node will establish a bloom filter (see [simpara\_title](#bloom_filters)) on its connections to peers to limit the transactions received to only those containing addresses of interest. When a peer sees a transaction that matches the bloom filter, it will send that block using a merkleblock message. The merkleblock message contains the block header as well as a merkle path that links the transaction of interest to the merkle root in the block. The SPV node can use this merkle path to connect the transaction to the block and verify that the transaction is included in the block. The SPV node also uses the block header to link the block to the rest of the blockchain. The combination of these two links, between the transaction and block, and between the block and blockchain, proves that the transaction is recorded in the blockchain. All in all, the SPV node will have received less than a kilobyte of data for the block header and merkle path, an amount of data that is more than a thousand times less than a full block (about 1 megabyte currently).

=== Bitcoin’s Test Blockchains

You might be surprised to learn that there is more than one Bitcoin blockchain. The "main" Bitcoin blockchain, the one created by Satoshi Nakamoto on January 3rd, 2009, the one with the genesis block we studied in this chapter, is called *mainnet*. There are other Bitcoin blockchains that are used for testing purposes: at this time *testnet*, *segnet*, and *regtest*. Let’s look at each in turn.

==== Testnet—Bitcoin’s Testing Playground

Testnet is the name of the test blockchain, network, and currency that is used for testing purposes. The testnet is a fully featured live P2P network, with wallets, test bitcoin (testnet coins), mining, and all the other features of mainnet. There are really only two differences: testnet coins are meant to be worthless and mining difficulty should be low enough that anyone can mine testnet coins relatively easily (keeping them worthless).

Any software development that is intended for production use on bitcoin’s mainnet should first be tested on testnet with test coins. This protects both the developers from monetary losses due to bugs and the network from unintended behavior due to bugs.

Keeping the coins worthless and the mining easy, however, is not easy. Despite pleas from developers, some people use advanced mining equipment (GPUs and ASICs) to mine on testnet. This increases the difficulty, makes it impossible to mine with a CPU, and eventually makes it difficult enough to get test coins that people start valuing them, so they’re not worthless. As a result, every now and then, the testnet has to be scrapped and restarted from a new genesis block, resetting the difficulty.

The current testnet is called *testnet3*, the third iteration of testnet, restarted in February 2011 to reset the difficulty from the previous testnet.

Keep in mind that testnet3 is a large blockchain, in excess of 25 GB in 2021. It will take a day or so to sync fully and use up resources on your computer. Not as much as mainnet, but not exactly "lightweight" either. One good way to run a testnet node is as a virtual machine image (e.g., VirtualBox, Docker, Cloud Server, etc.) dedicated for that purpose.

===== Using testnet

Bitcoin Core, like almost all other bitcoin software, has full support for operation on testnet instead of mainnet. All of Bitcoin Core’s functions work on testnet, including the wallet, mining testnet coins, and syncing a full testnet node.

To start Bitcoin Core on testnet instead of mainnet you use the testnet switch:

$ bitcoind -testnet

In the logs you should see that bitcoind is building a new blockchain in the testnet3 subdirectory of the default bitcoind directory:

bitcoind: Using data directory /home/username/.bitcoin/testnet3

To connect to bitcoind, you use the bitcoin-cli command-line tool, but you must also switch it to testnet mode:

$ bitcoin-cli -testnet getblockchaininfo  
{  
 "chain": "test",  
 "blocks": 1088,  
 "headers": 139999,  
 "bestblockhash": "0000000063d29909d475a1c4ba26da64b368e56cce5d925097bf3a2084370128",  
 "difficulty": 1,  
 "mediantime": 1337966158,  
 "verificationprogress": 0.001644065914099759,  
 "chainwork": "0000000000000000000000000000000000000000000000000000044104410441",  
 "pruned": false,  
 "softforks": [  
  
 [...]

You can also run on testnet3 with other full-node implementations, such as btcd (written in Go) and bcoin (written in JavaScript), to experiment and learn in other programming languages and frameworks.

In 2021, testnet3 supports all the features of mainnet, including Segregated Witness (see [simpara\_title](#segwit)). Therefore, testnet3 can also be used to test Segregated Witness features.

==== Segnet—The Segregated Witness Testnet

In 2016, a special-purpose testnet was launched to aid in development and testing of Segregated Witness (aka segwit; see [simpara\_title](#segwit)). This test blockchain is called segnet and can be joined by running a special version (branch) of Bitcoin Core.

Since segwit was added to testnet3, it is no longer necessary to use segnet for testing of segwit features.

In the future it is likely we will see other testnet blockchains that are specifically designed to test a single feature or major architectural change, like segnet.

==== Regtest—The Local Blockchain

Regtest, which stands for "Regression Testing," is a Bitcoin Core feature that allows you to create a local blockchain for testing purposes. Unlike testnet3, which is a public and shared test blockchain, the regtest blockchains are intended to be run as closed systems for local testing. You launch a regtest blockchain from scratch, creating a local genesis block. You may add other nodes to the network, or run it with a single node only to test the Bitcoin Core software.

To start Bitcoin Core in regtest mode, you use the regtest flag:

$ bitcoind -regtest

Just like with testnet, Bitcoin Core will initialize a new blockchain under the *regtest* subdirectory of your bitcoind default directory:

bitcoind: Using data directory /home/username/.bitcoin/regtest

To use the command-line tool, you need to specify the regtest flag too. Let’s try the getblockchaininfo command to inspect the regtest blockchain:

$ bitcoin-cli -regtest getblockchaininfo  
{  
 "chain": "regtest",  
 "blocks": 0,  
 "headers": 0,  
 "bestblockhash": "0f9188f13cb7b2c71f2a335e3a4fc328bf5beb436012afca590b1a11466e2206",  
 "difficulty": 4.656542373906925e-10,  
 "mediantime": 1296688602,  
 "verificationprogress": 1,  
 "chainwork": "0000000000000000000000000000000000000000000000000000000000000002",  
 "pruned": false,  
 [...]

As you can see, there are no blocks yet. Let’s mine some (500 blocks) and earn the reward:

$ bitcoin-cli -regtest generate 500  
[  
 "7afed70259f22c2bf11e406cb12ed5c0657b6e16a6477a9f8b28e2046b5ba1ca",  
 "1aca2f154a80a9863a9aac4c72047a6d3f385c4eec5441a4aafa6acaa1dada14",  
 "4334ecf6fb022f30fbd764c3ee778fabbd53b4a4d1950eae8a91f1f5158ed2d1",  
 "5f951d34065efeaf64e54e91d00b260294fcdfc7f05dbb5599aec84b957a7766",  
 "43744b5e77c1dfece9d05ab5f0e6796ebe627303163547e69e27f55d0f2b9353",  
 [...]  
 "6c31585a48d4fc2b3fd25521f4515b18aefb59d0def82bd9c2185c4ecb754327"  
]

It will only take a few seconds to mine all these blocks, which certainly makes it easy for testing. If you check your wallet balance, you will see that you earned reward for the first 400 blocks (coinbase rewards must be 100 blocks deep before you can spend them):

$ bitcoin-cli -regtest getbalance  
12462.50000000

=== Using Test Blockchains for Development

Bitcoin’s various blockchains (regtest, segnet, testnet3, mainnet) offer a range of testing environments for bitcoin development. Use the test blockchains whether you are developing for Bitcoin Core, or another full-node consensus client; an application such as a wallet, exchange, ecommerce site; or even developing novel smart contracts and complex scripts.

You can use the test blockchains to establish a development pipeline. Test your code locally on a regtest as you develop it. Once you are ready to try it on a public network, switch to testnet to expose your code to a more dynamic environment with more diversity of code and applications. Finally, once you are confident your code works as expected, switch to mainnet to deploy it in production. As you make changes, improvements, bug fixes, etc., start the pipeline again, deploying each change first on regtest, then on testnet, and finally into production.

== Mining and Consensus

=== Introduction

The word "mining" is somewhat misleading. By evoking the extraction of precious metals, it focuses our attention on the reward for mining, the new bitcoin created in each block. Although mining is incentivized by this reward, the primary purpose of mining is not the reward or the generation of new coins. If you view mining only as the process by which coins are created, you are mistaking the means (incentives) as the goal of the process. Mining is the mechanism that underpins the decentralized clearinghouse, by which transactions are validated and cleared. Mining is the invention that makes bitcoin special, a decentralized security mechanism that is the basis for P2P digital cash.

Mining *secures the Bitcoin system* and enables the emergence of network-wide *consensus without a central authority*. The reward of newly minted coins and transaction fees is an incentive scheme that aligns the actions of miners with the security of the network, while simultaneously implementing the monetary supply.

The purpose of mining is not the creation of new bitcoin. That’s the incentive system. Mining is the mechanism by which bitcoin’s *security* is *decentralized*.

Miners validate new transactions and record them on the global ledger. A new block, containing transactions that occurred since the last block, is "mined" every 10 minutes on average, thereby adding those transactions to the blockchain. Transactions that become part of a block and added to the blockchain are considered "confirmed," which allows the new owners of bitcoin to spend the bitcoin they received in those transactions.

Miners receive two types of rewards in return for the security provided by mining: new coins created with each new block, also known as a *block reward* or *coinbase reward*, and transaction fees from all the transactions included in the block. To earn this reward, miners compete to solve a difficult mathematical problem based on a cryptographic hash algorithm. The solution to the problem, called the Proof-of-Work, is included in the new block and acts as proof that the miner expended significant computing effort. The competition to solve the Proof-of-Work algorithm to earn the reward and the right to record transactions on the blockchain is the basis for bitcoin’s security model.

The process is called mining because the reward (new coin generation) is designed to simulate diminishing returns, just like mining for precious metals. Bitcoin’s money supply is created through mining, similar to how a central bank issues new money by printing bank notes. The maximum amount of newly created bitcoin a miner can add to a block decreases approximately every four years (or precisely every 210,000 blocks). It started at 50 bitcoin per block in January of 2009 and halved to 25 bitcoin per block in November of 2012. It halved to 12.5 bitcoin in July 2016 and again to 6.25 bitcoin in May 2020. Based on this formula, bitcoin mining rewards decrease exponentially until approximately the year 2140, when all bitcoin (20.99999998 million) will have been issued. After 2140, no new bitcoin will be issued.

Bitcoin miners also earn fees from transactions. Every transaction usually includes a transaction fee, in the form of a surplus of bitcoin between the transaction’s inputs and outputs. The winning bitcoin miner gets to "keep the change" on the transactions included in the winning block. Today, the fees represent 0.5% or less of a bitcoin miner’s income, the vast majority coming from the newly minted bitcoin. However, as the block reward decreases over time and the number of transactions per block increases, a greater proportion of bitcoin mining earnings will come from fees. Gradually, the mining reward will be dominated by transaction fees, which will form the primary incentive for miners. After 2140, the amount of new bitcoin in each block drops to zero and bitcoin mining will be incentivized only by transaction fees.

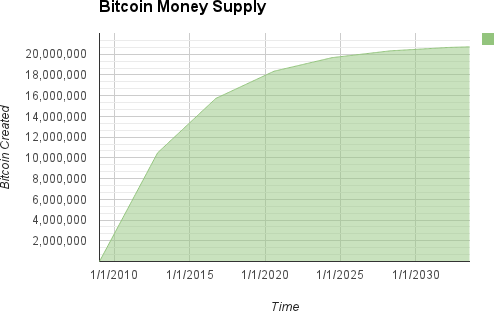
In this chapter, we will first examine mining as a monetary supply mechanism and then look at the most important function of mining: the decentralized consensus mechanism that underpins bitcoin’s security.

To understand mining and consensus, we will follow Alice’s transaction as it is received and added to a block by Jing’s mining equipment. Then we will follow the block as it is mined, added to the blockchain, and accepted by the Bitcoin network through the process of emergent consensus.

==== Bitcoin Economics and Currency Creation

Bitcoin are "minted" during the creation of each block at a fixed and diminishing rate. Each block, generated on average every 10 minutes, contains entirely new bitcoin, created from nothing. Every 210,000 blocks, or approximately every four years, the currency issuance rate is decreased by 50%. For the first four years of operation of the network, each block contained 50 new bitcoin.

In November 2012, the new bitcoin issuance rate was decreased to 25 bitcoin per block. In July of 2016 it was decreased to 12.5 bitcoin per block, and in May of 2020 it was decreased again to 6.25 bitcoin per block. The rate of new coins decreases like this exponentially over 32 "halvings" until block 6,720,000 (mined approximately in year 2137), when it reaches the minimum currency unit of 1 satoshi. Finally, after 6.93 million blocks, in approximately 2140, almost 2,099,999,997,690,000 satoshis, or almost 21 million bitcoin, will be issued. Thereafter, blocks will contain no new bitcoin, and miners will be rewarded solely through the transaction fees. [figure\_title](#bitcoin_money_supply) shows the total bitcoin in circulation over time, as the issuance of currency decreases.



Supply of bitcoin currency over time based on a geometrically decreasing issuance rate

The maximum number of coins mined is the *upper limit* of possible mining rewards for bitcoin. In practice, a miner may intentionally mine a block taking less than the full reward. Such blocks have already been mined and more may be mined in the future, resulting in a lower total issuance of the currency.

In the example code in [example\_title](#max_money), we calculate the total amount of bitcoin that will be issued.

# Original block reward for miners was 50 BTC  
start\_block\_reward = 50  
# 210000 is around every 4 years with a 10 minute block interval  
reward\_interval = 210000  
  
def max\_money():  
 # 50 BTC = 50 0000 0000 Satoshis  
 current\_reward = 50 \* 10\*\*8  
 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"

[example\_title](#max_money_run) shows the output produced by running this script.

$ python max\_money.py  
Total BTC to ever be created: 2100000000000000.0 Satoshis

The finite and diminishing issuance creates a fixed monetary supply that resists inflation. Unlike a fiat currency, which can be printed in infinite numbers by a central bank, bitcoin can never be inflated by printing.

The most important and debated consequence of fixed and diminishing monetary issuance is that the currency tends to be inherently *deflationary*. Deflation is the phenomenon of appreciation of value due to a mismatch in supply and demand that drives up the value (and exchange rate) of a currency. The opposite of inflation, price deflation, means that the money has more purchasing power over time.

Many economists argue that a deflationary economy is a disaster that should be avoided at all costs. That is because in a period of rapid deflation, people tend to hoard money instead of spending it, hoping that prices will fall. Such a phenomenon unfolded during Japan’s "Lost Decade," when a complete collapse of demand pushed the currency into a deflationary spiral.

Bitcoin experts argue that deflation is not bad per se. Rather, deflation is associated with a collapse in demand because that is the only example of deflation we have to study. In a fiat currency with the possibility of unlimited printing, it is very difficult to enter a deflationary spiral unless there is a complete collapse in demand and an unwillingness to print money. Deflation in bitcoin is not caused by a collapse in demand, but by a predictably constrained supply.

The positive aspect of deflation, of course, is that it is the opposite of inflation. Inflation causes a slow but inevitable debasement of currency, resulting in a form of hidden taxation that punishes savers in order to bail out debtors (including the biggest debtors, governments themselves). Currencies under government control suffer from the moral hazard of easy debt issuance that can later be erased through debasement at the expense of savers.

It remains to be seen whether the deflationary aspect of the currency is a problem when it is not driven by rapid economic retraction, or an advantage because the protection from inflation and debasement far outweighs the risks of deflation.

=== Decentralized Consensus

In the previous chapter we looked at the blockchain, the global public ledger (list) of all transactions, which everyone in the Bitcoin network accepts as the authoritative record of ownership.

But how can everyone in the network agree on a single universal "truth" about who owns what, without having to trust anyone? All traditional payment systems depend on a trust model that has a central authority providing a clearinghouse service, basically verifying and clearing all transactions. Bitcoin has no central authority, yet somehow every full node has a complete copy of a public ledger that it can trust as the authoritative record. The blockchain is not created by a central authority, but is assembled independently by every node in the network. Somehow, every node in the network, acting on information transmitted across insecure network connections, can arrive at the same conclusion and assemble a copy of the same public ledger as everyone else. This chapter examines the process by which the Bitcoin network achieves global consensus without central authority.

Satoshi Nakamoto’s main invention is the decentralized mechanism for *emergent consensus*. Emergent, because consensus is not achieved explicitly—there is no election or fixed moment when consensus occurs. Instead, consensus is an emergent artifact of the asynchronous interaction of thousands of independent nodes, all following simple rules. All the properties of bitcoin, including currency, transactions, payments, and the security model that does not depend on central authority or trust, derive from this invention.

Bitcoin’s decentralized consensus emerges from the interplay of four processes that occur independently on nodes across the network:

* Independent verification of each transaction, by every full node, based on a comprehensive list of criteria
* Independent aggregation of those transactions into new blocks by mining nodes, coupled with demonstrated computation through a Proof-of-Work algorithm
* Independent verification of the new blocks by every node and assembly into a chain
* Independent selection, by every node, of the chain with the most cumulative computation demonstrated through Proof-of-Work

In the next few sections we will examine these processes and how they interact to create the emergent property of network-wide consensus that allows any Bitcoin node to assemble its own copy of the authoritative, trusted, public, global ledger.

=== Independent Verification of Transactions

In [simpara\_title](#transactions), we saw how wallet software creates transactions by collecting UTXO, providing the appropriate unlocking scripts, and then constructing new outputs assigned to a new owner. The resulting transaction is then sent to the neighboring nodes in the Bitcoin network so that it can be propagated across the entire Bitcoin network.

However, before forwarding transactions to its neighbors, every Bitcoin node that receives a transaction will first verify the transaction. This ensures that only valid transactions are propagated across the network, while invalid transactions are discarded at the first node that encounters them.

Each node verifies every transaction against a long checklist of criteria:

* The transaction’s syntax and data structure must be correct.
* Neither lists of inputs or outputs are empty.
* The transaction size is less than the maximum allowed size for a block excluding witness data, as shown in [*tx\_check.cpp*](https://bit.ly/3t623VL).
* Each output value, as well as the total, must be within the allowed range of values (less than 21m coins, more than the *dust* threshold).
* None of the inputs have hash=0, N=–1 (coinbase transactions should not be relayed).
* nLocktime is equal to INT\_MAX, or nLocktime and nSequence values are satisfied according to MedianTimePast.
* The transaction size in bytes is greater than or equal to 82.
* The number of signature operations (SIGOPS) contained in the transaction is less than the signature operation limit.
* The unlocking script (scriptSig) can only push numbers on the stack, and the locking script (scriptPubkey) must match IsStandard forms (this rejects "nonstandard" transactions).
* A matching transaction in the pool, or in a block in the main branch, must exist.
* For each input, if the referenced output exists in any other transaction in the pool, the transaction must be rejected.
* For each input, look in the main branch and the transaction pool to find its parent transaction. If the parent transaction is missing for any input, this will be an orphan transaction. Add to the orphan transactions pool, if a matching transaction is not already in the pool.
* For each input, if its parent transaction is a coinbase transaction, it must have at least COINBASE\_MATURITY (100) confirmations.
* For each input, the referenced output must exist and cannot already be spent.
* Using the parent transactions to get input values, check that each input value, as well as the sum, are in the allowed range of values (less than 21m coins, more than 0).
* Reject if the sum of input values is less than sum of output values.
* Reject if transaction fee would be too low (minRelayTxFee) to get into an empty block.
* The unlocking scripts for each input must validate against the corresponding output locking scripts.

These conditions can be seen in detail in the functions AcceptToMemoryPool, CheckTransaction, and CheckInputs in Bitcoin Core. Note that the conditions change over time, to address new types of denial-of-service attacks or sometimes to relax the rules so as to include more types of transactions.

By independently verifying each transaction as it is received and before propagating it, every node builds a pool of valid (but unconfirmed) transactions known as the *transaction pool*, *memory pool*, or *mempool*.

=== Mining Nodes

Some of the nodes on the Bitcoin network are specialized nodes called *miners*. In [Introduction](#ch01_intro_what_is_bitcoin) we introduced Jing, a computer engineering student in Shanghai, China, who is a bitcoin miner. Jing earns bitcoin by running a "mining rig," which is a specialized computer-hardware system designed to mine bitcoin. Jing’s specialized mining hardware is connected to a server running a full Bitcoin node. Unlike Jing, some miners mine without a full node, as we will see in [simpara\_title](#mining_pools). Like every other full node, Jing’s node receives and propagates unconfirmed transactions on the Bitcoin network. Jing’s node, however, also aggregates these transactions into new blocks.

Jing’s node is listening for new blocks, propagated on the Bitcoin network, as do all nodes. However, the arrival of a new block has special significance for a mining node. The competition among miners effectively ends with the propagation of a new block that acts as an announcement of a winner. To miners, receiving a valid new block means someone else won the competition and they lost. However, the end of one round of a competition is also the beginning of the next round. The new block is not just a checkered flag, marking the end of the race; it is also the starting pistol in the race for the next block.

=== Aggregating Transactions into Blocks

After validating transactions, a Bitcoin node will add them to the *memory pool*, or *transaction pool*, where transactions await until they can be included (mined) into a block. Jing’s node collects, validates, and relays new transactions just like any other node. Unlike other nodes, however, Jing’s node will then aggregate these transactions into a *candidate block*.

Let’s follow the blocks that were created during the time Alice bought a cup of coffee from Bob’s Cafe (see [simpara\_title](#cup_of_coffee)). Alice’s transaction was included in block 277,316. For the purpose of demonstrating the concepts in this chapter, let’s assume that block was mined by Jing’s mining system and follow Alice’s transaction as it becomes part of this new block.

Jing’s mining node maintains a local copy of the blockchain. By the time Alice buys the cup of coffee, Jing’s node has assembled a chain up to block 277,314. Jing’s node is listening for transactions, trying to mine a new block and also listening for blocks discovered by other nodes. As Jing’s node is mining, it receives block 277,315 through the Bitcoin network. The arrival of this block signifies the end of the competition for block 277,315 and the beginning of the competition to create block 277,316.

During the previous 10 minutes, while Jing’s node was searching for a solution to block 277,315, it was also collecting transactions in preparation for the next block. By now it has collected a few hundred transactions in the memory pool. Upon receiving block 277,315 and validating it, Jing’s node will also compare it against all the transactions in the memory pool and remove any that were included in block 277,315. Whatever transactions remain in the memory pool are unconfirmed and are waiting to be recorded in a new block.

Jing’s node immediately constructs a new empty block, a candidate for block 277,316. This block is called a *candidate block* because it is not yet a valid block, as it does not contain a valid Proof-of-Work. The block becomes valid only if the miner succeeds in finding a solution to the Proof-of-Work algorithm.

When Jing’s node aggregates all the transactions from the memory pool, the new candidate block has 418 transactions with total transaction fees of 0.09094928 bitcoin. You can see this block in the blockchain using the Bitcoin Core client command-line interface, as shown in [example\_title](#block277316).

$ bitcoin-cli getblockhash 277316  
  
0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4  
  
$ bitcoin-cli getblock 0000000000000001b6b9a13b095e96db41c4a928b97ef2d9\  
44a9b31b2cc7bdc4

{  
 "hash" : "0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",  
 "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" : "000000000000000000000000000000000000000000000934695e92aaf53afa1a",  
 "previousblockhash" : "0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569"  
}

==== The Coinbase Transaction

The first transaction in any block is a special transaction, called a *coinbase transaction*. This transaction is constructed by Jing’s node and contains his *reward* for the mining effort.

When block 277,316 was mined, the reward was 25 bitcoin per block. Since then, two "halving" periods have elapsed. The block reward changed to 12.5 bitcoin in July 2016 and to 6.25 bitcoin in May 2020.

Jing’s node creates the coinbase transaction as a payment to his own wallet: "Pay Jing’s address 25.09094928 bitcoin." The total amount of reward that Jing collects for mining a block is the sum of the coinbase reward (25 new bitcoin) and the transaction fees (0.09094928) from all the transactions included in the block as shown in [example\_title](#generation_tx_example).

$ bitcoin-cli getrawtransaction d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1

{  
 "hex" : "01000000010000000000000000000000000000000000000000000000000000000000000000ffffffff0f03443b0403858402062f503253482fffffffff0110c08d9500000000232102aa970c592640d19de03ff6f329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",  
 "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"  
 ]  
 }  
 }  
 ]  
}

Unlike regular transactions, the coinbase transaction does not consume (spend) UTXO as inputs. Instead, it has only one input, called the *coinbase*, which creates bitcoin from nothing. The coinbase transaction has one output, payable to the miner’s own Bitcoin address. The output of the coinbase transaction sends the value of 25.09094928 bitcoin to the miner’s Bitcoin address; in this case it is 1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N.

==== Coinbase Reward and Fees

To construct the coinbase transaction, Jing’s node first calculates the total amount of transaction fees by adding all the inputs and outputs of the 418 transactions that were added to the block. The fees are calculated as:

Total Fees = Sum(Inputs) – Sum(Outputs)

In block 277,316, the total transaction fees are 0.09094928 bitcoin.

Next, Jing’s node calculates the correct reward for the new block. The reward is calculated based on the block height, starting at 50 bitcoin per block and reduced by half every 210,000 blocks. Because this block is at height 277,316, the correct reward is 25 bitcoin.

The calculation can be seen in function GetBlockSubsidy in the Bitcoin Core client, as shown in [example\_title](#getblocksubsidy_source).

CAmount GetBlockSubsidy(int nHeight, const Consensus::Params& consensusParams)  
{  
 int halvings = nHeight / consensusParams.nSubsidyHalvingInterval;  
 // Force block reward to zero when right shift is undefined.  
 if (halvings >= 64)  
 return 0;  
  
 CAmount nSubsidy = 50 \* COIN;  
 // Subsidy is cut in half every 210,000 blocks which will occur approximately every 4 years.  
 nSubsidy >>= halvings;  
 return nSubsidy;  
}

The initial subsidy is calculated in satoshis by multiplying 50 with the COIN constant (100,000,000 satoshis). This sets the initial reward (nSubsidy) at 5 billion satoshis.

Next, the function calculates the number of halvings that have occurred by dividing the current block height by the halving interval (SubsidyHalvingInterval). In the case of block 277,316, with a halving interval every 210,000 blocks, the result is 1 halving.

The maximum number of halvings allowed is 64, so the code imposes a zero reward (returns only the fees) if the 64 halvings is exceeded.

Next, the function uses the binary-right-shift operator to divide the reward (nSubsidy) by two for each round of halving. In the case of block 277,316, this would binary-right-shift the reward of 5 billion satoshis once (one halving) and result in 2.5 billion satoshis, or 25 bitcoin. The binary-right-shift operator is used because it is more efficient than multiple repeated divisions. To avoid a potential bug, the shift operation is skipped after 63 halvings, and the subsidy is set to 0.

Finally, the coinbase reward (nSubsidy) is added to the transaction fees (nFees), and the sum is returned.

If Jing’s mining node writes the coinbase transaction, what stops Jing from "rewarding" himself 100 or 1000 bitcoin? The answer is that an incorrect reward would result in the block being deemed invalid by everyone else, wasting Jing’s electricity used for Proof-of-Work. Jing only gets to spend the reward if the block is accepted by everyone.

==== Structure of the Coinbase Transaction

With these calculations, Jing’s node then constructs the coinbase transaction to pay himself 25.09094928 bitcoin.

As you can see in [example\_title](#generation_tx_example), the coinbase transaction has a special format. Instead of a transaction input specifying a previous UTXO to spend, it has a "coinbase" input. We examined transaction inputs in [table\_title](#tx_in_structure). Let’s compare a regular transaction input with a coinbase transaction input. [table\_title](#table_8-1) shows the structure of a regular transaction input, while [table\_title](#table_8-2) shows the structure of the coinbase transaction’s input.

The structure of a "normal" transaction input

| Size | Field | Description |
| --- | --- | --- |
| 32 bytes | Transaction Hash | Pointer to the transaction containing the UTXO to be spent |
| 4 bytes | Output Index | The index number of the UTXO to be spent, first one is 0 |
| 1–9 bytes (VarInt) | Unlocking-Script Size | Unlocking-Script length in bytes, to follow |
| Variable | Unlocking-Script | A script that fulfills the conditions of the UTXO locking script |
| 4 bytes | Sequence Number | Usually set to 0xFFFFFFFF to opt out of BIP 125 and BIP 68 |

The structure of a coinbase transaction input

| Size | Field | Description |
| --- | --- | --- |
| 32 bytes | Transaction Hash | All bits are zero: Not a transaction hash reference |
| 4 bytes | Output Index | All bits are ones: 0xFFFFFFFF |
| 1–9 bytes (VarInt) | Coinbase Data Size | Length of the coinbase data, from 2 to 100 bytes |
| Variable | Coinbase Data | Arbitrary data used for extra nonce and mining tags. In v2 blocks; must begin with block height |
| 4 bytes | Sequence Number | Set to 0xFFFFFFFF |

In a coinbase transaction, the first two fields are set to values that do not represent a UTXO reference. Instead of a "transaction hash," the first field is filled with 32 bytes all set to zero. The "output index" is filled with 4 bytes all set to 0xFF (255 decimal). The "Unlocking Script" (scriptSig) is replaced by coinbase data, a data field used by the miners, as we will see next.

==== Coinbase Data

Coinbase transactions do not have an unlocking script (aka, scriptSig) field. Instead, this field is replaced by coinbase data, which must be between 2 and 100 bytes. Except for the first few bytes, the rest of the coinbase data can be used by miners in any way they want; it is arbitrary data.

In the genesis block, for example, Satoshi Nakamoto added the text "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" in the coinbase data, using it as a proof of the date and to convey a message. Currently, miners use the coinbase data to include extra nonce values and strings identifying the mining pool.

The first few bytes of the coinbase used to be arbitrary, but that is no longer the case. As per BIP-34, version-2 blocks (blocks with the version field set to 2) must contain the block height index as a script "push" operation in the beginning of the coinbase field.

In block 277,316 we see that the coinbase (see [example\_title](#generation_tx_example)), which is in the unlocking script or scriptSig field of the transaction input, contains the hexadecimal value 03443b0403858402062f503253482f. Let’s decode this value.

The first byte, 03, instructs the script execution engine to push the next three bytes onto the script stack (see [table\_title](#tx_script_ops_table_pushdata)). The next three bytes, 0x443b04, are the block height encoded in little-endian format (backward, least-significant byte first). Reverse the order of the bytes and the result is 0x043b44, which is 277,316 in decimal.

The next few hexadecimal digits (0385840206) are used to encode an extra *nonce* (see [simpara\_title](#extra_nonce)), or random value, used to find a suitable Proof-of-Work solution.

The final part of the coinbase data (2f503253482f) is the ASCII-encoded string /P2SH/, which indicates that the mining node that mined this block provides support for the P2SH improvement defined in BIP-16. The introduction of the P2SH capability required signaling by miners to endorse either BIP-16 or BIP-17. Those endorsing the BIP-16 implementation were to include the string /P2SH/ in their coinbase data. Those endorsing the BIP-17 implementation of P2SH were to include the string p2sh/CHV in their coinbase data. Finally, the BIP-16 was elected as the winner, and many miners continued including the string /P2SH/ in their coinbase to indicate that they provide support for this feature.

[example\_title](#satoshi_words) uses the libbitcoin library introduced in [simpara\_title](#alt_libraries) to extract the coinbase data from the genesis block, displaying Satoshi’s message. Note that the libbitcoin library contains a static copy of the genesis block, so the example code can retrieve the genesis block directly from the library.

/\*  
 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;  
}

We compile the code with the GNU C++ compiler and run the resulting executable, as shown in [example\_title](#satoshi_words_run).

# Compile the code  
$ g++ -o satoshi-words satoshi-words.cpp $(pkg-config --cflags --libs libbitcoin)  
# Run the executable  
$ ./satoshi-words  
^D��<GS>^A^DEThe Times 03/Jan/2009 Chancellor on brink of second bailout for banks

=== Constructing the Block Header

To construct the block header, the mining node needs to fill in six fields, as listed in [table\_title](#block_header_structure_ch10).

The structure of the block header

| Size | Field | Description |
| --- | --- | --- |
| 4 bytes | Version | A version number to track software/protocol upgrades |
| 32 bytes | Previous Block Hash | A reference to the hash of the previous (parent) block in the chain |
| 32 bytes | Merkle Root | A hash of the root of the merkle tree of this block’s transactions |
| 4 bytes | Timestamp | The approximate creation time of this block (seconds from Unix Epoch) |
| 4 bytes | Target | The Proof-of-Work algorithm target for this block |
| 4 bytes | Nonce | A counter used for the Proof-of-Work algorithm |

At the time that block 277,316 was mined, the version number describing the block structure is version 2, which is encoded in little-endian format in 4 bytes as 0x02000000.

Next, the mining node needs to add the "Previous Block Hash" (also known as prevhash). That is the hash of the block header of block 277,315, the previous block received from the network, which Jing’s node has accepted and selected as the *parent* of the candidate block 277,316. The block header hash for block 277,315 is:

0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569

By selecting the specific *parent* block, indicated by the Previous Block Hash field in the candidate block header, Jing is committing his mining power to extending the chain that ends in that specific block. In essence, this is how Jing "votes" with his mining power for the longest-difficulty valid chain.

The next step is to summarize all the transactions with a merkle tree, in order to add the merkle root to the block header. The coinbase transaction is listed as the first transaction in the block. Then, 418 more transactions are added after it, for a total of 419 transactions in the block. As we saw in the [simpara\_title](#merkle_trees), there must be an even number of "leaf" nodes in the tree, so the last transaction is duplicated, creating 420 nodes, each containing the hash of one transaction. The transaction hashes are then combined, in pairs, creating each level of the tree, until all the transactions are summarized into one node at the "root" of the tree. The root of the merkle tree summarizes all the transactions into a single 32-byte value, which you can see listed as "merkle root" in [example\_title](#block277316), and here:

c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e

Jing’s mining node will then add a 4-byte timestamp, encoded as a Unix "epoch" timestamp, which is based on the number of seconds elapsed since midnight UTC, Thursday, January 1, 1970. The time 1388185914 is equal to Friday, December 27, 2013, 23:11:54 UTC.

Jing’s node then fills in the target, which defines the required Proof-of-Work to make this a valid block. The target is stored in the block as a "target bits" metric, which is a mantissa-exponent encoding of the target. The encoding has a 1-byte exponent, followed by a 3-byte mantissa (coefficient). In block 277,316, for example, the target bits value is 0x1903a30c. The first part 0x19 is a hexadecimal exponent, while the next part, 0x03a30c, is the coefficient. The concept of a target is explained in [simpara\_title](#target) and the "target bits" representation is explained in [simpara\_title](#target_bits).

The final field is the nonce, which is initialized to zero.

With all the other fields filled, the block header is now complete and the process of mining can begin. The goal is now to find a value for the nonce that results in a block header hash that is equal to or less than the target. The mining node will need to test billions or trillions of nonce values before a nonce is found that satisfies the requirement.

=== Mining the Block

Now that a candidate block has been constructed by Jing’s node, it is time for Jing’s hardware mining rig to "mine" the block, to find a solution to the Proof-of-Work algorithm that makes the block valid. Throughout this book we have studied cryptographic hash functions as used in various aspects of the Bitcoin system. The hash function SHA256 is the function used in bitcoin’s mining process.

In the simplest terms, mining is the process of hashing the block header repeatedly, changing one parameter, until the resulting hash matches a specific target. The hash function’s result cannot be determined in advance, nor can a pattern be created that will produce a specific hash value. This feature of hash functions means that the only way to produce a hash result matching a specific target is to try again and again, randomly modifying the input until the desired hash result appears by chance.

==== Proof-of-Work Algorithm

A hash algorithm takes an arbitrary-length data input and produces a fixed-length deterministic result, a digital fingerprint of the input. For any specific input, the resulting hash will always be the same and can be easily calculated and verified by anyone implementing the same hash algorithm. The key characteristic of a cryptographic hash algorithm is that it is computationally infeasible to find two different inputs that produce the same fingerprint (known as a *collision*). As a corollary, it is also virtually impossible to select an input in such a way as to produce a desired fingerprint, other than trying random inputs.

With SHA256, the output is always 256 bits long, regardless of the size of the input. In [example\_title](#sha256_example1), we will use the Python interpreter to calculate the SHA256 hash of the phrase, "I am Satoshi Nakamoto".

$ python

Python 3.7.3  
>>> import hashlib  
>>> hashlib.sha256(b"I am Satoshi Nakamoto").hexdigest()  
'5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e'

[example\_title](#sha256_example1) shows the result of calculating the hash of "I am Satoshi Nakamoto": 5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e. This 256-bit number is the *hash* or *digest* of the phrase and depends on every part of the phrase. Adding a single letter, punctuation mark, or any other character will produce a different hash.

Now, if we change the phrase, we should expect to see completely different hashes. Let’s try that by adding a number to the end of our phrase, using the simple Python scripting in [example\_title](#sha256_example_generator).

# example of iterating a nonce in a hashing algorithm's input  
  
import hashlib  
   
text = "I am Satoshi Nakamoto"  
  
# iterate nonce from 0 to 19  
for nonce in xrange(20):   
   
 # add the nonce to the end of the text  
 input = text + str(nonce)   
   
 # calculate the SHA-256 hash of the input (text+nonce)  
 hash = hashlib.sha256(input).hexdigest()   
   
 # show the input and hash result  
 print input, '=>', hash

Running this will produce the hashes of several phrases, made different by adding a number at the end of the text. By incrementing the number, we can get different hashes, as shown in [example\_title](#sha256_example_generator_output).

$ 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...

Each phrase produces a completely different hash result. They seem completely random, but you can reproduce the exact results in this example on any computer with Python and see the same exact hashes.

The number used as a variable in such a scenario is called a *nonce*. The nonce is used to vary the output of a cryptographic function, in this case to vary the SHA256 fingerprint of the phrase.

To make a challenge out of this algorithm, let’s set a target: find a phrase that produces a hexadecimal hash that starts with a zero. Fortunately, this isn’t difficult! [example\_title](#sha256_example_generator_output) shows that the phrase "I am Satoshi Nakamoto13" produces the hash 0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba9531041a5, which fits our criteria. It took 13 attempts to find it. In terms of probabilities, if the output of the hash function is evenly distributed we would expect to find a result with a 0 as the hexadecimal prefix once every 16 hashes (one out of 16 hexadecimal digits 0 through F). In numerical terms, that means finding a hash value that is less than 0x1000000000000000000000000000000000000000000000000000000000000000. We call this threshold the *target* and the goal is to find a hash that is numerically equal to or less than the target. If we decrease the target, the task of finding a hash that is less than the target becomes more and more difficult.

To give a simple analogy, imagine a game where players throw a pair of dice repeatedly, trying to throw equal to or less than a specified target. In the first round, the target is 11. Unless you throw double-six, you win. In the next round the target is 10. Players must throw 10 or less to win, again an easy task. Let’s say a few rounds later the target is down to 5. Now, more than half the dice throws will exceed the target and therefore be invalid. It takes exponentially more dice throws to win, the lower the target gets. Eventually, when the target is 2 (the minimum possible), only one throw out of every 36, or 2% of them, will produce a winning result.

From the perspective of an observer who knows that the target of the dice game is 2, if someone has succeeded in casting a winning throw it can be assumed that they attempted, on average, 36 throws. In other words, one can estimate the amount of work it takes to succeed from the difficulty imposed by the target. When the algorithm is based on a deterministic function such as SHA256, the input itself constitutes *proof* that a certain amount of *work* was done to produce a result equal to or below the target. Hence, *Proof-of-Work*.

Even though each attempt produces a random outcome, the probability of any possible outcome can be calculated in advance. Therefore, an outcome of specified difficulty constitutes proof of a specific amount of work.

In [example\_title](#sha256_example_generator_output), the winning "nonce" is 13 and this result can be confirmed by anyone independently. Anyone can add the number 13 as a suffix to the phrase "I am Satoshi Nakamoto" and compute the hash, verifying that it is less than the target. The successful result is also Proof-of-Work, because it proves we did the work to find that nonce. While it only takes one hash computation to verify, it took us 13 hash computations to find a nonce that worked. If we had a lower target (higher difficulty) it would take many more hash computations to find a suitable nonce, but only one hash computation for anyone to verify. Furthermore, by knowing the target, anyone can estimate the difficulty using statistics and therefore know how much work was needed to find such a nonce.

The Proof-of-Work must produce a hash that is *equal to or less than* the target. A higher target means it is less difficult to find a hash that is equal to or below the target. A lower target means it is more difficult to find a hash equal to or below the target. The target and difficulty are inversely related.

Bitcoin’s Proof-of-Work is very similar to the challenge shown in [example\_title](#sha256_example_generator_output). The miner constructs a candidate block filled with transactions. Next, the miner calculates the hash of this block’s header and sees if it is equal to or smaller than the current *target*. If the hash is greater than the target, the miner will modify the nonce (usually just incrementing it by one) and try again. At the current difficulty in the Bitcoin network, miners have to try quadrillions of times before finding a nonce that results in a low enough block header hash.

A very simplified Proof-of-Work algorithm is implemented in Python in [example\_title](#pow_example1).

#!/usr/bin/env python  
# example of proof-of-work algorithm  
  
import hashlib  
import time  
  
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)).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

Running this code, you can set the desired difficulty (in bits, how many of the leading bits must be zero) and see how long it takes for your computer to find a solution. In [example\_title](#pow_example_outputs), you can see how it works on an average laptop.

$ 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

As you can see, increasing the difficulty by 1 bit causes a doubling in the time it takes to find a solution. If you think of the entire 256-bit number space, each time you constrain one more bit to zero, you decrease the search space by half. In [example\_title](#pow_example_outputs), it takes 84 million hash attempts to find a nonce that produces a hash with 26 leading bits as zero. Even at a speed of more than 120,000 hashes per second, it still requires 10 minutes on a laptop to find this solution.

At the time of writing, the network is attempting to find a block whose header hash is equal to or less than:

0000000000000000029AB9000000000000000000000000000000000000000000

As you can see, there are a lot of zeros at the beginning of that target, meaning that the acceptable range of hashes is much smaller, hence it’s more difficult to find a valid hash. It will take on average more than 1.8 zeta-hashes (thousand billion billion hashes) for the network to discover the next block. That seems like an impossible task, but fortunately the network is bringing 3 exa-hashes per second (EH/sec) of processing power to bear, which will be able to find a block in about 10 minutes on average.

==== Target Representation

In [example\_title](#block277316), we saw that the block contains the target, in a notation called "target bits" or just "bits," which in block 277,316 has the value of 0x1903a30c. This notation expresses the Proof-of-Work target as a coefficient/exponent format, with the first two hexadecimal digits for the exponent and the next six hex digits as the coefficient. In this block, therefore, the exponent is 0x19 and the coefficient is 0x03a30c.

The formula to calculate the difficulty target from this representation is:

target = coefficient \* 2

(8\*(exponent–3))

Using that formula, and the difficulty bits value 0x1903a30c, we get:

target = 0x03a30c \* 2

0x08\*(0x19-0x03)

=> target = 0x03a30c \* 2

(0x08\*0x16)

=> target = 0x03a30c \* 2

0xB0

which in decimal is:

=> target = 238,348 \* 2

176

=> target =

22,829,202,948,393,929,850,749,706,076,701,368,331,072,452,018,388,575,715,328

switching back to hexadecimal:

=> target =

0x0000000000000003A30C00000000000000000000000000000000000000000000

This means that a valid block for height 277,316 is one that has a block header hash that is less than this target. In binary that number must have more than 60 leading bits set to zero. With this level of difficulty, a single miner processing 1 trillion hashes per second (1 terahash per second or 1 TH/sec) would only find a solution once every 8,496 blocks or once every 59 days, on average.

==== Retargeting to Adjust Difficulty

As we saw, the target determines the difficulty and therefore affects how long it takes to find a solution to the Proof-of-Work algorithm. This leads to the obvious questions: Why is the difficulty adjustable, who adjusts it, and how?

Bitcoin’s blocks are generated every 10 minutes, on average. This is bitcoin’s heartbeat and underpins the frequency of currency issuance and the speed of transaction settlement. It has to remain constant not just over the short term, but over a period of many decades. Over this time, it is expected that computer power will continue to increase at a rapid pace. Furthermore, the number of participants in mining and the computers they use will also constantly change. To keep the block generation time at 10 minutes, the difficulty of mining must be adjusted to account for these changes. In fact, the Proof-of-Work target is a dynamic parameter that is periodically adjusted to meet a 10-minute block interval goal. In simple terms, the target is set so that the current mining power will result in a 10-minute block interval.

How, then, is such an adjustment made in a completely decentralized network? Retargeting occurs automatically and on every node independently. Every 2,016 blocks, all nodes retarget the Proof-of-Work. The equation for retargeting measures the time it took to find the last 2,016 blocks and compares that to the expected time of 20,160 minutes (2,016 blocks times the desired 10-minute block interval). The ratio between the actual timespan and desired timespan is calculated and a proportionate adjustment (up or down) is made to the target. In simple terms: If the network is finding blocks faster than every 10 minutes, the difficulty increases (target decreases). If block discovery is slower than expected, the difficulty decreases (target increases).

The equation can be summarized as:

New Target = Old Target \* (Actual Time of Last 2016 Blocks / 20160 minutes)

[example\_title](#retarget_code) shows the code used in the Bitcoin Core client.

// Limit adjustment step  
 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;  
  
 // Retarget  
 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;

While the target calibration happens every 2,016 blocks, because of an off-by-one error in the original Bitcoin Core client it is based on the total time of the previous 2,015 blocks (not 2,016 as it should be), resulting in a retargeting bias toward higher difficulty by 0.05%.

The parameters Interval (2,016 blocks) and TargetTimespan (two weeks as 1,209,600 seconds) are defined in *chainparams.cpp*.

To avoid extreme volatility in the difficulty, the retargeting adjustment must be less than a factor of four (4) per cycle. If the required target adjustment is greater than a factor of four, it will be adjusted by a factor of 4 and not more. Any further adjustment will be accomplished in the next retargeting period because the imbalance will persist through the next 2,016 blocks. Therefore, large discrepancies between hashing power and difficulty might take several 2,016 block cycles to balance out.

The difficulty of mining a bitcoin block is approximately '10 minutes of processing' for the entire network, based on the time it took to mine the previous 2,016 blocks, adjusted every 2,016 blocks. This is achieved by lowering or raising the target.

Note that the target is independent of the number of transactions or the value of transactions. This means that the amount of hashing power and therefore electricity expended to secure bitcoin is also entirely independent of the number of transactions. Bitcoin can scale up, achieve broader adoption, and remain secure without any increase in hashing power from today’s level. The increase in hashing power represents market forces as new miners enter the market to compete for the reward. As long as enough hashing power is under the control of miners acting honestly in pursuit of the reward, it is enough to prevent "takeover" attacks and, therefore, it is enough to secure bitcoin.

The difficulty of mining is closely related to the cost of electricity and the exchange rate of bitcoin vis-a-vis the currency used to pay for electricity. High-performance mining systems are about as efficient as possible with the current generation of silicon fabrication, converting electricity into hashing computation at the highest rate possible. The primary influence on the mining market is the price of one kilowatt-hour of electricity in bitcoin, because that determines the profitability of mining and therefore the incentives to enter or exit the mining market.

=== Successfully Mining the Block

As we saw earlier, Jing’s node has constructed a candidate block and prepared it for mining. Jing has several hardware mining rigs with application-specific integrated circuits, where hundreds of thousands of integrated circuits run the SHA256 algorithm in parallel at incredible speeds. Many of these specialized machines are connected to his mining node over USB or a local area network. Next, the mining node running on Jing’s desktop transmits the block header to his mining hardware, which starts testing trillions of nonces per second. Because the nonce is only 32 bits, after exhausting all the nonce possibilities (about 4 billion), the mining hardware changes the block header (adjusting the coinbase extra nonce space or timestamp) and resets the nonce counter, testing new combinations.

Almost 11 minutes after starting to mine block 277,316, one of the hardware mining machines finds a solution and sends it back to the mining node.

When inserted into the block header, the nonce 924,591,752 produces a block hash of:

0000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

which is less than the target:

0000000000000003A30C00000000000000000000000000000000000000000000

Immediately, Jing’s mining node transmits the block to all its peers. They receive, validate, and then propagate the new block. As the block ripples out across the network, each node adds it to its own copy of the blockchain, extending it to a new height of 277,316 blocks. As mining nodes receive and validate the block, they abandon their efforts to find a block at the same height and immediately start computing the next block in the chain, using Jing’s block as the "parent." By building on top of Jing’s newly discovered block, the other miners are essentially "voting" with their mining power and endorsing Jing’s block and the chain it extends.

In the next section, we’ll look at the process each node uses to validate a block and select the longest chain, creating the consensus that forms the decentralized blockchain.

=== Validating a New Block

The third step in bitcoin’s consensus mechanism is independent validation of each new block by every node on the network. As the newly solved block moves across the network, each node performs a series of tests to validate it before propagating it to its peers. This ensures that only valid blocks are propagated on the network. The independent validation also ensures that miners who act honestly get their blocks incorporated in the blockchain, thus earning the reward. Those miners who act dishonestly have their blocks rejected and not only lose the reward, but also waste the effort expended to find a Proof-of-Work solution, thus incurring the cost of electricity without compensation.

When a node receives a new block, it will validate the block by checking it against a long list of criteria that must all be met; otherwise, the block is rejected. These criteria can be seen in the Bitcoin Core client in the functions CheckBlock and CheckBlockHeader and include:

* The block data structure is syntactically valid
* The block header hash is equal to or less than the target (enforces the Proof-of-Work)
* The block timestamp is less than two hours in the future (allowing for time errors)
* The block size is within acceptable limits
* The first transaction (and only the first) is a coinbase transaction
* All transactions within the block are valid using the transaction checklist discussed in [simpara\_title](#tx_verification)

The independent validation of each new block by every node on the network ensures that the miners cannot cheat. In previous sections we saw how miners get to write a transaction that awards them the new bitcoin created within the block and claim the transaction fees. Why don’t miners write themselves a transaction for a thousand bitcoin instead of the correct reward? Because every node validates blocks according to the same rules. An invalid coinbase transaction would make the entire block invalid, which would result in the block being rejected and, therefore, that transaction would never become part of the ledger. The miners have to construct a perfect block, based on the shared rules that all nodes follow, and mine it with a correct solution to the Proof-of-Work. To do so, they expend a lot of electricity in mining, and if they cheat, all the electricity and effort is wasted. This is why independent validation is a key component of decentralized consensus.

=== Assembling and Selecting Chains of Blocks

The final step in bitcoin’s decentralized consensus mechanism is the assembly of blocks into chains and the selection of the chain with the most Proof-of-Work. Once a node has validated a new block, it will then attempt to assemble a chain by connecting the block to the existing blockchain.

Nodes maintain three sets of blocks: those connected to the main blockchain, those that form branches off the main blockchain (secondary chains), and finally, blocks that do not have a known parent in the known chains (orphans). Invalid blocks are rejected as soon as any one of the validation criteria fails and are therefore not included in any chain.

The "main chain" at any time is whichever *valid* chain of blocks has the most cumulative Proof-of-Work associated with it. Under most circumstances this is also the chain with the most blocks in it, unless there are two equal-length chains and one has more Proof-of-Work. The main chain will also have branches with blocks that are "siblings" to the blocks on the main chain. These blocks are valid but not part of the main chain. They are kept for future reference, in case one of those chains is extended to exceed the main chain in work. In the next section ([simpara\_title](#forks)), we will see how secondary chains occur as a result of an almost simultaneous mining of blocks at the same height.

When a new block is received, a node will try to slot it into the existing blockchain. The node will look at the block’s "previous block hash" field, which is the reference to the block’s parent. Then, the node will attempt to find that parent in the existing blockchain. Most of the time, the parent will be the "tip" of the main chain, meaning this new block extends the main chain. For example, the new block 277,316 has a reference to the hash of its parent block 277,315. Most nodes that receive 277,316 will already have block 277,315 as the tip of their main chain and will therefore link the new block and extend that chain.

Sometimes, as we will see in [simpara\_title](#forks), the new block extends a chain that is not the main chain. In that case, the node will attach the new block to the secondary chain it extends and then compare the work of the secondary chain to the main chain. If the secondary chain has more cumulative work than the main chain, the node will *reconverge* on the secondary chain, meaning it will select the secondary chain as its new main chain, making the old main chain a secondary chain. If the node is a miner, it will now construct a block extending this new, longer, chain.

If a valid block is received and no parent is found in the existing chains, that block is considered an "orphan." Orphan blocks are saved in the orphan block pool where they will stay until their parent is received. Once the parent is received and linked into the existing chains, the orphan can be pulled out of the orphan pool and linked to the parent, making it part of a chain. Orphan blocks usually occur when two blocks that were mined within a short time of each other are received in reverse order (child before parent).

By selecting the greatest-cumulative-work valid chain, all nodes eventually achieve network-wide consensus. Temporary discrepancies between chains are resolved eventually as more work is added, extending one of the possible chains. Mining nodes "vote" with their mining power by choosing which chain to extend by mining the next block. When they mine a new block and extend the chain, the new block itself represents their vote.

In the next section we will look at how discrepancies between competing chains (forks) are resolved by the independent selection of the greatest-cumulative-work chain.

==== Blockchain Forks

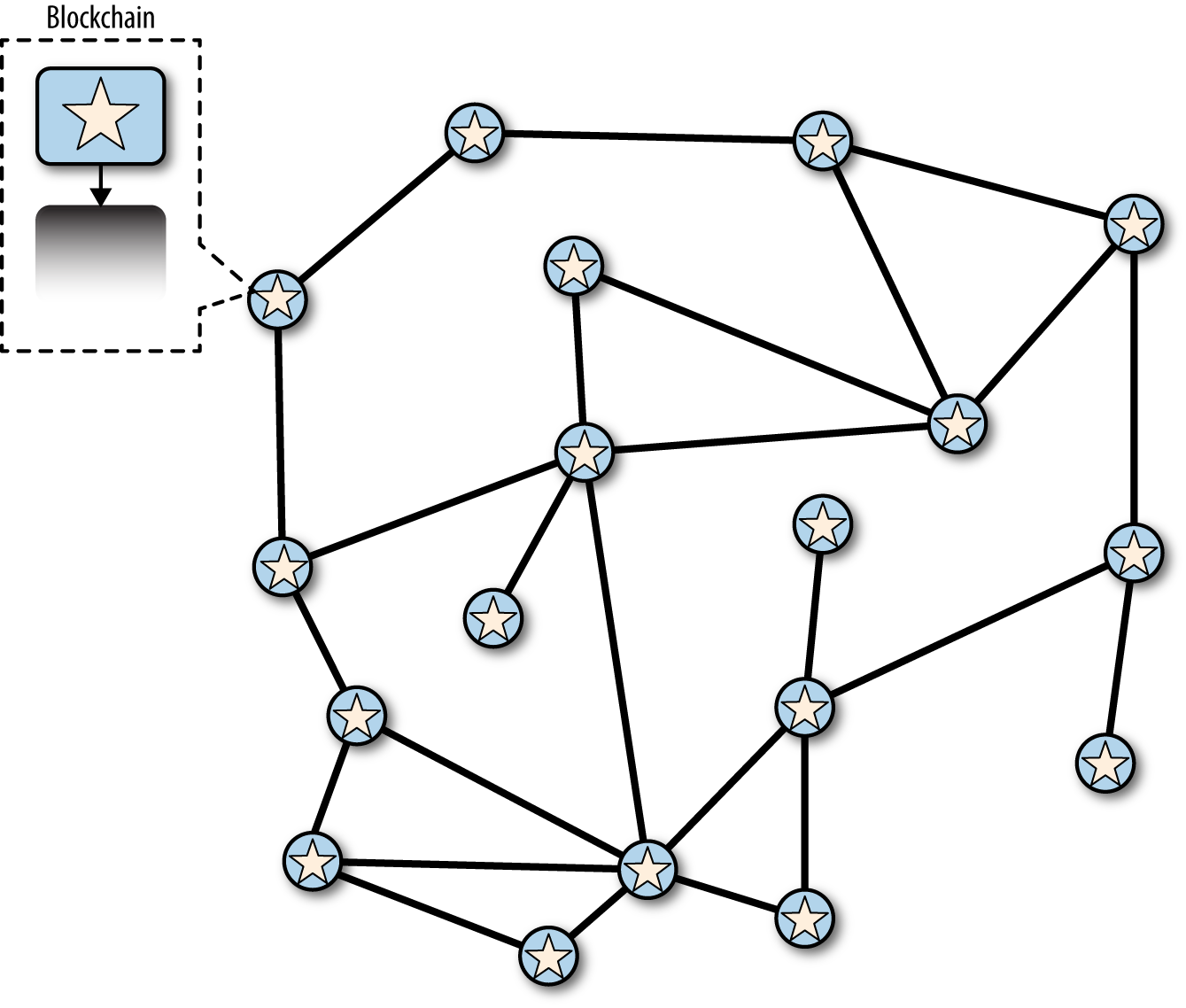
Because the blockchain is a decentralized data structure, different copies of it are not always consistent. Blocks might arrive at different nodes at different times, causing the nodes to have different perspectives of the blockchain. To resolve this, each node always selects and attempts to extend the chain of blocks that represents the most Proof-of-Work, also known as the longest chain or greatest cumulative work chain. By summing the work recorded in each block in a chain, a node can calculate the total amount of work that has been expended to create that chain. As long as all nodes select the greatest-cumulative-work chain, the global Bitcoin network eventually converges to a consistent state. Forks occur as temporary inconsistencies between versions of the blockchain, which are resolved by eventual reconvergence as more blocks are added to one of the forks.

The blockchain forks described in this section occur naturally (accidentally) as a result of transmission delays in the global network. Later in this chapter, we will also look at deliberately induced forks (hard forks and soft forks), which are used to modify the consensus rules.

In the next few diagrams, we follow the progress of a "fork" event across the network. The diagram is a simplified representation of the Bitcoin network. For illustration purposes, different blocks are shown as different shapes (star, triangle, upside-down triangle, rhombus), spreading across the network. Each node in the network is represented as a circle.

Each node has its own perspective of the global blockchain. As each node receives blocks from its neighbors, it updates its own copy of the blockchain, selecting the greatest-cumulative-work chain. For illustration purposes, each node contains a shape that represents the block that it believes is currently the tip of the main chain. So, if you see a star shape in the node, that means that the star block is the tip of the main chain, as far as that node is concerned.

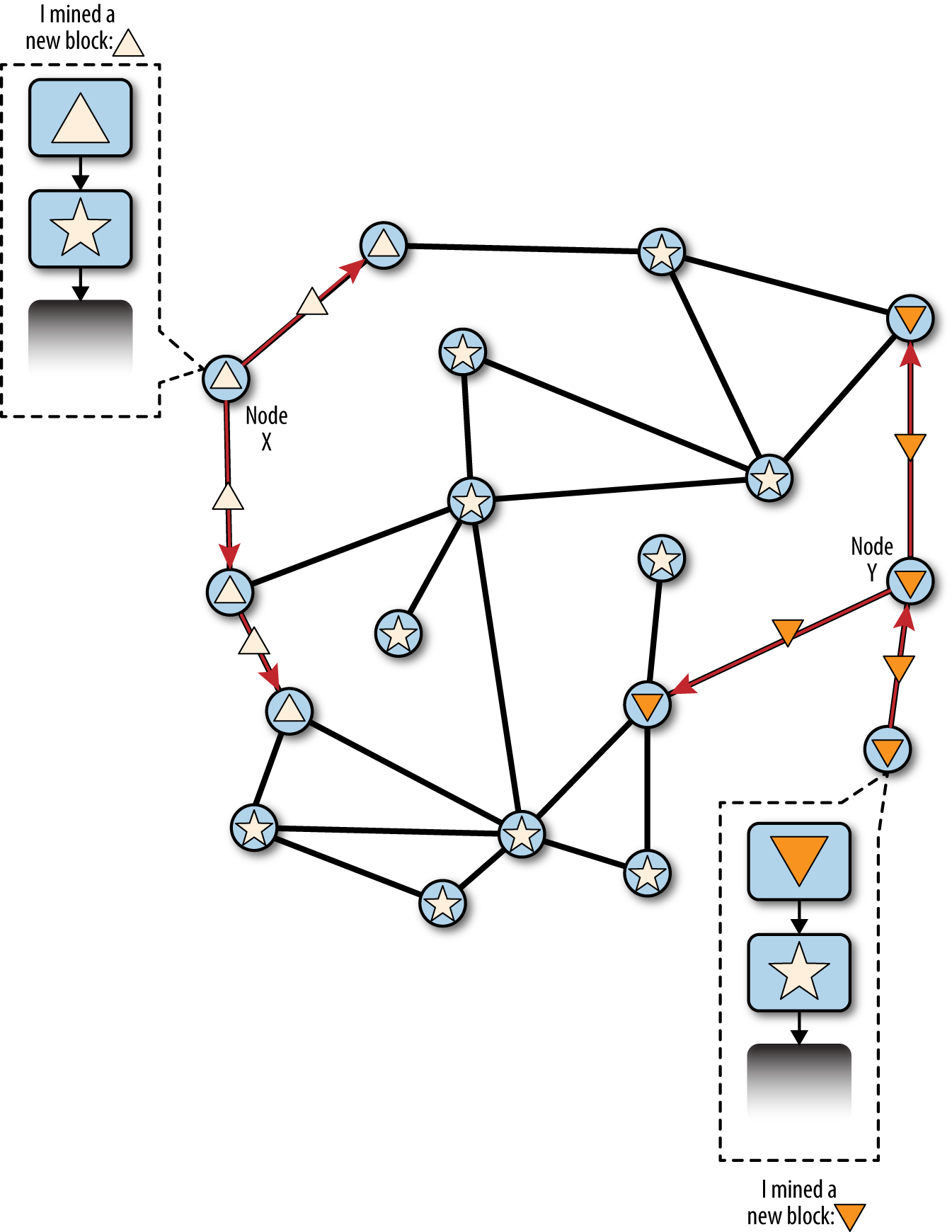
In the first diagram ([figure\_title](#fork1)), the network has a unified perspective of the blockchain, with the star block as the tip of the main chain.



Before the fork—all nodes have the same perspective

A "fork" occurs whenever there are two different valid blocks at the same block height competing to form the longest blockchain. This occurs under normal conditions whenever two miners solve the Proof-of-Work algorithm within a short period of time from each other. As both miners discover a solution for their respective candidate blocks, they immediately broadcast their own "winning" block to their immediate neighbors who begin propagating the block across the network. Each node that receives a valid block will incorporate it into its blockchain, extending the blockchain by one block. If that node later sees another valid block extending the same parent (at the same block height), it connects the second block on a secondary chain, forking its main chain. As a result, some nodes will "see" one winning block first, while other nodes will see the other winning block first, and two competing versions of the blockchain will emerge.

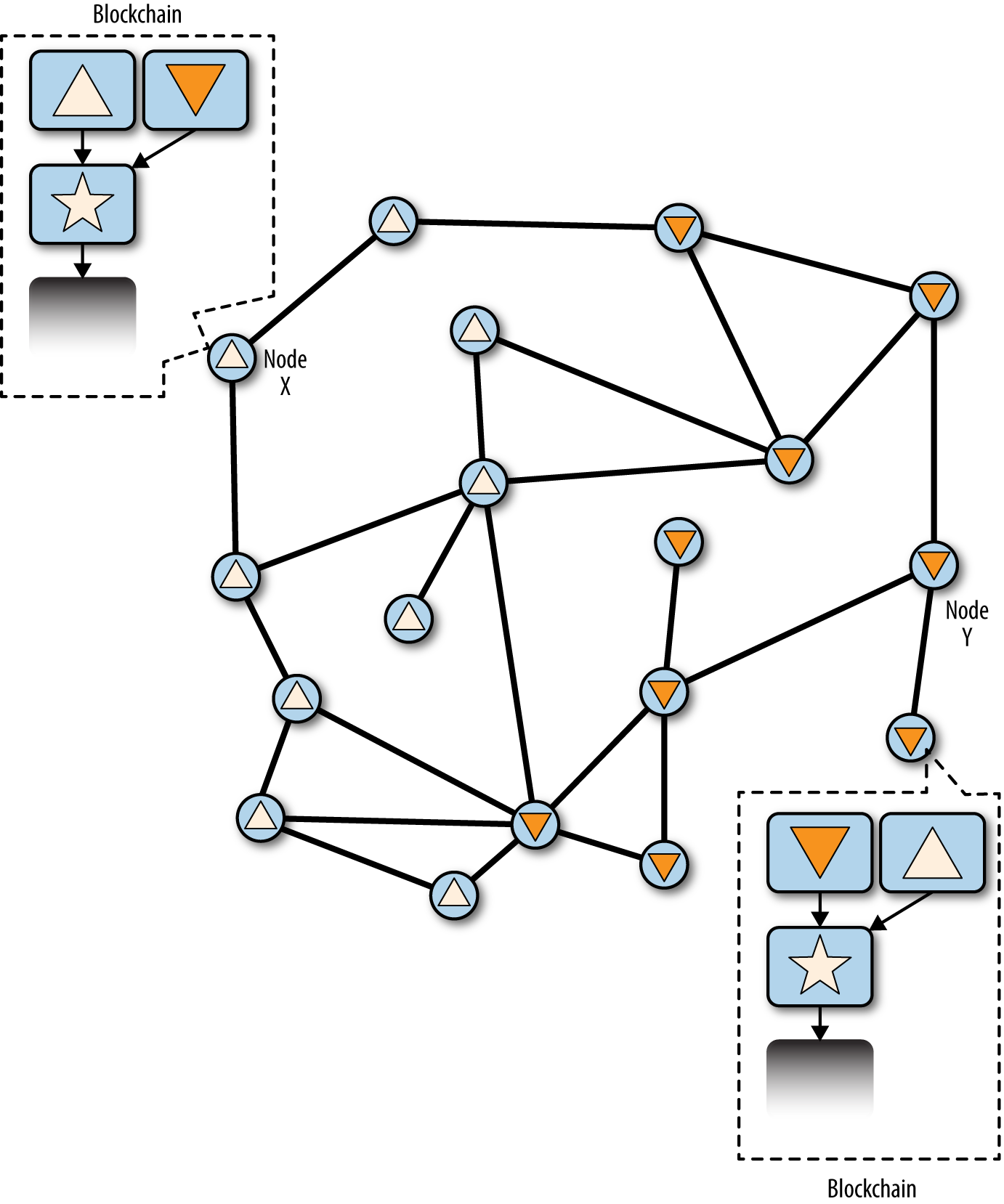
In [figure\_title](#fork2), we see two miners (Node X and Node Y) who mine two different blocks almost simultaneously. Both of these blocks are children of the star block, and extend the chain by building on top of the star block. To help us track it, one is visualized as a triangle block originating from Node X, and the other is shown as an upside-down triangle block originating from Node Y.



Visualization of a blockchain fork event: two blocks found simultaneously

Let’s assume, for example, that the miner Node X finds a Proof-of-Work solution for a block "triangle" that extends the blockchain, building on top of the parent block "star." Almost simultaneously, the miner Node Y who was also extending the chain from block "star" finds a solution for block "upside-down triangle," his candidate block. Now, there are two possible blocks; one we call "triangle," originating in Node X; and one we call "upside-down triangle," originating in Node Y. Both blocks were successfully mined, both blocks are valid (contain a valid solution to the Proof-of-Work), and both blocks extend the same parent (block "star"). Both blocks likely contain most of the same transactions, with only perhaps a few differences in the order of transactions.

As the two blocks propagate, some nodes receive block "triangle" first and some receive block "upside-down triangle" first. As shown in [figure\_title](#fork3), the network splits into two different perspectives of the blockchain; one side topped with the triangle block, the other with the upside-down-triangle block.



Visualization of a blockchain fork event: two blocks propagate, splitting the network

In [figure\_title](#fork3), the miner Node X mined (created) the triangle block and extended the star chain with it. Therefore, Node X initially considers the chain with "triangle" block as the main chain. Later, Node X also received the "upside-down triangle" block that was mined by Node Y. Since it was received second, it is assumed to have "lost" the race. Yet, the "upside-down triangle" block is not discarded. It is linked to the "star" block parent and forms a secondary chain. While Node X assumes its main chain is the winning chain, it keeps the "losing" chain so that it has the information needed to reconverge if the "losing" chain ends up "winning."

On the other side of the network, the miner Node Y constructs a blockchain based on its own perspective of the sequence of events. The miner Node Y mined (created) the "upside-down triangle" and initially considers that chain as the main chain (the "winner" chain). When it later received the "triangle" block that was mined by Node X, it connected it to the "star" block parent as a secondary chain.

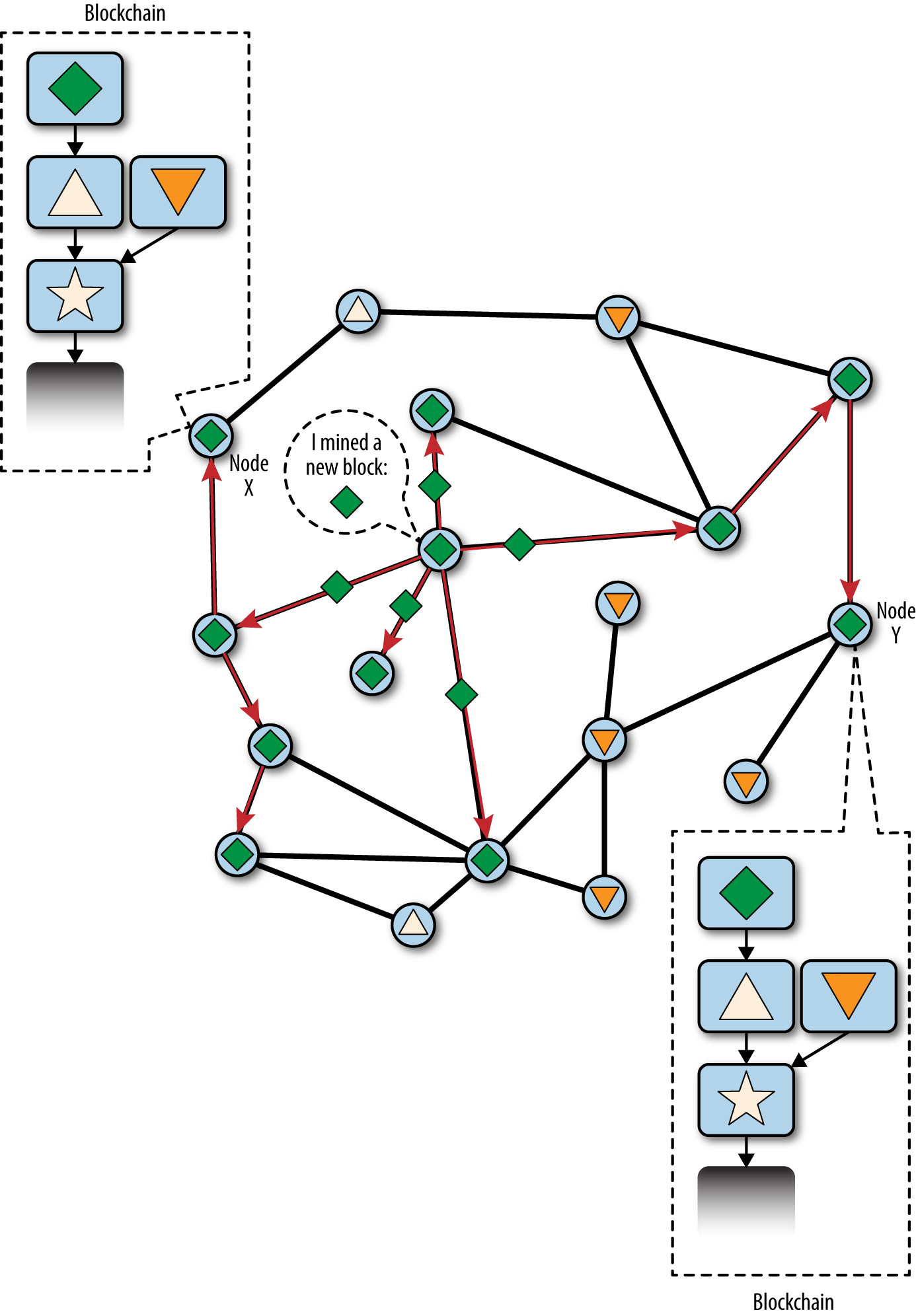
Neither side is "correct," or "incorrect." Both are valid perspectives of the blockchain. Only in hindsight will one prevail, based on how these two competing chains are extended by additional work.

Each mining node whose perspective resembles Node X will immediately begin mining a candidate block that extends the chain with "triangle" as its tip. By linking "triangle" as the parent of their candidate blocks, they are voting with their hashing power. Their vote supports the chain that they have elected as the main chain.

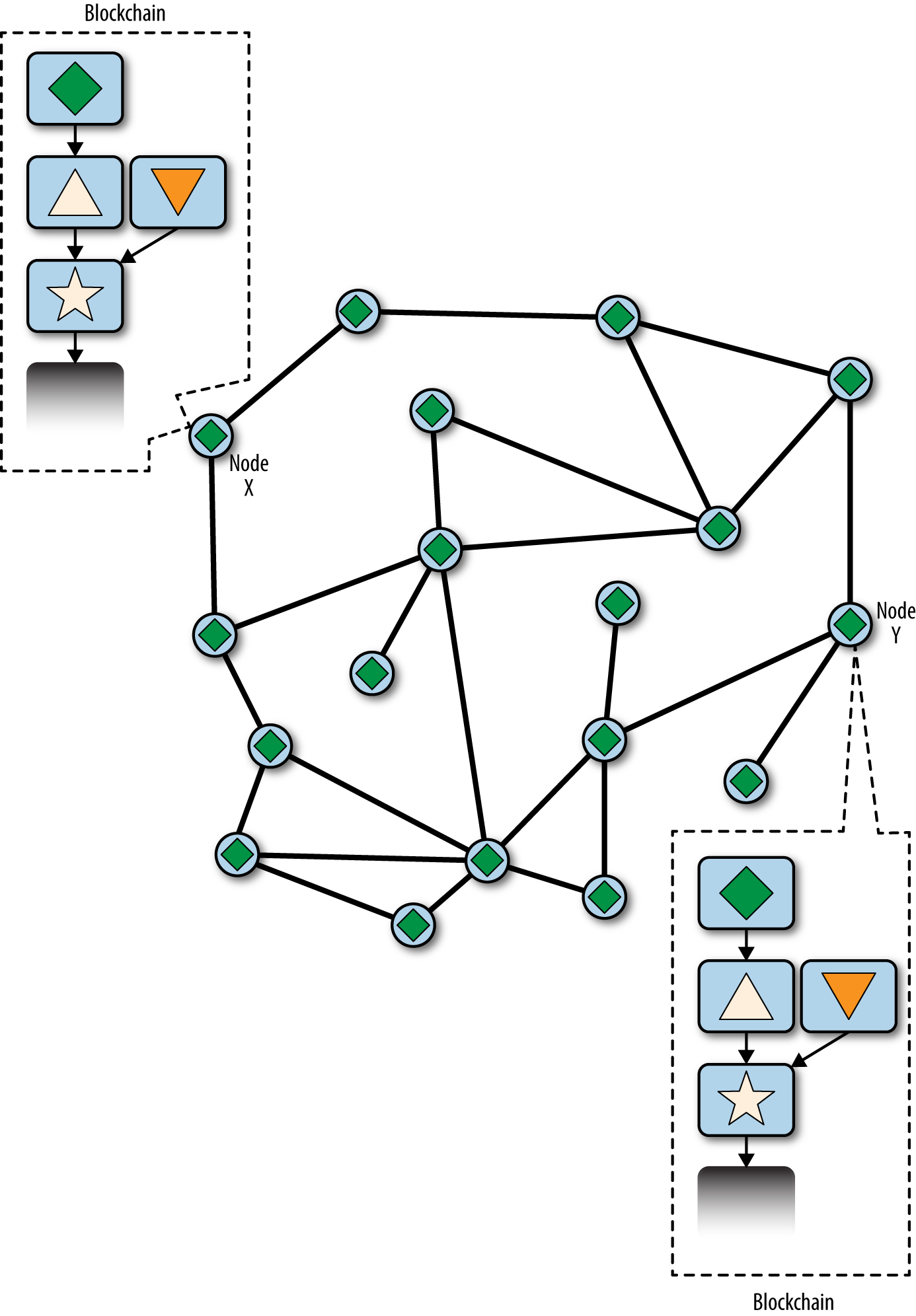
Any mining node whose perspective resembles Node Y will start building a candidate block with "upside-down triangle" as its parent, extending the chain that they believe is the main chain. And so, the race begins again.

Forks are almost always resolved within one block time (10 minutes on average). While part of the network’s hashing power is dedicated to building on top of "triangle" as the parent, another part of the hashing power is focused on building on top of "upside-down triangle." Even if the hashing power is almost evenly split, it is likely that one set of miners will find a solution and propagate it before the other set of miners have found any solutions. Let’s say, for example, that the miners building on top of "triangle" find a new block "rhombus" that extends the chain (e.g., star-triangle-rhombus). They immediately propagate this new block and the entire network sees it as a valid solution as shown in [figure\_title](#fork4).

All nodes that had chosen "triangle" as the winner in the previous round will simply extend the chain one more block. The nodes that chose "upside-down triangle" as the winner, however, will now see two chains: star-triangle-rhombus and star-upside-down-triangle. The chain star-triangle-rhombus is now longer (more cumulative work) than the other chain. As a result, those nodes will set the chain star-triangle-rhombus as the main chain and change the star-upside-down-triangle chain to a secondary chain, as shown in [figure\_title](#fork5). This is a chain reconvergence, because those nodes are forced to revise their view of the blockchain to incorporate the new evidence of a longer chain. Any miners working on extending the chain star-upside-down-triangle will now stop that work because their candidate block is now considered a child of a stale block, as its parent "upside-down-triangle" is no longer on the longest chain. Since the upside-down-triangle block is now obsolete, the miner Node Y (which mined this block) will not be able to spend the mining reward for this block, even though this block was valid and was successfully mined. The transactions within "upside-down-triangle" that are not within "triangle" are re-inserted in the mempool for inclusion in the next block to become a part of the main chain. The entire network reconverges on a single blockchain star-triangle-rhombus, with "rhombus" as the last block in the chain. All miners immediately start working on candidate blocks that reference "rhombus" as their parent to extend the star-triangle-rhombus chain.



Visualization of a blockchain fork event: a new block extends one fork, reconverging the network. Both Node X and Node Y now consider "upside-down-triangle" block as a stale block.



Visualization of a blockchain fork event: the network reconverges on a new longest chain

It is theoretically possible for a fork to extend to two blocks, if two blocks are found almost simultaneously by miners on opposite "sides" of a previous fork. However, the chance of that happening is very low. Whereas a one-block fork might occur every day, a two-block fork occurs at most once every few weeks.

Bitcoin’s block interval of 10 minutes is a design compromise between fast confirmation times (settlement of transactions) and the probability of a fork. A faster block time would make transactions clear faster but lead to more frequent blockchain forks, whereas a slower block time would decrease the number of forks but make settlement slower.

=== Mining and the Hashing Race

Bitcoin mining is an extremely competitive industry. The hashing power has increased exponentially every year of bitcoin’s existence. Some years the growth has reflected a complete change of technology, such as in 2010 and 2011 when many miners switched from using CPU mining to GPU mining and field programmable gate array (FPGA) mining. In 2013 the introduction of ASIC mining lead to another giant leap in mining power, by placing the SHA256 function directly on silicon chips specialized for the purpose of mining. The first such chips could deliver more mining power in a single box than the entire Bitcoin network in 2010.

The following list shows the total hashing power of the Bitcoin network in terahashes/sec (TH/sec), since its inception in 2009 (source: Blockchain.com):

2009

0.000004 – 0.00001 TH/sec (2.40× growth)

2010

0.00001 – 0.14 TH/sec (14,247× growth)

2011

0.14 – 9.49 TH/sec (63.92× growth)

2012

9.49 – 22 TH/sec (2.32× growth)

2013

22.04 – 15,942 TH/sec (723.32× growth)

2014

15,942 – 306,333 TH/sec (19.21× growth)

2015

306,333 – 881,232 TH/sec (2.87× growth)

2016

881,232 – 2,807,540 TH/sec (3.18× growth)

2017

2,807,540 – 18,206,558 TH/sec (6.48× growth)

2018

18,206,558 – 41,801,528 TH/sec (2.29× growth)

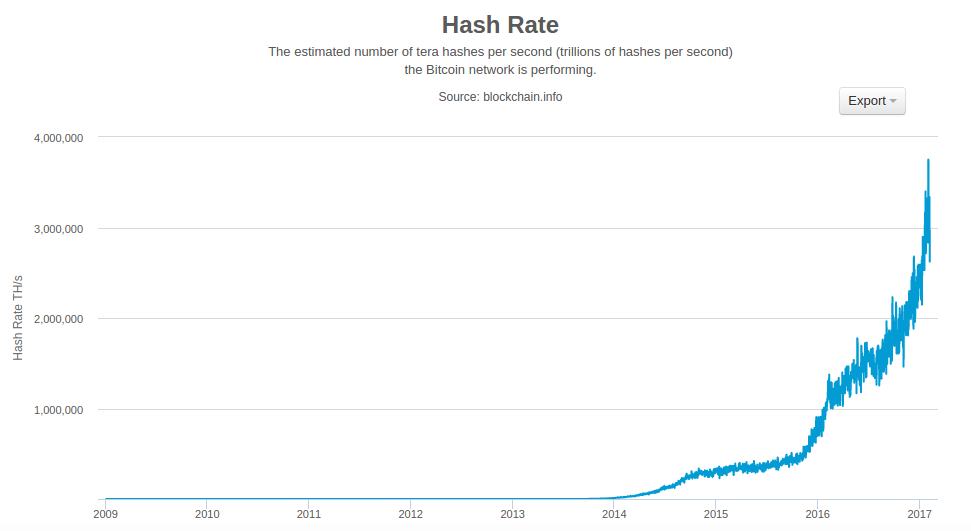
2019

41,801,528 – 109,757,127 TH/sec (2.62× growth)

2020

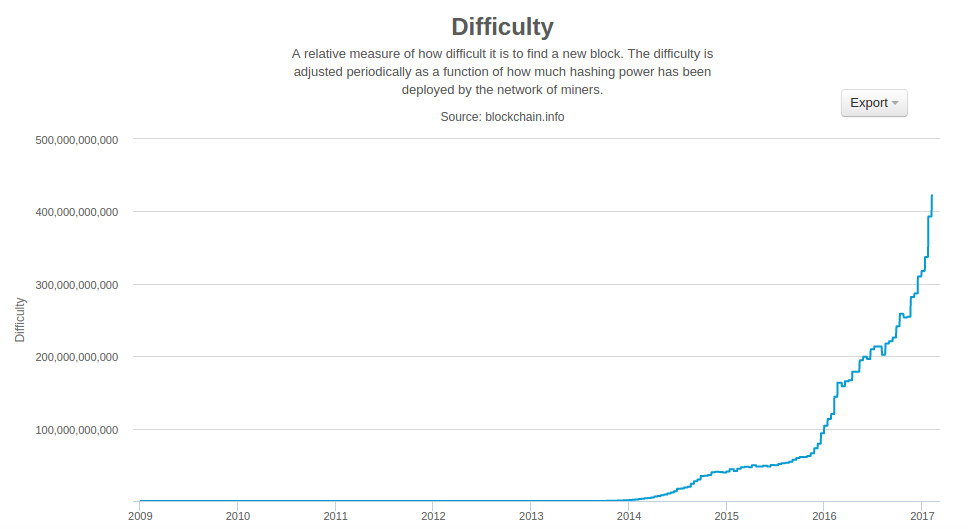
109,757,127 – 149,064,869 TH/sec (1.35× growth)

In the chart in [figure\_title](#network_hashing_power), we can see that Bitcoin network’s hashing power increased over the past two years. As you can see, the competition between miners and the growth of bitcoin has resulted in an exponential increase in the hashing power (total hashes per second across the network).



Total hashing power, terahashes per second (TH/sec) (chart on a linear scale)

As the amount of hashing power applied to mining bitcoin has exploded, the difficulty has risen to match it. The difficulty metric in the chart shown in [figure\_title](#bitcoin_difficulty) is measured as a ratio of current difficulty over minimum difficulty (the difficulty of the first block).



Bitcoin’s mining difficulty metric (chart on a logarithmic scale)

In the last two years, the ASIC mining chips have become increasingly denser, approaching the cutting edge of silicon fabrication with a feature size (resolution) of 7 nanometers (nm). Currently, ASIC manufacturers are aiming to overtake general-purpose CPU chip manufacturers, designing chips with a feature size of 5 nm, because the profitability of mining is driving this industry even faster than general computing. There are no more giant leaps left in bitcoin mining, because the industry has reached the forefront of Moore’s Law, which stipulates that computing density will double approximately every 18 months. Still, the mining power of the network continues to advance at an exponential pace as the race for higher density chips is matched with a race for higher density data centers where thousands of these chips can be deployed. It’s no longer about how much mining can be done with one chip, but how many chips can be squeezed into a building, while still dissipating the heat and providing adequate power.

==== The Extra Nonce Solution

Since 2012, bitcoin mining has evolved to resolve a fundamental limitation in the structure of the block header. In the early days of bitcoin, a miner could find a block by iterating through the nonce until the resulting hash was equal to or below the target. As difficulty increased, miners often cycled through all 4 billion values of the nonce without finding a block. However, this was easily resolved by updating the block timestamp to account for the elapsed time. Because the timestamp is part of the header, the change would allow miners to iterate through the values of the nonce again with different results. Once mining hardware exceeded 4 GH/sec, however, this approach became increasingly difficult because the nonce values were exhausted in less than a second. As ASIC mining equipment started pushing and then exceeding the TH/sec hash rate, the mining software needed more space for nonce values in order to find valid blocks. The timestamp could be stretched a bit, but moving it too far into the future would cause the block to become invalid. A new source of "change" was needed in the block header. The solution was to use the coinbase transaction as a source of extra nonce values. Because the coinbase script can store between 2 and 100 bytes of data, miners started using that space as extra nonce space, allowing them to explore a much larger range of block header values to find valid blocks. The coinbase transaction is included in the merkle tree, which means that any change in the coinbase script causes the merkle root to change. Eight bytes of extra nonce, plus the 4 bytes of "standard" nonce allow miners to explore a total 296 (8 followed by 28 zeros) possibilities *per second* without having to modify the timestamp. If, in the future, miners could run through all these possibilities, they could then modify the timestamp. There is also more space in the coinbase script for future expansion of the extra nonce space.

==== Mining Pools

In this highly competitive environment, individual miners working alone (also known as solo miners) don’t stand a chance. The likelihood of them finding a block to offset their electricity and hardware costs is so low that it represents a gamble, like playing the lottery. Even the fastest consumer ASIC mining system cannot keep up with commercial systems that stack tens of thousands of these chips in giant warehouses near hydroelectric powerstations. Miners now collaborate to form mining pools, pooling their hashing power and sharing the reward among thousands of participants. By participating in a pool, miners get a smaller share of the overall reward, but typically get rewarded every day, reducing uncertainty.

Let’s look at a specific example. Assume a miner has purchased mining hardware with a combined hashing rate of 14,000 gigahashes per second (GH/s), or 14 TH/s. In 2017 this equipment costs approximately $2,500 USD. The hardware consumes 1375 watts (1.3 kW) of electricity when running, 33 kW-hours a day, at a cost of $1 to $2 per day at very low electricity rates. At current bitcoin difficulty, the miner will be able to solo mine a block approximately once every 4 years. How do we work out that probability? It is based on a network-wide hashing rate of 3 EH/sec (in 2017), and the miner’s rate of 14 TH/sec:

P = (14 \* 10

12

/ 3 \* 10

18

) \* 210240 = 0.98

…​where 210240 is the number of blocks in four years. The miner has a 98% probability of finding a block over four years, based on the global hash rate at the beginning of the period.

If the miner does find a single block in that timeframe, the payout of 6.25 bitcoin, at approximately $1,000 per bitcoin, will result in a single payout of $6,250, which will produce a net profit of about $750. However, the chance of finding a block in a 4-year period depends on the miner’s luck. He might find two blocks in 4 years and make a larger profit. Or he might not find a block for 5 years and suffer a big financial loss. Even worse, the difficulty of the bitcoin Proof-of-Work algorithm is likely to go up significantly over that period, at the current rate of growth of hashing power, meaning the miner has, at most, one year to break even before the hardware is effectively obsolete and must be replaced by more powerful mining hardware. Financially this only makes sense at very low electricity cost (less than 1 cent per kW-hour) and only at very large scale.

Mining pools coordinate many hundreds or thousands of miners, over specialized pool-mining protocols. The individual miners configure their mining equipment to connect to a pool server, and specify a Bitcoin address, which will receive their share of the rewards. Their mining hardware remains connected to the pool server while mining, synchronizing their efforts with the other miners. Thus, the pool miners share the effort to mine a block and then share in the rewards.

Successful blocks pay the reward to a pool Bitcoin address, rather than individual miners. The pool server will periodically make payments to the miners' Bitcoin addresses, once their share of the rewards has reached a certain threshold. Typically, the pool server charges a percentage fee of the rewards for providing the pool-mining service.

Miners participating in a pool split the work of searching for a solution to a candidate block, earning "shares" for their mining contribution. The mining pool sets a higher target (lower difficulty) for earning a share, typically more than 1,000 times easier than the Bitcoin network’s target. When someone in the pool successfully mines a block, the reward is earned by the pool and then shared with all miners in proportion to the number of shares they contributed to the effort.

Pools are open to any miner, big or small, professional or amateur. A pool will therefore have some participants with a single small mining machine, and others with a garage full of high-end mining hardware. Some will be mining with a few tens of a kilowatt of electricity, others will be running a data center consuming a megawatt of power. How does a mining pool measure the individual contributions, so as to fairly distribute the rewards, without the possibility of cheating? The answer is to use bitcoin’s Proof-of-Work algorithm to measure each pool miner’s contribution, but set at a lower difficulty so that even the smallest pool miners win a share frequently enough to make it worthwhile to contribute to the pool. By setting a lower difficulty for earning shares, the pool measures the amount of work done by each miner. Each time a pool miner finds a block header hash that is equal to or less than the pool target, she proves she has done the hashing work to find that result. More importantly, the work to find shares contributes, in a statistically measurable way, to the overall effort to find a hash equal to or lower than the Bitcoin network’s target. Thousands of miners trying to find low-value hashes will eventually find one low enough to satisfy the Bitcoin network target.

Let’s return to the analogy of a dice game. If the dice players are throwing dice with a goal of throwing equal to or less than four (the overall network difficulty), a pool would set an easier target, counting how many times the pool players managed to throw equal to or less than eight. When pool players throw equal to or less than eight (the pool share target) but higher than four (higher than the overall network difficulty), they earn shares, but neither they nor the pool win the game because they don’t achieve the game target (equal to or less than four). The pool players will achieve the easier pool target much more often, earning them shares very regularly, even when they don’t achieve the harder target of winning the game. Every now and then, one of the pool players will throw a combined dice throw of equal to or less than four, the pool player wins a share and the whole pool wins the game. Then, the earnings can be distributed to the pool players based on the amount of shares each one has earned. Even though the target of eight-or-less wasn’t winning, it was a fair way to measure dice throws for the players, and it occasionally produces a four-or-less throw.

Similarly, a mining pool will set a (higher and easier) pool target that will ensure that an individual pool miner can find block header hashes that are equal to or less than the pool target often, earning shares. Every now and then, one of these attempts will produce a block header hash that is equal to or less than the Bitcoin network target, making it a valid block and the whole pool wins.

===== Managed pools

Most mining pools are "managed," meaning that there is a company or individual running a pool server. The owner of the pool server is called the *pool operator*, and he charges pool miners a percentage fee of the earnings.

The pool server runs specialized software and a pool-mining protocol that coordinate the activities of the pool miners. The pool server is also connected to one or more full Bitcoin nodes and has direct access to a full copy of the blockchain database. This allows the pool server to validate blocks and transactions on behalf of the pool miners, relieving them of the burden of running a full node. For pool miners, this is an important consideration, because a full node requires a dedicated computer with at least 300 to 350 GB of persistent storage (disk) and at least 2 to 4 GB of memory (RAM). Furthermore, the bitcoin software running on the full node needs to be monitored, maintained, and upgraded frequently. Any downtime caused by a lack of maintenance or lack of resources will hurt the miner’s profitability. For many miners, the ability to mine without running a full node is another big benefit of joining a managed pool.

Pool miners connect to the pool server using a mining protocol such as Stratum (STM) or GetBlockTemplate (GBT). An older standard called GetWork (GWK) has been mostly obsolete since late 2012, because it does not easily support mining at hash rates above 4 GH/s. Both the STM and GBT protocols create block *templates* that contain a template of a candidate block header. The pool server constructs a candidate block by aggregating transactions, adding a coinbase transaction (with extra nonce space), calculating the merkle root, and linking to the previous block hash. The header of the candidate block is then sent to each of the pool miners as a template. Each pool miner then mines using the block template, at a higher (easier) target than the Bitcoin network target, and sends any successful results back to the pool server to earn shares.

===== Peer-to-peer mining pool (P2Pool)

Managed pools create the possibility of cheating by the pool operator, who might direct the pool effort to double-spend transactions or invalidate blocks (see [simpara\_title](#consensus_attacks)). Furthermore, centralized pool servers represent a single-point-of-failure. If the pool server is down or is slowed by a denial-of-service attack, the pool miners cannot mine. In 2011, to resolve these issues of centralization, a new pool mining method was proposed and implemented: P2Pool, a peer-to-peer mining pool without a central operator.

P2Pool works by decentralizing the functions of the pool server, implementing a parallel blockchain-like system called a *share chain*. A share chain is a blockchain running at a lower difficulty than the Bitcoin blockchain. The share chain allows pool miners to collaborate in a decentralized pool by mining shares on the share chain at a rate of one share block every 30 seconds. Each of the blocks on the share chain records a proportionate share reward for the pool miners who contribute work, carrying the shares forward from the previous share block. When one of the share blocks also achieves the Bitcoin network target, it is propagated and included on the Bitcoin blockchain, rewarding all the pool miners who contributed to all the shares that preceded the winning share block. Essentially, instead of a pool server keeping track of pool miner shares and rewards, the share chain allows all pool miners to keep track of all shares using a decentralized consensus mechanism like bitcoin’s blockchain consensus mechanism.

P2Pool mining is more complex than pool mining because it requires that the pool miners run a dedicated computer with enough disk space, memory, and internet bandwidth to support a full Bitcoin node and the P2Pool node software. P2Pool miners connect their mining hardware to their local P2Pool node, which simulates the functions of a pool server by sending block templates to the mining hardware. On P2Pool, individual pool miners construct their own candidate blocks, aggregating transactions much like solo miners, but then mine collaboratively on the share chain. P2Pool is a hybrid approach that has the advantage of much more granular payouts than solo mining, but without giving too much control to a pool operator like managed pools.

Even though P2Pool reduces the concentration of power by mining pool operators, it is conceivably vulnerable to 51% attacks against the share chain itself. A much broader adoption of P2Pool does not solve the 51% attack problem for bitcoin itself. Rather, P2Pool makes bitcoin more robust overall, as part of a diversified mining ecosystem.

=== Consensus Attacks

Bitcoin’s consensus mechanism is, at least theoretically, vulnerable to attack by miners (or pools) that attempt to use their hashing power to dishonest or destructive ends. As we saw, the consensus mechanism depends on having a majority of the miners acting honestly out of self-interest. However, if a miner or group of miners can achieve a significant share of the mining power, they can attack the consensus mechanism so as to disrupt the security and availability of the Bitcoin network.

It is important to note that consensus attacks can only affect future consensus, or at best, the most recent past (tens of blocks). Bitcoin’s ledger becomes more and more immutable as time passes. While in theory, a fork can be achieved at any depth, in practice, the computing power needed to force a very deep fork is immense, making old blocks practically immutable. Consensus attacks also do not affect the security of the private keys and signing algorithm (ECDSA). A consensus attack cannot steal bitcoin, spend bitcoin without signatures, redirect bitcoin, or otherwise change past transactions or ownership records. Consensus attacks can only affect the most recent blocks and cause denial-of-service disruptions on the creation of future blocks.

One attack scenario against the consensus mechanism is called the "51% attack." In this scenario a group of miners, controlling a majority (51%) of the total network’s hashing power, collude to attack bitcoin. With the ability to mine the majority of the blocks, the attacking miners can cause deliberate "forks" in the blockchain and double-spend transactions or execute denial-of-service attacks against specific transactions or addresses. A fork/double-spend attack is where the attacker causes previously confirmed blocks to be invalidated by forking below them and re-converging on an alternate chain. With sufficient power, an attacker can invalidate six or more blocks in a row, causing transactions that were considered immutable (six confirmations) to be invalidated. Note that a double-spend can only be done on the attacker’s own transactions, for which the attacker can produce a valid signature. Double-spending one’s own transactions is profitable if by invalidating a transaction the attacker can get an irreversible exchange payment or product without paying for it.

Let’s examine a practical example of a 51% attack. In the first chapter, we looked at a transaction between Alice and Bob for a cup of coffee. Bob, the cafe owner, is willing to accept payment for cups of coffee without waiting for confirmation (mining in a block), because the risk of a double-spend on a cup of coffee is low in comparison to the convenience of rapid customer service. This is similar to the practice of coffee shops that accept credit card payments without a signature for amounts below $25, because the risk of a credit-card chargeback is low while the cost of delaying the transaction to obtain a signature is comparatively larger. In contrast, selling a more expensive item for bitcoin runs the risk of a double-spend attack, where the buyer broadcasts a competing transaction that spends the same inputs (UTXO) and cancels the payment to the merchant. A double-spend attack can happen in two ways: either before a transaction is confirmed, or if the attacker takes advantage of a blockchain fork to undo several blocks. A 51% attack allows attackers to double-spend their own transactions in the new chain, thus undoing the corresponding transaction in the old chain.

In our example, malicious attacker Mallory goes to Carol’s gallery and purchases a beautiful triptych painting depicting Satoshi Nakamoto as Prometheus. Carol sells "The Great Fire" paintings for $250,000 in bitcoin to Mallory. Instead of waiting for six or more confirmations on the transaction, Carol wraps and hands the paintings to Mallory after only one confirmation. Mallory works with an accomplice, Paul, who operates a large mining pool, and the accomplice launches a 51% attack as soon as Mallory’s transaction is included in a block. Paul directs the mining pool to remine the same block height as the block containing Mallory’s transaction, replacing Mallory’s payment to Carol with a transaction that double-spends the same input as Mallory’s payment. The double-spend transaction consumes the same UTXO and pays it back to Mallory’s wallet, instead of paying it to Carol, essentially allowing Mallory to keep the bitcoin. Paul then directs the mining pool to mine an additional block, so as to make the chain containing the double-spend transaction longer than the original chain (causing a fork below the block containing Mallory’s transaction). When the blockchain fork resolves in favor of the new (longer) chain, the double-spent transaction replaces the original payment to Carol. Carol is now missing the three paintings and also has no bitcoin payment. Throughout all this activity, Paul’s mining pool participants might remain blissfully unaware of the double-spend attempt, because they mine with automated miners and cannot monitor every transaction or block.

To protect against this kind of attack, a merchant selling large-value items must wait at least six confirmations before giving the product to the buyer. Alternatively, the merchant should use an escrow multisignature account, again waiting for several confirmations after the escrow account is funded. The more confirmations elapse, the harder it becomes to invalidate a transaction with a 51% attack. For high-value items, payment by bitcoin will still be convenient and efficient even if the buyer has to wait 24 hours for delivery, which would correspond to approximately 144 confirmations.

In addition to a double-spend attack, the other scenario for a consensus attack is to deny service to specific bitcoin participants (specific Bitcoin addresses). An attacker with a majority of the mining power can simply ignore specific transactions. If they are included in a block mined by another miner, the attacker can deliberately fork and remine that block, again excluding the specific transactions. This type of attack can result in a sustained denial-of-service against a specific address or set of addresses for as long as the attacker controls the majority of the mining power.

Despite its name, the 51% attack scenario doesn’t actually require 51% of the hashing power. In fact, such an attack can be attempted with a smaller percentage of the hashing power. The 51% threshold is simply the level at which such an attack is almost guaranteed to succeed. A consensus attack is essentially a tug-of-war for the next block and the "stronger" group is more likely to win. With less hashing power, the probability of success is reduced, because other miners control the generation of some blocks with their "honest" mining power. One way to look at it is that the more hashing power an attacker has, the longer the fork he can deliberately create, the more blocks in the recent past he can invalidate, or the more blocks in the future he can control. Security research groups have used statistical modeling to claim that various types of consensus attacks are possible with as little as 30% of the hashing power.

The massive increase of total hashing power has arguably made bitcoin impervious to attacks by a single miner. There is no possible way for a solo miner to control more than a small percentage of the total mining power. However, the centralization of control caused by mining pools has introduced the risk of for-profit attacks by a mining pool operator. The pool operator in a managed pool controls the construction of candidate blocks and also controls which transactions are included. This gives the pool operator the power to exclude transactions or introduce double-spend transactions. If such abuse of power is done in a limited and subtle way, a pool operator could conceivably profit from a consensus attack without being noticed.

Not all attackers will be motivated by profit, however. One potential attack scenario is where an attacker intends to disrupt the Bitcoin network without the possibility of profiting from such disruption. A malicious attack aimed at crippling bitcoin would require enormous investment and covert planning, but could conceivably be launched by a well-funded, most likely state-sponsored, attacker. Alternatively, a well-funded attacker could attack bitcoin’s consensus by simultaneously amassing mining hardware, compromising pool operators, and attacking other pools with denial-of-service. All of these scenarios are theoretically possible, but increasingly impractical as the Bitcoin network’s overall hashing power continues to grow exponentially.

Undoubtedly, a serious consensus attack would erode confidence in bitcoin in the short term, possibly causing a significant price decline. However, the Bitcoin network and software are constantly evolving, so consensus attacks would be met with immediate countermeasures by the bitcoin community, making bitcoin more robust.

=== Changing the Consensus Rules

The rules of consensus determine the validity of transactions and blocks. These rules are the basis for collaboration between all Bitcoin nodes and are responsible for the convergence of all local perspectives into a single consistent blockchain across the entire network.

While the consensus rules are invariable in the short term and must be consistent across all nodes, they are not invariable in the long term. In order to evolve and develop the Bitcoin system, the rules have to change from time to time to accommodate new features, improvements, or bug fixes. Unlike traditional software development, however, upgrades to a consensus system are much more difficult and require coordination between all the participants.

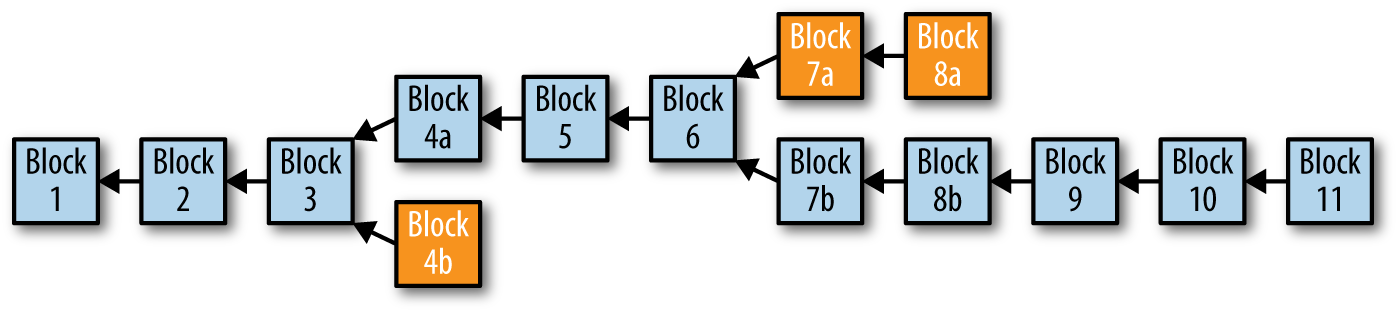
==== Hard Forks In [simpara\_title](#forks) we looked at how the Bitcoin network may briefly diverge, with two parts of the network following two different branches of the blockchain for a short time. We saw how this process occurs naturally, as part of the normal operation of the network and how the network reconverges on a common blockchain after one or more blocks are mined.

There is another scenario in which the network may diverge into following two chains: a change in the consensus rules. This type of fork is called a *hard fork*, because after the fork the network does not reconverge onto a single chain. Instead, the two chains evolve independently. Hard forks occur when part of the network is operating under a different set of consensus rules than the rest of the network. This may occur because of a bug or because of a deliberate change in the implementation of the consensus rules.

Hard forks can be used to change the rules of consensus, but they require coordination between all participants in the system. Any nodes that do not upgrade to the new consensus rules are unable to participate in the consensus mechanism and are forced onto a separate chain at the moment of the hard fork. Thus, a change introduced by a hard fork can be thought of as not "forward compatible," in that non-upgraded systems can’t process the new consensus rules after the hard fork event.

Let’s examine the mechanics of a hard fork with a specific example.

[figure\_title](#blockchainwithforks) shows a blockchain with two forks. At block height 4, a one-block fork occurs. This is the type of spontaneous fork we saw in [simpara\_title](#forks). With the mining of block 5, the network reconverges on one chain and the fork is resolved.



A blockchain with forks

Later, however, at block height 6, a hard fork occurs. Let’s assume that a new implementation of the client is released with a change in the consensus rules. Starting on block height 7, miners running this new implementation will accept a new type of digital signature, let’s call it a "Smores" signature, that is not ECDSA based. Immediately after, a node running the new implementation creates a transaction that contains a Smores signature and a miner with the updated software mines block 7b containing this transaction.

Any node or miner that has not upgraded the software to validate Smores signatures is now unable to process block 7b. From their perspective, both the transaction that contained a Smores signature and block 7b that contained that transaction are invalid, because they are evaluating them based upon the old consensus rules. These nodes will reject the transaction and the block and will not propagate them. Any miners that are using the old rules will not accept block 7b and will continue to mine a candidate block whose parent is block 6. In fact, miners using the old rules may not even receive block 7b if all the nodes they are connected to are also obeying the old rules and therefore not propagating the block. Eventually, they will be able to mine block 7a, which is valid under the old rules and does not contain any transactions with Smores signatures.

The two chains continue to diverge from this point. Miners on the "b" chain will continue to accept and mine transactions containing Smores signatures, while miners on the "a" chain will continue to ignore these transactions. Even if block 8b does not contain any Smores-signed transactions, the miners on the "a" chain cannot process it. To them it appears to be an orphan block, as its parent "7b" is not recognized as a valid block.

==== Hard Forks: Software, Network, Mining, and Chain

For software developers, the term "fork" has another meaning, adding confusion to the term "hard fork." In open source software, a fork occurs when a group of developers choose to follow a different software roadmap and start a competing implementation of an open source project. We’ve already discussed two circumstances that will lead to a hard fork in bitcoin: a bug in the consensus rules and a deliberate modification of the consensus rules. In the case of a deliberate change to the consensus rules, a software fork precedes the hard fork. However, for this type of hard fork to occur, a new software implementation of the consensus rules must be developed, adopted, and launched.

Examples of software forks that have attempted to change consensus rules include Bitcoin XT, Bitcoin Classic, and most recently Bitcoin Unlimited. However, none of these software forks have resulted in a hard fork. While a software fork is a necessary precondition, it is not in itself sufficient for a hard fork to occur. For a hard fork to occur, the competing implementation must be adopted and the new rules activated, by miners, wallets, and intermediary nodes. Conversely, there are numerous alternative implementations of Bitcoin Core, and even software forks, that do not change the consensus rules and barring a bug, can coexist on the network and interoperate without causing a hard fork.

Consensus rules may differ in obvious and explicit ways, in the validation of transactions or blocks. The rules may also differ in more subtle ways, in the implementation of the consensus rules as they apply to bitcoin scripts or cryptographic primitives such as digital signatures. Finally, the consensus rules may differ in unanticipated ways because of implicit consensus constraints imposed by system limitations or implementation details. An example of the latter was seen in the unanticipated hard fork during the upgrade of Bitcoin Core 0.7 to 0.8, which was caused by a limitation in the Berkeley DB implementation used to store blocks.

Conceptually, we can think of a hard fork as developing in four stages: a software fork, a network fork, a mining fork, and a chain fork.

The process begins when an alternative implementation of the client, with modified consensus rules, is created by developers.

When this forked implementation is deployed in the network, a certain percentage of miners, wallet users, and intermediate nodes may adopt and run this implementation. A resulting fork will depend upon whether the new consensus rules apply to blocks, transactions, or some other aspect of the system. If the new consensus rules pertain to transactions, then a wallet creating a transaction under the new rules may precipitate a network fork, followed by a hard fork when the transaction is mined into a block. If the new rules pertain to blocks, then the hard fork process will begin when a block is mined under the new rules.

First, the network will fork. Nodes based on the original implementation of the consensus rules will reject any transactions and blocks that are created under the new rules. Furthermore, the nodes following the original consensus rules will temporarily ban and disconnect from any nodes that are sending them these invalid transactions and blocks. As a result, the network will partition into two: old nodes will only remain connected to old nodes and new nodes will only be connected to new nodes. A single transaction or block based on the new rules will ripple through the network and result in the partition into two networks.

Once a miner using the new rules mines a block, the mining power and chain will also fork. New miners will mine on top of the new block, while old miners will mine a separate chain based on the old rules. The partitioned network will make it so that the miners operating on separate consensus rules won’t likely receive each other’s blocks, as they are connected to two separate networks.

==== Diverging Miners and Difficulty

As miners diverge into mining two different chains, the hashing power is split between the chains. The mining power can be split in any proportion between the two chains. The new rules may only be followed by a minority, or by the vast majority of the mining power.

Let’s assume, for example, an 80%–20% split, with the majority of the mining power using the new consensus rules. Let’s also assume that the fork occurs immediately after a retargeting period.

The two chains would each inherit the difficulty from the retargeting period. The new consensus rules would have 80% of the previously available mining power committed to them. From the perspective of this chain, the mining power has suddenly declined by 20% vis-a-vis the previous period. Blocks will be found on average every 12.5 minutes, representing the 20% decline in mining power available to extend this chain. This rate of block issuance will continue (barring any changes in hashing power) until 2016 blocks are mined, which will take approximately 25,200 minutes (at 12.5 minutes per block), or 17.5 days. After 17.5 days, a retarget will occur and the difficulty will adjust (reduced by 20%) to produce 10-minute blocks again, based on the reduced amount of hashing power in this chain.

The minority chain, mining under the old rules with only 20% of the hashing power, will face a much more difficult task. On this chain, blocks will now be mined every 50 minutes on average. The difficulty will not be adjusted for 2016 blocks, which will take 100,800 minutes, or approximately 10 weeks to mine. Assuming a fixed capacity per block, this will also result in a reduction of transaction capacity by a factor of 5, as there are fewer blocks per hour available to record transactions.

==== Contentious Hard Forks

This is the dawn of consensus software development. Just as open source development changed both the methods and products of software and created new methodologies, new tools, and new communities in its wake, consensus software development also represents a new frontier in computer science. Out of the debates, experiments, and tribulations of the bitcoin development roadmap, we will see new development tools, practices, methodologies, and communities emerge.

Hard forks are seen as risky because they force a minority to either upgrade or remain on a minority chain. The risk of splitting the entire system into two competing systems is seen by many as an unacceptable risk. As a result, many developers are reluctant to use the hard fork mechanism to implement upgrades to the consensus rules, unless there is near-unanimous support from the entire network. Any hard fork proposals that do not have near-unanimous support are considered too "contentious" to attempt without risking a partition of the system.

The issue of hard forks is highly controversial in the bitcoin development community, especially as it relates to any proposed changes to the consensus rules controlling the maximum block size limit. Some developers are opposed to any form of hard fork, seeing it as too risky. Others see the mechanism of hard fork as an essential tool for upgrading the consensus rules in a way that avoids "technical debt" and provides a clean break with the past. Finally, some developers see hard forks as a mechanism that should be used rarely, with a lot of advance planning and only under near-unanimous consensus.

Already we have seen the emergence of new methodologies to address the risks of hard forks. In the next section, we will look at soft forks, and the BIP-34 and BIP-9 methods for signaling and activation of consensus modifications.

==== Soft Forks

Not all consensus rule changes cause a hard fork. Only consensus changes that are forward-incompatible cause a fork. If the change is implemented in such a way that a non-upgraded client still sees the transaction or block as valid under the previous rules, the change can happen without a fork.

The term *soft fork* was introduced to distinguish this upgrade method from a "hard fork." In practice, a soft fork is not a fork at all. A soft fork is a forward-compatible change to the consensus rules that allows non-upgraded clients to continue to operate in consensus with the new rules.

One aspect of soft forks that is not immediately obvious is that soft fork upgrades can only be used to constrain the consensus rules, not to expand them. In order to be forward compatible, transactions and blocks created under the new rules must be valid under the old rules too, but not vice versa. The new rules can only limit what is valid; otherwise, they will trigger a hard fork when rejected under the old rules.

Soft forks can be implemented in a number of ways—the term does not specify a particular method, rather a set of methods that all have one thing in common: they don’t require all nodes to upgrade or force non-upgraded nodes out of consensus.

===== Soft forks redefining NOP opcodes

A number of soft forks have been implemented in bitcoin, based on the re-interpretation of NOP opcodes. Bitcoin Script had ten opcodes reserved for future use, NOP1 through NOP10. Under the consensus rules, the presence of these opcodes in a script is interpreted as a null-potent operator, meaning they have no effect. Execution continues after the NOP opcode as if it wasn’t there.

A soft fork therefore can modify the semantics of a NOP code to give it new meaning. For example, BIP-65 (CHECKLOCKTIMEVERIFY) reinterpreted the NOP2 opcode. Clients implementing BIP-65 interpret NOP2 as OP\_CHECKLOCKTIMEVERIFY and impose an absolute locktime consensus rule on UTXO that contain this opcode in their locking scripts. This change is a soft fork because a transaction that is valid under BIP-65 is also valid on any client that is not implementing (ignorant of) BIP-65. To the old clients, the script contains an NOP code, which is ignored.

===== Other ways to soft fork upgrade

The reinterpretation of NOP opcodes was both planned for and an obvious mechanism for consensus upgrades. Recently, however, another soft fork mechanism was introduced that does not rely on NOP opcodes for a very specific type of consensus change. This is examined in more detail in [simpara\_title](#segwit). Segwit is an architectural change to the structure of a transaction, which moves the unlocking script (witness) from inside the transaction to an external data structure (segregating it). Segwit was initially envisioned as a hard fork upgrade, as it modified a fundamental structure (transaction). In November 2015, a developer working on Bitcoin Core proposed a mechanism by which segwit could be introduced as a soft fork. The mechanism used for this is a modification of the locking script of UTXO created under segwit rules, such that non-upgraded clients see the locking script as redeemable with any unlocking script whatsoever. As a result, segwit can be introduced without requiring every node to upgrade or split from the chain: a soft fork.

It is likely that there are other, yet to be discovered, mechanisms by which upgrades can be made in a forward-compatible way as a soft fork.

==== Criticisms of Soft Forks

Soft forks based on the NOP opcodes are relatively uncontroversial. The NOP opcodes were placed in Bitcoin Script with the explicit goal of allowing non-disruptive upgrades.

However, many developers are concerned that other methods of soft fork upgrades make unacceptable tradeoffs. Common criticisms of soft fork changes include:

Technical debt

Because soft forks are more technically complex than a hard fork upgrade, they introduce *technical debt*, a term that refers to increasing the future cost of code maintenance because of design tradeoffs made in the past. Code complexity in turn increases the likelihood of bugs and security vulnerabilities.

Validation relaxation

Non-upgraded clients see transactions as valid, without evaluating the modified consensus rules. In effect, the non-upgraded clients are not validating using the full range of consensus rules, as they are blind to the new rules. This applies to NOP-based upgrades, as well as other soft fork upgrades.

Irreversible upgrades

Because soft forks create transactions with additional consensus constraints, they become irreversible upgrades in practice. If a soft fork upgrade were to be reversed after being activated, any transactions created under the new rules could result in a loss of funds under the old rules. For example, if a CLTV transaction is evaluated under the old rules, there is no timelock constraint and it can be spent at any time. Therefore, critics contend that a failed soft fork that had to be reversed because of a bug would almost certainly lead to loss of funds.

=== Soft Fork Signaling with Block Version

Since soft forks allow non-upgraded clients to continue to operate within consensus, the mechanism for "activating" a soft fork is through miners signaling readiness: a majority of miners must agree that they are ready and willing to enforce the new consensus rules. To coordinate their actions, there is a signaling mechanism that allows them to show their support for a consensus rule change. This mechanism was introduced with the activation of BIP-34 in March 2013 and replaced by the activation of BIP-9 in July 2016.

==== BIP-34 Signaling and Activation

The first implementation, in BIP-34, used the block version field to allow miners to signal readiness for a specific consensus rule change. Prior to BIP-34, the block version was set to "1" by *convention* not enforced by *consensus*.

BIP-34 defined a consensus rule change that required the coinbase data field of a coinbase transaction input to contain the block height. Prior to BIP-34, the coinbase data could contain any arbitrary data the miners chose to include. After activation of BIP-34, valid blocks had to contain a specific block-height at the beginning of the coinbase data and be identified with a version number greater than or equal to "2."

To signal the change and activation of BIP-34, miners set the block version to "2," instead of "1." This did not immediately make version "1" blocks invalid. Once activated, version "1" blocks would become invalid and all version "2" blocks would be required to contain the block height in the coinbase to be valid.

BIP-34 defined a two-step activation mechanism, based on a rolling window of 1000 blocks. A miner would signal his or her individual readiness for BIP-34 by constructing blocks with "2" as the version number. Strictly speaking, these blocks did not yet have to comply with the new consensus rule of including the block-height in the coinbase transaction because the consensus rule had not yet been activated. The consensus rules activated in two steps:

* If 75% (750 of the most recent 1000 blocks) are marked with version "2," then version "2" blocks must contain block height in the coinbase transaction or they are rejected as invalid. Version "1" blocks are still accepted by the network and do not need to contain block-height. The old and new consensus rules coexist during this period.
* When 95% (950 of the most recent 1000 blocks) are version "2," version "1" blocks are no longer considered valid. Version "2" blocks are valid only if they contain the block-height in the coinbase (as per the previous threshold). Thereafter, all blocks must comply with the new consensus rules, and all valid blocks must contain block-height in the coinbase transaction.

After successful signaling and activation under the BIP-34 rules, this mechanism was used twice more to activate soft forks:

* [BIP-66](https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki) Strict DER Encoding of Signatures was activated by BIP-34 style signaling with a block version "3" and invalidating version "2" blocks.
* [BIP-65](https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki) CHECKLOCKTIMEVERIFY was activated by BIP-34 style signaling with a block version "4" and invalidating version "3" blocks.

After the activation of BIP-65, the signaling and activation mechanism of BIP-34 was retired and replaced with the BIP-9 signaling mechanism described next.

The standard is defined in [BIP-34 (Block v2, Height in Coinbase)](https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki).

==== BIP-9 Signaling and Activation

The mechanism used by BIP-34, BIP-66, and BIP-65 was successful in activating three soft forks. However, it was replaced because it had several limitations:

* By using the integer value of the block version, only one soft fork could be activated at a time, so it required coordination between soft fork proposals and agreement on their prioritization and sequencing.
* Furthermore, because the block version was incremented, the mechanism didn’t offer a straightforward way to reject a change and then propose a different one. If old clients were still running, they could mistake signaling for a new change as signaling for the previously rejected change.
* Each new change irrevocably reduced the available block versions for future changes.

BIP-9 was proposed to overcome these challenges and improve the rate and ease of implementing future changes.

BIP-9 interprets the block version as a bit field instead of an integer. Because the block version was originally used as an integer, versions 1 through 4, only 29 bits remain available to be used as a bit field. This leaves 29 bits that can be used to independently and simultaneously signal readiness on 29 different proposals.

BIP-9 also sets a maximum time for signaling and activation. This way miners don’t need to signal forever. If a proposal is not activated within the TIMEOUT period (defined in the proposal), the proposal is considered rejected. The proposal may be resubmitted for signaling with a different bit, renewing the activation period.

Furthermore, after the TIMEOUT has passed and a feature has been activated or rejected, the signaling bit can be reused for another feature without confusion. Therefore, up to 29 changes can be signaled in parallel and after TIMEOUT the bits can be "recycled" to propose new changes.

While signaling bits can be reused or recycled, as long as the voting period does not overlap, the authors of BIP-9 recommend that bits are reused only when necessary; unexpected behavior could occur due to bugs in older software. In short, we should not expect to see reuse until all 29 bits have been used once.

Proposed changes are identified by a data structure that contains the following fields:

name

A short description used to distinguish between proposals. Most often the BIP describing the proposal, as "bipN," where N is the BIP number.

bit

0 through 28, the bit in the block version that miners use to signal approval for this proposal.

starttime

The time (based on Median Time Past, or MTP) that signaling starts after which the bit’s value is interpreted as signaling readiness for the proposal.

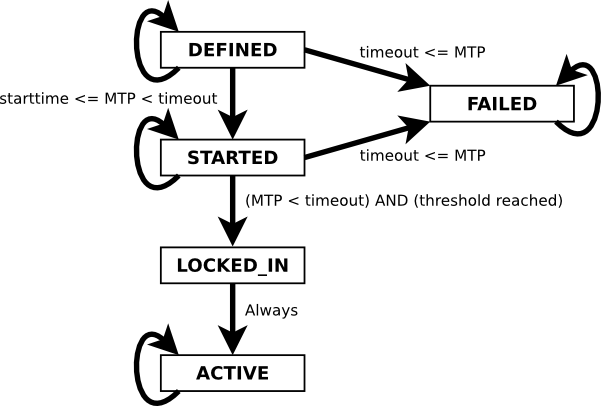
endtime

The time (based on MTP) after which the change is considered rejected if it has not reached the activation threshold.

Unlike BIP-34, BIP-9 counts activation signaling in whole intervals based on the difficulty retarget period of 2016 blocks. For every retarget period, if the sum of blocks signaling for a proposal exceeds 95% (1916 of 2016), the proposal will be activated one retarget period later.

BIP-9 offers a proposal state diagram to illustrate the various stages and transitions for a proposal, as shown in [figure\_title](#bip9states).

Proposals start in the DEFINED state, once their parameters are known (defined) in the bitcoin software. For blocks with MTP after the start time, the proposal state transitions to STARTED. If the voting threshold is exceeded within a retarget period and the timeout has not been exceeded, the proposal state transitions to LOCKED\_IN. One retarget period later, the proposal becomes ACTIVE. Proposals remain in the ACTIVE state perpetually once they reach that state. If the timeout elapses before the voting threshold has been reached, the proposal state changes to FAILED, indicating a rejected proposal. FAILED proposals remain in that state perpetually.



BIP-9 state transition diagram

BIP-9 was first implemented for the activation of CHECKSEQUENCEVERIFY and associated BIPs (68, 112, 113). The proposal named "csv" was activated successfully in July of 2016.

The standard is defined in [BIP-9 (Version bits with timeout and delay)](https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki).

=== Consensus Software Development

Consensus software continues to evolve and there is much discussion on the various mechanisms for changing the consensus rules. By its very nature, bitcoin sets a very high bar on coordination and consensus for changes. As a decentralized system, it has no "authority" that can impose its will on the participants of the network. Power is diffused between multiple constituencies such as miners, core developers, wallet developers, exchanges, merchants, and end users. Decisions cannot be made unilaterally by any of these constituencies. For example, while miners can theoretically change the rules by simple majority (51%), they are constrained by the consent of the other constituencies. If they act unilaterally, the rest of the participants may simply refuse to follow them, keeping the economic activity on a minority chain. Without economic activity (transactions, merchants, wallets, exchanges), the miners will be mining a worthless coin with empty blocks. This diffusion of power means that all the participants must coordinate, or no changes can be made. Status quo is the stable state of this system with only a few changes possible if there is strong consensus by a very large majority. The 95% threshold for soft forks is reflective of this reality.

It is important to recognize that there is no perfect solution for consensus development. Both hard forks and soft forks involve tradeoffs. For some types of changes, soft forks may be a better choice; for others, hard forks may be a better choice. There is no perfect choice; both carry risks. The one constant characteristic of consensus software development is that change is difficult and consensus forces compromise.

Some see this as a weakness of consensus systems. In time, you may come to see it as I do, as the system’s greatest strength.

== Bitcoin Security

Securing bitcoin is challenging because bitcoin is not an abstract reference to value, like a balance in a bank account. Bitcoin is very much like digital cash or gold. You’ve probably heard the expression, "Possession is nine-tenths of the law." Well, in bitcoin, possession is ten-tenths of the law. Possession of the keys to unlock the bitcoin is equivalent to possession of cash or a chunk of precious metal. You can lose it, misplace it, have it stolen, or accidentally give the wrong amount to someone. In every one of these cases, users have no recourse, just as if they dropped cash on a public sidewalk.

However, bitcoin has capabilities that cash, gold, and bank accounts do not. A bitcoin wallet, containing your keys, can be backed up like any file. It can be stored in multiple copies, even printed on paper for hard-copy backup. You can’t "back up" cash, gold, or bank accounts. Bitcoin is different enough from anything that has come before that we need to think about bitcoin security in a novel way too.

=== Security Principles

The core principle in bitcoin is decentralization and it has important implications for security. A centralized model, such as a traditional bank or payment network, depends on access control and vetting to keep bad actors out of the system. By comparison, a decentralized system like bitcoin pushes the responsibility and control to the users. Because security of the network is based on Proof-of-Work, not access control, the network can be open and no encryption is required for bitcoin traffic.

On a traditional payment network, such as a credit card system, the payment is open-ended because it contains the user’s private identifier (the credit card number). After the initial charge, anyone with access to the identifier can "pull" funds and charge the owner again and again. Thus, the payment network has to be secured end-to-end with encryption and must ensure that no eavesdroppers or intermediaries can compromise the payment traffic, in transit or when it is stored (at rest). If a bad actor gains access to the system, he can compromise current transactions *and* payment tokens that can be used to create new transactions. Worse, when customer data is compromised, the customers are exposed to identity theft and must take action to prevent fraudulent use of the compromised accounts.

Bitcoin is dramatically different. A bitcoin transaction authorizes only a specific value to a specific recipient and cannot be forged or modified. It does not reveal any private information, such as the identities of the parties, and cannot be used to authorize additional payments. Therefore, a bitcoin payment network does not need to be encrypted or protected from eavesdropping. In fact, you can broadcast bitcoin transactions over an open public channel, such as unsecured WiFi or Bluetooth, with no loss of security.

Bitcoin’s decentralized security model puts a lot of power in the hands of the users. With that power comes responsibility for maintaining the secrecy of the keys. For most users that is not easy to do, especially on general-purpose computing devices such as internet-connected smartphones or laptops. Although bitcoin’s decentralized model prevents the type of mass compromise seen with credit cards, many users are not able to adequately secure their keys and get hacked, one by one.

==== Developing Bitcoin Systems Securely

The most important principle for bitcoin developers is decentralization. Most developers will be familiar with centralized security models and might be tempted to apply these models to their bitcoin applications, with disastrous results.

Bitcoin’s security relies on decentralized control over keys and on independent transaction validation by miners. If you want to leverage bitcoin’s security, you need to ensure that you remain within the bitcoin security model. In simple terms: don’t take control of keys away from users and don’t take transactions off the blockchain.

For example, many early bitcoin exchanges concentrated all user funds in a single "hot" wallet with keys stored on a single server. Such a design removes control from users and centralizes control over keys in a single system. Many such systems have been hacked, with disastrous consequences for their customers.

Another common mistake is to take transactions "off blockchain" in a misguided effort to reduce transaction fees or accelerate transaction processing. An "off blockchain" system will record transactions on an internal, centralized ledger and only occasionally synchronize them to the Bitcoin blockchain. This practice, again, substitutes decentralized bitcoin security with a proprietary and centralized approach. When transactions are off blockchain, improperly secured centralized ledgers can be falsified, diverting funds and depleting reserves, unnoticed.

Unless you are prepared to invest heavily in operational security, multiple layers of access control, and audits (as the traditional banks do) you should think very carefully before taking funds outside of bitcoin’s decentralized security context. Even if you have the funds and discipline to implement a robust security model, such a design merely replicates the fragile model of traditional financial networks, plagued by identity theft, corruption, and embezzlement. To take advantage of bitcoin’s unique decentralized security model, you have to avoid the temptation of centralized architectures that might feel familiar but ultimately subvert bitcoin’s security.

==== The Root of Trust

Traditional security architecture is based upon a concept called the *root of trust*, which is a trusted core used as the foundation for the security of the overall system or application. Security architecture is developed around the root of trust as a series of concentric circles, like layers in an onion, extending trust outward from the center. Each layer builds upon the more-trusted inner layer using access controls, digital signatures, encryption, and other security primitives. As software systems become more complex, they are more likely to contain bugs, which make them vulnerable to security compromise. As a result, the more complex a software system becomes, the harder it is to secure. The root of trust concept ensures that most of the trust is placed within the least complex part of the system, and therefore least vulnerable, parts of the system, while more complex software is layered around it. This security architecture is repeated at different scales, first establishing a root of trust within the hardware of a single system, then extending that root of trust through the operating system to higher-level system services, and finally across many servers layered in concentric circles of diminishing trust.

Bitcoin security architecture is different. In bitcoin, the consensus system creates a trusted public ledger that is completely decentralized. A correctly validated blockchain uses the genesis block as the root of trust, building a chain of trust up to the current block. Bitcoin systems can and should use the blockchain as their root of trust. When designing a complex bitcoin application that consists of services on many different systems, you should carefully examine the security architecture in order to ascertain where trust is being placed. Ultimately, the only thing that should be explicitly trusted is a fully validated blockchain. If your application explicitly or implicitly vests trust in anything but the blockchain, that should be a source of concern because it introduces vulnerability. A good method to evaluate the security architecture of your application is to consider each individual component and evaluate a hypothetical scenario where that component is completely compromised and under the control of a malicious actor. Take each component of your application, in turn, and assess the impacts on the overall security if that component is compromised. If your application is no longer secure when components are compromised, that shows you have misplaced trust in those components. A bitcoin application without vulnerabilities should be vulnerable only to a compromise of the bitcoin consensus mechanism, meaning that its root of trust is based on the strongest part of the bitcoin security architecture.

The numerous examples of hacked bitcoin exchanges serve to underscore this point because their security architecture and design fails even under the most casual scrutiny. These centralized implementations had invested trust explicitly in numerous components outside the Bitcoin blockchain, such as hot wallets, centralized ledger databases, vulnerable encryption keys, and similar schemes.

=== User Security Best Practices

Humans have used physical security controls for thousands of years. By comparison, our experience with digital security is less than 50 years old. Modern general-purpose operating systems are not very secure and not particularly suited to storing digital money. Our computers are constantly exposed to external threats via always-on internet connections. They run thousands of software components from hundreds of authors, often with unconstrained access to the user’s files. A single piece of rogue software, among the many thousands installed on your computer, can compromise your keyboard and files, stealing any bitcoin stored in wallet applications. The level of computer maintenance required to keep a computer virus-free and trojan-free is beyond the skill level of all but a tiny minority of computer users.

Despite decades of research and advancements in information security, digital assets are still woefully vulnerable to a determined adversary. Even the most highly protected and restricted systems, in financial services companies, intelligence agencies, and defense contractors, are frequently breached. Bitcoin creates digital assets that have intrinsic value and can be stolen and diverted to new owners instantly and irrevocably. This creates a massive incentive for hackers. Until now, hackers had to convert identity information or account tokens—such as credit cards and bank accounts—into value after compromising them. Despite the difficulty of fencing and laundering financial information, we have seen ever-escalating thefts. Bitcoin escalates this problem because it doesn’t need to be fenced or laundered; it is intrinsic value within a digital asset.

Fortunately, bitcoin also creates the incentives to improve computer security. Whereas previously the risk of computer compromise was vague and indirect, bitcoin makes these risks clear and obvious. Holding bitcoin on a computer serves to focus the user’s mind on the need for improved computer security. As a direct result of the proliferation and increased adoption of bitcoin and other digital currencies, we have seen an escalation in both hacking techniques and security solutions. In simple terms, hackers now have a very juicy target and users have a clear incentive to defend themselves.

Over the past three years, as a direct result of bitcoin adoption, we have seen tremendous innovation in the realm of information security in the form of hardware encryption, key storage and hardware wallets, multisignature technology, and digital escrow. In the following sections we will examine various best practices for practical user security.

==== Physical Bitcoin Storage

Because most users are far more comfortable with physical security than information security, a very effective method for protecting bitcoin is to convert them into physical form. Bitcoin keys are nothing more than long numbers. This means that they can be stored in a physical form, such as printed on paper or etched on a metal coin. Securing the keys then becomes as simple as physically securing the printed copy of the bitcoin keys. A set of bitcoin keys that is printed on paper is called a "paper wallet," and there are many free tools that can be used to create them. I personally keep the vast majority of my bitcoin (99% or more) stored on paper wallets, encrypted with BIP-38, with multiple copies locked in safes. Keeping bitcoin offline is called *cold storage* and it is one of the most effective security techniques. A cold storage system is one where the keys are generated on an offline system (one never connected to the internet) and stored offline either on paper or on digital media, such as a USB memory stick.

==== Hardware Wallets

In the long term, bitcoin security increasingly will take the form of hardware tamper-proof wallets. Unlike a smartphone or desktop computer, a bitcoin hardware wallet has just one purpose: to hold bitcoin securely. Without general-purpose software to compromise and with limited interfaces, hardware wallets can deliver an almost foolproof level of security to nonexpert users. I expect to see hardware wallets become the predominant method of bitcoin storage. For an example of such a hardware wallet, see the [Trezor](https://trezor.io/).

==== Balancing Risk

Although most users are rightly concerned about bitcoin theft, there is an even bigger risk. Data files get lost all the time. If they contain bitcoin, the loss is much more painful. In the effort to secure their bitcoin wallets, users must be very careful not to go too far and end up losing the bitcoin. In July 2011, a well-known bitcoin awareness and education project lost almost 7,000 bitcoin. In their effort to prevent theft, the owners had implemented a complex series of encrypted backups. In the end they accidentally lost the encryption keys, making the backups worthless and losing a fortune. Like hiding money by burying it in the desert, if you secure your bitcoin too well you might not be able to find it again.

==== Diversifying Risk

Would you carry your entire net worth in cash in your wallet? Most people would consider that reckless, yet bitcoin users often keep all their bitcoin in a single wallet. Instead, users should spread the risk among multiple and diverse bitcoin wallets. Prudent users will keep only a small fraction, perhaps less than 5%, of their bitcoin in an online or mobile wallet as "pocket change." The rest should be split between a few different storage mechanisms, such as a desktop wallet and offline (cold storage).

==== Multisig and Governance

Whenever a company or individual stores large amounts of bitcoin, they should consider using a multisignature Bitcoin address. Multisignature addresses secure funds by requiring a minimum number of signatures to make a payment. The signing keys should be stored in a number of different locations and under the control of different people. In a corporate environment, for example, the keys should be generated independently and held by several company executives, to ensure no single person can compromise the funds. Multisignature addresses can also offer redundancy, where a single person holds several keys that are stored in different locations.

==== Survivability

One important security consideration that is often overlooked is availability, especially in the context of incapacity or death of the key holder. Bitcoin users are told to use complex passwords and keep their keys secure and private, not sharing them with anyone. Unfortunately, that practice makes it almost impossible for the user’s family to recover any funds if the user is not available to unlock them. In most cases, in fact, the families of bitcoin users might be completely unaware of the existence of the bitcoin funds.

If you have a lot of bitcoin, you should consider sharing access details with a trusted relative or lawyer. A more complex survivability scheme can be set up with multi-signature access and estate planning through a lawyer specialized as a "digital asset executor."

=== Conclusion

Bitcoin is a completely new, unprecedented, and complex technology. Over time we will develop better security tools and practices that are easier to use by nonexperts. For now, bitcoin users can use many of the tips discussed here to enjoy a secure and trouble-free bitcoin experience.

== Blockchain Applications

Let’s now build on our understanding of bitcoin by looking at it as an *application platform*. Nowadays, many people use the term "blockchain" to refer to any application platform that shares the design principles of bitcoin. The term is often misused and applied to many things that fail to deliver the primary features that bitcoin’s blockchain delivers.

In this chapter we will look at the features offered by the Bitcoin blockchain, as an application platform. We will consider the application building *primitives*, which form the building blocks of any blockchain application. We will look at several important applications that use these primitives, such as payment (state) channels and routed payment channels (Lightning Network).

=== Introduction

The Bitcoin system was designed as a decentralized currency and payment system. However, most of its functionality is derived from much lower-level constructs that can be used for much broader applications. Bitcoin wasn’t built with components such as accounts, users, balances, and payments. Instead, it uses a transactional scripting language with low-level cryptographic functions, as we saw in [simpara\_title](#transactions). Just as the higher-level concepts of accounts, balances, and payments can be derived from these basic primitives, so can many other complex applications. Thus, the Bitcoin blockchain can become an application platform offering trust services to applications, such as smart contracts, far surpassing the original purpose of digital currency and payments.

=== Building Blocks (Primitives)

When operating correctly and over the long term, the Bitcoin system offers certain guarantees, which can be used as building blocks to create applications. These include:

No Double-Spend

The most fundamental guarantee of bitcoin’s decentralized consensus algorithm ensures that no UTXO can be spent twice.

Immutability

Once a transaction is recorded in the blockchain and sufficient work has been added with subsequent blocks, the transaction’s data becomes immutable. Immutability is underwritten by energy, as rewriting the blockchain requires the expenditure of energy to produce Proof-of-Work. The energy required and therefore the degree of immutability increases with the amount of work committed on top of the block containing a transaction.

Neutrality

The decentralized Bitcoin network propagates valid transactions regardless of the origin or content of those transactions. This means that anyone can create a valid transaction with sufficient fees and trust they will be able to transmit that transaction and have it included in the blockchain at any time.

Secure Timestamping

The consensus rules reject any block whose timestamp is too far in the past or future. This ensures that timestamps on blocks can be trusted. The timestamp on a block implies an unspent-before guarantee for the inputs of all included transactions.

Authorization

Digital signatures, validated in a decentralized network, offer authorization guarantees. Scripts that contain a requirement for a digital signature cannot be executed without authorization by the holder of the private key implied in the script.

Auditability

All transactions are public and can be audited. All transactions and blocks can be linked back in an unbroken chain to the genesis block.

Accounting

In any transaction (except the coinbase transaction) the value of inputs is equal to the value of outputs plus fees. It is not possible to create or destroy bitcoin value in a transaction. The outputs cannot exceed the inputs.

Nonexpiration

A valid transaction does not expire. If it is valid today, it will be valid in the near future, as long as the inputs remain unspent and the consensus rules do not change.

Integrity

A bitcoin transaction signed with SIGHASH\_ALL or parts of a transaction signed by another SIGHASH type cannot be modified without invalidating the signature, thus invalidating the transaction itself.

Transaction Atomicity

Bitcoin transactions are atomic. They are either valid and confirmed (mined) or not. Partial transactions cannot be mined and there is no interim state for a transaction. At any point in time a transaction is either mined, or not.

Discrete (Indivisible) Units of Value

Transaction outputs are discrete and indivisible units of value. They can either be spent or unspent, in full. They cannot be divided or partially spent.

Quorum of Control

Multisignature constraints in scripts impose a quorum of authorization, predefined in the multisignature scheme. The M-of-N requirement is enforced by the consensus rules.

Timelock/Aging

Any script clause containing a relative or absolute timelock can only be executed after its age exceeds the time specified.

Replication

The decentralized storage of the blockchain ensures that when a transaction is mined, after sufficient confirmations, it is replicated across the network and becomes durable and resilient to power loss, data loss, etc.

Forgery Protection

A transaction can only spend existing, validated outputs. It is not possible to create or counterfeit value.

Consistency

In the absence of miner partitions, blocks that are recorded in the blockchain are subject to reorganization or disagreement with exponentially decreasing likelihood, based on the depth at which they are recorded. Once deeply recorded, the computation and energy required to change makes change practically infeasible.

Recording External State

A transaction can commit a data value, via OP\_RETURN, representing a state transition in an external state machine.

Predictable Issuance

Less than 21 million bitcoin will be issued, at a predictable rate.

The list of building blocks is not complete and more are added with each new feature introduced into bitcoin.

=== Applications from Building Blocks

The building blocks offered by bitcoin are elements of a trust platform that can be used to compose applications. Here are some examples of applications that exist today and the building blocks they use:

Proof-of-Existence (Digital Notary)

Immutability + Timestamp + Durability. A digital fingerprint can be committed with a transaction to the blockchain, proving that a document existed (Timestamp) at the time it was recorded. The fingerprint cannot be modified ex-post-facto (Immutability) and the proof will be stored permanently (Durability).

Kickstarter (Lighthouse)

Consistency + Atomicity + Integrity. If you sign one input and the output (Integrity) of a fundraiser transaction, others can contribute to the fundraiser but it cannot be spent (Atomicity) until the goal (output value) is funded (Consistency).

Payment Channels

Quorum of Control + Timelock + No Double Spend + Nonexpiration + Censorship Resistance + Authorization. A multisig 2-of-2 (Quorum) with a timelock (Timelock) used as the "settlement" transaction of a payment channel can be held (Nonexpiration) and spent at any time (Censorship Resistance) by either party (Authorization). The two parties can then create commitment transactions that double-spend (No Double-Spend) the settlement on a shorter timelock (Timelock).

=== Counterparty

Counterparty is a protocol layer built on top of bitcoin. The Counterparty protocol offers the ability to create and trade virtual assets and tokens. In addition, Counterparty offers a decentralized exchange for assets. Counterparty is also implementing smart contracts, based on the Ethereum Virtual Machine (EVM).

Counterparty embeds metadata in bitcoin transactions, using the OP\_RETURN opcode or 1-of-N multisignature addresses that encode metadata in the place of public keys. Using these mechanisms, Counterparty implements a protocol layer encoded in bitcoin transactions. The additional protocol layer can be interpreted by applications that are Counterparty-aware, such as wallets and blockchain explorers, or any application built using the Counterparty libraries.

Counterparty can be used as a platform for other applications and services, in turn. For example, Tokenly is a platform built on top of Counterparty that allows content creators, artists, and companies to issue tokens that express digital ownership and can be used to rent, access, trade, or shop for content, products, and services. Other applications leveraging Counterparty include games (Spells of Genesis) and grid computing projects (Folding Coin).

More details about Counterparty can be found at <https://counterparty.io>. The open source project can be found at <https://github.com/CounterpartyXCP>.

=== Payment Channels and State Channels

*Payment channels* are a trustless mechanism for exchanging bitcoin transactions between two parties, outside of the Bitcoin blockchain. These transactions, which would be valid if settled on the Bitcoin blockchain, are held off-chain instead, acting as *promissory notes* for eventual batch settlement. Because the transactions are not settled, they can be exchanged without the usual settlement latency, allowing extremely high transaction throughput, low (submillisecond) latency, and fine (satoshi-level) granularity.

Actually, the term *channel* is a metaphor. State channels are virtual constructs represented by the exchange of state between two parties, outside of the blockchain. There are no "channels" per se and the underlying data transport mechanism is not the channel. We use the term channel to represent the relationship and shared state between two parties, outside of the blockchain.

To further explain this concept, think of a TCP stream. From the perspective of higher-level protocols it is a "socket" connecting two applications across the internet. But if you look at the network traffic, a TCP stream is just a virtual channel over IP packets. Each endpoint of the TCP stream sequences and assembles IP packets to create the illusion of a stream of bytes. Underneath, it’s all disconnected packets. Similarly, a payment channel is just a series of transactions. If properly sequenced and connected, they create redeemable obligations that you can trust even though you don’t trust the other side of the channel.

In this section we will look at various forms of payment channels. First, we will examine the mechanisms used to construct a one-way (unidirectional) payment channel for a metered micropayment service, such as streaming video. Then, we will expand on this mechanism and introduce bidirectional payment channels. Finally, we will look at how bidirectional channels can be connected end-to-end to form multihop channels in a routed network, first proposed under the name *Lightning Network*.

Payment channels are part of the broader concept of a *state channel*, which represents an off-chain alteration of state, secured by eventual settlement in a blockchain. A payment channel is a state channel where the state being altered is the balance of a virtual currency.

==== State Channels—Basic Concepts and Terminology

A state channel is established between two parties, through a transaction that locks a shared state on the blockchain. This is called the *funding transaction* or *anchor transaction*. This single transaction must be transmitted to the network and mined to establish the channel. In the example of a payment channel, the locked state is the initial balance (in currency) of the channel.

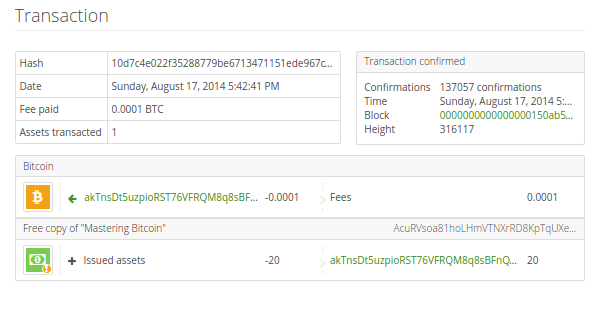
The two parties then exchange signed transactions, called *commitment transactions*, that alter the initial state. These transactions are valid transactions in that they *could* be submitted for settlement by either party, but instead are held off-chain by each party pending the channel closure. State updates can be created as fast as each party can create, sign, and transmit a transaction to the other party. In practice this means that thousands of transactions per second can be exchanged.

When exchanging commitment transactions the two parties also invalidate the previous states, so that the most up-to-date commitment transaction is always the only one that can be redeemed. This prevents either party from cheating by unilaterally closing the channel with an expired prior state that is more favorable to them than the current state. We will examine the various mechanisms that can be used to invalidate prior state in the rest of this chapter.

Finally, the channel can be closed either cooperatively, by submitting a final *settlement transaction* to the blockchain, or unilaterally, by either party submitting the last commitment transaction to the blockchain. A unilateral close option is needed in case one of the parties unexpectedly disconnects. The settlement transaction represents the final state of the channel and is settled on the blockchain.

In the entire lifetime of the channel, only two transactions need to be submitted for mining on the blockchain: the funding and settlement transactions. In between these two states, the two parties can exchange any number of commitment transactions that are never seen by anyone else, nor submitted to the blockchain.

[figure\_title](#payment_channel) illustrates a payment channel between Bob and Alice, showing the funding, commitment, and settlement transactions.

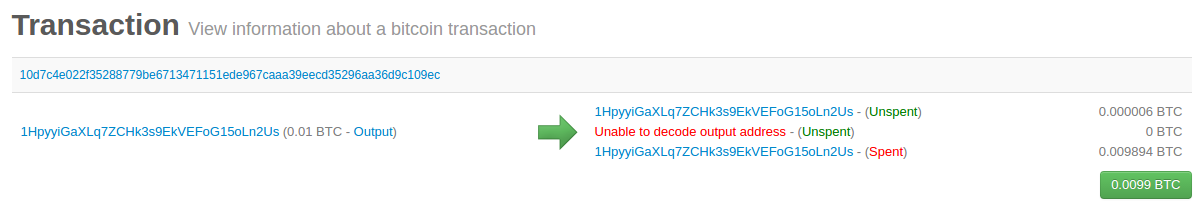


A payment channel between Bob and Alice, showing the funding, commitment, and settlement transactions

==== Simple Payment Channel Example

To explain state channels, we start with a very simple example. We demonstrate a one-way channel, meaning that value is flowing in one direction only. We will also start with the naive assumption that no one is trying to cheat, to keep things simple. Once we have the basic channel idea explained, we will then look at what it takes to make it trustless so that neither party *can* cheat, even if they are trying to.

For this example we will assume two participants: Emma and Fabian. Fabian offers a video streaming service that is billed by the second using a micropayment channel. Fabian charges 0.01 millibit (0.00001 BTC) per second of video, equivalent to 36 millibits (0.036 BTC) per hour of video. Emma is a user who purchases this streaming video service from Fabian. [figure\_title](#emma_fabian_streaming_video) shows Emma buying the video streaming service from Fabian using a payment channel.



Emma purchases streaming video from Fabian with a payment channel, paying for each second of video

In this example, Fabian and Emma are using special software that handles both the payment channel and the video streaming. Emma is running the software in her browser, Fabian is running it on a server. The software includes basic bitcoin wallet functionality and can create and sign bitcoin transactions. Both the concept and the term "payment channel" are completely hidden from the users. What they see is video that is paid for by the second.

To set up the payment channel, Emma and Fabian establish a 2-of-2 multisignature address, with each of them holding one of the keys. From Emma’s perspective, the software in her browser presents a QR code with a P2SH address (starting with "3"), and asks her to submit a "deposit" for up to 1 hour of video. The address is then funded by Emma. Emma’s transaction, paying to the multisignature address, is the funding or anchor transaction for the payment channel.

For this example, let’s say that Emma funds the channel with 36 millibits (0.036 BTC). This will allow Emma to consume *up to* 1 hour of streaming video. The funding transaction in this case sets the maximum amount that can be transmitted in this channel, setting the *channel capacity*.

The funding transaction consumes one or more inputs from Emma’s wallet, sourcing the funds. It creates one output with a value of 36 millibits paid to the multisignature 2-of-2 address controlled jointly between Emma and Fabian. It may have additional outputs for change back to Emma’s wallet.

Once the funding transaction is confirmed, Emma can start streaming video. Emma’s software creates and signs a commitment transaction that changes the channel balance to credit 0.01 millibit to Fabian’s address and refund 35.99 millibits back to Emma. The transaction signed by Emma consumes the 36 millibits output created by the funding transaction and creates two outputs: one for her refund, the other for Fabian’s payment. The transaction is only partially signed—it requires two signatures (2-of-2), but only has Emma’s signature. When Fabian’s server receives this transaction, it adds the second signature (for the 2-of-2 input) and returns it to Emma together with 1 second worth of video. Now both parties have a fully signed commitment transaction that either can redeem, representing the correct up-to-date balance of the channel. Neither party broadcasts this transaction to the network.

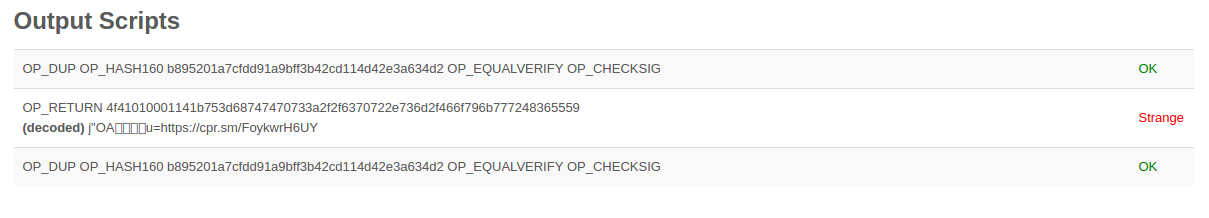
In the next round, Emma’s software creates and signs another commitment transaction (commitment #2) that consumes the *same* 2-of-2 output from the funding transaction. The second commitment transaction allocates one output of 0.02 millibits to Fabian’s address and one output of 35.98 millibits back to Emma’s address. This new transaction is payment for two cumulative seconds of video. Fabian’s software signs and returns the second commitment transaction, together with another second of video.

In this way, Emma’s software continues to send commitment transactions to Fabian’s server in exchange for streaming video. The balance of the channel gradually accumulates in favor of Fabian, as Emma consumes more seconds of video. Let’s say Emma watches 600 seconds (10 minutes) of video, creating and signing 600 commitment transactions. The last commitment transaction (#600) will have two outputs, splitting the balance of the channel, 6 millibits to Fabian and 30 millibits to Emma.

Finally, Emma selects "Stop" to stop streaming video. Either Fabian or Emma can now transmit the final state transaction for settlement. This last transaction is the *settlement transaction* and pays Fabian for all the video Emma consumed, refunding the remainder of the funding transaction to Emma.

[figure\_title](#video_payment_channel) shows the channel between Emma and Fabian and the commitment transactions that update the balance of the channel.

In the end, only two transactions are recorded on the blockchain: the funding transaction that established the channel and a settlement transaction that allocated the final balance correctly between the two participants.



Emma’s payment channel with Fabian, showing the commitment transactions that update the balance of the channel

==== Making Trustless Channels

The channel we just described works, but only if both parties cooperate, without any failures or attempts to cheat. Let’s look at some of the scenarios that break this channel and see what is needed to fix those:

* Once the funding transaction happens, Emma needs Fabian’s signature to get any money back. If Fabian disappears, Emma’s funds are locked in a 2-of-2 and effectively lost. This channel, as constructed, leads to a loss of funds if one of the parties disconnects before there is at least one commitment transaction signed by both parties.
* While the channel is running, Emma can take any of the commitment transactions Fabian has countersigned and transmit one to the blockchain. Why pay for 600 seconds of video, if she can transmit commitment transaction #1 and only pay for 1 second of video? The channel fails because Emma can cheat by broadcasting a prior commitment that is in her favor.

Both of these problems can be solved with timelocks—let’s look at how we could use transaction-level timelocks (nLocktime).

Emma cannot risk funding a 2-of-2 multisig unless she has a guaranteed refund. To solve this problem, Emma constructs the funding and refund transactions at the same time. She signs the funding transaction but doesn’t transmit it to anyone. Emma transmits only the refund transaction to Fabian and obtains his signature.

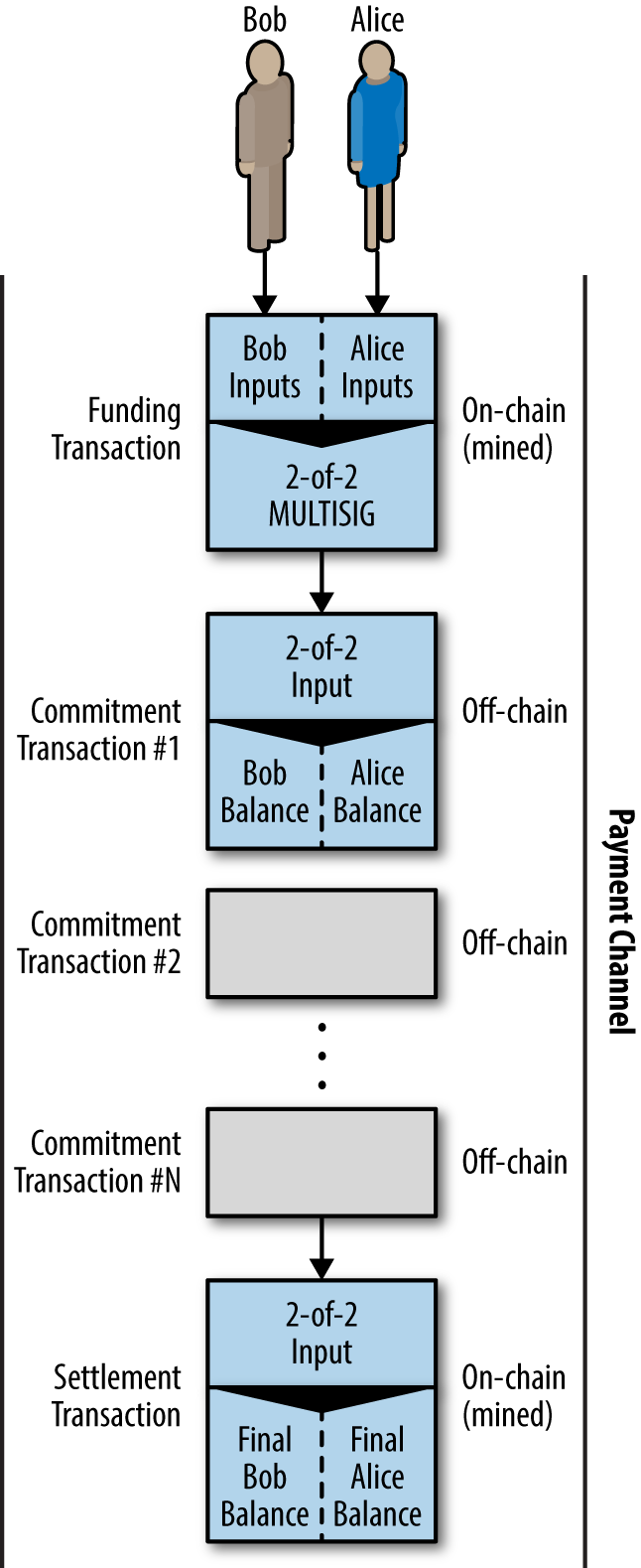
The refund transaction acts as the first commitment transaction and its timelock establishes the upper bound for the channel’s life. In this case, Emma could set the nLocktime to 30 days or 4320 blocks into the future. All subsequent commitment transactions must have a shorter timelock, so that they can be redeemed before the refund transaction.

Now that Emma has a fully signed refund transaction, she can confidently transmit the signed funding transaction knowing that she can eventually, after the timelock expires, redeem the refund transaction even if Fabian disappears.

Every commitment transaction the parties exchange during the life of the channel will be timelocked into the future. But the delay will be slightly shorter for each commitment so the most recent commitment can be redeemed before the prior commitment it invalidates. Because of the nLockTime, neither party can successfully propagate any of the commitment transactions until their timelock expires. If all goes well, they will cooperate and close the channel gracefully with a settlement transaction, making it unnecessary to transmit an intermediate commitment transaction. If not, the most recent commitment transaction can be propagated to settle the account and invalidate all prior commitment transactions.

For example, if commitment transaction #1 is timelocked to 4320 blocks in the future, then commitment transaction #2 is timelocked to 4319 blocks in the future. Commitment transaction #600 can be spent 600 blocks before commitment transaction #1 becomes valid.

[figure\_title](#timelocked_commitments) shows each commitment transaction setting a shorter timelock, allowing it to be spent before the previous commitments become valid.



Each commitment sets a shorter timelock, allowing it to be spent before the previous commitments become valid

Each subsequent commitment transaction must have a shorter timelock so that it may be broadcast before its predecessors and before the refund transaction. The ability to broadcast a commitment earlier ensures it will be able to spend the funding output and preclude any other commitment transaction from being redeemed by spending the output. The guarantees offered by the Bitcoin blockchain, preventing double-spends and enforcing timelocks, effectively allow each commitment transaction to invalidate its predecessors.

State channels use timelocks to enforce smart contracts across a time dimension. In this example we saw how the time dimension guarantees that the most recent commitment transaction becomes valid before any earlier commitments. Thus, the most recent commitment transaction can be transmitted, spending the inputs and invalidating prior commitment transactions. The enforcement of smart contracts with absolute timelocks protects against cheating by one of the parties. This implementation needs nothing more than absolute transaction-level timelocks (nLocktime). Next, we will see how script-level timelocks, CHECKLOCKTIMEVERIFY and CHECKSEQUENCEVERIFY, can be used to construct more flexible, useful, and sophisticated state channels.

The first form of unidirectional payment channel was demonstrated as a prototype video streaming application in 2015 by an Argentinian team of developers.

Timelocks are not the only way to invalidate prior commitment transactions. In the next sections we will see how a revocation key can be used to achieve the same result. Timelocks are effective but they have two distinct disadvantages. By establishing a maximum timelock when the channel is first opened, they limit the lifetime of the channel. Worse, they force channel implementations to strike a balance between allowing long-lived channels and forcing one of the participants to wait a very long time for a refund in case of premature closure. For example, if you allow the channel to remain open for 30 days, by setting the refund timelock to 30 days, if one of the parties disappears immediately the other party must wait 30 days for a refund. The more distant the endpoint, the more distant the refund.

The second problem is that since each subsequent commitment transaction must decrement the timelock, there is an explicit limit on the number of commitment transactions that can be exchanged between the parties. For example, a 30-day channel, setting a timelock of 4320 blocks into the future, can only accommodate 4320 intermediate commitment transactions before it must be closed. There is a danger in setting the timelock commitment transaction interval at 1 block. By setting the timelock interval between commitment transactions to 1 block, a developer is creating a very high burden for the channel participants who have to be vigilant, remain online and watching, and be ready to transmit the right commitment transaction at any time.

Now that we understand how timelocks can be used to invalidate prior commitments, we can see the difference between closing the channel cooperatively and closing it unilaterally by broadcasting a commitment transaction. All commitment transactions are timelocked, therefore broadcasting a commitment transaction will always involve waiting until the timelock has expired. But if the two parties agree on what the final balance is and know they both hold commitment transactions that will eventually make that balance a reality, they can construct a settlement transaction without a timelock representing that same balance. In a cooperative close, either party takes the most recent commitment transaction and builds a settlement transaction that is identical in every way except that it omits the timelock. Both parties can sign this settlement transaction knowing there is no way to cheat and get a more favorable balance. By cooperatively signing and transmitting the settlement transaction they can close the channel and redeem their balance immediately. Worst case, one of the parties can be petty, refuse to cooperate, and force the other party to do a unilateral close with the most recent commitment transaction. But if they do that, they have to wait for their funds too.

==== Asymmetric Revocable Commitments

A better way to handle the prior commitment states is to explicitly revoke them. However, this is not easy to achieve. A key characteristic of bitcoin is that once a transaction is valid, it remains valid and does not expire. The only way to cancel a transaction is by double-spending its inputs with another transaction before it is mined. That’s why we used timelocks in the simple payment channel example above to ensure that more recent commitments could be spent before older commitments were valid. However, sequencing commitments in time creates a number of constraints that make payment channels difficult to use.

Even though a transaction cannot be canceled, it can be constructed in such a way as to make it undesirable to use. The way we do that is by giving each party a *revocation key* that can be used to punish the other party if they try to cheat. This mechanism for revoking prior commitment transactions was first proposed as part of the Lightning Network.

To explain revocation keys, we will construct a more complex payment channel between two exchanges run by Hitesh and Irene. Hitesh and Irene run bitcoin exchanges in India and the USA, respectively. Customers of Hitesh’s Indian exchange often send payments to customers of Irene’s USA exchange and vice versa. Currently, these transactions occur on the Bitcoin blockchain, but this means paying fees and waiting several blocks for confirmations. Setting up a payment channel between the exchanges will significantly reduce the cost and accelerate the transaction flow.

Hitesh and Irene start the channel by collaboratively constructing a funding transaction, each funding the channel with 5 bitcoin. The initial balance is 5 bitcoin for Hitesh and 5 bitcoin for Irene. The funding transaction locks the channel state in a 2-of-2 multisig, just like in the example of a simple channel.

The funding transaction may have one or more inputs from Hitesh (adding up to 5 bitcoin or more), and one or more inputs from Irene (adding up to 5 bitcoin or more). The inputs have to slightly exceed the channel capacity in order to cover the transaction fees. The transaction has one output that locks the 10 total bitcoin to a 2-of-2 multisig address controlled by both Hitesh and Irene. The funding transaction may also have one or more outputs returning change to Hitesh and Irene if their inputs exceeded their intended channel contribution. This is a single transaction with inputs offered and signed by two parties. It has to be constructed in collaboration and signed by each party before it is transmitted.

Now, instead of creating a single commitment transaction that both parties sign, Hitesh and Irene create two different commitment transactions that are *asymmetric*.

Hitesh has a commitment transaction with two outputs. The first output pays Irene the 5 bitcoin she is owed *immediately*. The second output pays Hitesh the 5 bitcoin he is owed, but only after a timelock of 1000 blocks. The transaction outputs look like this:

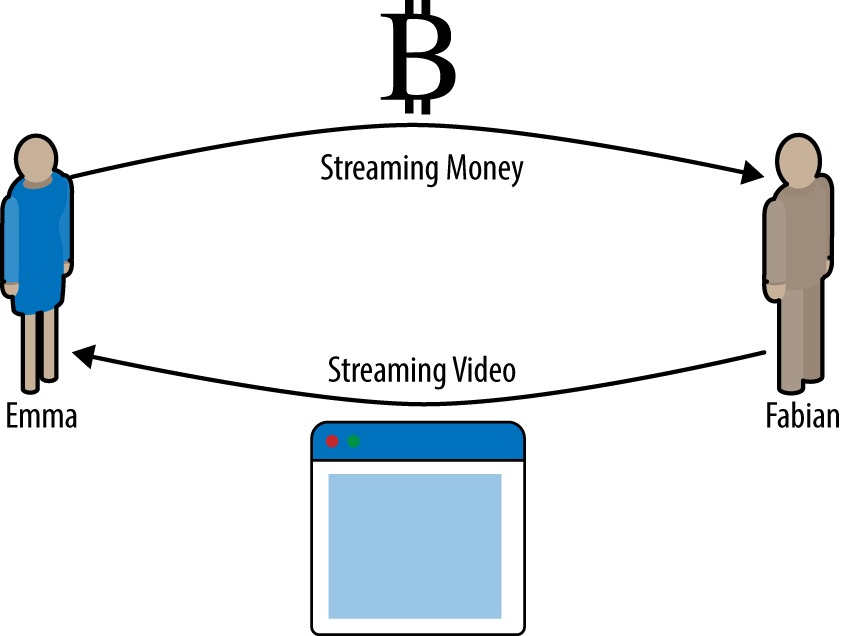
Input: 2-of-2 funding output, signed by Irene  
  
Output 0 <5 bitcoin>:  
 <Irene's Public Key> CHECKSIG  
  
Output 1 <5 bitcoin>:  
 <1000 blocks>  
 CHECKSEQUENCEVERIFY  
 DROP  
 <Hitesh's Public Key> CHECKSIG

Irene has a different commitment transaction with two outputs. The first output pays Hitesh the 5 bitcoin he is owed immediately. The second output pays Irene the 5 bitcoin she is owed but only after a timelock of 1000 blocks. The commitment transaction Irene holds (signed by Hitesh) looks like this:

Input: 2-of-2 funding output, signed by Hitesh  
  
Output 0 <5 bitcoin>:  
 <Hitesh's Public Key> CHECKSIG  
  
Output 1 <5 bitcoin>:  
 <1000 blocks>  
 CHECKSEQUENCEVERIFY  
 DROP  
 <Irene's Public Key> CHECKSIG

This way, each party has a commitment transaction, spending the 2-of-2 funding output. This input is signed by the *other* party. At any time the party holding the transaction can also sign (completing the 2-of-2) and broadcast. However, if they broadcast the commitment transaction, it pays the other party immediately whereas they have to wait for a timelock to expire. By imposing a delay on the redemption of one of the outputs, we put each party at a slight disadvantage when they choose to unilaterally broadcast a commitment transaction. But a time delay alone isn’t enough to encourage fair conduct.

[figure\_title](#asymmetric_commitments) shows two asymmetric commitment transactions, where the output paying the holder of the commitment is delayed.



Two asymmetric commitment transactions with delayed payment for the party holding the transaction

Now we introduce the final element of this scheme: a revocation key that prevents a cheater from broadcasting an expired commitment. The revocation key allows the wronged party to punish the cheater by taking the entire balance of the channel.

The revocation key is composed of two secrets, each half generated independently by each channel participant. It is similar to a 2-of-2 multisig, but constructed using elliptic curve arithmetic, so that both parties know the revocation public key but each party knows only half the revocation secret key.

In each round, both parties reveal their half of the revocation secret to the other party, thereby giving the other party (who now has both halves) the means to claim the penalty output if this revoked transaction is ever broadcast.

Each of the commitment transactions has a "delayed" output. The redemption script for that output allows one party to redeem it after 1000 blocks, *or* the other party to redeem it if they have a revocation key, penalizing transmission of a revoked commitment.

So when Hitesh creates a commitment transaction for Irene to sign, he makes the second output payable to himself after 1000 blocks, or to the revocation public key (of which he only knows half the secret). Hitesh constructs this transaction. He will only reveal his half of the revocation secret to Irene when he is ready to move to a new channel state and wants to revoke this commitment.

The second output’s script looks like this:

Output 0 <5 bitcoin>:  
 <Irene's Public Key> CHECKSIG  
  
Output 1 <5 bitcoin>:  
IF  
 # Revocation penalty output  
 <Revocation Public Key>  
ELSE  
 <1000 blocks>  
 CHECKSEQUENCEVERIFY  
 DROP  
 <Hitesh's Public Key>  
ENDIF  
CHECKSIG

Irene can confidently sign this transaction, since if transmitted it will immediately pay her what she is owed. Hitesh holds the transaction, but knows that if he transmits it in a unilateral channel closing, he will have to wait 1000 blocks to get paid.

When the channel is advanced to the next state, Hitesh has to *revoke* this commitment transaction before Irene agrees to sign the next commitment transaction. To do that, all he has to do is send his half of the *revocation key* to Irene. Once Irene has both halves of the revocation secret key for this commitment, she can sign the next commitment with confidence. She knows that if Hitesh tries to cheat by publishing the prior commitment, she can use the revocation key to redeem Hitesh’s delayed output. *If Hitesh cheats, Irene gets BOTH outputs*. Meanwhile, Hitesh only has half the revocation secret for that revocation public key and can’t redeem the output until 1000 blocks. Irene will be able to redeem the output and punish Hitesh before the 1000 blocks have elapsed.

The revocation protocol is bilateral, meaning that in each round, as the channel state is advanced, the two parties exchange new commitments, exchange revocation secrets for the previous commitments, and sign each other’s new commitment transactions. As they accept a new state, they make the prior state impossible to use, by giving each other the necessary revocation secrets to punish any cheating.

Let’s look at an example of how it works. One of Irene’s customers wants to send 2 bitcoin to one of Hitesh’s customers. To transmit 2 bitcoin across the channel, Hitesh and Irene must advance the channel state to reflect the new balance. They will commit to a new state (state number 2) where the channel’s 10 bitcoin are split, 7 bitcoin to Hitesh and 3 bitcoin to Irene. To advance the state of the channel, they will each create new commitment transactions reflecting the new channel balance.

As before, these commitment transactions are asymmetric so that the commitment transaction each party holds forces them to wait if they redeem it. Crucially, before signing new commitment transactions, they must first exchange revocation keys to invalidate the prior commitment. In this particular case, Hitesh’s interests are aligned with the real state of the channel and therefore he has no reason to broadcast a prior state. However, for Irene, state number 1 leaves her with a higher balance than state 2. When Irene gives Hitesh the revocation key for her prior commitment transaction (state number 1) she is effectively revoking her ability to profit from regressing the channel to a prior state because with the revocation key, Hitesh can redeem both outputs of the prior commitment transaction without delay. Meaning if Irene broadcasts the prior state, Hitesh can exercise his right to take all of the outputs.

Importantly, the revocation doesn’t happen automatically. While Hitesh has the ability to punish Irene for cheating, he has to watch the blockchain diligently for signs of cheating. If he sees a prior commitment transaction broadcast, he has 1000 blocks to take action and use the revocation key to thwart Irene’s cheating and punish her by taking the entire balance, all 10 bitcoin.

Asymmetric revocable commitments with relative time locks (CSV) are a much better way to implement payment channels and a very significant innovation in this technology. With this construct, the channel can remain open indefinitely and can have billions of intermediate commitment transactions. In prototype implementations of Lightning Network, the commitment state is identified by a 48-bit index, allowing more than 281 trillion (2.8 x 1014) state transitions in any single channel!

==== Hash Time Lock Contracts (HTLC)

Payment channels can be further extended with a special type of smart contract that allows the participants to commit funds to a redeemable secret, with an expiration time. This feature is called a *Hash Time Lock Contract*, or *HTLC*, and is used in both bidirectional and routed payment channels.

Let’s first explain the "hash" part of the HTLC. To create an HTLC, the intended recipient of the payment will first create a secret R. They then calculate the hash of this secret H:

H = Hash(R)

This produces a hash H that can be included in an output’s locking script. Whoever knows the secret can use it to redeem the output. The secret R is also referred to as a *preimage* to the hash function. The preimage is just the data that is used as input to a hash function.

The second part of an HTLC is the "time lock" component. If the secret is not revealed, the payer of the HTLC can get a "refund" after some time. This is achieved with an absolute time lock using CHECKLOCKTIMEVERIFY.

The script implementing an HTLC might look like this:

IF  
 # Payment if you have the secret R  
 HASH160 <H> EQUALVERIFY  
ELSE  
 # Refund after timeout.  
 <locktime> CHECKLOCKTIMEVERIFY DROP  
 <Payer Public Key> CHECKSIG  
ENDIF

Anyone who knows the secret R, which when hashed equals to H, can redeem this output by exercising the first clause of the IF flow.

If the secret is not revealed and the HTLC claimed, after a certain number of blocks the payer can claim a refund using the second clause in the IF flow.

This is a basic implementation of an HTLC. This type of HTLC can be redeemed by *anyone* who has the secret R. An HTLC can take many different forms with slight variations to the script. For example, adding a CHECKSIG operator and a public key in the first clause restricts redemption of the hash to a named recipient, who must also know the secret R.

=== Routed Payment Channels (Lightning Network)

The Lightning Network is a proposed routed network of bidirectional payment channels connected end-to-end. A network like this can allow any participant to route a payment from channel to channel without trusting any of the intermediaries. The Lightning Network was [first described by Joseph Poon and Thadeus Dryja in February 2015](https://lightning.network/lightning-network-paper.pdf), building on the concept of payment channels as proposed and elaborated upon by many others.

"Lightning Network" refers to a specific design for a routed payment channel network, which has now been implemented by at least five different open source teams. The independent implementations are coordinated by a set of interoperability standards described in the [*Basics of Lightning Technology (BOLT)* paper](https://bit.ly/2rBHeoL).

Prototype implementations of the Lightning Network have been released by several teams.

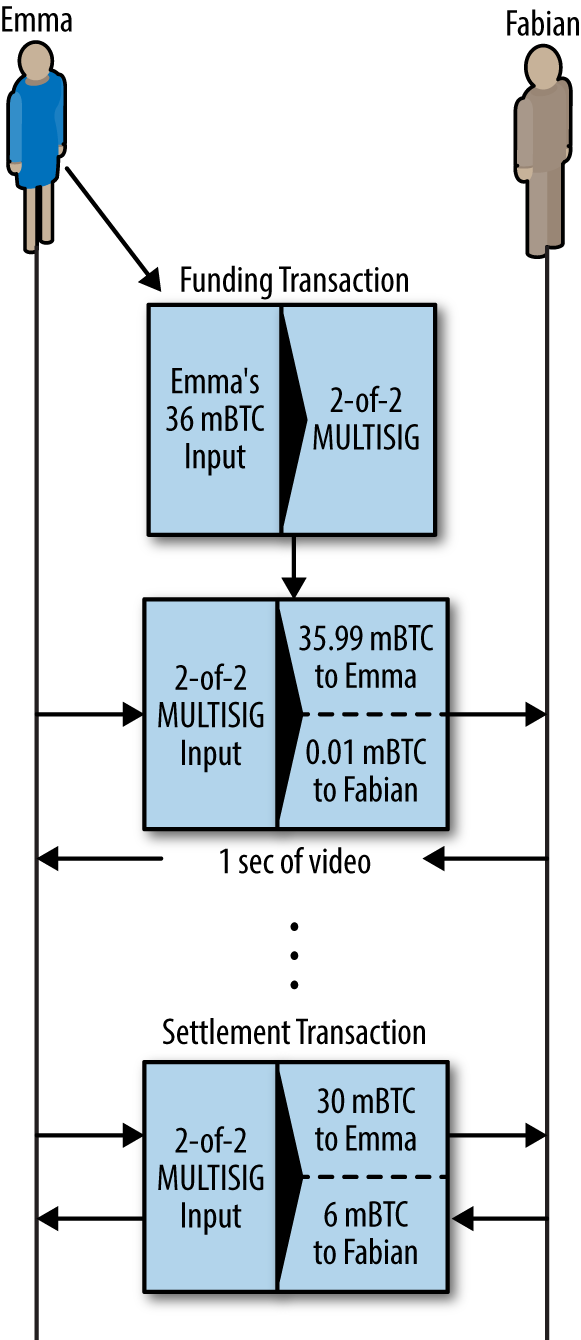
The Lightning Network is one possible way of implementing routed payment channels. There are several other designs that aim to achieve similar goals, such as Teechan and Tumblebit.

==== Basic Lightning Network Example

Let’s see how this works.

In this example, we have five participants: Alice, Bob, Carol, Diana, and Eric. These five participants have opened payment channels with each other, in pairs. Alice has a payment channel with Bob. Bob is connected to Carol, Carol to Diana, and Diana to Eric. For simplicity let’s assume each channel is funded with 2 bitcoin by each participant, for a total capacity of 4 bitcoin in each channel.

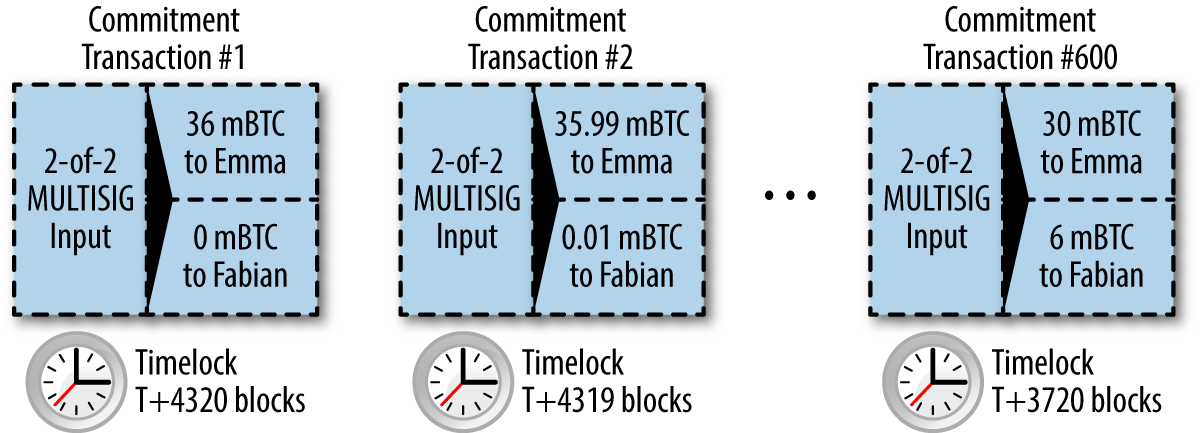
[figure\_title](#lightning_network_fig) shows five participants in a Lightning Network, connected by bidirectional payment channels that can be linked to make a payment from Alice to Eric ([simpara\_title](#lightning_network)).



A series of bidirectional payment channels linked to form a Lightning Network that can route a payment from Alice to Eric

Alice wants to pay Eric 1 bitcoin. However, Alice is not connected to Eric by a payment channel. Creating a payment channel requires a funding transaction, which must be committed to the Bitcoin blockchain. Alice does not want to open a new payment channel and commit more of her funds. Is there a way to pay Eric, indirectly?

[figure\_title](#ln_payment_process) shows the step-by-step process of routing a payment from Alice to Eric, through a series of HTLC commitments on the payment channels connecting the participants.



Step-by-step payment routing through a Lightning Network

Alice is running a Lightning Network (LN) node that is keeping track of her payment channel to Bob and has the ability to discover routes between payment channels. Alice’s LN node also has the ability to connect over the internet to Eric’s LN node. Eric’s LN node creates a secret R using a random number generator. Eric’s node does not reveal this secret to anyone. Instead, Eric’s node calculates a hash H of the secret R and transmits this hash to Alice’s node (see [figure\_title](#ln_payment_process) step 1).

Now Alice’s LN node constructs a route between Alice’s LN node and Eric’s LN node. The routing algorithm used will be examined in more detail later, but for now let’s assume that Alice’s node can find an efficient route.

Alice’s node then constructs an HTLC, payable to the hash H, with a 10-block refund timeout (current block + 10), for an amount of 1.003 bitcoin (see [figure\_title](#ln_payment_process) step 2). The extra 0.003 will be used to compensate the intermediate nodes for their participation in this payment route. Alice offers this HTLC to Bob, deducting 1.003 bitcoin from her channel balance with Bob and committing it to the HTLC. The HTLC has the following meaning: *"Alice is committing 1.003 of her channel balance to be paid to Bob if Bob knows the secret, or refunded back to Alice’s balance if 10 blocks elapse."* The channel balance between Alice and Bob is now expressed by commitment transactions with three outputs: 2 bitcoin balance to Bob, 0.997 bitcoin balance to Alice, 1.003 bitcoin committed in Alice’s HTLC. Alice’s balance is reduced by the amount committed to the HTLC.

Bob now has a commitment that if he is able to get the secret R within the next 10 blocks, he can claim the 1.003 locked by Alice. With this commitment in hand, Bob’s node constructs an HTLC on his payment channel with Carol. Bob’s HTLC commits 1.002 bitcoin to hash H for 9 blocks, which Carol can redeem if she has secret R (see [figure\_title](#ln_payment_process) step 3). Bob knows that if Carol can claim his HTLC, she has to produce R. If Bob has R in nine blocks, he can use it to claim Alice’s HTLC to him. He also makes 0.001 bitcoin for committing his channel balance for nine blocks. If Carol is unable to claim his HTLC and he is unable to claim Alice’s HTLC, everything reverts back to the prior channel balances and no one is at a loss. The channel balance between Bob and Carol is now: 2 to Carol, 0.998 to Bob, 1.002 committed by Bob to the HTLC.

Carol now has a commitment that if she gets R within the next nine blocks, she can claim 1.002 bitcoin locked by Bob. Now she can make an HTLC commitment on her channel with Diana. She commits an HTLC of 1.001 bitcoin to hash H, for eight blocks, which Diana can redeem if she has secret R (see [figure\_title](#ln_payment_process) step 4). From Carol’s perspective, if this works she is 0.001 bitcoin better off and if it doesn’t she loses nothing. Her HTLC to Diana is only viable if R is revealed, at which point she can claim the HTLC from Bob. The channel balance between Carol and Diana is now: 2 to Diana, 0.999 to Carol, 1.001 committed by Carol to the HTLC.

Finally, Diana can offer an HTLC to Eric, committing 1 bitcoin for seven blocks to hash H (see [figure\_title](#ln_payment_process) step 5). The channel balance between Diana and Eric is now: 2 to Eric, 1 to Diana, 1 committed by Diana to the HTLC.

However, at this hop in the route, Eric *has* secret R. He can therefore claim the HTLC offered by Diana. He sends R to Diana and claims the 1 bitcoin, adding it to his channel balance (see [figure\_title](#ln_payment_process) step 6). The channel balance is now: 1 to Diana, 3 to Eric.

Now, Diana has secret R. Therefore, she can now claim the HTLC from Carol. Diana transmits R to Carol and adds the 1.001 bitcoin to her channel balance (see [figure\_title](#ln_payment_process) step 7). Now the channel balance between Carol and Diana is: 0.999 to Carol, 3.001 to Diana. Diana has "earned" 0.001 for participating in this payment route.

Flowing back through the route, the secret R allows each participant to claim the outstanding HTLCs. Carol claims 1.002 from Bob, setting the balance on their channel to: 0.998 to Bob, 3.002 to Carol (see [figure\_title](#ln_payment_process) step 8). Finally, Bob claims the HTLC from Alice (see [figure\_title](#ln_payment_process) step 9). Their channel balance is updated as: 0.997 to Alice, 3.003 to Bob.

Alice has paid Eric 1 bitcoin without opening a channel to Eric. None of the intermediate parties in the payment route had to trust each other. For the short-term commitment of their funds in the channel they are able to earn a small fee, with the only risk being a small delay in refund if the channel was closed or the routed payment failed.

==== Lightning Network Transport and Routing

All communications between LN nodes are encrypted point-to-point. In addition, nodes have a long-term public key that they use as an identifier and to authenticate each other.

Whenever a node wishes to send a payment to another node, it must first construct a *path* through the network by connecting payment channels with sufficient capacity. Nodes advertise routing information, including what channels they have open, how much capacity each channel has, and what fees they charge to route payments. The routing information can be shared in a variety of ways and different routing protocols are likely to emerge as Lightning Network technology advances. Some Lightning Network implementations use the IRC protocol as a convenient mechanism for nodes to announce routing information. Another implementation of route discovery uses a P2P model where nodes propagate channel announcements to their peers, in a "flooding" model, similar to how bitcoin propagates transactions. Future plans include a proposal called [Flare](https://bit.ly/2r5TACm), which is a hybrid routing model with local node "neighborhoods" and longer-range beacon nodes.

In our previous example, Alice’s node uses one of these route discovery mechanisms to find one or more paths connecting her node to Eric’s node. Once Alice’s node has constructed a path, she will initialize that path through the network, by propagating a series of encrypted and nested instructions to connect each of the adjacent payment channels.

Importantly, this path is only known to Alice’s node. All other participants in the payment route see only the adjacent nodes. From Carol’s perspective, this looks like a payment from Bob to Diana. Carol does not know that Bob is actually relaying a payment from Alice. She also doesn’t know that Diana will be relaying a payment to Eric.

This is a critical feature of the Lightning Network, because it ensures privacy of payments and makes it very difficult to apply surveillance, censorship, or blacklists. But how does Alice establish this payment path, without revealing anything to the intermediary nodes?

The Lightning Network implements an onion-routed protocol based on a scheme called [Sphinx](https://bit.ly/2q6ZDrP). This routing protocol ensures that a payment sender can construct and communicate a path through the Lightning Network such that:

* Intermediate nodes can verify and decrypt their portion of route information and find the next hop.
* Other than the previous and next hops, they cannot learn about any other nodes that are part of the path.
* They cannot identify the length of the payment path, or their own position in that path.
* Each part of the path is encrypted in such a way that a network-level attacker cannot associate the packets from different parts of the path to each other.
* Unlike Tor (an onion-routed anonymization protocol on the internet), there are no "exit nodes" that can be placed under surveillance. The payments do not need to be transmitted to the Bitcoin blockchain; the nodes just update channel balances.

Using this onion-routed protocol, Alice wraps each element of the path in a layer of encryption, starting with the end and working backward. She encrypts a message to Eric with Eric’s public key. This message is wrapped in a message encrypted to Diana, identifying Eric as the next recipient. The message to Diana is wrapped in a message encrypted to Carol’s public key and identifying Diana as the next recipient. The message to Carol is encrypted to Bob’s key. Thus, Alice has constructed this encrypted multilayer "onion" of messages. She sends this to Bob, who can only decrypt and unwrap the outer layer. Inside, Bob finds a message addressed to Carol that he can forward to Carol but cannot decipher himself. Following the path, the messages get forwarded, decrypted, forwarded, etc., all the way to Eric. Each participant knows only the previous and next node in each hop.

Each element of the path contains information on the HTLC that must be extended to the next hop, the amount that is being sent, the fee to include, and the CLTV locktime (in blocks) expiration of the HTLC. As the route information propagates, the nodes make HTLC commitments forward to the next hop.

At this point, you might be wondering how it is possible that the nodes do not know the length of the path and their position in that path. After all, they receive a message and forward it to the next hop. Doesn’t it get shorter, allowing them to deduce the path size and their position? To prevent this, the path is always fixed at 20 hops and padded with random data. Each node sees the next hop and a fixed-length encrypted message to forward. Only the final recipient sees that there is no next hop. To everyone else it seems as if there are always 20 more hops to go.

==== Lightning Network Benefits

A Lightning Network is a second-layer routing technology. It can be applied to any blockchain that supports some basic capabilities, such as multisignature transactions, timelocks, and basic smart contracts.

If a Lightning Network is layered on top of the Bitcoin network, the Bitcoin network can gain a significant increase in capacity, privacy, granularity, and speed, without sacrificing the principles of trustless operation without intermediaries:

Privacy

Lightning Network payments are much more private than payments on the Bitcoin blockchain, as they are not public. While participants in a route can see payments propagated across their channels, they do not know the sender or recipient.

Fungibility

A Lightning Network makes it much more difficult to apply surveillance and blacklists on bitcoin, increasing the fungibility of the currency.

Speed

Bitcoin transactions using Lightning Network are settled in milliseconds, rather than minutes, as HTLCs are cleared without committing transactions to a block.

Granularity

A Lightning Network can enable payments at least as small as the bitcoin "dust" limit, perhaps even smaller. Some proposals allow for subsatoshi increments.

Capacity

A Lightning Network increases the capacity of the Bitcoin system by several orders of magnitude. There is no practical upper bound to the number of payments per second that can be routed over a Lightning Network, as it depends only on the capacity and speed of each node.

Trustless Operation

A Lightning Network uses bitcoin transactions between nodes that operate as peers without trusting each other. Thus, a Lightning Network preserves the principles of the Bitcoin system, while expanding its operating parameters significantly.

Of course, as mentioned previously, the Lightning Network protocol is not the only way to implement routed payment channels. Other proposed systems include Tumblebit and Teechan. At this time, however, the Lightning Network has already been deployed on testnet. Several different teams have developed competing implementations of LN and are working toward a common interoperability standard (called BOLT). It is likely that Lightning Network will be the first routed payment channel network to be deployed in production.

=== Conclusion

We have examined just a few of the emerging applications that can be built using the Bitcoin blockchain as a trust platform. These applications expand the scope of bitcoin beyond payments and beyond financial instruments, to encompass many other applications where trust is critical. By decentralizing the basis of trust, the Bitcoin blockchain is a platform that will spawn many revolutionary applications in a wide variety of industries.

Unresolved directive in book.asciidoc - include::for\_translation\_mastering-bitcoin\_appdx-bitcoinwhitepape.asciidoc[]

== Transaction Script Language Operators, Constants, and Symbols

Tables and descriptions sourced from <https://en.bitcoin.it/wiki/Script>.

[table\_title](#tx_script_ops_table_pushdata) shows operators for pushing values onto the stack.

Push value onto stack

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_0 or OP\_FALSE | 0x00 | An empty array is pushed onto the stack |
| 1 to 75 | 0x01 to 0x4b | Push the next N bytes onto the stack, where N is 1 to 75 bytes |
| OP\_PUSHDATA1 | 0x4c | The next script byte contains N, push the following N bytes onto the stack |
| OP\_PUSHDATA2 | 0x4d | The next two script bytes contain N, push the following N bytes onto the stack |
| OP\_PUSHDATA4 | 0x4e | The next four script bytes contain N, push the following N bytes onto the stack |
| OP\_1NEGATE | 0x4f | Push the value "–1" onto the stack |
| OP\_RESERVED | 0x50 | Halt—Invalid transaction unless found in an unexecuted OP\_IF clause |
| OP\_1 or OP\_TRUE | 0x51 | Push the value "1" onto the stack |
| OP\_2 to OP\_16 | 0x52 to 0x60 | For OP\_N, push the value "N" onto the stack, e.g., OP\_2 pushes "2" |

[table\_title](#tx_script_ops_table_control) shows conditional flow control operators.

Conditional flow control

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_NOP | 0x61 | Do nothing |
| OP\_VER | 0x62 | Halt—Invalid transaction unless found in an unexecuted OP\_IF clause |
| OP\_IF | 0x63 | Execute the statements following if top of stack is not 0 |
| OP\_NOTIF | 0x64 | Execute the statements following if top of stack is 0 |
| OP\_VERIF | 0x65 | Halt—Invalid transaction |
| OP\_VERNOTIF | 0x66 | Halt—Invalid transaction |
| OP\_ELSE | 0x67 | Execute only if the previous statements were not executed |
| OP\_ENDIF | 0x68 | End the OP\_IF, OP\_NOTIF, OP\_ELSE block |
| OP\_VERIFY | 0x69 | Check the top of the stack, halt and invalidate transaction if not TRUE |
| OP\_RETURN | 0x6a | Halt and invalidate transaction |

[table\_title](#tx_script_ops_table_timelock) shows operators used for timelocks.

Timelock operations

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_CHECKLOCKTIMEVERIFY (previously OP\_NOP2) | 0xb1 | Marks transaction as invalid if the top stack item is greater than the transaction’s nLockTime field, otherwise script evaluation continues as though an OP\_NOP was executed. Transaction is also invalid if 1. the stack is empty; or 2. the top stack item is negative; or 3. the top stack item is greater than or equal to 500000000 while the transaction’s nLockTime field is less than 500000000, or vice versa; or 4. the input’s nSequence field is equal to 0xffffffff. The precise semantics are described in BIP-65 |
| OP\_CHECKSEQUENCEVERIFY (previously OP\_NOP3) | 0xb2 | Marks transaction as invalid if the relative lock time of the input (enforced by BIP 0068 with nSequence) is not equal to or longer than the value of the top stack item. The precise semantics are described in BIP-112 |

[table\_title](#tx_script_ops_table_stack) shows operators used to manipulate the stack.

Stack operations

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_TOALTSTACK | 0x6b | Pop top item from stack and push to alternative stack |
| OP\_FROMALTSTACK | 0x6c | Pop top item from alternative stack and push to stack |
| OP\_2DROP | 0x6d | Pop top two stack items |
| OP\_2DUP | 0x6e | Duplicate top two stack items |
| OP\_3DUP | 0x6f | Duplicate top three stack items |
| OP\_2OVER | 0x70 | Copy the third and fourth items in the stack to the top |
| OP\_2ROT | 0x71 | Move the fifth and sixth items in the stack to the top |
| OP\_2SWAP | 0x72 | Swap the two top pairs of items in the stack |
| OP\_IFDUP | 0x73 | Duplicate the top item in the stack if it is not 0 |
| OP\_DEPTH | 0x74 | Count the items on the stack and push the resulting count |
| OP\_DROP | 0x75 | Pop the top item in the stack |
| OP\_DUP | 0x76 | Duplicate the top item in the stack |
| OP\_NIP | 0x77 | Pop the second item in the stack |
| OP\_OVER | 0x78 | Copy the second item in the stack and push it onto the top |
| OP\_PICK | 0x79 | Pop value N from top, then copy the Nth item to the top of the stack |
| OP\_ROLL | 0x7a | Pop value N from top, then move the Nth item to the top of the stack |
| OP\_ROT | 0x7b | Rotate the top three items in the stack |
| OP\_SWAP | 0x7c | Swap the top two items on the stack |
| OP\_TUCK | 0x7d | Copy the top item and insert it between the top and second item |

[table\_title](#tx_script_ops_table_splice) shows string operators.

String splice operations

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| *OP\_CAT* | 0x7e | Disabled (concatenates top two items) |
| *OP\_SUBSTR* | 0x7f | Disabled (returns substring) |
| *OP\_LEFT* | 0x80 | Disabled (returns left substring) |
| *OP\_RIGHT* | 0x81 | Disabled (returns right substring) |
| OP\_SIZE | 0x82 | Calculate string length of top item and push the result |

[table\_title](#tx_script_ops_table_binmath) shows binary arithmetic and boolean logic operators.

Binary arithmetic and conditionals

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| *OP\_INVERT* | 0x83 | Disabled (Flip the bits of the top item) |
| *OP\_AND* | 0x84 | Disabled (Boolean AND of two top items) |
| *OP\_OR* | 0x85 | Disabled (Boolean OR of two top items) |
| *OP\_XOR* | 0x86 | Disabled (Boolean XOR of two top items) |
| OP\_EQUAL | 0x87 | Push TRUE (1) if top two items are exactly equal, push FALSE (0) otherwise |
| OP\_EQUALVERIFY | 0x88 | Same as OP\_EQUAL, but run OP\_VERIFY after to halt if not TRUE |
| OP\_RESERVED1 | 0x89 | Halt—Invalid transaction unless found in an unexecuted OP\_IF clause |
| OP\_RESERVED2 | 0x8a | Halt—Invalid transaction unless found in an unexecuted OP\_IF clause |

[table\_title](#tx_script_ops_table_numbers) shows numeric (arithmetic) operators.

Numeric operators

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_1ADD | 0x8b | Add 1 to the top item |
| OP\_1SUB | 0x8c | Subtract 1 from the top item |
| *OP\_2MUL* | 0x8d | Disabled (multiply top item by 2) |
| *OP\_2DIV* | 0x8e | Disabled (divide top item by 2) |
| OP\_NEGATE | 0x8f | Flip the sign of top item |
| OP\_ABS | 0x90 | Change the sign of the top item to positive |
| OP\_NOT | 0x91 | If top item is 0 or 1 Boolean flip it, otherwise return 0 |
| OP\_0NOTEQUAL | 0x92 | If top item is 0 return 0, otherwise return 1 |
| OP\_ADD | 0x93 | Pop top two items, add them and push result |
| OP\_SUB | 0x94 | Pop top two items, subtract first from second, push result |
| *OP\_MUL* | 0x95 | Disabled (multiply top two items) |
| *OP\_DIV* | 0x96 | Disabled (divide second item by first item) |
| *OP\_MOD* | 0x97 | Disabled (remainder divide second item by first item) |
| *OP\_LSHIFT* | 0x98 | Disabled (shift second item left by first item number of bits) |
| *OP\_RSHIFT* | 0x99 | Disabled (shift second item right by first item number of bits) |
| OP\_BOOLAND | 0x9a | Boolean AND of top two items |
| OP\_BOOLOR | 0x9b | Boolean OR of top two items |
| OP\_NUMEQUAL | 0x9c | Return TRUE if top two items are equal numbers |
| OP\_NUMEQUALVERIFY | 0x9d | Same as NUMEQUAL, then OP\_VERIFY to halt if not TRUE |
| OP\_NUMNOTEQUAL | 0x9e | Return TRUE if top two items are not equal numbers |
| OP\_LESSTHAN | 0x9f | Return TRUE if second item is less than top item |
| OP\_GREATERTHAN | 0xa0 | Return TRUE if second item is greater than top item |
| OP\_LESSTHANOREQUAL | 0xa1 | Return TRUE if second item is less than or equal to top item |
| OP\_GREATERTHANOREQUAL | 0xa2 | Return TRUE if second item is great than or equal to top item |
| OP\_MIN | 0xa3 | Return the smaller of the two top items |
| OP\_MAX | 0xa4 | Return the larger of the two top items |
| OP\_WITHIN | 0xa5 | Return TRUE if the third item is between the second item (or equal) and first item |

[table\_title](#tx_script_ops_table_crypto) shows cryptographic function operators.

Cryptographic and hashing operations

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_RIPEMD160 | 0xa6 | Return RIPEMD160 hash of top item |
| OP\_SHA1 | 0xa7 | Return SHA1 hash of top item |
| OP\_SHA256 | 0xa8 | Return SHA256 hash of top item |
| OP\_HASH160 | 0xa9 | Return RIPEMD160(SHA256(x)) hash of top item |
| OP\_HASH256 | 0xaa | Return SHA256(SHA256(x)) hash of top item |
| OP\_CODESEPARATOR | 0xab | Mark the beginning of signature-checked data |
| OP\_CHECKSIG | 0xac | Pop a public key and signature and validate the signature for the transaction’s hashed data, return TRUE if matching |
| OP\_CHECKSIGVERIFY | 0xad | Same as CHECKSIG, then OP\_VERIFY to halt if not TRUE |
| OP\_CHECKMULTISIG | 0xae | Run CHECKSIG for each pair of signature and public key provided. All must match. Bug in implementation pops an extra value, prefix with OP\_0 as workaround |
| OP\_CHECKMULTISIGVERIFY | 0xaf | Same as CHECKMULTISIG, then OP\_VERIFY to halt if not TRUE |

[table\_title](#tx_script_ops_table_nop) shows nonoperator symbols.

Nonoperators

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_NOP1 to OP\_NOP10 | 0xb0 to 0xb9 | Does nothing, ignored |

[table\_title](#tx_script_ops_table_internal) shows operator codes reserved for use by the internal script parser.

Reserved OP codes for internal use by the parser

| Symbol | Value (hex) | Description |
| --- | --- | --- |
| OP\_SMALLDATA | 0xf9 | Represents small data field |
| OP\_SMALLINTEGER | 0xfa | Represents small integer data field |
| OP\_PUBKEYS | 0xfb | Represents public key fields |
| OP\_PUBKEYHASH | 0xfd | Represents a public key hash field |
| OP\_PUBKEY | 0xfe | Represents a public key field |
| OP\_INVALIDOPCODE | 0xff | Represents any OP code not currently assigned |

== Bitcoin Improvement Proposals (ou Propositions d’améliorations du Bitcoin)

Les Bitcoin Improvement Proposals (ou Propositions d’améliorations du Bitcoin) sont des documents de conception fournissant des informations à la communauté bitcoin ou décrivant une nouvelle fonctionnalité pour bitcoin ou ses processus ou son environnement.

Conformément à BIP-01 *BIP Purpose and Guidelines*, il existe trois types de BIP :

*Standard* BIP

Décrit tout changement qui affecte la plupart ou toutes les implémentations de bitcoin, comme un changement du protocole réseau, un changement des règles de validité de bloc ou de transaction, ou tout changement ou ajout qui affecte l'interopérabilité des applications utilisant bitcoin.

*Informational* BIP

Décrit un problème de conception de bitcoin, ou fournit des directives générales ou des informations à la communauté bitcoin, mais ne propose pas de nouvelle fonctionnalité. Les BIP informationnels ne représentent pas nécessairement un consensus ou une recommandation de la communauté bitcoin, de sorte que les utilisateurs et les implémenteurs peuvent ignorer les BIP informationnels ou suivre leurs conseils.

*Process* BIP

Décrit un processus bitcoin ou propose une modification (ou un événement dans) un processus. Les BIP de processus ressemblent aux BIP standard mais s'appliquent à des domaines autres que le protocole Bitcoin lui-même. Ils pourraient proposer une implémentation, mais pas à la base de code de Bitcoin ; elles nécessitent souvent un consensus communautaire ; et contrairement aux BIP informatifs, ils sont plus que des recommandations, et les utilisateurs ne sont généralement pas libres de les ignorer. Les exemples incluent les procédures, les directives, les modifications du processus de prise de décision et les modifications des outils ou de l'environnement utilisés dans le développement de Bitcoin. Tout méta-BIP est également considéré comme un processus BIP.

Les BIP sont enregistrés dans un référentiel versionné sur GitHub : <https://github.com/bitcoin/bips> . [table\_title](#table_d-1) montre un instantané des BIP en avril 2017. Consultez le référentiel faisant autorité pour obtenir des informations à jour sur les BIP existants et leur contenu.("propositions de progressions Bitcoin", "copie de", id="BIPsnap15" )

Liste des BIPs

| BIP# | Titre | Propriétaire | Type | Statut |
| --- | --- | --- | --- | --- |
| [BIP-1](https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki) | BIP Purpose and Guidelines | Amir Taaki | Process | Replaced |
| [BIP-2](https://github.com/bitcoin/bips/blob/master/bip-0002.mediawiki) | BIP process, revised | Luke Dashjr | Process | Active |
| [BIP-8](https://github.com/bitcoin/bips/blob/master/bip-0008.mediawiki) | Version bits with guaranteed lock-in | Shaolin Fry | Informational | Draft |
| [BIP-9](https://github.com/bitcoin/bips/blob/master/bip-0009.mediawiki) | Version bits with timeout and delay | Pieter Wuille, Peter Todd, Greg Maxwell, Rusty Russell | Informational | Final |
| [BIP-10](https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki) | Multi-Sig Transaction Distribution | Alan Reiner | Informational | Withdrawn |
| [BIP-11](https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki) | M-of-N Standard Transactions | Gavin Andresen | Standard | Final |
| [BIP-12](https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki) | OP\_EVAL | Gavin Andresen | Standard | Withdrawn |
| [BIP-13](https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki) | Address Format for pay-to-script-hash | Gavin Andresen | Standard | Final |
| [BIP-14](https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki) | Protocol Version and User Agent | Amir Taaki, Patrick Strateman | Standard | Final |
| [BIP-15](https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki) | Aliases | Amir Taaki | Standard | Deferred |
| [BIP-16](https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki) | Pay to Script Hash | Gavin Andresen | Standard | Final |
| [BIP-17](https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki) | OP\_CHECKHASHVERIFY (CHV) | Luke Dashjr | Standard | Withdrawn |
| [BIP-18](https://github.com/bitcoin/bips/blob/master/bip-0018.mediawiki) | hashScriptCheck | Luke Dashjr | Standard | Proposed |
| [BIP-19](https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki) | M-of-N Standard Transactions (Low SigOp) | Luke Dashjr | Standard | Draft |
| [BIP-20](https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki) | URI Scheme | Luke Dashjr | Standard | Replaced |
| [BIP-21](https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki) | URI Scheme | Nils Schneider, Matt Corallo | Standard | Final |
| [BIP-22](https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki) | getblocktemplate - Fundamentals | Luke Dashjr | Standard | Final |
| [BIP-23](https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki) | getblocktemplate - Pooled Mining | Luke Dashjr | Standard | Final |
| [BIP-30](https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki) | Duplicate transactions | Pieter Wuille | Standard | Final |
| [BIP-31](https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki) | Pong message | Mike Hearn | Standard | Final |
| [BIP-32](https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki) | Hierarchical Deterministic Wallets | Pieter Wuille | Informational | Final |
| [BIP-33](https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki) | Stratized Nodes | Amir Taaki | Standard | Draft |
| [BIP-34](https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki) | Block v2, Height in Coinbase | Gavin Andresen | Standard | Final |
| [BIP-35](https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki) | mempool message | Jeff Garzik | Standard | Final |
| [BIP-36](https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki) | Custom Services | Stefan Thomas | Standard | Draft |
| [BIP-37](https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki) | Connection Bloom filtering | Mike Hearn, Matt Corallo | Standard | Final |
| [BIP-39](https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki) | Mnemonic code for generating deterministic keys | Marek Palatinus, Pavol Rusnak, Aaron Voisine, Sean Bowe | Standard | Proposed |
| [BIP-40](https://github.com/bitcoin/bips/blob/master/bip-0040.mediawiki) | Stratum wire protocol | Marek Palatinus | Standard | BIP number allocated |
| [BIP-41](https://github.com/bitcoin/bips/blob/master/bip-0041.mediawiki) | Stratum mining protocol | Marek Palatinus | Standard | BIP number allocated |
| [BIP-42](https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki) | A finite monetary supply for Bitcoin | Pieter Wuille | Standard | Draft |
| [BIP-43](https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki) | Purpose Field for Deterministic Wallets | Marek Palatinus, Pavol Rusnak | Informational | Draft |
| [BIP-44](https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki) | Multi-Account Hierarchy for Deterministic Wallets | Marek Palatinus, Pavol Rusnak | Standard | Proposed |
| [BIP-45](https://github.com/bitcoin/bips/blob/master/bip-0045.mediawiki) | Structure for Deterministic P2SH Multisignature Wallets | Manuel Araoz, Ryan X. Charles, Matias Alejo Garcia | Standard | Proposed |
| [BIP-47](https://github.com/bitcoin/bips/blob/master/bip-0047.mediawiki) | Reusable Payment Codes for Hierarchical Deterministic Wallets | Justus Ranvier | Informational | Draft |
| [BIP-49](https://github.com/bitcoin/bips/blob/master/bip-0049.mediawiki) | Derivation scheme for P2WPKH-nested-in-P2SH based accounts | Daniel Weigl | Informational | Draft |
| [BIP-50](https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki) | March 2013 Chain Fork Post-Mortem | Gavin Andresen | Informational | Final |
| [BIP-60](https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki) | Fixed Length "version" Message (Relay-Transactions Field) | Amir Taaki | Standard | Draft |
| [BIP-61](https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki) | Reject P2P message | Gavin Andresen | Standard | Final |
| [BIP-62](https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki) | Dealing with malleability | Pieter Wuille | Standard | Withdrawn |
| [BIP-63](https://github.com/bitcoin/bips/blob/master/bip-0063.mediawiki) | Stealth Addresses | Peter Todd | Standard | BIP number allocated |
| [BIP-64](https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki) | getutxo message | Mike Hearn | Standard | Draft |
| [BIP-65](https://github.com/bitcoin/bips/blob/master/bip-0065.mediawiki) | OP\_CHECKLOCKTIMEVERIFY | Peter Todd | Standard | Final |
| [BIP-66](https://github.com/bitcoin/bips/blob/master/bip-0066.mediawiki) | Strict DER signatures | Pieter Wuille | Standard | Final |
| [BIP-67](https://github.com/bitcoin/bips/blob/master/bip-0067.mediawiki) | Deterministic Pay-to-script-hash multi-signature addresses through public key sorting | Thomas Kerin, Jean-Pierre Rupp, Ruben de Vries | Standard | Proposed |
| [BIP-68](https://github.com/bitcoin/bips/blob/master/bip-0068.mediawiki) | Relative lock-time using consensus-enforced sequence numbers | Mark Friedenbach, BtcDrak, Nicolas Dorier, kinoshitajona | Standard | Final |
| [BIP-69](https://github.com/bitcoin/bips/blob/master/bip-0069.mediawiki) | Lexicographical Indexing of Transaction Inputs and Outputs | Kristov Atlas | Informational | Proposed |
| [BIP-70](https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki) | Payment Protocol | Gavin Andresen, Mike Hearn | Standard | Final |
| [BIP-71](https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki) | Payment Protocol MIME types | Gavin Andresen | Standard | Final |
| [BIP-72](https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki) | bitcoin: uri extensions for Payment Protocol | Gavin Andresen | Standard | Final |
| [BIP-73](https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki) | Use "Accept" header for response type negotiation with Payment Request URLs | Stephen Pair | Standard | Final |
| [BIP-74](https://github.com/bitcoin/bips/blob/master/bip-0074.mediawiki) | Allow zero value OP\_RETURN in Payment Protocol | Toby Padilla | Standard | Draft |
| [BIP-75](https://github.com/bitcoin/bips/blob/master/bip-0075.mediawiki) | Out of Band Address Exchange using Payment Protocol Encryption | Justin Newton, Matt David, Aaron Voisine, James MacWhyte | Standard | Draft |
| [BIP-80](https://github.com/bitcoin/bips/blob/master/bip-0080.mediawiki) | Hierarchy for Non-Colored Voting Pool Deterministic Multisig Wallets | Justus Ranvier, Jimmy Song | Informational | Deferred |
| [BIP-81](https://github.com/bitcoin/bips/blob/master/bip-0081.mediawiki) | Hierarchy for Colored Voting Pool Deterministic Multisig Wallets | Justus Ranvier, Jimmy Song | Informational | Deferred |
| [BIP-83](https://github.com/bitcoin/bips/blob/master/bip-0083.mediawiki) | Dynamic Hierarchical Deterministic Key Trees | Eric Lombrozo | Standard | Draft |
| [BIP-90](https://github.com/bitcoin/bips/blob/master/bip-0090.mediawiki) | Buried Deployments | Suhas Daftuar | Informational | Draft |
| [BIP-99](https://github.com/bitcoin/bips/blob/master/bip-0099.mediawiki) | Motivation and deployment of consensus rule changes ([soft/hard]forks) | Jorge Timón | Informational | Draft |
| [BIP-101](https://github.com/bitcoin/bips/blob/master/bip-0101.mediawiki) | Increase maximum block size | Gavin Andresen | Standard | Withdrawn |
| [BIP-102](https://github.com/bitcoin/bips/blob/master/bip-0102.mediawiki) | Block size increase to 2MB | Jeff Garzik | Standard | Draft |
| [BIP-103](https://github.com/bitcoin/bips/blob/master/bip-0103.mediawiki) | Block size following technological growth | Pieter Wuille | Standard | Draft |
| [BIP-104](https://github.com/bitcoin/bips/blob/master/bip-0104.mediawiki) | 'Block75' - Max block size like difficulty | t.khan | Standard | Draft |
| [BIP-105](https://github.com/bitcoin/bips/blob/master/bip-0105.mediawiki) | Consensus based block size retargeting algorithm | BtcDrak | Standard | Draft |
| [BIP-106](https://github.com/bitcoin/bips/blob/master/bip-0106.mediawiki) | Dynamically Controlled Bitcoin Block Size Max Cap | Upal Chakraborty | Standard | Draft |
| [BIP-107](https://github.com/bitcoin/bips/blob/master/bip-0107.mediawiki) | Dynamic limit on the block size | Washington Y. Sanchez | Standard | Draft |
| [BIP-109](https://github.com/bitcoin/bips/blob/master/bip-0109.mediawiki) | Two million byte size limit with sigop and sighash limits | Gavin Andresen | Standard | Rejected |
| [BIP-111](https://github.com/bitcoin/bips/blob/master/bip-0111.mediawiki) | NODE\_BLOOM service bit | Matt Corallo, Peter Todd | Standard | Proposed |
| [BIP-112](https://github.com/bitcoin/bips/blob/master/bip-0112.mediawiki) | CHECKSEQUENCEVERIFY | BtcDrak, Mark Friedenbach, Eric Lombrozo | Standard | Final |
| [BIP-113](https://github.com/bitcoin/bips/blob/master/bip-0113.mediawiki) | Median time-past as endpoint for lock-time calculations | Thomas Kerin, Mark Friedenbach | Standard | Final |
| [BIP-114](https://github.com/bitcoin/bips/blob/master/bip-0114.mediawiki) | Merkelized Abstract Syntax Tree | Johnson Lau | Standard | Draft |
| [BIP-120](https://github.com/bitcoin/bips/blob/master/bip-0120.mediawiki) | Proof of Payment | Kalle Rosenbaum | Standard | Draft |
| [BIP-121](https://github.com/bitcoin/bips/blob/master/bip-0121.mediawiki) | Proof of Payment URI scheme | Kalle Rosenbaum | Standard | Draft |
| [BIP-122](https://github.com/bitcoin/bips/blob/master/bip-0122.mediawiki) | URI scheme for Blockchain references / exploration | Marco Pontello | Standard | Draft |
| [BIP-123](https://github.com/bitcoin/bips/blob/master/bip-0123.mediawiki) | BIP Classification | Eric Lombrozo | Process | Active |
| [BIP-124](https://github.com/bitcoin/bips/blob/master/bip-0124.mediawiki) | Hierarchical Deterministic Script Templates | Eric Lombrozo, William Swanson | Informational | Draft |
| [BIP-125](https://github.com/bitcoin/bips/blob/master/bip-0125.mediawiki) | Opt-in Full Replace-by-Fee Signaling | David A. Harding, Peter Todd | Standard | Proposed |
| [BIP-126](https://github.com/bitcoin/bips/blob/master/bip-0126.mediawiki) | Best Practices for Heterogeneous Input Script Transactions | Kristov Atlas | Informational | Draft |
| [BIP-130](https://github.com/bitcoin/bips/blob/master/bip-0130.mediawiki) | sendheaders message | Suhas Daftuar | Standard | Proposed |
| [BIP-131](https://github.com/bitcoin/bips/blob/master/bip-0131.mediawiki) | "Coalescing Transaction" Specification (wildcard inputs) | Chris Priest | Standard | Draft |
| [BIP-132](https://github.com/bitcoin/bips/blob/master/bip-0132.mediawiki) | Committee-based BIP Acceptance Process | Andy Chase | Process | Withdrawn |
| [BIP-133](https://github.com/bitcoin/bips/blob/master/bip-0133.mediawiki) | feefilter message | Alex Morcos | Standard | Draft |
| [BIP-134](https://github.com/bitcoin/bips/blob/master/bip-0134.mediawiki) | Flexible Transactions | Tom Zander | Standard | Draft |
| [BIP-140](https://github.com/bitcoin/bips/blob/master/bip-0140.mediawiki) | Normalized TXID | Christian Decker | Standard | Draft |
| [BIP-141](https://github.com/bitcoin/bips/blob/master/bip-0141.mediawiki) | Segregated Witness (Consensus layer) | Eric Lombrozo, Johnson Lau, Pieter Wuille | Standard | Draft |
| [BIP-142](https://github.com/bitcoin/bips/blob/master/bip-0142.mediawiki) | Address Format for Segregated Witness | Johnson Lau | Standard | Deferred |
| [BIP-143](https://github.com/bitcoin/bips/blob/master/bip-0143.mediawiki) | Transaction Signature Verification for Version 0 Witness Program | Johnson Lau, Pieter Wuille | Standard | Draft |
| [BIP-144](https://github.com/bitcoin/bips/blob/master/bip-0144.mediawiki) | Segregated Witness (Peer Services) | Eric Lombrozo, Pieter Wuille | Standard | Draft |
| [BIP-145](https://github.com/bitcoin/bips/blob/master/bip-0145.mediawiki) | getblocktemplate Updates for Segregated Witness | Luke Dashjr | Standard | Draft |
| [BIP-146](https://github.com/bitcoin/bips/blob/master/bip-0146.mediawiki) | Dealing with signature encoding malleability | Johnson Lau, Pieter Wuille | Standard | Draft |
| [BIP-147](https://github.com/bitcoin/bips/blob/master/bip-0147.mediawiki) | Dealing with dummy stack element malleability | Johnson Lau | Standard | Draft |
| [BIP-148](https://github.com/bitcoin/bips/blob/master/bip-0148.mediawiki) | Mandatory activation of segwit deployment | Shaolin Fry | Standard | Draft |
| [BIP-150](https://github.com/bitcoin/bips/blob/master/bip-0150.mediawiki) | Peer Authentication | Jonas Schnelli | Standard | Draft |
| [BIP-151](https://github.com/bitcoin/bips/blob/master/bip-0151.mediawiki) | Peer-to-Peer Communication Encryption | Jonas Schnelli | Standard | Draft |
| [BIP-152](https://github.com/bitcoin/bips/blob/master/bip-0152.mediawiki) | Compact Block Relay | Matt Corallo | Standard | Draft |
| [BIP-171](https://github.com/bitcoin/bips/blob/master/bip-0171.mediawiki) | Currency/exchange rate information API | Luke Dashjr | Standard | Draft |
| [BIP-180](https://github.com/bitcoin/bips/blob/master/bip-0180.mediawiki) | Block size/weight fraud proof | Luke Dashjr | Standard | Draft |
| [BIP-199](https://github.com/bitcoin/bips/blob/master/bip-0199.mediawiki) | Hashed Time-Locked Contract transactions | Sean Bowe, Daira Hopwood | Standard | Draft |

include::for\_translation\_mastering-bitcoin\_appdx-bitcor.asciidoc[]\_fr\_CA.txt

== pycoin, ku, and tx

The Python library [pycoin](https://github.com/richardkiss/pycoin), originally written and maintained by Richard Kiss, is a Python-based library that supports manipulation of bitcoin keys and transactions, even supporting the scripting language enough to properly deal with nonstandard transactions.

The pycoin library supports both Python 2 (2.7.x) and Python 3 (3.3 and later) and comes with some handy command-line utilities, ku and tx.

=== Key Utility (KU)

The command-line utility ku ("key utility") is a Swiss Army knife for manipulating keys. It supports BIP-32 keys, WIF, and addresses (bitcoin and alt coins). Following are some examples.

Create a BIP-32 key using the default entropy sources of GPG and */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 : 76460638240546478364843397478278468101877117767873462127021560368290114016034  
public pair y : 59807879657469774102040120298272207730921291736633247737077406753676825777701  
 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

Create a BIP-32 key from a passphrase:

The passphrase in this example is way too easy to guess.

$ 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 : 81821982719381104061777349269130419024493616650993589394553404347774393168191  
public pair y : 58994218069605424278320703250689780154785099509277691723126325051200459038290  
 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

Get info as 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": "04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1c1d3ae68541d9a04df54a2c48cc241c2983544de52",  
 "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"  
}

Public BIP32 key:

$ ku -w -P P:foo  
xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFidhjFj82pVShWu9curWmb2zy

Generate a subkey:

$ ku -w -s3/2 P:foo  
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6iogWKxx6mefEw4M8EroLgKj

Hardened subkey:

$ ku -w -s3/2H P:foo  
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8iDkBN45k67UKsJUXM1JfRCdn1

WIF:

$ ku -W P:foo  
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH

Address:

$ ku -a P:foo  
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii

Generate a bunch of subkeys:

$ ku P:foo -s 0/0-5 -w  
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3bhXxKWB89Ggn2dzLcoJsuEdRK  
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVej5QHckc5Adtwxa28ACmANi9XhCrRvtFqQcUxt8rUgFz3souMiDdWxJDZnQxzx  
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxan6srXUPBtj3PTxQFkZJAiwoUpmvtrxKZu4zfsnr3pqyy2vthpkwuoVq  
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtFXdiEY8UsRNJfqK6DAdyZXoMvtaLHyWQx3FS4A9zw  
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKdT84Mm7Wtc2xmzFuKg7iYf7XFHKkSsaYKWKJbR54bnyAD9GzjUYbAYTtN4ruo

Generate the corresponding addresses:

$ ku P:foo -s 0/0-5 -a  
1MrjE78H1R1rqdFrmkjdHnPUdLCJALbv3x  
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu  
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb  
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK  
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML  
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM

Generate the corresponding WIFs:

$ ku P:foo -s 0/0-5 -W  
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx  
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ  
L4B3ygQxK6zH2NQGxLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz  
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF  
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuvFZ3v8SKHYFDwkbM765Nrfd  
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C

Check that it works by choosing a BIP32 string (the one corresponding to subkey 0/3):

$ ku -W xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF  
$ ku -a xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzKL1Y8Gk9aX6QbryA5raK73p  
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK

Yep, looks familiar.

From secret exponent:

$ 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

Litecoin version:

$ 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

Dogecoin WIF:

$ ku -nD -W 1  
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot

From public pair (on Testnet):

$ ku -nT 55066263022277343669578718895168534326250603453777594175500187360389116729240,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

From hash160:

$ ku 751e76e8199196d454941c45d1b3a323f1433bd6  
  
input : 751e76e8199196d454941c45d1b3a323f1433bd6  
network : Bitcoin  
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6  
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH

As a Dogecoin address:

$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6  
  
input : 751e76e8199196d454941c45d1b3a323f1433bd6  
network : Dogecoin  
hash160 : 751e76e8199196d454941c45d1b3a323f1433bd6  
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE

==== Transaction Utility (TX)

The command-line utility tx will display transactions in human-readable form, fetch base transactions from pycoin’s transaction cache or from web services (blockchain.info, blockcypher.com, blockr.io and chain.so are currently supported), merge transactions, add or delete inputs or outputs, and sign transactions.

Following are some examples.

View the famous "pizza" transaction:

$ 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 path-to-output-file]  
 argument [argument ...]  
tx: error: can't find Tx with id 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a

Oops! We don’t have web services set up. Let’s do that now:

$ PYCOIN\_CACHE\_DIR=~/.pycoin\_cache  
$ PYCOIN\_BTC\_PROVIDERS="block.io blockchain.info blockexplorer.com"  
$ export PYCOIN\_CACHE\_DIR PYCOIN\_BTC\_PROVIDERS

It’s not done automatically so a command-line tool won’t leak potentially private information about what transactions you’re interested in to a third-party website. If you don’t care, you could put these lines into your *.profile*.

Let’s try again:

$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a  
Version: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes  
TxIn count: 1; TxOut count: 1  
Lock time: 0 (valid anytime)  
Input:  
 0: (unknown) from 1e133f7de73ac7d074e2746a3d6717dfc99ecaa8e9f9fade2cb8b0b20a5e0441:0  
Output:  
 0: 1CZDM6oTttND6WPdt3D6bydo7DYKzd9Qik receives 10000000.00000 mBTC  
Total output 10000000.00000 mBTC  
including unspents in hex dump since transaction not fully signed  
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000  
  
\*\* can't validate transaction as source transactions missing

The final line appears because to validate the transactions' signatures, you technically need the source transactions. So let’s add -a to augment the transactions with source information:

$ 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  
Version: 1 tx hash 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a 159 bytes  
TxIn count: 1; TxOut count: 1  
Lock time: 0 (valid anytime)  
Input:  
 0: 17WFx2GQZUmh6Up2NDNCEDk3deYomdNCfk from 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  
  
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a493046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e80000001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000  
  
all incoming transaction values validated

Now, let’s look at unspent outputs for a specific address (UTXO). In block #1, we see a coinbase transaction to 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX. Let’s use fetch\_unspent to find all coins in this address:

$ 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  
 0e3e2357e806b6cdb1f70b54c3a3a17b6714ee1f0e68bebb44a74b1efd512098/0/410496b538e853519c726a2c91e61ec11600ae1390813a627c66fb8be7947be63c52da7589379515d4e0a604f8141781e62294721166bf621e73a82cbf2342c858eeac/5000000000

== Bitcoin Explorer (bx) Commands

Bitcoin Explorer (bx) is a command-line tool that offers a variety of commands for key management and transaction construction. It is part of the libbitcoin bitcoin library.

Usage: bx COMMAND [--help]  
  
Info: The bx commands are:  
  
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

For more information, see the [Bitcoin Explorer homepage](https://github.com/libbitcoin/libbitcoin-explorer) and [Bitcoin Explorer user documentation](https://github.com/libbitcoin/libbitcoin-explorer/wiki).

=== Examples of bx Command Use

Let’s look at some examples of using Bitcoin Explorer commands to experiment with keys and addresses.

Generate a random "seed" value using the seed command, which uses the operating system’s random number generator. Pass the seed to the ec-new command to generate a new private key. We save the standard output into the file *private\_key*:

$ bx seed | bx ec-new > private\_key  
$ cat private\_key  
73096ed11ab9f1db6135857958ece7d73ea7c30862145bcc4bbc7649075de474

Now, generate the public key from that private key using the ec-to-public command. We pass the *private\_key* file into the standard input and save the standard output of the command into a new file *public\_key*:

$ bx ec-to-public < private\_key > public\_key  
$ cat public\_key  
02fca46a6006a62dfdd2dbb2149359d0d97a04f430f12a7626dd409256c12be500

We can reformat the public\_key as an address using the ec-to-address command. We pass the *public\_key* into standard input:

$ bx ec-to-address < public\_key  
17re1S4Q8ZHyCP8Kw7xQad1Lr6XUzWUnkG

Keys generated in this manner produce a type-0 nondeterministic wallet. That means that each key is generated from an independent seed. Bitcoin Explorer commands can also generate keys deterministically, in accordance with BIP-32. In this case, a "master" key is created from a seed and then extended deterministically to produce a tree of subkeys, resulting in a type-2 deterministic wallet.

First, we use the seed and hd-new commands to generate a master key that will be used as the basis to derive a hierarchy of keys:

$ bx seed > seed  
$ cat seed  
eb68ee9f3df6bd4441a9feadec179ff1  
  
$ bx hd-new < seed > master  
$ cat master  
xprv9s21ZrQH143K2BEhMYpNQoUvAgiEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBqkYoWWMCRwzfuJk8RwF1SVEpAQ

We now use the hd-private command to generate a hardened "account" key and a sequence of two private keys within the account:

$ bx hd-private --hard < master > account  
$ cat account  
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwcgS8PYbgoR2NWHeLyvg4DhoEE68A1n  
  
$ bx hd-private --index 0 < account  
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVEsuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F3DWPkX7Tp1V1u97HKG1FJwUpU  
  
$ bx hd-private --index 1 < account  
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdjsuMVERadbgDbziCRJru9n6tzEWrASVpEdrZrFidt1RDfn4yA3

Next, we use the hd-public command to generate the corresponding sequence of two public keys:

$ bx hd-public --index 0 < account  
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz5wzaSW4FiGrseNDR4LKqTy  
  
$ bx hd-public --index 1 < account  
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQDhohEcDFTEYMvYzwRD7Juf8

The public keys can also be derived from their corresponding private keys using the hd-to-public command:

$ bx hd-private --index 0 < account | bx hd-to-public  
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz5wzaSW4FiGrseNDR4LKqTy  
  
$ bx hd-private --index 1 < account | bx hd-to-public  
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQDhohEcDFTEYMvYzwRD7Juf8

We can generate a practically limitless number of keys in a deterministic chain, all derived from a single seed. This technique is used in many wallet applications to generate keys that can be backed up and restored with a single seed value. This is easier than having to back up the wallet with all its randomly generated keys every time a new key is created.

The seed can be encoded using the mnemonic-encode command:

$ bx hd-mnemonic < seed > words  
adore repeat vision worst especially veil inch woman cast recall dwell appreciate

The seed can then be decoded using the mnemonic-decode command:

$ bx mnemonic-decode < words  
eb68ee9f3df6bd4441a9feadec179ff1

Mnemonic encoding can make the seed easier to record and even remember.

1. "Bitcoin: A Peer-to-Peer Electronic Cash System," Satoshi Nakamoto (<https://bitcoin.org/bitcoin.pdf>). [↑](#footnote-ref-62)