Éloge de 'Mastering Bitcoin'

> "Quand je parle de bitcoin au public, on me demande parfois 'mais comment ça marche au fond ?' J'ai maintenant une très bonne réponse à cette question, car quiconque lit *Mastering Bitcoin* aura une compréhension profonde de son fonctionnement, et aura toutes les clés en main pour écrire la prochaine génération d'extraordinaires applications de cryptomonnaie.
>
> — Gavin Andresen, Chief Scientist de la Fondation Bitcoin

> "Bitcoin et les technologies de blockchain deviennent des briques de base fondamentales pour la prochaine génération d'internet. Les esprits les plus brillants de la Silicon Valley travaillent dessus. Le livre d'Andreas vous permettra de prendre part à cette révolution du monde de la finance par le logiciel."
>
> — Naval Ravikant, Co-fondateur d'AngelList

> "*Mastering Bitcoin* est la meilleure référence technique aujourd'hui disponible sur bitcoin. Et bitcoin sera probablement vu rétrospectivement comme la technologie la plus importante de cette décennie. C'est pourquoi ce livre est un must-have absolu pour tous les développeurs, en particulier ceux intéressé pour bâtir des applications avec le protocole bitcoin. Hautement recommandé."
>
> — Balaji S. Srinivasan (@balajis), General Partner, Andreessen Horowitz

> "L'invention de la Blockchain Bitcoin représente une plateforme entièrement nouvelle, sur laquelle pourra se construire un écosystème aussi grand et varié qu'internet. Un des maîtres à penser de la communauté, Andreas Antonopoulos est le meilleur auteur possible.
>
> — Roger Ver, Entrepreneur Bitcoin et Investisseur

Index

Préface

Un Livre sur Bitcoin

Je suis tombé pour la première fois sur bitcoin à la mi-2011. Ma réaction initiale a été quelque chose comme "Pff ! De l'argent pour geeks !", et je n'y ai plus prêté attention pendant les six mois qui ont suivi, faute d'en avoir perçu l'importance. J'ai observé cette réaction chez plusieurs des personnes les plus intelligentes que je connaisse, ce qui me réconforte quelque peu. La seconde fois que j'ai croisé la route de bitcoin, au cours d'une discussion sur une mailing list, j'ai décidé de lire le livre blanc de Satoshi Nakamoto, afin d'étudier la source de référence et de voir de quoi il en retournait. Je me souviens encore de ce moment où, ayant achevé de lire les neuf pages, j'ai réalisé que bitcoin n'était pas simplement une monnaie numérique, mais un réseau de confiance sur lequel pouvait se construire beaucoup plus que des monnaies. Ayant pris conscience que "ce n'est pas de l'argent, mais un réseau de confiance décentralisé", je me lançais dans un périple de quatre mois afin d'engloutir toutes les informations que je pouvais trouver sur bitcoin. J'étais obsédé, subjugué, collé 12 heures ou plus par jour à mon écran, lisant, écrivant, codant, et apprenant autant que je le pouvais. J'émergeai de cet état second avec 10 kilos de moins, faute de repas normaux, et décidé à me consacrer à travailler sur bitcoin.

Deux ans plus tard, après avoir créé plusieurs petites startups visant à explorer différents produits et services liés à bitcoin, j'ai décidé qu'il était temps d'écrire mon premier livre. Bitcoin était le sujet qui m'avait conduit à une frénésie de créativité et qui avait enflammé mes réflexions ; c'était la technologie la plus excitante que j'avais rencontrée depuis Internet. Le temps était venu de partager ma passion pour cette fantastique technologie, avec une audience plus large.

Public Visé

Ce livre est destiné principalement aux codeurs. Si vous savez utiliser un langage de programmation, ce livre vous enseignera comment les monnaies cryptographiques fonctionnent, comment les utiliser, et comment développer des logiciels qui s'en servent. Les premiers chapitres peuvent aussi constituer une introduction approfondie sur bitcoin pour les non-codeurs—ceux qui essaient de comprendre les mécanismes internes de bitcoin et des cryptomonnaies.

Les conventions utilisées dans ce Livre

Les conventions typographiques suivantes sont utilisées dans ce livre :

Italique

Indique les termes nouveaux, les URLs, adresses email, noms et extensions de fichiers.

Largeur constante

Est utilisée pour les extraits de programmes, ainsi qu'au sein des paragraphes afin d'évoquer des éléments de programmation tels qu'une variable, des noms de fonctions, des bases de données, des

types de données, des variables d'environnement, des déclarations ou des mots-clés.

`Largeur constante en gras`
: Utilisée pour des commandes ou d'autres textes qui peuvent être tapées telles quelles par un utilisateur.

Largeur constante en italique
: Utilisée pour les textes devant être remplacés par des valeurs fournies par les utilisateurs, ou par des valeurs dépendant d'un contexte.

TIP	Cette icône indique une astuce, une suggestion ou une note générale

WARNING	Cette icône indique une alerte ou un éventuel danger

Exemples de code

Les exemples fournis sont en Python, en C++, et utilisent des lignes commandes de type Unix pouvant être utilisées sous linux ou Mac OSX. Tous les extraits de code sont disponibles dans le repository Github, et sont accessibles en ligne sur Dépôt GitHub dans le sous-répertoire *code* du dépôt principal. Il est possible de créer de nouveaux embranchements, d'essayer les exemples de code, ou de soumettre des corrections via GitHub.

Tous les extraits de code peuvent être exécutés sur la plupart des systèmes d'exploitation et requièrent un minimum de composants installés pour la compilation et l'interprétations des langages utilisés. Quand cela est nécessaire, nous fournissons des instructions basiques d'installation des composants utiles avec une description pas à pas de la démarche à suivre et du résultat attendu.

Certains extraits de code ont été formatés pour des raison d'impression. Dans ce cas, les lignes ont été scindées par le caractère anti-slash (\), suivi d'un caractère nouvelle ligne. Quand vous retranscrirez ces exemples, supprimez ces deux caractères afin de ne former qu'une seule ligne à nouveau, et vous devriez obtenir un résultat identique à celui montré dans l'exemple.

Tous les exemples de code utilisent des valeurs et des calculs réels quand cela est possible, afin que vous puissiez obtenir les mêmes résultats dans le cas ou vous les exécutiez. Par exemple, les clés privées et les clé publiques ainsi que les adresses correspondantes utilisées dans cet ouvrages sont réelles. Les transactions données en exemple, les blocs et les références à la blockchain sont toutes réelles. Les transactions données en exemple, les blocs et les références au blockchain sont réellement présents dans la blockchain bitcoin en tant que parties intégrantes du registre public, afin que vous puissiez les retrouver à partir de n'importe quel système bitcoin.

Remerciements de l'auteur

Ce livre représente les efforts et la contribution de beaucoup de monde. Je suis reconnaissant de toute l'aide que j'ai pu recevoir de mes amis, mes collègues et même de parfaits étrangers qui m'ont rejoint

dans cet effort de rédaction du livre ultime sur les crypto-monnaies et le bitcoin.

Il est impossible de séparer le bitcoin en tant que technologie et le bitcoin en tant que communauté, et ce livre est autant le fruit de la communauté qu'il est un ouvrage sur la technologie. Mon travail sur ce livre a été encouragé, accueilli, supporté et récompensé par la communauté Bitcoin toute entière, et ce du début jusqu'à la fin. Plus que tout, ce livre m'aura permis de faire partie de cette communauté fantastique et je ne vous remercierai jamais assez de m'avoir accepté au sein de cette communauté. Il y a beaucoup trop de monde pour que je puisse les citer un par un – ceux que j'ai rencontré lors des conférences, séminaires, meetups, autour d'une pizza ou en petits comités privés, ceux avec qui j'ai communiqué via twitter, sur reddit, sur bitcointalk.org et sur Github ont tous eu un impact sur ce livre. Toute idée, analogie, question, réponse, et explication que vous trouverez dans cet ouvrage sont d'une manière ou d'une autre inspirées, testées ou améliorées au travers de mes interactions avec la communauté. Merci à vous tous pour votre soutien, sans vous ce livre n'aurait jamais vu le jour. Je vous serai à jamais reconnaissant.

Bien sûr, le chemin parcouru afin de devenir un auteur commence bien avant ce livre. Ma langue maternelle est le Grec, j'ai donc dû prendre des cours d'écriture en Anglais lors de ma première année d'université. Je remercie pour cela Diana Kordas, mon professeur d'anglais, qui m'a aidé à prendre confiance et élever mon niveau cette année-là. Plus tard, en tant que professionnel, j'ai développé mes compétences techniques sur le sujet des centres de données, en écrivant pour le magazine Network World. Je remercie John Dix et John Gallant qui m'ont donné mon premier poste d'auteur en tant que journaliste à Network World et à mon éditeur Michael Cooney et ma collègue Johna Till Johnson qui ont publié mes articles. Le fait d'avoir à écrire 500 mots par semaine pendant 4 ans m'a donné assez d'expérience pour envisager une carrière d'auteur. Merci à Jean pour m'avoir encouragé très tôt à devenir auteur, et pour avoir toujours cru et insisté sur le fait que cela était fait pour moi.

Je voudrais également remercier ceux qui m'ont soutenu lorsque j'ai proposé mon livre à O'Reilly, en donnant leur recommandation et relisant la proposition. 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. Un remerciement particulier à Richard Kagan et Tymon Mattoszko qui ont relu les premiers manuscrits et Matthew Owain Taylo qui les a révisés.

Merci à Cricket Liui, auteur du titre O'Reilly « DNS et BIND » qui m'a introduit chez O'Reilly. Merci aussi à Michael Loukides et Allyson MacDonald chez O'Reilly qui ont travaillé pendant des mois pour que ce livre sorte. Allyson a été particulièrement patiente quand des retards sont apparus à cause des incidents de la vie.

Les premieres versions des premiers chapitres ont été les plus dures, car le bitcoin est un sujet difficile à aborder. A chaque fois que je voulais aborder un sujet concernant la technologie du bitcoin, je me retrouvais à devoir parler de la technologie dans son intégralité. Je suis resté longtemps bloqué et découragé quand j'essayais de transformer un sujet techniquement très dense en une histoire facile à comprendre. J'ai finalement décidé de raconter l'histoire du bitcoin en me servant de scénarios de personnes utilisant le bitcoin et le livre a tout d'un coup été beaucoup plus simple à écrire. Je dois remercier mon mentor et ami Richard Kagan, qui m'a aidé à m'en sortir pendant ces moments de blocage, ainsi que Pmela Morgan qui a révisé les premières versions des premiers chapitres et m'a posé les bonnes questions pour m'aider à les améliorer. J'aimerais également remercier les

développeurs du groupe « San Francisco Bitcoin Developers Meetup » et Taariq Lewis, son cofondateur, pour m'avoir aidé à tester les premiers bouts de code.

Pendant l'écriture de cet ouvrage, j'ai rendu disponible mes premiers manuscrits sur Github et encouragé les gens à commenter mon travail. Plus d'une centaine de commentaires, suggestions, corrections et contributions m'ont été soumises en réponse. Ces contributeurs sont cités dans la section Première version du manuscrit (Contributions GitHub). J'aimerais remercier particulièrement Minh T. Nguyen qui s'est porté volontaire pour gérer toutes ces contributions et qui a été lui-même un contributeur actif. Merci aussi à Andrew Naugler pour ses illustrations.

Une fois que la première version du manuscrit était terminée, il a alors fallu effectuer plusieurs revues techniques. Merci a Cricket Liu et Lorne Lantz pour leur revue complète, leurs commentaires et leur aide précieuse.

Plusieurs développeurs bitcoin ont contribué aux exemples de code, revues, commentaires et encouragements. Merci à Amir Taaki pour ses exemples de code et ses nombreux commentaires; Vitalik Buterin et Richard Kiss pour leur aide sur la courbe elliptique et leurs contributions au code; Gavin Andresen pour ses corrections, commentaires et encouragements, Michalis Kargalis pour ses commentaires, contributions et sa critique de btcd; et Robin Inge pour ses propositions d'errata qui ont permis d'améliorer la seconde édition.

Je dois mon amour des mots et des livres à ma mère, Theresa, qui m'a élevé dans une maison ou les livres s'alignaient sur chaque mur. Ma mère m'a également acheté mon premier ordinateur en 1982, bien qu'elle se décrive elle-même comme technophobe. Mon père, Menelaos, un ingénieur civil qui vient juste de publier son premier livre à l'âge de 80 ans, a été celui qui m'a enseigné la pensée logique et analytique et l'amour de la science et de l'ingénierie.

Merci à vous tous pour m'avoir encouragé tout au long de cette aventure.

Première version du manuscrit (Contributions GitHub)

Beaucoup de contributeurs ont proposés leurs commentaires, leurs correction et ajouts à la première version GitHub. Merci à vous tous pour votre contribution à cet ouvrage. La liste qui suit contient les éminents contributeurs avec leur identifiant GitHub entre parenthèses:

- Minh T. Nguyen, éditeur de contribution GitHub (enderminh)
- Ed Eykholt (edeykholt)
- Michalis Kargakis (kargakis)
- Erik Wahlström (erikwam)
- Richard Kiss (richardkiss)
- Eric Winchell (winchell)
- Sergej Kotliar (ziggamon)
- Nagaraj Hubli (nagarajhubli)

- ethers
- Alex Waters (alexwaters)
- Mihail Russu (MihailRussu)
- Ish Ot Jr. (ishotjr)
- James Addison (jayaddison)
- Nekomata (nekomata-3)
- Simon de la Rouviere (simondlr)
- Chapman Shoop (belovachap)
- Holger Schinzel (schinzelh)
- effectsToCause (vericoin)
- Stephan Oeste (Emzy)
- Joe Bauers (joebauers)
- Jason Bisterfeldt (jbisterfeldt)
- Ed Leafe (EdLeafe)

Edition ouverte

Ceci est une édition libre de "Mastering Bitcoin", publié pour traduction sous licence Creative Commons. Creative Commons Attribution Share-Alike License (CC-BY-SA). Cette licence vous autorise à lire, partager, imprimer, vendre ou réutiliser ce livre, en totalité ou en partie si vous:

appliquez la même licence (Share-Alike) * Include attribution

Attribution

"Mastering Bitcoin" par Andreas M. Antonopoulos LLC https://bitcoinbook.info

Droits d'auteur 2016, Andreas M. Antonopoulos LLC

Traduction

Si vous lisez ce livre dans une autre langue que l'anglais, c'est qu'il a été traduit par des volontaires. Les personnes suivantes ont contribué à la traduction et l'adaptation:

Fabien Robyr * Name 2 * Name 3

Glossaire Succinct

Ce glossaire succinct contient de nombreux termes liés au bitcoin. Ces termes sont utilisés tout au long du livre, il est donc conseillé de marquer cette page pour la retrouver facilement.

adresse

Une adresse bitcoin ressemble à 1DSrfJdB2AnWaFNgSbv3MZC2m74996JafV. Elle consiste en une suite de chiffres et de lettres commençant par "1" (le chiffre un). De la même façon que vous demandez à quelqu'un de vous envoyer un email à votre adresse email, vous pouvez lui demander d'envoyer des bitcoins à votre adresse bitcoin.

bip

Bitcoin Improvement Proposals (en français : Propositions d'Amélioration de Bitcoin). Un ensemble de propositions que les membres de la communauté ont émises pour améliorer bitcoin. Par exemple, BIP0021 est une proposition pour améliorer le schéma d'URI spécifique à bitcoin.

bitcoin

Le nom de l'unité de monnaie (la pièce), le réseau, et le logiciel.

block

Un groupe de transactions, avec un horodatage et l'empreinte du bloc précédent. L'entête de bloc est hashé pour produire une preuve de travail, validant ainsi les transactions. Les blocs valides sont ajoutés à la blockchain principale par consensus distribué.

blockchain

Une chaîne de bloc valides, chaque bloc étant relié à son prédécesseur jusqu'au premier bloc, appelé le genesis bloc.

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 blockchain, la transaction a une deuxième confirmation, et ainsi de suite. Au bout six confirmations ou plus, on considère que la transaction est irréversible.

difficulté

Un réglage de l'ensemble du réseau qui contrôle la quantité de calcul requise pour fournir une preuve de travail.

difficulté cible

Une difficulté telle que le réseau pourra calculer un bloc toutes les 10 minutes environ.

reciblage de difficulté

Un recalcul de la difficulté appliquée à l'ensemble du réseau qui a lieu une fois tous les 2106 bloques en prenant en considération la puissance de hachage des 2106 bloques précédents.

frais

Celui qui envoie la transaction inclut souvent une commission pour que le réseau traite la transaction sollicitée. La plupart des transactions requièrent une commission minimum de 0,5 mBTC.

hash

L'empreinte numérique de données binaire."hash")

bloc genesis

Le premier bloque de la chaîne de bloques, utilisé pour initialiser la crypto-monnaie.

mineur

Un noeud du réseau qui trouve une preuve de travail valide pour de nouveaux bloques, en effectuant des hachages répétés.

réseau

Un réseau pair à pair qui propage les transactions et les bloques vers chaque noeud bitcoin du réseau.

Preuve de Travail

Une donnée qui requiert un travail significatif pour être calculée. Dans le cadre de bitcoin, les mineurs doivent trouver une solution numérique à l'algorithme SHA256, qui permet de remplir un objectif défini pour l'ensemble du réseau, appelé la difficulté.

récompense

Une somme incluse dans chaque nouveau bloc comme récompense pour le mineur qui a trouvé la solution de la Preuve de Travail. Elle s'élève actuellement à 25BTC par bloc.

clé secrète (ou clé privée)

Le numéro secret qui débloque les bitcoins à envoyer à l'adresse correspondante. Une clé secrète ressemble à 5J76sF8L5jTtzE96r66Sf8cka9y44wdpJjMwCxR3tzLh3ibVPxh.

transaction

En termes simples, un transfert de bitcoins d'une adresse à une autre. Plus précisément, une transaction est une structure de données signée qui exprime une transfert de valeur. Les transactions sont transmises sur le réseau bitcoin, collectées par les mineurs, et incluses dans des bloques, de façon permanente dans la chaîne de bloques.

porte-monnaie

Logiciel qui contient toutes vos adresses bitcoin et clés privées. Utilisé pour envoyer, recevoir et stocker vos bitcoins.

Introduction

Qu'est ce que Bitcoin ?

Bitcoin est un ensemble de concepts et de technologies formant la base d'un écosystème de monnaie numérique. Les unités de monnaie appelées bitcoins sont utilisées pour conserver et transmettre de la valeur parmi les participants du 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 être utilisés. La pile du protocole bitcoin, disponible en tant que logiciel open source, peut être exécutée sur une large gamme d'ordinateurs, y compris les ordinateurs portables ou les smartphones, rendant cette technologie facilement accessible.

Les utilisateurs peuvent transférer des bitcoins sur le réseau pour faire tout ce qui peut se faire avec des monnaies traditionnelles, acheter ou vendre des biens et services, envoyer de l'argent à des individus ou des organisations ou accorder des crédits. La technologie Bitcoin inclut des fonctionnalités qui sont basées sur le cryptage et les signatures numériques afin de s'assurer de la sécurité du réseau bitcoin. Les bitcoins peuvent être achetés, vendus et échangés contre d'autres monnaies sur des échanges spécialisés. Le bitcoin est dans un sens la forme de monnaie parfaite pour internet puisqu'elle est rapide, sûre et sans frontières.

Contrairement aux monnaies classiques, les bitcoins sont entièrement virtuels. Il n'existe pas de pièce physique ou même de pièce numérique. Les pièces sont inclues dans les transactions transmettant de la valeur de l'émetteur au destinataire. Les utilisateurs de bitcoin possèdent des clés qui prouvent la possession des transactions sur le réseau bitcoin et déverrouillent la valeur pour la dépenser et la transférer à un autre destinataire. Ces clés sont souvent enregistrées dans un portefeuille numérique présent sur l'ordinateur de chaque utilisateur. La possession d'une clé pour déverrouiller une transaction est l'unique pré-requis pour dépenser des bitcoins, ce système donne ainsi entièrement le contrôle aux utilisateurs.

Bitcoin est un système pair-à-pair entièrement distribué. Ainsi, il n'y a aucun serveur ou point de contrôle « central ». Les bitcoins sont créés au travers d'un processus appelé « minage », qui implique de trouver la solution à un problème difficile à résoudre. N'importe quel participant au réseau bitcoin (c'est à dire, n'importe quel ordinateur opérant la pile complète bitcoin) peut agir en tant que mineur, en utilisant la puissance de calcul qu'il a à sa disposition afin de résoudre le problème. Toutes les 10 minutes en moyenne, une nouvelle solution est trouvée par quelqu'un qui est alors capable de valider les transactions des dernières 10 minutes. En résumé, le minage bitcoin décentralise l'émission de monnaie et les procédures de rapprochement rendant inutile l'intervention d'un organisme similaire aux banques centrales.

Le protocole bitcoin inclut des algorithmes prédéfinis qui régulent la fonction de minage sur le réseau. La difficulté de l'exécution de la tâche effectuée par les mineurs –afin d'enregistrer un bloc de transaction sur le réseau bitcoin– est ajustée de façon à ce qu'en moyenne quelqu'un y arrive toutes les 10 minutes, peu importe le nombre de mineurs (et de CPUs) travaillant sur cette tâche à un instant t. Le protocole divise de moitié la quantité de bitcoins créées tous les quatre ans et limite le nombre total de

bitcoins émis à un total de 21 millions de pièces. Par conséquent, le nombre total de bitcoins en circulation suit une courbe aisément prévisible qui atteindra 21 millions d'unités vers l'année 2140. Vu sa vitesse d'émission allant en diminuant, sur le long terme, la monnaie bitcoin est déflationniste. Enfin, le bitcoin ne peut pas être gonflé artificiellement en générant de la monnaie au-delà du taux d'émission attendu.

Bitcoin est aussi le nom d'un protocole, d'un réseau, et d'une innovation dans l'informatique distribuée. Le bitcoin en tant que monnaie n'est vraiment que la première application de cette invention. En tant que développeur, je vois un peu le bitcoin comme l'Internet de l'argent, un réseau chargé de propager de la valeur et de sécuriser la possession de biens numériques via des calculs distribués. Le bitcoin est beaucoup plus que ce qu'il semble être à première vue.

Dans ce chapitre nous commencerons par expliquer les concepts et termes principaux, nous installerons les logiciels nécessaires, et utiliserons le bitcoin pour des transactions simples. Dans les chapitres suivants, nous analyserons toutes les couches technologiques qui rendent bitcoin possible et nous examinerons le fonctionnement interne du réseau et du protocole bitcoin.

Les monnaies numériques avant le Bitcoin

L'émergence de monnaies numériques viables est étroitement liée au développement de la cryptographie. Ce n'est guère surprenant lorsque l'on considère le challenge fondamental de l'utilisation de bits pour représenter de la valeur pouvant être échangée pour des biens et des services. Les deux premières questions que se posent ceux qui acceptent de la monnaie numérique sont :

1. Comment puis-je m'assurer que la monnaie est authentique et non une contrefaçon ?
2. Comment être sûr que personne d'autre ne peut revendiquer la propriété de cette monnaie à ma place (le fameux problème de la double dépense)

Les émetteurs de papier-monnaie livrent un combat sans fin contre la contrefaçon en utilisant des papiers et des techniques d'impression de plus en plus sophistiqués. La monnaie physique résout simplement le problème de la double dépense car un même billet ne peut se trouver à deux endroits en même temps. Cependant, la monnaie conventionnelle peut se transmettre de façon numérique. Dans ce cas, les problèmes de contrefaçon et de double dépense sont adressés par la vérification de toutes les transactions électroniques au travers d'autorités centrales qui possèdent une vision globale de la monnaie en circulation. Pour ce qui est des monnaies numériques qui ne peuvent profiter de l'utilisation d'encres particulières ou de bandes holographiques, la cryptographie fournit les bases de la confiance en la légitimité de la revendication de valeur d'un utilisateur. Plus spécifiquement, les signatures cryptographiques électroniques permettent à un utilisateur de signer un bien numérique ou une transaction prouvant la possession de ce bien. Avec l'architecture appropriée, les signatures numériques peuvent également être utilisées pour résoudre le problème de la double dépense.

Quand la cryptographie a commencé à devenir plus largement disponible et comprise vers la fin des années 1980, beaucoup de chercheurs ont commencé a essayer d'utiliser la cryptographie

pour construire des monnaies numériques. Ces tous premiers projets ont donné le jour à des monnaies numériques généralement adossées à des monnaies nationales ou des métaux précieux tels que l'or.

Malgré le fait que ces monnaies numériques fonctionnaient, elles étaient centralisées et au final elle étaient faciles à attaquer par les gouvernements et les hackers. Les premières monnaies numériques utilisaient un organisme de "clearing" afin de vérifier toutes les transactions à intervalles réguliers, comme le fait le système bancaire traditionnel. Malheureusement, dans la plupart des cas ces monnaies numériques naissantes ont été ciblées par des gouvernements inquiets qui les ont réduites à néant avec l'arme législative. Certaines ont échoué de façon spectaculaire quand leur société mère a brutalement disparu. Afin de résister face à l'intervention de différents protagonistes, que ce soit des gouvernement légitimes ou des entités criminelles, une monnaie numérique décentralisée était nécessaire afin d'éviter un unique point d'attaque. Bitcoin est construit de la sorte, une monnaie numérique complètement décentralisée dans sa conception et dénuée d'une autorité centrale ou point de contrôle pouvant être attaqué ou corrompu.

Bitcoin représente le point culminant des dizaines d'années de recherche en cryptographie et en systèmes distribués et inclut quatre innovations clés dans une combinaison unique et puissante. Bitcoin consiste en:

- Un réseau décentralisé pair-à-pair (le protocole bitcoin)
    - Un registre public de transaction (la blockchain)
    - Un système d'émission de monnaie décentralisé mathématique et déterministe (le minage distribué)
    - Un système décentralisé de vérification des transactions (les scripts de transaction)

Historique du Bitcoin

Bitcoin a été inventé en 2008 avec la publication d'un document intitulé « Bitcoin : Un système de cash électronique peer-to-peer » écrit sous le pseudonyme de Satoshi Nakamoto. Nakamoto a combiné plusieurs inventions précédentes telles que b-money et Hashcash afin de créer un système de cash électronique complètement décentralisé ne reposant sur aucune autorité centrale pour l'émission de monnaie ou le règlement et la validation des transactions. L'innovation principale a été d'utiliser un système de calcul distribué (appelé algorithme « proof-of-work ») réalisant une « élection » globale toutes les 10 minutes, permettant au réseau décentralisé d'arriver à un consensus sur l'état des transactions. Cela résout de façon élégante le problème de la double dépense ou une unité de monnaie ne peut être dépensée deux fois. Précédemment, le problème de double dépense était une faiblesse des monnaies numériques et était résolu en réalisant la vérification de toutes les transactions par un organisme de "clearing".

Le réseau bitcoin a commencé en 2009, basé sur une implémentation de référence publié par Nakamoto et révisée depuis par beaucoup d'autres programmeurs. Le calcul distribué qui fournit la

sécurité et la résistance au bitcoin a crû de façon exponentielle pour aujourd'hui surpasser la puissance de calcul des plus puissants super-calculateurs de ce monde. La totalité du marché Bitcoin est aujourd'hui estimée entre 5 et 10 milliards de dollars, en fonction du cours d'échange. La plus grosse transaction opérée à ce jour a été de 150 millions de dollars, transmise instantanément et opérée sans aucun frais.

Satoshi Nakamoto a cessé toute intervention publique en Avril 2011, laissant la responsabilité du développement du code et du réseau à un solide groupe de volontaires. L'identité de la personne ou du groupe de personnes derrière la création de Bitcoin reste à ce jour inconnue. Cependant, ni Satoshi Nakamoto ni personne n'exercent un quelque contrôle sur le système bitcoin, qui n'opère que selon des principes mathématiques totalement transparents. L'invention en elle-même est révolutionnaire et a déjà apporté beaucoup dans les domaines de du calcul distribué, de l'économie et de l'économétrie.

Une solution à un problème de calcul distribué

> L'invention de Satoshi Nakamoto est également une solution concrète à un problème jusqu'alors insoluble en calcul distribué, connu sous le nom du «Problème des généraux Byzantins ». Pour expliquer rapidement, ce problème consiste à essayer de se mettre d'accord sur les actions à mener en échangeant de l'information sur un réseau non fiable et possiblement compromis. La solution de Satoshi Nakamoto, qui utilise le concept de proof-of-work afin d'arriver à un consensus sans l'aide d'une autorité centrale, représente une percée dans la science du calcul distribué et possède un champ d'application au-delà de la monnaie. Elle peut être utilisée pour arriver à un consensus au sein de réseaux décentralisés afin de prouver la légitimité d'élections, les loteries, les registres de biens, la notarisation électronique et bien d'autres choses encore.

Les usages du bitcoin, ses utilisateurs et leurs scénarios

Bitcoin est une technologie, mais elle s'applique à l'argent qui est le langage fondamental pour échanger de la valeur entre les gens. Jetons un œil aux personnes qui utilisent le bitcoin et à certains des usages les plus communs de la monnaie et du protocole au travers de leurs histoires. Nous réutiliserons ces cas pratiques tout au long de l'ouvrage afin d'illustrer des usages dans un contexte réel de la vie de tous les jours et comment ils ont été rendus possible par les différentes technologies constitutives du bitcoin.

La vente de biens de faible valeur en Amérique du nord Alice vit dans le nord de la Californie. Elle a entendu parler du bitcoin par ses amis informaticiens et veut commencer à l'utiliser. Nous suivrons son histoire, de son apprentissage de ce qu'est le bitcoin, son achat de bitcoins jusqu'à son utilisation de bitcoins pour acheter un café au Bob's Cafe à Palo Alto. Cette histoire va nous faire découvrir les logiciels, les échanges et les transactions basiques du point de vue d'un consommateur lambda.

La vente de biens de forte valeur en Amérique du nord Carol est une galeriste de San Francisco. Elle vend des œuvres d'art ayant un prix élevé contre des bitcoins. Ce scénario nous permettra d'aborder les risques d'une attaque concertée des « 51% » pour les vendeurs de biens à forte valeur.

Les contrats de service offshore Bob, le propriétaire du café de Palo Alto, est en train de réaliser son nouveau site web. Pour ce faire, il a passé un contrat avec un développeur web indien, Gopesh, qui vit à Bangalore en Inde. Gopesh a accepté d'être payé en bitcoins. Ce scénario démontrera l'usage du bitcoin pour l'outsourcing, les contrats de services et les transferts internationaux.

Les dons de bienfaisance Eugenia est la directrice d'une organisation de bienfaisance aux Philippines. Elle a récemment découvert le bitcoin et souhaite l'utiliser afin de toucher de nouvelles personnes sur place et à l'étranger pour lever des fonds pour son organisation. Elle recherche également des moyens d'utiliser le bitcoin afin de distribuer rapidement des fonds pour les zones dans le besoin. Ce scénario nous montrera l'utilisation du bitcoin pour la levée de fonds à l'international et l'utilisation du registre public pour les organisations de bienfaisance.

L'Import/export

Mohammed est un importateur de biens électroniques à Dubaï. Il essaye d'utiliser le bitcoin pour acheter du matériel électronique aux Etats-Unis et en Chine et l'importer aux Emirat arabes unis afin d'accélérer les processus de paiement pour l'import. Cette histoire nous montrera comment le bitcoin peut être utilisé pour des paiements B2B internationaux pour l'achat de biens physiques.

Le minage de bitcoin Jing est un étudiant en ingénierie informatique à Shanghai. Il a construit une plateforme de minage afin de miner des bitcoins : utilisant ses connaissances en informatique pour obtenir un revenu complémentaire. Cette histoire nous permettra de découvrir l'aspect « industriel » du bitcoin : l'équipement spécialisé utilisé pour sécuriser le réseau bitcoin et émettre de la monnaie.

Chacun de ces scénarios est basé sur de véritables personnes et de véritables industries qui utilisent le bitcoin pour créer de nouveaux marchés, de nouvelles industries, et des solutions innovantes pour résoudre des problèmes économiques globaux.

Comment débuter

Afin de rejoindre le réseau bitcoin et commencer à utiliser cette monnaie, tout ce qu'un utilisateur a à faire est de télécharger une application ou d'utiliser une application web. Parce que Bitcoin est un standard, il y a beaucoup d'implémentations différentes de clients bitcoin. Il existe également une implémentation de référence, également connue sous le nom de client Satoshi, qui est géré comme un projet open source par une équipe de développeurs et qui découle de l'implémentation originelle écrite par Satoshi Nakamoto.

Il existe trois types de clients bitcoin :

Le client lourd

Un client lourd ou « nœud complet » (full node), est un client qui enregistre l'historique complet des transactions bitcoins (toutes les transactions de tous les utilisateurs et de tout temps), gère les portefeuilles de l'utilisateur et peut initier les transactions directement sur le réseau bitcoin. Il est similaire à un serveur autonome de courrier électronique dans le sens où il gère tous les aspects du protocole sans reposer sur aucun autre serveur ou service tiers.

Le client léger

Un client léger contient le portefeuille de l'utilisateur mais dépend de serveurs tiers pour l'accès aux transactions et au réseau bitcoin. Le client léger n'enregistre pas une copie complète de toutes les transactions et par conséquent doit faire confiance aux serveurs tiers pour la validation des transactions. Il est similaire à un client de messagerie qui se connecte à un serveur de courrier électronique pour accéder à sa boite aux lettres dans le sens ou il se repose sur un serveur tiers pour réaliser des interactions avec le réseau.

Le client web

Les clients web sont accessibles depuis un navigateur et enregistrent les portefeuilles des utilisateurs sur un serveur que possède une société tierce. Ils sont similaires à un client de messagerie en ligne qui repose entièrement sur un serveur tiers.

Bitcoin sur mobile

Les clients mobiles pour smartphones, tels que ceux fonctionnant sur Android, peuvent opérer soit comme des clients complets, des clients légers, ou des clients web. Certains clients mobiles sont synchronisés avec un client web ou un client lourd et fournissent un portefeuille multiplateforme utilisable sur plusieurs terminaux mais avec une source commune de fonds.

Le choix d'un client bitcoin dépend du contrôle que souhaite exercer l'utilisateur sur ses fonds. Un client lourd offrira le plus haut niveau de contrôle et d'indépendance pour l'utilisateur, mais déportera la responsabilité des sauvegardes et la sécurité sur ce dernier. A l'autre bout de l'éventail de choix, le client web est le plus facile à configurer et à utiliser mais en contrepartie la sécurité et le contrôle sont partagés entre l'utilisateur et la société possédant le service web et cela introduit un risque supplémentaire. Si le service web est compromis, comme beaucoup l'ont été, les utilisateurs perdent tous leurs fonds. D'un autre côté, si les utilisateurs possèdent un client lourd mais n'effectuent pas les sauvegardes nécessaires, ils peuvent perdre leurs fonds suite à un crash de leur ordinateur.

Dans cet ouvrage, nous montrerons l'usage de client variés, depuis le client de référence (le client Satoshi) jusqu'au client web. Certains exemples demanderont l'utilisation du client de référence qui en plus d'être un client lourd, propose des APIs vers le portefeuille, le réseau, et les services transactionnels. Si vous prévoyez d'explorer les interfaces de programmation du bitcoin, vous aurez besoin du client de référence.

Démarrage rapide

Alice, que nous avons présenté dans Les usages du bitcoin, ses utilisateurs et leurs scénarios, n'est pas une utilisatrice à fort bagage technique et n'a que très récemment entendu parler du bitcoin par un ami. Elle commence son aventure en visitant le site officiel bitcoin.org, où elle peut trouver une large sélection de clients bitcoin. Suivant le conseil du site bitcoin.org, elle opte pour le client léger Multibit.

Alice suit le lien fourni par bitcoin.org pour télécharger et installer Multibit sur son ordinateur. Multibit est disponible pour les systèmes d'exploitation Windows, Mac OS, et Linux.

WARNING

Un portefeuille bitcoin se doit d'être protégé par un mot de passe ou une « phrase » de passe. Il existe beaucoup d'acteurs malveillants qui essayent de casser les mots de passe faibles, prenez donc garde en choisissant un mot de passe qui ne peut être facilement craqué. Utilisez une combinaison de lettres minuscules et majuscules, nombres et symboles. Évitez les informations personnelles telles que les dates de naissance les noms ou votre équipe de sport favorite. Évitez tous les mots trouvables dans le dictionnaire, et ce dans n'importe quelle langue. Si vous le pouvez, utilisez un générateur de mot de passe pour créer un mot de passe complètement aléatoire d'une longueur d'au moins 12 caractères. Rappelez-vous de cela : bitcoin représente de l'argent et peut être instantanément déplacé n'importe où dans le monde. Si vos bitcoins ne sont pas assez protégés, ils peuvent être facilement volés.

Une fois qu'Alice a téléchargé et installé l'application Multibit, elle la lance et un écran d'accueil s'affiche comme montré sur L'écran d'accueil du client bitcoin Multibit.

Figure 1. L'écran d'accueil du client bitcoin Multibit

Multibit créée automatiquement un portefeuille et une nouvelle adresse bitcoin pour Alice, adresse qu'Alice peut voir en cliquant sur l'onglet Requête comme montré sur La nouvelle adresse bitcoin d'Alice dans l'onglet Requête du client Multibit.

Figure 2. La nouvelle adresse bitcoin d'Alice dans l'onglet Requête du client Multibit

La partie la plus importante de cet écran est l'*adresse bitcoin* d'Alice. Comme une adresse email, Alice peut partager cette adresse et n'importe qui peut l'utiliser pour envoyer de l'argent directement sur son nouveau portefeuille. Sur l'écran on peut voir une longue chaine de caractères composée de chiffres et de lettres 1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK. A coté de cette adresse bitcoin, il y a un code QR, une forme de code barre qui contient la même information dans un format pouvant être scanné par une caméra de smartphone. Le code QR est l'image carrée en noir et blanc sur le coté droit de la fenêtre. Alice peut copier l'adresse bitcoin ou le code QR dans son presse-papier en cliquant sur les boutons situés à coté de chacun d'eux. Cliquer sur le code QR l'agrandira, ce qui facilitera son scan par une camera de smartphone.

Alice peut aussi imprimer le code QR afin de pouvoir facilement donner son adresse à d'autres personnes sans avoir à taper la longue chaîne de caractère de son adresse.

TIP

Les adresses bitcoin commencent par le chiffre 1 ou 3. Comme les adresse email, elles peuvent être partagées à d'autres utilisateurs de bitcoin qui peuvent les utiliser pour envoyer des bitcoins directement vers votre portefeuille. Contrairement aux adresse email, vous avez la possibilité de créer de nouvelles adresses aussi souvent que vous le souhaitez, et toutes ces adresses dirigeront les fonds vers votre portefeuille. Un portefeuille est simplement une collection d'adresses et de clés permettant de débloquer les fonds qui y sont contenus. Vous pouvez augmenter votre anonymat en utilisant une nouvelle adresse pour chacune de vos transactions. Il n'y a pratiquement aucune limite au nombre d'adresses qu'un utilisateur peut créer.

Alice est maintenant prête à utiliser son nouveau portefeuille bitcoin.

Obtenir vos premiers bitcoins

Il n'est pas possible d'acheter des bitcoin à la banque ou dans un kiosque à devises pour le moment. En

2014, il est toujours assez difficile d'acquérir des bitcoins dans la plupart des pays. Il existe plusieurs échangeurs de devises spécialisés où vous pouvez acheter et vendre des bitcoins en échanges d'une devise locale. Voici des échanges opérant en ligne actuellement :

Bitstamp

Un échangeur de devises qui accepte plusieurs monnaies dont l'euro (EUR) et les dollars US (USD) par virement bancaire.

Coinbase

Un portefeuille bitcoin et une plateforme basée aux Etats-Unis où les marchands et les consommateurs peuvent effectuer des transactions en bitcoins. Coinbase facilite l'achat et la vente de bitcoins, en permettant aux utilisateurs d'y lier leurs comptes en banque américains via le système ACH.

Les échangeurs de crypto-monnaies de la sorte opèrent à l'intersection des devises nationales et des crypto-monnaies. Et de ce fait, ils sont sujets aux législation nationales et internationales, et sont souvent limités à un seul pays ou zone économique en se spécialisant dans les devises nationales concernées. Votre choix d'un échangeur de monnaie sera spécifique à la devise que vous utilisez et limité aux échangeurs opérant au sein de la juridiction de votre pays ou zone économique. Comme l'ouverture d'un compte en banque, cela prend plusieurs jours voire semaines pour mettre en place les comptes nécessaires pour utiliser ces services car ils requièrent plusieurs preuve d'identité pour se conformer((("KYC (Know Your Customer) aux politiques bancaires de KYC (connaître votre client, « know your customer » an anglais) et AML (anti-blanchiment, « anti money laundering » en anglais). Une fois que vous obtenez un compte sur un échangeur bitcoin, vous pouvez acheter et vendre des bitcoins rapidement exactement comme vous l'auriez fait en utilisant un compte de courtage de devises étrangères.

Vous pouvez trouvez une liste plus complète sur bitcoin charts, un site qui renseigne sur le cours du bitcoin et d'autres données concernant les marchés d'une multitude d'échangeurs de devises.

Il existe quatre autres méthodes afin d'obtenir des bitcoins pour un nouvel utilisateur :

•Trouver un ami qui possède des bitcoins et lui proposer d'en acheter directement. Beaucoup d'utilisateurs de bitcoin commencent de cette façon • Utiliser un service tel que localbitcoins.com pour trouver un vendeur dans votre zone géographique afin de lui acheter des bitcoins contre du cash en réalisant la transaction en personne. • Vendre un produit ou un service contre des bitcoins. Si vous être un développeur, vendez vos compétences informatique. • Utiliser un distributeur automatique de bitcoin dans votre ville. Vous pouvez trouver un distributeur de bitcoin automatique près de chez vous en utilisant la carte en ligne de CoinDesk.

Alice à découvert le bitcoin par un ami et donc avait un moyen simple de s'en procurer en attendant l'activation et la validation de son compte sur un échangeur californien.

Envoyer et Recevoir des Bitcoins

Alice a créé son portefeuille bitcoin et est maintenant prête à recevoir des fonds. Son application de

portefeuille a généré de façon aléatoire une clé privée (décrite en détail dans [private_keys]) ainsi qu'une adresse bitcoin correspondante. À ce stade, son adresse bitcoin n'est pas connue du réseau bitcoin et n'est « enregistrée » nulle part dans le système bitcoin. Son adresse bitcoin est simplement un nombre correspondant à une clé qu'elle peut utiliser pour contrôler l'accès à ses fonds. Il n'y a pas de compte ou d'association entre cette adresse et un compte. Tant que cette adresse n'est pas référencée comme destinataire de valeur dans une transaction envoyée sur le registre bitcoin (la blockchain), elle n'est qu'une adresse parmi toutes les adresses valides possibles de bitcoin. Une fois associée à un transaction, elle fait partie des adresses connues sur le réseau et Alice peut alors consulter son solde sur le registre public.

Alice rencontre son ami Joe, qui lui a fait découvrir le bitcoin, dans un restaurant du coin afin qu'ils puissent échanger des dollars US et mettre quelques bitcoins sur son compte. Elle a besoin d'apporter une impression de son adresse et du code QR comme affiché dans son portefeuille bitcoin. Cette adresse ne représente rien de sensible, du point de vue de la sécurité. Elle peut être affichée n'importe où sans mettre la sécurité de son compte en danger.

Alice veut juste convertir 10 dollars US en bitcoins car elle ne souhaite pas risquer un montant trop élevé d'argent dans une nouvelle technologie. Elle donne donc à Joe un billet de $10 et son adresse imprimée afin que Joe lui envoie l'équivalent en bitcoins.

Ensuite, Joe doit calculer le taux d'échange actuel afin de donner le montant correct de bitcoins à Alice. Il existe des centaines d'applications et de sites web permettant de consulter le taux d'échange en temps réel. Voici les plus populaires :

Bitcoin Charts

: Un service de listing de données de marché qui montre le taux du bitcoin sur plusieurs échangeurs du monde entier, exprimé dans de multiples devises

Bitcoin Average

: Un site web qui fournit une vue simple du taux moyen pondéré par le volume pour chaque devise.

ZeroBlock

: Une application gratuite sous Android et iOS qui affiche le prix du bitcoin sur différents échangeurs (voir ZeroBlock, une application pour obtenir le taux du bitcoin pour Android et iOS)

Bitcoin Wisdom

: Un autre service de listing de données de marché en temps réel.

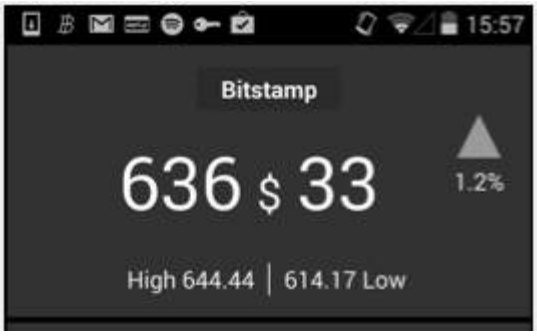


Figure 3. ZeroBlock, une application pour obtenir le taux du bitcoin pour Android et iOS

En utilisant une de ces applications ou sites web, Joe en conclut que le prix du bitcoin correspond à approximativement 100 dollars US pour un bitcoin. A ce taux il doit donc donner à Alice 0.10 bitcoin, ou 100 millibits en échange des 10 dollars US qu'elle lui a donné.

Une fois que Joe a trouvé le juste taux d'échange, il ouvre un client mobile bitcoin et sélectionne l'action « Envoyer » des bitcoins. Par exemple, si il utilisait le client mobile Blockchain sur un téléphone Android, il verrait un écran avec deux champs comme montré dans L'écran d'envoi de bitcoin de l'application du portefeuille mobile Blockchain.

- L'adresse bitcoin de destination pour la transaction
- Le montant de bitcoin à envoyer

A coté du champ correspondant à l'adresse bitcoin, il y a une petite icône ressemblant à un code QR. Elle permet a Joe de scanner le code QR à l'aide de la caméra de son smartphone afin de ne pas avoir à taper à la main l'adresse bitcoin d'Alice (1Cdid9KFAaatwczBwBttQcwXYCpvK8h7FK), qui est plutôt longue et difficile à taper sans faire d'erreur. Joe tapote donc sur l'icône QR code et active la caméra de son smartphone pour enfin scanner le portefeuille imprimé qu'Alice a apporté avec elle. L'application de portefeuille mobile remplit le champ correspondant à l'adresse bitcoin et Joe peut vérifier que l'adresse a été scannée correctement en comparant quelques caractères de l'adresse avec l'adresse imprimée par Alice.

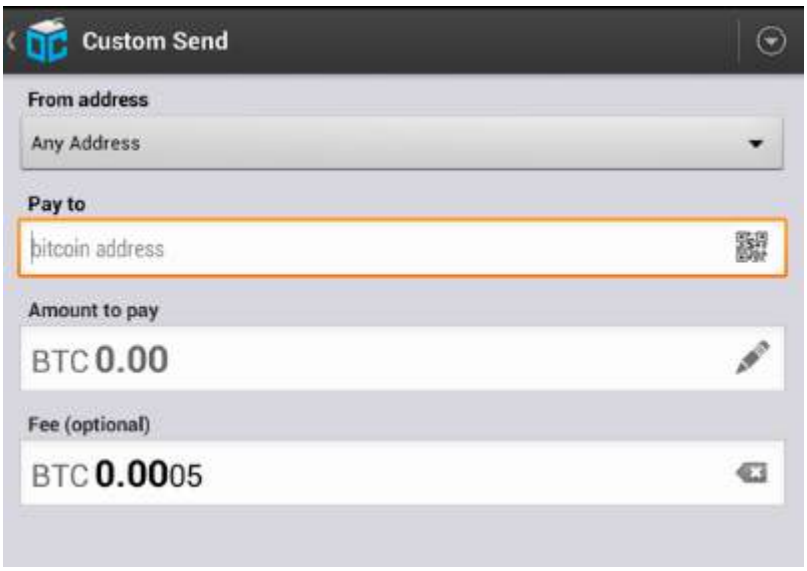


Figure 4. L'écran d'envoi de bitcoin de l'application du portefeuille mobile Blockchain

Joe rentre ensuite le montant de bitcoin pour la transaction soit 0.10 bitcoins. Il vérifie bien ce montant afin d'être sûr de rentrer le montant correct, tout simplement parce qu'il est sur le point de transmettre de l'argent et que la moindre erreur pourrait s'avérer coûteuse. Pour finir, il clique sur Envoyer pour transmettre la transaction. L'application de portefeuille mobile de Joe construit une transaction qui attribue 0.10 à l'adresse bitcoin fournie par Alice, déplaçant les fonds contenus dans le portefeuille de Joe et signant la transaction à l'aide de la clé privée de Joe. Cela signale au réseau bitcoin que Joe a autorisé le transfert de valeur depuis une de ses adresses vers la nouvelle adresse d'Alice. Pendant la transmission de la transaction via le protocole peer-to-peer, elle se propage rapidement sur le réseau bitcoin. En moins d'une seconde, la plupart des nœuds les mieux connectés du réseau reçoivent la transaction et voient l'adresse d'Alice pour la première fois.

Si Alice a un smartphone ou un ordinateur portable avec elle, elle sera aussi capable de voir cette transaction. Le registre bitcoin—un fichier sans cesse grandissant dans lequel sont enregistrées toutes les transactions passées—est public, ce qui veut dire que tout ce qu'elle a à faire est de chercher sa propre adresse et voir si des fonds y ont été envoyés. Elle peut faire cela facilement sur le site web blockchain.info en entrant son adresse dans le champ de recherche. Le site web lui fournira alors une page un listing de toutes les transactions de cette adresse. Si Alice consulte cette page, elle sera mise à jour pour afficher la transaction transférant 0.10 bitcoins vers son compte peu de temps après que Joe les ait envoyé.

Confirmations

Au début, l'adresse d'Alice affichera la transaction provenant de Joe comme « Non confirmée ». Cela veut dire que la transaction à été propagée sur le réseau mais qu'elle n'a pas encore été incluse dans le registre de transaction bitcoin, également appelé la blockchain. Afin d'y être intégrée, la transaction doit être prise en charge par un mineur afin d'être incluse dans un bloc de transactions. Une fois le nouveau bloc de transactions créé, au bout de 10 minutes à peu près, les transactions au sein de ce bloc seront acceptées en tant que « confirmées » sur le réseau et pourront alors être dépensées. Cette transaction est vue de tous instantanément, mais n'est réellement reconnue comme valide par tous une fois incluse dans un nouveau bloc miné.

Alice est maintenant l'heureuse propriétaire de 0.10 bitcoins qu'elle peut désormais dépenser. Dans le chapitre suivant nous décrirons son premier achat en bitcoins, et nous analyserons la transaction sous-jacente et les techniques de propagation en détail.

Fonctionnement de Bitcoin

Transaction, Blocs, Minage, et la Blockchain

Le système bitcoin, contrairement aux systèmes bancaires et de paiement traditionnels, est basé sur une confiance décentralisée. À la place d'une autorité centrale, avec bitcoin, la confiance est une résultante des interactions des différents participants dans le système bitcoin. Dans ce chapitre, nous allons faire un examen haut-niveau de bitcoin en suivant une transaction à travers le système, et constater comment elle devient acceptée par le mécanisme de consensus distribué et finalement enregistrée dans la blockchain, le livre de compte distribué de toutes les transactions.

Chaque exemple est basé sur une transaction réelle ayant eu lieu sur le réseau bitcoin, simulant les interactions entre les utilisateurs (Joe, Alice et Bob) par l'envoi de fonds d'un portefeuille à un autre. Tout en suivant une transaction à travers le réseau bitcoin, nous utiliserons un *explorateur de blockchain* pour visualiser chaque étape. Un explorateur de blockchain est une application web qui se comporte comme un moteur de recherche pour bitcoin, en permettant de chercher des adresses, transactions et blocs et d'observer les relations et flux entre eux.

Les explorateurs de blockchain les plus connus sont :

- Blockchain info
- Bitcoin Block Explorer
- insight
- blockr Block Reader

Chacun d'eux a une fonction de recherche qui peut prendre en paramètre une adresse, un hash de transaction, ou un numéro de bloc et trouver les données correspondantes sur le réseau Bitcoin et la blockchain. Avec chaque exemple, nous allons fournir une URL qui vous emmène directement à la page correspondante, afin que vous puissiez l'étudier en détail.

Aperçu de Bitcoin

Dans le schéma d'ensemble décrit dans Bitcoin overview, on voit que le système de Bitcoin est constitué d'utilisateurs avec des portefeuilles contenant des clés, de transactions qui se propagent à travers le réseau, et de mineurs qui produisent (par calcul concurrentiel) le consensus de la chaîne de bloques, qui est le grand livre de compte faisant autorité de toutes les transactions. Dans ce chapitre, nous allons suivre une seule transaction, comment elle se déplace à travers le réseau et allons examiner les interactions entre chaque partie du système Bitcoin, à un niveau élevé. Dans les chapitres suivants nous nous plongerons dans la technologie présente derrière les portefeuilles, le minage, et les systèmes marchands.

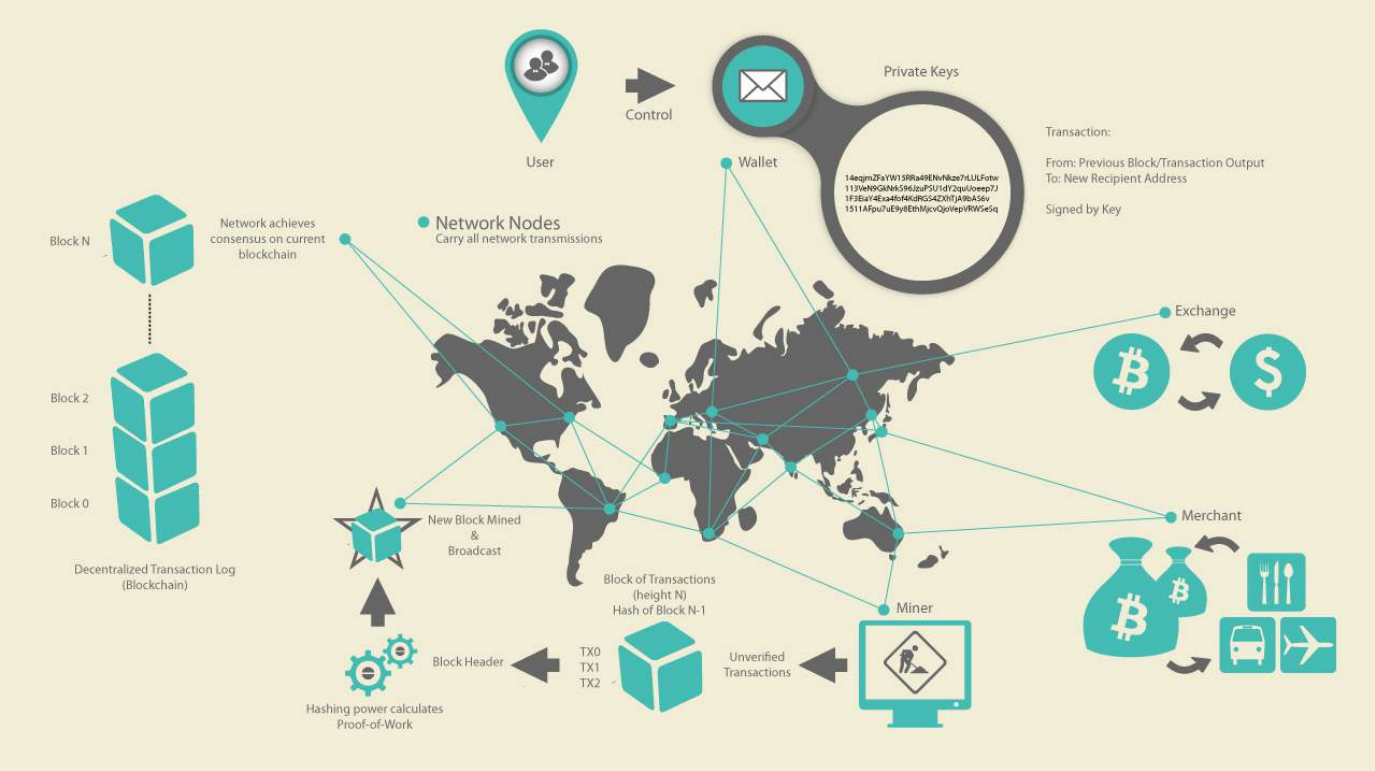


Figure 1. Bitcoin overview

Acheter un Café

Alice, qu'on a présentée dans le chapitre précédent, est une nouvelle utilisatrice qui vient tout juste d'acquérir ses premiers bitcoins. Dans [getting_first_bitcoin], Alice a donné rendez-vous à son ami Joe et a échangé avec lui de l'argent liquide contre des bitcoins. La transaction créée par Joe a alimenté le portefeuille d'Alice avec 0.10 BTC. Alice va maintenant réaliser son premier achat dans un commerce, en payant pour un café au Bar de Bob à Palo Alto en Californie. Le bar de Bob a récemment commencé à accepter les paiements bitcoin, en ajoutant une option à son terminal de paiement. Les prix dans le café de Bob sont listés dans la monnaie locale (dollar US), mais à la caisse, les clients ont le choix de payer soit en dollars, soit en bitcoins. Alice commande un café et Bob entre la transaction dans sa caisse. Le terminal de paiement va convertir le prix total du dollars en bitcoins au cours actuel du marché, et va afficher les prix dans les deux monnaies, avec un QR code contenant une *requête de paiement* pour cette transaction (voir QR code de la requête de paiement (Astuce: Essayez de le scanner !)) :

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

Figure 2. QR code de la requête de paiement (Astuce: Essayez de le scanner !)

Le QR code de la requête de paiement contient l'URL suivante, définie dans la BIP0021 :

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

Détail de cette URL

Une adresse bitcoin: "1GdK9UzpHBzqzX2A9JFP3Di4weBwqgmoQA"
Le montant du paiement: "0.015"
Un label pour l'adresse du destinataire: "Le Café de Bob"
Une description pour le paiement: "Commande au café de Bob"
```

TIP Contrairement à un QR code qui contient simplement une adresse bitcoin, une requête de paiement est une URL encodée comme QR code qui contient une adresse de destination, un montant, et une description telle que "Café de Bob." Cela permet à l'application de portefeuille Bitcoin de pré-remplir les détails du paiement et d'afficher une description à l'utilisateur. Vous pouvez scanner le QR code avec votre application de portefeuille pour voir ce qu'Alice verrait.

Bob : «Cela fait un dollar cinquante, soit quinze millibits."

Alice utilise son smartphone pour scanner le code barre sur l'écran de la caisse. Son smartphone affiche un paiement de 0.0150 BTC vers Café de Bob et elle sélectionne Envoyer pour autoriser le paiement. En quelques secondes (environ la même durée que pour une carte bancaire), Bob verra la transaction sur son terminal, et pourra finir la commande.

Dans les sections suivantes, nous examinerons cette transaction en détails : la manière dont le portefeuille d'Alice a validé la transaction, comment cette transaction est propagée au sein du réseau, la manière dont elle est vérifiée, et enfin comment Bob peut, grâce à de futures transactions, dépenser cet argent.

NOTE

Le réseau bitcoin gère des transactions en fractions de bitcoin, par exemple en milli-bitcoin (1/1000ème de bitcoin) ou même en satoshis (1/100 000 000ème de bitcoin). Dans ce livre le terme "bitcoin" se réfère à n'importe quelle quantité de monnaie bitcoin, de la plus petite unité (1 satoshi) jusqu'au nombre total (21 000 000) de bitcoins qui seront minés.

Les transactions Bitcoin

En termes simples, une transaction dit au réseau que le propriétaire d'un certain nombre de bitcoins a autorisé leur transfert vers un autre propriétaire. Le nouveau propriétaire peut alors dépenser ces bitcoins en créant une nouvelle transaction qui autorise le transfert vers un autre propriétaire, et ainsi de suite, tout au long d'une chaîne de propriété.

Les transactions sont similaires aux lignes dans les livres de compte à partie double (ou double entrée). En termes simples, chaque transaction contient une ou plusieurs entrées (inputs), équivalentes à des débits sur un compte bitcoin , et une ou plusieurs sorties (outputs) équivalentes à des crédits vers un compte bitcoin. Les sommes des entrées et des sorties ne sont pas forcément égales, et la différence représente les "frais de transaction" : une petite somme récupérée par le mineur qui ajoute la transaction au livre de compte bitcoin (la "blockchain"). [transaction-double-entry] montre une transaction bitcoin représentée comme une ligne dans un livre de comptes.

La transaction comprend également une preuve de propriété pour chaque montant de bitcoin (entrées) dont la valeur est transférée, sous la forme d'une signature numérique du propriétaire, qui peut être validée de façon indépendante par quiconque. Dans le langage bitcoin, «dépenser» c'est signer une transaction qui transfère la valeur d'une transaction précédente à un nouveau propriétaire identifié par une adresse bitcoin.

TIP

Les *transactions* transfèrent des fonds depuis leurs *entrées* vers leurs *sorties*. Une entrée représente la provenance des fonds, en général la sortie d'une transaction précedente. Une sortie matérialise le transfert des fonds en les associant à une clef, qui représente une contrainte permettant de vérouiller les fonds en specifiant le type de signature qui sera nécessaire pour les dépenser. Les sorties d'une transaction peuvent être utilisés en entrée de nouvelles transactions, formant ainsi une chaîne de propriété qui représente les transferts d'adresse en adresse (voir Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante).

1. Transaction vues comme les entrées double d'un livre de compte image::images/msbt_0203.png["Transaction Double-Entry"]

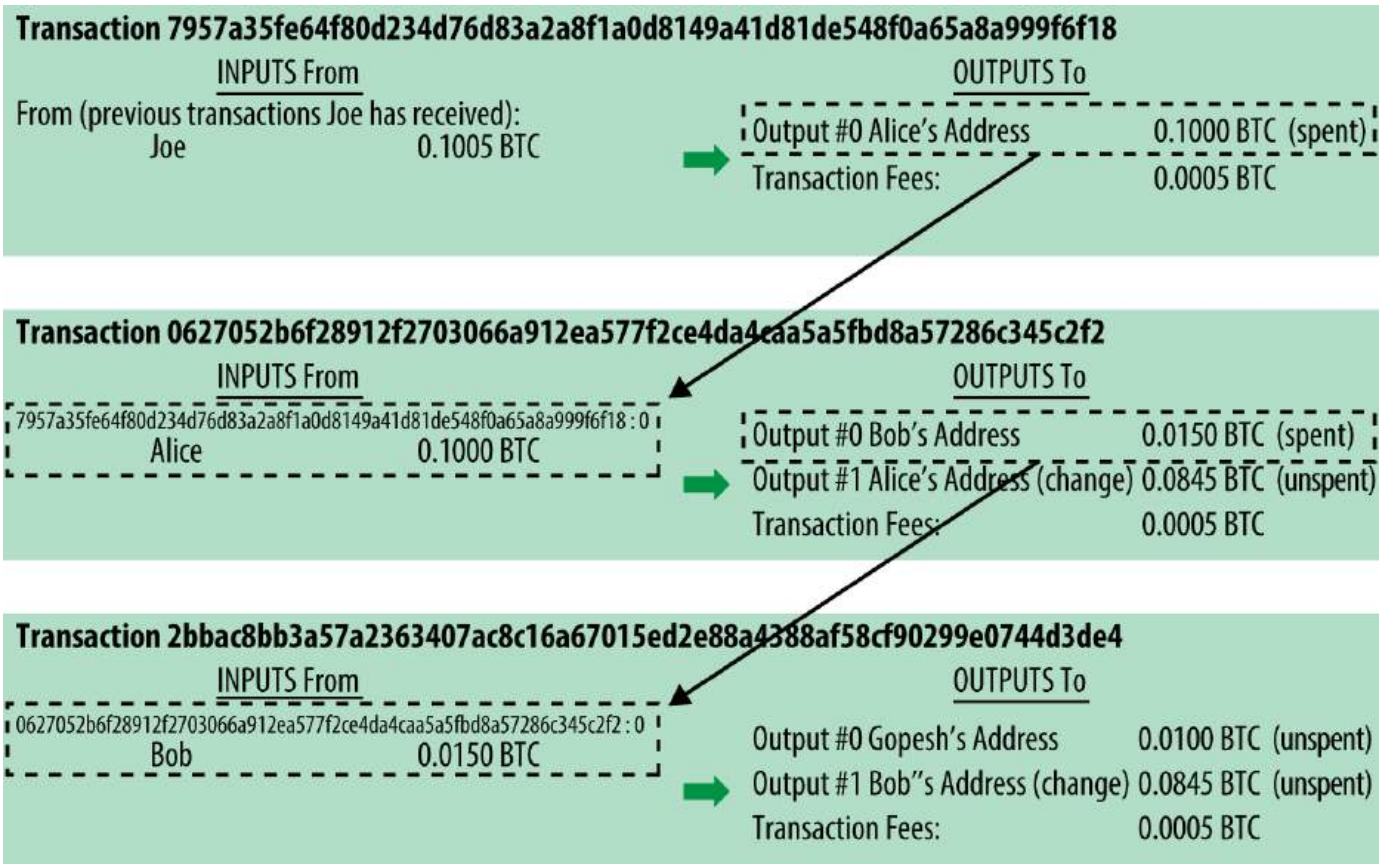


Figure 3. Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante

Le paiement d'Alice au Café de Bob utilise comme entrée une transaction antérieure. Au chapitre précédent, Alice avait reçu des bitcoin de son ami Joe en échange de cash. Cette transaction a "verrouillé" un certain nombre de bitcoin, utilisables seulement avec la clef d'Alice. Sa nouvelle transaction pour payer Bob utilise cette transaction en entrée et crée de nouvelles sorties pour payer le café et recevoir la monnaie. Les transactions forment une chaîne, les entrées des dernières transactions correspondent aux sorties des transactions précédentes. La clef d'Alice permet de créer une signature qui déverrouille les sorties précédentes, prouvant ainsi au réseau bitcoin que c'est elle qui détient ces fonds. Elle lie ce paiement à l'adresse de Bob, ce qui verrouille les fonds qui ne peuvent être utilisés que si Bob fournit une signature valable. Cela représente un transfert de valeur de Alice vers Bob. Cette chaîne de transaction, de Joe vers Alice puis Bob, est illustrée dans Une chaîne de transactions, où la sortie d'une transaction est l'entrée de la transaction suivante.

Les formes communes de Transaction

La transaction la plus courante est un simple paiement d'une adresse vers une autre, qui inclue souvent un peu de "change" renvoyé vers l'expéditeur. Ce type de transaction a une entrée et deux sorties et est illustrée ici : La transaction la plus commune.

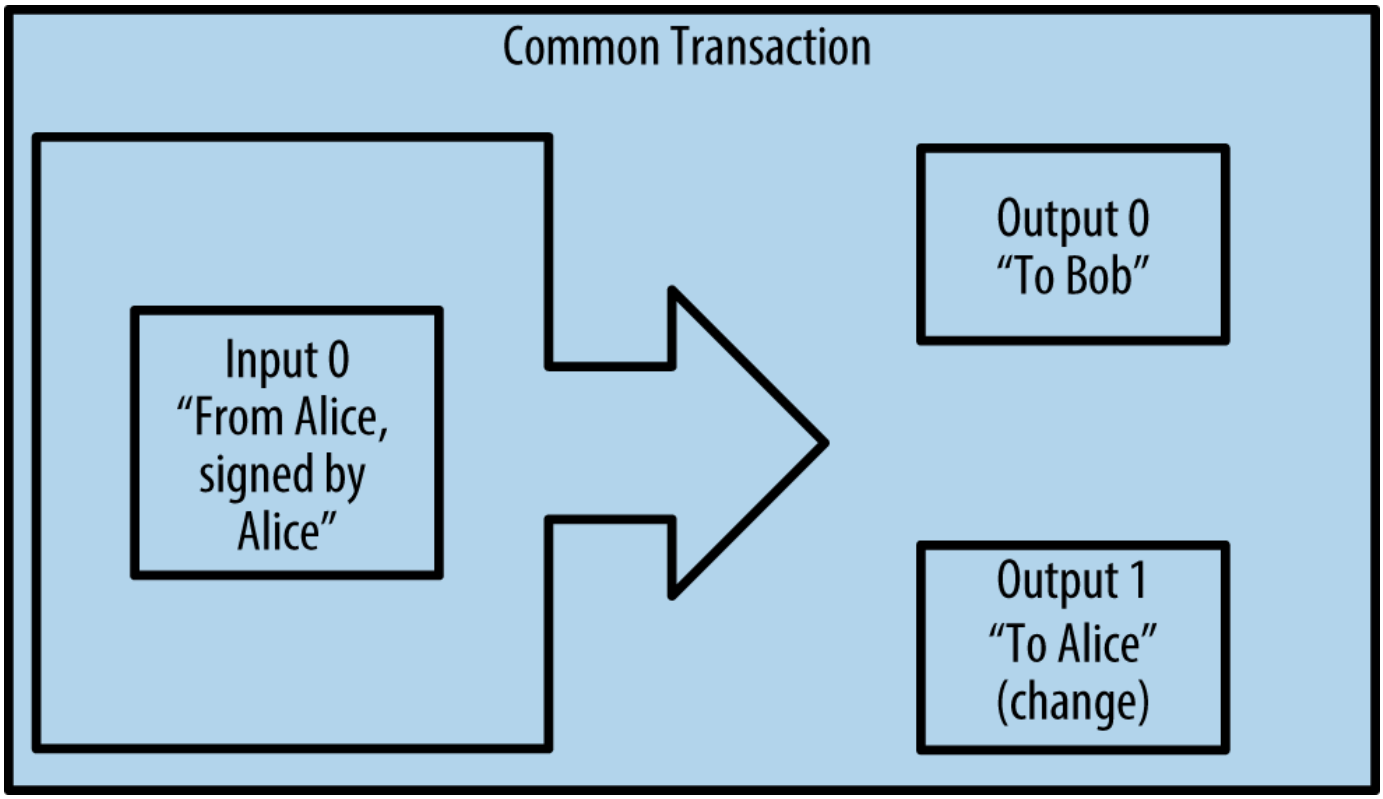


Figure 4. La transaction la plus commune

Un autre type de transaction courant est l'agrégation de plusieurs entrées vers une sortie unique (voir Transaction agrégeant des fonds), ce qui est équivalent à échanger un ensemble de pièces et de billets contre un seul billet plus gros. Ce type de transactions est parfois généré par les portefeuilles pour nettoyer la multitude de petites sommes reçues en tant que "change".

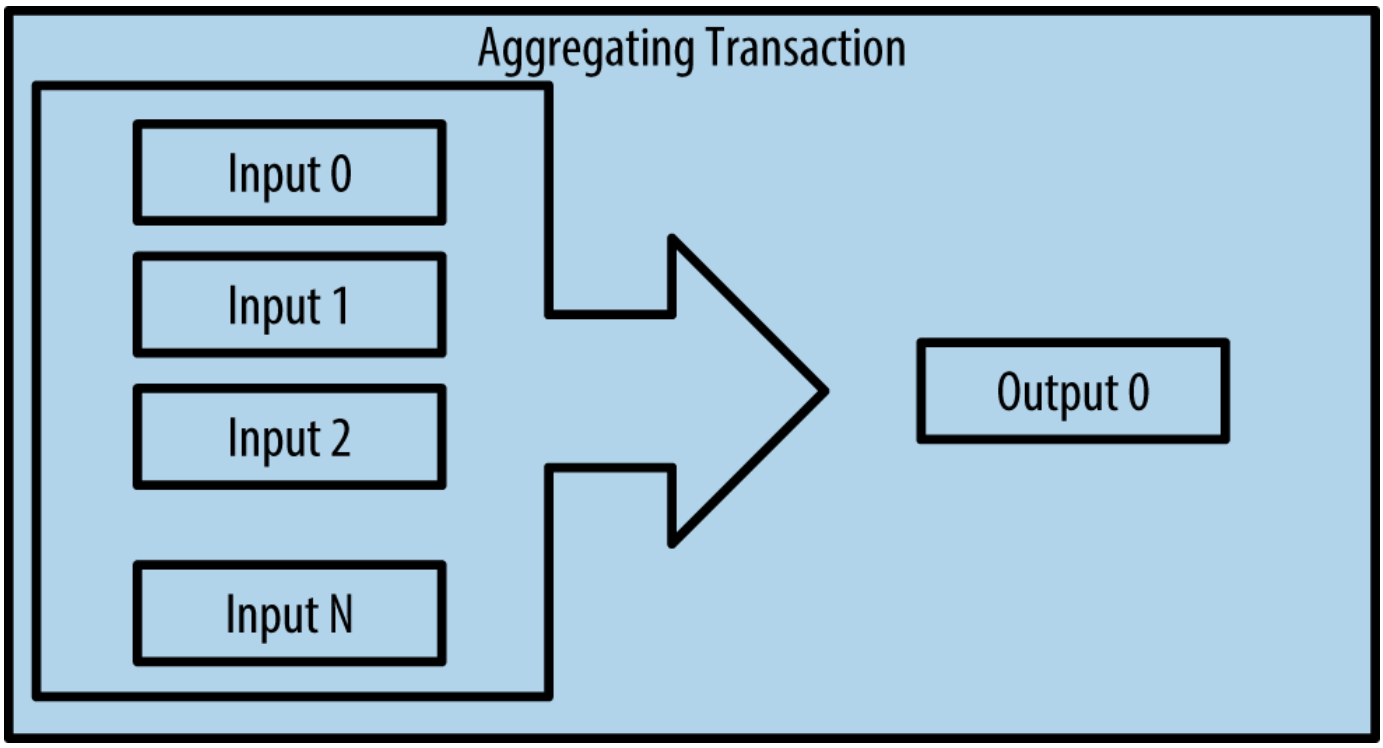


Figure 5. Transaction agrégeant des fonds

Enfin, un autre type de transaction courant distribue une entrée vers de nombreuses sorties qui représentent plusieurs destinataires (voir Transaction permettant de répartir des fonds.). Ce type de transaction est parfois utilisé par des entités commerciales pour distribuer des fonds, par exemple la paye des employés . range="endofrange", startref="ix_ch02-asciidoc3")

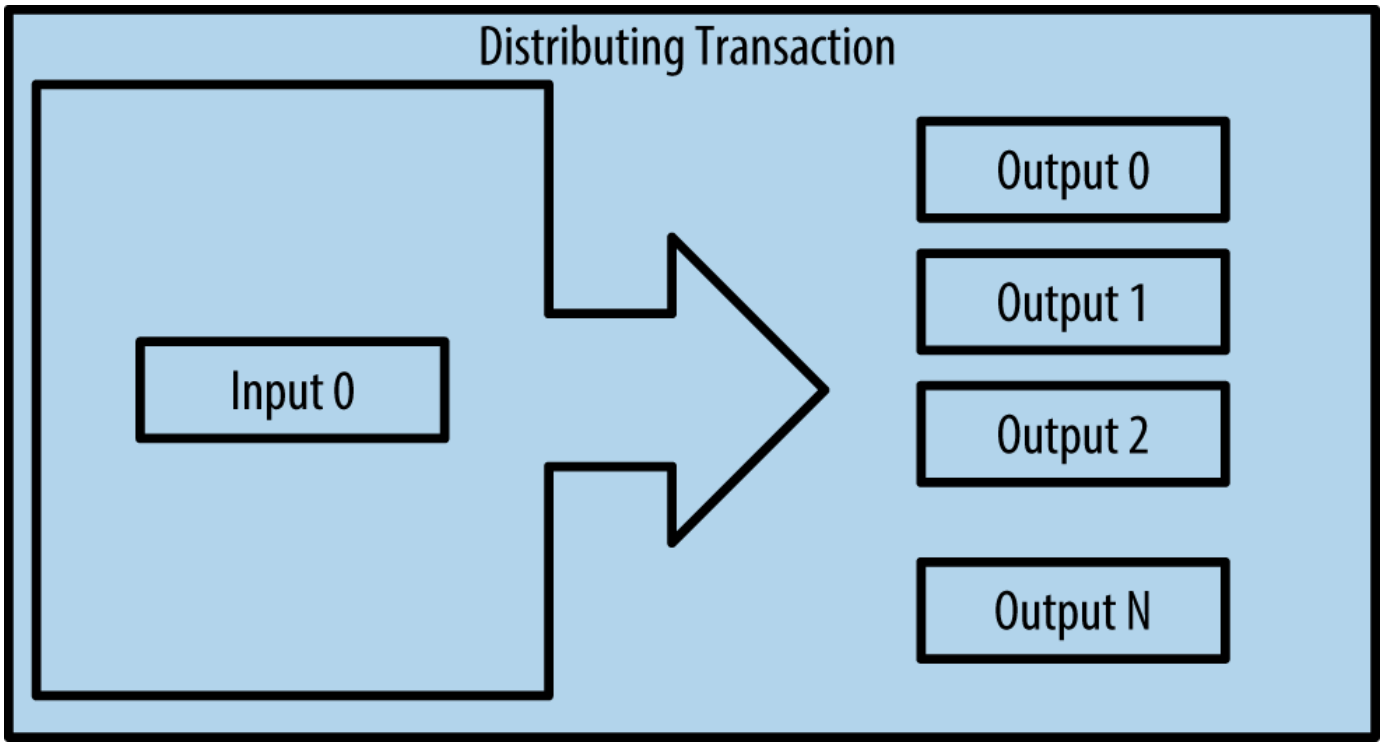


Figure 6. Transaction permettant de répartir des fonds.

Construire une Transaction

L'application portefeuille d'Alice contient les algorithmes permettant de sélectionner les entrées et sorties afin de construire des transactions conformes aux choix d'Alice. Elle n'a plus qu'à choisir une destination et un montant et le portefeuille fait le reste sans lui montrer les détails du processus. Il est important de noter qu'un portefeuille peut construire des transactions même s'il n'est pas connecté. De la même façon qu'un chèque peut être rédigé chez soi et posté à la banque plus tard, une transaction n'a pas besoin du réseau bitcoin pour être construite et signée. Il faut seulement qu'elle soit ensuite envoyée vers le réseau bitcoin pour être traitée.

Sélectionner les bonnes entrées

Le portefeuille d'Alice doit d'abord trouver les entrées qui peuvent correspondre au montant qu'elle veut envoyer vers Bob. La plupart des portefeuilles maintiennent une liste des "sorties non-dépensées" qui sont verrouillées avec une des clefs du portefeuille. Ici, le portefeuille contiendrait une copie de la sortie de la transaction de Joe (qu'il a envoyé contre du cash, voir [getting_first_bitcoin]). Un portefeuille fonctionnant en mode "index complet" ira jusqu'à maintenir une copie de toutes les sorties non-dépensées de l'ensemble des transactions de la blockchain. Cela permet de sélectionner les entrées mais aussi de vérifier rapidement que les transactions reçues par le portefeuille ont des entrées valides. Néanmoins, vu que le mode "index complet" utilise beaucoup d'espace disque, la plupart des

utilisateurs choisissent des portefeuilles "légers" qui ne gardent que les sorties non-dépensées correspondant aux clefs de l'utilisateur.

Si le portefeuille ne maintient pas de copie des sorties non-dépensées, il peut les demander au réseau bitcoin, soit en utilisant l'API d'un des nombreux services et explorateurs, soit en utilisant l'API JSON RPC d'un nœud bitcoin en mode "index complet". [example_2-1] illustre une requête REST, construite comme une requête HTTP GET vers une URL spécifique.Cette URL renverra toutes les sorties non-dépensées liées à une adresse, permettant ainsi de construire les entrées en sélectionnant les "bonnes" sorties à dépenser. Nous utilisons le client HHTP en ligne de commande *cURL* pour récupérer la réponse.

1. Recherche des sorties non-dépensées pour les adresses d'Alice

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

Example 1. Réponse pour cette recherche

```
{

    "unspent_outputs":[

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

    ]
}
```

Dans Réponse pour cette recherche la réponse montre une sortie non dépensée (qui n'a pas encore été utilisée) appartenant à l'adresse de 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, de 10 millions, ce qui équivaut à 0,10 Bitcoin. Avec cette information, l'application de portefeuille d'Alice peut construire une transaction pour transférer cette valeur à de nouvelles adresses appartenant à d'autres propriétaires.

TIP Voir transaction from Joe to Alice.

Comme vous pouvez le constater, il existe une sortie non-dépensée dans le porte-monnaie d'Alice qui contient assez de bitcoin pour payer son café. Si cela n'avait pas été le cas, l'application aurait du "fouiller" dans l'ensemble des sorties non-depensées, de la même façon que l'on cherche des pièces lorsque l'on rend la monnaie. Dans tous les cas il peut être nécessaire de spécifier, dans la nouvelle transaction que crée le porte-monnaie, qu'il faut récuperer un peu de change. C'est ce que nous verrons dans la prochaine section.

Création des sorties

La sortie d'une transaction se présente sous la forme d'un script qui vérouille le montant de cette sortie, qui ne pourra être collecté que si l'on fournit une solution à ce script. De façon simplifié, la sortie de la transaction d'Alice contient unn script qui signifie "cette sortie sera payée à celui qui fournira une signature valide pour la clef correspondant à l'adresse publique de Bob". Bob étant la seule personne possédant les 2 clefs (publique et privée) correspondant à cette adresse, lui seul pourra fournir une telle signature. Alice va donc vérouiller cette sortie en exigeant une signature provenant de Bob.

Cette transaction contiendra une deuxième sortie, car Alice ne possède qu'une sortie non-dépensée d'un montant de 0.10 BTC, supérieur au prix du café de 0.015 BTC. Alice a donc besoin de récuperer 0.085 BTC de change. Son porte-monnaie va donc ajouter une deuxième sortie à sa transaction, qui contiendra donc 2 sorties: un paiement vers Bob, et un paiement vers elle-même qu'elle pourra dépenser plus tard.

Enfin, pour que la transaction soit traitée suffisamment vite par le réseau, le porte-monnaie d'Alice va y ajouter de petits frais de transaction. Le montant de ces frais n'est pas explicitement précisé dans la transaction, mais se calcule en faisant la différence entre les entrées et les sorties. Si le montant de la sortie de change crée par Alice est de 0.0845 au lieu de 0.085, cette différence vaudra 0.10 BTC (somme des entrées) - 0.015 (paiement pour Bob) - 0.0845 = 0.0005 BTC (un demi milli-bitcoin). Cette somme sera collectée par le mineur qui ajoutera à la blockchain le bloc contenant cette transaction.

Cette transaction peut être affichée avec un explorateur de blockchain, voir La transaction d'Alice au café de Bob.

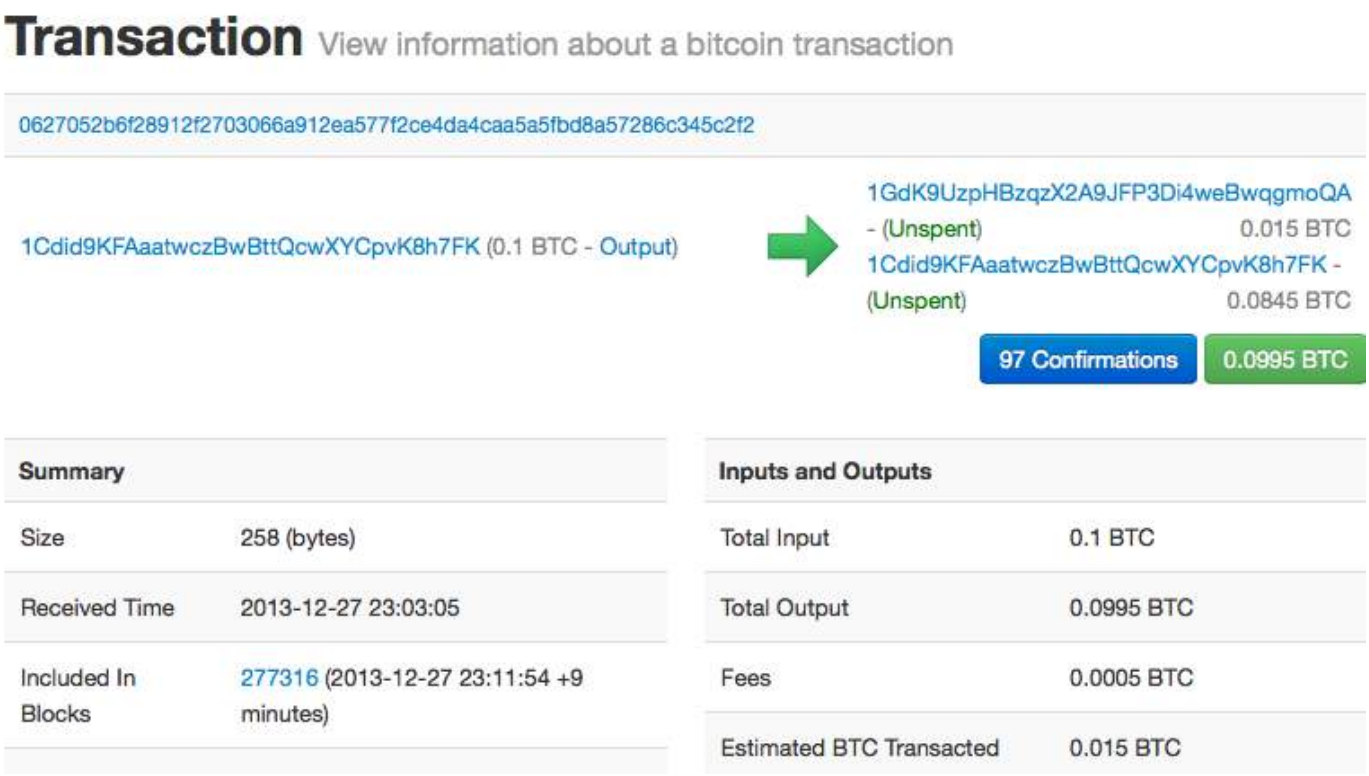


Figure 7. La transaction d'Alice au café de Bob

TIP Voir transaction from Alice to Bob's Cafe.

Ajout de la transaction au livre de compte

La transaction créée par Alice fait 258 octets et contient tout ce qui nécessaire à prouver qu'elle détient les fonds et peut les transférer vers de nouveaux détenteurs. Maintenant, il faut transmettre cette transaction au réseau bitcoin pour qu'elle soit ajoutée au livre de comptes distribué (la blockchain). Dans la prochaine section nous verrons comment une transaction est ajoutée à un bloc et comment ce bloc est "miné". Enfin nous verrons comment l'indice de confiance que le réseau accorde à ce bloc, une fois qu'il est ajouté à la blockchain, augmentera avec l'ajout de nouveaux blocs.

Transmettre la transaction

Une transaction contient toutes les informations nécessaires à son traitement, et la façon dont elle est transmise au réseau n'a aucune importance. Le réseau bitcoin est un réseau peer-to-peer, chaque noeud étant connecté à plusieurs autres noeuds. L'objectif du réseau est de propager les blocs et transactions à tous les membre de ce réseau.

Comment elle se propage

Le porte-monnaie d'Alice peut envoyer la nouvelle transaction à tous les clients bitcoin auxquels il est connecté, via n'importe quelle connection Internet: filaire, WiFi, mobile. Son porte-monnaie n'a pas besoin d'être directement connecté à celui de Bob et elle n'est pas obligée d'utiliser la connection internet de son café (mais ces 2 options sont possibles et ne posent pas de problèmes). N'importe quel

noeud (client) du réseau bitcoin qui reçoit une transaction valide qu'il n'a pas déja vu va immédiatement la propager à tous les noeuds auxquel il est connecté. De cette façon, les transactions se propagent rapidement sur le réseau peer-to-peer et atteignent en quelques secondes un pourcentage élevé des noeuds.

Ce que voit Bob

Si le porte-monnaie de Bob est directement connecté à celui d'Alice, ce sera peut-être le premier noeud à recevoir la transaction. Mais même si ce n'est pas le cas et que la transaction se propage à travers d'autres noeuds, elle attendra le porte-monnaie de Bob en quelques secondes. Elle sera immédiatement identifée comme un paiement vers Bob, parce qu'elle contient des sorties que les clefs de Bob peuvent dépenser. Le porte-monnaie de Bob peut aussi vérifier de façon indépendante que la transaction est bien construite, utilise des entrées non depensées et contient des frais de transaction suffisant pour être ajoutée au prochain bloc. Bob peut alors estimer qu'elle a de grandes chances d'être bientôt ajoutée à un bloc et confirmée.

TIP

Une idée fausse mais assez répandue est qu'il faut attendre 10 minutes pour qu'une transaction soit confirmée, voir jusqu'à 60 minutes pour 6 confirmations. Les confirmations permettent de s'assurer qu'une transaction a été acceptée par l'ensemble du réseau mais ne sont pas nécessaires pour de petits paiement, comme un café. Un marchand peut accepter de petits paiements sans confirmations, comme il le fait régulièrement pour des paiements par carte bancaire sans présentation de pièce d'indentité ou de signature.

Le minage de Bitcoin

La transaction se propage sur le réseau bitcoin, mais ne sera ajoutée au livre de compte distribué (la *blockchain*) que lorsqu'elle sera vérifiée et ajoutée à un bloc lors d'une opération appelée minage. Voir [ch8] pour une explication détaillée.

Le système de confiance utilisé par bitcoin se base sur le calcul. Les transactions sont regroupées par blocs dont la construction demande énormément de calculs mais dont la véfification est très simple. Le processus de minage a 2 objectifs:

- Le minage crée de nouveaux bitcoins dans chaque bloc, presque comme une banque centrale imprimant de la nouvelle monnaie. Le nombre de bitcoin créé par bloc est fixe et diminue avec le temps.
- Le minage crée de la confiance en assurant que les transactions ne sont confirmées que si un nombre de calculs suffisant a été dédié à la construction du bloc qui les contient. Encore plus de blocs signifie plus de calculs, dont plus de confiance.

Pour décrire le minage, on pourrait utiliser l'exemple d'un gigantesque concours de sudoku où les participants recommence une nouvelle grille dès que quelqu'un trouve une solution, et dont la difficulté des grilles s'ajuste pour qu'en moyenne une grille soit résolue toutes les 10 minutes. Imaginons une grille géante de sudoku, avec plusieurs milliers de lignes et de colonnes. Il serait assez

facile de vérifier qu'une grille terminée est bien remplie. Mais si seulement un petit nombre de cases a été rempli, le reste étant vide, il faudra beaucoup de travail pour la terminer! La difficulté des grilles peut être ajustée en changeant leurs tailles (en ajoutant ou en enlevant des lignes et de colonnes), mais elles peuvent toujours être vérifiées facilement même si elles sont très grandes. Les puzzles à résoudre dans le réseau bitcoin se basent sur les hash cryptographiques et présentent les mêmes caractéristiques que ces grilles de sudoku: ils sont très difficiles à résoudre mais il est très facile de vérifier qu'une solution est bonne, et leur difficulté peut être ajustée.

Au chapitre [user-stories], nous avons rencontré Jing, étudiant en informatique à Shangai. Jing participe au réseau bitcoin en minant. Toutes les 10 minutes environ, Jing participe à une gigantesque course avec des milliers d'autres mineurs afin de trouver une solution à un bloc de transactions. Trouver une telle solution, appelée "preuve de travail", demande à l'ensemble du réseau d'effectuer des quadrillions d'opérations de hash toutes les secondes. L'algorithme de cette preuve de travail demande à calculer en boucle le hash de l'entête d'un bloc et d'un nombre aléatoire avec l'alrogithme SHA256 jusqu'à ce qu'une solution compatible avec une cible prédéterminée soit trouvée. Le premier mineur qui trouve cette solution gagne la compétition et ajoute le bloc à la blockchain.

Jing a commencé à miner en 2010 avec un ordinateur de bureau rapide. Avec l'arrivée de nouveaux mineurs, la difficulté de minage a augmenté rapidement. Très vite Jing les autres mineurs ont commencé à utiliser du matériel spécialisé, comme des cartes graphiques haut de gamme (GPU) comme celles utilisées dans les PCs et consoles de jeu. Au moment ou j'écris ce texte, la difficulté est tellement élevé que le minage n'est rentable qu'avec des ASIC (Application Specific Integrated Circuits), des circuits imprimés sur mesure qui implémentent des centaines d'algorithmes de minage fonctionnant en parallèle sur une même puce. Jinr a aussi rejoint un pool de minage, similaire à un pool de loterie où les joueurs de regroupent pour partager les investissements et les gains. Jing utilise maintenant 2 mineurs ASIC connectés via USB pour miner des bitcoins 24h sur 24. Il paie sa consommation électrique en vendant une partie des bitcoins génerés. Son ordinateur fait tourner une copie de bitcoind, le client bitcoin de réference, utilisé comme backend pour son logiciel dédié au minage.

Minage des transactions et génération des blocs

Une transaction se propageant sur le réseau n'est vérifiée que lorsqu'elle est ajoutée au livre de compte distribué, la "blockchain". Toutes les 10 minutes en moyenne, les mineurs construisent un nouveau bloc qui contient toutes les transactions depuis le bloc précédent. Les porte-monnaies et autres applications envoient de nouvelles transactions en permanence vers le réseau. Chaque mineur maintient une liste temporaires de transactions non-vérifiées et y ajoute les transactions qu'il reçoit. Lorsqu'il construit un nouveau bloc, il y ajoute ces transactions non-vérifiées puis essaie de résoudre un problème très compliqué (la preuve de travail) ce qui permet de prouver que ce nouveau bloc est valide. Cette opération est détaillée au chapitre [mining].

Les transactions sont ajoutées à un nouveau bloc selon des règles de priorité liées entre autre aux frais de transactions. Quand un mineur reçoit un bloc du réseau bitcoin, cela veut dire qu'il a perdu la course pour ce bloc et il commence immédiatement une nouvelle course pour en miner un nouveau. Il lui rajoute des transactions, le hash du bloc précedent et commence à calculer la preuve de travail.

Chaque mineur ajoute aussi aux nouveaux blocs une transaction spéciale qui paie une récompense vers l'adresse du mineur (25 BTC aujourd'hui). S'il trouve une solution qui valide ce bloc, il gagne la récompense car ce sera son bloc, avec cette transaction vers son adresse, qui sera ajouté à la blockchain. Jing, qui fait partie d'un pool de minage, a configuré son mineur pour utiliser l'adresse du pool pour l'envoi de cette récompense, qui sera ensuite partagée entre les membres du pool en fonction de leur contribution lors du calcul du bloc.

La transaction d'Alice a été envoyée au réseau bitcoin et ajoutée à la liste des transaction non-vérifiées. Elle contient des frais de transaction suffisamment élevés pour être ajoutée à un nouveau bloc par le pool de mineur de Jing. Environ 5 minutes après son envoi sur le réseau, le mineur ASIC de Jing a trouvé une solution pour un nouveau bloc, le numéro 277316, qui contient 419 autres transactions. Le mineur de Jing a envoyé ce nouveau bloc vers le réseau bitcoin, et il sera validé par d'autres mineurs qui vont commencer la course pour le bloc suivant.

Voici le bloc qui contient Alice's transaction.

Quelques minutes plus tard un nouveau bloc, le numéro 277317, est miné par un autre mineur et s'ajoute à la blockchain, augmentant ainsi la quantité de calculs qu'il a fallut effectuer pour la générer. Comme il se base sur le bloc précedent (277316), il augmente d'autant la confiance que l'on peut accorder à ses transactions. L'ajout d'un nouveau bloc aprés celui contenant une transaction est considéré comme une "confirmation" de cette transaction. Au fur et à mesure que les blocs seront ajoutés il deviendra exponentiellement plus difficile d'annuler cette transaction: on pourra donc lui faire de plus en plus confiance.

On peut voir le bloc numéro 277316 sur le schémaLa transaction d'Alice incluse dans le bloc #277316, qui contient la transaction d'Alice. Il existe à ce moment une chaîne de 277316 blocs, chacun lié au précedent, jusqu'au bloc numéro 0, appelé aussi bloc "génésis" ("genesis bloc"). En prenant l'image d'une pile de bloc, plus sa hauteur augmente plus la difficulté de calcul des blocs augmente. Les blocs qui suivent celui contenant la transaction d'Alice peuvent être considérées comme des confirmations de sa validité. Par convention, on considére qu'un bloc confirmé 6 fois est irrévocable, car il faudrait une énorme puissance de calcul pour le remplacer et calculer 6 autres blocs. Nous examinerons le processus de minage, et comment il sécurise le réseau, au chapitre [ch8].

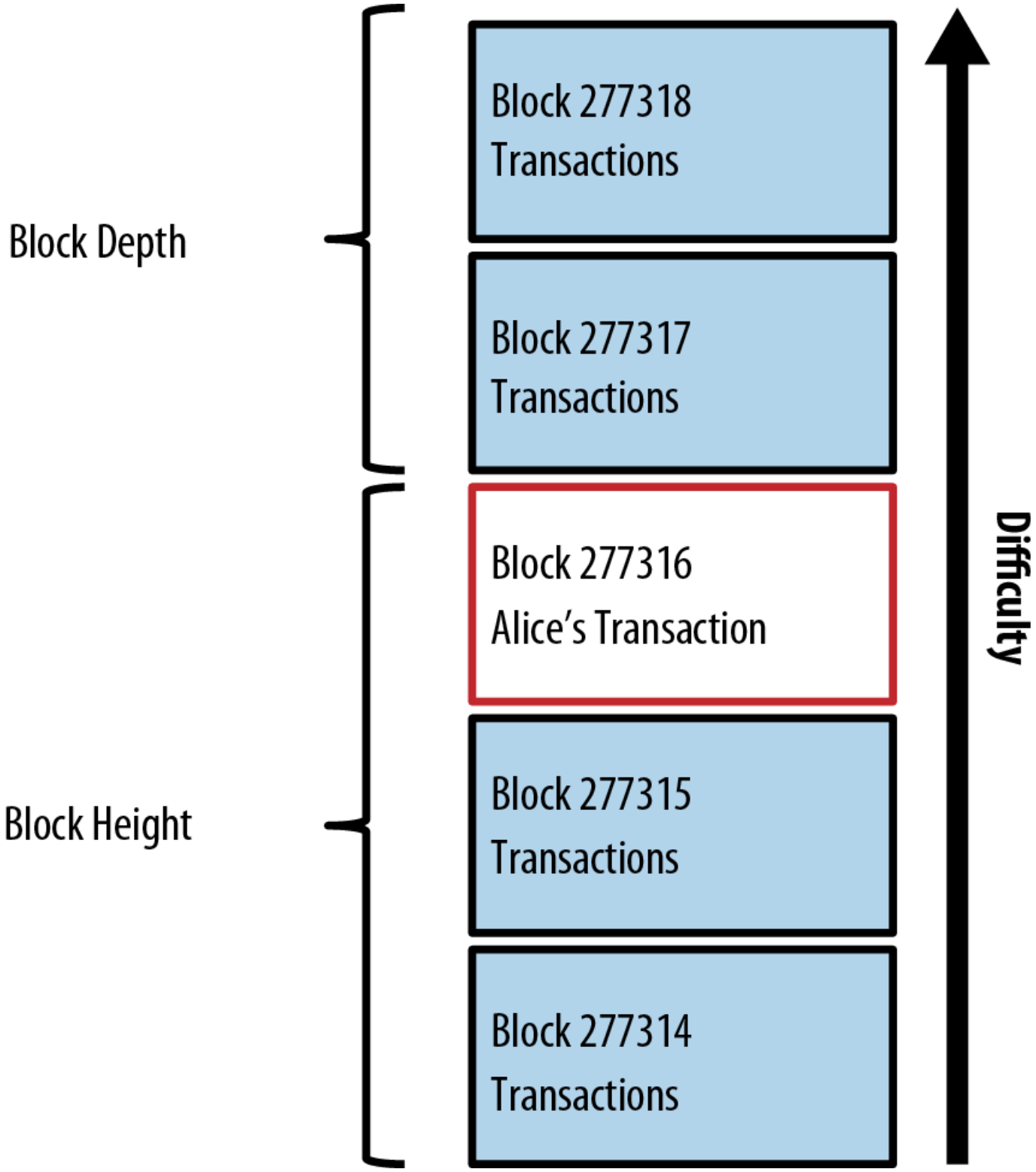


Figure 8. La transaction d'Alice incluse dans le bloc #277316

Dépenser la transaction

La transaction d'Alice fait maintenant partie d'un bloc de la blockchain, le livre de compte distribué et accessible par tous. Chaque client bitcoin peut vérifier de façon indépendante que cette transaction est valide et peut être dépensée. Les clients "full-node" (qui possèdent un indexage complet de la

blockchain) peuvent suivre les fonds, depuis leur création lors de la génération d'un bloc jusqu'au transfert vers l'adresse de Bob. Les clients légers (qui ne possédent pas la blockchain complète) peuvent effectuer une "vérification simplifiée de paiement" (Simplified Payment Verification, ou SPV) en vérifiant que la transaction fait partie de la blockchain et est suivie de plusieurs autres blocs, ce qui permet de s'assurer qu'elle est considérée comme valide par le réseau bitcoin.

Bob peut maintenant dépenser la sortie de cette transaction, et d'autres qui lui sont addressées, en créant une nouvelle transaction qui prend en entrée les sorties qu'il veut dépenser, et les transférer vers d'autres destinataires. Par exemple, Bob peut transférer les fonds envoyés par Alice vers un de ses fournisseurs. En pratique, son client bitcoin va aggréger plusieurs petites transactions en un seul paiement, par exemple en combinant tous les bitcoins reçus dans la journée. Cela revient à transférer les fonds vers une seule adresse, qui serait celle du "compte courant" de son entreprise. Un schéma représentant une aggrégation de transaction se trouve ici: Transaction agrégeant des fonds.

En dépensant les transactions d'Alice et d'autres clients, il allonge la chaîne de transactions qui sont ajoutée à la blockchain, visible et vérifiable par tous. Supposons que Bob paie son web designer Gopesh, à Bangalore, pour créer un nouveau site web. La chaîne de transactions ressemblera alors à La transaction d'Alice incluse dans la chaine de transactions de Joe vers Gopesh.

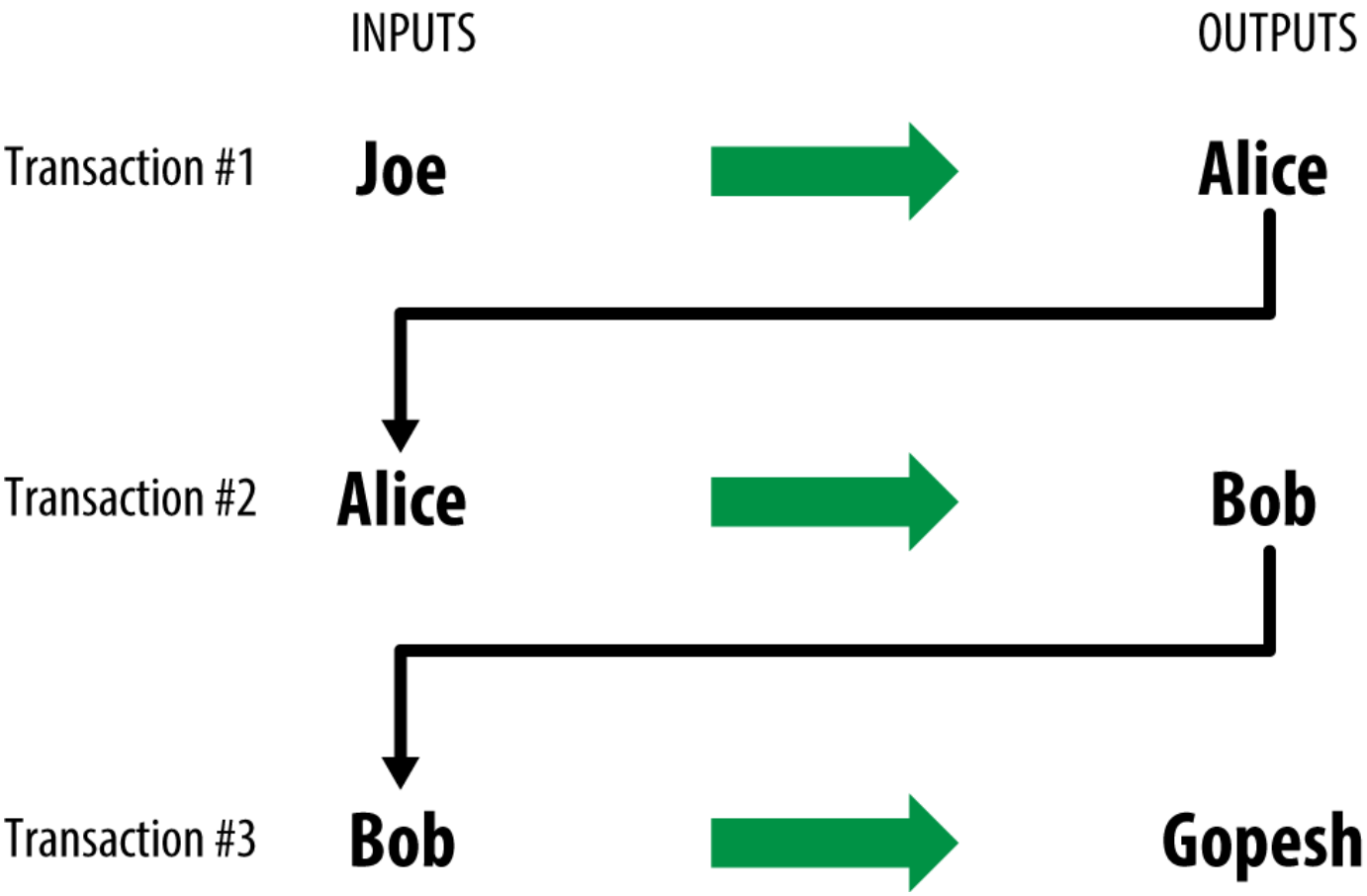


Figure 9. La transaction d'Alice incluse dans la chaine de transactions de Joe vers Gopesh

Le client bitcoin

Bitcoin Core: l'implémentation de référence

Vous pouvez télécharger le client de référence *Bitcoin Core*, aussi connu sous le nom de "Satoshi client," sur bitcoin.org. Le client de référence implémente tous les aspects du système bitcoin, ce qui comprend le porte-monnaie, un moteur de vérification des transactions avec une copie complète de tout le registre de transactions (blockchain), ainsi qu'une noeud réseau complet du réseau peer-to-peer bitcoin.

Sur la page Choisir votre portefeuille Bitcoin, choisissez Bitcoin core afin de télécharger le client de référence. Vous téléchargerez un installateur exécutable en fonction de votre système d'exploitation. Pour Window il s'agit soit d'une archive au format ZIP soit d'un .exe exécutable. Pour Mac OS il s'agit d'une image disque .dmg. Pour linux il s'agit soit d'une package PPA pour Ubuntu soit d'une archive tar.gz. La page bitcoin.org listant les clients bitcoin recommandés est présentée dans Choisir un client sur bitcoin.org.

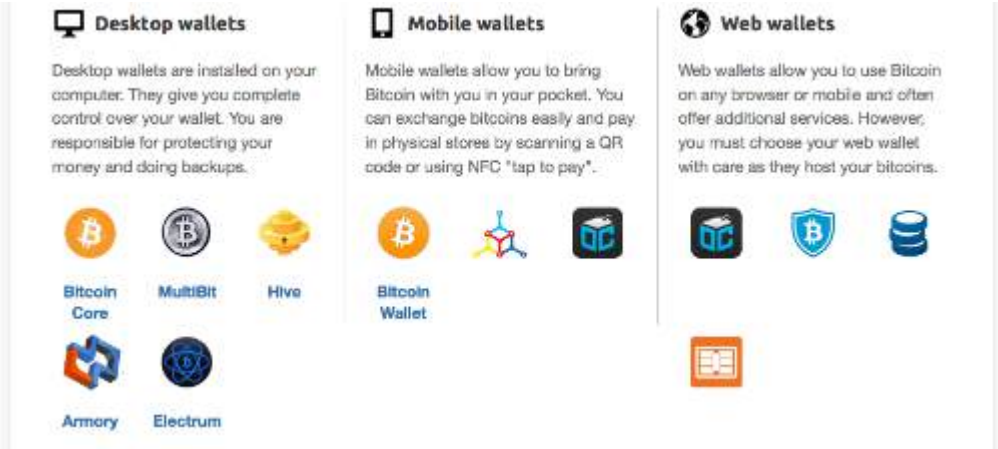


Figure 1. Choisir un client sur bitcoin.org

Lancement de Bitcoin Core pour la première fois

Si vous téléchargez un installateur tel qu'un .exe, .dmg ou PPA, vous pouvez l'installer de la même façon que n'importe quelle application sur votre système d'exploitation. Pour Windows, lancez le .exe et suivez les instructions pas-à-pas. Pour Mac OS, lancez le .dmg et placez l'icône Bitcoin-Qt dans votre répertoire *Applications*. Pour Ubuntu, double-cliquez sur le PPA dans votre explorateur de fichiers et il ouvrira le gestionnaire d'applications afin d'installer le package. Une fois l'installation terminée vous devriez avoir une nouvelle application nommée Bitcoin-Qt dans votre liste d'applications. Double-cliquez sur l'icône afin de démarrer le client bitcoin.

La première fois que vous lancez Bitcoin Core il commencera par télécharger la blockchain, un processus qui peut prendre plusieurs jours (voir Ecran de Bitcoin Core durant la synchronisation de la blockchain). Laissez le tourner en tâche de fond jusqu'à ce qu'il affiche "Synchronisé" au lieu de "Désynchronisé" à coté du solde.

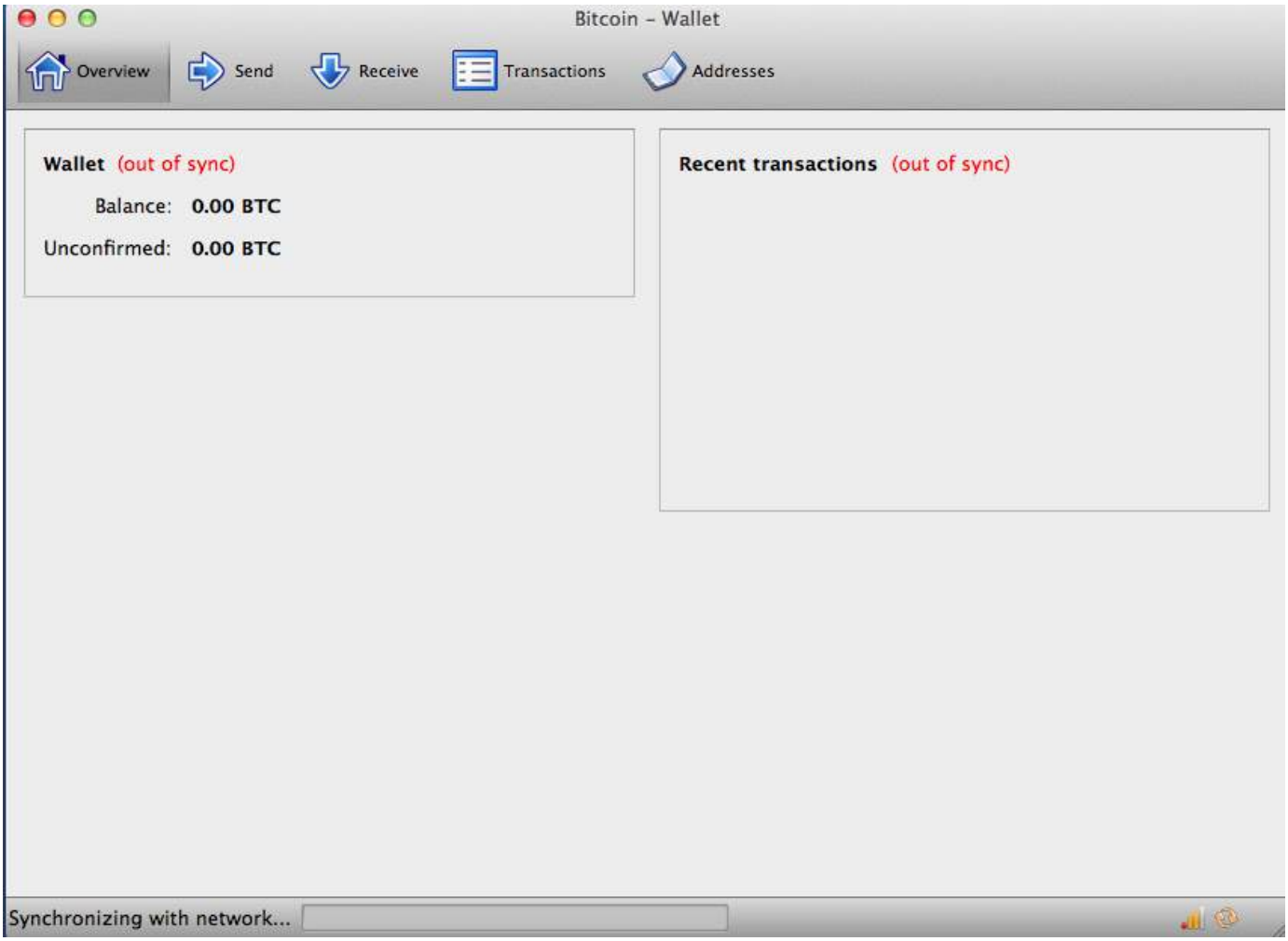


Figure 2. Ecran de Bitcoin Core durant la synchronisation de la blockchain

TIP

Bitcoin Core conserve une copie complète du registre de transactions (blockchain), contenant toutes les transactions ayant eu lieu sur le réseau bitcoin depuis son lancement en 2009. Cette masse de donnée s'élève à quelques giga-octets (approximativement 16 Go fin 2013) et est téléchargée progressivement sur plusieurs jours. Le client ne sera pas capable d'opérer des transactions ou de mettre à jour son solde tant que que la blockchain complète n'est pas téléchargée. Durant tout ce temps, le client affichera l'état "désynchronisé" en face du solde et affichera "En cours de synchronisation" en bas de la fenêtre. Assurez-vous que vous disposez d'assez d'espace disque et de bande passante pour compléter la synchronisation initiale.

Compiler Bitcoin Core depuis le Code Source

Pour les développeurs, il existe également la possibilité de télécharger le code source complet en archive ZIP ou celle de cloner le dépôt GitHub de référence. Sur la page Bitcoin de GitHub, sélectionnez "Download ZIP" sur la droite ou utilisez la commande git pour créer une copie locale du code source sur votre système. Dans l'exemple suivant, nous clonons le code source à partir d'une commande Unix, sous Linux ou Mac OS:

```
$ git clone https://github.com/bitcoin/bitcoin.git
Cloning into 'bitcoin'...
remote: Counting objects: 31864, done.
remote: Compressing objects: 100% (12007/12007), done.
remote: Total 31864 (delta 24480), reused 26530 (delta 19621)
Receiving objects: 100% (31864/31864), 18.47 MiB | 119 KiB/s, done.
Resolving deltas: 100% (24480/24480), done.
$
```

TIP Les instructions et résultats peuvent varier d'une version à l'autre. Suivez la documentation jointe au code même si elle diffère des instructions ci-présentes, et attendez-vous à ce que les résultats sur votre écran soient légèrement différents des exemples contenus ici.

Quand la copie est terminée, vous disposerez d'une copie complète du code source dans le répertoire *bitcoin*. Allez dans ce répertoire en tapant cd bitcoin en ligne de commande:

```
$ cd bitcoin
```

Par défaut, la copie locale sera synchronisée avec le code le plus récent, qui peut potentiellement correspondre à une version non-stable ou bêta de bitcoin. Avant de compiler le code, sélectionnez une version spécifique en effectuant un checkout d'un release *tag*. Les tags sont utilisés par les développeurs pour marquer des version spécifiques du code source. Afin de découvrir les tags disponible, nous utilisons la commande git tag:

```
$ git tag
v0.1.5
v0.1.6test1
v0.2.0
v0.2.10
v0.2.11
v0.2.12

[... beaucoup d'autres tags ...]

v0.8.4rc2
v0.8.5
v0.8.6
v0.8.6rc1
v0.9.0rc1
```

La liste de tags montre toutes les numéros de version de bitcoin. Par convention, les *release candidates*, qui sont destinées au test, ont le suffixe "rc". Les releases stables pouvant être utilisée en production ne

possèdent pas de suffixe. Dans la liste précédente, sélectionnez la version la plus récente, qui à l'heure ou j'écris est la v0.9.0rc1. Pour synchroniser le code local avec cette version, utilisez la commande git checkout:

```
$ git checkout v0.9.0rc1
Note: checking out 'v0.9.0rc1'.

HEAD is now at 15ec451... Merge pull request #3605
$
```

Le code source inclut la documentation qui se trouve dans plusieurs fichiers. Lisez la documentation principale qui se trouve dans le fichier *README.md* du répertoire bitcoin en tapant more README.md en ligne de commandes et en appyant sur la barre d'espace pour passer à la page suivante. Dans ce chapitre nous compilerons le client bitcoin en ligne de commandes, également connu sous le nom de bitcoind sur Linux. Lisez les instructions pour la compilation de bitcoind sur votre plateforme en tapant more doc/build-unix.md. D'autres instructions pour Mac OS et Windows peuvent être trouvées dans le répertoire *doc*, respectivement *build-osx.md* ou *build-msw.md*.

Passez en revue attentivement les pré-requis de la compilation qui constituent la première partie de la documentation. Ils correspondent à des librairies qui doivent être présentes sur votre système avant que vous puissiez débuter la compilation de bitcoin. Si ces pré-requis sont manquants, le processus échouera avec une erreur. Si cela arrive parce que vous avez omis un pré-requis, vous pouvez l'installez puis reprendre le processus là où vous l'aviez laissé. Une fois les pré-requis installés, vous démarrez la compilation en générant un ensemble de scripts de compilation en utilisant le script -autogen.sh_.

TIP

Le processus de compilation de Bitcoin Core a été changé depuis la version 0.9 pour utiliser les système autogen/configure/make. Les versions antérieures utilisent un simple makefile et fonctionnent d'une manière légèrement différente de l'exemple montré ici. Suivez les instructions relatives à la version que vous souhaitez compiler. Le système autogen/configure/make introduit dans la 0.9 est probablement le système qui sera utilisé dans toutes les versions futures et est également le système utilisés dans les exemples qui suivent.

```
$ ./autogen.sh
configure.ac:12: installing `src/build-aux/config.guess'
configure.ac:12: installing `src/build-aux/config.sub'
configure.ac:37: installing `src/build-aux/install-sh'
configure.ac:37: installing `src/build-aux/missing'
src/Makefile.am: installing `src/build-aux/depcomp'
$
```

Le script *autogen.sh* crée un ensemble de scripts de configuration qui iront interroger votre sytème

pour découvrir les paramètres à appliquer et s'assurer que vous disposez de toutes les librairies nécessaires pour compiler le code. Le plus important d'entre eux est le configure script qui offre de multiple options pour configurer le processus de compilation. Tapez ./configure --help pour voir les différentes options:

```
$ ./configure --help

`configure' configures Bitcoin Core 0.9.0 to adapt to many kinds of systems.

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

To assign environment variables (e.g., CC, CFLAGS...), specify them as
VAR=VALUE.  See below for descriptions of some of the useful variables.

Defaults for the options are specified in brackets.

Configuration:
  -h, --help              display this help and exit
      --help=short        display options specific to this package
      --help=recursive    display the short help of all the included packages
  -V, --version affiche les informations sur la version et quitte

[... many more options and variables are displayed below ...]

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]

[... plus d'options ...]

Use these variables to override the choices made by `configure' or to help
it to find libraries and programs with nonstandard names/locations.

Report bugs to <info@bitcoin.org>.

$
```

Le script configure vous permet d'activer ou de désactiver certaines fonctionnalités de bitcoind en utilisant les flags --enable-FONCTIONNALITE et --disable-FONCTIONNALITE ou FONCTIONNALITE est remplacé par le nom de la fonctionnalité comme listé dans l'aide. Dans ce chapitre nous compilerons un client bitcoind avec toutes les fonctionnalités par défaut. Nous n'utiliserons pas les flags de configuration, mais vous pouvez les étudier afin de comprendre quelles fonctionnalités sont optionnelles sur le client. Lancez le script +configure + pour découvrir automatiquement toutes les librairies nécessaires et créer un script de build adapté 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... no
checking for mawk... mawk
checking whether make sets $(MAKE)... yes

[... many more system features are tested ...]

configure: creating ./config.status
config.status: creating Makefile
config.status: creating src/Makefile
config.status: creating src/test/Makefile
config.status: creating src/qt/Makefile
config.status: creating src/qt/test/Makefile
config.status: creating share/setup.nsi
config.status: creating share/qt/Info.plist
config.status: creating qa/pull-tester/run-bitcoind-for-test.sh
config.status: creating qa/pull-tester/build-tests.sh
config.status: creating src/bitcoin-config.h
config.status: executing depfiles commands
$
```

Si tout se passe bien, la commande configure+ créera des scripts de build adaptés qui nous permettrons de compiler bitcoind. Si il subsiste des librairies manquantes ou des erreurs, la commande configure échouera avec une erreur au lieu de créer les scripts de build. Si une erreur apparait, il s'agit probablement d'une librairie manquante ou non compatible. Relisez encore la documentation afin de vous assurer que vous installez les bon pré-requis. Ensuite, relancez la commande configure et voyez si cela corrige l'erreur. La prochaine étape consiste en la compilation du code, étape qui peut prendre jusqu'a une heure pour s'exécuter. Pendant la compilation vous devriez voir des information apparaitre toutes les secondes ou toutes les minutes, ou une erreur si quelque chose se passe mal. Le processus de compilation peut être repris a n'importe quel moment si il est interrompu. Tapez make pour commencer à compiler:

```
$ make
Making all in src
make[1]: Entering directory `/home/ubuntu/bitcoin/src'
make  all-recursive
make[2]: Entering directory `/home/ubuntu/bitcoin/src'
Making all in .
make[3]: Entering directory `/home/ubuntu/bitcoin/src'
  CXX    addrman.o
  CXX    alert.o
  CXX    rpcserver.o
  CXX    bloom.o
  CXX    chainparams.o

[... beaucoup d'autres messages de compilation ...]

  CXX    test_bitcoin-wallet_tests.o
  CXX    test_bitcoin-rpc_wallet_tests.o
  CXXLD  test_bitcoin
make[4]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[3]: Leaving directory `/home/ubuntu/bitcoin/src/test'
make[2]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Leaving directory `/home/ubuntu/bitcoin/src'
make[1]: Entering directory `/home/ubuntu/bitcoin'
make[1]: Nothing to be done for `all-am'.
make[1]: Leaving directory `/home/ubuntu/bitcoin'
$
```

Si tout se passe bien, bitcoind est désormais compilé. La dernière étape est l'installation de l'exécutable bitcoind en utilisant la commande make :

```
$ sudo make install
Making install in src
Making install in .
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c bitcoind bitcoin-cli '/usr/local/bin'
Making install in test
make  install-am
 /bin/mkdir -p '/usr/local/bin'
  /usr/bin/install -c test_bitcoin '/usr/local/bin'
$
```

Vous pouvez vous assurer que bitcoin est correctement installé en demandant au système le chemin des deux exécutable comme suit:

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

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

L'installation par défaut de bitcoind se fait dans le répertoire */usr/local/bin*. Quand vous lancez bitcoind pour la première fois, il vous demandera de créer un fichier de configuration contenant un mot de passe sécurisé pour l'interface JSON-RPC. Lancez bitcoind en tapant bitcoind depuis le terminal:

```
$ bitcoind
Error: To use the "-server" option, you must set a rpcpassword in the configuration file:
/home/ubuntu/.bitcoin/bitcoin.conf
Il est recommandé d'utiliser le mot de passe aléatoire suivant:
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
(vous n'avez pas besoin de retenir ce mot de passe)
Le nom d'utilisateur et le mot de passe NE DOIVENT PAS être les mêmes.
Si ce fichier n'existe pas, créez-en un avec des permissions de lecture seulement.
Il est aussi recommandé de configurer une alerte pour être notifié des éventuels probl
èmes à l'aide de l'option +alertnotify+;
par exemple: alertnotify=echo %s | mail -s "Bitcoin Alert" admin@foo.com
```

Editez le fichier de configuration à l'aide de votre éditeur préféré et mettez à jour les paramètres en remplaçant le mot de passe par un mot de passe complexe comme recommandé par bitcoind. *N'utilisez pas* le mot de passe utilisé ici. Créez un fichier dans le répertoire *.bitcoin* nommé *.bitcoin/bitcoin.conf* et rentrez un nom d'utilisateur et un mot de passe:

```
rpcuser=bitcoinrpc
rpcpassword=2XA4DuKNCbtZXsBQRRNDEwEY2nM6M4H9Tx5dFjoAVVbK
```

Durant l'édition du fichier de configuration, vous souhaiterez peut-être ajouter quelques options, telles que txindex (voir L'index de base de données de transactions et l'option txindex). Pour un listing complet des options disponibles, tapez bitcoind -- help.

Lancez maintenant le client Bitcoin Core. La première fois que vous le lancez, il reconstruira la blockchain bitcoin en téléchargeant les blocs. Il s'agit d'un fichier de plusieurs giga-octets et il prendra 2 jours en moyenne pour être totalement téléchargé. Vous pouvez raccourcir le temps d'initialisation en téléchargeant une copie partielle de la blockchain en utilisant un client BitTorrent depuis SourceForge.

Lancez bitcoind en tâche de fond avec l'option -daemon:

```
$ bitcoind -daemon

Bitcoin version v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
Using OpenSSL version OpenSSL 1.0.1c 10 May 2012
Default data directory /home/bitcoin/.bitcoin
Using data directory /bitcoin/
Using at most 4 connections (1024 file descriptors available)
init message: Verifying wallet...
dbenv.open LogDir=/bitcoin/database ErrorFile=/bitcoin/db.log
Bound to [::]:8333
Bound to 0.0.0.0:8333
init message: Loading block index...
Opening LevelDB in /bitcoin/blocks/index
Opened LevelDB successfully
Opening LevelDB in /bitcoin/chainstate
Opened LevelDB successfully

[... d'autres messages de démarrage ...]
```

Utilisation de l'API JSON-RPC de BitcoinCore depuis la ligne de commande

Le client Bitcoin Core implémente une interface JSON-RPC qui peut également être interrogée avec l'outil en ligne de commande bitcoin-cli. La ligne de commande nous permet d'expérimenter de façon interactive les fonctionnalités qui sont également disponibles au travers de l'API. Pour commencer, utilisez la commande help pour consulter la liste des commande 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"
decodescript "hex"
dumpprivkey "bitcoinaddress"
dumpwallet "filename"
getaccount "bitcoinaddress"
getaccountaddress "account"
getaddednodeinfo dns ( "node" )
getaddressesbyaccount "account"
getbalance ( "account" minconf )
getbestblockhash
getblock "hash" ( verbose )
```

```
getblockchaininfo
getblockcount
getblockhash index
getblocktemplate ( "jsonrequestobject" )
getconnectioncount
getdifficulty
getgenerate
gethashespersec
getinfo
getmininginfo
getnettotals
getnetworkhashps ( blocks height )
getnetworkinfo
getnewaddress ( "account" )
getpeerinfo
getrawchangeaddress
getrawmempool ( verbose )
getrawtransaction "txid" ( verbose )
getreceivedbyaccount "account" ( minconf )
getreceivedbyaddress "bitcoinaddress" ( minconf )
gettransaction "txid"
gettxout "txid" n ( includemempool )
gettxoutsetinfo
getunconfirmedbalance
getwalletinfo
getwork ( "data" )
help ( "command" )
importprivkey "bitcoinprivkey" ( "label" rescan )
importwallet "filename"
keypoolrefill ( newsize )
listaccounts ( minconf )
listaddressgroupings
listlockunspent
listreceivedbyaccount ( minconf includeempty )
listreceivedbyaddress ( minconf includeempty )
listsinceblock ( "blockhash" target-confirmations )
listtransactions ( "account" count from )
listunspent ( minconf maxconf  ["address",...] )
lockunspent unlock [{"txid":"txid","vout":n},...]
move "fromaccount" "toaccount" amount ( minconf "comment" )
ping
sendfrom "fromaccount" "tobitcoinaddress" amount ( minconf "comment" "comment-to" )
sendmany "fromaccount" {"address":amount,...} ( minconf "comment" )
sendrawtransaction "hexstring" ( allowhighfees )
sendtoaddress "bitcoinaddress" amount ( "comment" "comment-to" )
setaccount "bitcoinaddress" "account"
setgenerate generate ( genproclimit )
settxfee amount
```

```
signmessage "bitcoinaddress" "message"
signrawtransaction "hexstring" (
[{"txid":"id","vout":n,"scriptPubKey":"hex","redeemScript":"hex"},...]
["privatekey1",...] sighashtype )
stop
submitblock "hexdata" ( "jsonparametersobject" )
validateaddress "bitcoinaddress"
verifychain ( checklevel numblocks )
verifymessage "bitcoinaddress" "signature" "message"
walletlock
walletpassphrase "passphrase" timeout
walletpassphrasechange "oldpassphrase" "newpassphrase"
```

Obtenir des information sur le statut du client Bitcoin Core

Cmmande: getinfo

La commande RPC bitcoin getingo affiche des informations à propos du statut du noeud bitcoin, le portefeuille, et la base de données de la blockchain. Utilisez bitcoin-cli pour la lancer:

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,
    "protocolversion" : 70002,
    "walletversion" : 60000,
    "balance" : 0.00000000,
    "blocks" : 286216,
    "timeoffset" : -72,
    "connections" : 4,
    "proxy" : "",
    "difficulty" : 2621404453.06461525,
    "testnet" : false,
    "keypoololdest" : 1374553827,
    "keypoolsize" : 101,
    "paytxfee" : 0.00000000,
    "errors" : ""
}
```

Les données sont retournées au format JavaScript Object Notation (JSON), un format facilement "consommable" par tous les langages de programmations mais également très lisible. Parmi ces données nous voyons les numéros de version du client bitcoin(90000), du protocole (70002), et du portefeuille(60000). Nous voyons le solde actuel contenu dans le portefeuille, qui est égal à zéro. Nous voyons l'actuelle hauteur de bloc, qui nous montre combien de blocs sont connus par ce client

(286216). Nous voyons également des statistiques variées concernant le réseau bitcoin et les paramètres du client. Nous explorerons ces paramètres en détail dans le reste de ce chapitre.

TIP	Cela prendra du temps, peut-être plus d'une journée, pour que le client bitcoind "rattrape" la hauteur actuelle de la blockchain le temps qu'il télécharge les blocs provenant d'autres clients bitcoin. Vous pouvez consulter l'état d'avancement en utilisant getinfo pour connaitre exactement le nombre de blocs du client.

Configuration et cryptage du portefeuile

Commandes: encryptwallet, walletpassphrase

Avant de procéder à la création de clés et à d'autres commandes, vous utiliserez la commande encryptwallet avec le mot de passe "foo". N'oubliez pas bien sûr de remplacer "foo" par un mot de passe complexe !

```
$ bitcoin-cli encryptwallet foo
wallet encrypted; Bitcoin server stopping, restart to run with encrypted wallet. The
keypool has been flushed, you need to make a new backup.
$
```

Vous pouvez vérifier que le portefeuille a bien été crypté en utilisant encore la commande getinfo. Cette fois-ci vous remarquerez une nouvelle entrée nommée unlocked_until. Il s'agit d'un compteur montrant combien de temps le mot de passe permettant le décryptage est enregistré en mémoire, et combien de temps le portefeuille est déverrouillé. La première fois il sera à zero, cela veut dire que le portefeuille est verrouillé:

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,

#[... autres informations...]

    "unlocked_until" : 0,
    "errors" : ""
}
$
```

Pour déverrouiller le portefeuille, utilisez la commande walletpassphrase qui prend deux paramètres—le mot de passe et un nombre de secondes au bout desquelles le portefeuille sera verrouillé à nouveau (une minuterie):

```
$ bitcoin-cli walletpassphrase foo 360
$
```

Vous pouvez vous assurer que le portefeuille est déverrouillé et voir l'état de la minuterie en tapant getinfo à nouveau:

```
$ bitcoin-cli getinfo
```

```
{
    "version" : 90000,

#[... autres informations ...]

    "unlocked_until" : 1392580909,
    "errors" : ""
}
```

Sauvegarde du portefeuille, export au format texte et restauration

Commandes: backupwallet, importwallet, dumpwallet

La prochaine étape consiste à nous entrainer à créer un fichier de sauvegarde du portefeuille pour ensuite le restaurer à partir de ce même fichier. Utilisez la commande backupwallet pour la sauvegarde en spécifiant le nom du fichier de destination en paramètre. Ici nous sauvegardons le portefeuille dans un fichier nommé *wallet.backup*:

```
$ bitcoin-cli backupwallet wallet.backup
$
```

Maintenant, pour restaurer le fichier de sauvegarde, utilisez la commande importwallet. Si votre portefeuille est verrouillé, vous devrez d'abord le déverrouiller (voir walletpassphrase dans la section précédente) pour importer le fichier de sauvegarde:

```
$ bitcoin-cli importwallet wallet.backup
$
```

La commande dumpwallet peut être utilisée pour exporter le portefeuille dans un fichier texte lisible par un humain:

```
$ bitcoin-cli dumpwallet wallet.txt
$ more wallet.txt
# Wallet dump created by Bitcoin v0.9.0rc1-beta (2014-01-31 09:30:15 +0100)
# * Created on 2014-02- 8dT20:34:55Z
# * Best block at time of backup was 286234
(000000000000000f74f0bc9d3c186267bc45c7b91c49a0386538ac24c0d3a44),
#   mined on 2014-02- 8dT20:24:01Z

KzTg2wn6Z8s7ai5NA9MVX4vstHRsqP26QKJCzLg4JvFrp6mMaGB9 2013-07- 4dT04:30:27Z change=1 #
addr=16pJ6XkwSQv5ma5FSXMRPaXEYrENCEg47F
Kz3dVz7R6mUpXzdZy4gJEVZxXJwA15f198eVui4CUivXotzLBDKY 2013-07- 4dT04:30:27Z change=1 #
addr=17oJds8kaN8LP8kuAkWTco6ZM7BGXFC3gk
[... beaucoup d'autres clés ...]

$
```

Adresses du portefeuille et reception des transactions

Commandes: getnewaddress, getreceivedbyaddress, listtransactions, getaddressesbyaccount, getbalance

Le client bitcoin de référence maintient un pool d'adresses dont la taille est spécifiée par keypoolsize quand vous utilisez la commande getinfo. Ces adresses sont générées automatiquement et peuvent être utilisées comme des adresses publique de réception ou des adresse utilisées pour la monnaie d'une transaction. Pour récupérer une de ces adresses, utilisez la commande getnewaddress:

```
$ bitcoin-cli getnewaddress
1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
```

Maintenant nous pouvons utiliser cette adresse pour envoyer de petits montants en bitcoin vers notre portefeuille depuis un portefeuille externe (en supposant que vous disposiez d'un peu de bitcoin sur un échange, un portefeuille web ou un autre client bitcoind). Pour cet exemple nous envoyons 50 millibits (0.050 bitcoin) vers l'adresse que nous avons récupérée.

Nous pouvons maintenant interroger le client bitcoind sur le montant reçu par cette adresse, et spécifier combien de confirmations sont requises avant que le montant puisse être effectivement pris en compte dans le solde. Pour cet exemple, nous spécifions zéro confirmations. Quelque secondes après avoir envoyé les bitcoins depuis l'autre portefeuille, nous pourront les voir dans notre portefeuille. Nous utilisons getreceivedbyaddress avec l'adresse et le nombre de confirmations à zéro (0):

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL 0
0.05000000
```

Si nous omettons le zero à la fin de cette commande, nous ne pourrons voir que les montant ayant au moins minconf confirmations, minconf étant le paramètre correspondant au minimum de confirmations avant que la transaction soit listée dans notre solde. Le paramètre minconf est spécifié dans le fichier de configuration de bitcoind. Parce que la transaction envoyant ces bitcoins vient tout juste d'être effectuée, elle n'a pas encore été confirmée et donc nous verrons un solde à zéro:

```
$ bitcoin-cli getreceivedbyaddress 1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL
0.00000000
```

Les transactions reçues par le portefeuille entier peuvent également être affichées en utilisant la commande listtransactions:

```
$ bitcoin-cli listtransactions
```

```
[
    {
        "account" : "",
        "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
        "category" : "receive",
        "amount" : 0.05000000,
        "confirmations" : 0,
        "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
        "time" : 1392660908,
        "timereceived" : 1392660908
    }
]
```

Nous pouvons lister toutes les adresses du portefeuille en utilisant la commande getaddressesbyaccount:

```
$ bitcoin-cli getaddressesbyaccount ""
```

```
[
    "1LQoTPYy1TyERbNV4zZbhEmgyfAipC6eqL",
    "17vrg8uwMQUibkvS2ECRX4zpcVJ78iFaZS",
    "1FvRHWhHBBZA8cGRRsGiAeqEzUmjJkJQWR",
    "1NVJK3JsL41BF1KyxrUyJW5XHjunjfp2jz",
    "14MZqqzCxjc99M5ipsQSRfieT7qPZcM7Df",
    "1BhrGvtKFjTAhGdPGbrEwP3xvFjkJBuFCa",
    "15nem8CX91XtQE8B1Hdv97jE8X44H3DQMT",
    "1Q3q6taTsUiv3mMemEuQQJ9sGLEGaSjo81",
    "1HoSiTg8sb16oE6SrmazQEwcGEv8obv9ns",
    "13fE8BGhBvnoy68yZKuWJ2hheYKovSDjqM",
    "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
    "1KHUmVfCJteJ21LmRXHSpPoe23rXKifAb2",
    "1LqJZz1D9yHxG4cLkdujnqG5jNNGmPeAMD"
]
```

Au final, la commande getbalance montrera le solde total du portefeuille en ajoutant toutes les transactions confirmées avec au moins minconf confirmations:

```
$ bitcoin-cli getbalance
0.05000000
```

TIP Si la transaction n'est pas encore confirmée, le solde retourné par getbalance sera égal a zéro. La configuration de l'option "minconf" détermine le nombre minimum de confirmations requises avant qu'une transaction soit prise en compte dans le solde.

Décodage et exploration des transactions

Commandes: gettransaction, getrawtransaction, decoderawtransaction

Nous explorerons maintenant la transaction entrante litée précedemment en utilisant la commande gettransaction. Nous pouvons récupérer une transaction à l'aide de son hash, l'attribut txid montré précédemment, à l'aide de la commande gettransaction:

```
{
    "amount" : 0.05000000,
    "confirmations" : 0,
    "txid" : "9ca8f969bd3ef5ec2a8685660fdbf7a8bd365524c2e1fc66c309acbae2c14ae3",
    "time" : 1392660908,
    "timereceived" : 1392660908,
    "details" : [
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.05000000
        }
    ]
}
```

TIP

Les identifiants de transactions (txid) ne font pas autorité tant qu'une transaction n'a pas été confirmée. L'absence de hash de transaction dans la blockchain ne veut pas dire que la transaction n'a pas été traitée. Cet aspect est connu sous le nom de"transaction malleability") "malléabilité de transaction", car les hash de transaction peuvent être modifié avant leur confirmation dans un bloc. Apres la confirmation, les txids sont immuables et font autorité.

Le format de transaction obtenu à l'aide de la commande gettransaction est un format simplifié. Pour récupérer le code complet d'une transaction et le décoder, nous utiliserons deux commandes: getrawtransaction et decoderawtransaction. La première, getrawtransaction, prend le *hash de transaction (txid)* comme paramètre et retourne la transaction complète en chaine de caractère hexadécimaux "bruts" (raw), exactement comme elle apparait sur le réseau bitcoin:

Pour décoder cette chaîne de caractères, utilisez la commande decoderawtransaction. Copiez et collez les caractère hexadécimaux en tant que premier paramètre de decoderawtransaction afin d'obtenir le contenu complet interprété au format de données JSON (pour des raisons de mise en forme la chaine hexadécimale est raccourcie dans l'exemple suivant):

Le décodage de la transaction montre tous les composants de la transaction, y compris les entrée et les sorties (les inputs et les outputs). Dans cet exemple nous voyons que la transaction qui a crédité notre nouvelle adresse avec 50 millibits a utilisé une entrée et a généré deux sorties. L'entrée de cette transaction était la sortie d'une transaction précédente confirmée (comme le montre le vin txid commençant par d3c7). Les deux sorties correspondent au crédit de 50 millibits et une sortie contenant la monnaie retournée à l'expéditeur.

Nous pouvons aller plus loin dans l'exploration de la blockchain en examinant la transaction précédentes référencée par son txid dans cette transaction en utilisant les même commandes (à savoir, gettransaction). En naviguant de transaction en transaction, nous pouvons suivre la chaîne des transaction et voir les coins transmis entre les propriétaire des adresses.

Une fois que la transaction que nous avons reçue est confirmée par son inclusion dans un bloc, la commande gettransaction nous retournera des information supplémentaires et montrera le *block hash (identifiant)* dans lequel la transaction a été incluse.

Nous voyons ici les nouvelles informations dans les entrées blockhash (le hash du bloc dans lequel la transaction est incluse) and dans blockindex qui possède la valeur 18 (indiquant que notre transaction était la 18ème transaction de ce bloc).

L'index de base de données de transactions et l'option txindex

Par défaut, Bitcoin Core construit une base de données contenant *uniquement* les transactions relatives au portefeuille de l'utilisateur. Si vous souhaitez accéder à *n'importe quelle* transaction avec des commandes telles que gettransaction, vous devez configurer Bitcoin Core pour qu'il construise un index complet des transactions à l'aide de l'option txindex. Positionnez l'option txindex=1 dans le fichier de configuration de Bitcoin Core (que vous trouverez généralement dans le répertoire *.bitcoin/bitcoin.conf*). Une fois que vous avez changé cette option, vous devrez redémarrer bitcoind et attendre qu'il reconstruise l'index.

Exploration des blocs

Commandes: getblock, getblockhash

Maintenant que nous connaissons le bloc dans lequel a été inclu notre transaction, nous pouvons effectuer des requêtes sur ce bloc. Nous utilisons la commande getblock avec le hash du bloc en paramètre:

Le bloc contient 367 transactions et comme vous pouvez le voir, la 18ème transaction (9ca8f9...) correspond au txid de celle créditant 50 millibits sur notre adresse. La donnée height nous indique qu'il s'agit du bloc numéro 286384 dans la blockchain.

Nous pouvons également récupérer un bloc avec la hauteur de bloc en utilisant la commande getblockhash qui prend la hauteur de bloc en paramètre et retourne le hash du bloc:

Ici nous récupérons le hash su "genesis block", le premier bloc miné par Satoshi Nakamoto à la hauteur zero. La récupération de ce bloc donne:

Les commandes getblock, getblockhash, et gettransaction peuvent être utilisées pour explorer la base de donnée de la blockchain:

Créer, Signer, et Soumettre des Transactions basées sur des <phrase role="keep-together">sorties non dépensées (Unspent Outputs)</phrase>

Commandes: listunspent, gettxout, createrawtransaction, decoderawtransaction, signrawtransaction, sendrawtransaction

Les transaction Bitcoin sont basée sur le concept de dépense de sorties ("outputs"), qui résultent de transactions précédentes, afin de créer une chaine de transactions qui transfère la propriété d'une adresse vers une autre adresse. Notre portefeuille a maintenant reçu une transaction qui a assigné une sortie à notre adresse. Une fois que cette transaction est confirmée, nous pouvons dépenser cette sortie.

Premièrement, nous utilisons la commande listunspent pour afficher toutes les sorties non dépensées et *confirmées* de notre portefeuille.

```
$ bitcoin-cli listunspent
```

Nous voyons que la transaction 9ca8f9... a créé une sortie (avec l'index vout 0) assignée à l'adresse 1hvzSo... pour le montant de 50 millibits. A cet instant la transaction a reçu sept confirmations. Les transactions utilisent des sorties précédemment créées comme entrées en y faisant référence via leur précédent txid (identifiant de transaction) et l'index vout. Nous allons donc maintenant créer une transaction qui dépensera le vout numéro 0 de la transaction 9ca8f9... en tant q'entrée afin de lui assigner une nouvelle sortie qui enverra le montant vers une nouvelle adresse.

Analysons maintenant en détail cette sortie. Nous utilisons la commande gettxout pour voir les détails de cette sortie non dépensée (unspent output). Les sorties de transaction sont toujours référencées par le txid et le vout, qui sont les paramètres que nous passerons à la commande gettxout :

Ce que nous voyons ici est la sortie qui a assigné 50 millibits à notre adresse 1hvz.... Pour dépenser cette sortie nous créerons une nouvelle transaction. Commençons par créer une adresse vers laquelle nous enverrons l'argent :

```
$ bitcoin-cli getnewaddress
1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb
```

Nous enverrons 25 millibits vers la nouvelle adresse 1LnfTn... que nous venons de créer dans notre portefeuille. Dans notre nouvelle transaction, nous dépenserons la sortie de 50 millibits et enverrons 25 millibits vers cette nouvelles adresse. Parce que nous nous devons de dépenser la sortie *entière* de la précédente transaction, nous devons également générer de la monnaie. Nous enverrons la monnaie vers l'adresse 1hvz... d'où les fonds proviennent initialement. Enfin, nous aurons aussi à payer des frais pour cette transaction. Pour payer les frais, nous réduisons la monnaie de 0,5 millibits, et nous retournons 24,5 millibits de monnaie. La différence entre la somme des nouvelles sorties (25 mBTC + 24,5 mBTC = 49,5 mBTC) et l'entrée (50 mBTC) sera récupérée par les mineurs comme frais de transaction.

Nous utilisons la commande createrawtransaction pour créer cette transaction. Les paramètres que nous fournissons à la commande createrawtransaction sont l'entrée de transaction (les 50 millibits non dépensés de notre transaction confirmée) et les deux sorties de transaction (l'argent envoyé à la nouvelle adresse et la monnaie renvoyée vers l'adresse précédente) :

La commande createrawtransaction produit une chaine de caractère hexadécimale brute qui encode les détails de la transaction que nous avons fournis. Vérifions que tout est correct en décodant cette chain brute en utilisant la commande decoderawtransaction :

Cela semble OK ! Notre nouvelle transaction "consomme" les sorties non dépensée de notre transaction confirmée pour ensuite les dépenser dans deux sorties, une de 25 millibits vers notre nouvelle adresse et un de 24,5 millibits comme monnaie renvoyée vers l'adresse initiale. La différence de 0,5 millibits représente les frais de transactions et sera créditée au mineur qui trouvera le bloc dans lequel cette transaction sera incluse.

Comme vous pouvez le remarquer, la transaction contient un champ scriptSig vide car nous ne l'avons pas encore signée. Sans signature, cette transaction n'est pas utilisable; nous n'avons pas encore prouvé que nous *possédons* l'adresse d'où proviennent les sorties non dépensées. En la signant, nous déverrouillons la sortie et prouvons qu'elle nous appartient afin de pouvoir la dépenser. Nous utilisons la commande signrawtransaction pour signer la transaction. Elle prend la transaction en tant que chaine de caractères hexadécimale brute comme paramètre :

TIP Un portefeuille crypté doit être déverrouillé de pouvoir signer une transaction car le fait de signer une transaction nécessite l'accès aux clés secretes du portefeuille.

La commande signrawtransaction retourne une autre transaction au format texte hexadecimal brut. Nous la décodons pour découvrir ce qui a changé, à l'aide de la commande decoderawtransaction :

Maintenant les entrée utilisées par la transaction contiennent un valeur scriptSig, qui correspond à la signature numérique prouvant qu'elles appartiennent à l'adresse 1hvz... et déverrouillant la sortie afin qu'elle soit dépensée.La signature rend la transaction vérifiable par n'importe quel noeud du réseau bitcoin.

Nous pouvons maintenant soumettre soumettre cette transaction nouvellement crée au réseau. Nous réalisons cela à l'aide de la commande sendrawtransaction, qui prend la chaine hexadécimale brute produite par signrawtransaction en paramètre :

La commande sendrawtransaction renvoie un *hash de transaction (txid)* tout en soumettant la transaction au réseau. Nous pouvons désormais effectuer une requête sur cet identifiant de transaction avec la commande gettransaction :

```
{
    "amount" : 0.00000000,
    "fee" : -0.00050000,
    "confirmations" : 0,
    "txid" : "ae74538baa914f3799081ba78429d5d84f36a0127438e9f721dff584ac17b346",
    "time" : 1392666702,
    "timereceived" : 1392666702,
    "details" : [
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "send",
            "amount" : -0.02500000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "send",
            "amount" : -0.02450000,
            "fee" : -0.00050000
        },
        {
            "account" : "",
            "address" : "1LnfTndy3qzXGN19Jwscj1T8LR3MVe3JDb",
            "category" : "receive",
            "amount" : 0.02500000
        },
        {
            "account" : "",
            "address" : "1hvzSofGwT8cjb8JU7nBsCSfEVQX5u9CL",
            "category" : "receive",
            "amount" : 0.02450000
        }
    ]
}
```

Comme nous l'avons fait précédemment, nous pouvons examiner le détail de cette transaction avec les commandes getrawtransaction et decodetransaction. Ce commandes nous retourneront exactement la meme chaine hexadécimale que nous avons produite et décodée auparavant, juste avant de l'envoyer sur le réseau.

Clients alternatifs, Libraries, et Toolkits

en plus du client de référence (bitcoind),d'autres clients et librairies peuvent être utilisés pour

interagir avec le réseau bitcoin et ses structures de données. Ils sont implémentés dans différents languages de programmation et offrent aux développeur des interfaces natives dans leurs languages de prédilection..

Voici une liste d'implémentations alternatives:

libbitcoin
: Kit de développement multiplateforme en C++

bitcoin explorer
: Outil Bitcoin en ligne de commande

bitcoin server
: Client Bitcoin complet et serveur de requêtes

bitcoinj
: Une libraire Java complète

btcd
: Un client bitcoin complet dans le language Go

Bits of Proof (BOP)
: Un implémentation de bitcoin en Java pour l'entreprise

picocoin
: Une implémentation de librairie pour client léger en C

pybitcointools
: Une librairie bitcoin en Python

pycoin
: Une autre librairie bitcoin en Python

Beaucoup d'autres librairies existent dans une multitude de languages de programmation et de nouvelles sont crées au fil du temps.

Libbitcoin et le Bitcoin Explorer

La librairie libbitcoin est un kit de développement multiplateforme en C++ qui supporte la client lourd libbitcoin-server et l'outil en ligne de commande Bitcoin Explorer (bx).

Les commande de bx offrent les mêmes fonctionnalités que celles du client bitcoind que nous avons utilisées dans ce chapitre. Les commandes bx offrent également des outils de gestion de clés qui ne sont pas présents dans bitcoind, tels que les clés déterministes de type-2, l'encodage mnémonique de clé, ainsi que les adresses furtives (sltealth address), le paiement et un moteur de requêtes.

Installation de Bitcoin Explorer

Pour utiliser le Bitcoin Explorer, vous n'avez qu'a télécharger l'exécutable signé correspondant à votre système. des distributions sont disponibles pour le mainnet et le testnet pour Linux, OS X, et Windows.

Tapez bx sans aucun paramètre pour afficher la liste de commandes disponibles (voir [appdx_bx]).

Bitcoin Explorer fournit également un installateur pour compiler à partir du code source sous Linux et OS X, ainsi qu'un projet Visual Studio pour Windows.

TIP	Bitcoin Explorer offre tout un tas de commandes très utiles pour encoder et décoder des adresses et convertir différents formats et représentations. Utilisez les pour explorer les différents formats tels que Base16 (hexadécimal), Base58, Base58Check, Base64, etc.

Installation de Libbitcoin

La librairie libbitcoin fournit un installateur pour compiler à partir du code source sous Linux et OS X, ainsi qu'un projet Visual Studio pour Windows. Il est également possible de compiler à l'aide de l'outil Autotools.

TIP	Le programme d'installation de Bitcoin explorer installe à la fois bx et la librairie libbitcoin, si vous avez compilé bx depuis les sources vous pouvez donc ignorer cette étape.

pycoin

La librairie Python *pycoin*, écrite et maintenue au départ par Richard Kiss, est une librairie Python qui supporte la manipulation de clés et de transactions bitcoin. Son niveau de support du langage de script permet même de traiter les transactions non standard.

La librairie pycoin supporte à la fois Python 2 (2.7.x) et Python 3 (après la version 3.3), elle est fournie avec des utilitaires en ligne de commande très pratiques, ku et tx. Pour installer pycoin 0.42 avec Python 3 in dans un environnement virtuel (venv), utilisez la commande suivante :

```
$ python3 -m venv /tmp/pycoin
$ . /tmp/pycoin/bin/activate
$ pip install pycoin==0.42
Downloading/unpacking pycoin==0.42
  Downloading pycoin-0.42.tar.gz (66kB): 66kB downloaded
  Running setup.py (path:/tmp/pycoin/build/pycoin/setup.py) egg_info for package
pycoin

Installing collected packages: pycoin
  Running setup.py install for pycoin

    Installing tx script to /tmp/pycoin/bin
    Installing cache_tx script to /tmp/pycoin/bin
    Installing bu script to /tmp/pycoin/bin
    Installing fetch_unspent script to /tmp/pycoin/bin
    Installing block script to /tmp/pycoin/bin
    Installing spend script to /tmp/pycoin/bin
    Installing ku script to /tmp/pycoin/bin
    Installing genwallet script to /tmp/pycoin/bin
Successfully installed pycoin
Cleaning up...
$
```

Voici un exemple de script Python qui sélectionne et dépense quelques bitcoin en utilisant la librairie pycoin :

```
#!/usr/bin/env python

from pycoin.key import Key

from pycoin.key.validate import is_address_valid, is_wif_valid
from pycoin.services import spendables_for_address
from pycoin.tx.tx_utils import create_signed_tx

def get_address(which):
    while 1:
        print("enter the %s address=> " % which, end='')
        address = input()
        is_valid = is_address_valid(address)
        if is_valid:
            return address
        print("invalid address, please try again")

src_address = get_address("source")
spendables = spendables_for_address(src_address)
print(spendables)

while 1:
    print("enter the WIF for %s=> " % src_address, end='')
    wif = input()
    is_valid = is_wif_valid(wif)
    if is_valid:
        break
    print("invalid wif, please try again")

key = Key.from_text(wif)
if src_address not in (key.address(use_uncompressed=False),
key.address(use_uncompressed=True)):
    print("** WIF doesn't correspond to %s" % src_address)
print("The secret exponent is %d" % key.secret_exponent())

dst_address = get_address("destination")

tx = create_signed_tx(spendables, payables=[dst_address], wifs=[wif])

print("here is the signed output transaction")
print(tx.as_hex())
```

Pour obtenir des exemples utilisant les utilitaires ku et tx, reportez-vous à [appdxbitcoinimpproposals].

btcd

btcd est un client bitcoin complet (ful-node) écrit en Go. Il télécharge, valide et alimente la blockchain en utilisant exactement les mêmes règles (y compris les bugs) concernant l'acceptation des blocs que l'implémentation de référence, bitcoind. Il relaie également les nouveaux blocs minés, maintient un pool de transactions, et relaie les transaction qui ne sont pas encore incluses dans un bloc. Il s'assure que toutes les transactions admises dans son pool suivent les règles requises et inclut également une grand majorité des vérification strictes en se basant sur les exigences des mineurs (transactions "standard").

Une différence majeure entre btcd et bitcoind est qu'il n'inclut pas la fonctionnalité de portefeuille, et cela de façon intentionnelle. Cela veut dire que vous ne pouvez pas effectuer de paiement ou recevoir des fonds directement via btcd. Cette fonctionnalité est fournie par les projets btcwallet et btcgui, qui sont en cours de développement. On peut citer d'autres différences notables entre btcd et bitcoind comme le support par btcd des requête HTTP POST (comme bitcoind) et les Websockets, et le fait que les connexions RPC de btcd soient en TLS par défaut.

Installation de btcd

Pour installer btcd sous Windows, téléchargez et lancez le msi disponible sur GitHub, ou lancez la commande suivante sous Linux, en supposant bien sûr que vous avez installé le langage Go :

```
$ go get github.com/conformal/btcd/...
```

Pour mettre à jour la dernière version de btcd, utilisez la commande :

```
$ go get -u -v github.com/conformal/btcd/...
```

Configuration de btcd

btcd possède plusieurs options de configuration que vous pouvez lister en utilisant la commande :

```
$ btcd --help
```

btcd est fourni avec quelques outils tels que btcctl, qui est un utilitaire en ligne de commande qui peut être utilisé à la fois pour contrôler et interroger btcd via RPC. Btcd n'active pas le serveur RPC par défaut; vous devez configurer au minimum à la fois un nom d'utilisateur et un mot de passe dans les fichiers de configuration suivants :

- *btcd.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

- *btcctl.conf*:

```
[Application Options]
rpcuser=myuser
rpcpass=SomeDecentp4ssw0rd
```

Ou alors si vous voulez surcharger les fichiers de configuration depuis la ligne de commande :

```
$ btcd -u myuser -P SomeDecentp4ssw0rd
$ btcctl -u myuser -P SomeDecentp4ssw0rd
```

Pour une liste des options disponibles, utilisez la commande suivante :

```
$ btcctl --help
```

Clés, Adresses, Portefeuilles

Introduction

La détention de bitcoins se fait via des *clés numériques*, des adresses bitcoin_, et des *signatures électroniques*. Les clés ne sont pas stockées sur le réseau, mais sont plutôt créées et stockées par les utilisateurs dans un fichier, sous forme d'une base de données basique appelée un *portefeuille*. Les clés d'un portefeuille sont totalement indépendantes du protocole bitcoin, et peuvent être créées et gérées par un logiciel tiers, indépendamment de la blockchain et même sans accès à Internet. Ce sont ces clés qui rendent possibles de nombreuses caractéristiques intéressantes de bitcoin, comme le contrôle et la confiance décentralisée, la preuve de propriété, et le modèle de sécurité protégé par la cryptographie.

Une transaction bitcoin doit contenir une signature valide pour être ajoutée à la blockchain, et cette signature ne peut être construite qu'avec des clefs valides. Cela signifie que n'importe quelle personne possédant une copie de ces clefs peut utiliser les fonds qui y sont associés. Les clefs sont créées par paires: une clef privée (secrète) et une clef publique. On peut comparer la clef publique à un numéro de compte en banque, et la clef privée au PIN de la carte bleue (ou à la signature sur un chèque) qui permet d'utiliser les fonds de ce compte en banque. En général, on n'a pas besoin de manipuler directement ces clefs, qui sont gérées et sauvegardées par le porte-monnaie bitcoin.

Dans les sorties d'une transaction bitcoin, la clef publique du destinataire est représentée par une empreinte numérique appelée une *addresse bitcoin*, et qui est équivalente au nom du bénéficiaire sur un chèque ("payez à ..."). Dans la plupart des cas, une adresse est simplement l'empreinte d'une clef publique, et est donc équivalente à cette clef publique. Mais il est aussi possible de créer des transactions dont les adresses de destination correspondent à autre chose qu'une simple clef publique, un script par exemple (voir plus loin dans ce chapitre). Plus généralement, les adresses bitcoin sont une représentation générique de différents types de destinataires, ce qui permet d'utiliser les transactions bitcoins pour de nombreux types de paiement, de la même façon qu'un chèque peut être utilisé pour payer des personnes privées, des entreprises, retirer de l'argent liquide... Une adresse bitcoin est simplement la représentation des clefs qui sera visible et partagée, via la blockchain, avec tout le monde.

Dans ce chapitre, nous allons présenter les portemonnaies, qui gèrent les clefs cryptographiques. Nous étudierons comment ces clefs sont générées, sauvegardées et utilisées. Nous passerons en revue les différents formats utilisés représenter pour les clefs privées et publiques, les adresses et les adresses de script. Puis nous étudierons des usages spécifiques: signature de message, preuve de propriété, génération d'adresses personnalisées, portemonnaies papier.

Cryptographie à clef publique et Cryptomonnaies

La cryptographie à clef publique a été inventée dans les années 1970 et est un des fondements de la sécurité informatique.

Depuis l'invention de la cryptographie à clef publique, plusieurs fonctions mathématiques comme

l'exponentiation modulaire ou la multiplication en courbes elliptiques ont été découvertes. Ces fonctions sont pratiquement impossibles à inverser: on peut calculer y = f(x) si on connait x, mais il est pratiquement impossible de calculer x tel que f(x) = y si on connait y. En utilisant ce type de fonctions, la cryptographie permet de protéger des secrets et de créer des signatures numériques inviolables. Bitcoin utilise une cryptographie à clef publique basée sur les courbes elliptiques.

Bitcoin utilise la cryptographie à clef publique pour générer une paire de clefs qui permet de contrôler les fonds: une clef privée, et une clef publique unique (dérivée à partir de la clef privée). La clef publique est utilisée pour recevoir des bitcoins, et la clef privée pour signer des transactions qui dépensent ces bitcoins.

Il existe une relation mathématique entre la clef privée et la clef publique qui permet d'utiliser la clef privée pour signer un message, et la clef publique pour valider cette signature sans réveler la clef privée.

Pour dépenser ses bitcoins, un utilisateur crée une transaction qui contient sa clef publique et une signature (différente pour chaque transaction, mais génerée avec la même clef privée). Lorsque cette transaction est publiée sur le réseau bitcoin, tout le monde peut vérifier que la signature est valide et donc que l'utilisateur a bien le droit de transférer les fonds.

TIP La plupart des portemonnaies sauvegardent les clefs publiques et privées ensemble en tant que *paire de clefs*. Mais vu qu'il est possible de calculer la clef publique à partir de la clef privée, on peut aussi se contenter de ne sauvegarder que la clef privée.

Clefs publiques et privées

Un portemonnaie bitcoin gère un ensemble de paires de clefs privé/publique. La clef privée (k) est un nombre, choisi aléatoirement en général. La clef publique (K) est générée à partir de la clef privée en utilisant la multiplication en courbes elliptiques, une fonction cryptographique non-inversible. On utilise ensuite un hash cryptographique pour générer l'adresse (A) à partir de la clef publique. Dans cette section, nous commencerons par générer la clef privée, étudierons les bases mathématiques des courbes elliptiques, calculerons la clef publique à partir de la clef privée, et générerons une adresse bitcoin à partir de la clef publique. Les relations entre clef privée, clef publique et adresse sont illustrées ici: Clef privée, clef publique et adresse bitcoin.

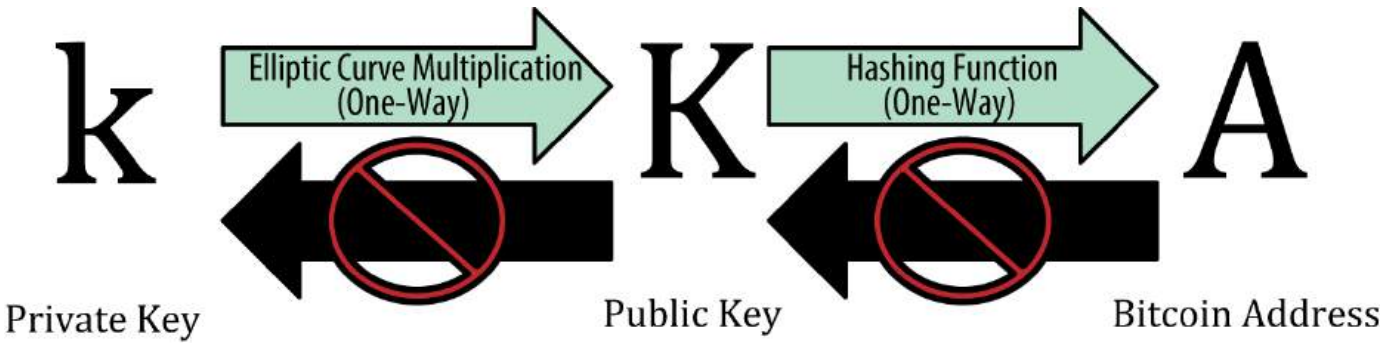


Figure 1. Clef privée, clef publique et adresse bitcoin

Clefs privées

Une privée est juste un nombre choisi au hasard. Pour un utilisateur bitcoin, la possession et la gestion de ses clefs privées est fondamentale pour le contrôle de ses fonds. La clef privée est utilisée pour créer des signatures qui permettent de prouver que l'on détient les fonds que l'on veut dépenser. Elle doit impérativement rester secrète: la divulguer revient à perdre le contrôle des fonds qu'elle protège. Elle doit aussi être sauvegardée: si elle est perdue il sera impossible de la régénérer et les fonds associés seront perdus aussi.

TIP

"private keys","creating by hand")Une clef privée bitcoin est juste un nombre, que l'on peut générer aléatoirement avec un papier, un crayon et une pièce de monnaie: on joue à pile ou face 256 fois pour générer les 256 bits d'une clef privée que l'on pourra importer dans son portemonnaie. La clef publique pourra ensuite être générée à partir de la clef privée.

Générer une clef privée à partir d'un nombre aléatoire

"private keys","generating from random numbers", id="ix_ch04-asciidoc3", range="startofrange")Pour générer des clefs la première étape, et la plus importante, est de trouver une bonne source d'entropie, c'est-à-dire de données alátoires. Créer une clef bitcoin revient à choisir un nombre aléatoire entre 1 et 2^256. La méthode utilisée importe peu, tant qu'elle est imprévisible et non répétable. Le logiciel bitcoin utilise le générateur aléatoire du systême d'exploitation pour générer 256 bits d'entropie. En général, ce générateur est initialisé par des données aléatoires produites par l'utilisateur, par exemple en lui demandant de bouger sa souris au hasard pendant quelques secondes. Mais pour les vrais paranoïaques, rien ne vaut une feuille de papier, un stylo et un dé.

Plus précisement, une clef privée peut être n'importe quel nombre compris entre 1 et +n - 1, n étant l'ordre de la courbe utilisée par bitcoin (n = 1.158 * 10^{77}, juste un peu moins que 2^{256})(voir La cryptographie sur les courbes elliptiques). Pour créer une telle clef, on choisit un nombre aléatoire à 256 bits et on vérifie qu'il est inférieur à n -1. Programmatiquement, on utilise habituellement une longue suite de nombres aléatoires générés par une source cryptographiquement sûre, que l'on hash avec l'algorithme SHA256 (qui produit un résultat sur 256 bits). Si le résultat est pls petit que n - 1, on a une clef privée valide, sinon on recommence.

TIP

Ne codez pas vous-même un générateur de nombres aléatoires, et n'utilisez pas les générateurs les plus simples fournis par votre langage de programmation.Utilisez un générateur de nombres aléatoires cryptographiquement sûr (CSPRNG en anglais) initialisé avec une graine (seed) provenant d'une source ayant suffisamment d'entropie. Vérifiez dans la documentation que votre générateur de nombres aléatoires est bien cryptographiquement sûr. Une implémentation correct du CSPRNG est indispensable pour la sécurité de vos clefs.

Voici la représentation d'une clef privée générée aléatoirement au format hexadécimal (256 bits groupés en 64 nombres sur 4 bits, chaque nombre de 4 bits étant représenté par une lettre).

```
1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
```

TIP La taille de l'espace des clefs bitcoin, 2^{256}, est une nombre gigantesque, qui défie l'imagination. Cela correspond environ à 10^{77} en base 10. On estime que le nombre total d'atomes dans l'univers visible est à peu près 10^{80}.

Pour générer une nouvelle clef avec le client Bitcoin Core (voir [ch03_bitcoin_client]), utilisez la commande getnewaddress. Pour des raisons de sécurité, seule la clef publique sera affichée. Utilisez la commande dumpprivkey pour afficher la clef privée. Elle sera affichée au format WIF (*Wallet Import Format*), un format qui contient la clef et une checksum, le tout encodé en base 58. Nous examinerons ce format en détail au chapitre Les formats de clefs privées. Voici un exemple de génération et affichage de clefs avec ces 2 commandes:

```
$ bitcoind getnewaddress
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
$ bitcoind dumpprivkey 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ
```

La commande dumpprivkey ouvre le portemonnaie et extrait la clef privée générée par la commande getnewaddress. Il est impossible pour bitcoind de retrouver la clef privée à partir de la clef publique, à moins qu'elles ne soient toutes les 2 stockées dans le portemonnaie.

TIP La commande dumpprivkey ne génère par la clef privée à partir de la clef publique, c'est impossible. Elle va juste retrouver la clef privée déja stockée dans le portemonnaie et créée par la commande getnewaddress.

On peut aussi utiliser l'outil en ligne de commande Bitcoin Explorer (voir [libbitcoin]) pour générer et afficher des clefs privées avec les commandes seed, ec-new and ec-to-wif:

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

Clefs publiques

La clef public est calculée à partir de la clef privée en utilisant la multiplication en courbes elliptiques, qui n'est pas inversible: \(K = k * G\) où *k* est la clef privée, *G* est un point particulier appelé le *générateur de la courbe* et *K* est la clef publique. L'opération inverse —calculer *k* à partir de *K*, c'est-à-dire résoudre le logarithme discret — est aussi difficile qu'une attaque par force brute: essayer toutes les valeurs possibles pour k. Afin de montrer comment calculer la clef publique à partir de la clef privée, nous allons étudier plus précisement la cryptographie en courbes elliptiques.

La cryptographie sur les courbes elliptiques

La cryptographie sur les courbes elliptiques est un type pariculier de cyptographie assymétrique (i.e. à clef privée/clef publique) basée sur le problème du logarithme discret dans le groupe correspondant à la courbe elliptique (que l'on définit en introduisant les notions d'addition de points sur la courbe, et de multiplication de point par un entier).

An elliptic curve est un exemple de courbe elliptique, similaire à celle utilisée par bitcoin.

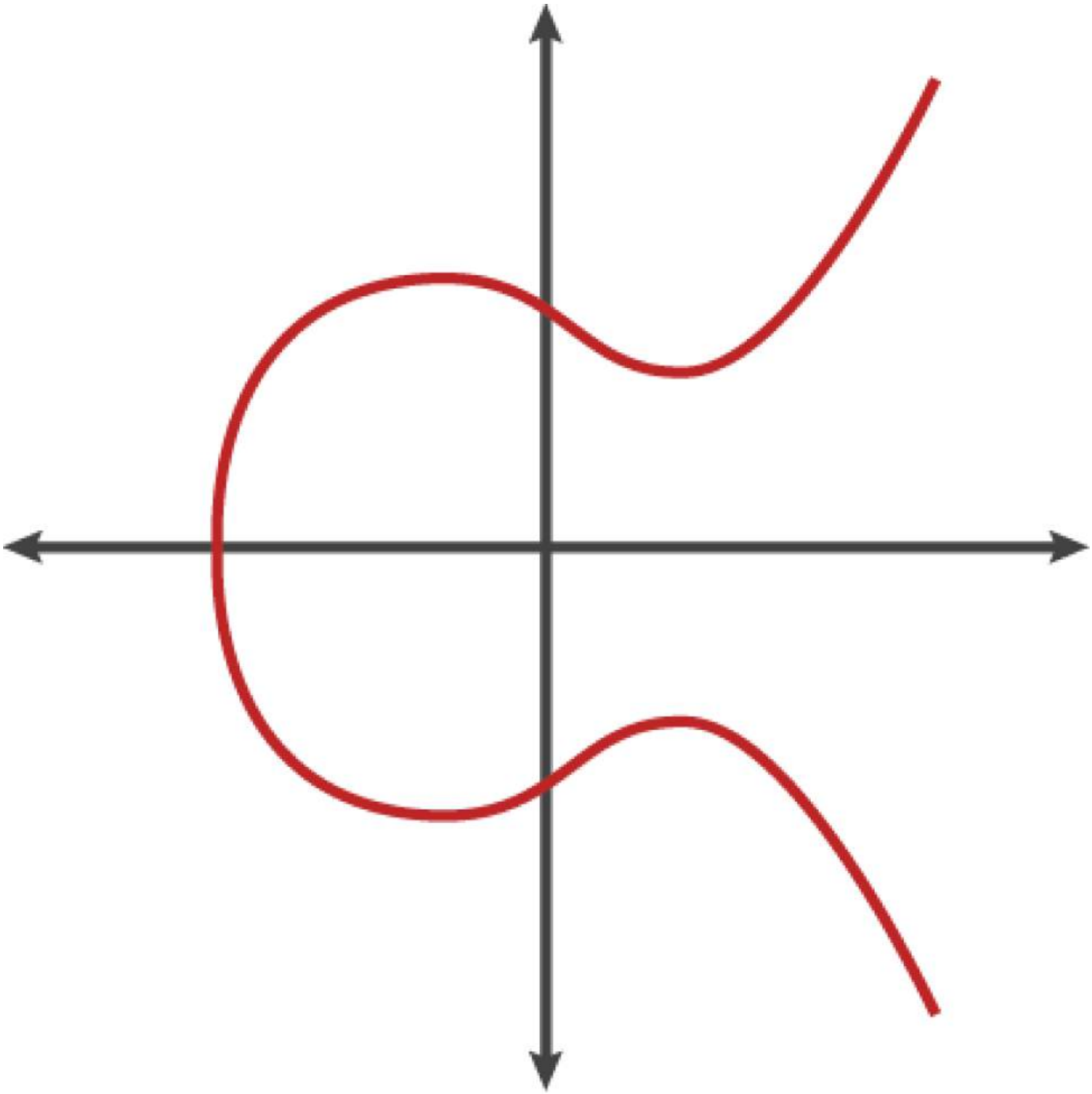


Figure 2. An elliptic curve

Bitcoin utilise une courbe particulière définie dans le standard "secp256k1 curve standard") secp256k1,

maintenu par le National Institute of Standards and Technology (NIST). La courbe secp256k1 est définie par la fonction suivante:

ou

```
_mod p_ (modulo le nombre premier p) signifie que cette courbe est définie sur le corps
fini d'ordre _p_ premier, que l'on écrit aussi latexmath:[\(\mathbb{F}_p\)], ou p =
2^256^ – 2^32^ – 2^9^ – 2^8^ – 2^7^ – 2^6^ – 2^4^ – 1, un nombre premier très grand.
```

Cette courbe n'est pas définie sur le corps des réels mais sur un corps fini d'ordre premier: elle ressemble à un ensemble de points répartis au hasard, ce qui la rend difficile à visualiser. Néanmois, les mathématiques qui régissent cette courbe sont les mêmes que pour les courbes sur les nombre réels. Par exemple, la figure Cryptographie en courbes elliptiques: visualisation d'une courbe définie sur F(p), avec p=17 montre la même courbe mais définie sur un corps d'ordre 17, beaucoup plus petit. On peut voir la forme de la courbe dans le nuage de points. La courbe secp256k1 utilisée par bitcoin peut être imaginée comme une forme beaucoup plus complexe sur une grille infiniment plus grande.

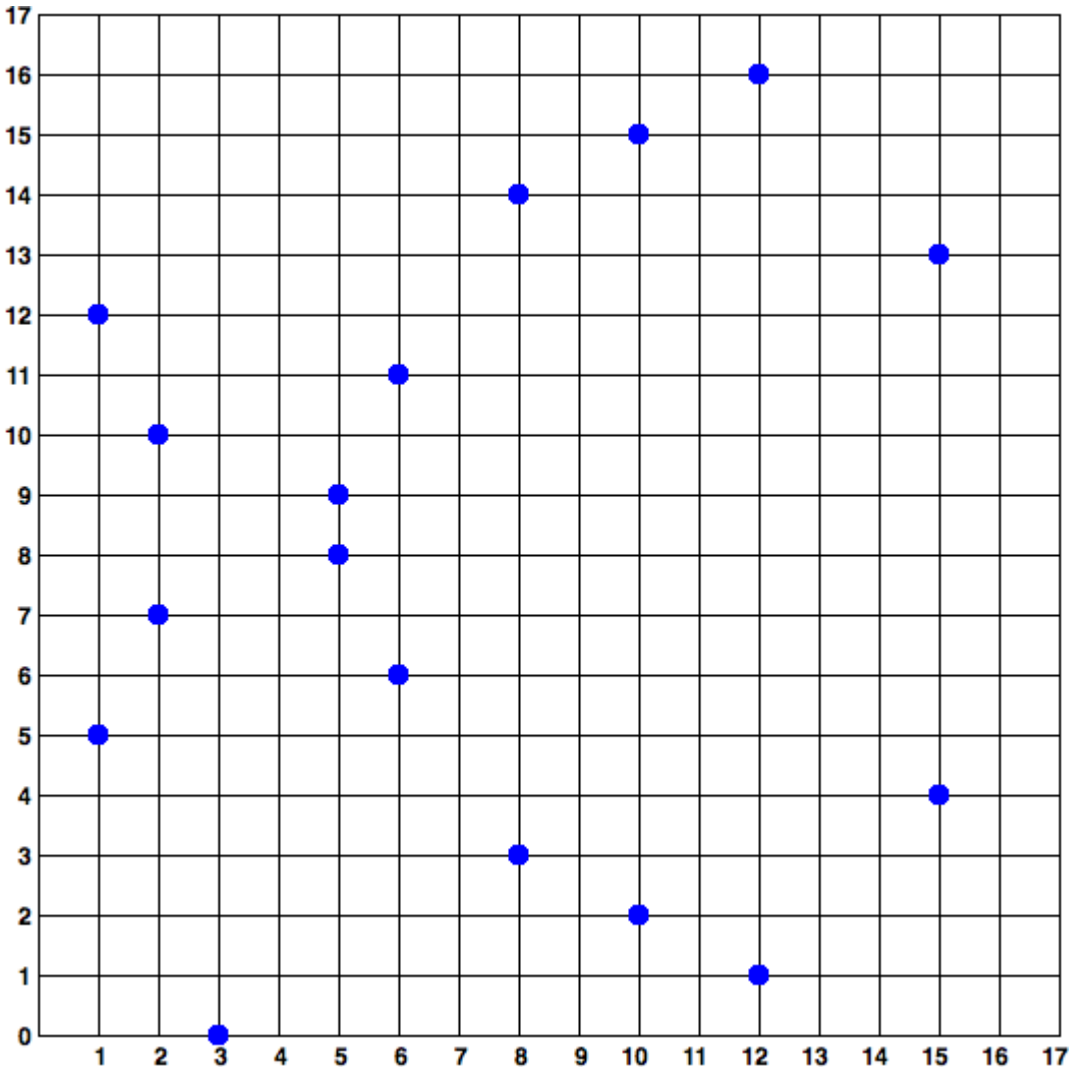


Figure 3. Cryptographie en courbes elliptiques: visualisation d'une courbe définie sur F(p), avec p=17

Par exemple, voici un point P de coordonnées (x, y) sur la courbe secp256k1. On peut le vérifier avec Python:

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

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

En courbes elliptiques il existe un point appelé le "point à l'infini" qui correspond en gros à l'élement neutre 0 pour l'addition. On le représente parfois par x = y = 0 (ce qui ne satisfait pas l'équation de la courbe, mais c'est un c'est un cas particulier que l'on peut facilement traiter à part).

Il existe aussi un opérateur +, appelé "addition," qui se comporte comme l'addition sur les nombre réels que l'on apprend à l'école élementaire. Soit P_1 et P_2 2 points sur la courbe elliptique, il existe un troisième point $P_3 = P_1 + P_2$, qui est aussi sur la courbe elliptique.

D'un point de vue géométrique, on calcule P_3 en traçant une ligne entre P_1 et P_2. Cette ligne coupe la courbe elliptique en exactement un point que l'on appelera $P_3' = (x, y)$. On prend alors le symmétrique de P_3' par rapport à l'axe des x, ce qui donne $P_3 = (x, –y)$.

Certains cas particuliers illustrent pourquoi on a besoin du "point à l'infini".

Si P_1 et P_2 sont en fait le même point, la droite entre P_1 et P_2 est en fait la tangente à la courbe au point P_1. Cette tangente coupe la courbe en exactement un point. Les techniques issues de l'analyse numérique permet de calculer cette tangente. Etonnamment, ces techniques fonctionnent alors que l'on se restreint à des points sur la courbes dont les 2 coordonnées sont entières!

Dans certains cas (par exemple si P_1 and P_2 ont la même abscisse x mais des ordonnnées y différentes), la ligne qui les relie sera verticale, et P3 sera le "point à l'infini".

Si P_1 est le "point à l'infini," alors $P_1 + P_2 = P_2$. De même, si P_2 est le point à l'infini, alors $P_1 + P_2 = P_1$. Le "point à l'infini" joue le même rôle que l'élement neutre 0.

Il s'avère que + est associatif, ce qui signifie que (A + B) + C = A + (B + C). On peut donc écrire A + B + C sans parenthèses et sans ambiguité.

Maintenant que nous avons défini l'addition, on peut s'en servir pour définir la multiplication: Pour un point P sur la courbe elliptique et un entier k, on définit kP = P + P + P + ... + P (k fois). Remarque: en Anglais k est parfois appelé "exponent" (exposant) ce qui peut être une source de confusion.

Generation d'une clef publique

On part d'une clef privée sous la forme d'un nombre aléatoire *k*, que l'on multiplie par un point particulier de la courbe appelé *point générateur G* pour obtenir un autre point sur la courbe: la clef publique *K*. Ce point générateur est défini dans le standard secp256k1 et est toujours le même, pour toutes les clefs:

ou k est la clef privée, G le générateur de la courbe et K la clef publique (qui est un point sur la courbe). Comme le point générateur est toujours le même, la même clef privée k donnera toujours la même clef publique K. Il y a lien direct entre k et K, mais on ne peut le calculer que dans un sens, de k vers K. C'est pourquoi une clef publique peut être partagée avec tout le monde, sans risque de révéler la clef privée.

TIP On peut calculer la clef publique à partir de la clef privée, mais pas l'inverse, car les fonctions mathématiques utilisées ne sont pas inversibles.

Concrètement, avec la multiplication en courbes elliptiques, on prend la clef privé k générée précedemment que l'on multiplie par le générateur G pour obtenir la clef publique K:

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

La clef publique K est le point K = (x,y):

```
K = (x, y)

avec:

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Pour visualiser la multiplication d'un point par un entier, on utilisera une courbe elliptique plus simple, définie sur les nombres réels - les bases mathématiques sont les mêmes. L'objectif est de trouver le multiple kG du point générateur G, c'est-à-dire ajouter G à lui-même k fois. En courbes elliptiques, ajouter un point à lui-même revient à trouver l'intersection entre la tangente en ce point et la courbe, et prendre le symétrique par rapport à l'axe des x.

Cryptographie sur les courbes elliptiques: visualisation de la multiplication d'un point G par un entier k sur une courbe elliptique. montre comment calculer G, 2G, 4G de façon géométrique.

TIP La plupart des implémentations bitcoin utilisent la librairie cryptographique OpenSSL pour la cryptographie en courbes elliptiques. Par exemple, pour calculer la clef privée on utilise la fonction EC_POINT_mul().

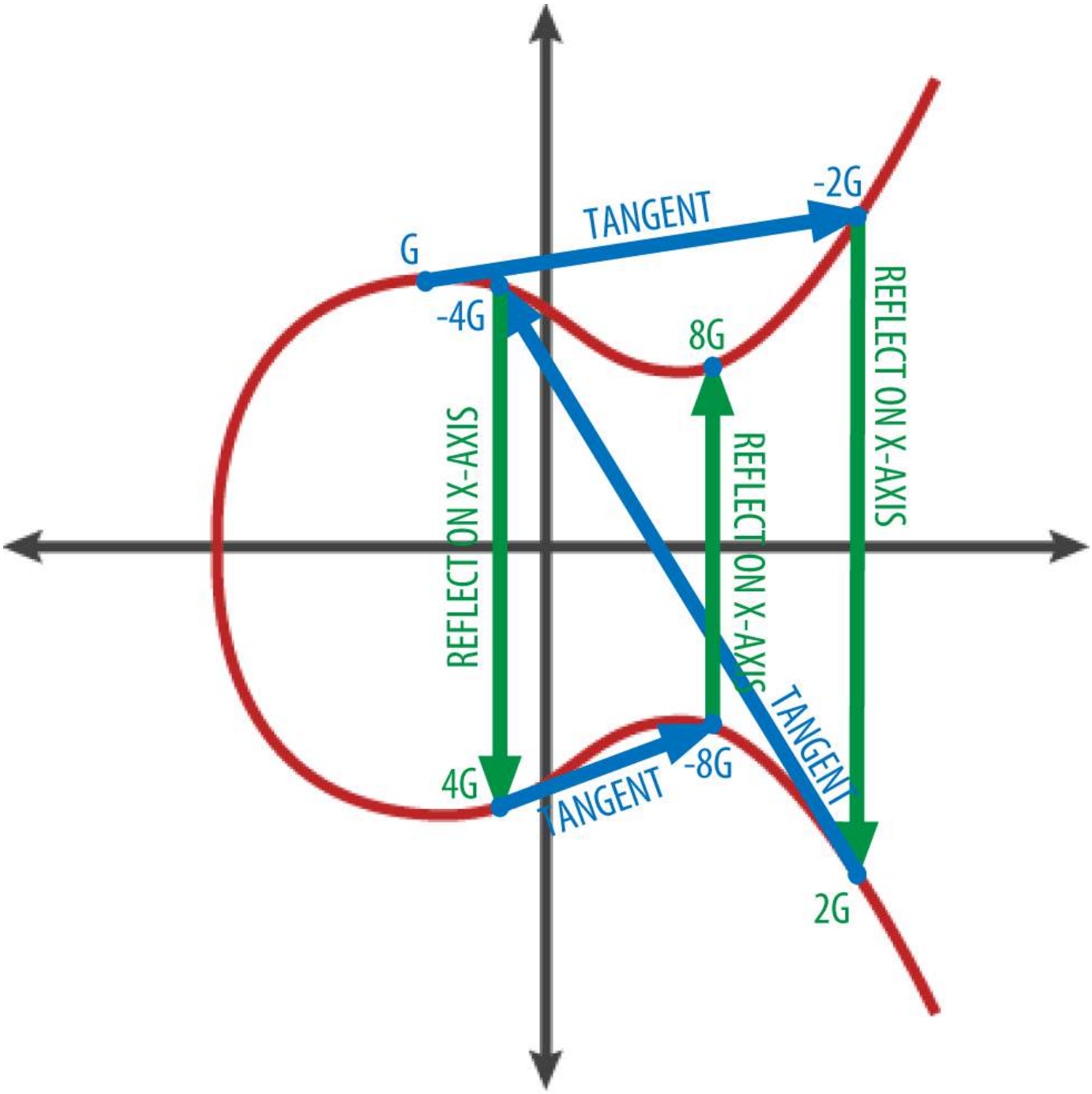


Figure 4. Cryptographie sur les courbes elliptiques: visualisation de la multiplication d'un point G par un entier k sur une courbe elliptique.

Adresses Bitcoin

Une adresse bitcoin est une chaîne de caractères (chiffres et lettres) que l'on peut communiquer avec n'importe quelle personne souhaitant vous envoyer des fonds. Une adresse génerée à partir d'une clef

publique commence par "1". Voici un exemple d'adresse bitcoin:

```
1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
```

Dans une transaction bitcoin, l'adresse est ce qui correrspond au "déstinataire" des fonds. L'adresse correspond à la ligne "Payez à" sur un chèque. Sur cette ligne, on peut inscrire le nom d'une personne, d'une entreprise, ou même "au porteur" pour que n'importe qui puisse encaisser le chèque et l'utiliser comme du cash ce qui fait des chèques un moyen de paiement très flexible, beaucoup plus que s'il fallait inscrire un numéro de compte en banque. Les adresses bitcoin ont la même flexibilité: elles peuvent représenter le détenteur d'une paire de clefs publique/privée, ou autre chose comme un script de paiement (voir [p2sh]). Pour l'instant, examinons le cas le plus simple: une adresse bitcoin dérivée d'une clef publique.

On utilise un hash cryptographique pour dériver une adresse bitcoin à partir d'une clef publique. Un hash cryptographique est une fonction non inversible qui calcule une empreinte numérique à partir d'une donnée de taille quelconque. Les hash cryptographiques sont très utilisés dans bitcoin: pour les adresses bitcoin, les adresses de script, et la "preuve de travail" (Proof Of Work) utilisée pour le minage. Les algorithmes de hash utilisés pour transformer une clef publique en adresse bitcoin sont "Secure Hash Algorithm (SHA)") Secure Hash Algorithm (SHA) et RACE Integrity Primitives Evaluation Message Digest (RIPEMD), plus précisement les versions SHA256 et RIPEMD160.

On calcule d'abord le hash SHA256 de la clef publique K, puis on calcule le hash RIPEMD160 du résultat, ce qui donne une nombre de 160 bits (20 octets):

K est la clef publique et A l'adresse bitcoin correspondante.

TIP Une adresse bitcoin *n'est pas* la même chose qu'une clef publique: les adresses sont calculées à partir des clefs publiques en utilisant une fonction non inversible.

Les adresses bitcoin sont presque toujours affichées en utilisant l' "Base58Check" (voir Encodages Base58 et Base58Check), qui utilise 58 caractères (un encodage en base 58) et une checksum pour améliorer la lisibilité, éviter les amibguïtés et les erreurs lors de la saisie et de la copie des adresses. L'encodage Base58Check est très utilisé dans bitcoin pour afficher de façon fiable des nombres aux utilisateurs: adresses bitcoin, clefs privées, clefs chiffrées, hash de scripts. Au prochain paragraphe nous étudierons l'encodage Base58Check. Voici un exemple de conversion de clef publique en adresse bitcoin: Clef publique vers adresse bitcoin: conversion d'une clef publique en adresse bitcoin.

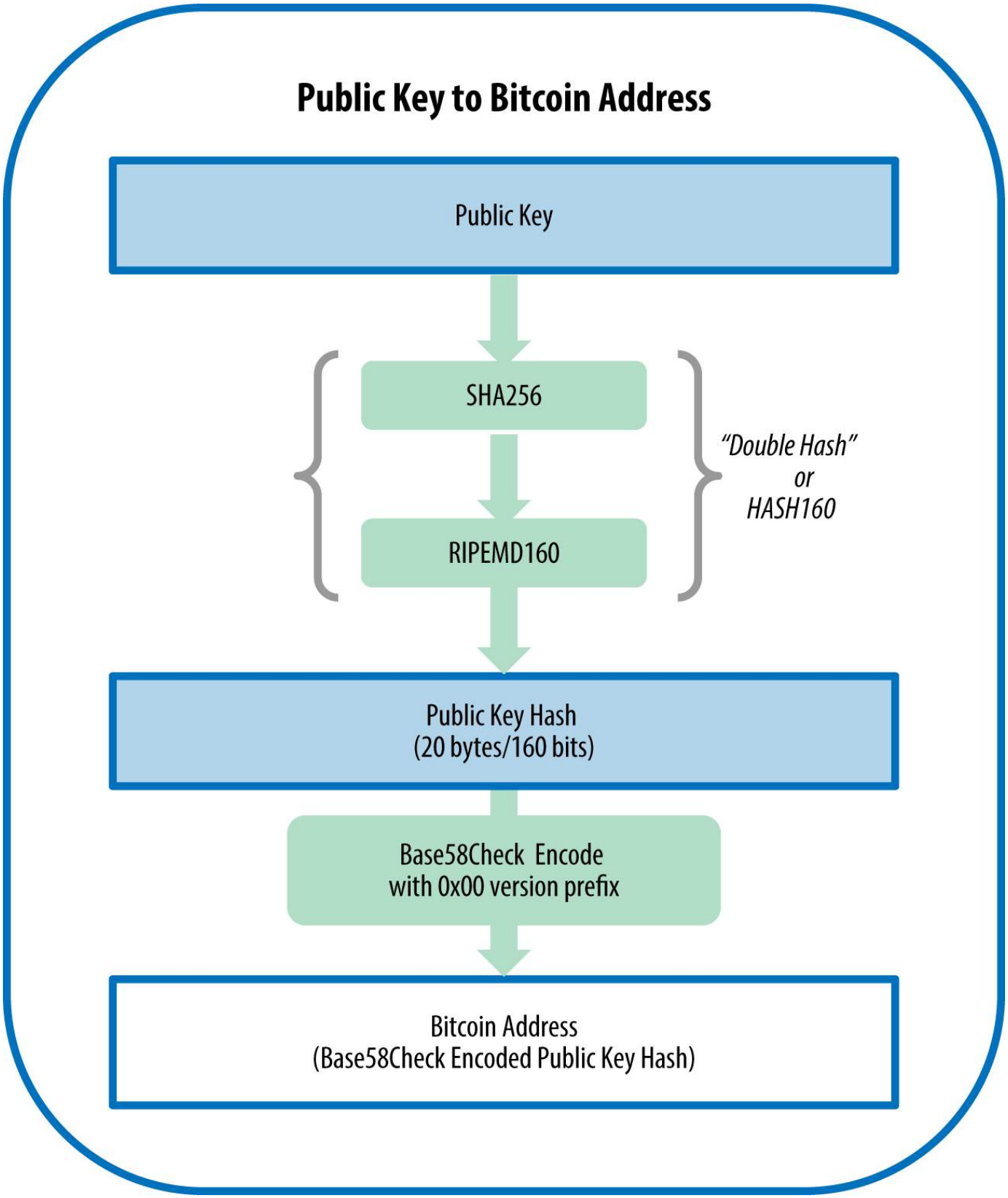


Figure 5. Clef publique vers adresse bitcoin: conversion d'une clef publique en adresse bitcoin

Encodages Base58 et Base58Check

Pour représenter de grands nombres de façon compacte on utilise souvent en informatique des représentations en base N avec N > 10. Le système décimal, en base 10, utilise les chiffres de 0 à 9. Mais

le système hexadécimal, en base 16, utilise aussi les lettres de A à F pour représenter les nombres de façon plus compacte que le système décimal. L'encodage Base-64 est encore plus compact et utilise 64 symboles (les 26 lettres de l'alphabet en minuscule, les 26 lettres en majuscule, les 10 chiffres et 2 symboles supplémentaires: "+" et "/"). Il est très utilisé pour transmettre des données binaires sous forme de texte, comme des attachements à des emails par exemple. Base58 est un système permettant d'encoder des données binaires sous forme de texte, développé pour bitcoin et utilisé par beaucoup d'autres crypto-monnaies. Il offre un bon conpromis entre compacité, lisibilité et précention/detection des erreurs. Base58 utilise un sous-ensemble des symboles de Base64 (lettres minuscules et majuscules, chiffres), en éliminant les symboles qui peuvent prêter à confusion, notamment avec certains jeux de caratctères: 0 (zéro) et O (la lettre O), l (L minuscule) et I (i majuscule), et les symboles "\+" et "/". Au final, il ne reste que les lettres minuscules et majuscules et les chiffres, sans les symboles (0, O, l, I).

Example 1. alphabet Base58 utilisé par bitcoin

```
123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz
```

Base58Check est un format basé sur Base58 très utilisé par bitcoin. Il contient une checksum qui permet de se protéger contre les fautes de frappes et erreurs de transmissions. Cette checksum, de 4 octets, est ajoutée à la fin des données encodées. Elle est calculée à partir du hash des données encodées et permet de détecter et éviter les erreurs. Lorsque l'on décode du Base58Check, on calcule la checksum et on la compare à celle qui se trouve à la fin des données encodées. Si ce n'est pas la même, cela veut dire qu'il y a une erreur et que les données Base58Check sont invalides. Par exemple, cela permet à un portemonnaie bitcoin de détecter qu'une adresse bitcoin contient une faute de frappe, et évite ainsi d'envoyer des fonds qui seront perdus.

Pour convertir des données binaires (représentant un nombre) au format Base58Check, on ajouter d'abord un préfixe appelé "version" (généralement sur un octet), qui sert à identifier le type de donnée que l'on encode. Par exemple, pour une adresse bitcoin le préfixe est 0 (0x00 en hexa), et pour une cle privée c'est 128 (0x80 en hexa). La liste des préfixes les plus courants est disponible ici: [base58check_versions].

Ensuite on calcule un "double hash" SHA256 du préfixe et des données: on hash préfixe + données, et on hash le résultat.

```
checksum = SHA256(SHA256(prefix+data))
```

Cela donne un hash sur 32 octets (256 bits) dont on garde les 4 premiers octets qui constituent notre checksum. Cette checksum est rajoutée à la fin des données.

Le résultat est donc composé de 3 parties: préfixe, données et checksum. Il est ensuite encodé en base 58 en utilisant l'alphabet présenté un peu plus haut. L'encodage Base58Check est illustré ici: encodage Base58Check: un format basé sur Base58 et comprenant un préfixe et une checksum, pour un encodage sans ambiguïté des données bitcoin.

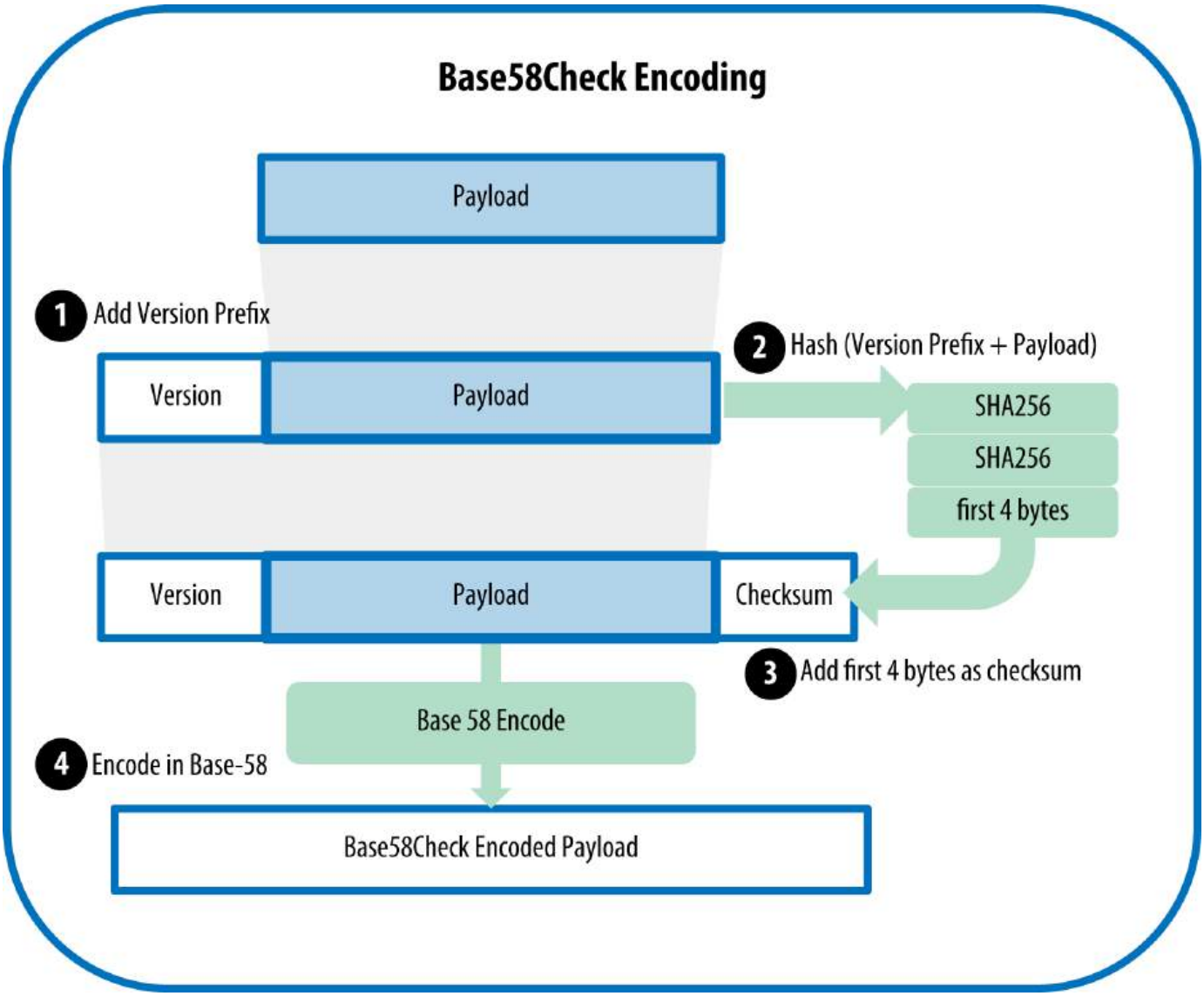


Figure 6. encodage Base58Check: un format basé sur Base58 et comprenant un préfixe et une checksum, pour un encodage sans ambiguïté des données bitcoin.

Dans bitcoin, la plupart des données que voient les utilisateurs sont encodées en Base58Check afin de les rendre plus compactes, plus faciles à lire et de détecter les erreurs facilement. L'octet de version permet de distinguer facilement le type de données encodées: le résultat commencera par des caractères spécifiques, ce qui permet aux utilisateurs de reconnaitre facilement ce qui est encodé et comment s'en servir. Par exemple, c'est ce qui permet de différencier les adresses bitcoin, qui commencent par un 1, des clefs privées qui, une fois encodées en Base58Check (format WIF), commencent par un 5. Quelques valeurs de l'octet version et du résultat une fois encodé se trouvent ici: [base58check_versions].

1. préfixes utilisés par Base58Check, et résultat une fois encodé.

Type	préfixe Version (hex)	résultat Base58
Bitcoin Address	0x00	1

Pay-to-Script-Hash Address	0x05	3
Bitcoin Testnet Address	0x6F	m or n
Private Key WIF	0x80	5, K or L
BIP38 Encrypted Private Key	0x0142	6P
BIP32 Extended Public Key	0x0488B21E	xpub

Examinons les étapes du processus de création d'une adresse bitcoin: on part de la clef privée, on calcule la clef publique (un point sur la courbe elliptique), puis le double hash et enfin l'encodage Base58Check. Le code C++ Création d'une adresse bitcoin au format Base58Check à partir d'une clef privée implémente toutes ces étapes, de la clef privée à l'adresse bitcoin au format Base58Check. Ce code utilise la librairie libbitcoin, que nous avons présenté au chapitre [alt_libraries] .

Example 2. Création d'une adresse bitcoin au format Base58Check à partir d'une clef privée

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

int main()
{
    // Private secret key.
    bc::ec_secret secret;
    bool success = bc::decode_base16(secret,
        "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");
    assert(success);
    // Get public key.
    bc::ec_point public_key = bc::secret_to_public_key(secret);
    std::cout << "Public key: " << bc::encode_hex(public_key) << std::endl;

    // Create Bitcoin address.
    // Normally you can use:
    //   bc::payment_address payaddr;
    //   bc::set_public_key(payaddr, public_key);
    //   const std::string address = payaddr.encoded();

    // Compute hash of public key for P2PKH address.
    const bc::short_hash hash = bc::bitcoin_short_hash(public_key);

    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;
}
```

Ce code utilise toujours la même clef privée, et produit donc toujours la même adresse: Compilation et

exécution du code addr.

Example 3. Compilation et exécution du code addr

```
# Compillation de addr.cpp
$ g++ -o addr addr.cpp $(pkg-config --cflags --libs libbitcoin)
# execution du programme addr
$ ./addr
Public key: 0202a406624211f2abbdc68da3df929f938c3399dd79fac1b51b0e4ad1d26a47aa
Address: 1PRTTaJesdNovgne6Ehcdu1fpEdX7913CK
```

Les différentes formats de clefs

Il existe différents formats pour représenter les clefs publiques et privées. Les résultats semblent différents mais il s'agit toujours de représentations des même nombres. Ces formats sont surtout utilisés pour rendre les clefs faciles à lire et à recopier, sans risque d'erreur.

Les formats de clefs privées

Les clefs privées peuvent être représentées de différentes manières, qui correspondent toujours au même nombre de 256 bits. Les 3 formats les plus utilisés pour les clefs privées sont décrits ici:Formats d'encodage des clefs privées

Table 1. Formats d'encodage des clefs privées

Type	Préfixe	Description
Hex	Aucun	64 caractères (chiffres et lettres de A à F)
WIF	5	encodage Base58Check: encodage Base58 avec un préfixe(version) de 128 et une checksum de 32 bits
WIF-compressed	K ou L	Comme au-dessus, mais en ajoutant le suffixe 0x01 à la clef avant d'encoder

Exemple: la même clef, différents formats montre la clef privée encodée avec ces 3 formats/

Table 2. Exemple: la même clef, différents formats

Format	Clef Privée
Hex	1e99423a4ed27608a15a2616a2b0e9e52ced330ac530edcc32c8ffc6a526aedd

Format	Clef Privée
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

Ce sont toutes des représentations différentes du même nombre (la clef privée). Ces formats ont l'air différents mais on peut facilement convertir de l'un vers l'autre.

On va utiliser la commande wif-to-ec de l'outil Bitcoin Explorer (voir [libbitcoin]) pour vérifier que les 2 clefs WIF représentent bien la même clef privée:

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

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

Décodage du format Base58Check

L'outil Bitcoin Explorer (voir[libbitcoin]) permet d'écrire facilement des scripts et outils en ligne de commande pour manipuler les clefs, adresses et transactions bitcoin. On peut utiliser Bitcoin Explorer pour décoder le format Base58Check en ligne de commande.

On utilise la commande base58check-decode pour décoder la clef non-compressée:

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

Le résultat comprend: la clef ("payload"), le préfixe WIF (Wallet Import Format) qui vaut 128 (0x80), et la checksum.

On remarque qu'à la fin de la clef compressée on trouve le suffixe 01, ce qui signifie que la clef publique correspondant à cette clef privée devra être compressée.

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

Encodage d'hexadécimal vers Base58Check

Pour encoder vers Base58Check (le contraire de la commande précedente), on utilise la commande base58check-encode de l'outil Bitcoin Explorer (voir[libbitcoin]): on entre la clef privée au format hexa, suivie du Wallet Import Format (WIF) préfixe version 128:

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

Encodage hexadécimal (clef compressée) vers Basse58Check

Pour encoder une clef privée au format Base58Check en mode "compressé" (voir Clefs privées compressées), on rajoute le suffixe 01 à la fin de la clef, et on encode:

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

Le résultat, au format WIP compressé, commence par "K". Cela signifie que la clef privée qu'il représente se termine par le suffixe "01" et qu'elle sera utilisée pour produire des clefs publiques compressées (voir Clefs publiques compressées).

Les formats de clefs publiques

Il existe aussi différents formats pour les clefs publiques. En particulier, on peut les représenter de façon *compressée* ou *non-compressée*.

Comme nous l'avons vu précedemment, une clef publique est un point (x,y) sur une courbe elliptique. On l'encode généralement avec le préfixe 04 suivi de 2 nombres sur 256 bits: x et y. Le préfixe 04 permet de distinguer les clefs publiques non-compressées des clefs publiques compressées, qui commencent par 02 ou 03.

Voici la clef publique générée à partir de la clef privée que nous avons créée plus tôt, représentée en tant que point de coordonnées x et y:

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

Voici la même clef publique, sous la forme d'un nombre sur 520 bits (65 octets, donc 130 caractères en hexa): le préfixe 04, suivi par x et y, ce qui donne 04 x y:

```
K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A<?pdf-
cr?>07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB
```

Clefs publiques compressées

<?dbhtml orphans="4"?>Les clefs publiques compressées ont été ajoutées à bitcoin pour réduire la taille des transactions et donc sauver de l’espace disque sur les noeuds bitcoin. La plupart des transactions contiennent une clef publique, nécessaire pour valider que l’utilisateur a le droit de dépenser les fonds associés. Chaque clef publique prend 520 bits (prefix \+ x \+ y), ce qui, multiplié par plusieurs centaines de transactions par bloc, donc plusieurs dizaines de milliers de transactions par jour, prend beaucoup de place dans la blockchain.

Comme nous l'avons vu dans la section Clefs publiques, une clef publique est un point (x, y) sur une courbe elliptique. Cela veut dire que (x, y) est une solution à l'équation de la courbe. Donc, si l'on connait *x*, on peut calculer *y* en résolvant l'équation $y^2 \bmod p = (x^3 + 7) \bmod p$. Cela nous permet de ne garder que *x* pour représenter le point, et on économise ains 256 bits. Cela donne une réduction de la taille des transactions de presque 50%, ce qui avec le temps permet d'économiser beaucoup de place!

Alors que les clef publiques non-compressées commencent par le préfixe 04, les clefs publiques compressées commencent par 02 ou 03. Il y a 2 préfixes possibles parce qu'il y a toujours 2 solutions possibles à l'équation de la courbe: à gauche du signe = on trouve y^2, donc les 2 solutions seront + ou - la racine carrée de ce qu'il y a à droite du signe =. Visuellement, cela se traduit de la façon suivante: la courbe (voir An elliptic curve) est symmétrique par rapport à l'axe des x: pour chaque point (x, y) sur la courbe, le point (x, -y) est aussi sur la courbe. Pour pouvoir identifier un point de façon unique, il nous faut donc x et le *signe* de y. Mais comme les calculs sur la courbe elliptique se font sur un corps fini d'ordre p premier, signe et parité sont équivalents: on va donc utiliser le préfixe 02 si y est pair, et 03 si y est impair. Cela nous permettra de recalculer *y* sans ambiguïté à partir de *x*. Ce procédé de compression des clefs publiques est illustré ici: Compression des clefs publiques.

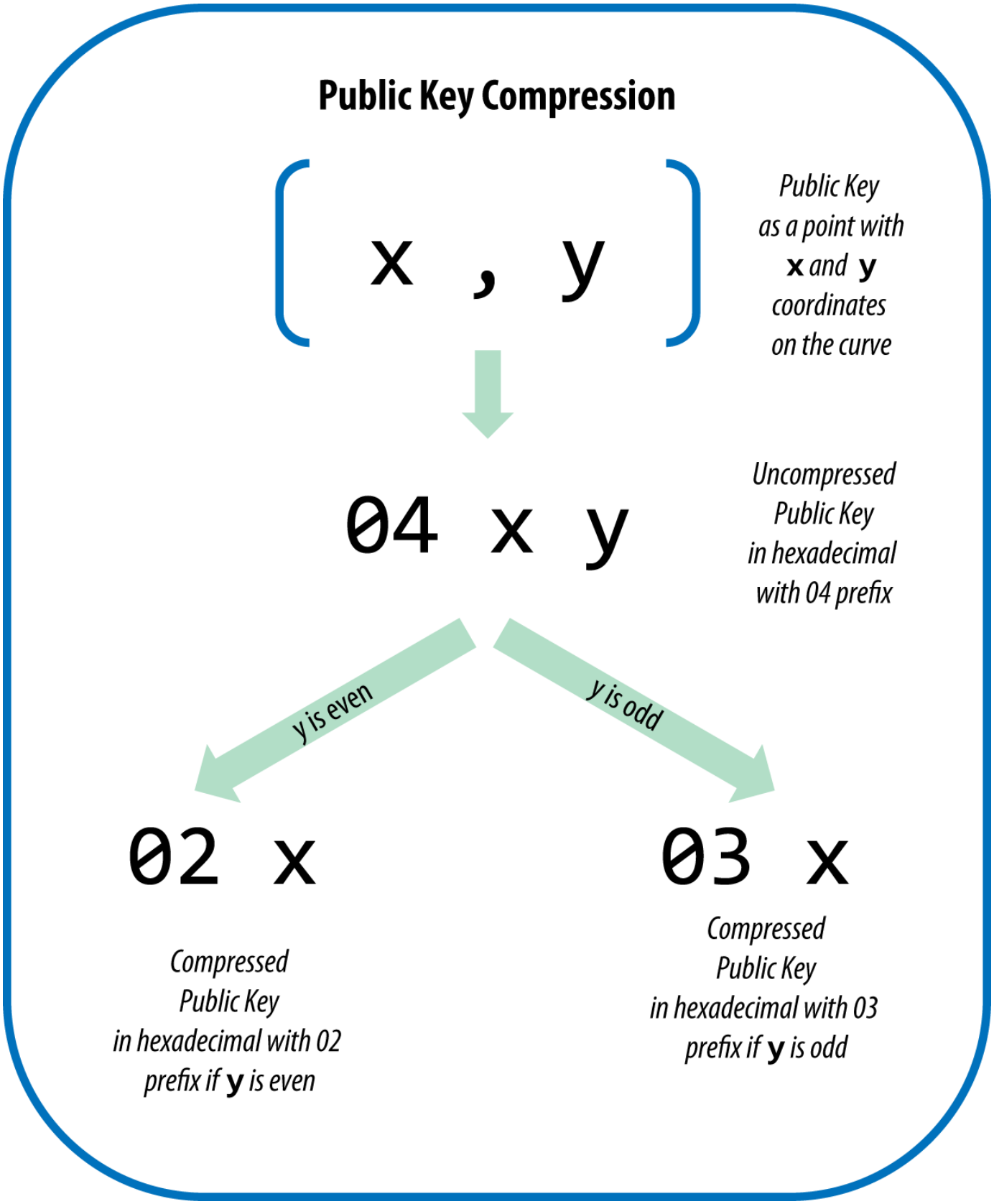


Figure 7. Compression des clefs publiques

Voici la même clef publique que précedemment, sous forme compressée (33 octets, soit 264 bits ou 66 caractéres en hexa), avec le préfix 03 qui signifie que *y* est impair:

```
K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A
```

Cette clef publique compressée correspond à la même clef privée, mais sa représentation est différente de celle de la même clef publique non-compressée: si on la convertie en adresse bitcoin en utilisant le double hash (RIPEMD160(SHA256(K))) on obtiendra une autre adresse. Cela peut être une source de confusion: 2 adresses bitcoin différentes, mais qui correspondent à la même clef publique (sous forme compressée et non-compressée) et à la même clef privée.

La plupart des clients bitcoin utilisent maintenant les clefs publiques compressées, ce qui permet de réduire fortement la taille des transactions et donc de la blockchain. Néanmoins, tous les clients ne supportent pas encore les clefs publiques compressées. Ceux qui les supportent doivent accepter les transactions provenant de vieux clients qui ne les supportent pas. C'est particulièrement important lorsqu'un portemonnaie importe des clefs privées provenant d'un autre portemonnaie, parce qu'il devra scanner la blockchain pour y trouver les transactions correspondant à ces clefs. Quelles adresses bitcoin doit-il chercher ? Les adresses correspondant aux clefs publiques compressées ? Ou celles correspondant aux clefs publiques non-compressées ? Les 2 sont valides, correspondent aux même clefs privées, mais restent des adresses différentes!

Pour résoudre ce problème, les portemonnaies récents utilise un format WIF (Wallet Import Format) différent pour indiquer que les clefs privées ont été utilisées pour générer des clefs publiques *compressées* et donc des adresses bitcoin *compressées*. Cela permet au portemonnaie qui les importe de distinguer les clefs privées provenant de vieux portemonnaies de celles provenant de portemonnaies récents, et de rechercher dans la blockchain les transactions correspondant aux clefs publiques compressées ou non-compressées selon le cas. Nous examinerons ce processus plus en détail dans la prochaine section.

Clefs privées compressées

Ironiquement, l'expression "clef privée compressée" est trompeuse car les clefs privées exportées au format WIF compressé font un octet *de plus* que les clef privées "non-compressées", à cause du suffixe 01 ajouté à la fin pour indiquer qu'elles proviennent d'un portemonnaie récent et doivent être utilisées pour générer des clefs publiques compressées. En fait, les clefs privées ne sont pas compressées (et ne peuvent pas être compressées). L'expression "clef privée compressée" signifie en fait "la clef publique issue de cette clef privée doit être compressée". Pour éviter de semer la confusion, on appellera ce format d'export "WIF compressé" ou "WIF" et on évitera d'utiliser le terme "compressé" pour les clefs privées.

Attention, ces formats en sont *pas* interchangeables. Les portemonnaies récents qui utilisent des clefs publiques compressées vont toujours exporter les clefs privées au format WIF compressé (commençant par K ou L). Les portemonnaie plus ancien ne supportant pas les clefs publiques compressées exportent les clefs privées au format WIF (commençant par un 5). L'objectif est de permettre au portemonnaie qui importe les clefs de savoir s'il doit rechercher dans la blockchain des adresses et clefs publiques compressées ou non-compressées.

Si un portemonnaie bitcoin supporte les clefs publiques compressées, il les utilisera dans toutes les

transactions. Les clefs privées gérées par le portemonnaie seront utilisées pour générer des clefs publiques (i.e. des points sur la courbe) compressées, qui elles-mêmes généreront des adresses compressées, que le retrouvera dans les transactions. Lorsqu'un portemonnaie qui supporte les clefs publique compressées exporte des clefs privées, il utilise un format WIF modifié: l'octet 01 est ajouté à la fin de la clef privée, qui est ensuite encodée au format Base58Check: on appelle ce format "WIF compressé", et le résultat commence par K ou L, au lieu de "5" comme c'est le cas pour les clefs exportées au format WIF (non compressé) par de vieux portemonnaie.

Exemple: la même clef, différents formats montre la même clef, encodée aux formats WIF et WIF compressé.

Table 3. Exemple: la même clef, différents formats

Format	**Clef Privée**
Hex	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD
WIF	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Hex-compressé	1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD_01_
WIF-compressed	KxFC1jmwwCoACiCAWZ3eXa96mBM6tb3TYzGmf6YwgdGWZgawvrtJ

TIP Le terme "clef privée compressée" est mal choisi! Les clefs privées ne sont pas compressées, mais les clefs publiques qui en sont dérivées doivent être compressées ainsi que les adresses bitcoin correspondantes. Ironiquement, une clef privée au format "WIF compressé" fait un octet de plus à cause du suffixe 01 ajouté à la fin.

Implementation des clefs et adresses en Python

La librairie bitcoin la plus complète en Python est pybitcointools développée par Vitalik Buterin. L'exempleGénération et affichage de clefs et adresses avec la librairie pybitcointools utilise la librairie pybitcointools (importée en tant que "bitcoin") pour générer et afficher des clefs et adresses avec différents formats.

Example 4. Génération et affichage de clefs et adresses avec la librairie pybitcointools

```
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)
```

Execution de key-to-address-ecc-example.py est le résultat de l'execution de ce code.

Example 5. Execution de key-to-address-ecc-example.py

Un exemple d'utilisation des courbes elliptiques pour les clefs bitcoin est un autre exemple basé sur la librairie ECDSA Python pour la cryptographie en courbes elliptiques, sans avoir besoin de librairies bitcoin spécifiques.

Example 6. Un exemple d'utilisation des courbes elliptiques pour les clefs bitcoin

```
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
```

Installation de la librairie Python ECDSA et execution du script ec_math.py est le résultat de l'execution de ce script.

NOTE

Cet exemple utilise os.urandom, qui se base sur un générateur de nombres aléatoires cryptographiquement sûr (CSRNG) fourni par le système d'exploitation. Sur un OS de type UNIX comme Linux, il les lit depuis le fichier /dev/urandom, et sous Windows il appelle CryptGenRandom(). Si aucun générateur convenable n'est disponible, l'exception NotImplementedError est lancée. Bien que le générateur aléatoire utilisé ici soit suffisant pour un exemple, il n'est *pas* suffisamment sécurisé pour générer des clefs bitcoin pour un systême en production.

Example 7. Installation de la librairie Python ECDSA et execution du script ec_math.py

```
$ # Installation du gestionnaire de paquets Python PIP
$ sudo apt-get install python-pip
$ # Installation de la librairie Python ECDSA
$ sudo pip install ecdsa
$ # execution du script
$ python ec-math.py
Secret:
38090835015954358862481132628887443905906204995912378278060168703580660294000
EC point:
(70048853531867179489857750497606966272382583471322935454624595540007269312627,
105262206478686743191060800263479589329920209527285803935736021686045542353380)
BTC public key: 029ade3effb0a67d5c8609850d797366af428f4a0d5194cb221d807770a1522873
```

Portemonnaie

Les portemonnaie sont des conteneurs de clefs privées, implémentés la plupart de temps sous forme de fichiers structurés ou de base de données très simples. Il est aussi possible d'utiliser un algorithme de génération de clef *déterministe*. Chaque clef privée est dérivée, grâce à une fonction à sens-unique, d'une clef précedente, créant ainsi une chaîne de clefs. Il suffit ainsi de connaitre la première clef (aussi appelée *clef maître*) pour recréer la chaîne et générer toutes les clefs. Dans ce chapitre nous examinerons différentes méthodes de génération de clef et comment elles sont utilisées par les portemonnaie.

TIP

Les portemonnaie bitcoin contiennent des clefs, pas de l'argent. Chaque utilisateur a son portemonnaie qui gère ses clefs: en fait, ce sont des "trousseau de clefs" qui gèrent des paires de clefs publiques/privées (voir Clefs publiques et privées). Les utilisateurs signent des transactions avec leurs clefs, prouvant ainsi qu'ils ont le droit de dépenser les sorties des transactions qui leur appartiennent (leurs "bitcoins"). Dans la blockchain, les bitcoins correspondent aux sorties des transactions, que l'on appelle souvent "vout" ou "txout".

Portemonnaie non-déterministes (aléatoires)

Dans les premiers clients bitcoin, les portemonnaie généraient leurs clefs privées de façon aléatoire. Ce type de portemonnaie est appelé portemonnaie non-déterministe de type 0. Par exemple, le client Bitcoin Core génère 100 clefs aléatoires lors de sa première utilisation, et génère ensuite d'autres clefs si nécessaire, afin que chaque clef ne soit utilisée qu'une seule fois. On surnomme ce type de portemonnaie "Just un Paquet de Clefs" (JBOK: Just a Bunch Of Keys" en anglais). Ils sont difficiles et fastidieux à gérer (sauvegarde, import de clefs, ...) et on les remplace de plus en plus par des portemonnaie déterministes. Le problème avec les clefs aléatoires est qu'il faut toutes les sauvegarder, sinon les fonds associés sont perdus. Il faut donc sauvegarder son portemonnaie très souvent, surtout si on génère beaucoup de clefs. C'est peu compatible avec la bonne pratique qui est de n'utiliser

chaque adresse que pour une seule transaction. Réutiliser les mêmes adresses crée des liens entre elles et les transactions qui les utilisent, ce qui pose un problème de confidentialité. Les portemonnaie non-déterministyes de type 0 sont un mauvais choix, surtout si l'on veut éviter de réutiliser les mêmes adresses (ce qui oblige à générer beaucoup de clefs, et sauvegarder son portemonnaie fréquemment). Bien que le client Bitcoin Core intègre un portemonnaie de type 0, son utilisation est déconseillée par les développeurs de Bitcoin Core. Portemonnaie non déterministe (aléatoire) de type 0: un ensemble de clefs générées aléatoirement illustre un portemonnaie non-déterministe, qui contient un ensemble de clefs aléatoires.

Portemonnaies déterministes

Les portemonnaie déterministes contiennent des clefs privées qui sont toutes dérivées d'une même "graine" (valeur d'initialisation) grâce à une fonction de hash à sens unique. Les clefs privées sont dérivées à partir de cette graine (qui est un nombre aléatoire) et d'autres données, telles que l'index de la clef ou son "code chaîne" ("chaincode", voir Portemonnaie déterministe hiérarchique (BIP0032/BIP0044)). La graine est suffisante pour retrouver toutes les clefs d'un portemonnaie déterministe, et on peut donc ne faire q'une seule sauvegarde, au moment de la création du portemonnaie. On n'a besoin que de cette graine pour exporter/importer les clefs, ce qui facilite les transferts entre différents portemonnaie.

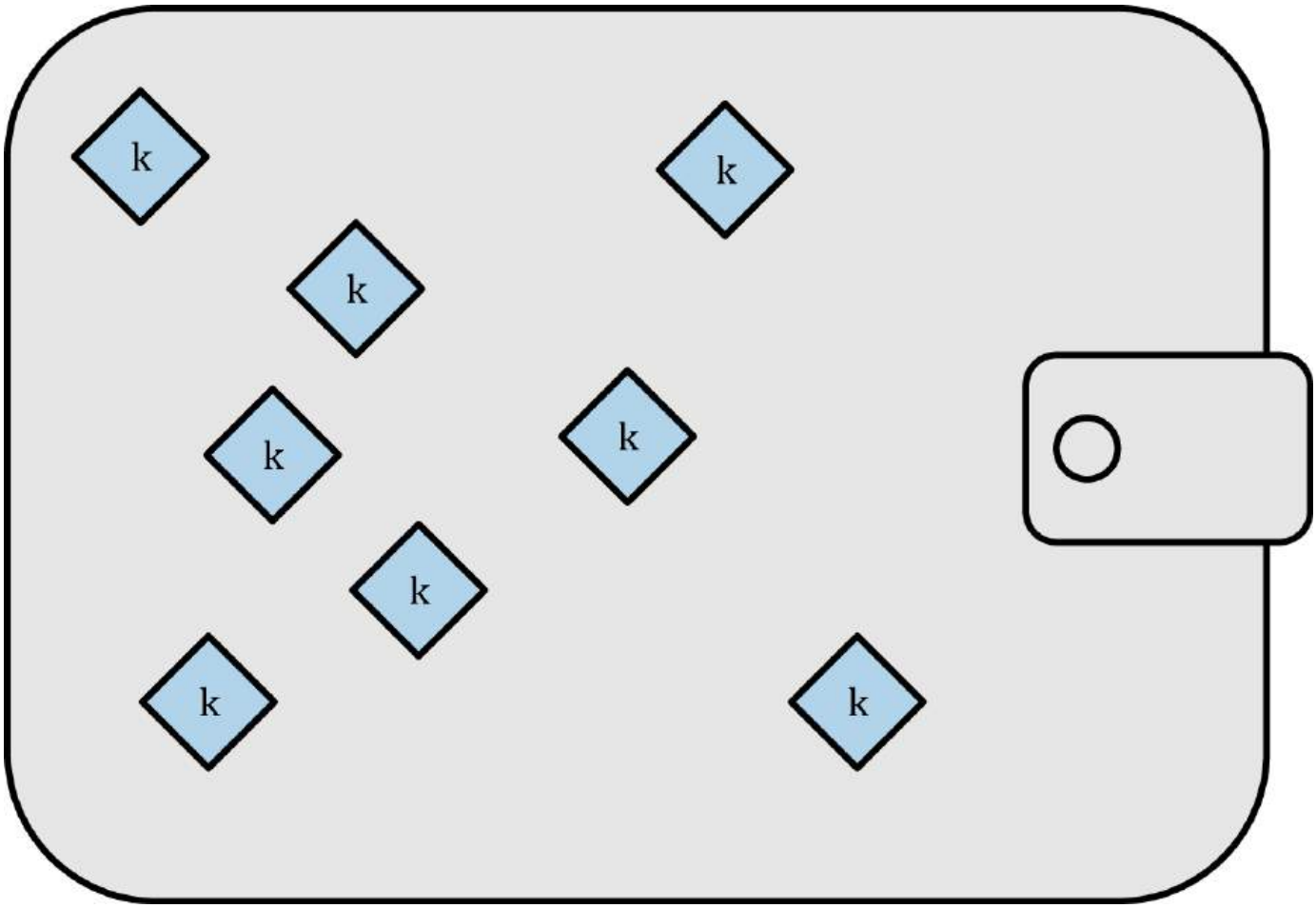


Figure 8. Portemonnaie non déterministe (aléatoire) de type 0: un ensemble de clefs générées aléatoirement

Code mnémonique

Pour représnter la graine utilisée par un portemonnaie déterministe, qui est un nombre aléatoire, on utilise un encodage basé sur une suite de mots appelé codes mnémoniques. Connaitre cette suite de mot est suffisant pour retrouver la graine, et donc toutes les clefs du portemonnaie. Lors de sa création, un portemonnaie déterministe va afficher une suite de 12 à 24 codes mnémoniques, que l'utilisateur va conserver comme sauvegarde du portemonnaie. Elle lui permettra de re-créer les même clefs, avec n'importe quel portemonnaie compatible. Les codes mnémoniques sont plus facile à lire et écrire que des suites de nombres aléatoires, ce qui les rend plus faciles à utiliser pour sauvegarder les portemonnaie.

Les codes mnémoniques sont définis dans le BIP 39 (voir[bip0039]), qui n'est pas encore définitevement accepté mais a encore le status de "proposition". Il existe un autre standard implémenté, avant la rédaction de BIP39, par le portemonnaie Electrum et basé sur d'autres codes mnémoniques. BIP0039 est utilisé par le portemonnaie Trezor et par d'autres implémentations, mais n'est pas compatible avec le portemonnaie Electrum.

Voici comment BIP0039 défini la création d'une graine et de son code mnémonique:

1. Générer un nombre aléatoire (entropie) de 128 à 256 bits.
2. Générer une checksum en prenant les premiers bits du hash SHA256 de ce nombre
3. Ajouter cette checksum à la fin du nombre
4. Découper le résultat en morceaux de 11 bits, et utiliser chaque morceau comme index dans un dictionnaire de 2048 mots prédéfinis
5. le résultat, qui est une suite de 12 à 24 mots, représente notre code mnémonique.

Codes mnémoniques: entropie et nombre de mots illustre le lien entre la taille du nombre aléatoire (entropie) et le nombre de mots du code mnémonique.

Table 4. Codes mnémoniques: entropie et nombre de mots

Entropie (bits)	Checksum (bits)	Entropie+checksum	Nombre de mots
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	8	264	24

Le code mnémonique représente un nombre de 128 à 256 bits, qui est utiliser pour dériver une nombre plus long (512 bits) grâce à la fonction de dérivation de clef PBKDF2. Le résultat est utilisé comme graine pour initialiser un portemonnaie déterministe et calculer toutes ses clefs.

Les tables <xref linkend="table_4-6" xrefstyle="select: labelnumber"/> et <xref linkend="table_4-7"

xrefstyle="select: labelnumber"/> illustre quelques exemples de clefs et de codes mnémoniques utilisés pour les générer

Table 5. Code mnémonique (entropie) sur 128 bits et graine correpondante

Entropie (128 bits)	0c1e24e5917779d297e14d45f14e1a1a
Code mnémonique (12 mots)	army van defense carry jealous true garbage claim echo media make crunch
Graine (512 bits)	3338a6d2ee71c7f28eb5b882159634cd46a898463e9d2d0980f8e80dfbba5b0fa0291e5fb888a599b44b93187be6ee3ab5fd3ead7dd646341b2cdb8d08d13bf7

Table 6. Code mnémonique sur 256 bits et graine correpondante

Entropie (256 bits)	2041546864449caff939d32d574753fe684d3c947c3346713dd8423e74abcf8c
Code mnémonique (24 mots)	cake apple borrow silk endorse fitness top denial coil riot stay wolf luggage oxygen faint major edit measure invite love trap field dilemma oblige
Graine (512 bits)	3972e432e99040f75ebe13a660110c3e29d131a2c808c7ee5f1631d0a977fcf473bee22fce540af281bf7cdeade0dd2c1c795bd02f1e4049e205a0158906c343

Portemonnaie déterministe hiérarchique (BIP0032/BIP0044)

Les portemonnaie déterministes ont été créés pour faciliter la dérivation de multiples clef privées à partir d'une "graine" unique. Les portemonnaies déterministes les plus avancés sont les portemonnaie *déterministes hiérarchiques* souvent appelés *portemonnaie HD* (HD pour Hierarchical Deterministic en anglais), définis par le standard BIP0032. Les portemonnaie déterministes hiérarchiques permettent de gérer des arbres de clefs: une clef parent peut générer une série de clefs filles, qui peuvent elles-mêmes générer une série de clefs filles, et ainsi de suite sans limite de profondeur. Cette structure en arbre est illustrée ici: Portemonnaie déterministe hiérarchique de type 2: un arbre de clefs généré à partir d'une graine unique..

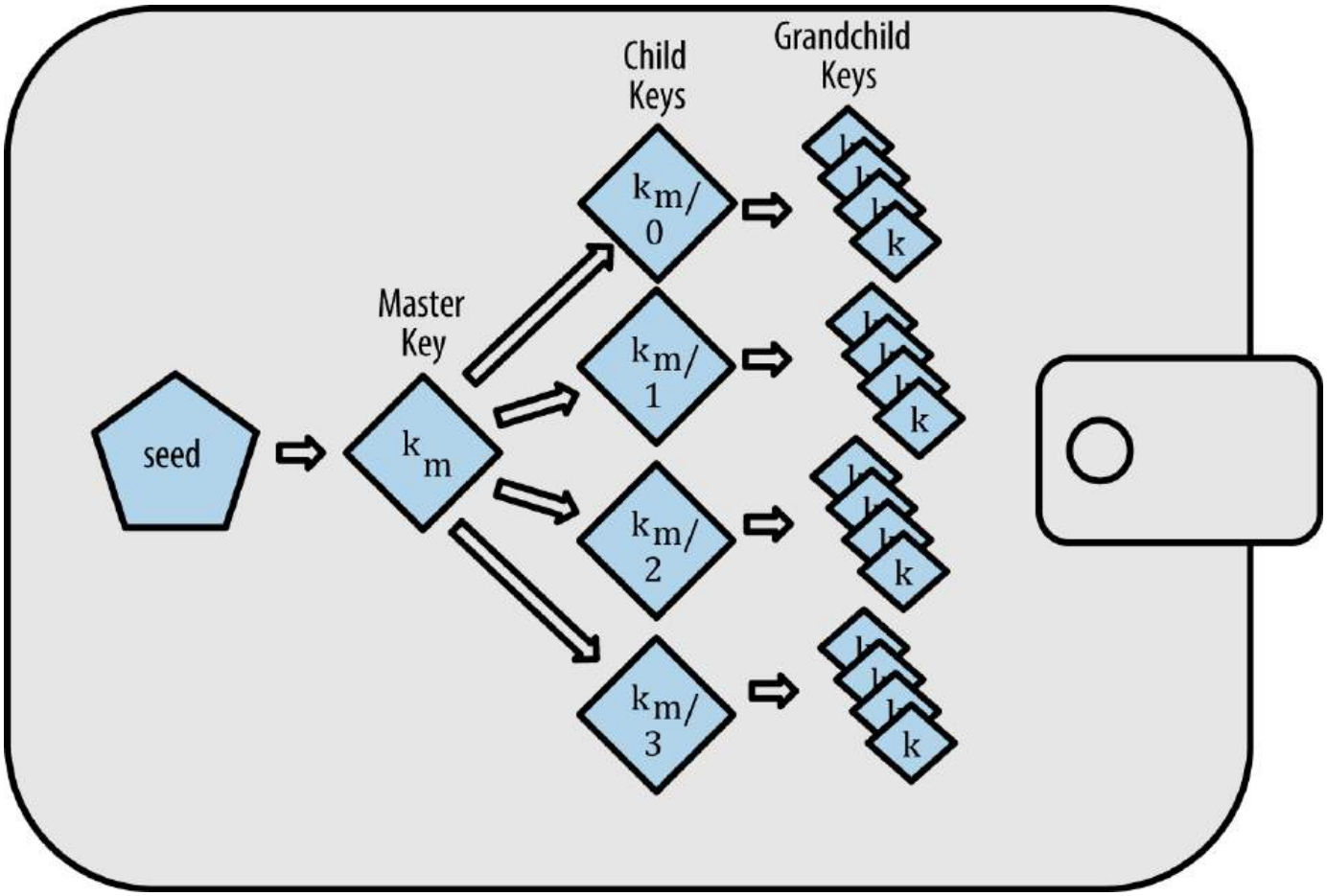


Figure 9. Portemonnaie déterministe hiérarchique de type 2: un arbre de clefs généré à partir d'une graine unique.

TIP	Si vous développez un portemonnaie bitcoin, faites un portemonnaie HD et respectez les standard BIP0032 et BIP0044.

Les portemonnaie HD offrent 2 avantages significatifs par rapport aux portemonnaie aléatoire. Premierement, les clefs sont gérées sous forme d'arbre, dont la structure peut être modelée sur la structure opérationnelle d'une entreprise: une branche peut être utilisée pour recevoir des paiements, une autre pour gérer le change produit par l'envoi de fonds. On peut aussi faire correspondre les branches aux filialies, aux différents departements ou entités fonctionnelles, ou à différentes catégories comptables.

Le deuxième avantage des portemonnaie HD est que l'on peut gérer des arbres de clefs publiques sans connaitre les clefs privées correspondantes. On peut donc les utiliser pour recevoir des paiements sur des serveurs non sécurisés, en utilisant une nouvelle clef publique pour chaque transaction. Les clefs publiques n'ont pas besoin d'être préchargées ou calculées à l'avance, mais le serveur ne connait pas les clefs privées qui servent à dépenser les fonds reçus.

création d'un portemonnaie HD à partir d'une graine

Les portemonnaie HD sont initialisés avec une *graine racine* unique, qui est un nombre aléatoire de 128, 256 ou 512 bits. Tout le reste est dérivé de cette graine, ce qui permet de le recréer entièrement

dans un autre portemonnaie compatible HD. Il est ainsi très facile de sauvegarder, restaurer, exporter et importer des portemonnaie HD, même s'ils contiennent des milliers ou des millions de clefs, simplement en transférant la graine racine. Cette graine est souvent représentée sous la forme d'une *liste de mots mnémoniques*, comme expliqué dans la section précédente Code mnémonique, ce qui la rend plus facile à recopier.

La création d'une clef maître et d'un code chaîne pour un portemonnaie HD est illustré ici: Création d'une clef maître et d'un code chaîne à partir d'une graine racine.

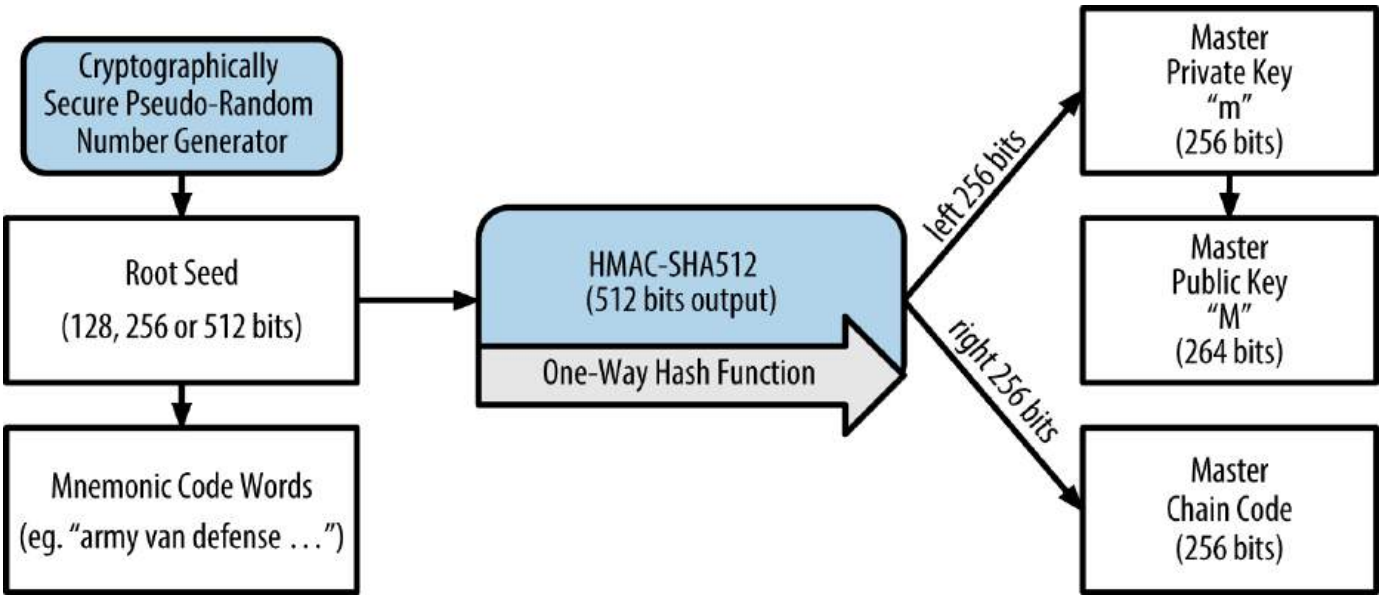


Figure 10. Création d'une clef maître et d'un code chaîne à partir d'une graine racine

La graine racine est hashée avec l'algorithme HMAC-SHA512, et le résultat est utilisé pour créer une *clef maître* (m) et un *code chaîne maître*. On utilise ensuite la multiplication en courbes elliptiques pour générer une clef publique maître (M) à partir de (m), comme vu précédemment: M = m * G. Le code chaîne sert de source d'entropie pour les fonctions de dérivation de clefs filles à partir de clefs parents, comme nous le verrons dans la prochaine section.

Dérivation des clefs privées filles

Les portemonnaie hiérarchiques déterministes utilisent une *fonction de dérivation de clef fille* (CKD: Child Key Derivation en anglais) pour dériver les clefs filles des clefs parents.

La fonction de dérivation de clef fille est basé sur une fonction de hash à sens unique qui combine:

- Une clef parent publique (clef ECDSA compressée) ou privée
- Une graine appelée code chaîne (256 bits)
- Un numéro d'index (entier sur 32 bits)

Le code chaîne permet d'introduire des données pseudo-aléatoires dans le processus de dérivation, afin que l'index ne soit pas suffisant pour dériver d'autres clefs. Ainsi, avoir une clef fille ne permet pas de trouver ses soeurs, à moins de connaître le code chaîne. Le code chaîne initial (à la racine de

l'arbre) est créé à partir de données aléatoires, et les codes chaines utilisés pour les clefs filles sont dérivés du code chaine parent.

Ces trois éléments sont combinés et hashés pour générer les clefs filles de la façon suivante:

La clef publique parent, le code chaîne et l'index sont hashés avec l'algorithme HMAC-SHA512 qui produit un hash sur 512 bits. Ce hash est coupé en 2 moitiés de 256 bits. La moitié de droite deviendra le code chaine de la clef fille, la moitié de gauche et l'index sont ajoutés à la clef privée parent pour donner la clef privée fille. Dérivation d'une clef privée fille à partir d'une clef privée parent, illustre ce processus, avec l'index 0 pour produire la "0ieme" (première) clef fille de la clef parent.

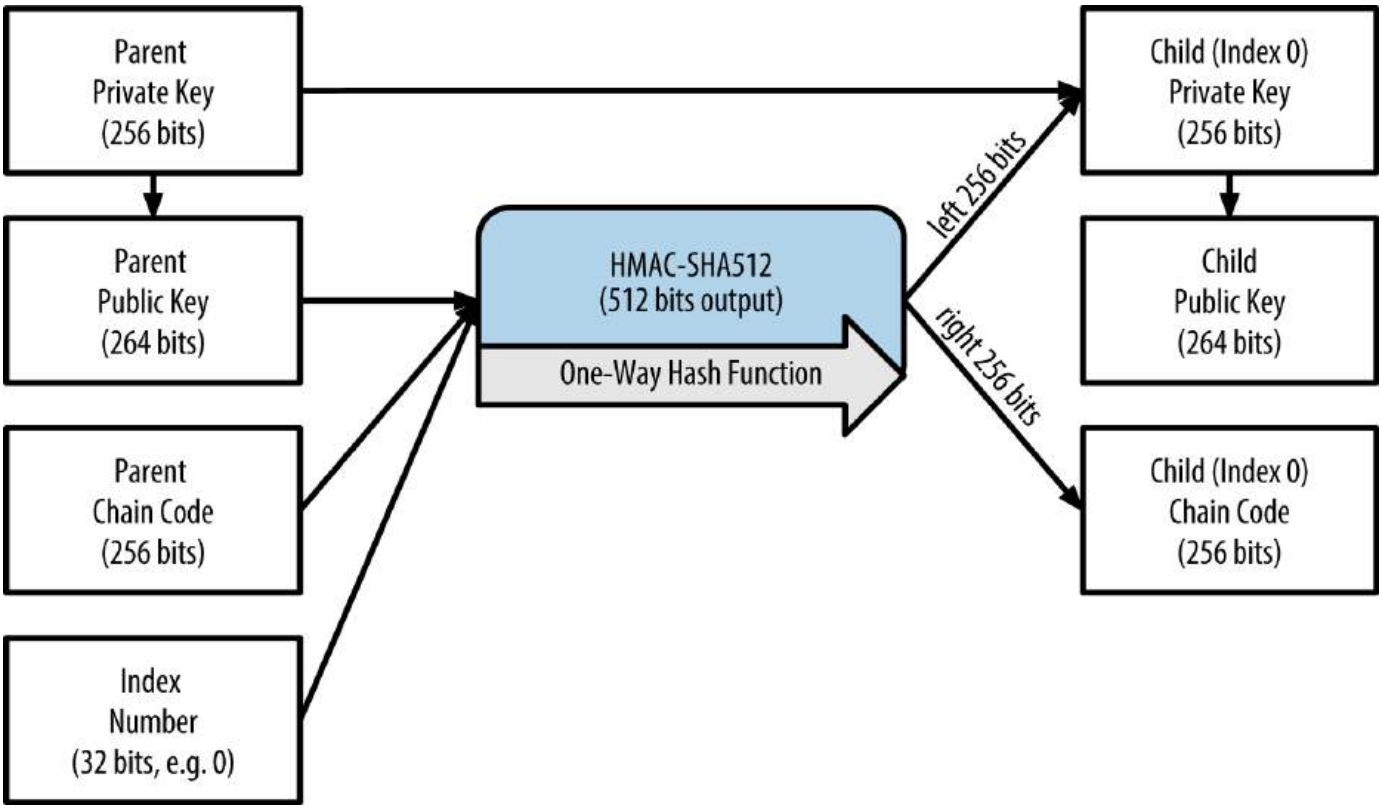


Figure 11. Dérivation d'une clef privée fille à partir d'une clef privée parent

En incrémentant l'index, on peut créer des séries de clefs filles: fille 0, fille 1, fille 2, etc. Chaque clef parent peut avoir 2 milliards de clefs filles.

On peut répéter ce processus à l'infini: à chaque niveau de l'arbre, chaque clef fille devenant à son tour une clef parent et peut générer d'autres clefs filles, et ainsi de suite.

Utilisation des clefs filles

Il est impossible de distinguer les clefs privées filles de clefs non-déterministes (aléatoires). Comme la fonction de dérivation est une fonction à sens unique, on ne peut pas retrouver la clef parent à partir de la clef fille. On ne peut pas non plus retrouver ses soeurs. Si vous connaissez la nième clef fille, vous ne pouvez pas retrouver la clef fille n-1 ou n+1 ou n'importe quelle autre. Pour calculer les clefs filles il est indispensable de connaître la clef parent et le code chaine. Sans celui-ci, on ne pourra pas non plus calculer les petits-enfants du parent: pour calculer une nouvelle branche il faut la clef privée fille et

son code chaine.

Alors que peut-on faire avec la clef fille seule ? On peut créer une clef publique et une adresse bitcoin. On peut alors utiliser la clef pour signer des transactions afin de dépenser les fonds reçus à cette adresse.

TIP	On ne peut pas distinguer des clefs privées dérivées, les clefs publiques associées et les adresses bitcoin correspondantes de clefs et adresses générées aléatoirement. Seule le portemonnaie HD qui les a créées connait l'arbre de clefs dont elles font partie. Une fois créées, elles fonctionnent exactement comme des clefs "normales".

Clefs étendues

Comme vu précédemment, la fonction de dérivation de clefs permet de créer des clefs filles à partir de n'importe quelle position dans l'arbre de clefs en combinant trois éléments: une clef, un code chaine, et l'index de la clef que l'on veut dériver. Les deux éléments essentiels sont la clef et le code chaine. L'ensemble clef + code chaine est appelé une *clef étendue*. Mais on pourrait aussi l'appeler "clef extensible" car on peut l'utiliser pour dériver de nouvelles clefs.

Les clefs étendues sont stockées et représentées sous la forme d'une chaine de 512 bits: 256 bits pour la clef, suivis de 256 bits pour le code chaine. Il y a 2 sortes de clefs étendues: les clefs privées étendues (composée d'une clef privée et d'un code chaine) que l'on peut utiliser pour dériver des clefs privées filles (et ensuite les clefs publiques filles associées), et les clefs publiques étendues (composées d'une clef publique et d'une code chaine) que l'on peut utiliser pour dériver des clefs public filles, comme illustré ici: Generation d'une clef publique.

On peut voir une clef étendue comme la racine d'une branche dans l'arbre de clefs d'un portemonnaie HD. A partir de cette racine on peut dériver le reste de la branche. Avec une clef privée étendue on peut calculer une branche complète, alors qu'avec une clef publique étendue on ne peut calculer qu'une branche de clefs publiques.

TIP	Une clef étendue est composée d'une clef privée ou publique et d'un code chaine. On peut l'utiliser pour calculer une branche de l'arbre de clefs. Si on donne cette clef étendue, on donne l'accès à l'ensemble de cette branche.

Les clefs étendues sont encodées au format Base58Check pour être facilement exportées/importées depuis différents portemonnaie compatibles BIP0032. On utilise un préfixe version spécifique, ce qui fait que le résultat commence par "xprv" et "xpub" afin de rendre les clefs facilement identifiables. Les clefs étendues font 512 bits, et elles sont encodées avec des informations supplémentaires (profondeur, identifiant du parent, index…) ce qui donne un encodage Base58Check beaucoup plus long que ce que nous avons vu jusqu'à présent.

Voici un exemple d'encodage d'une clef privée étendue au format Base58Check:

```
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6GoNMK
Uga5biW6Hx4tws2six3b9c
```

Voic la clef publique étendue correspondante au format Base58Check:

```
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunSDMst
weyLXhRgPxdp14sk9tJPW9
```

Dérivation des clef publiques filles

Comme évoqué précédemment, un des points forts des portemonnaie hiérarchiques déterministes est la possibilité de dériver des clefs publiques filles depuis une clef publique parent, *sans* connaitre les clefs privées. On a ainsi deux façons de dériver les clefs publiques filles: depuis les clefs privées filles, ou depuis la clef publique parent.

On peut ainsi utiliser une clef publique étendue pour dériver toutes les clefs *publiques* (et seulement les clefs publiques) de la branche dont elle est la racine.

On peut ainsi déployer des clefs de façon extrêmement sécurisée: sur un serveur (ou une application) on ne déploie que la clef publique étendue, mais aucune clef privée. On peut ainsi créer une infinité de clefs publiques et d'adresses bitcoin, mais sans pouvoir dépenser les fonds associés. En parallèle, on déploie la clef privée étendue sur un autre serveur sécurisé qui pourra dériver les clefs privées nécessaires pour signer les transactions et dépenser les fonds.

Ce type de déploiement est souvent utilisé dans les solutions de type eCommerce: on déploie la clef publique étendue sur le serveur web de l'application. En utilisant la fonction de dérivation de clef publique, le serveur web peut dériver une nouvelle adresse bitcoin pour chaque transaction (par exemple pour le panier de l'utilisateur). Il n'y a aucune clef privée sur le serveur web, donc aucun risque de vol ou piratage. Sans les portemonnaie HD, il faudrait d'abord générer des milliers d'adresses bitcoin sur un autre serveur et ensuite les pré-charger sur le serveur web: ce serait fastidieux et il faudrait sans cesse vérifier que le serveur n'a pas épuisé toutes ses clefs.

Ce scenario est aussi utilisé pour le stockage de bitcoin offline ("cold storage") et pas les portemonnaie matériels. La clef privée étendue est sauvegardée sur papier ou dans un portemonnaie matériel (comme le portemonnaie matériel Trezor), et la clef publique étendue peut être utilisée en mode connecté pour générer autant d'adresses bitcoin que l'on veut. Pour dépenser les fonds, on utilise la clef privée étendue soit via un client en mode déconnecté soir en faisant signer les transactions par le portemonnaie matériel. La dérivation de clefs publiques filles à partir d'une clef publique parent est illustrée ici: Dérivation d'une clef publique fille à partir d'une clef publique étendue parent

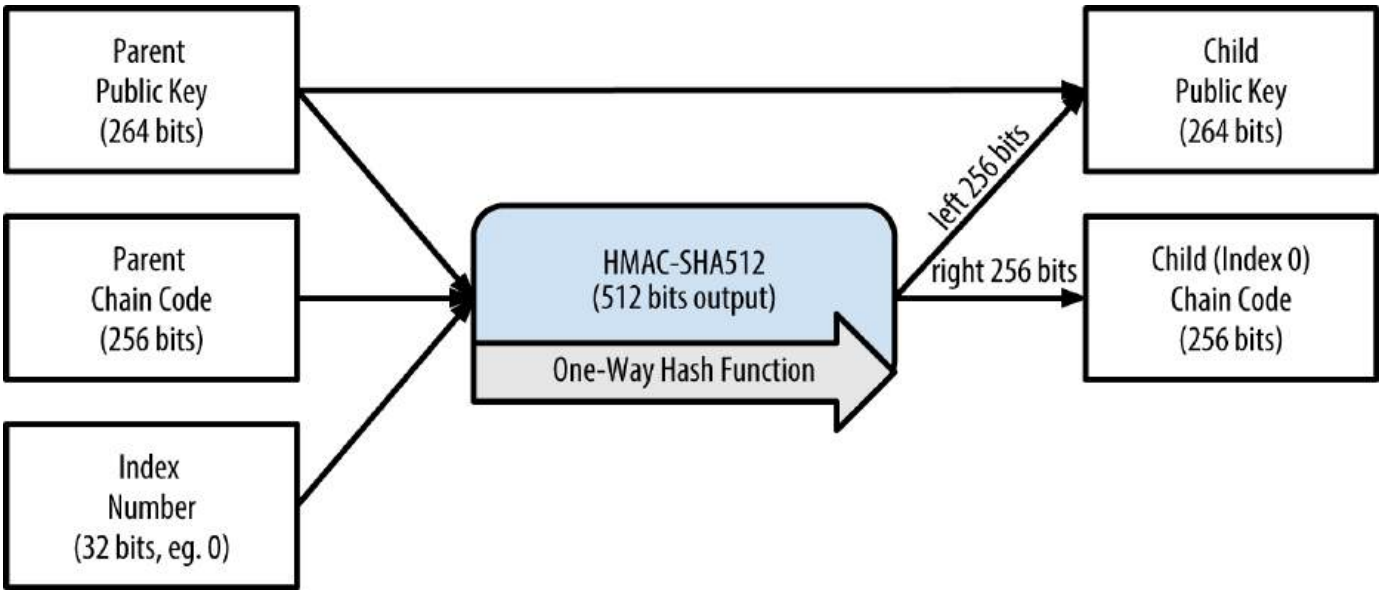


Figure 12. Dérivation d'une clef publique fille à partir d'une clef publique étendue parent

Dérivation de clefs filles durcies

Pouvoir dériver un arbre de clefs publiques à partir d'une clef publique étendue est très utile, mais il y a un problème de sécurité. Une clef publique étendue ne permet pas de calculer les clefs privées filles, mais elle contient le code chaine: si on connait une clef privée fille, on peut l'utiliser pour retrouver les autres. Avec une seule clef privée fille, et le code chaine de la clef parent, on peut retrouver toutes les clefs filles. Et pire encore: on peut aussi retrouver la clef privée parent.

Pour remédier à ce problème, les portemonnaie HD ont une autre fonctione de dérivation, appelée fonction de dérivation *durcie*, qui "casse" le lien entre la clef publique parent et le code chaine de la clef fille. Pour calculer le code chaine, la fonction de dérivation durcie utilise la clef privée parent (au lieu de la clef publique parent). Cela crée une "barrière" au milieu de la chaine parent/enfant, car ce code chaine ne peut pas être utilisé pour compromettre une clef parent ou la soeur d'une clef privée. La fonction de dérivation durcie est quasiment identique à la fonction de dérivation de clef privée normale, la seule différence est que l'on hash la clef privée parent au lieu de la clef publique parent, comme illustré ici: Dérivation d'une clef durcie; on n'utilise plus la clef publique parent.

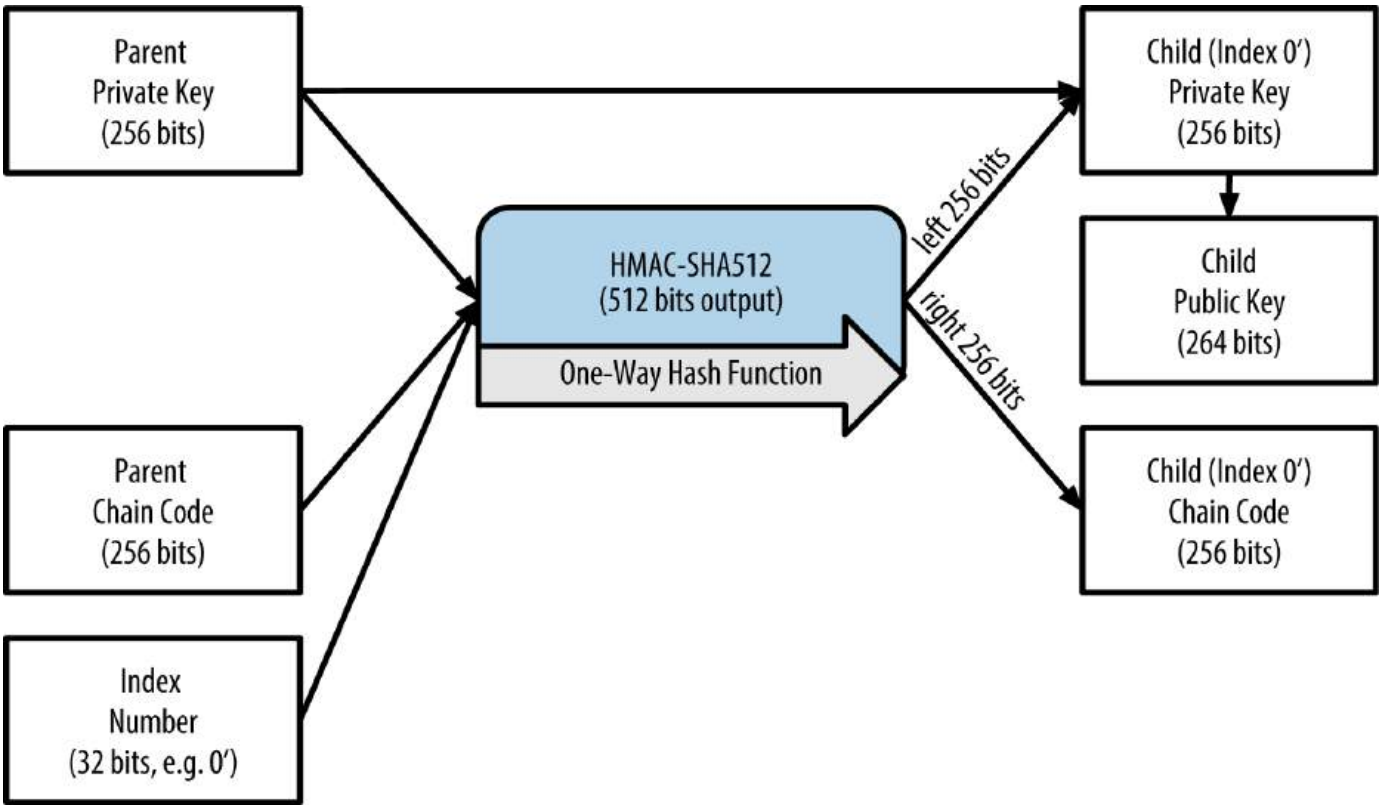


Figure 13. Dérivation d'une clef durcie; on n'utilise plus la clef publique parent

La clef privée et le code chaine calculés par la fonction de dérivation durcie sont complétement différent de ceux obtenus avec la fonction de dérivation normale, et on peut les utiliser pour générer des clefs publiques étendues qui ne sont pas vulnérables, car leur code chaine ne peut pas être utilisé pour retrouver des clefs privées. La fonction de dérivation durcie crée un "trou" dans l'arbre de clefs juste au-dessus des clefs publiques étendues.

Pour faire simple: si vous voulez utilisez une clef publique étendue pour dériver un arbre de clefs publiques sans prendre le risque d'exposer vos clefs par une fuite du code chaine, vous devriez la dériver d'une clef durcie et pas d'une clef normale. La bonne pratique est la suivante: le premier niveau de clefs dérivées d'une clef maitre est toujours calculé avec la fonction de dérivation durcie, pour éviter de compromettre la clef maitre.

Numérotation des clefs normales et durcies

L'index, ou numéro de clef, utilisé par la fonction de dérivation est un entier sur 32 bits. Pour pouvoir distinguer facilement les clefs normales des clefs durcies, la plage des index est coupée en 2: de 0 à 2^{31}–1 (0x0 to 0x7FFFFFFF) pour les index normaux, et de 2^{31} and 2^{32}–1 (0x80000000 to 0xFFFFFFFF) pour les index durcis. Donc, si l'index d'une clef est inférieur ou égal à 2^{31} c'est une clef normale, s'il est supérieur à 2^{31} c'est une clef durcie.

Pour faciliter la représentation des index on utilise la notation suivante: pour les index normaux on compte à partir de 0, pour les index durcis on compte à partir de 0x80000000 et on ajoute le signe "prime". Pour la première clef fille normale l’index s’écrit donc 0, et pour la première clef durcie (index = 0x80000000) on écrit <markup>0'</markup>. De même, pour la deuxième clef

durcie (index = 0x80000001) on écrit 1', et ainsi de suite. Quand voyez i' pour un index de clef, le vrai numéro est 2³¹+i.

Identifiants (chemins) des clefs des portemonnaies HD

Pour identifier les clefs dans un portemonnaie HD, on utilise une représentation de leur "chemin" dans l'arbre des clefs, avec le caractère "/" utilisé comme séparateur (voir exemples de chemins de clefs HD). Pour les clefs privées on représente la clef privée maitre par un "m", et pour les clefs publiques on représente la clef publique maitre par un "M". Par exemple, la première clef privée dérivée de la clef privée maitre est m/0, et la première clef publique est M/0. La deuxième clef fille de la première clef fille est m/0/1, et ainsi de suite.

On suit le chemin en partant de la droite, jusqu'à remonter à la clef maitre. Par exemple, m/x/y/z représente la z-ième fille de la clef m/x/y, qui est la y-ième fille de m/x, qui est la x-ième fille de m.

Table 7. exemples de chemins de clefs HD

chemin HD	**description de la clef**
m/0	Première (0) clef privée fille de la clef privée maitre (m)
m/0/0	Première petite-fille, fille de la première clef privée fille (m/0)
m/0'/0	Première clef fille normale de la première clef fille *durcie* (m/0')
m/1/0	Première clef privée fille de la deuxième fille (m/1)
M/23/17/0/0	Première clef publique fille de la première fille de la 18ème fille de la 24ème fille

Parcourir l'arbre de clefs HD

Les arbres de clefs HD offrent beaucoup de possibilités. Chaque clef étendue parent peut générer 4 milliards de clef filles: 2 milliards de clefs normales et 2 milliards de clef durcies. Chaque clef fille peut à son tour générer 4 milliards de clef filles, et ainsi de suite. L'arbre peut être aussi profond que l'on veut: on peut générer une infinité de clefs. Il peut alors devenir difficile de s'y retrouver dans l'arbre de clef, et en particulier il est délicat de l'exporter vers un autre portemonnaie, car il y a un infinité de possibilités pour organiser l'arbre en branches et sous-branches.

Pour mieux gérer cette complexité, il existe 2 BIP qui proposent de standardiser l'arbre de clefs des portemonnaie HD. BIP0043 propose d'utiliser l'index de la première clef fille durcie d'une branche comme indicateur de "l'objet" de cette branche. En se conformant à BIP0043, un portemonnaie HD ne doit avoir qu'une seule branche de niveau 1, dont la structure est définie par cet objet. Par exemple, si un portemonnaie HD contient uniquement la branche m/i'/, l'objet de cette branche est identifié par l'index "i".

BIP0044 est une extension de BIP0043 qui propose d'utiliser comme "objet" l'index 44. Tous les portemonnaies HD conformes à BIP0044 n'ont qu'une seule branche: m/44'/.

BIP004 défini une structure d'arbre sur 5 niveaux:

m / objet' / type_de_monnaie' / compte' / change / index_addresse

Le premier niveau "objet" est toujours 44'. Le deuxième niveau "type_de_monnaie" identifie le type de cryptomonnaie ce qui permet de gérer plusieurs crytomonnaies avec le même portemonnaie HD, chaque cryptomonnaie ayant sa propre branche dédiée. Pour l’instant, 3 valeurs possibles ont été définies: m/44'/0' pour Buitcoin, <markup>m/44'/1'</markup> pour Bitcoin Testne; et <markup>m/44'/2'</markup> pour Litecoin.

Le troisième niveau, "compte", permet aux utilisateurs de diviser leur portemonnaie en différents comptes, pour des raisons comptables ou organisationnelles. Par exemple, un portemonnaie HD pourrait gérer les 2 comptes suivants: <markup>m/44'/0'/0'</markup> et <markup>m/44'/0'/1'</markup>. Chaque compte correspond à la racine de sa propre sous-branche de clef.

Le quatrième niveau, "change", est séparé en 2 branches, une pour recevoir des fonds et l'autre pour gérer les adresses de change. On remarquera qu'on utilise ici des clefs normales, alors qu'aux niveaux précédents on utilisait des clefs durcies. Ainsi, on peut exporter les clefs publiques étendues correspondant à ces 2 branches vers des environnements non-sécurisés. Les index des adresses utilisées par le portemonnaie sont dérivés de ces 2 branches et forment ainsi le cinquième niveau de l'arbre, "index_adresse".

Table 8. Exemples d'arbres BIP0044

chemin HD	**description de la clef**
M/44'/0'/0'/0/2	La troisième clef pour recevoir des fonds, pour le premier compte bitcoin
M/44'/0'/3'/1/14	La quinzième addresse de change du quatrième compte bitcoin
m/44'/2'/0'/0/1	La deuxième clef privée du premier compte Litecoin, utilisée pour signer des transactions

Explorer les portemonnaie HD avec Bitcoin Explorer

Avec l'outil en ligne de commande Bitcoin Explorer présenté au chapitre [ch03_bitcoin_client], on peut générer et dériver des clefs déterministes BIP0032, et les afficher sous différents formats:

```
$ bx seed | bx hd-new > m # creation d'une nouvelle clef privée maitre sauvegardée
dans le fichier "m"
$ cat m # affichage de la clef privée étendue maitre
xprv9s21ZrQH143K38iQ9Y5p6qoB8C75TE71NfpyQPdfGvzghDt39DHPFpovvtWZaRgY5uPwV7RpEgHs7cvdg
fiSjLjjbuGKGcjRyU7RGGSS8Xa
$ cat m | bx hd-public # génération de la clef publique étendue M/0
xpub67xpozcx8pe95XVuZLHXZeG6XWXHpGq6Qv5cmNfi7cS5mtjJ2tgypeQbBs2UAR6KECeeMVKZBPLrtJunS
DMstweyLXhRgPxdp14sk9tJPW9
$ cat m | bx hd-private # génération de la clef privée étendue m/0
xprv9tyUQV64JT5qs3RSTJkXCWKMyUgoQp7F3hA1xzG6ZGu6u6Q9VMNjGr67Lctvy5P8oyaYAL9CAWrUE9i6G
oNMKUga5biW6Hx4tws2six3b9c
$ cat m | bx hd-private | bx hd-to-wif # affichage de la clef privée m/0 au format
WIF
L1pbvV86crAGoDzqmgY85xURkz3c435Z9nirMt52UbnGjYMzKBUN
$ cat m | bx hd-public | bx hd-to-address # affichage de l'adresse bitcoin
correspondant à M/0
1CHCnCjgMNb6digimckNQ6TBVcTWBAmPHK
$ cat m | bx hd-private | bx hd-private --index 12 --hard | bx hd-private --index 4 #
génération de m/0/12'/4
xprv9yL8ndfdPVeDWJenF18oiHguRUj8jHmVrqqD97YQHeTcR3LCeh53q5PXPkLsy2kRaqgwoS6YZBLatRZRy
UeAkRPe1kLR1P6Mn7jUrXFquUt
```

Clefs et adresses: concepts avancés

Dans les prochaines sections, nous éudierons plusieurs utilisations avancées des clefs et adresses: clef privées chiffrées, scripts et adresses multisignatures, adresses personnalisées, et portemonnaie papier.

Clefs privées chiffrées (BIP0038)

Il faut absolument que les clefs privées restent secrètes. Ce besoin de *confidentialité* entre en conflit avec un autre besoin tout aussi important: il faut aussi que les clefs privées restent *disponibles*. En effet faire en sorte que les clefs restent secrètes est nettement plus difficile quand on doit aussi les sauvegarder pour éviter de les perdre. Certains portemonnaie proposent des sauvegardes chiffrées qui contiennent les clefs privées, mais il faut conserver ces sauvegardes, et on peut parfois avoir besoin de transférer les clefs vers un autre portemonnaie - lors d'une mise à jour par exemple. On peut aussi sauvegarder les clefs privées sur papier (voir Portemoinnaie papier), ou sur un lecteur externe comme une clef usb. Mais que se passe-t-il si la sauvegarde est perdue ou volée ? Pour adresser ces besoins contradictoires, BIP0038 (voir [bip0038]) propose un standard pour le chiffrer les clefs privées de façon pratique, portable, et utilisable par de nombreux portemonnaie et clients.

BIP0038 propose un standard pour chiffrer les clefs privées avec une phrase de passe et encoder le résultat au format Base58Check, afin de le rendre facile à sauvegarder et exporter vers d'autres portemonnaie sans exposer la clef privée. La clef est chiffrée avec l'algorithme Advanced Encryption

Standard (AES), standardisé par le National Institute of Standards and Technology (NIST) et utilisé pour de nombreuses applications militaires et commerciales.

Le processus de chiffrement proposé par BIP0038 prend en entrée une clef privée, généralement encodée au format WIF (une chaine de caractère au format Base58Check, commençant par "5"), et une phrase de passe (c'est à dire un long mot de passe) qui est souvent une suite de mot ou un chaine de caractères complexe. Le résultat est encodé au format Base58Check et commence par 6P. Si vous voyez une clef qui commence par 6P, cela veut dire qu'elle est chiffrée et que vous aurez besoin de la phrase de passe pour la déchiffrer et la convertir au format WIF (qui commence par 5) pour l'importer dans un portemonnaie. La plupart des portemonnaie reconnaissent les clefs chiffrées au format BIP0038 et demanderont le phrase de passe lors de l'importation. D'autres applications, comme le très utile client web Bit Address (onglet Wallet Details) peuvent être utilisées pour décrypter les clefs BIP0038.

L'utilisation la plus courante de BIP0038 est la création de portemonnaie papiers. Si l'utilisateur choisit une bonne phrase de passe, créer un portemonnaie papier contenant une clef privée chiffrée au format BIP0038 est un moyen incroyablement sécurisé de stocker des bitcoin hors connection ("cold storage").

Utilisez bitaddress.org pour essayer de déchiffrer les clefs de la table Exemples de clefs chiffrées au format BIP0038.

Table 9. Exemples de clefs chiffrées au format BIP0038

Clef privée (WIF)	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn
Phrase de passe	MyTestPassphrase
Clef chiffrée (BIP0038)	6PRTHL6mWa48xSopbU1cKrVjpKbBZxcLRRCdctLJ3z5yxE87MobKoXdTsJ

Adresses Pay-to-Script Hash (P2SH) et Multi-Sig

Comme nous l'avons déja vu, les adresses bitcoin usuelles sont dérivées d'une clef publique (elle même dérivée d'une clef privée) et commencent par "1". N'importe qui peut envoyer des bitcoins vers une adresse commençant par "1" mais il faut la clef privée et le hash de la clef publique pour les dépenser.

Les adresses bitcoin commençant par "3" sont des adresses pay-to-script hash (P2SH), parfois appelées à tort adresses multi-signatures ou multi-sig. Ces adresses ne représentent pas le hash d'une clef publique, mais le hash d'un script. Elles ont été proposées par le BIP0016 (voir [bip0016]) en janvier 2012, et rapidement adoptées car elles offrent beaucoup plus de possibilités que les adresses classiques pay-to-public-key-hash (P2PKH), qui commencent par "1". Pour utiliser les fonds envoyés vers une adresses P2SH il ne suffit pas de présenter une signature et un hash de clef publique, il faut aussi résoudre les contraintes définies dans le script qui a été hashé pour créeer l'adresse.

Les adresses "pay-to-script hash" sont créées à partir des scripts des transactions, qui définissent qui peut dépenser les sorties de ces transactions (voir [p2sh]). Pour encoder une adresse "pay-to-script hash" on utilise le même double hash que pour les adresses, sauf que c'est le script que l'on hash et pas

la clef publique:

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

Le résultat est encodé au format Base58Check en utilisant le préfixe 5, ce qui donne des adresses commençant par 3. Par exemple, on peut générer l'adresse P2SH 3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM avec l'outil Bitcoin Explorer et les commandes script-encode, sha256, ripemd160, et base58check-encode (voir[libbitcoin]) de la façon suivante:

```
$ echo dup hash160 [ 89abcdefabbaabbaabbaabbaabbaabbaabbaabba ] equalverify checksig >
script
$ bx script-encode < script | bx sha256 | bx ripemd160 | bx base58check-encode --version
5
3F6i6kwkevjR7AsAd4te2YB2zZyASEm1HM
```

TIP Les transactions P2SH ne sont pas forcément des transactions multi-signature, même si c'est souvent le cas. Le script qui est hashé peut implémenter d'autres types de transactions.

Adresses multi-signature et P2SH

Aujourd'hui, les adresses P2SH représentent le plus souvent des scripts multi-signature. Comme leur nom l'indique, pour résoudre ces scripts et donc dépenser les fonds il faut plus d'une signature: il faut présenter M signatures (M est appelé le "seuil") qui soient valables pour M clefs parmi N clefs possibles, avec M plus petit ou égal à N. On parle aussi de multi-signature M-sur-N (M-of-N). Par exemple, notre ami Bob le propriétaire du café de [ch01_intro_what_is_bitcoin] pourrait utiliser des adresses multi-signature 1-sur-2, les 2 clefs étant la sienne et celle de sa femme. De cette façon, sa femme ou lui peuvent dépenser les fonds, un peu comme un compte bancaire joint ou les 2 époux peuvent signer des chèques. De la même façon, Gopesh, le web designer que Bob a engagé pour faire son site web, pourrait utiliser des adresses multi-signature 2-sur-3, pour être certain que les fonds ne peuvent être dépensés que si au moins 2 associés ont signés la transaction.

Au chapitre [transactions] nous étudierons comment créer des transactions qui dépensent les fonds envoyés vers des adresses P2SH (et multi-signature).

Adresses personnalisées

Les adresses personnalisées sont des adresses bitcoin valides mais qui contiennent des mots ou des messages. Par exemple, 1LoveBPzzD72PUXLzCkYAtGFYmK5vYNR33 est une adresse valide qui commence par le mot "Love". Pour générer une adresse personnalisée, il faut essayer des milliards de clefs privées, jusqu'à ce que l'adresse obtenue contienne le mot recherché. Il existe quelques optimisations possible mais globalement, on tire une clef privée au sort, on génère la clef publique correspondante, puis l'adresse à partir de la clef publique, et on vérifie si l'adresse correspondant à la

personnalisation souhaitée, en répétant l'opération plusieurs milliards de fois si nécessaire.

Une fois que l'on a trouvé la clef privée qui permet de générer l'adresse personnalisée voulue, on utilise cette adresse comme n'importe quelle autre adresse bitcoin. Les adresses personnalisées offrent exactement la même sécurité que les autres adresses, et se basent sur les mêmes algorithmes (ECC et SHA). Il n'est pas plus facile de retrouver la clef privée que pour une adresse normale.

Au chapitre [ch01_intro_what_is_bitcoin], nous avions fait la connaissance d'Eugenia, qui dirige une association caritative aux Philippines. Supposons qu'elle souhaite organiser une collecte en bitcoin, et veuille utiliser une adresse personnalisée commençant par "1Kids" pour faire la promotion de sa collecte. Regardons comment générer cette adresse, et l'impact sur la sécurité de la collecte.

Génération d'une adresse personnalisée

Il est important de garder à l'esprit qu'une adresse bitcoin n'est qu'un nombre représenté en base 58. Chercher une adresse commençant par "1Kids" revient à chercher une adresse comprise entre 1Kids11111111111111111111111111111 et 1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzz. Il y a environ 58^{29} (environ$1.4 * 10^{51}$) adresses entre ces 2 valeurs, qui commencent toutes par "1Kids". La table Les plages d'adresses commençant par "1Kids" décrit les plages d'adresses commençant par "1Kids".

Table 10. Les plages d'adresses commençant par "1Kids"

De	1Kids11111111111111111111111111111
	1Kids11111111111111111111111111112
	1Kids11111111111111111111111111113
	...
A	1Kidszzzzzzzzzzzzzzzzzzzzzzzzzzzz

Regardons à quel nombre "1Kids" correspond en base 58, et essayons de calculer la probabilité de trouver une adresse correspondante (voir Probabilité de trouver une adresse personnalisée (commençant par 1KidsCharity) et temps moyen pour la trouver sur un PC de bureau.). Un PC de bureau standard, sans matériel dédié, peut essayer environ 100 000 clefs par secondes.

Table 11. Probabilité de trouver une adresse personnalisée (commençant par 1KidsCharity) et temps moyen pour la trouver sur un PC de bureau.

Longueur	Mot recherché	Fréquence	Temps moyen de recherche
1	1K	1 clef sur 58	< 1 milliseconde
2	1Ki	1 clef sur 3364	50 millisecondes
3	1Kid	1 clef sur 195000	< 2 secondes
4	1Kids	1 clef sur 11 millions	1 minute

Longueur	Mot recherché	Fréquence	Temps moyen de recherche
5	1KidsC	1 clef sur 656 million	1 heure
6	1KidsCh	1 clef sur 38 milliards	2 days
7	1KidsCha	1 clef sur 2.2 trillions	3–4 mois
8	1KidsChar	1 clef sur 128 trillions	13–18 ans
9	1KidsChari	1 clef sur 7 quadrillions	800 ans
10	1KidsCharit	1 clef sur 400 quadrillions	46,000 ans
11	1KidsCharity	1 sur 23 quintillions	2.5 millions d'années

Comme vous pouvez le constater, Eugenia n'est pas prête d'avoir son adresse "1KidsCharity", même en utilisant des millers d'ordinateurs. Chaque caractère supplémentaire multiplie le temps de recherche par 58. Chercher des adresses commençant par des motifs de plus de 7 lettres se fait en général sur des PC dédiés disposant de plusieurs processeurs graphiques (GPUs). Ce sont souvent d'ancien mineurs de bitcoin qui ne sont plus rentables mais qu'on peut utiliser pour chercher des adresses personnalisées. Ce type de calcul est beaucoup plus rapide (plusieurs ordres de grandeur) sur GPU que sur un CPU généraliste.

Pour générer une adresse personnalisée il est aussi possible de s'adresser à un "pool" (regroupement) spécialisé comme Vanity Pool. Un "pool" permet à des personnes possédant des GPU de se regrouper et de gagner de l'argent en cherchant des adresses personnalisées. Pour une somme modique (0.01 bitcoin soit environ $5 au moment de l'écriture de ce livre) Eugenia peut acheter une adresse personnalisée commençant par un motif de 7 caractères, et avoir le résultat en quelques heures au lieu d'attendre plusieurs mois en faisant la recherche sur un ordinateur de bureau.

La génération d'une adresse personnalisée est une recherche par force brute: on essaie une clef au hasard, on vérifie si l'adresse contient le motif souhaité, on recommence jusqu'à ce qu'on ait trouvé. Mineur d'adresses personnalisées est un exemple de "mineur" d'adresses personnalisées écrit en C++ et basé sur la librarie libbitcoin que nous avons déja vu au chapitre [alt_libraries].

Example 8. Mineur d'adresses personnalisées

```
#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_hex(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 pubkey...
    bc::ec_point pubkey = bc::secret_to_public_key(secret);
    // Finally create address.
    bc::payment_address payaddr;
```

```
    bc::set_public_key(payaddr, pubkey);
    // Return encoded form.
    return payaddr.encoded();
}

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

NOTE Cet exemple utilise std::random_device. Certaines implémentations utilisent un générateur aléatoire cryptographiquement sur (CSRNG) fourni pas l'OS. Dans le cas d'OS de type UNIX, comme Linux, ces nombres sont lus depuis le fichier /dev/urandom. Le générateur aléatoire utilisé ici est suffisant pour une démonstration, mais il n'est *pas assez sécurisé* pour une utilisation en production.

Ce code soit être compilé avec un compilateur C et linké avec la librarie libbitcoin (qui doit d'abord être installée). Pour exécuter l'exemple, lancez le programme vanity-miner++ sans paramètres (voir Compilation et exécution du programme vanity-miner), il essaiera de trouver une adresse personnalisée commençant par "1kid".

Example 9. Compilation et exécution du programme vanity-miner

```
$ # Compilation avec g++
$ g++ -o vanity-miner vanity-miner.cpp $(pkg-config --cflags --libs libbitcoin)
$ # Lancement de l'exemple
$ ./vanity-miner
Found vanity address! 1KiDzkG4MxmovZryZRj8tK81oQRhbZ46YT
Secret: 57cc268a05f83a23ac9d930bc8565bac4e277055f4794cbd1a39e5e71c038f3f
$ # Nouvelle exécution qui produira un résultat différent
$ ./vanity-miner
Found vanity address! 1Kidxr3wsmMzzouwXibKfwTYs5Pau8TUFn
Secret: 7f65bbbbe6d8caae74a0c6a0d2d7b5c6663d71b60337299a1a2cf34c04b2a623
# Utilisation de la commande "time" pour mesurer le temps pris pour trouver le r
ésultat
$ time ./vanity-miner
Found vanity address! 1KidPWhKgGRQWD5PP5TAnGfDyfWp5yceXM
Secret: 2a802e7a53d8aa237cd059377b616d2bfcfa4b0140bc85fa008f2d3d4b225349

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

Il ne faut que quelques secondes pour trouver une adresse commençant par "kid", comme nous pouvons le mesurer avec la commande unix time. Essayez de changer le motif recherché dans le code source et regardez combien il faut de temps pour trouver un motif de 4 ou 5 caractères!

Sécurité des adresses personnalisées

Les adresses personnalisées peuvent être utilisées pour renforcer ou affaiblir la sécurité de vos bitcoins, et doivent être vues comme une arme à double tranchant. Une adresse personnalisée très distinctive peut être un atout en terme de sécurité, car il sera difficile à un attaquant de la remplacer par son adresse pour tromper vos clients. Mais il est aussi possible pour un attaquant de générer une adresse personnalisée qui *ressemble* à n'importe quelle autre (ou même qui ressemble à une autre adresse personnalisée) pour tromper vos clients.

Eugenia pourrait utiliser une adresse de dons aléatoire (par exemple 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy) ou une adresse personnalisée commençant par 1Kids et facilement reconnaissable.

Dans les 2 cas, utiliser une adresse fixe (plutôt qu'une adresse générée dynamiquement, pour chaque donneur) est dangereux car un pirate pourrait attaquer votre site et remplacer cette adresse par la sienne, et ainsi capter les dons. Si vous publiez votre adresse de dons à différents endroits, vos utilisateurs peuvent essayer de vérifier visuellement qu'ils paient vers la bonne adresse, qui apparait sur votre site, vos emails, etc.... Dans le cas d'une adresse aléatoire comme 1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy, l'utilisateur moyen se contentera de vérifier que

l’adresse commence bien par "1J7mdg". Utilisant un générateur d’adresses personnalisées, un attaquant pourrait la remplacer par une adresse qui commence de la même façon, voir Génération d’une adresse personnalisée qui ressemble à une adresse aléatoire.

Table 12. Génération d’une adresse personnalisée qui ressemble à une adresse aléatoire

Adresse aléatoire originale	1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy
Adresse personnalisée(mêmes 4 premiers caractères)	1J7md1QqU4LpctBetHS2ZoyLV5d6dShhEy
Adresse personnalisée(mêmes 5 premiers caractères)	1J7mdgYqyNd4ya3UEcq31Q7sqRMXw2XZ6n
Adresse personnalisée(mêmes 6 premiers caractères)	1J7mdg5WxGENmwyJP9xuGhG5KRzu99BBCX

Est-ce que les adresses personnalisées améliorent la sécurité? Si Eugenia utilise l’adresse 1Kids33q44erFfpeXrmDSz7zEqG2FesZEN, ses utilisateurs vont probablement vérifier que l’adresse de dons commence bien par "1Kids33" et obliger un attaquant à générer une adresse personnalisée dont les 6 premiers caractères sont les mêmes (2 de plus que "1Kids"), soit un effort 3364 fois (58 × 58) plus important que pour le calcul de l’adresse d’Eugenia. En gros, l’investissement d’Eugenia dans une adresse personnalisée oblige un attaquant à générer une adresse personnalisée plus couteuse. Si Eugenia avait voulu une adresse personnalisée en choisissant les 8 premiers caractères, l’attaquant devrait sans doute spécifier les 10 premiers, ce qui serait infaisable sur un ordinateur personnel et très couteux en passant par un mineur dédié ou un pool de minage. Ainsi ce qui est abordable en termes de couts pour Eugenia devient trop couteux pour un attaquant, surtout si les gains potentiels de la fraude ne couvrent pas la génération d’une adresse personnalisée.

Portemoinnaie papier

Les portemonnaie papier sont des clefs privées bitcoin imprimées sur papier. On peut aussi imprimer les adresses bitcoin associées mais ce n’est pas obligatoire car on peut toujours les générer à partir des clefs privées. Les portemonnaie papier sont une bonne solution de sauvegarde ou de stockage en mode déconnecté ("cold storage"). Utilisé comme sauvegarde, cela permet de se protéger contre une panne de disque dur, la perte ou le vol de son ordinateur, ou la suppression accidentelle de ses clefs. Utilisé comme "cold storage", un portemonnaie papier dont les clefs ont été générées sur un ordinateur non connecté à internet, et qui n’ont jamais été importées sur un ordinateur connecté, est à l’abri des hackers, key-loggers et autre piratages.

Il existe beaucoup de portemonnaie papier, mais à la base il s’agit simplement de clefs et adresses imprimées sur papier. Le portemonnaie papier le plus simple — une clef privée et une adresse bitcoin imprimées sur du papier est un exemple de portemonnaie papier le plus simple possible.

Table 13. Le portemonnaie papier le plus simple — une clef privée et une adresse bitcoin imprimées sur du papier

Adresse publique	Clef privée (WIF)
1424C2F4bC9JidNjjTUZCbUxv6Sa1Mt62x	5J3mBbAH58CpQ3Y5RNJpUKPE62SQ5tfcvU2JpbnkeyhfsYB1Jcn

Pour générer un portemonnaie papier on utilise des outils comme le générateur JavaScript *bitaddress.org*. Cette page contient le code nécessaire à la génération de clef et de portemonnaie papier dans la navigateur, même sans connexion internet. Pour l'utiliser, sauvez la page HTML sur disque, ou sur une clef USB. Déconnectez-vous d'internet et ouvrez la page HTML depuis un navigateur. Encore mieux: bootez sur un OS vierge, en utilisant un Linux sur CD bootable par exemple. Vous pourrez ensuite créer des clefs et les imprimer avec une imprimante reliée par un cable USB, pour obtenir un portemonnaie papier dont les clefs n'existe que sur papier et n'ont jamais été stockées sur un système connecté à internet. Il n'y a plus qu'à le ranger dans un coffre ignifugé et "envoyer" des bitcoins vers les adresses du portemonnaie pour avoir une solution de "cold storage" simple mais très sécurisée. Exemple de portemonnaie papier simple généré sur bitaddress.org montre un portemonnaie papier généré avec bitaddress.org.

Figure 14. Exemple de portemonnaie papier simple généré sur bitaddress.org

Le problème avec les portemonnaie papier c'est qu'on peut les voler. Un voleur qui a accès au portemonnaie peut le voler ou le prendre en photo et récupérer les bitcoins associés à ses clefs. On peut utiliser un portemonnaie papier plus avancé dont les clefs sont chiffrées selon le standard BIP0038. Les clefs sont protégées par une phrase de passe mémorisée par l'utilisateur, et le portemonnaie est inutilisable sans cette phrase. C'est une meilleure solution qu'un portemonnaie chiffré car les clefs n'ont jamais été sur un ordinateur connecté: il faut les voler dans un coffre ou un système de stockage sécurisé. Exemple de portemonnaie papier chiffré généré sur bitaddress.org. La phrase de passe est "test".montre un portemonnaie papier dont les clefs privées sont chiffrées (BIP0038) créé avec bitaddress.org.

Figure 15. Exemple de portemonnaie papier chiffré généré sur bitaddress.org. La phrase de passe est "test".

WARNING

Bien que l'on puisse envoyer des fonds vers un portemonnaie papier en plusieurs fois, il est conseillé de tout retirer en une seule fois, et de tout dépenser. En effet si on ne dépense pas tout certains portemonnaie vont générer une adresse de change. De plus, si l'ordinateur utilisé pour signer la transaction est compromis, on risque de dévoiler la clef privée. En dépensant tout en une fois, on diminue le risque de dévoiler la clef. Si vous n'avez besoin que d'une petite partie des fonds, envoyez le reste vers un autre portemonnaie papier dans la même transaction.

Il existe une grande variété de portemonnaie papier (design, format, …). Certains sont destinés à être donnés en cadeau et suivent une thématique précise (Noel, fêtes de fin d’année). D’autres sont prévus pour être conservés dans un coffre, la clef privée étant protégée (par un film opaque, parce que le portemonnaie est plié et collé avec une bande adhésive de sécurité). On peut voir ici <xref linkend="paper_wallet_bpw" xrefstyle="select: labelnumber"/> through <xref linkend="paper_wallet_spw" xrefstyle="select: labelnumber"/> différents portemonnaie papier et leur caractéristiques.

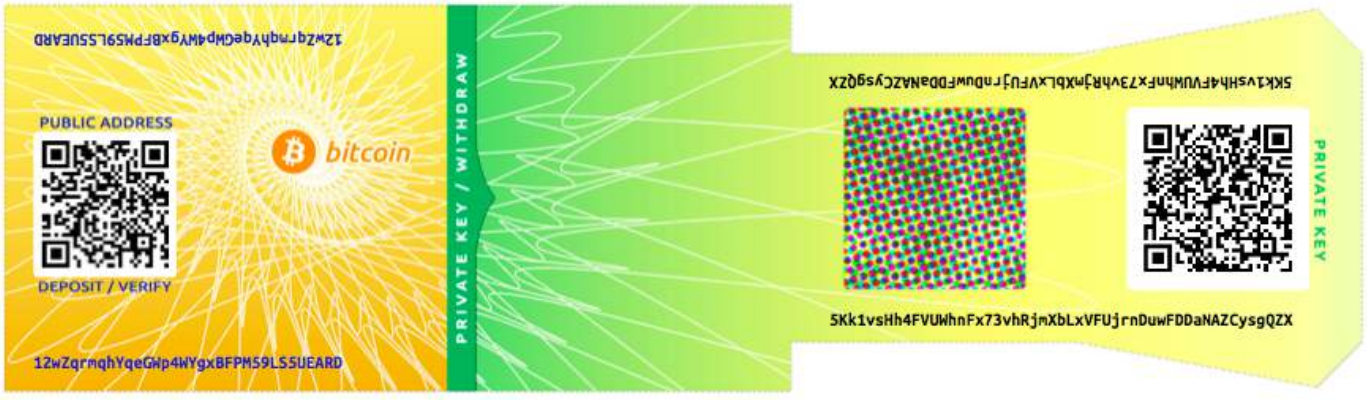


Figure 16. Porte monnaie vendu par bitcoinpaperwallet.com, la clef est protégée par un volet qui se replie.

Figure 17. Portemonnaie bitcoinpaperwallet.com replié, la clef privée est masquée.

Sur d'autres portemonnaie, on trouve plusieurs copies des clefs et adresses sur des coupons détachables, ce qui permet de stocker plusieurs copies et de les protéger contre les incendies, inondations et autres désastres naturels.

Figure 18. Portemonnaie papier avec plusieurs copies de la clef sur des coupons détachables.

Transactions

Introduction

Les transactions sont la partie la plus importante du système bitcoin. Tout le système vise en effet à assurer que des transactions puissent être créées, diffusées sur le réseau, validées, et enfin ajoutée au registre global des transactions (la blockchain). Les transactions sont des structures de données qui encodent le transfert de valeur entre membres du système bitcoin. Chaque transaction est une entrée publique dans la blockchain bitcoin, le registre mondial de comptabilité en partie double.

Dans ce chapitre, nous allons examiner les différentes formes de transactions, ce qu'elles contiennent, comment elles sont créées, vérifiées, et intégrées dans l'enregistrement permanent de toutes les transactions.

Cycle de vie d'une transaction

Le cycle de vie d'une transaction commence avec sa création, aussi désignée de la façon suivante *origination*. La transaction est ensuite signée avec une ou plusieurs signatures signifiant l'autorisation de dépenser les fonds référencés dans la transaction. Elle est ensuite diffusée sur le réseau bitcoin, au sein duquel chaque nœud réseau (participant) valide et continue sa diffusion jusqu'à ce qu'elle ait atteint tous les nœuds réseau ou presque. Enfin, la transaction est vérifiée par un nœud de minage et intégrée dans un bloc de transactions qui est enregistré sur la blockchain.

Une fois enregistrée sur la blockchain et confirmée par les blocs suivants (confirmations), la transaction devient partie intégrante du registre bitcoin et est acceptée comme valide par tous les participants. Les fonds alloués à un nouveau propriétaire par la transaction peuvent alors être dépensés dans une nouvelle transaction, ajoutant un nouveau maillon à la chaîne que forment les propriétaires successifs et répétant à nouveau le cycle de vie d'une transaction.

Créer des transactions

D'une certaine manière, il peut être utile d'envisager la transaction comme un chèque. Comme un chèque, une transaction est un instrument qui exprime l'intention de transférer de l'argent et est invisible dans le système financier jusqu'au moment où quelqu'un en exige le paiement. Comme dans le cas d'un chèque, l'émetteur d'une transaction n'est pas nécessairement celui qui signe la transaction.

Les transactions peuvent être créées en ligne ou hors ligne par n'importe qui, même quelqu'un qui n'est pas autoriser à signer sur le compte impliqué. Par exemple, un commis peut écrire lui-même des chèques que le PDG devra signer. De même, il peut créer des transactions bitcoin qui nécessiteront la signature électronique du PDG pour être valides. Cependant, le chèque indique un compte précis comme origine des fonds, tandis qu'une transaction bitcoin indiquera une transaction antérieure comme source.

Dès qu'une transaction a été créée, elle est signée par le ou les propriétaires de la source des fonds. Si

elle est correctement formulée et signée, elle est alors valide et contient toutes les informations nécessaires à l'exécution du transfert de fonds. Enfin, la transaction validée doit être mise sur le réseau bitcoin afin de pouvoir être propagée jusqu'à un mineur et intégrée au registre public (blockchain).

Diffuser les transactions sur le réseau bitcoin

Tout d'abord, une transaction doit être délivrée sur le réseau bitcoin pour pouvoir être diffusée et incluse dans la blockchain. Une transaction bitcoin n'est essentiellement que 300 ou 400 bytes de données et doit atteindre n'importe lequel des dizaines milliers de nœuds que compte le réseau. Les émetteurs n'ont pas besoin de faire confiance aux noeuds qu'ils utilisent pour diffuser leur transaction, tant qu'ils en utilisent plus qu'un pour s'assurer de leur bonne diffusion. Les noeuds n'ont pas besoin de faire confiance les émetteurs ou d'établir leur "identité". Parce que la transaction est signée et ne contient aucune information confidentielle, clés privées ou identifiants, elle peut être diffusée publiquement au moyen de n'importe quel réseau sous-jacent convenant à cet usage. Contrairement aux transactions, par exemple, des cartes de crédit, qui contiennent des informations sensibles et ne peuvent être transmises que sur des réseaux cryptés, une transaction bitcoin peut être envoyée sur n'importe quel réseau. Tant que la transaction peut atteindre un premier noeud qui va la diffuser sur le réseau bitcoin, la façon dont elle atteint ce premier noeud importe peu.

Les transactions bitcoin peuvent donc être transmises sur le réseau bitcoin par des réseaux non sécurisés tels que WiFi, Bluetooth, Chirp, code-barres, ou par un copier-coller sur une page internet. Dans des cas extrêmes, une transaction bitcoin pourrait être transmise par packet radio, relai satellite, ou ondes courtes en transmission par rafales, étalement de spectre ou saut de fréquence afin d'éviter les tentatives d'interception et de brouillage. Une transaction bitcoin pourrait même être encodée sous la forme d'un smiley (émoticône) et postée sur un forum public, envoyée dans un message textuel ou sur le chat de Skype. Il est virtuellement impossible d'empêcher qui que ce soit de créer et exécuter une transaction bitcoin, car ce dernier a transformé l'argent en structure de données.

Propager les Transactions sur le Réseau Bitcoin

Une fois qu'une transaction est envoyée à n'importe quel nœud connecté au réseau bitcoin, elle doit être validée par ce nœud. Si elle est valide, ce nœud la diffusera ensuite aux autres nœuds auxquels il est connecté, et un message sera envoyé simultanément à l'émetteur de la transaction pour l'avertir du succès de la transaction. Si la transaction est invalide, le nœud la rejettera et enverra simultanément un message à l'émetteur pour l'en avertir.

Le réseau bitcoin est un réseau peer-to-peer, ce qui signifie que chaque nœud bitcoin est connecté aux autres nœuds qu'il découvre par un protocole peer-to-peer à chaque initialisation. Le réseau pris dans son intégralité est composé de maillons connectés de manière lâche, sans topologie fixe ni structure définie, et ainsi tous les nœuds sont égaux, d'où le terme de "pairs". Les messages, incluant les transactions et les blocs, sont diffusés de chaque nœud à tous les pairs auxquels ils sont connectés, une opération appelée "flooding". Une transaction nouvellement validée, une fois injectée dans n'importe quel nœud du réseau, est envoyée à tous les nœuds auxquels il est connecté (neighbors), puis chacun de ces nœuds la transmet à ses propres voisins, et ainsi de suite. Une transaction valide va ainsi être diffusée en quelques secondes à travers le réseau, comme une onde se propageant de façon

exponentielle, jusqu'à ce que tous les nœuds du réseau l'aient reçue.

Le réseau bitcoin est conçu pour diffuser les transactions et les blocs à tous les nœuds d'une façon efficace et résiliente, ainsi que résistante aux attaques. Pour éviter le spamming, les attaques par déni de service, ou toute autre attaque par nuisance contre le système bitcoin, chaque nœud valide indépendamment chaque transaction avant de la diffuser plus loin dans le réseau. Une transaction incorrectement formulée ne passera donc pas le premier nœud. Les règles permettant de valider les transactions sont expliquées en détails dans [tx_verification].

Structure des Transactions

Une transaction est une*structure de données* qui encode un transfert de valeur d'une source de fonds, appelé *input*, à un point de destination, appelée *output*. Les input et output de transaction ne sont pas liés à des comptes ou des identités. Au contraire, vous devriez plutôt les imaginer comme des montants en bitcoin—des morceaux de bitcoin—verrouillés grâce à un secret particulier que seulement le propriétaire, ou une personne qui connaîtrait son secret, pourrait déverrouiller. Une transaction comprend un certain nombre de champs, comme le montre La structure d'une transaction.

Table 1. La structure d'une transaction

Taille	**Champ**	**Description**
4 octets	Version	Spécifie quelles règles suit cette transaction
1-9 octets (VarInt)	Compteur d'Input	Combien d'inputs sont inclus
Variable	Inputs	Un ou plusieurs inputs de transaction
1-9 octets (VarInt)	Compteur d'Output	Combien d'output sont inclus
Variable	Outputs	Un ou plusieurs outputs de transaction
4 octets	Locktime	Un horodateur Unix ou le numéro d'un bloc

Temps de verrouillage d’une transaction

Le locktime, ou temps de verrouillage, aussi appelé "nLockTime" selon le nom utilisé dans le client de référence, définit le moment où une transaction devient valide afin d’être relayée sur le réseau ou ajoutée à la blockchain. Cette valeur est le plus souvent nulle, permettant la diffusion et l’exécution immédiate de la transaction. Si non-nulle et inférieure à 500 million, elle est interprétée comme un numéro de bloc, ce qui signifie que la transaction n’est pas valide et ne sera pas relayée et incluse dans la blockchain tant que celle-ci n’aura pas atteint le bloc correspondant à la valeur indiquée. Si cette valeur est supérieure à 500 million, elle est interprétée comme une indication de temps Unix Epoch (secondes depuis le 1er janvier 1970) et la transaction n’est pas valide avant le temps indiqué. Les transactions dont la valeur locktime indique un bloc ou un moment futur doivent être conservées dans leur système d’origine et transmises au réseau bitcoin seulement une fois celles-ci valides. Utiliser le locktime revient ainsi à post-dater un chèque.

Outputs et Inputs de transaction

Le bloc constitutif fondamental d’une transaction bitcoin est un *unspent transaction output*, ou UTXO. Les UTXO sont des morceaux indivisibles de bitcoin liés à un utilisateur précis, enregistrés sur la blockchain, et reconnu comme des unités monétaires par le réseau entier. Le réseau bitcoin garde la trace de tous les UTXO disponible (non dépensés), dont le montant se compte aujourd’hui en millions. A chaque fois qu’un utilisateur reçoit des bitcoins, le montant est enregistré dans la blockchain en tant qu’UTXO. Ainsi, les bitcoins d’un utilisateur en tant qu’UTXO peuvent être éparpillés entre des centaines de transactions et de blocs. En effet, il n’existe rien de comparable à un solde propre à un compte ou à une adresse ; il n’y a que des UTXO épars, liés à un utilisateur précis. Le concept du solde d’un utilisateur de bitcoin est une invention des applications de wallet. Le wallet calcule le solde de l’utilisateur en examinant la blockchain et en agrégeant tous les UTXO qui lui appartiennent.

TIP Il n’y a pas de comptes ou de soldes en bitcoins ; il n’y a que des *unspent transaction outputs* (UTXO) éparpillés sur la blockchain.

Un UTXO peut avoir une valeur arbitraire, libellée comme un multiple desatoshis. De même que le dollar peut être divisé en "cents", la plus petite subdivision de bitcoin, le "satoshi", représente un cent-millionième de bitcoin. Bien qu’un UTXO puisse représenter n’importe quelle valeur arbitraire, une fois créé il est aussi indivisible qu’une pièce qu’on ne peut pas couper en deux. Si le montant d’un UTXO est plus important que la valeur de la transaction, il doit néanmoins être consommé intégralement dans le processus et générer un retour de monnaie à l’émetteur. En d’autres termes, si vous avez un UTXO de 20 bitcoins et que vous désirez payer 1 bitcoin, alors la transaction doit d’abord consommer les 20 bitcoins de l’UTXO avant de produire deux outputs : le paiement d'1 bitcoin aux destinataires de la transaction et un autre paiement de 19 bitcoins vers votre wallet, comme si on vous rendait la monnaie. En conséquence, la plupart des transactions en bitcoin génère un rendu de monnaie.

Imaginez une femme en pleine séance de shopping désirant acheter une boisson à 1,50$. Elle va sortir son porte-feuille et essayer de rassembler une combinaison de pièces et de billets pour couvrir la totalité de ces 1,50$. Si possible, elle prendra le montant exact (un billet d'un dollar et deux pièces de 25 cents), ou alors une combinaison de pièces (six pièces de 25 cents), ou même, si nécessaire, une dénomination supérieure comme un billet de 5 dollars. Si elle donne au commerçant un montant trop élevé, comme le billet de 5 dont nous venons de parler, elle s'attendra naturellement à ce qu'on lui rende 3,50$ de monnaies, qu'elle remettra dans son porte-feuille et gardera disponible pour de futures transactions.

De même, une transaction bitcoin doit être créée à partir de l'UTXO d'un utilisateur dans n'importe quelle dénomination disponible pour cet utilisateur. Les utilisateurs ne peuvent pas couper un UTXO en deux, comme ils ne peuvent pas le faire avec un billet d'un dollar et encore s'en servir dans leurs transactions. L'application wallet de l'utilisateur sélectionnera en général dans les UTXO de l'utilisateur disponibles différentes unités pour composer un montant supérieur ou égal au montant désiré de la transaction.

Comme dans la vraie vie, l'application bitcoin peut user de plusieurs stratégies pour arriver au montant de l'achat : combiner plusieurs unités d'un montant inférieur, trouver la monnaie exacte ou utiliser une unité d'un montant plus grand que la transaction et récupérer la monnaie. Tout cet assemblage complexe de UTXO est réalisé par le porte-monnaie de l'utilisateur automatiquement de façon transparente et invisible. L'information concernant cette construction ne vous est utile que dans le cas ou vous construisez vous-mêmes en codant des transactions à partir d'UTXO.

Les UTXO consommées par une transaction sont appelées les inputs de transaction, les UTXO créées par une transaction sont appelées outputs de transaction. De cette façon des montants de valer en bitcoin sont déplacés de propriétaires en propriétaires dans une chaine de transactions consommant et créant des UTXO. Les transaction consomment des UTXO en les déverrouillant avec la signature du propriétaire actuel et créent des UTXO en les verrouillant sur l'adresse du nouveau propriétaire.

L'exception de cette chaine d'inputs et d'outputs est un type spécial de transaction appelé la transaction *coinbase*, qui est la première transaction d'un bloc. Cette transaction est positionnée par le mineur "gagnant" et créée de tous nouveaux bitcoins payables à ce mineur comme récompense du minage. C'est ainsi que se fait la création monétaire bitcoin à l'issue du processus de minage comme nous le verrons dans le [ch8].

TIP	Qui vient en premier ? Les inputs ou les outputs, l'oeuf ou la poule ? Les outputs viennent en premier car les transaction coinbase, qui génèrent de nouveaux bitcoins, n'ont pas d'inputs et créent des outputs ex nihilo.

Outputs de transaction

Touts les transaction bitcoin créent des outputs qui sont enregistrés dnas le registre bitcoin. Presque tous ces outputs, à l'exception d'un (voir Data Output (OP_RETURN)), créent des montants de bitcoin que l'on pourra dépenser appelés *unspent transaction outputs* ou UTXO qui sont reconnues ensuite par le réseau comme disponibles à la dépense pour de futures transactions. Envoyer des bitcoins à

quelqu'un consiste en la création de UTXO associées à son adresse qu'il pourra dépenser par la suite.

les UTXO sont pistées par tous les noeuds complet bitcoin (les full nodes) dans un ensemble de données appelé le *UTXO set* ou le *UTXO pool* sauvegardé en base de données. Les nouvelles transaction consomment (dépensent) un ou plusieurs de ces outputs contenus dans le UTXO set.

Les outputs de transaction sont constitués de deux parties :

- Un montant de bitcoin en *satoshis*, l'unité la plus petite de bitcoin
- Un *locking script* (ou script de verrouillage), également appelé "charge" qui "verrouille" ce montant en spécifiant les conditions nécessaires à la dépense de cet output

Le langage de script de transaction utilisé pour le locking script mentionné précédemment est décrit en détail dansScript des Transactions et Langage de Script. La structure d'un output de transaction détaille la structure d'un output de transaction.

Table 2. La structure d'un output de transaction

Taille	**Champ**	**Description**
8 octets	Montant	Valeur bitcoin en satoshis (10^{-8} bitcoin)
1-9 octets (VarInt)	Taille locking script	Longueur du locking script en octets
Variable	Locking-Script	Un script qui définit les conditions nécessaires à l a dépense de l'output

Dans Un script qui fait appel à l'API blockchain.info pour trouver les UTXO associés à une adresse, nous utilisons l'API de blockchain.info pour trouver les outputs non dépensés (UTXO) d'une adresse donnée.

Example 1. Un script qui fait appel à l'API blockchain.info pour trouver les UTXO associés à une adresse

```
# get unspent outputs from blockchain API

import json
import requests

# example address
address = '1Dorian4RoXcnBv9hnQ4Y2C1an6NJ4UrjX'

# The API URL is https://blockchain.info/unspent?active=<address>
# It returns a JSON object with a list "unspent_outputs", containing UTXO, like this:
#{  "unspent_outputs":[
#   {
#     "tx_hash":"ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167",
#     "tx_index":51919767,
#     "tx_output_n": 1,
#     "script":"76a9148c7e252f8d64b0b6e313985915110fcfefcf4a2d88ac",
#     "value": 8000000,
#     "value_hex": "7a1200",
#     "confirmations":28691
#   },
# ...
#]}

resp = requests.get('https://blockchain.info/unspent?active=%s' % address)
utxo_set = json.loads(resp.text)["unspent_outputs"]

for utxo in utxo_set:
    print "%s:%d - %ld Satoshis" % (utxo['tx_hash'], utxo['tx_output_n'],
utxo['value'])
```

En exécutant ce script nous obtenons une liste d'identifiants de transactions, le numéro d'index d'un output de transaction non dépensé (UTXO) et la valeur de cet UTXO en satoshis. Le locking script n'est pas affiché dans le résultat Lancement du script get-utxo.py.

Example 2. Lancement du script get-utxo.py

```
$ python get-utxo.py
ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167:1 - 8000000 Satoshis
6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 - 16050000
Satoshis
74d788804e2aae10891d72753d1520da1206e6f4f20481cc1555b7f2cb44aca0:0 - 5000000 Satoshis
b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4:0 - 10000000
Satoshis
...
```

Conditions de dépense (charges)

Les outputs de transactions associent un montant spécifique (en satoshis) à une *charge* spécifique ou locking script qui définit les conditions à remplir pour dépenser ce montant. La plupart du temps le locking script va verrouiller l'output sur une adresse bitcoin spécifique, transférant par conséquent la possession de ce montant au nouveau propriétaire. Quand Alice a payé le Bob's Cafe pour son espresso, sa transaction a créé un output *chargé* ou verrouillé sur l'adresse bitcoin du café. Cet output de 0.015 bitcoin a été enregistré dans la blockchain et ajouté dans le UTXO set et a par conséquent été rendu visible dans le porte-monnaie de Bob comme faisant partie de son solde disponible. Quand Bob veut dépenser ce montant, sa transaction va libérer la charge et déverrouiller l'output en fournissant le locking script contenant la signature provenant de sa clé privée.

Inputs de transaction

On peut définir simplement les inputs de transaction comme étant des pointeurs vers des UTXO. Ils pointent vers un UTXO spécifique en référençant un hash de transaction et un numéro de séquence ou est enregistré l'UTXO dans la blockchain. Pour dépenser un UTXO, un input de transaction inclut également un script de déverrouillage qui remplit les conditions de dépenses de l'UTXO. Le script de déverrouillage est généralement une signature prouvant la possession de l'adresse bitcoin contenue dans le locking script.

Quand les utilisateurs effectuent un paiement, leurs porte-monnaies construisent une transaction en piochant dans les UTXO disponibles. Par exemple, pour effectuer un paiement de 0.015 bitcoin, le porte-monnaie peut sélectionner un UTXO de 0.0& et un autre de 0.005, pour obtenir le montant désiré.

In Un script pour calculer combien de bitcoins seront émis, we show the use of a "greedy" algorithm to select from available UTXO in order to make a specific payment amount. In the example, the available UTXO are provided as a constant array, but in reality, the available UTXO would be retrieved with an RPC call to Bitcoin Core, or to a third-party API as shown in Un script qui fait appel à l'API blockchain.info pour trouver les UTXO associés à une adresse.

Example 3. Un script pour calculer combien de bitcoins seront émis

```
# Selects outputs from a UTXO list using a greedy algorithm.

from sys import argv

class OutputInfo:

    def __init__(self, tx_hash, tx_index, value):
        self.tx_hash = tx_hash
        self.tx_index = tx_index
        self.value = value

    def __repr__(self):
        return "<%s:%s with %s Satoshis>" % (self.tx_hash, self.tx_index,
                                             self.value)

# Select optimal outputs for a send from unspent outputs list.
# Returns output list and remaining change to be sent to
# a change address.
def select_outputs_greedy(unspent, min_value):
    # Fail if empty.
    if not unspent:
        return None
    # Partition into 2 lists.
    lessers = [utxo for utxo in unspent if utxo.value < min_value]
    greaters = [utxo for utxo in unspent if utxo.value >= min_value]
    key_func = lambda utxo: utxo.value
    if greaters:
        # Not-empty. Find the smallest greater.
        min_greater = min(greaters)
        change = min_greater.value - min_value
        return [min_greater], change
    # Not found in greaters. Try several lessers instead.
    # Rearrange them from biggest to smallest. We want to use the least
    # amount of inputs as possible.
    lessers.sort(key=key_func, reverse=True)
    result = []
    accum = 0
    for utxo in lessers:
        result.append(utxo)
        accum += utxo.value
        if accum >= min_value:
            change = accum - min_value
            return result, "Change: %d Satoshis" % change
    # No results found.
    return None, 0

def main():
    unspent = [
```

```
OutputInfo("ebadfaa92f1fd29e2fe296eda702c48bd11ffd52313e986e99ddad9084062167", 1,
8000000),

OutputInfo("6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf", 0,
16050000),

OutputInfo("b2affea89ff82557c60d635a2a3137b8f88f12ecec85082f7d0a1f82ee203ac4", 0,
10000000),

OutputInfo("7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1", 0,
25000000),

OutputInfo("55ea01bd7e9afd3d3ab9790199e777d62a0709cf0725e80a7350fdb22d7b8ec6", 17,
5470541),

OutputInfo("12b6a7934c1df821945ee9ee3b3326d07ca7a65fd6416ea44ce8c3db0c078c64", 0,
10000000),

OutputInfo("7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818", 0,
16100000),
    ]

    if len(argv) > 1:
        target = long(argv[1])
    else:
        target = 55000000

    print "For transaction amount %d Satoshis (%f bitcoin) use: " % (target,
target/10.0**8)
    print select_outputs_greedy(unspent, target)

if __name__ == "__main__":
    main()
```

Si nous lançons le script *select-utxo.py* sans aucun paramètre, il essaiera de construire un ensemble d'UTXO pour un paiement de 55,000,000 satoshis (0.55 bitcoin). Si vous fournissez un montant spécifique en paramètre, le script va sélectionner des UTXO pour le montant désiré. Dans Lancement du script elect-utxo.py, nous lançons le script en essayant d'effectuer un paiement de 0.5 bitcoin ou 50,000,000 satoshis.

Example 4. Lancement du script elect-utxo.py

```
$ python select-utxo.py 50000000
Pour un montant de transaction de 50000000 Satoshis (0.500000 bitcoin) :
([<7dbc497969c7475e45d952c4a872e213fb15d45e5cd3473c386a71a1b0c136a1:0 with 25000000
Satoshis>, <7f42eda67921ee92eae5f79bd37c68c9cb859b899ce70dba68c48338857b7818:0 with
16100000 Satoshis>,
<6596fd070679de96e405d52b51b8e1d644029108ec4cbfe451454486796a1ecf:0 with 16050000
Satoshis>], 'Change: 7150000 Satoshis')
```

Une fois les UTXO sélectionnés, le porte-monnaie construit les script de déverrouillage contenant les signature pour chaque UTXO, les rendant par là dépensables en satisfaisant les conditions du locking script. Le porte-monnaie ajoute ces référence vers les UTXO et leurs script de déverrouillage en tant qu'inputs de la transaction. La structure d'un input de transaction montre la structure d'un input de transaction.

Table 3. La structure d'un input de transaction

Taille	Champ	Description
32 octets	Hash de transaction	Pointeur vers la transaction contenant l'UTXO à dépenser
4 octets	Index de l'output	Le numéro d'index de l'UTXO à dépenser; le premier étant 0
1-9 octets (VarInt)	Taille du script de déverrouillage	Longueur du script de déverrouillage en octets, (to follow)
Variable	Script de déverrouillage	Un script qui remplit les conditions du script de verrouillage de l'UTXO.
4 octets	Numéro de séquence	Fonctionnalité de remplacement de transaction actuellement désactivée, fixé à 0xFFFFFFFF

NOTE

Le numéro de séquence est utilisé afin de ne pas tenir compte d'une transaction en fonction de l'expiration du temps de verrouillage (locktime) qui est une fonctionnalité actuellement désactivée dans bitcoin. La plupart des transactions positionnent cette valeur à la valeur maximale d'un entier (0xFFFFFFFF) quiest ignorée par le réseau bitcoin. Si une transaction possède un locktime différent de zéro, au moins un des ses inputs doit avoir un numéro de séquence inférieur à 0xFFFFFFFF pour activer le locktime.

Frais de transaction

La plupart des transactions incluent des frais de transaction qui servent à récompenser les mineurs pour la sécurisation du réseau. Le minage les frais et les récompenses collectées par les mineurs sont décrits en détail dans le [ch8]. Cette section décrit comment les frais de transactions sont inclus dans une transaction classique. La plupart des porte-monnaies calculent et incluent les frais de transactions automatiquement. Cependant, dans le cas où vous construisez les transactions via du code ou en utilisant l'interface en ligne de commande, vous devez prendre en compte manuellement ces frais les inclure dans la construction de vos transactions.

Les frais de transaction servent d'incitation à inclure (miner) une transaction dans le prochain bloc et aussi à décourager le "spam" de transactions ou tout autre abus du système en imposant un coût faible pour chaque transaction. Les frais de transaction sont collectés par le mineur qui mine le bloc contenant la transaction dans la blockchain.

Les frais de transaction sont calculés en fonction de la taille de la transaction en kilo-octet et non en fonction de la valeur de la transaction en bitcoin. D'une manière générale, les frais de transaction dépendent des forces en présence dans le marché du réseau bitcoin. Les mineurs priorisent les transaction en fonction de différents critères, dont les frais, et peuvent même traiter des transactions gratuitement dans certaines circonstances. Les frais de transaction affectent la priorité de traitement, ce qui veut dire qu'une transaction avec des frais insuffisants ou nuls peut se retrouver reportée et traitée au bon vouloir des mineurs dans les prochains blocs ou bien tout simplement ignorée. Les frais de transaction ne sont pas obligatoires et les transactions sans frais peuvent être traitées. Cependant, inclure des frais de transaction améliore la priorisation de leur traitement.

Avec le temps, la façon dont les frais sont calculés et l'effet qu'ils ont sur la priorisation des transactions ont évolué. Au debut, les frais de transactions étaient fixes et constants sur le réseau. Petit à petit, la structure des frais s'est assouplie de manière à être influencée par les forces de marché en s'adaptant à la capacité du réseau et au volume total de transactions. Les frais de transaction minimum sont actuellement fixés à 0.0001 bitcoin ou un dixième de millibitcoin par kilo-octet, alors qu'il étaient d'un millibitcoin auparavant. La plupart des transaction pèsent moins d'un kilo-octet ; cependant, les transaction comportant de multiples inputs et outputs peuvent être plus lourdes. Dans les futures révisions du protocole bitcoin, il est prévu que les application de porte-monnaie utilisent une analyse statistique pour calculer le frais les plus appropriés à attacher aux transactions en se basant sur les frais moyens des transaction récentes.

L'algorithme actuel utilisés par les mineurs pour prioriser les transaction pour l'inclusion dans un bloc en se basant sur les frais est détaillé dans le [ch8].

Ajouter des frais aux transactions

La structure de données des transaction ne comporte pas de champ spécifique au frais. Au lieu de ça, les frais sont le résultat de la différence entre la somme des input et la somme des outputs. Tout montant restant une fois que tous les outputs ont été soustraits de tous les inputs représente les frais qui seront collectés par les mineurs.

Les frais de transaction sont implicites comme l'excédent entre les entrées moins les sorties:

```
Frais = Somme (Inputs) - Somme (Outputs)
```

Il s'agit d'un élément assez déconcertant des transactions et très important à comprendre car si vous construisez vos propres transaction, vous devez vous assurer que vous n'y incluez pas trop de frais en omettant de dépenser des inputs. Cela veut dire que vous devez prendre tous les inputs en compte, en créant de la monnaie si nécessaire, ou vous risquez d'offrir un gros pourboire au mineurs !

Par exemple, si vous utilisez un UTXO de 20 bitcoins pour faire un paiement de 1 bitcoin, vous devrez inclure la monnaie via un output de 19 bitcoin vers votre propre porte-monnaie. Dans le cas contraire, les 19 bitcoins restants seront considérés comme des frais de transaction et seront collectés par les mineurs qui mineront votre transaction dans un bloc. Malgré le fait que votre transaction aura une très haute priorité et que vous ferez un mineur très heureux, il est fort probable que ce n'était pas le but recherché initialement.

WARNING	Si vous oubliez d'ajouter un output contenant la monnaie dans une transaction construite manuellement, vous paierez le montant normalement prévu pour la monnaie en frais de transaction. "Garder la monnaie!" n'est peut-être pas ce que vous aviez prévu.

Regardons comment cela ce passe en pratique en regardant encore l'achat du café par Alice. Alive veut dépenser 0.015 bitcoin pour le payer. Pour être sûr que la transaction soit procédée rapidement, ell voudra inclure un frais de transaction, disons 0.001. Cela voudra dire que le coût total de la transaction sera 0.016. Son portefeuille doit donc se procurer un ensemble d'UTXO qui s'ajoutent à 0.016 ou plus et, si nécessaire, créer un changement. Disons que son portefeuille a un UTXO disponible de 0.2-bitcoin . Il devra donc utiliser cet UTXO, créer une sortie pour le café de bob pour 0.015, et une seconde sortie pour 0.184 bitcoin comme monnaie qui retournera dans son portefeuille, laissant 0.001 bitcoin non alloué, comme un frais implicite pour la transaction

Regardons maintenant un scénario différent. Eugenia, qui dirige une association caritative pour enfants aux Philppines, a fini une levée de fond pour acheter des livres d'école. Elle a reçu plusieurs milliers de petites donation du monde entier, atteignant 50 bitcoin, donc son portefeuille est rempli de petits paiements (UTXO). Maintenant elle veut acheter des centaines de livres d'école à un imprimeur local, en payant en 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 2 to 3 kilobytes in size. As a result, it will require a higher fee than the minimal network fee of 0.0001 bitcoin.

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.

Chaînage de Transactions et Transactions Orphelines

As we have seen, transactions form a chain, whereby one transaction spends the outputs of the previous transaction (known as the parent) and creates outputs for a subsequent transaction (known as the child). Sometimes an entire chain of transactions depending on each other—say a parent, child, and grandchild transaction—are created at the same time, to fulfill a complex transactional workflow that requires valid children to be signed before the parent is signed. For example, this is a technique used in CoinJoin transactions where multiple parties join transactions together to protect their privacy.

When a chain of transactions is transmitted across the network, they don't always arrive in the same order. Sometimes, the child might arrive before the parent. In that case, the nodes that see a child first can see that it references a parent transaction that is not yet known. Rather than reject the child, they put it in a temporary pool to await the arrival of its parent and propagate it to every other node. The pool of transactions without parents is known as the *orphan transaction pool*. Once the parent arrives, any orphans that reference the UTXO created by the parent are released from the pool, revalidated recursively, and then the entire chain of transactions can be included in the transaction pool, ready to be mined in a block. Transaction chains can be arbitrarily long, with any number of generations transmitted simultaneously. The mechanism of holding orphans in the orphan pool ensures that otherwise valid transactions will not be rejected just because their parent has been delayed and that eventually the chain they belong to is reconstructed in the correct order, regardless of the order of arrival.

There is a limit to the number of orphan transactions stored in memory, to prevent a denial-of-service attack against bitcoin nodes. The limit is defined as MAX_ORPHAN_TRANSACTIONS in the source code of the bitcoin reference client. If the number of orphan transactions in the pool exceeds MAX_ORPHAN_TRANSACTIONS, one or more randomly selected orphan transactions are evicted from the pool, until the pool size is back within limits.

Script des Transactions et Langage de Script

Bitcoin clients validate transactions by executing a script, written in a Forth-like scripting language. Both the locking script (encumbrance) placed on a UTXO and the unlocking script that usually contains a signature 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.

Today, most transactions processed through the bitcoin network have the form "Alice pays Bob" and are based on the same script called a Pay-to-Public-Key-Hash script. However, the use of scripts to lock outputs and unlock inputs means that through use of the programming language, transactions can contain an infinite number of conditions. Bitcoin transactions are not limited to the "Alice pays Bob" form and pattern.

This is only the tip of the iceberg of possibilities that can be expressed with this scripting language. In this section, we will demonstrate the components of the bitcoin transaction scripting language and show how it can be used to express complex conditions for spending and how those conditions can be satisfied by unlocking scripts.

TIP	La validation des transactions Bitcoin n'est pas basée sur un modèle statique, mais au contraire par l'exécution d'un langage de script. Ce langage permet d'exprimer une variété presque infinie de conditions. C'est ainsi que bitcoin prend la caractéristique de "l'argent programmable".

Construction de Script (Vérrouillage + Déverrouillage)

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 an encumbrance placed on an output, and 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. 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.

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, and 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 is called *scriptSig*, because it usually contained a digital signature. In most bitcoin applications, the source code refers to the unlocking script as scriptSig. 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 client will validate transactions by executing the locking and unlocking scripts together. For each input in the transaction, the validation software will first retrieve the UTXO referenced by the input. That UTXO contains a locking script defining the conditions required to spend it. The validation software will then take the unlocking script contained in the input that is attempting to spend this UTXO and execute the two 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 executed without errors (e.g., it has no "dangling" operators left over), the main stack (not the alternate 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. 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 UTXO results in the UTXO being marked as "spent" and removed from the set of available (unspent) UTXO.

Combining scriptSig and scriptPubKey to evaluate a transaction script 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.

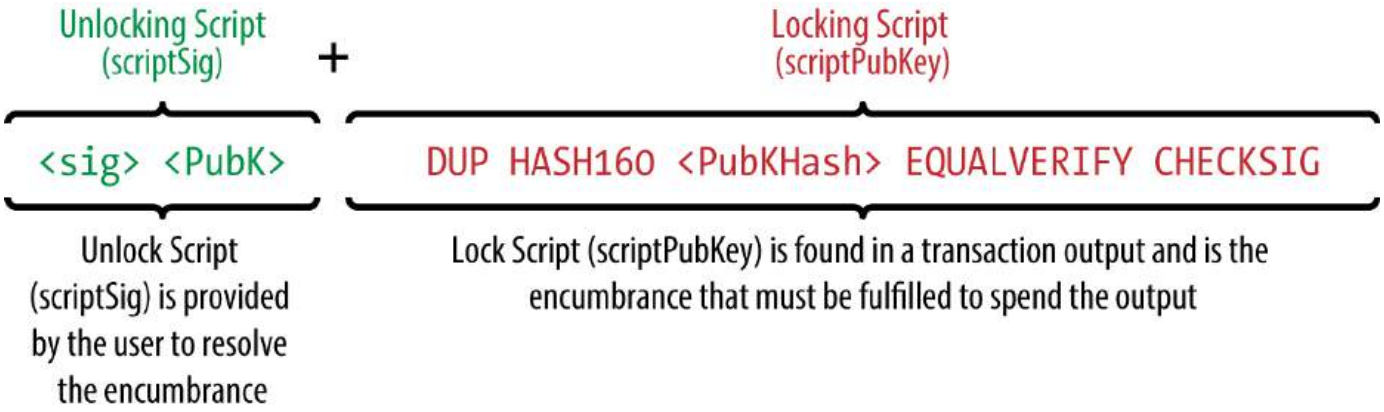


Figure 1. Combining scriptSig and scriptPubKey to evaluate a transaction script

Langage de Scripting

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 1960's programming languages. 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, such as a handheld calculator. It requires minimal processing and cannot do many of the fancy things modern programming languages can do. In the case of programmable money, that is a deliberate security feature.

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, which 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.

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.

In Bitcoin's script validation doing simple math, 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.

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.

Although most locking scripts refer to a bitcoin address or public key, 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 Bitcoin's script validation doing simple math, 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.

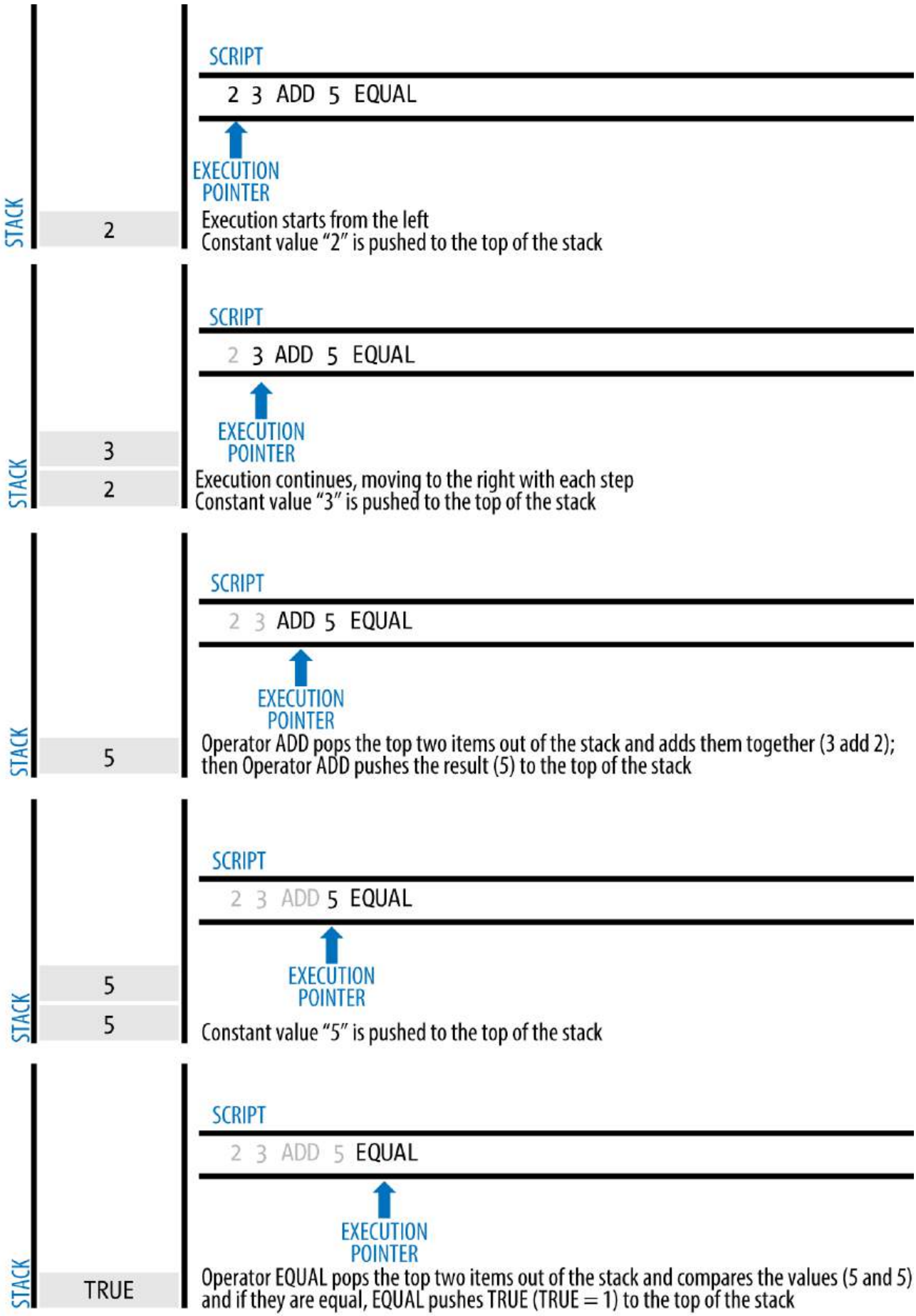

SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
2
Execution starts from the left
Constant value "2" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
3
2
Execution continues, moving to the right with each step
Constant value "3" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
Operator ADD pops the top two items out of the stack and adds them together (3 add 2);
then Operator ADD pushes the result (5) to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
5
5
Constant value "5" is pushed to the top of the stack
SCRIPT
2 3 ADD 5 EQUAL
EXECUTION POINTER
STACK
TRUE
Operator EQUAL pops the top two items out of the stack and compares the values (5 and 5)
and if they are equal, EQUAL pushes TRUE (TRUE = 1) to the top of the stack

Figure 2. Bitcoin's script validation doing simple math

TIP Transactions are valid if the top result on the stack is TRUE (noted as {0x01}), any other non-zero value 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 {}) 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 [tx_script_ops] for details.

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.

Transactions Standard

In the first few years of bitcoin's development, the developers introduced some limitations in the types of scripts that could be processed by the reference client. These limitations are encoded in a function called isStandard(), which defines five types of "standard" transactions. These limitations are temporary and might be lifted by the time you read this. Until then, the five standard types of transaction scripts are the only ones that will be accepted by the reference client and most miners who run the reference client. Although it is possible to create a nonstandard transaction containing a script that is not one of the standard types, you must find a miner who does not follow these limitations to mine that transaction into a block.

Check the source code of the Bitcoin Core client (the reference implementation) to see what is currently allowed as a valid transaction script.

The five standard types of transaction scripts are pay-to-public-key-hash (P2PKH), public-key, multi-

signature (limited to 15 keys), pay-to-script-hash (P2SH), and data output (OP_RETURN), which are described in more detail in the following sections.

Pay-to-Public-Key-Hash (P2PKH)

The vast majority of transactions processed on the bitcoin network are P2PKH transactions. These contain a locking script that encumbers the output with a public key hash, more commonly known as a bitcoin address. Transactions that pay a bitcoin address contain P2PKH scripts. 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.

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_EQUAL 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_EQUAL 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 <xref linkend="P2PubKHash1" xrefstyle="select: labelnumber"/> and <xref linkend="P2PubKHash2" xrefstyle="select: labelnumber"/> show (in two parts) a step-by-step execution of the combined script, which will prove this is a valid transaction.

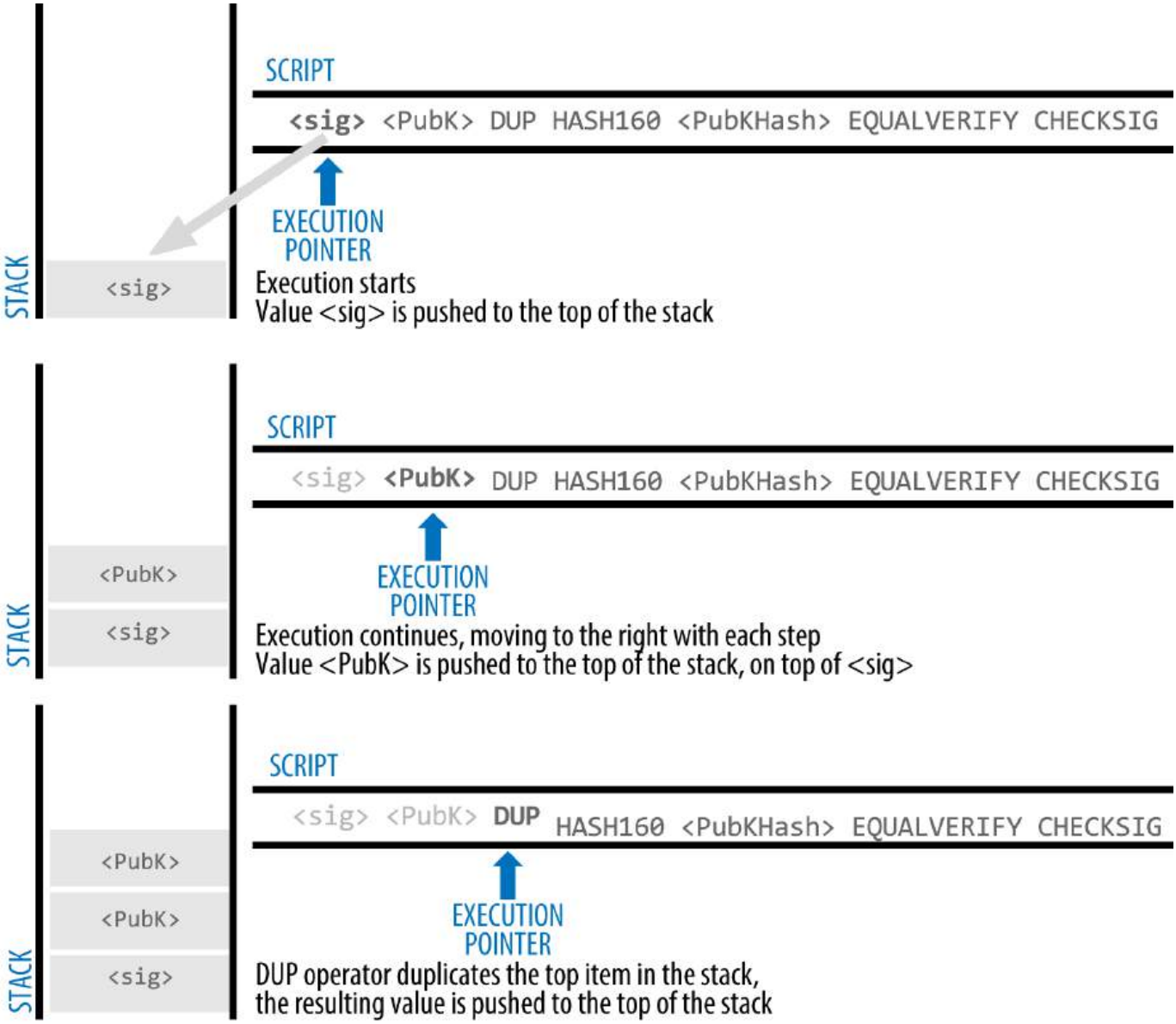


Figure 3. Evaluating a script for a P2PKH transaction (Part 1 of 2)

Pay-to-Public-Key

Pay-to-public-key is a simpler form of a bitcoin payment than pay-to-public-key-hash. With this script form, the public key itself is stored in the locking script, rather than a public-key-hash as with P2PKH earlier, which is much shorter. Pay-to-public-key-hash was invented by Satoshi to make bitcoin addresses shorter, for ease of use. Pay-to-public-key is now most often seen in coinbase transactions, generated by older mining software that has not been updated to use P2PKH.

A pay-to-public-key locking script looks like this:

```
<Public Key A> OP_CHECKSIG
```

The corresponding unlocking script that must be presented to unlock this type of output is a simple

signature, like this:

```
<Signature from Private Key A>
```

The combined script, which is validated by the transaction validation software, is:

```
<Signature from Private Key A> <Public Key A> OP_CHECKSIG
```

This script is a simple invocation of the CHECKSIG operator, which validates the signature as belonging to the correct key and returns TRUE on the stack.

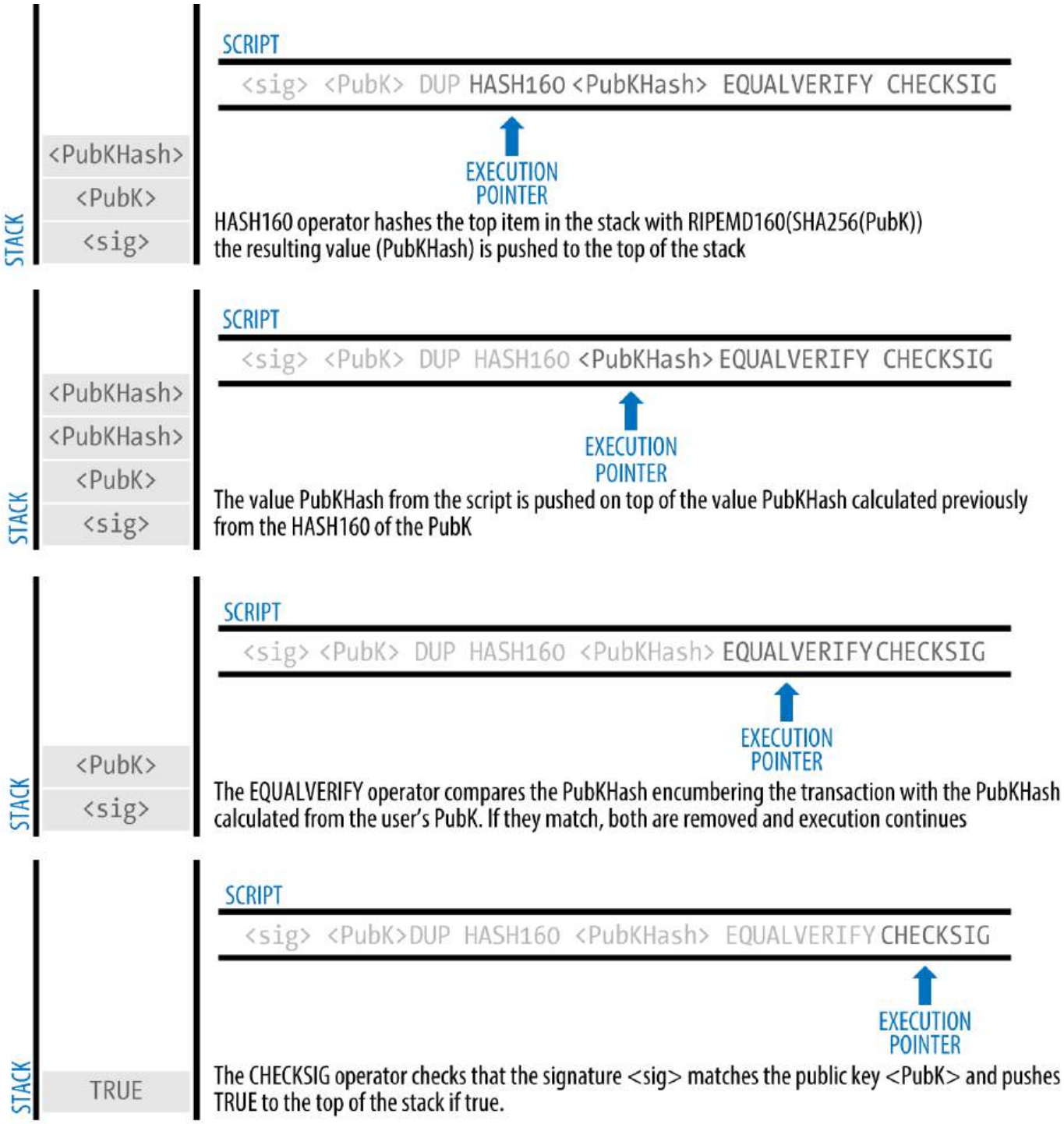


Figure 4. Evaluating a script for a P2PKH transaction (Part 2 of 2)

Multi-Signature

Multi-signature scripts set a condition where N public keys are recorded in the script and at least M of those must provide signatures to release the encumbrance. 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 multi-signature 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 multi-signature scripts are limited to at most 15 listed public keys, meaning you can do

anything from a 1-of-1 to a 15-of-15 multi-signature or any combination within that range. The limitation to 15 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.

The general form of a locking script setting an M-of-N multi-signature condition is:

```
M <Public Key 1> <Public Key 2> ... <Public Key N> N OP_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 multi-signature condition looks like this:

```
2 <Public Key A> <Public Key B> <Public Key C> 3 OP_CHECKMULTISIG
```

The preceding locking script can be satisfied with an unlocking script containing pairs of signatures and public keys:

```
OP_0 <Signature B> <Signature C>
```

or any combination of two signatures from the private keys corresponding to the three listed public keys.

NOTE The prefix OP_0 is required because of a bug in the original implementation of CHECKMULTISIG where one item too many is popped off the stack. It is ignored by CHECKMULTISIG and is simply a placeholder.

The two scripts together would form the combined validation script:

```
OP_0 <Signature B> <Signature C> 2 <Public Key A> <Public Key B> <Public Key C> 3
OP_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.

Data Output (OP_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 non-payment 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 free-form 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 OP_RETURN operator. OP_RETURN allows developers to add 80 bytes of nonpayment data to a transaction output. However, unlike the use of "fake" UTXO, the OP_RETURN operator creates an explicitly *provably unspendable* output, which does not need to be stored in the UTXO set. OP_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.

Les scripts OP_RETURN ressemblent à ceci :

```
OP_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 digital notarization service uses the 8-byte prefix "DOCPROOF," which is ASCII encoded as 44f4350524f4f46 in hexadecimal.

Keep in mind that there is no "unlocking script" that corresponds to OP_RETURN that could possibly be used to "spend" an OP_RETURN output. The whole point of OP_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—OP_RETURN is *provably un-spendable*. OP_RETURN is usually an output with a zero bitcoin amount, because any bitcoin assigned to such an output is effectively lost forever. If an OP_RETURN is encountered by the script validation software, it results immediately in halting the execution of the validation script and marking the transaction as invalid. Thus, if you accidentally reference an OP_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 OP_RETURN output. However, a single OP_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 OP_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 OP_RETURN data, 40 bytes by default.

NOTE	OP_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 OP_RETURN, or only relay and mine OP_RETURN containing less than 80 bytes of data.

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 [ch01_intro_what_is_bitcoin] we introduced Mohammed, an electronics importer based in Dubai. Mohammed's company uses bitcoin's multi-signature feature extensively for its corporate accounts. Multi-signature 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 multi-signature script for all customer payments, known in accounting terms as "accounts receivable," or AR. With the multi-signature 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 multi-signature scheme like that offers corporate governance controls and protects against theft, embezzlement, or loss.

Le script résultant est assez long et ressemble à ceci:

```
2 <Mohammed's Public Key> <Partner1 Public Key> <Partner2 Public Key> <Partner3 Public
Key> <Attorney Public Key> 5 OP_CHECKMULTISIG
```

Although multi-signature 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 output scripts difficult in practice.

Pay-to-script-hash (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. Script complexe sans P2SH shows the script without P2SH and Script complexe avec P2SH shows the same script encoded with P2SH.

Table 4. Script complexe sans P2SH

Locking Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Unlocking Script	Sig1 Sig2

Table 5. Script complexe avec P2SH

Redeem Script	2 PubKey1 PubKey2 PubKey3 PubKey4 PubKey5 5 OP_CHECKMULTISIG
Locking Script	OP_HASH160 <20-byte hash of redeem script> OP_EQUAL
Unlocking Script	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 to the recipient (spender) of the transaction.

Let's look at Mohammed's company, the complex multi-signature script, and the resulting P2SH scripts.

First, the multi-signature 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 OP_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
04C16B8698A9ABF84250A7C3EA7EEDEF9897D1C8C6ADF47F06CF73370D74DCCA01CDCA79DCC5C395D7EEC6984
D83F1F50C900A24DD47F569FD4193AF5DE762C58704A2192968D8655D6A935BEAF2CA23E3FB87A3495E7AF308
EDF08DAC3C1FCBFC2C75B4B0F4D0B1B70CD2423657738C0C2B1D5CE65C97D78D0E34224858008E8B49047E632
48B75DB7379BE9CDA8CE5751D16485F431E46117B9D0C1837C9D5737812F393DA7D4420D7E1A9162F0279CFC1
0F1E8E8F3020DECDBC3C0DD389D99779650421D65CBD7149B255382ED7F78E946580657EE6FDA162A187543A9
D85BAAA93A4AB3A8F044DADA618D087227440645ABE8A35DA8C5B73997AD343BE5C2AFD94A5043752580AFA1E
CED3C68D446BCAB69AC0BA7DF50D56231BE0AABF1FDEEC78A6A45E394BA29A1EDF518C022DD618DA774D207D1
37AAB59E0B000EB7ED238F4D800 5 OP_CHECKMULTISIG
```

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. The 20-byte hash of the preceding script is:

```
54c557e07dde5bb6cb791c7a540e0a4796f5e97e
```

A P2SH transaction locks the output to this hash instead of the longer script, using the locking script:

```
OP_HASH160 54c557e07dde5bb6cb791c7a540e0a4796f5e97e OP_EQUAL
```

which, as you can see, is much shorter. Instead of "pay to this 5-key multi-signature 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 wants 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 OP_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 OP_CHECKMULTISIG> OP_HASH160 <redeem scriptHash> OP_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 OP_CHECKMULTISIG
```

Adresses Pay-to-script-hash

Another important part of the P2SH feature is the ability to encode a script hash as an address, as defined in BIP0013. 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. 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.

Les adresses P2SH cache toute la complexité pour que la personne qui fait un paiement ne voit pas le script.

Bénéfices de Pay-to-script-hash

The pay-to-script-hash feature offers the following benefits compared to the direct use of complex scripts in locking outputs:

- Les scripts complexes sont remplacés par des empreintes plus petites dans la sortie de la transaction diminuant sa taille.
- Scripts can be coded as an address, so the sender and the sender's wallet don't need complex engineering to implement P2SH.
- P2SH transfère le travail de la construction du script au receiveur et non à l'envoyeur.
- P2SH shifts the burden in data storage for the long script from the output (which is in the UTXO set) to the input (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 transaction fee cost of a long script from the sender to the recipient, who has to include the long redeem script to spend it.

Redeem script and isStandard 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 multi-sig nature, excluding OP_RETURN and P2SH itself.

Dans la version 0.9.2 du client Bitcoin Core, les transactions P2SH peuvent contenir n'importe quel script valide, rendant le standard P2SH bien plus flexible et permettant l'expérimentation de nouveaux et complexes types de transactions.

Note that you are not able to put a P2SH inside a P2SH redeem script, because the P2SH specification is not recursive. You are also still not able to use OP_RETURN in a redeem script because OP_RETURN cannot be redeemed by definition.

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 transaction it will be processed regardless. 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 encumbrance even if it corresponds to an invalid redeem script, because the script hash gives no indication of the script it represents.

WARNING

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.

Le réseau Bitcoin

L’architecture réseau pair-à-pair

Bitcoin est une architecture réseau pair-à-pair au-dessus d’Internet. Le terme pair-à-pair, ou P2P, signifie que les ordinateurs qui participent au réseau sont les pairs les uns des autres, qu’ils sont tous égaux, qu’il n’y a pas de noeuds "spéciaux", et que tous les noeuds ont à charge de fournir des services réseau. Les noeuds du réseau s’interconnectent en un réseau maillé (mesh network) avec une topologie "plate". Il n’y a pas de serveur, pas de service centralisé, et pas de hiérarchie au sein du réseau. Les noeuds d’un réseau pair-à-pair à la fois fournissent et consomment des services, la réciprocité agissant comme une motivation pour participer. Les réseaux pair-à-pair sont intrinsèquement résilients, décentralisés, et ouverts. L’exemple par excellence d’une architecture réseau P2P était l’Internet lui-même à ces débuts, où les noeuds sur le réseau IP étaient égaux. Aujourd’hui, l’architecture d’Internet est plus hiérarchique, mais le Protocole Internet (IP) conserve sa topologie plate. Au-delà du bitcoin, l’application la plus large et fructueuse des technologies P2P est le partage de fichiers avec Napster en tant que pionnier et BitTorrent comme la plus récente évolution de l’architecture.

L’architecture réseau P2P Bitcoin est bien plus qu’un choix de topologie. Bitcoin fut conçu comme un système pair-à-pair d’argent digital, et son architecture réseau est à la fois le reflet et le fondement de cette caractéristique essentielle. La décentralisation du contrôle est un principe fondamentale du design et ceci ne pouvait être réalisé et maintenu que par un consensus de réseau P2P décentralisé et plat.

Le terme "réseau bitcoin" fait référence à l’ensemble des noeuds exécutant le protocole P2P bitcoin. En plus du protocole P2P bitcoin, il existe d’autres protocoles tels que Stratum, lesquels sont utilisés pour le minage et les portefeuilles légers ou mobiles. Ces protocoles additionnels sont pourvus par des serveurs de passerelle (gateway routing servers) qui accèdent au réseau bitcoin en utilisant le protocole P2P bitcoin, puis étendent ce réseau aux noeuds exécutant d’autres protocoles. Par exemple, les serveurs Stratum connectent les noeuds de minage Stratum via le protocole Stratum sur le réseau bitcoin principal et rattachent le protocole Stratum au protocole P2P bitcoin. On utilise le terme de "réseau bitcoin étendu" pour faire référence à l’ensemble du réseau, lequel inclu le protocole P2P bitcoin, les protocoles servant au minage (pool-mining protocols), le protocole Stratum, et tout autres protocoles reliant les différents composants du système bitcoin.

Types et rôles des noeuds

Bien que les noeuds du réseau P2P bitcoin soient égaux, ils peuvent avoir différents rôles selon leur fonction. Un nœud bitcoin est un ensemble de fonctions : routage, base de donnée blockchain, minage, et portefeuille. Un « nœud complet » (full node) possédant ces quatre fonctions est représenté dans Un nœud du réseau bitcoin possédant les quatre fonctions : portefeuille, mineur, base de données blockchain complète, et routage.

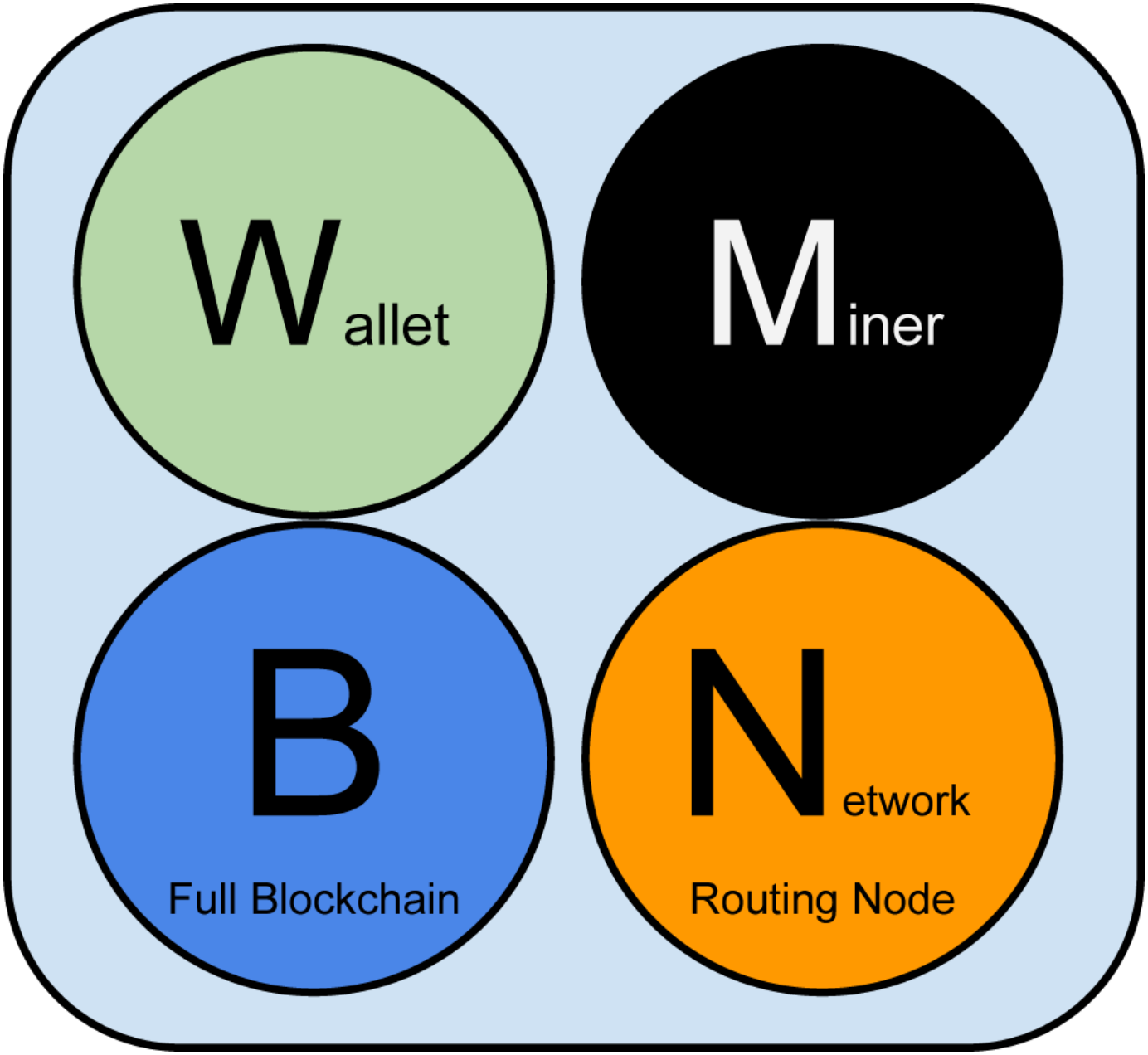


Figure 1. Un nœud du réseau bitcoin possédant les quatre fonctions : portefeuille, mineur, base de données blockchain complète, et routage

Tous les noeuds comprennent la fonction de routage pour faire partie du réseau et peuvent avoir d'autres fonctionnalités. Tous les noeuds valident et propagent transactions et blocs, découvrent et maintiennent les connections aux pairs. Dans l'exemple du nœud complet dans Un nœud du réseau bitcoin possédant les quatre fonctions : portefeuille, mineur, base de données blockchain complète, et routage, la fonction de routage est indiquée par un cercle orange nommé "Noeud de routage réseau."

Certains noeuds, appelés noeuds complets, conservent aussi une copie complète et à jour de la blockchain. Les noeuds complets peuvent vérifier de façon autonome et avec autorité n'importe quelle transaction sans référence externe. Quelques noeuds conservent seulement un sous-ensemble de la blockchain et vérifient les transactions en utilisant une méthode appelée *vérification de paiement simplifiée*, ou SPV (simplified payment verification). Ces noeuds sont connus comme SPV ou noeuds légers. Dans l'exemple du nœud complet sur le schéma, la fonction de base de données blockchain du

nœud complet est indiquée par un cercle bleu nommé "Blockchain complète." Dans Le réseau bitcoin étendu montrant les différents types de nœuds, les passerelles et les protocoles, les noeuds SPV sont représentés sans cercle bleu, indiquant qu'ils ne possèdent pas une copie complète de la blockchain.

Les noeuds de minage rivalisent entre eux pour créer de nouveaux blocs, utilisant du matériel informatique spécialisé en vue de résoudre l'algorithme preuve-de-travail. Certains noeuds de minage sont aussi des noeuds complets, conservant une copie complète de la blockchain, pendant que d'autres sont des noeuds légers faisant partie d'une pool de minage et dépendent d'un serveur de pool pour maintenir un nœud complet. La fonction de minage est représentée dans le nœud complet par un cercle noir nommé "Mineur."

Les portefeuilles utilisateur peuvent faire partie d'un nœud complet, comme c'est généralement le cas avec les clients bitcoin sur les ordinateurs de bureau. De plus en plus, de nombreux portefeuilles utilisateur, spécialement ceux s'exécutant sur des appareils aux ressources limitées tels que les smartphones, sont des noeuds SPV. La fonction de portefeuille est représentée dans Un nœud du réseau bitcoin possédant les quatre fonctions : portefeuille, mineur, base de données blockchain complète, et routage par un cercle vert nommé "Portefeuille".

Outre les principaux types de noeuds sur le protocole bitcoin P2P, il y a des serveurs et des noeuds utilisant d'autres protocoles, tels que les protocoles spécialisés de pool de minage et les protocoles de clients légers et d'accès.

Les différents types de nœuds sur le réseau bitcoin étendu représente les types de noeuds les plus communs sur le réseau étendu bitcoin.

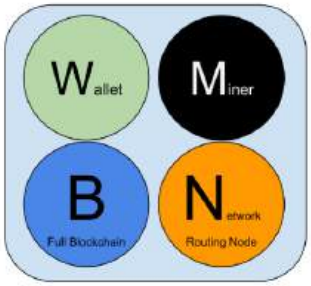
Le réseau étendu bitcoin

Le réseau bitcoin principal, utilisant le protocole P2P bitcoin, est composé d'environ 7000 à 10 000 noeuds en écoute, exécutant divers versions du client de référence bitcoin (Bitcoin Core) et quelques centaines de noeuds exécutant divers autres implementations du protocole P2P bitcoin, telles que BitcoinJ, Libbitcoin, et btcd. Un faible pourcentage des noeuds sur le réseau P2P bitcoin sont aussi des noeuds de minage, rivalisant dans le processus de minage, validant les transactions, et créant de nouveaux blocs. Divers grandes entreprises s'interfacent avec le réseau bitcoin en exécutant des noeuds complets, clients basés sur le client Bitcoin Core, avec des copies complètes de la blockchain et un nœud de réseau, mais sans les fonctions de minage ou de portefeuille. Ces noeuds agissent comme des routeurs de périphérie (network edge routers), permettant à divers autres services (places de marché, portefeuilles, explorateurs de blocs, traitement des paiements marchand) de bâtir par-dessus.

Le réseau bitcoin étendu inclut le réseau exécutant le protocole P2P bitcoin, décrit précédemment, de même que les noeuds exécutant des protocoles spécialisés. Rattaché au réseau P2P bitcoin principal, nous avons un certain nombre de serveurs de pool et de passerelles de protocole (protocol gateways) qui connectent les noeuds exécutant d'autres protocoles. Ces autres noeuds de protocole sont pour la plupart des noeuds de minage en pool (see [ch8]) et des portefeuilles-clients léger, qui ne conservent pas une copie complète de la blockchain.

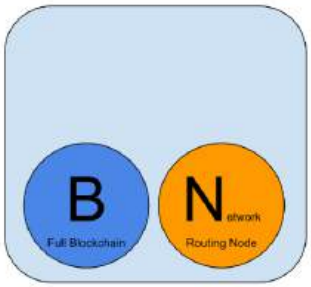
Le réseau bitcoin étendu montrant les différents types de nœuds, les passerelles et les protocoles

représente le réseau bitcoin étendu avec les divers types de noeuds, les serveurs passerelle (gateway servers), les routeurs de périphérie (edge routers), et les clients-portefeuille ainsi que les divers protocoles qu'ils utilisent pour se connecter les uns aux autres.



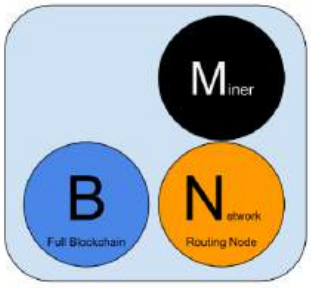
Reference Client (Bitcoin Core)

Contains a Wallet, Miner, full Blockchain database, and Network routing node on the bitcoin P2P network.



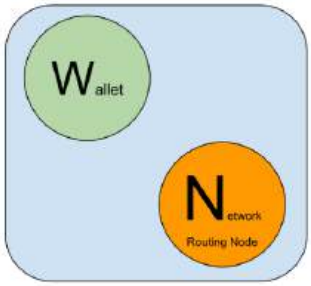
Full Block Chain Node

Contains a full Blockchain database, and Network routing node on the bitcoin P2P network.



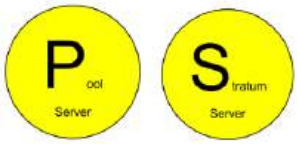
Solo Miner

Contains a mining function with a full copy of the blockchain and a bitcoin P2P network routing node.



Lightweight (SPV) wallet

Contains a Wallet and a Network node on the bitcoin P2P protocol, without a blockchain.

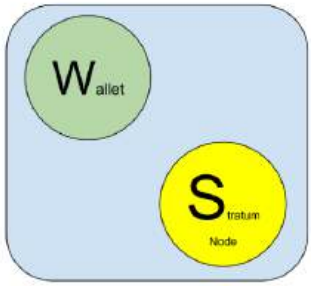


Pool Protocol Servers

Gateway routers connecting the bitcoin P2P network to nodes running other protocols such as pool mining nodes or Stratum nodes.

Mining Nodes

Contain a mining function, without a blockchain, with the Stratum protocol node (S) or other pool (P) mining protocol node.



Lightweight (SPV) Stratum wallet

Contains a Wallet and a Network node on the Stratum protocol, without a blockchain.

Figure 2. Les différents types de nœuds sur le réseau bitcoin étendu

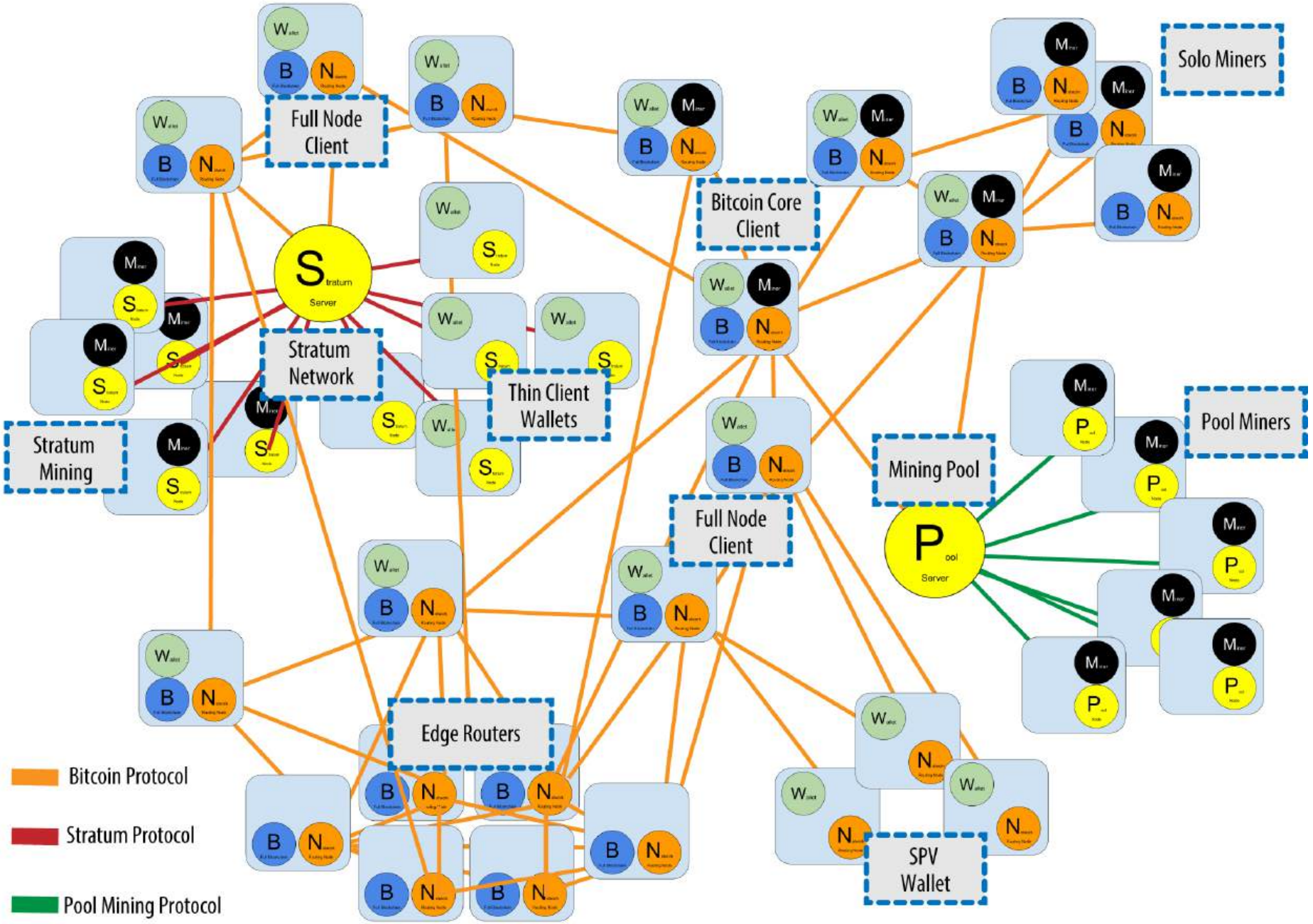

Full Node Client
Solo Miners
Bitcoin Core Client
Stratum Server
Stratum Network
Thin Client Wallets
Stratum Mining
Mining Pool
Pool Miners
Pool Server
Full Node Client
Edge Routers
SPV Wallet
Bitcoin Protocol
Stratum Protocol
Pool Mining Protocol

Figure 3. Le réseau bitcoin étendu montrant les différents types de nœuds, les passerelles et les protocoles

Découverte du réseau

Lorsqu'un un nouveau node démarre, il doit découvrir les autres noeuds sur le réseau afin d'y prendre part. Pour commencer ce processus, un nouveau nœud doit découvrir au moins un autre nœud existant sur ce réseau et s'y connecter. La position géographique des autres noeuds n'est pas importante; la topologie du réseau bitcoin n'étant pas géographiquement définie. Par conséquent, n'importe quel nœud bitcoin existant peut être sélectionné au hasard.

Pour se connecter à un pair connu, les noeuds établissent une connexion TCP, habituellement sur le port 8333 (le port généralement connu comme étant celui utilisé par bitcoin), ou un port alternatif si disponible. Lors de la connexion, le nœud va démarrer une prise de contact ou "handshake" (see La prise de contact initiale entre pairs) en transmettant un message de version, lequel contient des informations basiques d'identification, incluant :

PROTOCOL_VERSION
: Une constante qui définie la version du protocole P2P bitcoin que le client "parle" (e.g., 70002)

nLocalServices
: Une liste de services locaux supportés par le nœud, actuellement juste NODE_NETWORK

nTime
: La date actuelle

addrYou
: L'adresse IP du nœud distant telle que vue à partir de ce nœud

addrMe
: L'adresse IP du nœud local, telle que découverte par le nœud local

subver
: Une sous-version montrant le type de logiciel exécuté sur ce nœud (e.g., "/Satoshi:0.9.2.1/")+

BestHeight
: La hauteur du bloc dans la blockchain de ce nœud

(Voir GitHub pour un exemple du message réseau de la version.)

Le nœud pair répond avec + verack + pour reconnaître et établir une connexion, et envoie éventuellement son propre message de version s'il souhaite amorcer une connexion et se connecter à son tour comme un pair.

Comment un nouveau nœud trouve des pairs ? La première méthode consiste à interroger un DNS en utilisant un certain nombre de graines DNS (DNS seeds), des serveurs DNS qui fournissent une liste

d’adresses IP de noeuds bitcoin. Certaines de ces graines DNS fournissent une liste statique des adresses IP des noeuds bitcoin stable en écoute. D’autres sont des implémentations personnalisées de BIND (Berkeley Internet Name Daemon) qui renvoient un sous-ensemble aléatoire d’adresses à partir d’une liste d’adresses de nœuds bitcoin recueillies par un robot ou d’un nœud bitcoin en activité depuis longtemps. Le client Bitcoin Core contient les noms de cinq graines DNS différentes. Le fait que les propriétaires et l’implémentation des différentes graines DNS soient très variés permet un haut niveau de fiabilité dans le processus d’amorçage initial. Dans le client Bitcoin Core, la possibilité d’utiliser les graines DNS est contrôlé par l’option dnsseed (défini à 1 par défaut, pour utiliser la graine DNS).

Autrement, on doit donner à un nœud d’amorçage qui ne sait rien du réseau l’adresse IP d’au moins un nœud bitcoin, après quoi il peut établir des connexions par le biais de nouvelles présentations. L’argument de ligne de commande -seednode peut être utilisé pour se connecter à un nœud juste pour les présentations, l’utilisant comme une graine. Après que le nœud-graine initial soit utilisé pour former des présentations, le client va se déconnecter et utiliser les pairs nouvellement découverts.

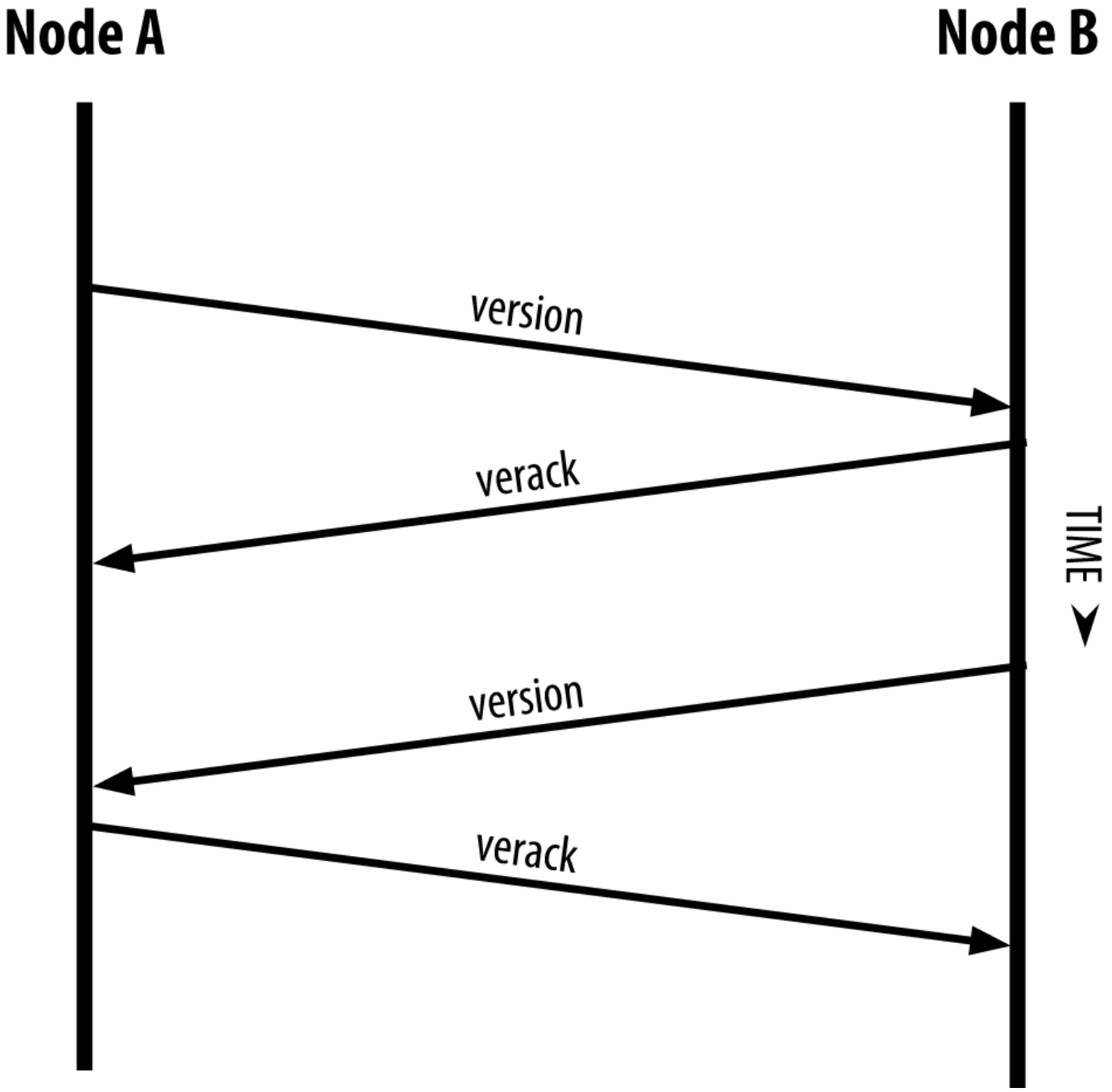


Figure 4. La prise de contact initiale entre pairs

Une fois qu'une ou plusieurs connections ont été établies, le nouveau nœud va envoyé à ses voisins un message addr contenant sa propre adresse IP. Les voisins vont, en retour, faire suivre le message addr à leurs voisins, assurant ainsi que le nœud nouvellement connecté deviennent bien connu et mieux connecté. En outre, le nœud nouvellement connecté peut envoyé getaddr à ses voisins, leur demandant de renvoyer une liste d'adresses IP d'autres pairs. De la sorte, un nœud peut trouver des pairs avec qui se connecter et annoncer son existence sur le réseau pour que d'autres nœud puisse le trouver. La propagation et la découverte des adresses représente le protocole de découverte d'adresse.

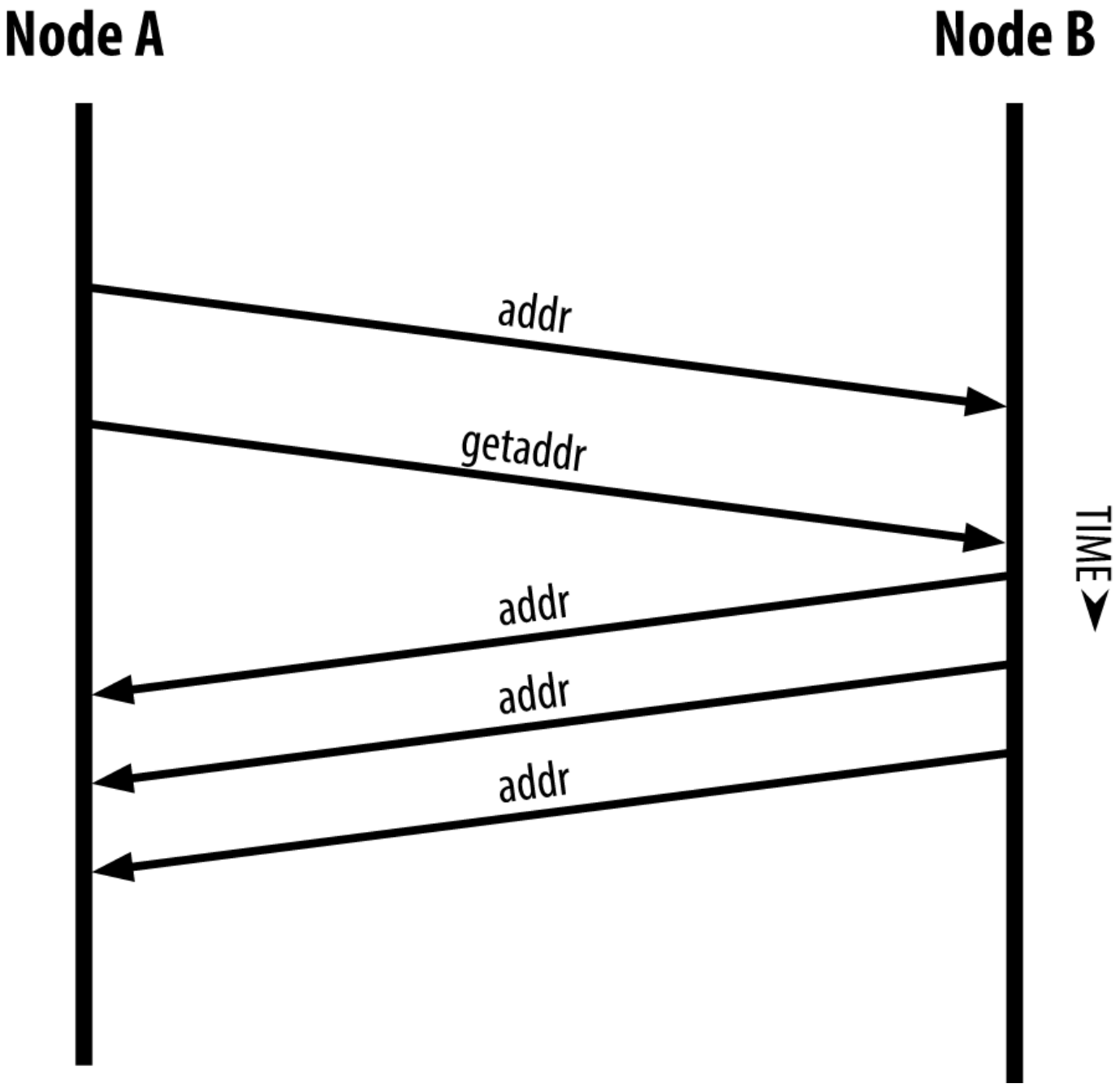


Figure 5. La propagation et la découverte des adresses

Un nœud doit se connecter à quelques pairs différents afin d'établir divers chemins au sein du réseau bitcoin. Les chemins ne sont pas fiables – les noeuds vont et viennent – et par conséquent le nœud doit continuer à découvrir de nouveaux noeuds à mesure qu'il perd les anciennes connections de même qu'il aidera les autres noeuds lorsqu'ils s'initialisent. Une seule connection est nécessaire au démarrage, parce que le premier nœud peut proposer des présentations à ces noeuds pairs et ces noeuds pairs peuvent proposer à nouveau des présentations. Il est aussi inutile et coûteux en terme de ressources réseau de se connecter à plus d'une poignée de noeuds. Après le démarrage, un nœud va se rappeler ses connections réussies les plus récentes, afin que s'il redémarre il puisse rapidement rétablir les connections avec son précédent réseau de pairs. Si aucun de ces anciens pairs ne répond à sa requête de connection, le nœud peut utiliser les noeuds graine pour démarrer de nouveau.

Sur un nœud exécutant le client Bitcoin Core, vous pouvez lister les connexions des pairs avec la commande 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
    }
]
```

Pour remplacer la gestion automatique des pairs et spécifier une liste d'adresses IP, les utilisateurs peuvent fournir l'option -connect=<IPAddress> et spécifier une ou plusieurs adresses IP. Si cette option est utilisée, le nœud se connectera uniquement aux adresses IP sélectionnées, au lieu de découvrir et de maintenir les connexions pairs automatiquement.

S’il n’y a pas de trafic sur une connexion, les nœuds vont envoyer régulièrement un message afin de la maintenir. Si un nœud n’a pas communiqué sur une connexion pendant plus de 90 minutes, on présume qu’elle est interrompue et un nouveau pair sera recherché. Ainsi, le réseau s’ajuste dynamiquement aux nœuds éphémères et aux problèmes de réseau, et peut organiquement grandir et rétrécir comme bon lui semble sans aucun contrôle central.

Les noeuds complets

Les nœuds complets sont des noeuds qui maintiennent une blockchain complète avec toutes les transactions. De façon plus fidèle, on devrait probablement les appeler "noeuds à blockchain complète". Dans les premières années de bitcoin, tous les nœuds étaient des nœuds complets et actuellement le client Bitcoin Core est un nœud à blockchain complète. Au cours des deux dernières années, cependant, de nouvelles formes de clients bitcoin ont été introduites, lesquels ne maintient pas de blockchain complète, mais se présentent comme des clients légers. Nous allons les examiner plus en détail dans la section suivante.

Les nœuds à blockchain complète entretiennent une copie complète et à jour de la blockchain bitcoin avec toutes les transactions, qu’ils construisent et vérifient de façon indépendante, en commençant par le premier bloc (bloc de genèse) et ce jusqu’à au dernier bloc connu sur le réseau. Un nœud à blockchain complète peut indépendamment et avec autorité vérifier toute transaction sans avoir recours à quiconque ni dépendre d’un autre nœud ou d’une source d’information. Le nœud à blockchain complète s’appui sur le réseau pour recevoir des mises à jour sur les nouveaux blocs de transactions, lesquels sont ensuite vérifiés et intégrés à sa copie locale de la blockchain.

Exécuter un nœud à blockchain complète vous procure l’expérience bitcoin la plus pure : une vérification indépendante de toutes les transactions sans avoir besoin de s’appuyer sur, ou faire confiance à, n’importe quels autres systèmes. Il est facile de dire si vous utilisez un nœud complet car il nécessite plus de 20 giga-octets de stockage persistant (espace disque) afin de stocker la blockchain dans son intégralité. Si vous avez besoin de beaucoup d’espace disque et qu’il vous faut deux à trois jours pour vous synchroniser au réseau, vous utilisez un nœud complet. Ceci est le prix à payer pour une complète indépendance et une entière liberté vis-à-vis d’une autorité centrale.

Il existe quelques implémentations alternatives aux clients bitcoin à blockchain complète, créées avec différents languages de programmation et architectures logicielles. Cependant, l’implémentation la plus commune est le client de référence Bitcoin Core, également connu sous le nom de client Satoshi. Plus de 90% des noeuds sur le réseau bitcoin exécutent diverses versions de Bitcoin Core. Il est identifié comme "Satoshi" dans la sous-version envoyée dans le message version et affiché par la commande getpeerinfo comme vu précédemment; par exemple, /Satoshi:0.8.6/.

Echanger l'"Inventaire"

La première chose qu’un noeud complet va faire une fois connecté à ses pairs est d’essayer de construire une blockchain complète. Si c’est un tout nouveau nœud et qu’il n’a pas de blockchain, il ne connaît qu’un bloc, le bloc de genèse, qui est constamment intégré dans le logiciel client. Commençant par le bloc #0 (le bloc de la genèse), le nouveau nœud devra télécharger des centaines de milliers de

blocs pour se synchroniser avec le réseau et rétablir une blockchain complète.

Le processus de synchronisation de la blockchain commence avec le message version, parce qu'il contient BestHeight, la hauteur actuelle de la blockchain d'un nœud (nombre de blocs). Un noeud voyant les messages version de ses pairs, saura combien de blocs chacun possède, et sera en mesure de comparer cela au nombre de blocs qu'il a dans sa propre blockchain. Les nœuds pairés échangeront un message getblocks qui contient le hash (empreinte digitale) du bloc au sommet de leur blockchain locale. Un des pairs sera en mesure d'identifier le hash reçu comme appartenant à un bloc qui n'est pas au sommet, mais qui appartient plutôt à un bloc plus ancien, pour en déduire que sa propre blockchain locale est plus longue que celle de son pair.

Le pair qui a la plus longue blockchain a plus de blocs que l'autre noeud et est en mesure d'identifier quels blocs les autres noeuds ont besoin de "rattraper". Il identifiera les 500 premiers blocs à partager et transmettra leurs hash en utilisant un message inv (inventaire). Le nœud n'ayant pas ces blocs pourra donc les récupérer, ceci en émettant une série de messages getdata demandant les données complètes du bloc et en identifiant les blocs demandés avec les hash obtenus par le message inv.

Supposons, par exemple, qu'un noeud ne possède que le bloc de genèse. Il recevra alors un message inv de ses pairs contenant les hash des 500 blocs suivants dans la chaîne. Il commencera alors à demander des blocs à tout ses pairs connectés, répartissant la charge et assurant qu'il ne submerge aucun pair de ses sollicitations. Le nœud garde la trace du nombre de blocs «en transit» par connexion de pair, c'est-à-dire les blocs qu'il a demandé mais pas reçus, en vérifiant que l'on ne dépasse pas une limite (MAX_BLOCKS_IN_TRANSIT_PER_PEER). De cette façon, s'il a besoin de beaucoup de blocs, il ne demandera que les nouveaux à mesure que les requêtes précédentes soient satisfaites, permettant aux pairs de contrôler le rythme des mises à jour et de ne pas surcharger le réseau. À mesure que chaque bloc est reçu, il est ajouté à la blockchain, comme nous allons le voir dans [blockchain]. À mesure que la blockchain locale se construit, plus de blocs sont demandés et reçus, et le processus continue jusqu'à ce que le noeud rattrape le reste du réseau.

Ce processus de comparaison de la blockchain locale avec les pairs et de récupération de tous les blocs manquants arrive chaque fois qu'un noeud passe hors ligne durant un certain laps de temps. Qu'un noeud ait été déconnecté pendant quelques minutes et auquel il manque quelques blocs, ou pendant un mois et a besoin de quelques milliers de blocs, il commence par l'envoi de getblocks, obtient une réponse inv et commence à télécharger les blocs manquants. Noeud synchronisant la blockchain en récupérant les blocs d'un pair représente le protocole d'inventaire et de propagation de bloc.

Les noeuds de Vérification de Paiement Simplifié (SPV)

Tous les nœuds n'ont pas la capacité de stocker la blockchain dans son intégralité. Beaucoup de clients bitcoin sont conçus pour fonctionner sur des appareils à l'espace - et à la puissance - limités, tels que smartphones, tablettes ou systèmes embarqués. Pour ces appareils, une méthode de *vérification de paiement simplifiée* (SPV) est utilisé pour leur permettre de fonctionner sans conserver l'entière blockchain. Ces types de clients sont appelés clients SPV ou clients légers. À mesure que l'adoption du bitcoin croît, le nœud SPV devient la forme la plus commune de noeud bitcoin, en particulier pour les portefeuilles bitcoin.

Les nœuds SPV téléchargent uniquement les en-têtes de bloc, ignorant les transactions incluses dans chaque bloc. La chaîne de blocs obtenue, sans transactions, est 1000 fois plus petite que la blockchain complète. Les nœuds SPV ne peuvent pas construire une image complète de tous les UTXOs prêts à être dépensé parce qu'ils ne sont pas informés de toutes les transactions sur le réseau. Les nœuds SPV vérifient les transactions en utilisant une méthodologie légèrement différente qui repose sur les pairs pour leurs fournir des vues partielles de parties pertinentes de la blockchain, à la demande.

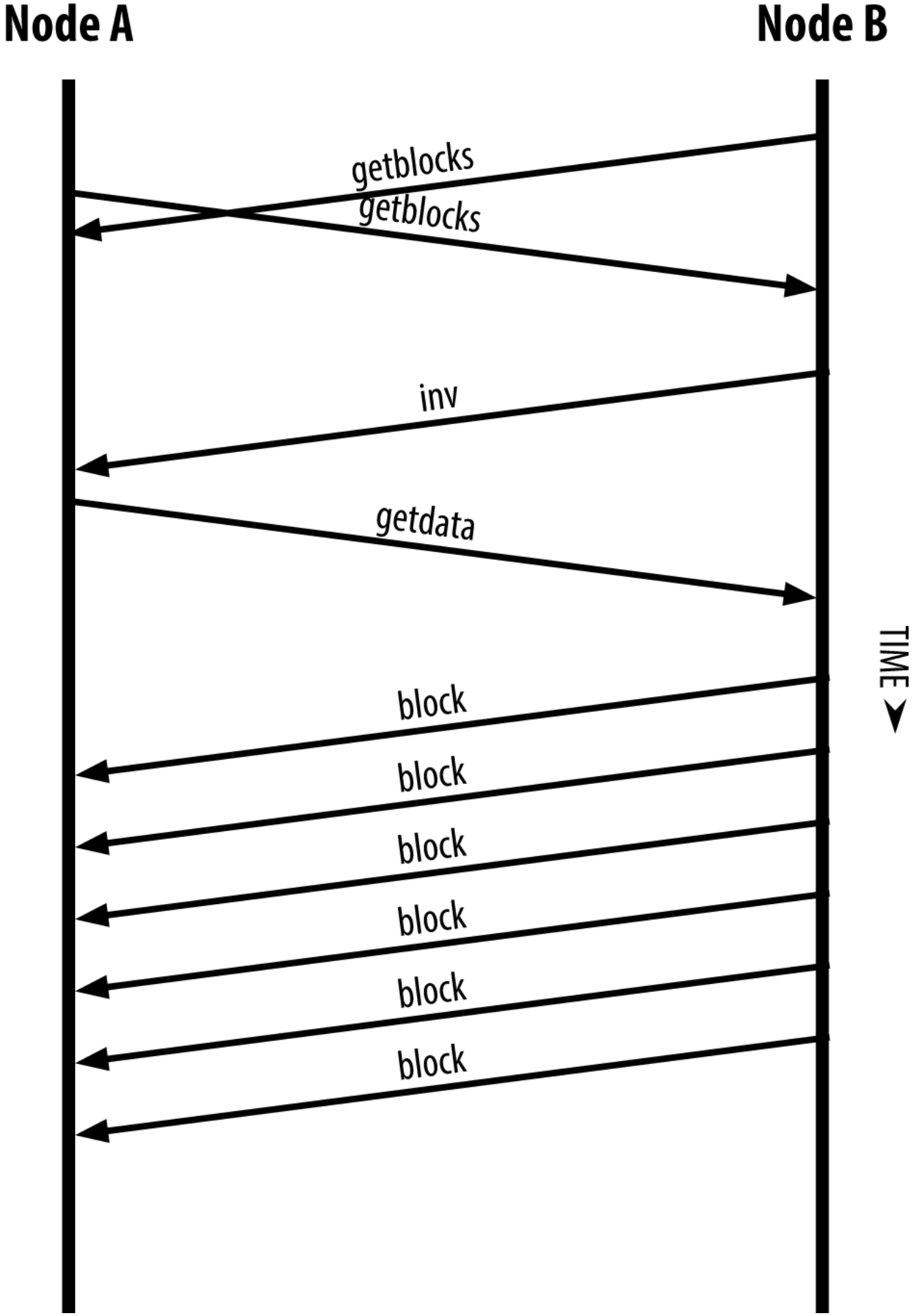

Node A
Node B
getblocks
getblocks
inv
getdata
block
block
block
block
block
block
TIME

Figure 6. Noeud synchronisant la blockchain en récupérant les blocs d'un pair

Par analogie, un noeud complet est comme un touriste dans une ville étrange, équipé d'une carte détaillée de chaque rue et de chaque adresse. En comparaison, un noeud SPV est comme un touriste dans une ville étrange demandant à chaque intersection son chemin à des étrangers pris au hasard, n'ayant connaissance que d'une avenue principale. Bien que les deux touristes soient capables de vérifier l'existence d'une rue en la visitant, le touriste sans carte ne connait aucune des allées latérales, et ne sait pas qu'il existe d'autres rues. Situé devant le 23 Church Street, le touriste sans carte ne peut pas savoir s'il y a une douzaine d'autres "23 Church Street" dans la ville et s'il s'agit de la bonne. La meilleure chose à faire pour le touriste sans carte est de demander à suffisamment de gens et espérer que certains d'entre eux n'essaient pas de l'agresser.

La vérification de paiement simplifié vérifie les transactions par référence à leur *profondeur* dans la blockchain au lieu de leur *hauteur*. Alors qu'un noeud à blockchain complète construira une chaîne entièrement vérifiée composée de milliers de blocs et transactions, descendant dans la blockchain (remontant dans le temps) jusqu'à atteindre le bloc de genèse, un noeud SPV vérifiera la chaîne de tous les blocs (mais pas de toutes les transactions) et liera cette chaîne à la transaction qui importe.

Par exemple, lorsque l'on examine une transaction dans le bloc 300 000, un noeud complet relie l'ensemble de ces 300 000 blocs jusqu'au bloc de genèse et construit une base de données complète d'UTXO, établissant la validité d'une transaction en confirmant que ces UTXO soient effectivement non dépensés. Un noeud SPV est incapable de confirmer si un UTXO demeure non dépensé. Au lieu de cela, le noeud SPV va établir un lien entre la transaction et le bloc qui la contient, utilisant un *chemin de merkle* (see [merkle_trees]). Ensuite, le noeud SPV attends de voir les six blocs 300 001 à 300 006 empilés au-dessus du bloc contenant la transaction et vérifie cela en établissant sa profondeur entre les blocs 300 006 à 300 001. Le fait que d'autres nœuds sur le réseau aient accepté le bloc 300 000 puis aient fait le travail nécessaire pour produire six autres blocs par-dessus celui-ci est la preuve, par procuration, que la transaction n'était pas une double-dépense.

Un nœud SPV ne peut pas être persuadé qu'une transaction existe dans un bloc lorsque, de fait, elle n'existe pas. Le nœud SPV établit l'existence d'une transaction dans un bloc en demandant une preuve de type chemin de Merkle et en validant la preuve de travail dans la chaîne de blocs. Toutefois, l'existence d'une transaction peut être «caché» d'un noeud SPV. Un nœud de SPV peut certainement prouver qu'une transaction existe, mais ne peut pas vérifier qu'une transaction, telle qu'une double-dépense du même UTXO, n'existe pas, car il ne dispose pas d'un registre de toutes les transactions. Cette vulnérabilité peut être utilisé dans une attaque par déni de service ou pour une attaque double-dépense contre un nœud SPV. Pour se défendre contre cela, un noeud SPV doit se connecter au hasard à plusieurs nœuds, pour augmenter la probabilité qu'il est en contact avec au moins un noeud honnête. Ce besoin de se connecter au hasard signifie que les nœuds SPV sont également vulnérables aux attaques de partitionnement réseau ou aux attaques Sybil, où ils sont reliées à de faux nœuds ou de faux réseaux et n'ont pas accès aux nœuds honnêtes ou au réseau bitcoin réel.

Pour la plupart des applications pratiques, les nœuds SPV suffisamment connectés sont assez sûr, trouvant le juste équilibre entre besoins en ressources, aspect pratique et sécurité. Pour une sécurité infaillible, cependant, rien ne vaut l'exécution d'un noeud à blockchain complète.

TIP

Un nœud à blockchain complète contrôle une transaction en passant en revue la totalité de la chaîne qui comprend plusieurs milliers de blocs sous elle afin de garantir que le UTXO n'est pas dépensé, alors qu'un nœud SPV vérifie à quelle profondeur le bloc est enfoui sous une poignée de blocs.

Pour obtenir les en-têtes de bloc, les nœuds SPV utilisent un message getheaders au lieu de getblocks. Le pair qui répondra enverra jusqu'à 2000 en-têtes de bloc en utilisant un seul message headers. Le processus est par ailleurs le même que celui utilisé par un noeud complet pour extraire des blocs complets. Les nœuds SPV définissent également un filtre sur la connexion à leurs pairs, pour filtrer le flux de futures blocs et transactions envoyés par les pairs. Toutes les transactions d'intérêt sont récupérés en utilisant une requête getdata. En réponse, le pair génère un message tx contenant les transactions. Noeud SPV synchronisant les en-têtes de bloc représente la synchronisation des en-têtes de bloc.

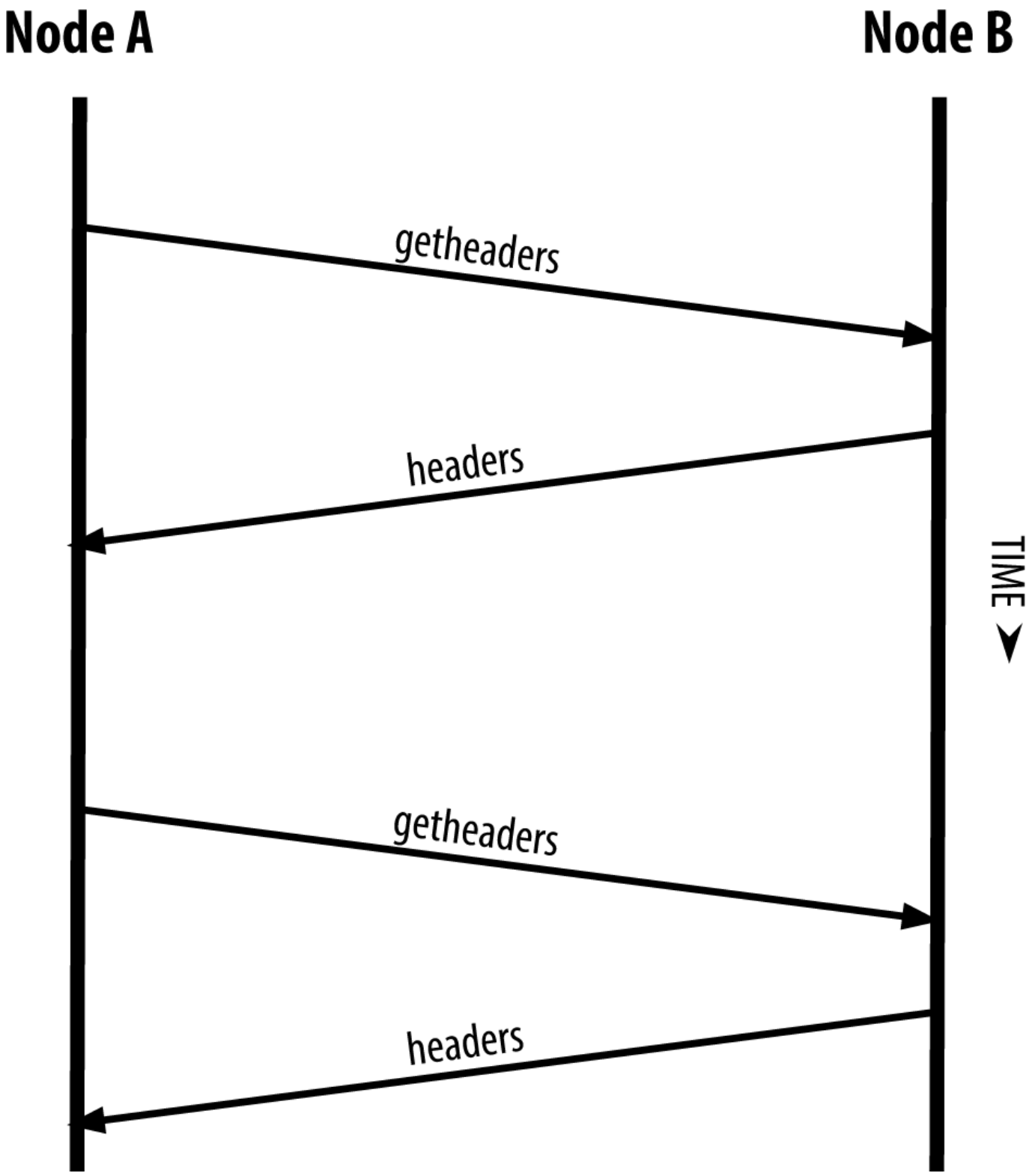


Figure 7. Noeud SPV synchronisant les en-têtes de bloc

Parce que les nœuds SPV ont besoin de récupérer des transactions spécifiques afin de les vérifier de manière sélective, ils engendrent également un risque concernant la vie privée. Contrairement aux noeuds à blockchain complète, qui collectent toutes les transactions au sein de chaque bloc, les demandes du nœud SPV pour obtenir des données spécifiques peuvent révéler, par inadvertance, les adresses contenues dans leur portefeuille. Par exemple, un tiers surveillant un réseau pourrait garder

une trace de toutes les transactions demandées par un portefeuille sur un nœud SPV et les utiliser pour associer les adresses bitcoin avec l'utilisateur de ce portefeuille, détruisant la vie privée de l'utilisateur.

Peu de temps après l'introduction de nœuds SPV/légers, les développeurs de Bitcoin ont ajouté une fonctionnalité appelée *bloom filters* pour faire face à ce risque de concernant la vie privée. Les filtres de Bloom permettent aux nœuds SPV de recevoir un sous-ensemble des transactions sans révéler précisément quelles sont les adresses qui les intéressent, à travers un mécanisme de filtrage qui s'appui sur des probabilités plutôt que sur des modèles fixes.

Les filtres de Bloom

Un filtre de bloom est un filtre de recherche probabiliste, un moyen pour décrire un pattern souhaité sans le préciser exactement. Les filtres de bloom offrent un moyen efficace pour exprimer un pattern de recherche tout en protégeant la vie privée. Ils sont utilisés par les nœuds SPV pour demander à leurs pairs des transactions correspondant à un pattern spécifique, sans révéler exactement qu'elles adresses ils recherchent.

Dans notre analogie précédente, un touriste sans carte demandait à connaître le chemin vers une adresse spécifique, "23 Church St." Si elle demande à des étrangers le chemin vers cette rue, elle révèle par inadvertance sa destination. Un filtre de bloom est comme demander, "Y a t-il dans ce quartier des rues dont le nom se termine par R-C-H ?" Une telle question en dévoile un peu moins sur la destination souhaitée que de demander "23 Church St." En utilisant cette technique, un touriste pourrait préciser l'adresse souhaitée avec plus de détails comme "se terminant par U-R-C-H" ou avec moins de détails comme "se terminant par H." En faisant varier la précision de la recherche, le touriste révèle plus ou moins d'informations, avec pour contre-partie l'obtention de résultats plus ou moins précis. Si elle demande un pattern moins précis, elle reçoit beaucoup plus d'adresses possibles et conserve sa vie privée, mais la plupart des résultats ne seront pas pertinents. Si elle demande un pattern très précis, elle obtient moins de résultats mais perd sa vie privée.

Les filtres de Bloom remplissent cette fonction en permettant à un noeud SPV de spécifier un pattern de recherche pour trouver des transactions, lequel peut être ajusté entre précision et vie privée. Un filtre de bloom plus précis produira des résultats juste, au détriment de divulguer les adresses utilisées dans le portefeuille de l'utilisateur. Un filtre de bloom moins spécifique va produire plus de données concernant plus de transactions, dont beaucoup inutile pour le noeud, mais permettra au noeud de maintenir une meilleure vie privée.

Un nœud SPV va initialiser un filtre de bloom à « vide » et dans cet état le filtre de bloom ne correspondra à aucun pattern. Le nœud SPV va ensuite faire une liste de toutes les adresses contenues dans son portefeuille et créer un pattern de recherche correspondant à la sortie de transaction (transaction output) de chaque adresse. Habituellement, le pattern de recherche est un script pay-to-public-key-hash qui est le script de verrouillage attendu qui sera présent dans toute transaction payant le public-key-hash (adresse). Autrement, si le nœud SPV traque le solde d'une adresse P2SH, le pattern de recherche sera un script pay-to-script-hash. Le noeud SPV ajoute ensuite chacun des patterns de recherche au filtre de bloom, de sorte que le filtre puisse reconnaître le pattern de recherche s'il est présent dans une transaction. Enfin, le filtre de bloom est envoyé aux pairs et les pairs utilise cela pour

trouver les transactions et les transmettre au noeud SPV.

Les filtres de Bloom sont implémentés comme un tableau de taille variable de N chiffres binaires (un champ de bits) et un nombre variable M de fonctions de hachage. Les fonctions de hachage sont conçus pour produire toujours un signal de sortie qui est compris entre 1 et N, correspondant au tableau de chiffres binaires. Les fonctions de hachage sont générés de façon déterministe, de sorte que n'importe quel noeud mettant en œuvre un filtre de bloom utilisera toujours les mêmes fonctions de hachage et obtiendra le même résultat pour une entrée spécifique. En choisissant différentes longueur (N) de filtres de bloom et un nombre différent (M) de fonctions de hachage, le filtre de bloom peut être ajusté, faisant varier le niveau de précision et donc la vie privée.

Dans Un exemple de filtre bloom simpliste, avec un champ de 16 bits et trois fonctions de hachage, nous utilisons un très petit tableau de 16 bits et un ensemble de trois fonctions de hachage afin de démontrer comment les filtres de bloom fonctionnent.

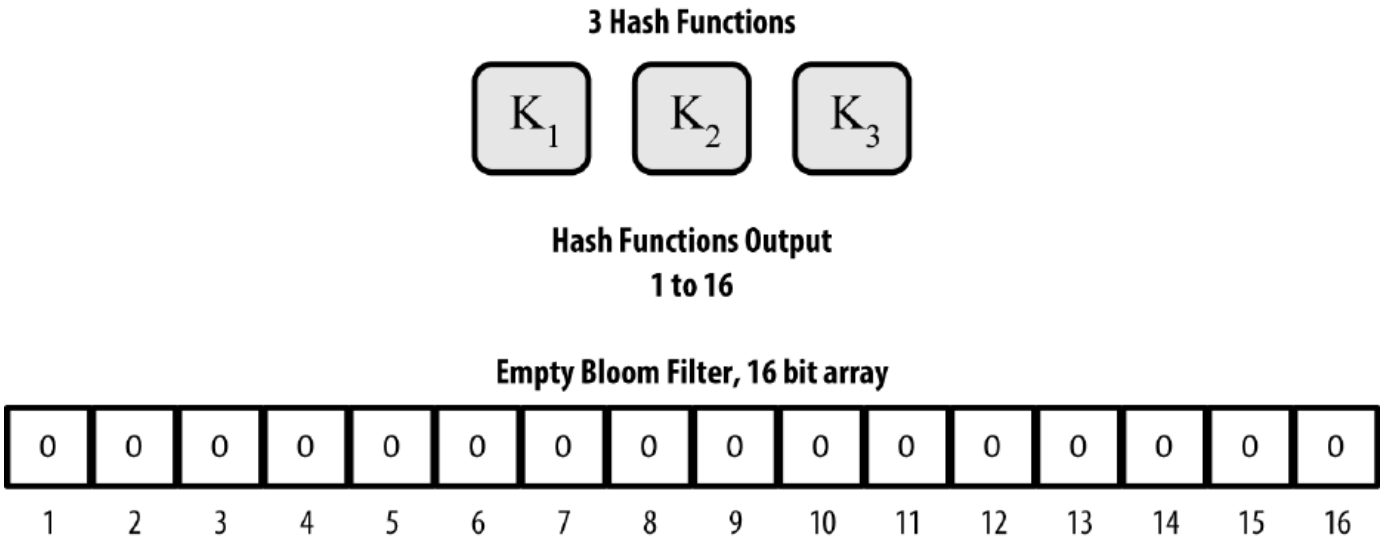


Figure 8. Un exemple de filtre bloom simpliste, avec un champ de 16 bits et trois fonctions de hachage

Le filtre de bloom est initialisée afin que les bits du tableau soient tous à zéro. Pour ajouter un pattern au filtre de bloom, le pattern est haché tour à tour par chaque fonction de hachage. Appliquer la première fonction de hachage aux entrées résulte en un nombre compris entre 1 et N. Le bit correspondant dans le tableau (indexé de 1 à N) est trouvé et mis à 1, enregistrant ainsi la sortie de la fonction de hachage. Ensuite, la fonction de hachage suivante est utilisée pour définir un autre bit et ainsi de suite. Une fois que toutes les fonctions de hachage M ont été appliquées, le pattern de recherche sera "enregistré" dans le filtre de bloom en tant que M bits qui ont été modifiés de 0 à 1.

Ajout d'un pattern «A» à notre filtre de bloom simple est un exemple d'ajout d'un pattern «A» au filtre de bloom simple représenté dans Un exemple de filtre bloom simpliste, avec un champ de 16 bits et trois fonctions de hachage.

Ajouter un second pattern est aussi simple que de répéter ce processus. Le pattern est haché par chaque fonction de hachage tour à tour et le résultat est enregistré en mettant les bits à 1. A noter qu'à mesure qu'un filtre de bloom est rempli avec d'autres patterns, le résultat d'une fonction de hachage

peut coïncider avec un bit qui est déjà fixé à 1, auquel cas le bit n'est pas modifié. En essence, à mesure que plusieurs patterns enregistre sur des bits déjà fixé à 1, le filtre de bloom commence à devenir saturé et la précision du filtre diminue. C'est la raison pour laquelle le filtre est une structure de données probabilistes – il devient moins précis à mesure que l'on ajoute des patterns. La précision dépend du nombre de patterns ajoutés versus la taille du tableau de bits (N) et du nombre de fonctions de hachage (M). Un tableau de bits plus large avec plus de fonctions de hachage peut enregistrer plus de patterns avec une précision plus élevée. Un tableau de bits plus petit ou moins de fonctions de hachage enregistreront moins de patterns et le résultat perdra en précision.

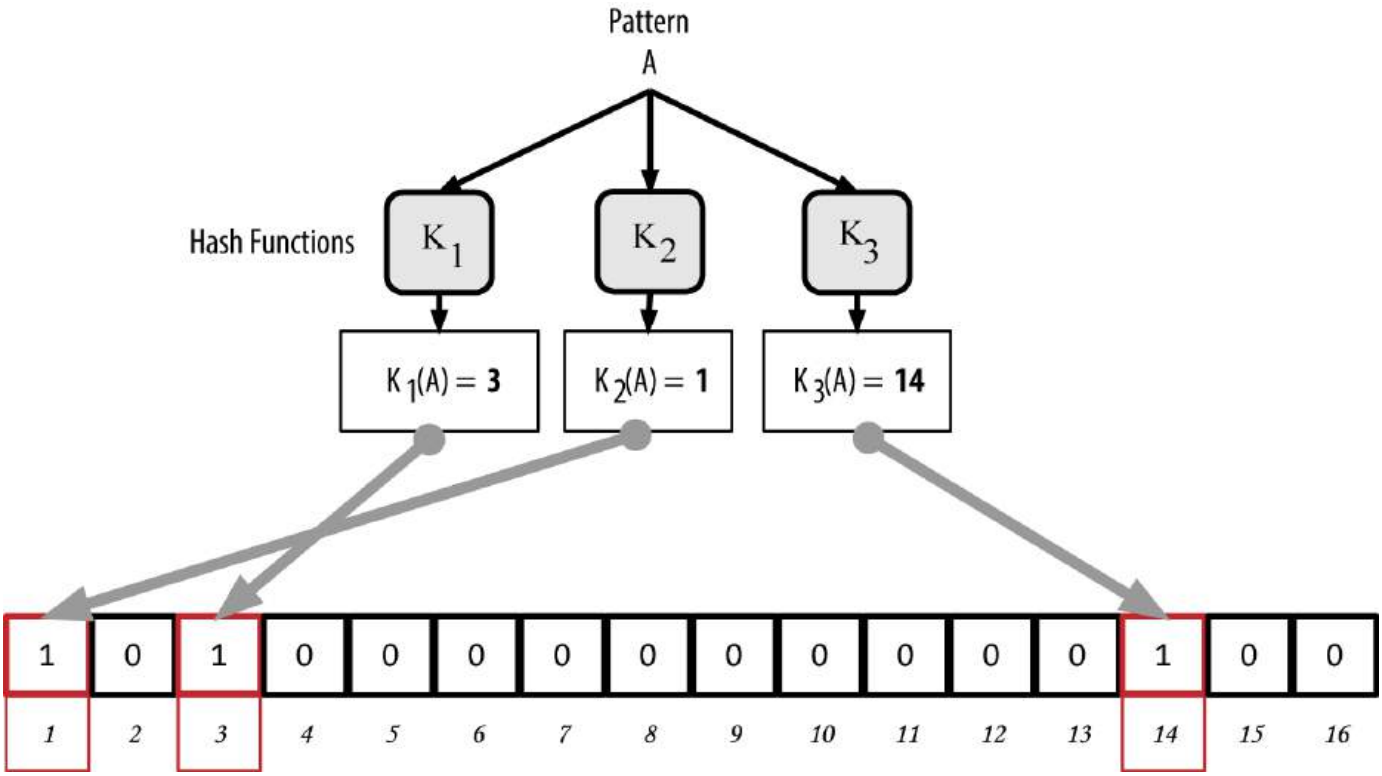


Figure 9. Ajout d'un pattern «A» à notre filtre de bloom simple

Ajout d'un second pattern «B» à notre filtre de bloom simple est un exemple d'ajout d'un deuxième pattern "B" au filtre de bloom simple.

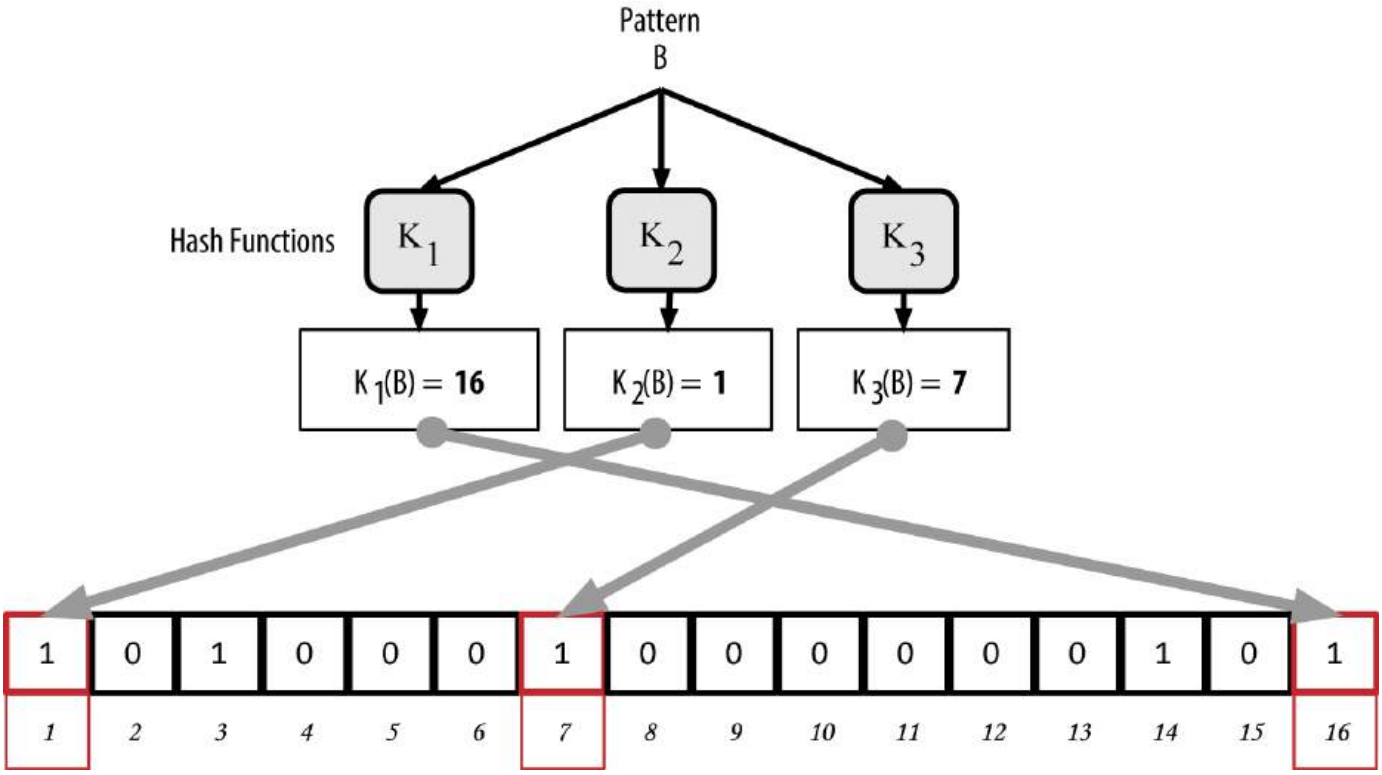


Figure 10. Ajout d'un second pattern «B» à notre filtre de bloom simple

Pour tester si un pattern fait partie d'un filtre de bloom, le pattern est haché par chaque fonction de hachage et le pattern de bits résultant est confronté au tableau de bits. Si tous les bits indexés par les fonctions de hachage sont mis à 1, alors le motif est *probablement* enregistré dans le filtre de bloom. Parce que les bits peuvent être fixé à cause du chevauchement de plusieurs pattern, la réponse n'est pas absolue, mais est plutôt probabiliste. En termes simples, le résultat positif d'un filtre de bloom est "Peut-être, oui."

Test l'existence du pattern "X" dans le filtre de bloom. Le résultat est une correspondance positive probabiliste, ce qui signifie «Peut-être." est un exemple consistant à tester l'existence d'un pattern "X" dans notre simple filtre de bloom. Les bits correspondants sont fixés à 1, de sorte que le motif correspond probablement.

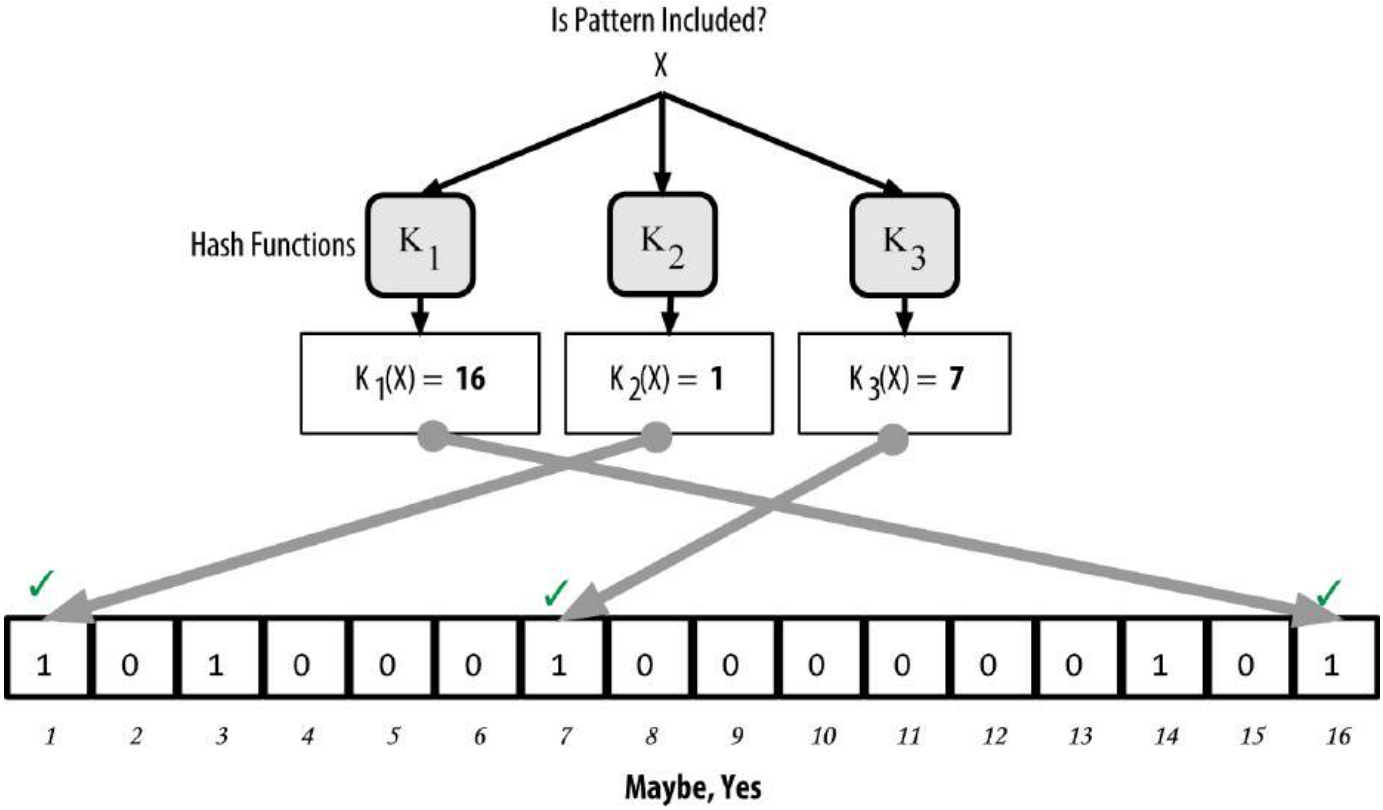


Figure 11. Test l'existence du pattern "X" dans le filtre de bloom. Le résultat est une correspondance positive probabiliste, ce qui signifie «Peut-être."

Au contraire, si un modèle est testé sur un filtre de bloom et que tous les bits sont à 0, cela prouve que le modèle n'a pas été enregistrée dans le filtre de bloom. Un résultat négatif n'est pas une probabilité, c'est une certitude. En termes simples, un match négatif sur un filtre de bloom est un "Certainement pas!"

Test l'existence du pattern "Y" dans le filtre de bloom. Le résultat est un match négatif définitif, ce qui signifie "Certainement pas!" est un exemple consistant à tester l'existence du pattern "Y" sur le filtre de bloom. Un des bits correspondant est fixé à 0, donc le motif ne correspond certainement pas.

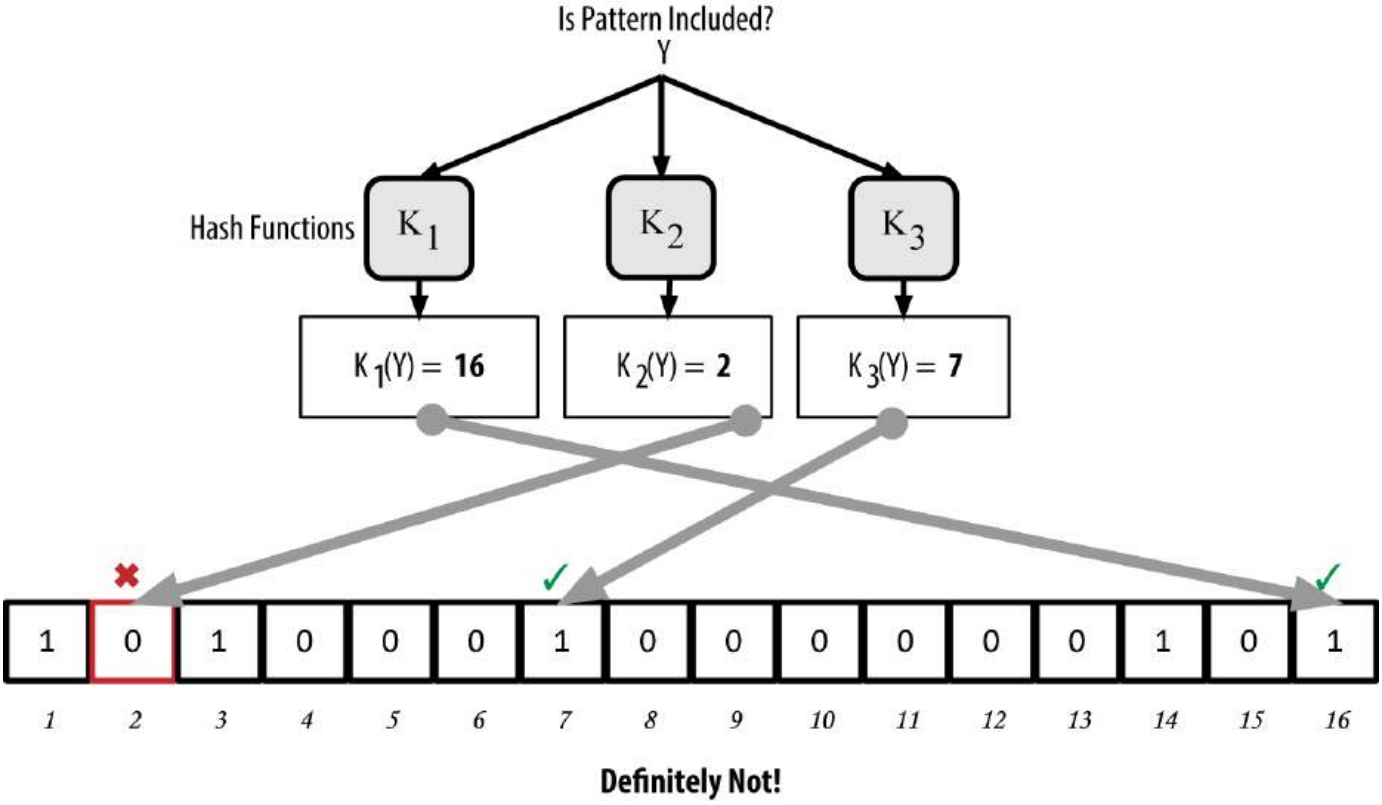


Figure 12. Test l'existence du pattern "Y" dans le filtre de bloom. Le résultat est un match négatif définitif, ce qui signifie "Certainement pas!"

L'implémentation bitcoin des filtres de bloom est décrite dans le Bitcoin Improvement Proposal 37 (BIP0037). Voir [appdxbitcoinimpproposals] ou visite GitHub.

Filtres de bloom et mises à jour de l'inventaire

Les filtres de bloom sont utilisés pour filtrer les transactions (et les blocs les contenant) qu'un noeud SPV reçoit de ses pairs. Les nœuds SPV vont créer un filtre qui correspond uniquement aux adresses détenues dans le portefeuille du noeud SPV. Le nœud SPV enverra alors un message filterload au pair, contenant le filtre de bloom à utiliser sur la connexion. Après qu'un filtre soit établi, le pair testera les sorties de chaque transaction avec le filtre de bloom. Seules les transactions qui correspondent au filtre sont envoyés au noeud.

En réponse au message getdata d'un noeud, les pairs enverront un message merkleblock qui ne contient que les en-têtes de bloc pour les blocs correspondant au filtre et un chemin de Merkle (voir [merkle_trees]) pour chaque transaction correspondante. Le pair enverra également des messages tx contenant les transactions qui correspondent au filtre.

Le nœud fixant le filtre de bloom peut de façon interactive ajouter des motifs au filtre en envoyant un message filteradd. Pour effacer le filtre de bloom, le nœud peut envoyer un message filterclear. Parce qu'il n'est pas possible de supprimer un pattern d'un filtre de bloom, un nœud doit remettre à zéro et renvoyer un nouveau filtre de bloom si un pattern n'est plus désiré.

Les pools de transaction

Presque chaque noeud sur le réseau bitcoin maintient une liste temporaire des transactions non confirmées appelée *memory pool*, *mempool* ou *transaction pool*. Les nœuds utilisent cette pool pour garder une trace des transactions connues du réseau, mais non encore incluses dans la blockchain. Par exemple, un nœud qui détient le portefeuille d'un utilisateur utilisera la pool de transaction pour suivre les paiements entrants vers le portefeuille qui ont été reçus sur le réseau mais non encore confirmés.

A mesure que les transactions soient reçues et vérifiées, elles sont ajoutées à la pool de transaction et relayées vers les noeuds voisins pour se propager sur le réseau.

Certaines implémentations de nœuds maintiennent également une pool séparée pour les transactions orphelines. Si les entrées d'une transaction se réfèrent à une transaction qui n'est pas encore connue, comme un parent manquant, la transaction orpheline sera stockée temporairement dans la pool des orphelines jusqu'à ce que la transaction parente arrive.

Lorsqu'une transaction est ajoutée à la pool de transaction, la pool des orphelines est vérifiée pour toutes les orphelines qui référencent les sorties de cette transaction (ses enfants). Toute orpheline correspondante est alors validée. Si elle est valide, elle est retirée de la pool des orphelines et ajoutée à la pool de transaction, complétant la chaîne qui a commencé avec la transaction parente. À la lumière de la transaction nouvellement ajoutée, qui n'est plus une orpheline, le processus est répété de manière récursive à la recherche d'autres descendants, jusqu'à ce qu'il n'y en ait plus. Grâce à ce processus, l'arrivée d'une transaction parente déclenche une cascade de reconstruction de toute une chaîne de transactions interdépendantes en réunissant les orphelines à leurs parents jusqu'au bout de la chaîne.

Tant la pool de transactions que la pool des orphelines (lorsque mise en place) est stockée dans la mémoire locale et n'est pas enregistrée sur un stockage persistant; plutôt, elles sont peuplées dynamiquement par les messages de réseau entrants. Quand un noeud commence, les deux pools sont vides et sont progressivement remplies avec de nouvelles transactions reçues sur le réseau.

Certaines implémentations du client bitcoin maintiennent également une base de données d'UTXO ou pool d'UTXO, qui représente l'ensemble de toutes les sorties non dépensées sur la blockchain. Bien que le terme "pool d'UTXO" semble similaire à la pool de transaction, elle représente un ensemble différent de données. Contrairement aux pools de transaction et d'orpheline, la pool d'UTXO n'est pas initialisée à vide, mais contient au lieu de ça des millions d'entrées de sorties de transaction non dépensées, dont certaines remontent à 2009. La pool d'UTXO peut résider dans la mémoire locale ou en tant que table de base de données indexée sur un stockage persistant .

Alors que les pools de transaction et d'orpheline représentent le point de vue local d'un seul nœud et peuvent varier considérablement d'un nœud à l'autre en fonction de quand le noeud a été démarré ou redémarré, la pool d'UTXO représente le consensus émergent du réseau et varie donc peu d'un noeud à l'autre. En outre, les pool de transaction et d'orpheline ne contiennent que des transactions non confirmées, tandis que la pool d'UTXO ne contient que des sorties confirmées.

Messages d’alerte

Les messages d’alerte sont une fonction rarement utilisée, mais sont néanmoins implémentés dans la plupart des nœuds. Les messages d’alerte sont le "système de diffusion d’urgence" de bitcoin, un moyen par lequel les développeurs principaux de bitcoin peuvent envoyer un message texte d’urgence à tous les nœuds bitcoin. Cette fonctionnalité est implémentée pour permettre à l’équipe de développement de notifier tous les utilisateurs de bitcoin d’un grave problème dans le réseau, tel un bug critique qui nécessite une action de l’utilisateur. Le système d’alerte n’a été utilisé qu’une poignée de fois, notamment au début de 2013 lorsqu’un bug critique de base de données causa un fork multibloc dans la blockchain bitcoin.

Les messages d’alerte sont propagés par le message alerte. Le message d’alerte contient plusieurs champs, incluant :

ID

Un identifiant d’alerte afin que les alertes en double puissent être détectées

Expiration

Une durée après laquelle l’alerte expire

RelayUntil

Un temps au bout duquel l’alerte ne doit pas être relayée

MinVer, MaxVer

L’étendue des versions du protocole bitcoin auquelle s’applique cette alerte

subVer

La version du logiciel client à laquelle s’applique cette alerte

Priority

Un niveau de priorité pour l’alerte, actuellement inutilisé

Les alertes sont signés de manière cryptographique par une clé publique. La clé privée correspondante est détenue par quelques membres choisis de l’équipe de développement bitcoin. La signature numérique garantit que de fausses alertes ne soient propagées sur le réseau.

Chaque nœud recevant ce message d’alerte va le vérifier, contrôler son expiration, et le propager à tous ses pairs, assurant ainsi la propagation rapide à travers l’ensemble du réseau. En plus de propager l’alerte, les noeuds peuvent mettre en oeuvre une fonction d’interface utilisateur pour présenter l’alerte à l’utilisateur.

Dans le client Bitcoin Core, l’alerte est configuré avec l’option en ligne de commande -alertnotify, qui spécifie une commande à exécuter lorsqu’une alerte est reçue. Le message d’alerte est passé en paramètre à la commande alertnotify. Le plus souvent, la commande alertnotify est réglée pour générer un message électronique à l’administrateur du nœud, contenant le message d’alerte. L’alerte

est également affiché comme une boîte de dialogue pop-up dans l'interface graphique utilisateur (bitcoin-Qt) si elle est en marche.

D'autres implémentations du protocole bitcoin peuvent gérer l'alerte de différentes manières. De nombreux systèmes de minage à hardware intégré ne mettent pas en œuvre la fonction de message d'alerte parce qu'ils n'ont aucune interface utilisateur. Il est fortement recommandé que les mineurs exécutant ces systèmes de minage s'abonnent à des alertes via un administrateur de pool ou en exécutant un noeud léger seulement à des fins d'alerte.

La 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 [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, 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 bitcoins—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 d'un bloc

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 250 bytes and the average block contains more than 500 transactions. A complete block, with all transactions, is therefore 1,000 times larger than the block header. La structure d'un bloc describes the structure of a block.

Table 1. La structure d'un bloc

Taille	Champ	Description
4 octets	Taille de bloc	La taille du bloc, en octets, suivant ce champ
80 octets	Entête de bloc	Plusieurs champs forment l'entête de bloc
1-9 bytes (VarInt)	Transaction Counter	How many transactions follow
Variable	Transactions	Les transactions enregistrées dans ce bloc

Entête de bloc

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 [ch8]. The third piece of metadata is the merkle tree root, a data structure used to efficiently summarize all the transactions in the block. La structure de l'entête de bloc describes the structure of a block header.

Table 2. La structure de l'entête de bloc

Taille	Champ	Description
4 octets	Version	Un numéro de version pour suivre les mises à jour logicielles/de protocole
32 octets	Hash du bloc précédent	Une référence au hachage du précédent (parent) bloc dans la chaîne

Taille	Champ	Description
32 octets	Merkle Root	Un hachage de la racine de l'arbre de Merkle des transactions de ce bloc
4 octets	Timestamp	Le temps de création approximative de ce bloc (secondes depuis le début de l'Ere Unix (Unix Epoch))
4 octets	Le niveau de difficulté	Le niveau de difficulté de l'algorithme proof-of-work pour ce bloc
4 octets	Nonce	Un compteur utilisé pour l'algorithme proof-of-work

The nonce, difficulty target, and timestamp are used in the mining process and will be discussed in more detail in [ch8].

Identifiants de Bloc: Hashage d'Entête de Bloc et Hauteur de Bloc

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, <phrase role="keep-together">because only the block header is used to compute it. For example,</phrase> 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 hashing the block header.

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 <phrase role="keep-together"><emphasis>block height</emphasis>. The first block ever created is at block height 0 (zero) and is the</phrase> <phrase role="keep-together">same block that was previously referenced by the following block hash</phrase> 000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f. A block can thus be identified 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. The block height on January 1, 2014, was approximately 278,000, meaning

there were 278,000 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 [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.

TIP

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.

Le Bloc de Genèse

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.

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.info, and you will find a page describing the contents of this block, with a URL containing that hash:

https://blockchain.info/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

https://blockexplorer.com/block/000000000019d6689c085ae165831e934ff763ae46a2a6c172b3f1b60a8ce26f

Using the Bitcoin Core reference client on the command line:

```
$ bitcoind 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.

Lier les Blocs dans la 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.

Le nœud bitcoin reçoit alors le nouveau bloc du réseau, qu'il décrypte comme suivant:

```
{
    "size" : 43560,
    "version" : 2,
    "previousblockhash" :
        "0000000000000027e7ba6fe7bad39faf3b5a83daed765f05f7d1b71a1632249",
    "merkleroot" :
        "5e049f4030e0ab2debb92378f53c0a6e09548aea083f3ab25e1d94ea1155e29d",
    "time" : 1388185038,
    "difficulty" : 1180923195.25802612,
    "nonce" : 4215469401,
    "tx" : [
        "257e7497fb8bc68421eb2c7b699dbab234831600e7352f0d9e6522c7cf3f6c77",

 #[... beaucoup d'autres transactions ...]

        "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. Blocks linked in a chain, by reference to the previous block header hash shows the chain of three blocks, linked by references in the previousblockhash field.

Les Arbres de Merkle

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.

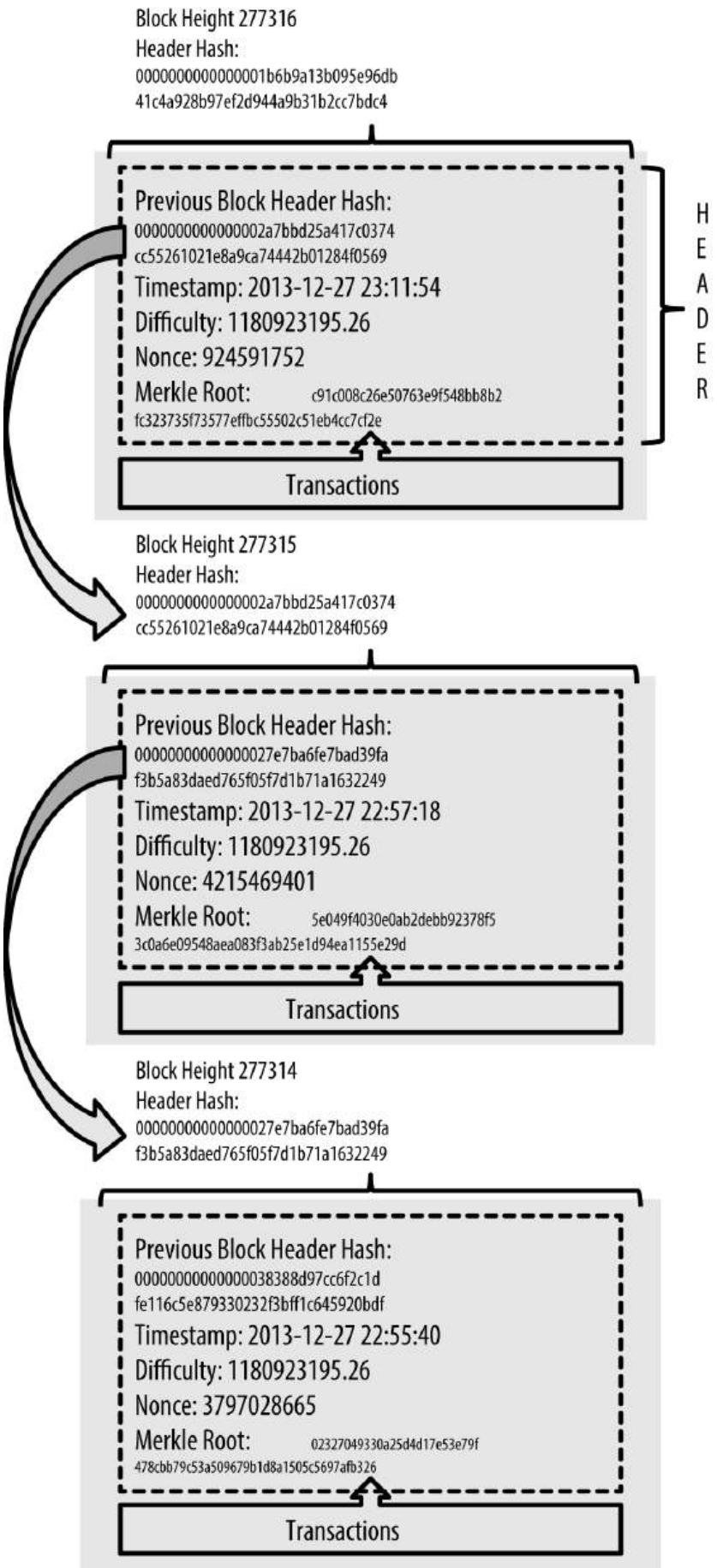

Block Height 277316
Header Hash:
0000000000000001b6b9a13b095e96db
41c4a928b97ef2d944a9b31b2cc7bdc4
Previous Block Header Hash:
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Timestamp: 2013-12-27 23:11:54
Difficulty: 1180923195.26
Nonce: 924591752
Merkle Root: c91c008c26e50763e9f548bb8b2
fc323735f73577effbc55502c51eb4cc7cf2e
Transactions
HEADER
Block Height 277315
Header Hash:
0000000000000002a7bbd25a417c0374
cc55261021e8a9ca74442b01284f0569
Previous Block Header Hash:
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
Timestamp: 2013-12-27 22:57:18
Difficulty: 1180923195.26
Nonce: 4215469401
Merkle Root: 5e049f4030e0ab2debb92378f5
3c0a6e09548aea083f3ab25e1d94ea1155e29d
Transactions
Block Height 277314
Header Hash:
00000000000000027e7ba6fe7bad39fa
f3b5a83daed765f05f7d1b71a1632249
Previous Block Header Hash:
00000000000000038388d97cc6f2c1d
fe116c5e879330232f3bff1c645920bdf
Timestamp: 2013-12-27 22:55:40
Difficulty: 1180923195.26
Nonce: 3797028665
Merkle Root: 02327049330a25d4d17e53e79f
478cbb79c53a509679b1d8a1505c5697afb326
Transactions

Figure 1. 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 Calculating the nodes in a merkle tree. 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 H_A, H_B, H_C, and H_D:

```
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 H_{AB}, 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.

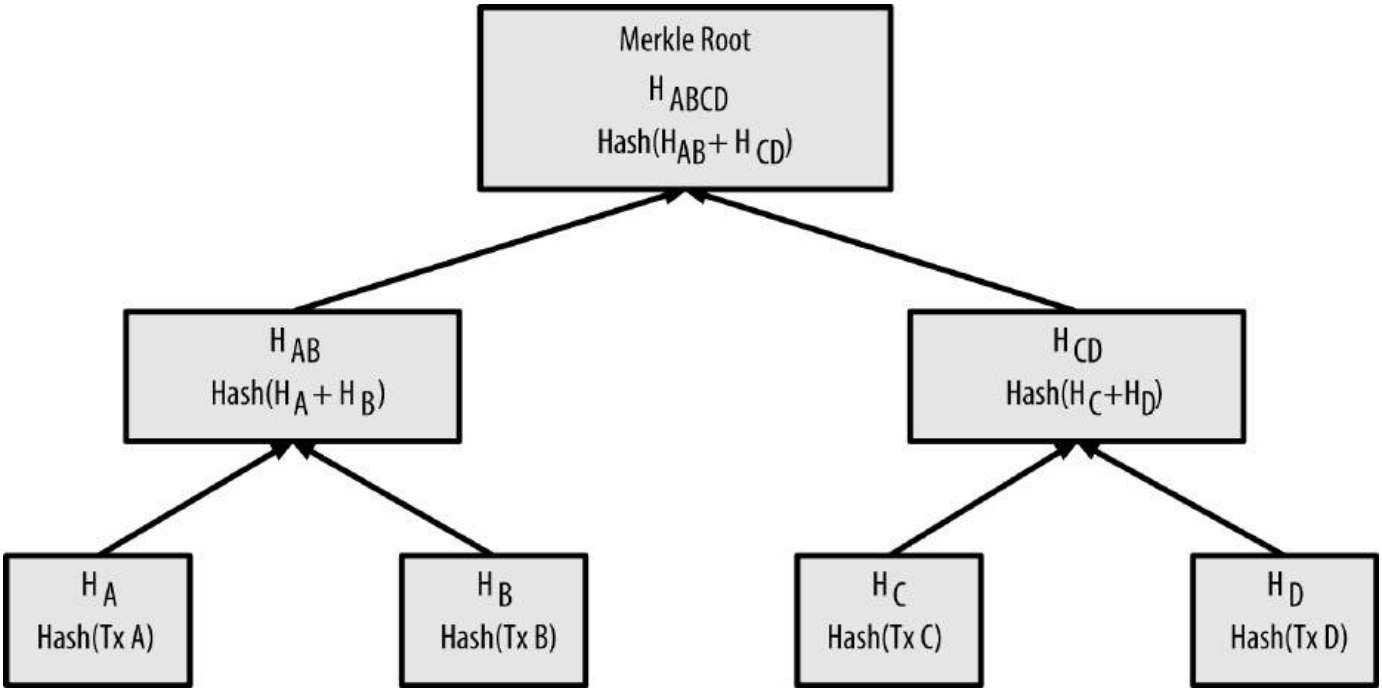


Figure 2. Calculating the nodes in a merkle tree

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 Duplicating one data element achieves an even number of data elements, where transaction C is duplicated.

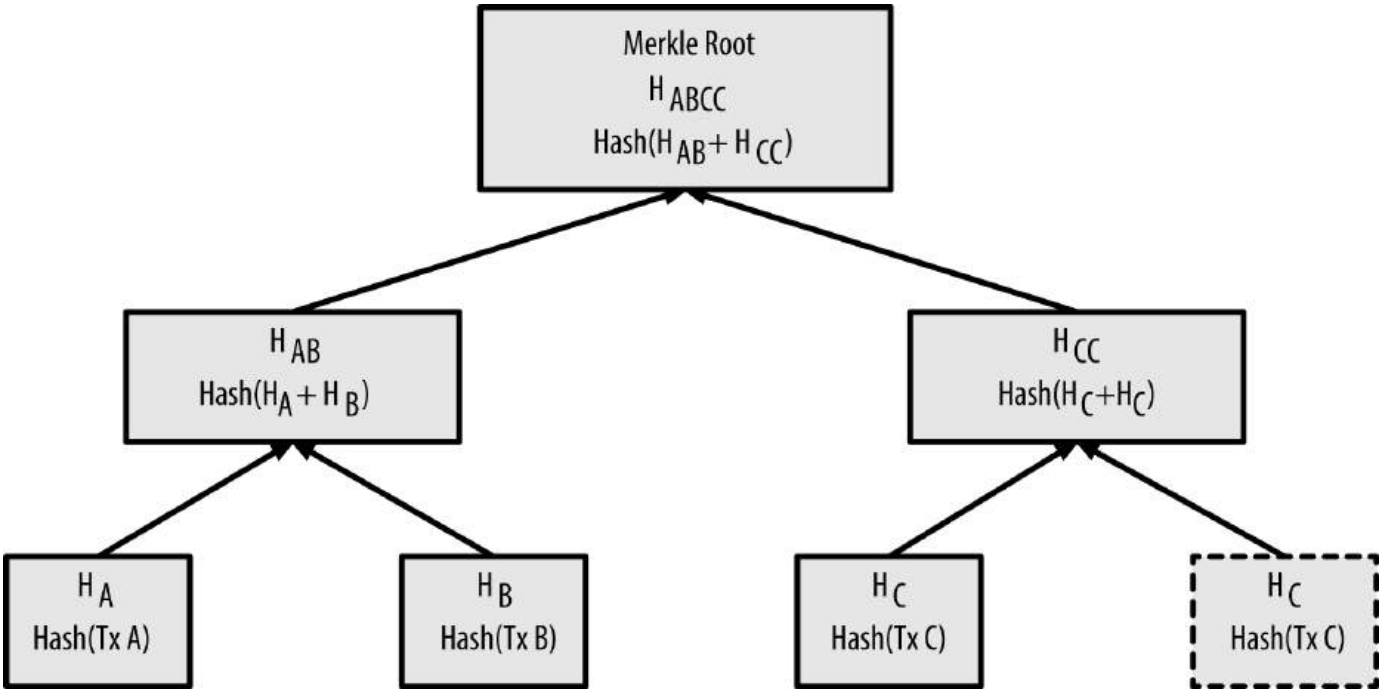


Figure 3. Duplicating one data element achieves an even number of data elements

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 A merkle tree summarizing many data elements, 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-size block.

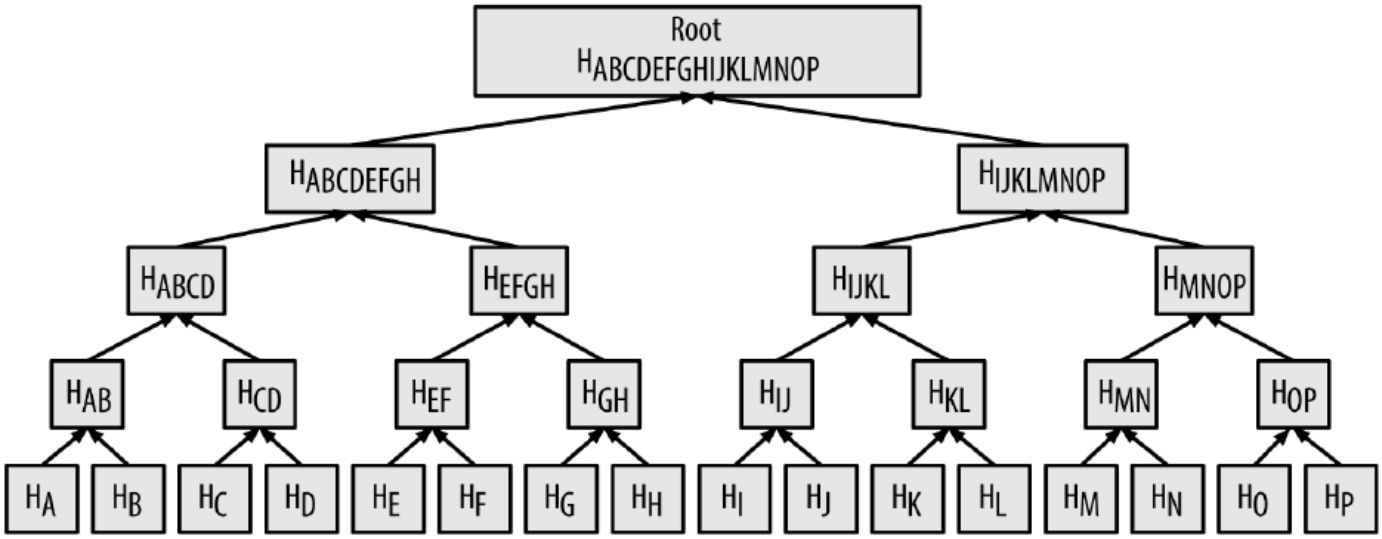


Figure 4. A merkle tree summarizing many data elements

In A merkle path used to prove inclusion of a data element, 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 (noted in blue in A merkle path used to prove inclusion of a data element) H_L, H_{IJ}, H_{MNOP} and $H_{ABCDEFGH}$. With those four hashes provided as an authentication path, any node can prove that H_K (noted in green in the diagram) is included in the merkle root by computing four additional pair-wise hashes H_{KL}, H_{IJKL}, $H_{IJKLMNOP}$, and the merkle tree root (outlined in a dotted line in the diagram).

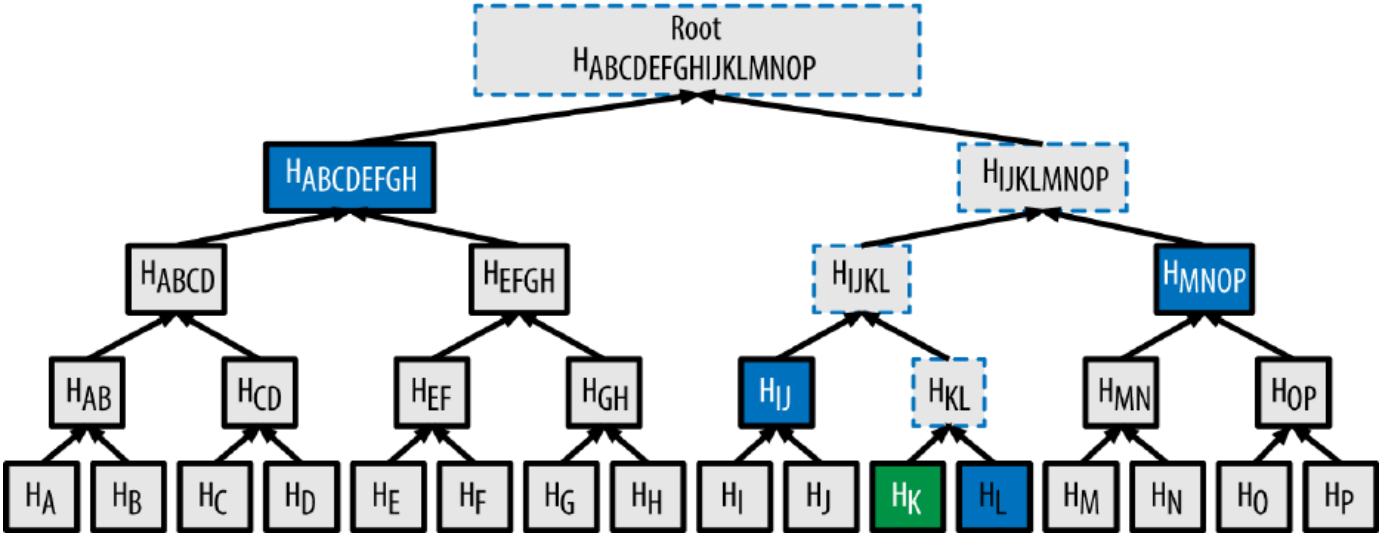


Figure 5. A merkle path used to prove inclusion of a data element

The code in Building a merkle tree 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.

Example 1. Building a merkle tree

```
#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::make_serializer(concat_data.begin());
            concat.write_hash(*it);
            concat.write_hash(*(it + 1));
            assert(concat.iterator() == concat_data.end());
            // 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_hex(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_hex(merkle_root) << std::endl;
    return 0;
}
```

Compiling and running the merkle example code shows the result of compiling and running the merkle code.

Example 2. Compiling and running the merkle example 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. Merkle tree efficiency shows the amount of data that needs to be exchanged as a merkle path to prove that a transaction is part of a block.

Table 3. 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 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).

Minage et Consensus

Introduction

Le minage est le processus par lequel de nouveaux bitcoins sont ajoutés à la masse monétaire. Il permet aussi de sécuriser le système bitcoin contre les transactions frauduleuses ou celles dépensant le même montant de bitcoin plus d'une fois, connu en tant que double-dépense. Les mineurs fournissent la puissance de calcul au réseau bitcoin en échange d'une récompense en bitcoins.

Les mineurs valident les nouvelles transactions et les enregistrent dans le registre global. Un nouveau bloc, contenant les transactions apparues depuis le dernier bloc, est « miné » toutes les 10 minutes en moyenne, ajoutant ainsi ces transactions à la blockchain. Les transactions qui font désormais partie d'un bloc et sont ajoutées à la blockchain sont considérées comme « confirmées », ce qui autorise les nouveaux propriétaires de bitcoin à dépenser ce qu'ils ont reçu dans de nouvelles transactions.

Les mineurs reçoivent deux types de récompenses en échange de leur minage : de nouveaux bitcoins créées avec chaque nouveau bloc, et les frais de transaction de toutes les transactions comprises dans le bloc. Pour gagner cette récompense, les mineurs s'affrontent pour résoudre un difficile problème mathématique basé sur un algorithme de hachage cryptographique. La solution au problème, appelée preuve de travail, est incluse dans le nouveau bloc et tient lieu de preuve aux efforts de calcul importants déployés par le mineur. La compétition pour résoudre l'algorithme preuve-de-travail afin de gagner la récompense et le droit d'enregistrer les transactions sur la blockchain est la base du modèle de sécurité bitcoin.

Le processus de génération de nouveaux bitcoins est appelé minage parce que la récompense est conçue de manière à simuler des rendements en baisse, similairement au minage de métaux précieux. La masse monétaire bitcoin est créée à travers le minage, de la même façon qu'une banque centrale émet de la monnaie en imprimant de nouveaux billets. La quantité de bitcoins nouvellement créés qu'un mineur peut ajouter à un bloc décroit approximativement tous les quatre ans (ou précisément tous les 210 000 blocs). Cela a commencé à 50 bitcoins par bloc en Janvier 2009 et fut réduit de moitié à 25 bitcoins par bloc en Novembre 2012. Il sera à nouveau réduit de moitié à 12,5 bitcoins par bloc dans le courant de l'année 2016. Sur la base de cette formule, les récompenses au minage de bitcoin diminuent de façon exponentielle jusqu'à environ l'année 2140, quand tous les bitcoins (20,99999998 millions) auront été délivrés. Après 2140, aucun nouveau bitcoin ne sera émis.

Les mineurs de bitcoins acquièrent également les frais de transactions. Chaque transaction peut inclure des frais de transaction, correspondant à l'excédent entre les entrées et les sorties d'une transaction. Le mineur gagnant parvient à "garder la monnaie" sur les transactions incluses dans le bloc gagnant. Aujourd'hui, les frais représentent 0,5% ou moins du revenu d'un mineur de bitcoin, la grande majorité provenant des bitcoins nouvellement frappés. Cependant, comme la récompense diminue avec le temps et que le nombre de transactions par blocs augmente, une plus grande proportion des revenus du minage proviendra des frais. Après 2140, tous les revenus des mineurs viendront des frais de transaction.

Le mot « minage » est quelque peu trompeur. En évoquant l'extraction de métaux précieux, il attire notre attention sur la récompense du minage, les nouveaux bitcoins dans chaque bloc. Bien que le minage soit encouragé par cette récompense, le but principal du minage n'est pas la récompense ou la génération de nouvelles pièces. Si vous voyez seulement le minage comme le processus par lequel les bitcoins sont créés, vous confondez les moyens (les mesures d'incitation) et le but du processus. Le minage est le processus principal de la chambre de compensation décentralisée par lequel les transactions sont validées. Le minage sécurise le système bitcoin et permet l'émergence d'un consensus sur tout le réseau sans autorité centrale.

Le minage est l'invention qui rend bitcoin spécial, un mécanisme de sécurité décentralisé qui constitue la base de la monnaie numérique pair-à-pair. La récompense en pièces nouvellement frappées et en frais de transaction est une mesure d'incitation qui aligne les actions des mineurs à la sécurité du réseau, œuvrant dans le même temps à la création monétaire.

Dans ce chapitre, nous examinerons d'abord le minage en tant que mécanisme de création monétaire puis nous étudierons sa fonction la plus importante : le mécanisme de consensus émergent décentralisé qui fonde la sécurité du bitcoin.

L'économie bitcoin et la création monétaire

Les bitcoins sont "frappés" lors de la création de chaque bloc à un taux défini qui va en diminuant. Chaque bloc, généré en moyenne toutes les 10 minutes, contient entièrement de nouveaux bitcoins, créés à partir de rien. Tous les 210 000 blocs, soit environ tous les quatre ans, le taux d'émission de la monnaie est diminué de 50%. Pour les quatre premières années de fonctionnement du réseau, chaque bloc contenait 50 nouveaux bitcoins.

En Novembre 2012, le nouveau taux d'émission de bitcoin a été réduit à 25 bitcoins par bloc et il diminuera de nouveau à 12,5 bitcoins au bloc 420 000, qui sera miné courant 2016. Le taux de nouvelle monnaie diminue de façon exponentiel sur 64 "division par deux" jusqu'au bloc 13 230 000 (miné approximativement en 2137), où il atteindra l'unité de monnaie minimum de 1 satoshi. Enfin, après 13,44 millions blocs, en 2140 environ, près de 2 099 999 997 690 000 satoshis, soit près de 21 millions de bitcoins, seront émis. En conséquence, les blocs ne contiendront pas de nouveaux bitcoins, et les mineurs seront récompensés uniquement grâce aux frais de transaction. L'émission de devise bitcoin au fil du temps basée sur un taux d'émission diminuant de façon mathématique représente le nombre total de bitcoins en circulation au fil du temps, ainsi que l'émission de monnaie qui diminue.

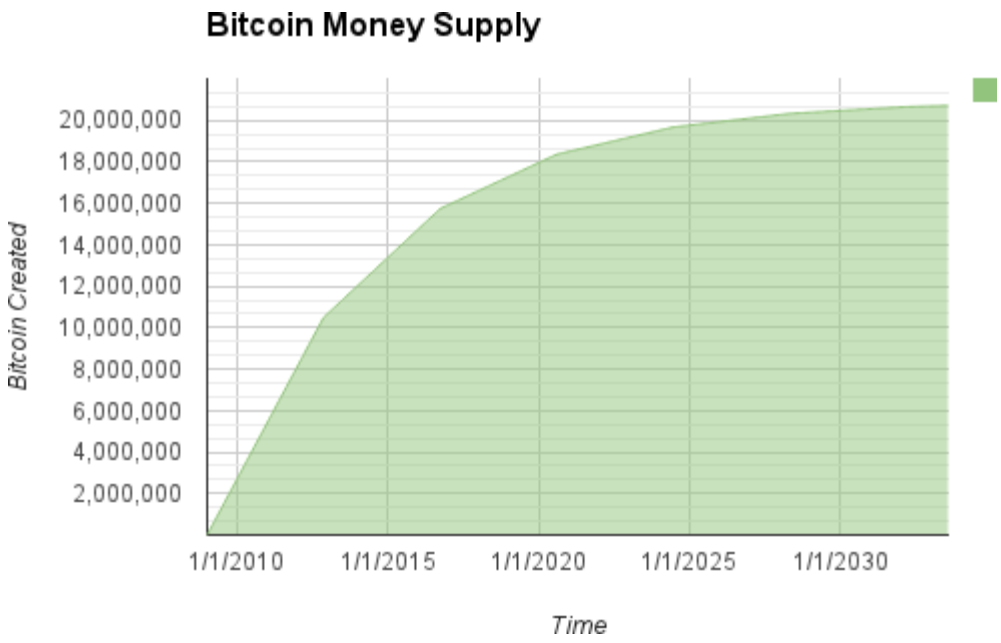


Figure 1. L'émission de devise bitcoin au fil du temps basée sur un taux d'émission diminuant de façon mathématique

NOTE

Le nombre maximum de pièces minées est la *limite supérieure* de possibles récompenses au minage. Dans la pratique, un mineur peut volontairement miner un bloc et ne pas récupérer la totalité de sa récompense. De tels blocs ont déjà été minés et d'autres peuvent l'être à l'avenir, résultant en une émission totale de la monnaie moindre.

Dans l'exemple de code [max_money], nous calculons la quantité totale de bitcoins qui sera émise.

Example 1. Un script pour calculer combien de bitcoins seront émis

```
# 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"
```

Exécution du script max_money.py représente la sortie produite par l'exécution de ce script.

Example 2. Exécution du script max_money.py

```
$ python max_money.py
Total BTC to ever be created: 2099999997690000 Satoshis
```

L'émission finie et en diminution crée une masse monétaire fixe qui résiste à l'inflation. Contrairement à une monnaie fiduciaire, qui peut être imprimé en nombre infini par une banque centrale, le bitcoin ne peut jamais être augmenté par impression.

Une monnaie déflationniste

La conséquence la plus importante et débattue d'une émission monétaire fixe et en diminution est que la monnaie aura tendance à être intrinsèquement *déflationniste*. La déflation est le phénomène de hausse de la valeur en raison de l'inadéquation entre l'offre et la demande, ce qui fait monter la valeur (et le taux de change) d'une monnaie. Le contraire de l'inflation, la déflation des prix signifie que l'argent a plus de pouvoir d'achat au fil du temps.

De nombreux économistes estiment qu'une économie déflationniste est une catastrophe qui doit être évitée à tout prix. En effet, dans une période de déflation rapide, les gens ont tendance à thésauriser l'argent au lieu de le dépenser, en espérant que les prix vont baisser. Un tel phénomène s'est déroulé au cours de la "Décennie perdue" au Japon, quand un effondrement complet de la demande a poussé la monnaie dans une spirale déflationniste.

Les experts bitcoin estiment que la déflation n'est pas mauvaise en soi. Au contraire, la déflation est associée à un effondrement de la demande parce que c'est le seul exemple de déflation que nous pouvons étudier. Dans une monnaie fiduciaire avec la possibilité d'impression illimitée, il est très difficile d'entrer dans une spirale déflationniste sauf s'il y a un effondrement complet de la demande et une réticence à imprimer de l'argent. La déflation du bitcoin n'est pas causée par un effondrement de la demande, mais par une réserve limitée de manière prévisible.

Dans la pratique, il est devenu évident que l'instinct de thésaurisation causé par une monnaie déflationniste peut être surmonté par des marchands prêts à faire des remises, jusqu'à ce que la remise surmonte l'instinct de thésaurisation de l'acheteur. Parce que le vendeur est aussi motivé pour thésauriser, la remise devient le prix d'équilibre sur lequel les deux instincts de thésaurisation se retrouvent au même niveau. Avec des rabais de 30% sur le prix bitcoin, la plupart des détaillants bitcoin n'ont pas de difficulté à surmonter l'instinct de thésaurisation et à générer des revenus. Il reste à déterminer si l'aspect déflationniste de la monnaie est vraiment un problème quand il n'est pas entraîné par une rétraction économique rapide.

Le consensus décentralisé

Dans le chapitre précédent, nous nous sommes intéressé à la blockchain, le registre (ou liste) public mondial de toutes les transactions, que tout le monde sur le réseau bitcoin accepte comme l'acte de propriété faisant autorité.

Mais comment l'ensemble du réseau peut s'entendre sur une unique « vérité » universelle concernant qui possède quoi, sans avoir à faire confiance à personne ? Tous les systèmes de paiement traditionnels dépendent d'un modèle de confiance qui a une autorité centrale fournissant un service de compensation, essentiellement la vérification et la validation de toutes les transactions. Bitcoin n'a pas d'autorité centrale, et pourtant chaque nœud complet a la copie complète d'un registre public auquel il peut se fier comme étant le registre faisant autorité. La blockchain n'est pas créée par une autorité centrale, mais est assemblée de façon indépendante par chaque noeud du réseau. D'une manière ou d'une autre, chaque nœud sur le réseau, agissant suite à la reception d'informations transmises à travers des connexions réseau non sécurisées, peut arriver à la même conclusion et assembler une copie du même registre public que tout le monde. Ce chapitre examine le processus par lequel le réseau bitcoin réalise un consensus global sans autorité centrale.

L'invention principale de Satoshi Nakamoto est le mécanisme décentralisé pour un *consensus émergent*. Emergent, parce que le consensus n'est pas atteint explicitement – il n'y a pas d'élection ou de moment établi où le consensus se produit. Au lieu de cela, le consensus est un artefact émergent de l'interaction asynchrone de milliers de noeuds indépendants, tous suivant des règles simples. Toutes les caractéristiques de bitcoin, incluant la monnaie, les transactions, les paiements et le modèle de sécurité qui ne dépend pas d'une autorité centrale ou de la confiance, dérivent de cette invention.

Le consensus décentralisé bitcoin émerge de l'interaction de quatre processus qui se produisent indépendamment sur des nœuds à travers le réseau :

- La vérification indépendante de chaque transaction, par chaque noeud complet, sur la base d'une liste exhaustive de critères
- L'agrégation indépendante de ces transactions dans de nouveaux blocs par les noeuds de minage, couplé aux calculs du matériel informatique prouvé par un algorithme proof-of-work
- La vérification indépendante des nouveaux blocs par chaque nœud et leur assemblage dans une chaîne
- La sélection indépendante, par chaque noeud, de la chaîne avec le plus de calculs démontrés par la preuve du travail

Dans les prochaines sections nous allons examiner ces processus et comment ils interagissent pour créer la caractéristique émergente de consensus de l'ensemble du réseau qui permet à n'importe quel nœud bitcoin d'assembler sa propre copie du registre mondiale faisant autorité, de confiance et public.

Vérification indépendante des transactions

Dans [transactions], nous avons vu comment les logiciels portefeuilles créent des transactions en

collectant les UTXO, fournissent les scripts de déverrouillage appropriés, construisant alors de nouvelles sorties assignées à un nouveau propriétaire. La transaction résultante est alors envoyée aux noeuds voisins dans le réseau bitcoin afin qu'elle puisse être propager à travers l'ensemble du réseau.

Cependant, avant de transmettre les transactions à ses voisins, chaque nœud bitcoin qui reçoit une transaction va d'abord la vérifier. Cela garantit que seules les transactions valides sont propagées à travers le réseau, tandis que les transactions non valides sont rejetées au premier noeud qui les rencontre.

Chaque nœud vérifie toutes les transactions en fonction d'une longue liste de critères :

- La syntaxe de la transaction et la structure de données doivent être correctes.
- Ni les listes d'entrées ou de sorties ne sont vides.
- La taille de la transaction en octets est inférieure à MAX_BLOCK_SIZE.
- Chaque valeur de sortie, ainsi que le total, doivent être dans la plage autorisée de valeurs (moins de 21 millions de pièces, plus de 0).
- Aucunes des entrées ont hash = 0, N = -1 (les transactions coinbase ne doivent pas être transmises).
- nLockTime est inférieur ou égal à INT_MAX.
- La taille en octets de la transaction est supérieure ou égale à 100.
- Le nombre d'opérations de signature contenues dans la transaction est inférieur à la limite d'opération de signature.
- Le script de déverrouillage (scriptSig) peut seulement placer des nombres sur la pile, et le script de verrouillage (scriptPubkey) doit correspondre aux règles isStandard (ceci rejette les transactions "non standard").
- Une transaction correspondante dans la pool, ou dans un bloc de la branche principale, doit exister.
- Pour chaque entrée, si la sortie référencée existe dans toute autre opération dans la pool, la transaction doit être rejetée.
- Pour chaque entrée, regarder dans la branche principale et la pool de transaction pour trouver la transaction de sortie référencée. Si la transaction de sortie est manquante pour une entrée, ce sera une transaction d'orphelin. Ajouter à la pool des transactions orphelines, si une transaction correspondante n'est pas déjà dans la pool.
- Pour chaque entrée, si la transaction de sortie référencée est une sortie coinbase, il doit avoir au moins COINBASE_MATURITY (100) confirmations.
- Pour chaque entrée, la sortie référencée doit exister et ne peut pas être déjà dépensée.
- En utilisant les transactions de sortie référencées pour obtenir les valeurs d'entrée, vérifier que chaque valeur d'entrée, ainsi que la somme, sont dans la plage de valeurs autorisée (moins de 21 millions de pièces, plus de 0).
- Rejeter si la somme des valeurs d'entrée est inférieur à la somme des valeurs de sortie.

- Rejeter si les frais de transaction serait trop faible pour être placer dans un bloc vide.
- Les scripts de déverrouillage pour chaque entrée doivent être valider par les scripts de verrouillage de sortie correspondants.

Ces conditions peuvent être vues en détail dans les fonctions AcceptToMemoryPool, CheckTransaction et CheckInputs dans le client de référence bitcoin. Notez que les conditions changent au fil du temps, pour répondre aux nouveaux types d'attaques par déni de service ou parfois pour assouplir les règles de manière à inclure davantage de types de transactions.

En vérifiant indépendamment chaque transaction dès sa réception et avant de la propager, chaque nœud construit une pool de transactions valides (mais non confirmées) connus sous le nom de *pool de transaction*, *pool de memoire* ou *mempool*.

Les nœuds de minage

Certains des nœuds du réseau bitcoin sont des noeuds spécialisés appelés *mineurs*. Dans [ch01_intro_what_is_bitcoin] nous avons introduit Jing, un étudiant en génie informatique à Shanghai, en Chine, qui est un mineur de bitcoin. Jing gagne des bitcoins en exécutant un "mining rig", du matériel informatique spécialement conçu pour miner des bitcoins. Le matériel de minage de Jing est connecté à un serveur exécutant un noeud bitcoin complet. Contrairement à Jing, certains mineurs minent sans un noeud complet, comme nous allons le voir dans Les pools de minage. Comme chaque autre noeud complet, le nœud de Jing reçoit et propage les transactions non confirmées sur le réseau bitcoin. Le nœud de Jing, toutefois, agrège également ces transactions dans de nouveaux blocs.

Le nœud de Jing est à l'écoute de nouveaux blocs propagés sur le réseau bitcoin, comme le font tous les nœuds. Cependant, l'arrivée d'un nouveau bloc a une signification particulière pour un nœud de minage. En effet, la concurrence entre les mineurs se termine avec la propagation d'un nouveau bloc qui agit comme une annonce signalant un gagnant. Pour les mineurs, la réception d'un nouveau bloc signifie que quelqu'un d'autre a remporté le concours et qu'ils ont perdu. Cependant, dans une compétition, la fin d'un tour est aussi le début d'un prochain. Le nouveau bloc n'est pas seulement la ligne d'arrivée marquant la fin de la course; il est aussi le coup d'envoi pour le bloc suivant.

Agréger les transactions dans des blocs

Après la validation des transactions, un noeud bitcoin les ajoutera à la *pool de mémoire* ou *pool de transaction*, où les transactions attendent jusqu'à ce qu'elles puissent être incluses (minées) dans un bloc. Le nœud de Jing recueille, valide et relaie les nouvelles transactions comme tout autre nœud. Contrairement à d'autres nœuds, cependant, le nœud de Jing va ensuite agréger ces transactions dans un *bloc candidat*.

Suivons les blocs qui ont été créés depuis qu'Alice a acheté une tasse de café au bar de Bob (voir [cup_of_coffee]). La transaction d'Alice a été incluse dans le bloc 277 316. Dans le but d'expliquer certains concepts dans ce chapitre, supposons que le bloc a été miné par le système de minage de Jing et voyons la transaction d'Alice au moment où elle intègre ce nouveau bloc.

Le noeud de minage de Jing conserve une copie locale de la blockchain, la liste de tous les blocs créés depuis le début du système bitcoin en 2009. Au moment où Alice achète sa tasse de café, le nœud de Jing a assemblé une chaîne allant jusqu'au bloc 277 314. Le nœud de Jing est à l'écoute de transactions, essayant de miner un nouveau bloc et aussi écoutant les blocs découverts par d'autres nœuds. Alors que le nœud de Jing est en train de miner, il reçoit le bloc 277 315 via le réseau bitcoin. L'arrivée de ce bloc signifie la fin de la compétition pour le bloc 277 315 et le début de la compétition pour créer le bloc 277 316.

Durant les 10 minutes précédentes, alors que le nœud de Jing était à la recherche d'une solution pour le bloc 277 315, il recueillait également les transactions en préparation du bloc suivant. Actuellement, il a rassemblé quelques centaines de transactions dans la pool mémoire. Après avoir reçu le bloc 277 315 et l'avoir valider, le nœud de Jing vérifiera également toutes les transactions dans la pool mémoire et supprimera celles qui ont été incluses dans le bloc 277 315. N'importe quelles transactions restant dans la pool mémoire sont non-confirmées et sont en attente d'être enregistrées dans un nouveau bloc.

Le nœud de Jing construit immédiatement un nouveau bloc vide, un candidat pour le bloc 277 316. Ce bloc est appelé un bloc candidat parce qu'il n'est pas encore un bloc valide, ne contenant pas une preuve de travail valide. Le bloc devient valide seulement si le mineur parvient à trouver une solution à l'algorithme proof-of-work.

Age d'une transaction, frais et priorité

Pour construire le bloc candidat, le nœud bitcoin de Jing sélectionne les transactions de la pool mémoire en appliquant un indicateur de priorité à chaque transaction et en ajoutant en premier les transactions les plus prioritaires. Les transactions sont prioritaires en fonction de « l'âge » de l'UTXO qui est dépensé dans leurs entrées, permettant aux entrées âgées et à forte valeur d'être prioritaire sur les entrées récentes et plus petites. Les transactions prioritaires peuvent être envoyées sans frais, si il y a assez d'espace dans le bloc.

La priorité d'une transaction est calculée comme étant la somme de la valeur et de l'âge des entrées divisé par la taille totale de la transaction :

```
Priorité = Somme (Valeur de l'entrée * Age de l'entrée) / Taille de la transaction
```

Dans cette équation, la valeur d'une entrée est mesurée dans l'unité de base, c'est-à-dire en satoshis (1/100 de million de bitcoin). L'âge d'un UTXO est le nombre de blocs qui se sont écoulées depuis que l'UTXO a été enregistré sur la blockchain, mesurant sa "profondeur" dans la blockchain. La taille de la transaction est mesurée en octets.

Pour qu'une transaction soit considérée comme « hautement prioritaire », sa priorité doit être supérieure à 57 600 000, ce qui correspond à un bitcoin (100 millions de satoshis), âgé d'un jour (144 blocs), dans une transaction d'une taille totale de 250 octets :

```
Priorité élevée > 100 000 000 satoshis * 144 blocs / 250 octets = 57 600 000
```

Les 50 premiers kilo-octets d'espace de transaction dans un bloc sont mis de côté pour les transactions à haute priorité. Le nœud de Jing remplira les 50 premiers kilo-octets, passant en priorité les plus prioritaires d'entre elles, quel que soit les frais. Cela permet aux transactions hautement prioritaires d'être traiter, même si elles n'ont aucun frais.

Le noeud de minage de Jing remplit alors le reste du bloc jusqu'à la taille de bloc maximale (MAX_BLOCK_SIZE dans le code), avec des transactions qui portent au moins les frais minimals, favorisant celles avec les plus grands frais par kilo-octet de transaction.

S'il reste de l'espace dans le bloc, le nœud de minage de Jing pourrait choisir de le remplir avec des transactions sans frais. Certains mineurs choisissent de miner des transactions sans frais sur une base best-effort. D'autres mineurs peuvent choisir d'ignorer les transactions sans frais.

Toutes les transactions laissées dans la pool mémoire, après que le bloc soit rempli, resteront dans la pool pour inclusion dans le bloc suivant. Comme les transactions restent dans la pool mémoire, leurs entrées « vieillissent », au fur et à mesure que l'UTXO qu'ils dépensent s'enfoncent plus profondément dans la blockchain avec de nouveaux blocs ajoutés par dessus. Parce que la priorité d'une transaction dépend de l'âge de ses entrées, les transactions restantes dans la pool vieilliront et augmenteront par conséquent en priorité. A la longue, une transaction sans frais peut atteindre une priorité suffisamment élevée pour être incluse dans un bloc gratuitement.

Les transactions bitcoin ne disposent pas d'un délai d'expiration. Une transaction qui est valable aujourd'hui sera valide à perpétuité. Toutefois, si une transaction est seulement propagée une seule fois à travers le réseau, elle persistera aussi longtemps qu'elle est maintenu dans la pool mémoire d'un nœud de minage. Quand un nœud de minage est redémarré, sa pool mémoire est effacée, car c'est une forme non-persistante de stockage. Bien qu'une transaction valide aurait pu être propagée à travers le réseau, si ce n'est pas fait il est possible qu'elle ne réside dans la pool mémoire d'aucuns mineurs. Les logiciels de portefeuille devraient retransmettre ces transactions ou les reconstruire avec des frais plus élevés si elles ne sont pas exécutées avec succès dans un laps de temps raisonnable.

Lorsque le nœud de Jing agrège toutes les transactions à partir de la pool mémoire, le nouveau bloc candidat a 418 transactions avec un total de frais de transaction de 0,09094928 bitcoin. Vous pouvez voir ce bloc dans la blockchain en utilisant l'interface en ligne de commande du client Bitcoin Core, comme indiqué dans Bloc 277 316.

```
$ bitcoin-cli getblockhash
277316000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4

$ bitcoin-cli getblock
000000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4
```

Example 3. Bloc 277 316

```
{
    "hash" : "000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35561,
    "size" : 218629,
    "height" : 277316,
    "version" : 2,
    "merkleroot" :
"c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e",
    "tx" : [
        "d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f",
        "b268b45c59b39d759614757718b9918caf0ba9d97c56f3b91956ff877c503fbe",

        ... 417 autres transactions ...

       ],
    "time" : 1388185914,
    "nonce" : 924591752,
    "bits" : "1903a30c",
    "difficulty" : 1180923195.25802612,
    "chainwork" : "0000000000000000000000000000000000000000000000934695e92aaf53afa1a",
    "previousblockhash" :
"000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569",
    "nextblockhash" :
"000000000000000010236c269dd6ed714dd5db39d36b33959079d78dfd431ba7"
}
```

La transaction de production

La première transaction ajoutée au bloc est une transaction spéciale, appelée *transaction de production* ou *transaction coinbase*. Cette transaction est réalisée par le noeud de Jing et constitue sa récompense pour l'effort de minage. Le nœud de Jing crée la transaction de production comme un paiement à son propre portefeuille : "Paie l'adresse de Jing 25,09094928 bitcoins." Le montant total de la récompense que Jing recueille pour le minage d'un bloc est la somme de la récompense coinbase (25 nouveaux bitcoins) et les frais de transaction (0,09094928) de toutes les transactions comprises dans le bloc comme indiqué dans Transaction de production :

```
$ bitcoin-cli getrawtransaction
d5ada064c6417ca25c4308bd158c34b77e1c0eca2a73cda16c737e7424afba2f 1
```

Example 4. Transaction de production

```
{
    "hex" :
"010000000100000000000000000000000000000000000000000000000000000000ffffffff0f
03443b0403858402062f503253482fffffffff0110c08d9500000000232102aa970c592640d19de03ff6f
329d6fd2eecb023263b9ba5d1b81c29b523da8b21ac00000000",
    "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"
                ]
            }
        }
    ],
    "blockhash" : "000000000000001b6b9a13b095e96db41c4a928b97ef2d944a9b31b2cc7bdc4",
    "confirmations" : 35566,
    "time" : 1388185914,
    "blocktime" : 1388185914
}
```

Contrairement aux transactions normales, la transaction de production ne consomme pas (dépense) d'UTXO comme entrées. Au lieu de cela, elle a une seule entrée, appelé le *coinbase*, qui crée des bitcoins à partir de rien. La transaction de production dispose d'une sortie, payable à l'adresse bitcoin du mineur. La sortie de la transaction de production envoie la valeur de 25,09094928 bitcoins à l'adresse bitcoin du mineur, dans ce cas 1MxTkeEP2PmHSMze5tUZ1hAV3YTKu2Gh1N.

Récompense coinbase et frais

Pour construire la transaction de production, le nœud de Jing calcule en premier le montant total des frais de transaction en ajoutant toutes les entrées et sorties des 418 transactions qui ont été ajoutés au bloc. Les frais sont calculés comme suit :

```
Total des frais = Somme (entrées) - Somme (sorties)
```

Dans le bloc 277 316, les frais totaux de transaction sont de 0,09094928 bitcoins.

Ensuite, le nœud de Jing calcule la récompense correcte pour le nouveau bloc. La récompense est calculé en fonction de la hauteur du bloc, à partir de 50 bitcoins par bloc et réduit de moitié tous les 210 000 blocs. Parce que ce bloc est à la hauteur de 277 316, la bonne récompense est 25 bitcoins.

Le calcul peut être vu dans la fonction GetBlockValue dans le client Bitcoin Core, comme représenté sur Calculer la récompense de bloc – Fonction GetBlockValue, client Bitcoin Core, main.cpp, ligne 1305.

Example 5. Calculer la récompense de bloc – Fonction GetBlockValue, client Bitcoin Core, main.cpp, ligne 1305

```
int64_t GetBlockValue(int nHeight, int64_t nFees)
{
    int64_t nSubsidy = 50 * COIN;
    int halvings = nHeight / Params().SubsidyHalvingInterval();

    // Force la récompense à zéro lorsque le décalage à droite est indéfinie.
    if (halvings >= 64)
        return nFees;

    // 'Subsidy' est réduit de moitié tous les 210 000 blocs qui vont se produire
environ tous les 4 ans.
    nSubsidy >>= halvings;

    return nSubsidy + nFees;
}
```

La subvention initiale est calculé en satoshis en multipliant 50 avec la constante COIN (100 000 000 satoshis). Ceci définit la récompense initiale (nSubsidy) à 5 milliards de satoshis.

Ensuite, la fonction calcule le nombre de halvings (réduction de moitié, dédoublement) qui ont eu lieu en divisant la hauteur du bloc courant par l'interval de dédoublement (SubsidyHalvingInterval). Dans le cas du bloc 277 316, avec un interval de dédoublement tous les 210 000 blocs, le résultat est de 1 dédoublement.

Le nombre maximum de dédoublements permis est de 64, afin que le code impose une récompense de zéro (retourne seulement les frais) si les 64 dédoublements sont dépassés.

Ensuite, la fonction utilise l'opérateur binaire de décalage vers la droite pour diviser la récompense (nSubsidy) par deux pour chaque tour de dédoublement. Dans le cas du bloc 277 316, cela va être appliqué une fois à la récompense de 5 milliards satoshis (une réduction de moitié) et se traduire par 2,5 milliards de satoshis, ou 25 bitcoins. L'opérateur binaire de décalage vers la droite est utilisé car il est plus efficace pour la division par deux que la division d'entier ou de nombre à virgule flottante.

Enfin, la récompense coinbase (nSubsidy) est ajouté aux frais de transaction (nFees), et la somme est retournée.

Structure de la transaction de production

Avec ces calculs, le nœud de Jing construit alors la transaction de génération pour se payer à lui-même 25,09094928 bitcoins.

Comme vous pouvez le voir dans Transaction de production, la transaction de génération a un format spécial. Au lieu d'une entrée de transaction spécifiant un précédent UTXO, il a une entrée "coinbase". Nous avons examiné les entrées de transaction dans [tx_in_structure]. Comparons une entrée de transaction ordinaire avec une entrée de transaction de génération. La structure d'une entrée de transaction "normale" représente la structure d'une transaction ordinaire, tandis que La structure d'une entrée de transaction de génération représente la structure d'une transaction de génération.

Table 1. La structure d'une entrée de transaction "normale"

Taille	Champ	Description
32 octets	Hash de transaction	Pointeur vers la transaction contenant l'UTXO à dépenser
4 octets	Index de la sortie	Le numéro d'index de l'UTXO à dépenser, le premier est 0
1-9 octets (VarInt)	Taille du script de déverrouillage	Longueur du script de déverrouillage en octets, (to follow)
Variable	Script de déverrouillage	Un script qui remplit les conditions du script de verrouillage de l'UTXO.
4 octets	Numéro de séquence	Fonctionnalité de remplacement de transaction actuellement désactivée, fixé à 0xFFFFFFFF

Table 2. La structure d'une entrée de transaction de génération

Taille	Champ	Description
32 octets	Hash de transaction	Tous les bits sont à zéro : pas une référence de hash de transaction
4 octets	Index de sortie	Tous les bits sont à 1 : 0xFFFFFFFF
1-9 octets (VarInt)	Taille des données coinbase	Longueur des données coinbase, de 2 à 100 octets
Variable	Données coinbase	Données arbitraires utilisés comme nonce supplémentaire et balises de minage dans les blocs v2, doit commencer avec la hauteur de bloc
4 octets	Numéro de séquence	Fixé à 0xFFFFFFFF

Dans une transaction de génération, les deux premiers champs sont réglés à des valeurs qui ne représentent pas une référence d'UTXO. Au lieu d'un "Hash de transaction", le premier champ est rempli avec 32 octets tous mis à zéro. L' "Index de sortie" est rempli avec 4 octets tous fixés à 0xFF (255 en décimal). Le "Script de déverrouillage" est remplacé par les données coinbase, un champ arbitraire de données utilisé par les mineurs.

Les données coinbase

Les transactions de génération ne disposent pas d'un champ pour un script de déverrouillage (alias, scriptSig). Au lieu de cela, ce champ est remplacé par les données coinbase, qui doivent être comprises entre 2 et 100 octets. Sauf pour les quelques premiers octets, le reste des données coinbase peut être utilisé par les mineurs de quelque façon qu'ils veulent; ce sont des données arbitraires.

Dans le bloc de genèse, par exemple, Satoshi Nakamoto a ajouté le texte "The Times 03/Jan/2009 Chancellor on brink of second bailout for banks" (Le chancelier est sur le point de lancer un second plan de sauvetage pour les banques) dans les données coinbase, en l'utilisant comme une preuve de la date et pour transmettre un message. Actuellement, les mineurs utilisent les données coinbase pour inclure des valeurs de nonce supplémentaire et des chaînes de caractères identifiant la pool de minage, comme nous le verrons dans les sections suivantes.

Les quelques premiers octets de la coinbase étaient habituellement arbitraires, mais ce n'est plus le cas. Conformément à la Proposition d'Amélioration de Bitcoin 34 (BIP0034), les blocs de version 2 (blocs avec le champ de version réglé à 2) doivent contenir l'indice de la hauteur du bloc comme un script d'opération "push" au début du champ coinbase.

Dans le bloc 277 316, nous voyons que le coinbase (voir Transaction de production), qui est dans le "Script de déverrouillage" ou champ scriptSig de l'entrée de la transaction, contient la valeur hexadécimal 03443b0403858402062f503253482f. Décodons cette valeur.

Le premier octet, 03, demande au moteur d'exécution de script de pousser les trois prochains octets sur la pile de script (voir [tx_script_ops_table_pushdata]). Les trois prochains octets, 0x443b04, sont la hauteur de bloc encodée en format little-endian (en arrière, l'octet le moins significatif en premier). Inversez l'ordre des octets et le résultat est 0x043b44, qui correspond à 277 316 en décimal.

Les chiffres hexadécimaux suivants (03858402062) sont utilisés pour encoder un *nonce* supplémentaire (voir La solution du nonce supplémentaire), soit une valeur aléatoire, utilisé pour trouver une solution à la preuve de travail appropriée.

La dernière partie des données coinbase (2f503253482f) est la chaîne de caractères ASCII /P2SH/, ce qui indique que le noeud de minage qui mine ce bloc supporte l'amélioration pay-to-script-hash (P2SH) définie dans le BIP0016. L'introduction de la fonction P2SH a requis un « vote » de la part des mineurs afin d'approuver soit BIP0016 ou BIP0017. Ceux approuvant la mise en œuvre de BIP0016 devait inclure /P2SH/ dans leurs données coinbase. Ceux approuvant l'implémentation de P2SH par le BIP0017 devait inclure la chaîne p2sh/CHV dans leurs données coinbase. Le BIP0016 a été élu comme le vainqueur, et de nombreux mineurs continuent d'inclure la chaîne /P2SH/ dans leur coinbase pour indiquer le support de cette fonctionnalité.

Extraire les données coinbase du bloc de genèse utilise la bibliothèque libbitcoin présentée dans [alt_libraries] pour extraire les données coinbase du bloc de genèse, affichant le message de Satoshi. Notez que la bibliothèque libbitcoin contient une copie statique du bloc de genèse, de sorte que l'exemple de code peut récupérer le bloc de genèse directement à partir de la bibliothèque.

Example 6. Extraire les données coinbase du bloc de genèse

```
/*
  Display the genesis block message by Satoshi.
*/
#include <iostream>
#include <bitcoin/bitcoin.hpp>

int main()
{
    // Create genesis block.
    bc::block_type block = bc::genesis_block();
    // Genesis block contains a single coinbase transaction.
    assert(block.transactions.size() == 1);
    // Get first transaction in block (coinbase).
    const bc::transaction_type& coinbase_tx = block.transactions[0];
    // Coinbase tx has a single input.
    assert(coinbase_tx.inputs.size() == 1);
    const bc::transaction_input_type& coinbase_input = coinbase_tx.inputs[0];
    // Convert the input script to its raw format.
    const bc::data_chunk& raw_message = save_script(coinbase_input.script);
    // Convert this to an std::string.
    std::string message;
    message.resize(raw_message.size());
    std::copy(raw_message.begin(), raw_message.end(), message.begin());
    // Display the genesis block message.
    std::cout << message << std::endl;
    return 0;
}
```

Nous compilons le code avec le compilateur GNU C++ et lançons l'exécutable produit, comme indiqué dans Compiler et exécuter l'exemple de code "satoshi-words".

Example 7. Compiler et exécuter l'exemple de code "satoshi-words"

```
$ # 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
```

Construire l’entête de bloc

Pour construire l’entête de bloc, le noeud de minage doit remplir six champs, comme indiqué dans La structure de l’entête de bloc.

Table 3. La structure de l’entête de bloc

Taille	Champ	Description
4 octets	Version	Un numéro de version pour suivre les mises à jour logicielles/de protocole
32 octets	Hash du bloc précédent	Une référence au hachage du précédent (parent) bloc dans la chaîne
32 octets	Merkle Root	Un hachage de la racine de l’arbre de Merkle des transactions de ce bloc
4 octets	Timestamp	Le temps de création approximative de ce bloc (secondes depuis le début de l’Ere Unix (Unix Epoch))
4 octets	Le niveau de difficulté	Le niveau de difficulté de l’algorithme proof-of-work pour ce bloc
4 octets	Nonce	Un compteur utilisé pour l’algorithme proof-of-work

Au moment où le bloc 277 316 fut miné, le numéro de version décrivant la structure du bloc est la version 2, qui est codée au format little-endian sur 4 octets tel que +0x02000000 +.

Ensuite, le nœud de minage doit ajouter le "Hash du bloc précédent". Tel est le hash de l’entête du bloc 277 315, le bloc précédent reçu du réseau, que le nœud de Jing a accepté et choisi comme le parent du bloc candidat 277 316. Le hash de l’entête de bloc pour le bloc 277 315 est :

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

La prochaine étape est de résumer toutes les transactions avec un arbre de Merkle, afin d’ajouter la racine de Merkle à l’entête du bloc. La transaction de génération est répertoriée comme la première transaction dans le bloc. Puis, plus de 418 transactions sont ajoutées, pour un total de 419 transactions dans le bloc. Comme nous l’avons vu dans le [merkle_trees], il doit y avoir un nombre pair de nœuds « feuilles » dans l’arbre, c’est pourquoi la dernière transaction est dupliquée, créant 420 noeuds, chacun contenant le hachage d’une transaction. Les hachages de transaction sont ensuite combinés, par paires,

créant chaque niveau de l'arbre, jusqu'à ce que toutes les transactions soient résumées dans un seul nœud à la "racine" de l'arbre. La racine de l'arbre de Merkle résume toutes les transactions en une valeur unique de 32 octets, que vous pouvez voir répertorié comme "racine de Merkle" dans Bloc 277 316, et ici :

```
c91c008c26e50763e9f548bb8b2fc323735f73577effbc55502c51eb4cc7cf2e
```

Le noeud de minage va ensuite ajouter un horodatage de 4 octets, codé comme un timestamp Unix "Epoch", qui est basée sur le nombre de secondes écoulées à partir du 1er Janvier 1970, à minuit heure UTC/GMT. Le temps 1388185914 est égal à vendredi 27 décembre 2013, 23:11:54 UTC/GMT.

Le noeud remplit alors le niveau de difficulté, qui définit la difficulté requise de la preuve de travail pour en faire un bloc valide. La difficulté est stockée dans le bloc en tant que "bits de difficulté", qui est un encodage de mantisse-exposant du niveau. L'encodage a un exposant de 1 octet, suivi par une mantisse de 3 octets (coefficient). Dans le bloc 277 316, par exemple, la valeur de la difficulté en bits est 0x1903a30c. La première partie 0x19 est un exposant hexadécimal, tandis que la partie suivante, 0x03a30c, est le coefficient. Le concept d'un niveau de difficulté est expliqué dans Cible de difficulté et de reciblage et la représentation des "bits de difficulté" est expliquée dans Représentation de la difficulté.

Le champ final est le nonce, qui est initialisée à zéro.

Avec tous les autres champs remplis, l'entête de bloc est maintenant terminé et le processus de minage peut commencer. L'objectif est maintenant de trouver une valeur pour le nonce qui se traduit par un hachage d'entête de bloc qui soit inférieur au niveau de difficulté. Le noeud de minage aura besoin de tester des milliards ou des trillions de valeurs de nonce avant qu'un nonce soit trouvé qui satisfasse les conditions requises.

Miner le bloc

Maintenant qu'un bloc candidat a été construit par le nœud de Jing, il est temps pour la plate-forme de minage de Jing de « miner » le bloc, de trouver une solution à l'algorithme proof-of-work qui rend le bloc valide. Tout au long de ce livre, nous avons étudié les fonctions de hachage cryptographique utilisées dans divers éléments du système bitcoin. La fonction de hachage SHA256 est la fonction utilisée dans le processus de minage du bitcoin.

En termes simples, le minage est un processus, répété à reprises, de hachage de l'entête de bloc, en changeant un paramètre jusqu'à ce que le hachage résultant corresponde à une cible spécifique. Le résultat de la fonction de hachage ne peut être déterminé à l'avance, ni ne peut être créé un motif qui va produire une valeur spécifique de hachage. Cette fonctionnalité de fonctions de hachage signifie que la seule façon de produire un résultat de hachage correspondant à une cible spécifique est d'essayer encore et encore, en modifiant de manière aléatoire l'entrée jusqu'à ce que le résultat de hachage souhaité apparaisse par hasard.

L’algorithme proof-of-work

Un algorithme de hachage prend une entrée de données de longueur arbitraire et produit un résultat déterministe de longueur fixe, une empreinte numérique de l’entrée. Pour toute entrée spécifique, le hachage sera toujours le même et peut facilement être calculé et vérifié par quiconque mettant en œuvre le même algorithme de hachage. La caractéristique clé d’un algorithme de hachage cryptographique est qu’il est pratiquement impossible de trouver deux entrées différentes qui produisent la même empreinte digitale. En corollaire, il est également pratiquement impossible de sélectionner une entrée de manière à produire une empreinte souhaitée, autrement qu’en testant des entrées au hasard.

Avec SHA256, la sortie est toujours de 256 bits de long, quelle que soit la taille de l’entrée. Dans Exemple de SHA256, nous allons utiliser l’interpréteur Python pour calculer le hachage SHA256 de la phrase, "I am Satoshi Nakamoto".

Example 8. Exemple de SHA256

```
$ python
```

```
Python 2.7.1
>>> import hashlib
>>> print hashlib.sha256("I am Satoshi Nakamoto").hexdigest()
5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e
```

Exemple de SHA256 représente le résultat du calcul du hachage de "I am Satoshi Nakamoto" : 5d7c7ba21cbbcd75d14800b100252d5b428e5b1213d27c385bc141ca6b47989e. Ce nombre de 256 bits est le *hachage* ou *résumé* de la phrase et dépend de chaque partie de la phrase. L’ajout d’une seule lettre, un signe de ponctuation ou tout autre caractère va produire un hachage différent.

Maintenant, si nous changeons la phrase, nous devrions nous attendre à voir des hachages complètement différents. Essayons cela en ajoutant un numéro à la fin de notre phrase, en utilisant le script Python simple dans SHA256 Un script pour générer de nombreux hachages en itérant sur un nonce.

Example 9. SHA256 Un script pour générer de nombreux hachages en itérant sur un nonce

```
# 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
```

Exécuter cela va produire les hachages de plusieurs phrases, fait différemment en ajoutant un numéro à la fin du texte. En incrémentant le nombre, nous pouvons obtenir différents hachages, comme indiqué dans Sortie SHA256 d'un script pour générer de nombreux hachages en itérant sur un nonce.

Example 10. Sortie SHA256 d'un script pour générer de nombreux hachages en itérant sur un nonce

```
$ python hash_example.py
```

```
I am Satoshi Nakamoto0 => a80a81401765c8eddee25df36728d732...
I am Satoshi Nakamoto1 => f7bc9a6304a4647bb41241a677b5345f...
I am Satoshi Nakamoto2 => ea758a8134b115298a1583ffb80ae629...
I am Satoshi Nakamoto3 => bfa9779618ff072c903d773de30c99bd...
I am Satoshi Nakamoto4 => bce8564de9a83c18c31944a66bde992f...
I am Satoshi Nakamoto5 => eb362c3cf3479be0a97a20163589038e...
I am Satoshi Nakamoto6 => 4a2fd48e3be420d0d28e202360cfbaba...
I am Satoshi Nakamoto7 => 790b5a1349a5f2b909bf74d0d166b17a...
I am Satoshi Nakamoto8 => 702c45e5b15aa54b625d68dd947f1597...
I am Satoshi Nakamoto9 => 7007cf7dd40f5e933cd89fff5b791ff0...
I am Satoshi Nakamoto10 => c2f38c81992f4614206a21537bd634a...
I am Satoshi Nakamoto11 => 7045da6ed8a914690f087690e1e8d66...
I am Satoshi Nakamoto12 => 60f01db30c1a0d4cbce2b4b22e88b9b...
I am Satoshi Nakamoto13 => 0ebc56d59a34f5082aaef3d66b37a66...
I am Satoshi Nakamoto14 => 27ead1ca85da66981fd9da01a8c6816...
I am Satoshi Nakamoto15 => 394809fb809c5f83ce97ab554a2812c...
I am Satoshi Nakamoto16 => 8fa4992219df33f50834465d3047429...
I am Satoshi Nakamoto17 => dca9b8b4f8d8e1521fa4eaa46f4f0cd...
I am Satoshi Nakamoto18 => 9989a401b2a3a318b01e9ca9a22b0f3...
I am Satoshi Nakamoto19 => cda56022ecb5b67b2bc93a2d764e75f...
```

Chaque phrase produit un résultat de hachage complètement différent. Ils semblent complètement aléatoire, mais vous pouvez reproduire les résultats de cet exemple sur n'importe quel ordinateur avec Python et voir exactement les mêmes hachages.

Le nombre utilisé en tant que variable dans un tel scénario est appelé un *nonce*. Le nonce est utilisé pour faire varier la sortie d'une fonction cryptographique, dans ce cas faire varier l'empreinte digitale SHA256 de la phrase.

Pour défier cet algorithme, nous allons fixer un objectif arbitraire : trouver une phrase qui produit un hachage hexadécimal qui commence par un zéro. Heureusement, ce ne est pas difficile! Sortie SHA256 d'un script pour générer de nombreux hachages en itérant sur un nonce montre que la phrase "I am Satoshi Nakamoto13" produit le hachage 0ebc56d59a34f5082aaef3d66b37a661696c2b618e62432727216ba9531041a5, ce qui correspond à nos critères. Il a fallu 13 tentatives pour le trouver. En termes de probabilités, si la sortie de la fonction de hachage est répartie uniformément nous nous attendons à trouver un résultat avec un 0 comme préfixe hexadécimal une fois tous les 16 hash (l'un des 16 chiffres hexadécimaux – de 0 à F). En termes numériques, cela signifie trouver une valeur de hachage qui est inférieure à 0x1000000000000000000000000000000000000000000000000000000000000000. Nous appelons ce seuil

la *cible* et le but est de trouver un hachage qui soit numériquement *moins que la cible*. Si on diminue la cible, trouver un hachage qui lui est inférieure devient de plus en plus difficile.

Pour donner une analogie simple, imaginez un jeu où les joueurs jettent une paire de dés à plusieurs reprises, en essayant de faire moins qu'une cible spécifiée. Au premier tour, la cible est de 12. Sauf si vous jetez un double-six, vous gagnez. Au prochain tour, la cible est 11. Les joueurs doivent jeter 10 ou moins pour gagner, à nouveau une tâche facile. Disons que quelques tours plus tard la cible est abaissée à 5. Maintenant, plus de la moitié des lancés seront supérieurs à 5 et donc invalide. De façon exponentielle plus la cible baisse et plus de lancés de dés seront nécessaires pour gagner. Finalement, lorsque la cible est de 2 (le minimum possible), un seul lancé tous les 36, ou 2% d'entre eux, produira un résultat gagnant.

Dans Sortie SHA256 d'un script pour générer de nombreux hachages en itérant sur un nonce, le "nonce" gagnant est 13 et ce résultat peut être confirmé par n'importe qui indépendamment. Tout le monde peut ajouter le numéro 13 comme suffixe à la phrase « I am Satoshi Nakamoto » et calculer le hachage, vérifiant qu'il est inférieure à la cible. Le résultat positif correspond aussi à la preuve de travail, car il prouve que nous avons fait le travail pour trouver ce nonce. Alors qu'il suffit d'un calcul de hachage pour le vérifier, il nous a fallu 13 calculs de hachage pour trouver un nonce qui a fonctionné. Si nous avions une cible inférieure (plus grande difficulté), il faudrait beaucoup plus de calculs de hachage pour trouver un nonce approprié, mais seulement un calcul de hachage pour quiconque veut le vérifier. En outre, en connaissant la cible, tout le monde peut estimer la difficulté en utilisant des statistiques et donc savoir combien de travail a été nécessaire pour trouver un tel nonce.

La preuve de travail bitcoin est très similaire au défi représenté dans Sortie SHA256 d'un script pour générer de nombreux hachages en itérant sur un nonce. Le mineur construit un bloc candidat rempli de transactions. Ensuite, le mineur calcule le hash de l'entête de ce bloc et voit si elle est inférieure à la *cible* actuelle. Si le hash n'est pas inférieur à la cible, le mineur modifiera le nonce (habituellement juste en incrémentant de un) et essayer à nouveau. À la difficulté actuelle dans le réseau bitcoin, les mineurs doivent essayer des quadrillions de fois avant de trouver un nonce qui résulte en un assez faible hash d'entête de bloc.

Un algorithme proof-of-work très simplifiée est mis en œuvre en Python dans Implémentation simplifiée de proof-of-work.

Example 11. Implémentation simplifiée de proof-of-work

```
#!/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
```

En exécutant ce code, vous pouvez régler la difficulté souhaitée (en bits, combien de bits en tête doivent être à zéro) et voir combien de temps il faudra à votre ordinateur pour trouver une solution. Dans Exécuter l'exemple de la preuve de travail avec diverses difficultés, vous pouvez voir comment cela fonctionne sur un ordinateur portable moyen.

Example 12. Exécuter l'exemple de la preuve de travail avec diverses difficultés

```
$ 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
```

Comme vous pouvez le voir, en augmentant la difficulté de 1 bit on provoque une augmentation exponentielle dans le temps qu'il faut pour trouver une solution. Si vous pensez à l'ensemble de l'espace de nombre 256 bits, chaque fois que vous fixez un bit de plus à zéro, vous diminuez l'espace de recherche de moitié. Dans Exécuter l'exemple de la preuve de travail avec diverses difficultés, il faut 84 millions de tentatives de hachage pour trouver un nonce qui produit un hachage avec les 26 premiers bits à zéro. Même à une vitesse de plus de 120 000 hachages par seconde, il nécessite encore 10 minutes sur un ordinateur portable pour trouver cette solution.

Au moment de l'écriture, le réseau tente de trouver un bloc dont le hash d'entête est inférieure à 000000000000004c296e6376db3a241271f43fd3f5de7ba18986e517a243baa7. Comme vous pouvez le voir, il y a beaucoup de zéros au début de ce hachage, ce qui signifie que la fourchette acceptable de hachage est beaucoup plus petite, donc il est plus difficile de trouver un hash valide. Il faudra en moyenne plus de 150 quadrillions de calculs de hachage par seconde pour le réseau pour découvrir le prochain bloc. Cela semble être une tâche impossible, mais heureusement, le réseau apporte 100 petahashes par seconde (PH / sec) de puissance de calcul à supporter, lequel sera en mesure de trouver un bloc en 10 minutes en moyenne.

Représentation de la difficulté

Dans Bloc 277 316, nous avons vu que le bloc contient la cible de difficulté, dans une notation appelé "bits de difficulté" ou seulement "bits", qui dans le bloc 277 316 a la valeur de 0x1903a30c. Cette notation exprime la cible de difficulté comme un format coefficient/exposant, avec les deux premiers chiffres hexadécimaux pour l'exposant et les six prochains chiffres hexadécimaux que le coefficient. Dans ce bloc, par conséquent, l'exposant est 0x19 et le coefficient est 0x03a30c.

La formule pour calculer la cible de difficulté de cette représentation est :

```
cible = coefficient * 2^(8 * (exposant - 3))
```

En utilisant cette formule, et la valeur de bits de difficulté 0x1903a30c, nous obtenons :

```
cible = 0x03a30c * 2^(0x08 * (0x19 - 0x03))^

=> cible = 0x03a30c * 2^(0x08 * 0x16)^

=> cible = 0x03a30c * 2 ^0xB0^
```

qui en décimal est :

```
=> cible = 238 348 * 2 ^176^

=> cible = 22 829 202 948 393 929 850 749 706 076 701 368 331 072 452 018 388 575 715 328
```

reconverti en hexadécimal :

```
=> cible = 0x00000000000000003A30C0000000000000000000000000000000000000000000
```

Cela signifie qu'un bloc valide d'une hauteur de 277 316 doit avoir un hash d'entête de bloc inférieur à la cible. En binaire ce nombre aurait plus des 60 premiers bits mis à zéro. Avec ce niveau de difficulté, un seul mineur traitant 1 trillion de hashes par seconde (1 tera-hachage par seconde ou 1 TH/sec) ne ferait que trouver une solution une fois tous les 8 496 blocs ou une fois tous les 59 jours, en moyenne.

Cible de difficulté et de reciblage

Comme nous l'avons vu, la cible détermine la difficulté et par conséquent influe sur le temps nécessaire pour trouver une solution à l'algorithme proof-of-work. Cela conduit à des questions évidentes : Pourquoi est-ce que la difficulté est réglable, qui la règle et comment ?

Les blocs bitcoin sont générés toutes les 10 minutes, en moyenne. Ceci est le rythme cardiaque de bitcoin et sous-tend la fréquence d'émission de la monnaie et la vitesse de règlement de la transaction. Cela doit rester constant et pas seulement à court terme, mais sur une période de plusieurs décennies. Pendant cette période, il est prévu que la puissance des ordinateurs continue d'augmenter à un rythme rapide. En outre, le nombre de participants au minage et les ordinateurs qu'ils utilisent vont également changer constamment. Pour garder le temps de génération de bloc à 10 minutes, la difficulté du minage doit être ajustée pour tenir compte de ces changements. En fait, la difficulté est un paramètre dynamique qui sera ajusté périodiquement pour atteindre une cible de bloc de 10 minutes. En termes simples, la cible de difficulté est réglée de façon à ce que la puissance de minage entraîne un intervalle

de bloc de 10 minutes.

Comment, alors, est un tel ajustement se fait dans un réseau totalement décentralisé ? Le reciblage de la difficulté se produit automatiquement et sur chaque nœud complet indépendamment. Tous les 2016 blocs, tous les nœuds reciblent la difficulté de preuve de travail. L'équation pour le reciblage de la difficulté mesure le temps qu'il a fallu pour trouver les 2016 derniers blocs et le compare au temps estimé qui est de 20 160 minutes (deux semaines sur la base d'un temps désiré de 10 minutes par bloc). Le rapport entre le laps de temps réel et le laps de temps désiré est calculé et l'ajustement correspondant de la difficulté (haut ou bas) est réalisé. En termes simples : si le réseau trouve des blocs plus rapidement que toutes les 10 minutes, la difficulté augmente. Si la découverte de bloc est plus lente que prévu, la difficulté diminue.

L'équation peut être résumée comme suit :

```
Nouvelle difficulté = Vieille difficulté * (Temps réel des derniers 2016 blocs / 20160
minutes)
```

Reciblage de la difficulté de la preuve de travail – GetNextWorkRequired() dans pow.cpp, ligne 43 montre le code utilisé dans le client Bitcoin Core.

Example 13. Reciblage de la difficulté de la preuve de travail – GetNextWorkRequired() dans pow.cpp, ligne 43

```
// Retour en arrière de l'équivalent de 14 jours de blocs
const CBlockIndex* pindexFirst = pindexLast;
for (int i = 0; pindexFirst && i < Params().Interval()-1; i++)
    pindexFirst = pindexFirst->pprev;
assert(pindexFirst);

// Étape d'ajustement de limite
int64_t nActualTimespan = pindexLast->GetBlockTime() - pindexFirst->GetBlockTime();
LogPrintf("  nActualTimespan = %d  before bounds\n", nActualTimespan);
if (nActualTimespan < Params().TargetTimespan()/4)
    nActualTimespan = Params().TargetTimespan()/4;
if (nActualTimespan > Params().TargetTimespan()*4)
    nActualTimespan = Params().TargetTimespan()*4;

// Reciblage
uint256 bnNew;
uint256 bnOld;
bnNew.SetCompact(pindexLast->nBits);
bnOld = bnNew;
bnNew *= nActualTimespan;
bnNew /= Params().TargetTimespan();

if (bnNew > Params().ProofOfWorkLimit())
    bnNew = Params().ProofOfWorkLimit();
```

NOTE Alors que le calibrage de la difficulté se produit tous les 2016 blocs, en raison d'une erreur off-by-one dans le client Bitcoin Core le calibrage est basé sur la durée totale des 2015 blocs précédents (non 2016 comme il se doit), ce qui entraîne un biais de reciblage vers une difficulté plus élevée de 0,05%.

Les paramètres Interval (2016 blocs) et TargetTimespan (deux semaines soit 1 209 600 de secondes) sont définis dans *chainparams.cpp*.

Pour éviter une volatilité extrême dans la difficulté, l'ajustement de reciblage doit être inférieure à un facteur de quatre (4) par cycle. Si le réglage de la difficulté requise est supérieure à un facteur de quatre, il sera ajustée par le maximum et pas plus. Toute autre ajustement sera effectué dans la prochaine période de reciblage car le déséquilibre persistera à travers les 2016 blocs suivants. Par conséquent, de grandes différences entre le pouvoir de hachage et la difficulté pourraient prendre plusieurs cycles de 2016 blocs pour s'équilibrer.

TIP La difficulté pour trouver un bloc bitcoin est d'environ "10 minutes de traitement" pour l'ensemble du réseau, basé sur le temps qu'il a fallu pour trouver les 2016 blocs précédents, ajusté tous les 2016 blocs.

Notez que la difficulté de la cible est indépendante du nombre de transactions ou la valeur des transactions. Cela signifie que la quantité de puissance hachage et donc de l'électricité dépensée pour sécuriser bitcoin est également tout à fait indépendante du nombre de transactions. L'adoption du bitcoin peut s'intensifier et le système restera sécurisé sans aucune augmentation de la puissance de hachage par rapport au niveau actuel. L'augmentation de la puissance de hachage dépend des forces du marché tel que de nouveaux mineurs arrivent sur le marché pour rivaliser pour la récompense. Tant qu'une puissance suffisante de hachage est sous le contrôle de mineurs agissant honnêtement à la poursuite de la récompense, cela est suffisant pour empêcher les attaques "takeover" et, par conséquent, suffisant pour sécurisé bitcoin.

La difficulté cible est étroitement liée au coût de l'électricité et au taux de change du bitcoin vers la monnaie utilisée pour payer l'électricité. Les systèmes de minage haute performance sont aussi efficace que possible avec la génération actuelle de fabrication de silicium, convertissant l'électricité en calcul de hachage au taux le plus élevé possible. Le principal facteur d'influence sur le marché du minage est le prix d'un kilowatt-heure en bitcoin, parce que ceci détermine la rentabilité du minage et par conséquent les motivations à entrer ou sortir du marché du minage.

Miner le bloc avec succès

Comme nous l'avons vu précédemment, le nœud de Jing a construit un bloc candidat et l'a préparé pour le minage. Jing a plusieurs plateformes de minage avec des ASIC (circuits intégrés pour application spécifique), où des centaines de milliers de circuits intégrés exécutent l'algorithme SHA256 en parallèle à des vitesses incroyables. Ces machines spécialisées sont connectés à son nœud de minage par USB. Ensuite, le nœud de minage en cours d'exécution sur l'ordinateur de Jing transmet l'entête de bloc à sa plateforme de minage, qui commence à tester des trillions de nonces par seconde.

Près de 11 minutes après avoir commencé à miner le bloc 277 316, l'une des machines de minage trouve une solution et l'envoie vers le nœud de minage. Lorsqu'elle est insérée dans l'entête de bloc, le nonce 4 215 469 401 produit un hachage de bloc de :

```
0000000000000002a7bbd25a417c0374cc55261021e8a9ca74442b01284f0569
```

qui est inférieure à la cible :

```
00000000000000003A30C00000000000000000000000000000000000000000
```

Immédiatement, le nœud de minage de Jing transmet le bloc à tous ses pairs. Ils reçoivent, valident puis propagent le nouveau bloc. Au fur et à mesure que le bloc navigue à travers le réseau, chaque noeud l'ajoute à sa propre copie de la blockchain, étendant celle-ci à une hauteur de 277 316 blocs.

Comme les nœuds de minage reçoivent et valident le bloc, ils abandonnent leurs efforts pour trouver un bloc à la même hauteur et commencent immédiatement à calculer le bloc suivant dans la chaîne.

Dans la section suivante, nous allons examiner le processus que chaque nœud utilise pour valider un bloc et sélectionner la chaîne la plus longue, créant le consensus qui forme la blockchain décentralisée.

Valider un nouveau bloc

La troisième étape dans le mécanisme de consensus de bitcoin est la validation indépendante de chaque nouveau bloc par chaque nœud du réseau. A mesure que les blocs nouvellement résolus sont poussés à travers le réseau, chaque noeud effectue une série de tests pour le valider avant de le propager à ses pairs. Cela garantit que seuls les blocs valides sont propagés sur le réseau. La validation indépendante assure également que les mineurs qui agissent honnêtement voient leurs blocs incorporés dans la blockchain, gagnant ainsi la récompense. Quant aux mineurs qui agissent malhonnêtement, leurs blocs sont rejetés et ne perdent pas seulement la récompense, mais perdent également l'effort déployé pour trouver une solution de preuve de travail, subissant ainsi sans compensation le coût de l'électricité.

Quand un nœud reçoit un nouveau bloc, il va valider le bloc en vérifiant une longue liste de critères qui doivent tous être respectés; sinon, le bloc est rejetée. Ces critères peuvent être vus dans le client Bitcoin Core dans les fonctions CheckBlock et CheckBlockHeader et comprennent :

- La structure de données du bloc est syntaxiquement valide
- Le hash d'entête de bloc est inférieure à la difficulté cible (met en œuvre la preuve de travail)
- Le timestamp du bloc est de moins de deux heures dans le future (permettant des erreurs de temps)
- La taille du bloc est dans des limites acceptables
- La première transaction (et seulement la première) est une transaction de génération coinbase
- Toutes les transactions dans le bloc sont valides, respectant les critères de validation de transaction discutés dans Vérification indépendante des transactions

La validation indépendante de chaque nouveau bloc par chaque nœud du réseau assure que les mineurs ne peuvent tricher. Dans les sections précédentes nous avons vu comment les mineurs arrivent à écrire une transaction qui leur accorde les nouveaux bitcoins créés au sein d'un bloc, de même que réclamer les frais de transaction. Pourquoi les mineurs n'écrivent-ils pas pour eux-même une transaction avec un millier de bitcoin au lieu de la bonne récompense ? Parce que chaque noeud valide les blocs selon des règles communes. Une transaction coinbase invalide rendrait l'ensemble du bloc invalide, ce qui se traduirait par le refus du bloc et, par conséquent, cette transaction ne serait jamais incluse dans le registre. Les mineurs doivent construire un bloc parfait, sur la base de règles communes que tous les nœuds suivent, et le miner avec une solution à la preuve de travail correcte. Pour ce faire, ils dépensent beaucoup d'électricité en minage, et si ils trichent, toute l'électricité et l'effort est gaspillé. Ceci est la raison pour laquelle la validation indépendante est une composante clé du consensus décentralisé.

Assemblage et sélection des chaînes de blocs

La dernière étape dans le mécanisme de consensus décentralisé de bitcoin est l'assemblage de blocs en chaînes et la sélection de la chaîne avec le plus de preuve de travail. Une fois qu'un nœud a validé un nouveau bloc, il tentera alors de former une chaîne en reliant le bloc à la blockchain existante.

Les nœuds maintiennent trois ensembles de blocs : ceux qui sont liés à la blockchain principale, ceux qui forment les branches qui dérivent de la blockchain principale (secondary chains), et enfin les blocs qui ne possèdent pas un parent connu dans les chaînes connues (orphans). Les blocs invalides sont rejetés dès qu'un des critères de validation échoue et ne sont alors pas inclus dans une chaîne.

La "chaîne principale" est, à n'importe quel point dans le temps, la chaîne de blocs qui a le plus difficulté cumulée. Dans la plupart des cas, c'est également la chaîne avec le plus grand nombre de bloc, sauf s'il y a deux chaînes de longueur égale, auquel cas l'une des deux aura davantage de preuve de travail. La chaîne principale aura également des branches avec des blocs qui sont « frères et sœurs » aux blocs de la chaîne principale. Ces blocs sont valables, mais ne font pas partie de la chaîne principale. Ils sont conservés pour consultation ultérieure, au cas où l'une de ces chaînes est étendue au point de dépasser en difficulté la chaîne principale. Dans la section suivante (Les forks de blockchain), nous allons voir comment les chaînes secondaires surviennent suite à un minage presque simultané de blocs de même hauteur.

Quand un nouveau bloc est reçu, un nœud va essayer de l'insérer dans la blockchain existante. Le nœud regardera le champ "hash du bloc précédent" du bloc, qui est la référence au nouveau parent du bloc. Ensuite, le nœud va tenter de trouver ce parent dans la blockchain existante. La plupart du temps, le parent sera le « sommet » de la chaîne principale, ce qui signifie que ce nouveau bloc étend la chaîne principale. Par exemple, le nouveau bloc 277 316 a une référence au hash de son bloc parent 277 315. La plupart des nœuds qui reçoivent 277 316 auront déjà le bloc 277 315 au sommet de leur chaîne principale et va donc lier le nouveau bloc et étendre cette chaîne.

Parfois, comme nous allons le voir dans Les forks de blockchain, le nouveau bloc étend une chaîne qui n'est pas la chaîne principale. Dans ce cas, le nœud va attacher le nouveau bloc à la chaîne secondaire qu'il étend, puis comparer la difficulté de la chaîne secondaire à la chaîne principale. Si la chaîne secondaire a davantage de difficulté cumulée que la chaîne principale, le noeud va *reconverger* sur la chaîne secondaire, ce qui signifie qu'il va choisir la chaîne secondaire comme sa nouvelle chaîne principale, faisant de la vieille chaîne principale une chaîne secondaire. Si le noeud est un mineur, il va maintenant construire un bloc étendant cette nouvelle chaîne, plus longue.

Si un bloc valide est reçu et qu'aucun parent ne se trouve dans les chaînes existantes, ce bloc est considéré comme un « orphelin ». Les blocs orphelins sont enregistrés dans la pool de bloc orphelin où ils resteront jusqu'à ce que leur parent soit reçu. Une fois que le parent est reçu et lié aux chaînes existantes, l'orphelin peut être retiré de la pool d'orphelins et lié au parent, l'ajoutant à la chaîne. Les blocs orphelins se produisent généralement lorsque deux blocs qui ont été miné dans un court laps de temps l'un de l'autre sont reçus dans l'ordre inverse (enfant avant parent).

En sélectionnant la chaîne de plus grande difficulté, tous les nœuds finissent par parvenir à un

consensus à l'échelle du réseau. Les écarts temporaires entre les chaînes sont résolus par la suite à mesure que plus de preuve de travail est ajouté, étendant une des chaînes possibles. Les nœuds de minage "votent" avec leur puissance de minage en choisissant quelle chaîne étendre en minant le bloc suivant. Quand ils extraient un nouveau bloc et étendent la chaîne, le nouveau bloc lui-même représente leur vote.

Dans la section suivante, nous allons voir comment les divergences entre chaînes concurrentes (forks) sont résolues par la sélection indépendante de la chaîne de difficulté la plus longue.

Les forks de blockchain

Parce que la blockchain est une structure de données décentralisée, différentes copies de celle-ci ne sont pas toujours identiques. Les blocs peuvent arriver à différents noeuds à différents moments, ce qui fait que les noeuds peuvent avoir une blockchain différente. Pour résoudre ce problème, chaque noeud sélectionne et tente toujours d'allonger la chaîne de blocs qui représente le plus de preuve de travail, aussi connue comme la chaîne la plus longue ou la chaîne de plus grande difficulté cumulée. En additionnant les difficultés enregistrées dans chaque bloc d'une chaîne, un nœud peut calculer le montant total de la preuve du travail qui a été dépensé pour créer cette chaîne. Tant que tous les nœuds sélectionnent la plus longue chaîne de difficulté cumulée, le réseau mondiale bitcoin converge vers un état cohérent. Les forks se produisent comme des incohérences temporaires entre les versions de la blockchain, qui sont résolus par une reconvergence à mesure que d'autres blocs sont ajoutés à l'un des forks.

Dans les prochains schémas, nous suivons la progression d'un « fork » à travers le réseau. Le diagramme est une représentation simplifiée de bitcoin en tant que réseau mondial. En réalité, la topologie du réseau bitcoin n'est pas organisé géographiquement. Au contraire, elle constitue un réseau maillé de noeuds interconnectés, lesquels peuvent être situés très loin l'une de l'autre sur le plan géographique. La représentation d'une topologie géographique est une simplification utilisée afin d'illustrer un fork. Dans le réseau bitcoin réel, la « distance » entre les nœuds est mesurée en « sauts » de noeud en noeud, et non selon leur emplacement physique. À titre d'illustration, les différents blocs sont représentés avec des couleurs différentes, se diffusant sur le réseau et colorant les connexions qu'ils traversent.

Dans le premier schéma (Visualisation d'un fork de blockchain – avant le fork), le réseau a une vue unique de la blockchain, avec le bloc bleu comme étant le sommet de la chaîne principale.

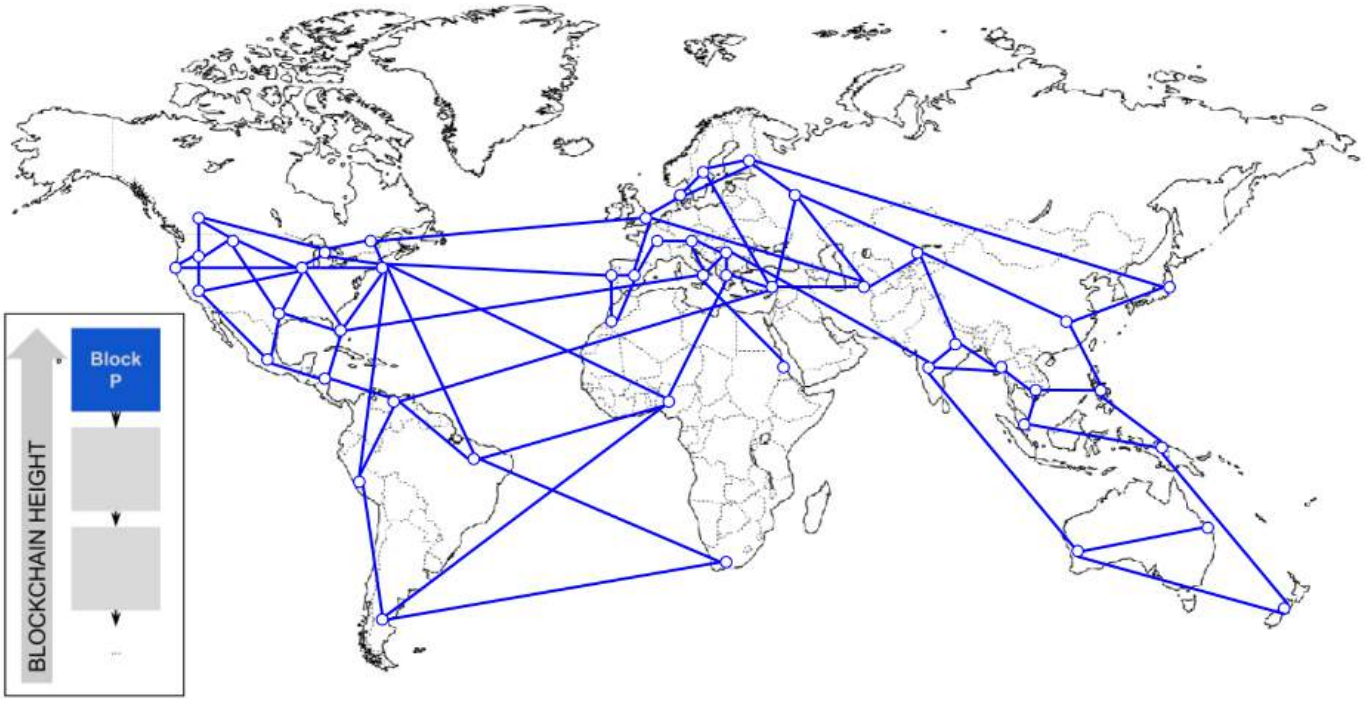


Figure 2. Visualisation d'un fork de blockchain – avant le fork

Un « fork » se produit chaque fois qu'il y a deux blocs candidats en compétition pour former la plus longue blockchain. Cela se produit dans des conditions normales à chaque fois que deux mineurs résolvent l'algorithme proof-of-work dans un court laps de temps l'un de l'autre. Comme les deux mineurs découvrent une solution pour leur bloc candidat respectif, ils ont immédiatement diffusé leur propre bloc « gagnant » à leurs voisins immédiats qui commencent à propager le bloc à travers le réseau. Chaque noeud qui reçoit un bloc valide l'incorpore dans sa blockchain, étendant la blockchain d'un bloc. Si ce nœud voit plus tard un autre bloc candidat étendant le même parent, il relie le deuxième candidat sur une chaîne secondaire. En conséquence, certains noeuds vont « voir » un bloc candidat en premier, tandis que d'autres nœuds verront un autre bloc candidat et deux versions concurrentes de la blockchain vont émerger.

Dans Visualisation d'un fork de blockchain : deux blocs trouvé simultanément, nous voyons deux mineurs qui extraient deux blocs différents presque simultanément. Ces deux blocs sont des enfants du bloc bleu, destiné à étendre la chaîne en se rajoutant sur le bloc bleu. Pour nous aider à le suivre, l'un est représenté par un bloc rouge provenant du Canada, l'autre par un bloc vert provenant d'Australie.

Supposons, par exemple, qu'un mineur au Canada trouve une solution de preuve de travail pour un bloc "rouge" qui étend la blockchain, construisant par dessus le bloc parent "bleu". Presque simultanément, un mineur australien qui était également en train d'élargir le bloc "bleu" trouve une solution pour le bloc "vert", son bloc candidat. Maintenant, il y a deux blocs possibles, celui que nous appelons « rouge », originaire du Canada, et celui que nous appelons « vert », originaire d'Australie. Les deux blocs sont valables, les deux blocs contiennent une solution valide à la preuve du travail, et les deux blocs étendent le même parent. Les deux blocs contiennent probablement les mêmes transactions pour la plupart, avec peut-être seulement quelques différences sur leur ordre.

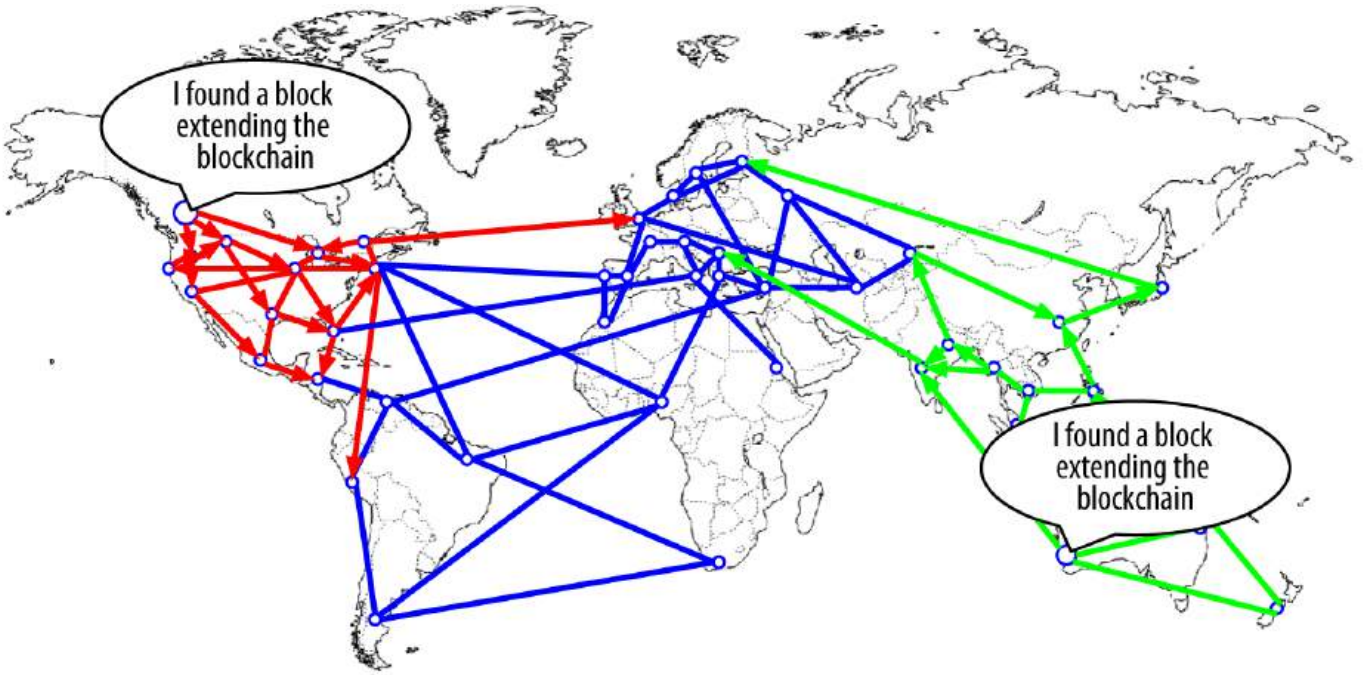


Figure 3. Visualisation d'un fork de blockchain : deux blocs trouvé simultanément

Comme les deux blocs se propagent, certains nœuds reçoivent le bloc "rouge" d'abord et certains reçoivent le bloc "vert" en premier. Comme le montre Visualisation d'un fork de blockchain : deux blocs se propagent, divisant le réseau, le réseau se divise en deux perspectives différentes de la blockchain, l'une avec un bloc rouge à son sommet, l'autre avec un bloc vert.

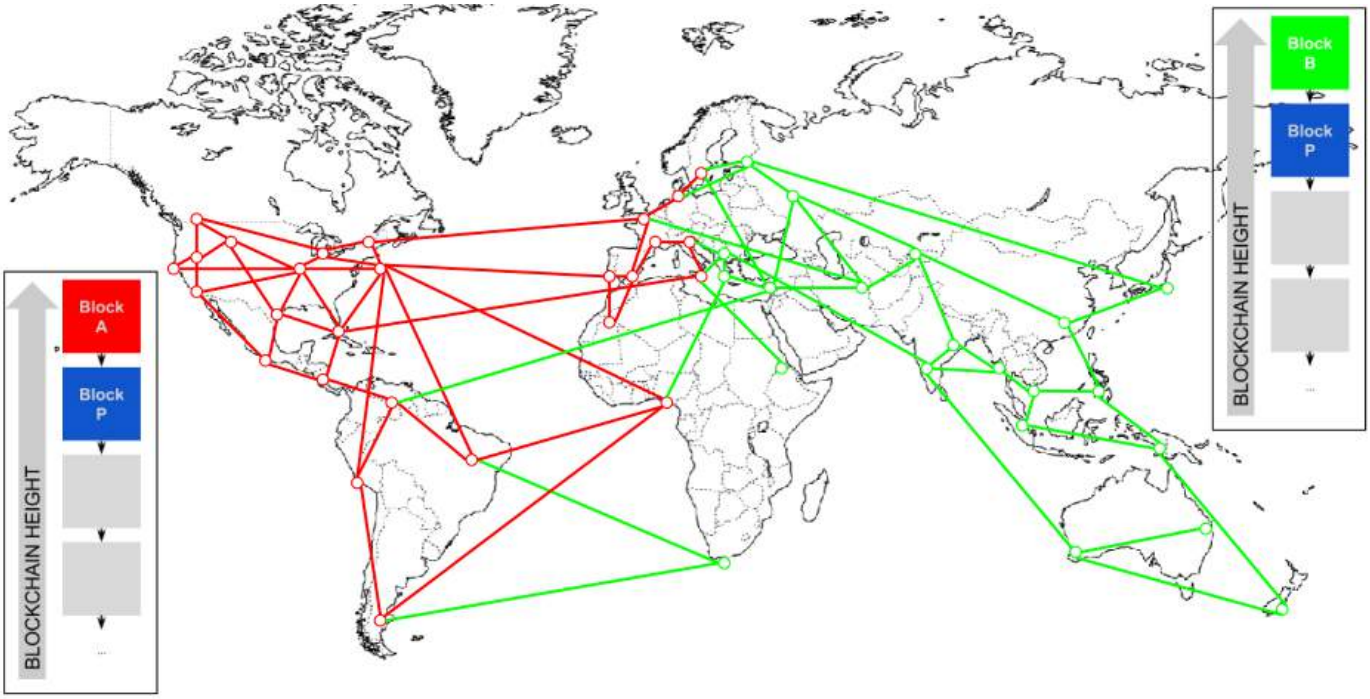


Figure 4. Visualisation d'un fork de blockchain : deux blocs se propagent, divisant le réseau

A partir de ce moment, les nœuds du réseau bitcoin les plus proche (topologiquement, pas géographiquement) du noeud canadien vont entendre parler bloc rouge d'abord et vont créer une nouvelle blockchain avec la plus grande difficulté cumulée avec le rouge comme le dernier bloc de la

chaîne (par exemple, bleu-rouge), en ignorant le bloc candidat vert qui arrive un peu plus tard. Pendant ce temps, les nœuds plus proches du noeud australien prendront ce bloc comme le vainqueur et prolongent la blockchain avec le vert comme le dernier bloc (par exemple, bleu-vert), en ignorant rouge qui arrive quelques secondes plus tard. Tous les mineurs qui ont vu le rouge en premier vont immédiatement construire des blocs candidats qui référence le rouge comme parent et commencent à essayer de résoudre la preuve de travail pour ces blocs candidats. Les mineurs qui ont accepté le vert à la place vont commencer à construire au-dessus du vert et étendre cette chaîne.

Les forks sont presque toujours résolus en un seul bloc. Quand une partie de la puissance de hachage du réseau est dédiée à construire au dessus du rouge comme parent, une autre partie de la puissance de hachage est concentrée à bâtir au dessus du vert. Même si la puissance de hachage est presque également divisée, il est probable qu'un ensemble de mineurs trouve une solution et la propage avant que l'autre ensemble n'aient trouvé une quelconque solution. Disons, par exemple, que les mineurs construisant au-dessus du vert trouvent un nouveau bloc rose qui étend la chaîne (par exemple, bleu-vert-rose). Ils propagent immédiatement ce nouveau bloc et l'ensemble du réseau le voit comme une solution valide comme indiqué dans Visualisation d'un fork de blockchain : un nouveau bloc étend un fork.

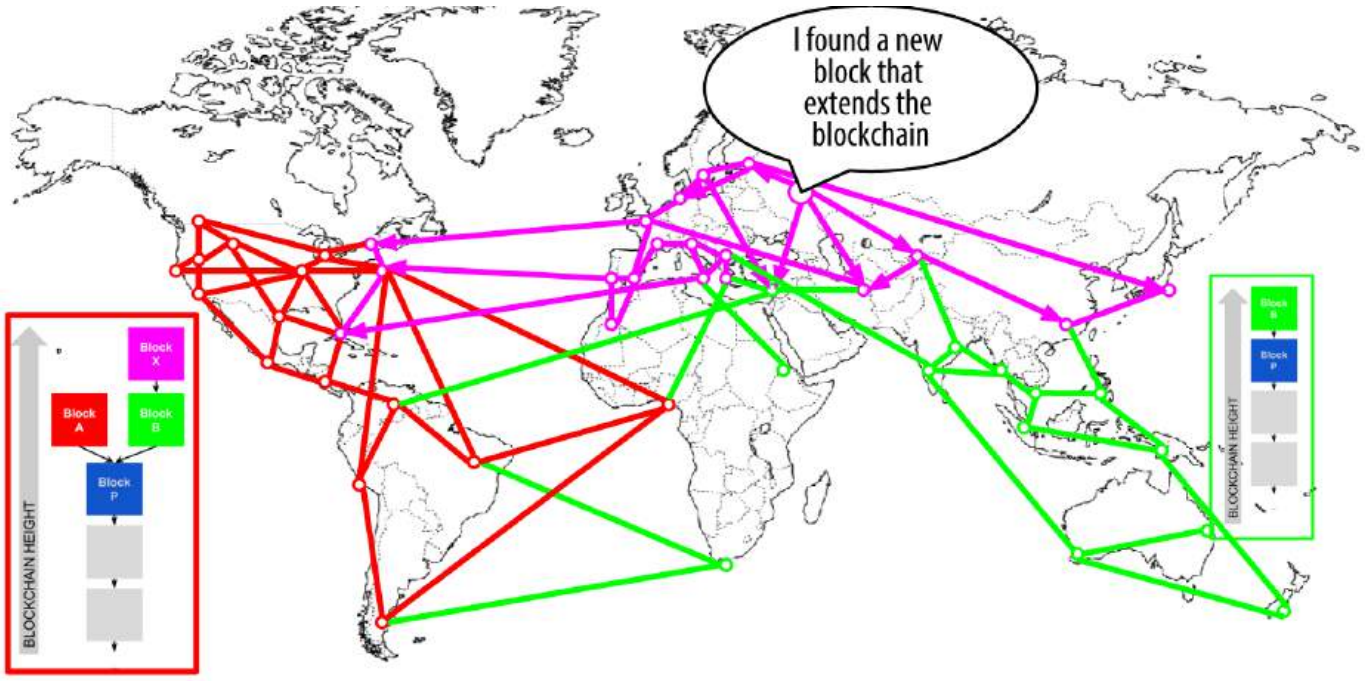


Figure 5. Visualisation d'un fork de blockchain : un nouveau bloc étend un fork

Tous les nœuds qui avaient choisi le vert comme le gagnant au tour précédent étendront simplement la chaîne d'un autre bloc. Les nœuds qui ont choisi le rouge comme le vainqueur, cependant, voient maintenant deux chaînes : bleu-vert-rose et bleu-rouge. La chaîne bleu-vert-rose est maintenant plus longue (plus de difficulté cumulée) que la chaîne bleu-rouge. En conséquence, ces nœuds vont établir la chaîne bleu-vert-rose comme chaîne principale et changer la chaîne bleu-rouge en chaîne secondaire, comme indiqué dans Visualisation d'un fork de blockchain : le réseau reconverge sur une nouvelle plus longue chaîne. Ceci est une reconvergence de chaîne, parce que ces nœuds sont obligés de réviser leur vue de la blockchain pour incorporer la nouvelle preuve d'une chaîne plus longue. Tous

les mineurs qui travaillent sur l'extension de la chaîne bleu-rouge va maintenant arrêter ce travail parce que leur bloc candidat est un « orphelin », comme son parent "rouge" n'est plus sur la plus longue chaîne. Les transactions au sein du rouge sont à nouveau remis dans la file d'attente pour être traiter dans le bloc suivant, parce que ce bloc n'est plus dans la chaîne principale. Le réseau entier re-converge sur une seule blockchain bleu-vert-rose, avec le rose comme dernier bloc de la chaîne. Tous les mineurs commencent immédiatement à travailler sur des blocs candidats qui font référence au rose comme leur parent pour allonger la chaîne bleu-vert-rose.

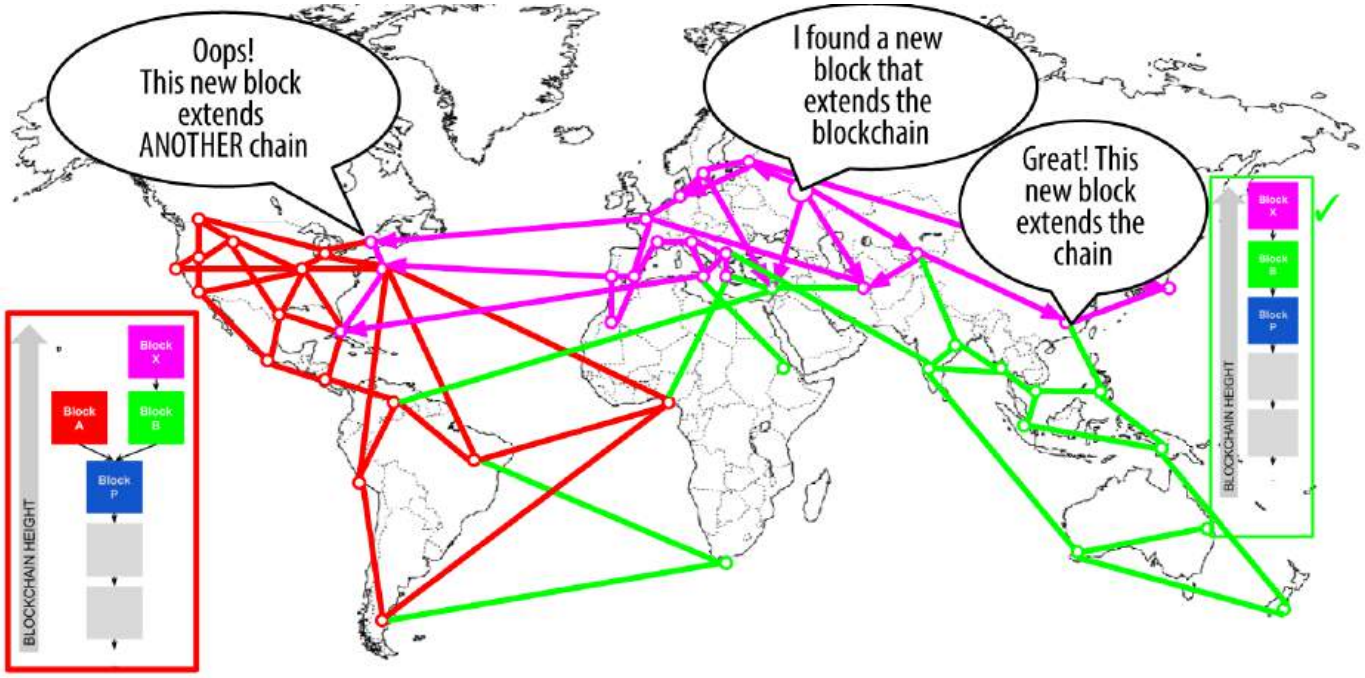


Figure 6. Visualisation d'un fork de blockchain : le réseau reconverge sur une nouvelle plus longue chaîne

Il est théoriquement possible pour un fork de s'étendre sur deux blocs, si deux blocs sont trouvés presque simultanément par les mineurs sur les "côtés" opposés d'un fork précédent. Cependant, la chance que cela se produise est très faible. Alors qu'un fork d'un bloc peut se produire chaque semaine, un fork à deux blocs est extrêmement rare.

L'intervalle de bloc de bitcoin de 10 minutes est un compromis entre des temps de confirmation rapides (de règlement des transactions) et la probabilité d'un fork. Un temps de bloc plus rapide rendrait la confirmation des transactions plus rapide, mais conduirait à des forks de blockchain plus fréquents, alors qu'un temps de bloc plus lent diminuerait le nombre de forks mais rendrait la confirmation plus lente.

Le minage et la course au hachage

Le minage de bitcoin est une industrie extrêmement compétitive. La puissance de hachage a augmenté de façon exponentielle d'année en année depuis l'existence de bitcoin. Certaines années la croissance a été le reflet d'un changement de technologie, comme en 2010 et 2011 où de nombreux mineurs sont passés du minage à CPU au minage à GPU et au "field programmable gate array" (FPGA). En 2013, l'introduction du minage à ASIC à conduit à un autre bond de géant dans la puissance de minage, en

plaçant la fonction SHA256 directement dans des puces de silicium spécialement conçues pour le minage. Les premières puces de la sorte pourraient fournir plus de puissance de minage que le réseau bitcoin dans son ensemble en 2010.

La liste suivante montre la puissance totale de hachage du réseau bitcoin, au cours des cinq premières années de fonctionnement :

2009

0.5 MH/sec–8 MH/sec (16× growth)

2010

8 MH/sec–116 GH/sec (14,500× growth)

2011

16 GH/sec–9 TH/sec (562× growth)

2012

9 TH/sec–23 TH/sec (2.5× growth)

2013

23 TH/sec–10 PH/sec (450× growth)

2014

10 PH/sec–150 PH/sec in August (15× growth)

Dans le tableau dans Puissance totale de hachage, gigahashes par seconde, sur deux ans, nous voyons l'augmentation de la puissance de hachage du réseau bitcoin au cours des deux dernières années. Comme vous pouvez le voir, la compétition entre les mineurs et la croissance de bitcoin a entraîné une augmentation exponentielle de la puissance de hachage (hashs totaux par seconde à travers le réseau).

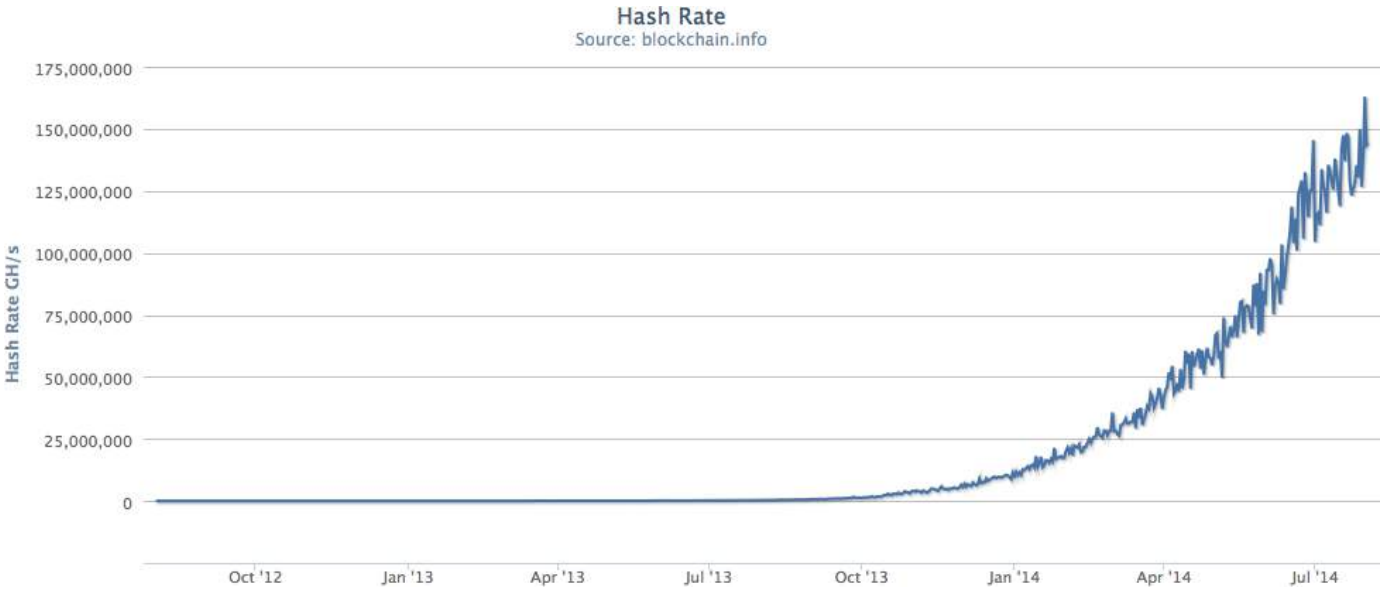


Figure 7. Puissance totale de hachage, gigahashes par seconde, sur deux ans

Comme la quantité de puissance de hachage appliquée au minage de bitcoin a explosé, la difficulté a augmenté en conséquence. L'indicateur de difficulté dans le tableau Indicateur de la difficulté du minage de bitcoin, sur deux ans est mesuré comme le rapport de difficulté actuel par rapport à la difficulté minimum (la difficulté du premier bloc).

Figure 8. Indicateur de la difficulté du minage de bitcoin, sur deux ans

Au cours des deux dernières années, les puces ASIC de minage sont devenues de plus en plus denses, approchant ce qui se fait de mieux en matière de puce en silicium avec une taille (résolution) de 22 nanomètres (nm). Actuellement, les fabricants d'ASIC visent à dépasser les fabricants de puces CPU à usage général, dessinant des puces avec une taille de 16nm, car la rentabilité du minage fait évoluer cette industrie encore plus vite que l'informatique en général. Il n'y a plus de sauts de géant dans le minage de bitcoin parce que l'industrie a atteint la pointe de la loi de Moore, qui stipule que la densité de calcul va doubler tous les 18 mois environ. Pourtant, la puissance de minage du réseau continue de progresser à un rythme exponentiel; à la course aux puces de plus haute densité correspond une course pour des datacenters de densité plus élevée où des milliers de ces puces peuvent être déployées. La question n'est plus de savoir combien de minage peut être fait avec une puce, mais combien de puces peuvent rentrer dans un bâtiment, tout en dissipant la chaleur et en fournissant une puissance suffisante.

La solution du nonce supplémentaire

Depuis 2012, le minage de bitcoin a évolué pour résoudre une limitation fondamentale dans la structure de l'entête de bloc. Dans les premiers temps de bitcoin, un mineur pouvait trouver un bloc en itérant le nonce jusqu'à ce que le hachage résultant soit en dessous de la cible. Comme la difficulté a augmenté, les mineurs enchaînaient souvent toutes les 4 milliards de valeurs du nonce sans trouver un bloc. Cependant, cela a été facilement résolu en mettant à jour le timestamp du bloc pour tenir compte du temps écoulé. Parce que le timestamp est une partie de l'entête, le changement permettrait aux mineurs de parcourir les valeurs du nonce à nouveau avec des résultats différents. Une fois que le matériel de minage dépassa 4 GH/sec, cependant, cette approche est devenue de plus en plus difficile parce que les valeurs de nonce étaient épuisées en moins d'une seconde. Comme l'équipement minier

ASIC a commencé à se développer et à dépasser le TH/sec de taux de hachage, le logiciel de minage avait besoin de plus d'espace pour les valeurs de nonce afin de trouver des blocs valides. Le timestamp pourrait être étiré un peu, mais le déplacer trop loin dans l'avenir aurait pour conséquence de rendre le bloc invalide. Une nouvelle source de « changement » était nécessaire dans l'entête de bloc. La solution fut d'utiliser la transaction coinbase comme une source de valeurs de nonce supplémentaires. Parce que le script coinbase peut stocker entre 2 et 100 octets de données, les mineurs ont commencé à utiliser cet espace comme espace de nonce supplémentaire, leur permettant d'explorer un éventail beaucoup plus large de valeurs d'entête de bloc pour trouver des blocs valides. La transaction coinbase est inclus dans l'arbre de Merkle, ce qui signifie que tout changement dans le script coinbase provoque une modification de la racine de Merkle. Huit octets de nonce supplémentaires, ainsi que les 4 octets de nonce "standard" permettent aux mineurs d'explorer un total de 2^{96} (8 suivi de 28 zéros) possibilités *par seconde* sans avoir à modifier le timestamp. Si, à l'avenir, les mineurs pouvaient parcourir toutes ces possibilités, ils pourraient alors modifier le timestamp. Il y a aussi plus d'espace dans le script coinbase pour une future expansion de l'espace de nonce supplémentaire.

Les pools de minage

Dans cet environnement hautement concurrentiel, les mineurs individuels travaillant seuls (également connus comme "mineurs solo") n'ont aucune chance. Pour eux, la probabilité de trouver un bloc qui compensera leurs coûts d'électricité et de matériel est si faible qu'elle représente un pari risqué, comme jouer à la loterie. Même le plus rapide des systèmes de minage ASIC pour particulier ne peut rivaliser avec les systèmes commerciaux qui empilent des dizaines de milliers de ces puces dans des entrepôts géants près de centrales hydro-électriques. Les mineurs collaborent désormais pour former des pools de minage, mettant en commun leur puissance de hachage et se partageant la récompense entre des milliers de participants. En participant à un pool, les mineurs reçoivent une plus petite part de la récompense globale, mais sont généralement récompensés chaque jour, réduisant l'incertitude.

Prenons un exemple concret. Supposons qu'un mineur a acheté du matériel de minage avec un taux de hachage total de 6000 gigahashes par seconde (GH/s), soit 6 TH/s. En Août 2014 cet équipement coûtait environ 10 000 $. Le matériel consomme 3 kilowatts (kW) d'électricité lors de l'exécution, 72 kW-heures par jour, à un coût de 7 $ ou 8 $ par jour en moyenne. Avec la difficulté bitcoin en cours, le mineur sera en mesure de miner en solo un bloc environ une fois tous les 155 jours, soit tous les 5 mois. Si le mineur trouve un seul bloc dans ce délai, le paiement de 25 bitcoins, à environ 600 $ par bitcoin, se traduira en un seul versement de 15 000 dollars, qui couvrira la totalité du coût du matériel et de l'électricité consommée au cours de cette période, laissant un bénéfice net d'environ 3000 $. Cependant, les chances de trouver un bloc dans une période de cinq mois dépendent de la chance du mineur. Il pourrait trouver deux blocs en cinq mois et faire un très grand profit. Ou il pourrait ne pas trouver un bloc pendant 10 mois et subir une perte financière. Pire encore, la difficulté de l'algorithme bitcoin proof-of-work est susceptible de monter de façon significative au cours de cette période, au rythme actuel de la montée de la puissance de hachage, ce qui signifie que le mineur a, au plus, six mois pour trouver un bloc avant que le matériel ne devienne obsolète et doive être remplacé par du matériel de minage plus puissant. Si ce mineur participe à un pool de minage, au lieu d'attendre une fois tous les cinq mois une aubaine de 15 000 $, il sera en mesure de gagner environ 500 $ à 750 $ par semaine. Les paiements réguliers d'une pool de minage vont l'aider à amortir le coût du matériel et de l'électricité au fil du temps sans prendre un risque énorme. Le matériel sera toujours obsolète dans six

à neuf mois et le risque est encore élevé, mais le chiffre d'affaires sera au moins régulier et fiable au cours de cette période.

Les pools de minage coordonnent plusieurs centaines ou milliers de mineurs, sur des protocoles spécialisés de minage en pool. Les mineurs individuels configurent leurs équipements de minage pour se connecter à un serveur de pool, après avoir créé un compte avec la pool. Leur matériel de minage reste connecté au serveur de pool pendant le minage, synchronisant leur effort avec les autres mineurs. Ainsi, les mineurs en pool se partagent l'effort du minage de bloc et se partage ensuite les récompenses.

Les blocs gagnants paient la récompense à une adresse bitcoin de la pool, plutôt qu'aux mineurs individuels. Le serveur de pool va périodiquement effectuer des paiements aux adresses bitcoin des mineurs, une fois leur part des récompenses a atteint un certain seuil. Typiquement, le serveur de pool exige un certain pourcentage des récompenses pour fournir le service de pool de minage.

Les mineurs participant à une pool se partagent le travail de recherche d'une solution d'un bloc candidat, gagnant des « parts » pour leur contribution au minage. La pool de minage se fixe un objectif de difficulté inférieure pour gagner une part, généralement plus de 1000 fois plus facile que la difficulté du réseau bitcoin. Quand quelqu'un dans la pool a miné un bloc avec succès, la récompense est gagnée par la pool et ensuite partagée avec tous les mineurs en proportion du nombre de parts qu'ils ont apporté à l'effort commun.

Les pools sont ouvertes à tout mineur, grand ou petit, professionnel ou amateur. Une pool aura par conséquent des participants avec une seule petite machine de minage, et d'autres avec un garage plein de matériel haut de gamme. Certains vont miner avec quelques dizaines de kilowatts d'électricité, d'autres piloteront un datacenter consommant un mégawatt d'électricité. Comment une pool de minage mesure les contributions individuelles, de manière à répartir équitablement les récompenses, sans possibilité de tricherie ? La réponse est d'utiliser l'algorithme proof-of-work, mais fixé à une difficulté inférieure, de sorte que même les plus petits mineurs de la pool gagnent une part suffisamment fréquemment pour qu'il continue de valoir la peine de contribuer à la pool. En fixant une difficulté inférieure afin de gagner des parts, la pool mesure la quantité de travail effectuée par chaque mineur. Chaque fois qu'un mineur de la pool trouve un hash d'entête de bloc qui est inférieure à la difficulté de la pool, il prouve qu'il a fait le travail de hachage pour trouver ce résultat. Plus important encore, le travail pour trouver des parts contribue, d'une manière statistiquement mesurable, à l'effort global pour trouver un hachage inférieur à la cible du réseau bitcoin. Des milliers de mineurs essayant de trouver des hashs de faible valeur finiront par en trouver un assez faible pour satisfaire la cible du réseau bitcoin.

Revenons à l'analogie du jeu de dés. Si les joueurs de dés jettent les dés avec pour objectif de jeter un résultat de moins de quatre (la difficulté globale du réseau), une pool fixerait une cible plus facile, comptant combien de fois les joueurs de la pool ont réussi à jeter moins de huit. Lorsque les joueurs de la pool jettent moins de huit (la cible de la pool), ils gagnent des parts, mais ils ne gagnent pas le jeu parce qu'ils ne réalisent pas la cible du jeu (moins de quatre). Les joueurs en pool vont atteindre la cible plus facile de la pool beaucoup plus souvent, gagnant des parts très régulièrement, même lorsqu'ils ne réalisent pas l'objectif plus difficile de gagner le jeu. De temps à autre, l'un des joueurs jettera moins de quatre et la pool gagnera. Ensuite, les gains peuvent être distribués aux joueurs de la

pool sur la base des parts qu'ils ont gagnés. Même si la cible de huit ou moins ne gagnait pas, c'est une manière équitable de mesurer les dés jetés pour les joueurs, et il produit parfois un jet de moins de quatre.

De même, une pool de minage fixera une difficulté de pool qui fera en sorte qu'un mineur de pool peut assez souvent trouver un hash d'entête de bloc qui soit moins que la difficulté de la pool, gagnant des parts. De temps en temps, une de ces tentatives va produire un hash d'entête de bloc qui est inférieure à la cible du réseau bitcoin, ce qui en fait un bloc valide et l'ensemble de la pool gagne.

Les pools gérées

La plupart des pools de minage sont « gérées », ce qui signifie qu'il y a une entreprise ou une personne exécutant un serveur de pool. Le propriétaire du serveur est appelé *opérateur de pool*, il facture aux mineurs en pool un pourcentage des gains.

Le serveur de pool fonctionne avec un logiciel spécialisé et un protocole de minage en pool qui coordonne les activités des mineurs de la pool. Le serveur est également connecté à un ou plusieurs nœuds bitcoin complets et dispose d'un accès direct à une copie complète de la base de données blockchain. Cela lui permet de valider des blocs et des transactions au nom des mineurs de la pool, leur évitant de devoir exécuter un noeud complet. Pour les mineurs en pool, cela est une considération importante, car un noeud complet nécessite un ordinateur dédié avec au moins 15 à 20 Go de stockage persistant (de disque dur) et au moins 2 Go de mémoire vive (RAM). En outre, le logiciel bitcoin s'exécutant sur un noeud complet doit être surveillé, entretenu et mis à jour fréquemment. Tous les temps d'arrêt causés par un manque d'entretien ou par un manque de ressources affectera les revenus du mineur. Pour de nombreux mineurs, la capacité à miner sans exécuter un noeud complet est un autre des grands avantages de rejoindre une pool gérée.

Les mineurs en pool se connectent au serveur de la pool en utilisant un protocole de minage tels que Stratum (STM) ou GetBlockTemplate (GBT). Une ancienne norme appelée GetWork (GWK) est quasiment obsolète depuis fin 2012, car elle ne supporte pas facilement le minage à taux de hachage dépassant les 4 GH/s. Tant les protocoles STM que GBT créent des *templates* de bloc qui contiennent un modèle d'entête de bloc candidat. Le serveur de pool construit un bloc candidat en agrégeant les transactions, en ajoutant une transaction coinbase (avec un espace de nonce en plus), le calcul de la racine de Merkle, et un lien vers le hash du bloc précédent. L'entête du bloc candidat est ensuite envoyé à chacun des mineurs de la pool comme un modèle. Chaque mineur de la pool mine alors à l'aide du template de bloc, à une difficulté inférieure à la difficulté de réseau bitcoin, et envoie chaque résultat positif au serveur de pool pour gagner des parts.

P2Pool

Les pools gérées créent une possibilité de tricherie de la part de l'opérateur de pool, qui pourraient diriger l'effort de la pool pour créer des transactions dépensées deux fois ou invalider des blocs (voir Les attaques par consensus). En outre, le modèle centralisé des serveurs de pool représente une possibilité de faille concentrée sur un point unique. Si le serveur de pool est en panne ou est ralenti par une attaque par déni de service, les mineurs en pool ne peuvent plus miner. En 2011, pour résoudre ces problèmes de centralisation, une nouvelle méthode de minage en pool a été proposé et mis en

œuvre : P2Pool est une pool de minage en peer-to-peer, sans opérateur central.

Une P2Pool fonctionne en décentralisant les fonctions du serveur de pool, mettant en œuvre un système parallèle de blockchain appelé un *share chain* ou *chaîne de parts*. Une chaîne de parts est une blockchain fonctionnant à une difficulté moindre que la blockchain bitcoin. La chaîne de parts permet aux mineurs en pool de participer à une pool décentralisée, en minant des parts sur la chaîne de parts au taux d'un bloc de part toutes les 30 secondes. Chacun des blocs sur la chaîne de parts enregistre une part de récompense proportionnée pour les mineurs en pool qui contribuent au travail, faisant progresser les parts à partir du bloc de part précédent. Lorsque l'un des blocs de part réalise également la cible de difficulté du réseau bitcoin, il se propage et est inclus dans la blockchain bitcoin, récompensant tous les mineurs de la pool qui ont contribué à toutes les parts qui ont précédé le bloc de part gagnant. Essentiellement, au lieu d'un serveur de pool qui garde trace des parts et des récompenses des mineurs en pool, la chaîne de parts permet à tous les mineurs de la pool de garder une trace de toutes les parts en utilisant un mécanisme de consensus décentralisé similaire au mécanisme de consensus de la blockchain bitcoin.

Le minage en P2Pool est plus complexe que le minage en pool car il nécessite que les mineurs aient un ordinateur dédié avec suffisamment d'espace disque, de mémoire et de bande passante internet pour soutenir un nœud bitcoin complet ainsi que le logiciel destiné aux noeuds P2Pool. Les mineurs P2Pool connectent leur matériel de minage à leur noeud P2Pool local, qui simule les fonctions d'un serveur de pool en envoyant les modèles de blocs au matériel de minage. En P2Pool, chaque mineur construit ses propres blocs candidats, agrégeant les transactions à la manière des mineurs solo, mais mine sur la chaîne de parts. Le P2Pool est une approche hybride qui a l'avantage de permettre des paiements beaucoup plus granulaire que le minage en solo, mais sans donner trop de contrôle à un opérateur de pool comme dans les pool gérées.

Récemment, la participation en P2Pool a augmenté de façon significative étant donné que la concentration du minage dans les pools de minage a atteint un niveau soulevant des inquiétudes quant à une attaque 51% (voir Les attaques par consensus). Le développement du protocole P2Pool se poursuit avec l'espoir d'éliminer la nécessité d'exécuter un nœud complet et donc faire que le minage décentralisé soit encore plus facile à utiliser.

Même si le P2Pool réduit la concentration du pouvoir chez les opérateurs de pools, il reste vulnérable à une attaque 51% contre la chaîne de parts elle-même. Une adoption plus large de P2Pool ne résout pas le problème d'attaque 51% pour bitcoin lui-même. Plutôt, le P2Pool fait de bitcoin un système plus robuste dans l'ensemble, en tant que partie d'un écosystème de minage riche.

Les attaques par consensus

Le mécanisme de consensus bitcoin est, au moins théoriquement, vulnérable aux attaques des mineurs (ou des pools) qui tentent d'utiliser leur puissance de hachage à des fins malhonnêtes ou destructrices. Comme nous l'avons vu, le mécanisme de consensus dépend de la présence d'une majorité de mineurs qui agissent honnêtement par intérêt personnel. Toutefois, si un mineur ou un groupe de mineurs peuvent obtenir une part importante de la puissance de minage, ils peuvent attaquer le mécanisme de consensus afin de perturber la sécurité et la disponibilité du réseau bitcoin.

Il est important de noter que les attaques par consensus ne peuvent affecter qu'un consensus futur, ou, au mieux, le passé le plus récent (quelques dizaines de blocs). Le registre bitcoin devient de plus en plus immuable à mesure que le temps passe. Si, en théorie, un fork peut être réalisé à toute profondeur, dans la pratique, la puissance de calcul nécessaire pour forcer un fork très profond est immense, ce qui rend les anciens blocs pratiquement immuable. Aussi, les attaques par consensus n'affecte pas la sécurité de la clé privée et l'algorithme de signature (ECDSA). Une attaque par consensus ne peut pas voler des bitcoins, dépenser des bitcoins sans signatures, rediriger des bitcoins, modifier des transactions passées ou des preuves de propriété. Les attaques par consensus ne peuvent affecter que les blocs les plus récents et causer des perturbations par déni de service sur la création des blocs à venir.

Un scénario d'attaque contre le mécanisme de consensus est appelé "l'attaque 51%". Dans ce scénario, un groupe de mineurs, contrôlant la majorité (51%) de la puissance de hachage de l'ensemble du réseau, s'entendent pour attaquer bitcoin. Avec la possibilité de miner la majorité des blocs, les mineurs attaquant peuvent provoquer des forks volontairement dans la blockchain, et dépenser des transactions deux fois ou exécuter des attaques par déni de service contre des transactions ou des adresses spécifiques. Une attaque fork/double-dépense est celle où l'attaquant rend des blocs déjà confirmés invalides en forkant en dessous d'eux et en reconvergeant sur une chaîne alternative. Avec une puissance suffisante, un attaquant peut invalider six blocs ou plus d'affilée, ayant pour conséquence de rendre des transactions considérées comme immuables (six confirmations) invalides. Notez qu'une double dépenses ne peut être fait que sur les transactions appartenant à l'attaquant, pour lesquelles l'attaquant peut produire une signature valide. Faire des transactions de double dépenses peut être rentable si en invalidant une transaction l'attaquant peut obtenir un paiement non réversible ou un produit sans avoir à payer pour cela.

Examinons un exemple pratique d'attaque 51%. Dans le premier chapitre, nous avons vu une transaction entre Alice et Bob pour une tasse de café. Bob, le propriétaire du café, est prêt à accepter le paiement de tasses de café sans attendre la confirmation (le minage dans un bloc), parce que le risque de double-dépense pour une tasse de café est faible en comparaison de la commodité d'un service clientèle rapide . Ceci est similaire à la pratique des magasins qui acceptent les paiements par carte de crédit sans signature pour des montants inférieurs à 25 $, car le risque d'un rejet de débit de carte de crédit est faible alors que le coût de retardement de la transaction pour obtenir une signature est en comparaison plus grande. En revanche, la vente d'un article plus cher en bitcoins court le risque d'une attaque double-dépenses, où l'acheteur diffuse une transaction concurrente qui envoie les mêmes entrées (UTXO) et annule le paiement au marchand. Un attaque double-dépenses peut arriver de deux façons : soit avant qu'une transaction soit confirmée, ou si l'attaquant profite d'un fork de blockchain pour annuler plusieurs blocs. Une attaque 51% permet à des attaquants à dépenser deux fois leurs propres transactions dans la nouvelle chaîne, annulant ainsi la transaction correspondante dans la vieille chaîne.

Dans notre exemple, Mallory, un attaquant malveillant, va à la galerie de Carol et achète une belle peinture en triptyque représentant Satoshi Nakamoto comme Prométhée. Carol vend les peintures "The Great Fire" pour 250 000 dollars en bitcoins à Mallory. Au lieu d'attendre six confirmations ou plus sur la transaction, Carol enveloppe et donne les peintures à Mallory après une seule confirmation. Mallory agit avec un complice, Paul, qui exploite une grande pool de minage, et le complice lance une

attaque 51% dès que la transaction de Mallory est incluse dans un bloc. Paul administre la pool de minage de manière à reminer un bloc de même hauteur que le bloc contenant la transaction de Mallory, remplaçant le paiement de Mallory à Carol avec une transaction qui double dépense la même entrée que le paiement de Mallory. La transaction double-dépenses consomme le même UTXO et paie de nouveau au portefeuille de Mallory, au lieu de payer celui de Carol, permettant à Mallory de garder les bitcoins. Paul dirige ensuite la pool de minage pour miner un bloc supplémentaire, afin de rendre la chaîne contenant la transaction double-dépenses plus vielle que la chaîne originale (provoquant un fork en dessous du bloc contenant la transaction de Mallory). Lorsque le fork de blockchain est résolu en faveur de la nouvelle (plus longue) chaîne, la transaction double-dépenses remplace le paiement initial à Carol. Carol perd alors les trois tableaux et n'a pas non plus le paiement de bitcoins. Tout au long de cette activité, les participants de la pool de minage de Paul pourraient rester parfaitement ignorant de la tentative de double-dépenses, parce qu'ils minent avec des mineurs automatisés et ne peuvent pas surveiller chaque transaction ou bloc.

Pour se protéger contre ce type d'attaque, un commerçant vendant des articles de grande valeur doit attendre au moins six confirmations avant de donner le produit à l'acheteur. Alternativement, le commerçant doit utiliser un entiercement grâce à compte multi-signature, et toujours attendre plusieurs confirmations après que le compte soit financé. Plus il y a de confirmations, plus il devient difficile d'invalider une transaction avec une attaque 51%. Pour les articles de grande valeur, le paiement par bitcoin sera toujours pratique et efficace même si l'acheteur doit attendre 24 heures pour la livraison, ce qui assurerait 144 confirmations.

En plus d'une attaque double-dépenses, l'autre scénario pour une attaque par consensus est de provoquer un déni de service contre des participants du réseau bitcoin (adresses bitcoin spécifiques). Un attaquant disposant d'une majorité de la puissance de minage peut simplement ignorer des transactions spécifiques. Si elles sont incluses dans un bloc miné par un autre mineur, l'attaquant peut délibérément forker et re-miner ce bloc, toujours en ignorant des transactions spécifiques. Ce type d'attaque peut entraîner un déni de service soutenu contre une adresse spécifique ou un ensemble d'adresses pour aussi longtemps que l'attaquant contrôle la majorité de la puissance de minage.

En dépit de son nom, le cas d'une attaque 51% ne nécessite pas réellement 51% de la puissance de hachage. En fait, une telle attaque peut être tentée avec un pourcentage plus faible. Le seuil de 51% est tout simplement le niveau auquel une telle attaque est presque assurée de réussir. Une attaque par consensus est essentiellement un combat pour le prochain bloc et le groupe le "plus fort" est plus susceptible de gagner. Avec moins de puissance de hachage, la probabilité de succès est réduite, parce que d'autres mineurs contrôlent la génération de certains blocs avec leur puissance de minage "honnête". Une façon de voir les choses est que plus un attaquant a de puissance de hachage, plus le fork qu'il peut créer sera long, plus les blocs récemment inclus pourront être invalider ou plus il peut contrôler de blocs à venir. Des groupes de recherche en sécurité ont utilisé la modélisation statistique pour prétendre que divers types d'attaques consensuelles sont possibles avec aussi peu que 30% de la puissance de hachage.

L'augmentation massive de la puissance totale de hachage a sans doute rendu bitcoin imperméable aux attaques par un mineur unique. Il n'y a aucune voie possible pour un mineur en solo de contrôler plus qu'un petit pourcentage de la puissance totale de minage. Cependant, la centralisation du contrôle

causée par les pools minières a introduit le risque d'attaques à but lucratif par un opérateur de pool. L'opérateur de pool dans une pool gérée contrôle la construction de blocs candidats et contrôle les transactions qui sont également incluses. Cela donne à l'opérateur de la pool le pouvoir d'exclure des transactions ou d'introduire des transactions double-dépenses. Si un tel abus de pouvoir est fait de manière limitée et subtile, un opérateur de pool pourrait éventuellement bénéficier d'une attaque par consensus sans être remarqué.

Tous les attaquants ne seront pas motivés par le profit cependant. Un scénario d'attaque potentiel est celui où un attaquant a l'intention de perturber le réseau bitcoin sans la possibilité de profiter de cette perturbation. Une attaque malveillante visant à paralyser bitcoin nécessiterait d'énormes investissements et une planification secrète, mais pourrait éventuellement être lancé par un très riche attaquant, probablement financé par un Etat. Alternativement, un riche attaquant pourrait attaquer le consensus bitcoin en amassant simultanément le matériel de minage, en compromettant les opérateurs de pools et en attaquant d'autres pools avec déni de service. Tous ces scénarios sont théoriquement possible, mais de plus en plus impraticable à mesure que la puissance de hachage globale du réseau bitcoin continue à croître de façon exponentielle.

Sans aucun doute, une attaque par consensus grave éroderait la confiance en bitcoin dans le court terme, pouvant provoquer une baisse significative des prix. Cependant, le réseau et le logiciel bitcoin sont en constante évolution, de sorte que les attaques par consensus seraient adressées avec des contre-mesures immédiates par la communauté bitcoin, faisant de bitcoin un système plus robuste que jamais.

Chaînes alternatives, Monnaies, <phrase role="keep-together"> et Applications</phrase>

Bitcoin est le résultat de 20 années de recherche dans le domaine des systèmes distribués et des monnaies, et a mis au jour une technologie révolutionnaire : un mécanisme de consensus décentralisé fondé sur une preuve de travail. Cette invention au coeur de bitcoin a donné naissance à une vague d'innovations dans la monnaie, les services financiers, les sciences économiques, les systèmes distribués, les systèmes de vote, la gouvernance d'entreprise, et la gestion des contrats.

Dans ce chapitre, nous allons examiner les nombreuses conséquences de l'invention du bitcoin et de la blockchain : les chaînes alternatives, les monnaies, et des applications développées depuis l'avènement de cette technologie en 2009. Nous nous concentrerons essentiellement sur les monnaies alternatives, ou *altcoins*, qui sont des monnaies numériques implémentées sur le même modèle que bitcoin, mais avec une blockchain et un réseau totalement indépendant.

Pour chaque altcoin cité dans ce chapitre, 50 ou plus seront passés sous silence, suscitant colère et fureur auprès de leurs créateurs et fans. Le but de ce chapitre n'est pas d'évaluer ou de classer les altcoins, ou même de ne mentionner que les plus significatifs sur la base de quelque critère subjectif. Au lieu de cela, nous allons mettre en exergue certains exemples qui montrent l'ampleur et la variété de l'écosystème, en relevant la première occurrence de chaque innovation ou différenciation significative. Certains des exemples les plus intéressants d'altcoins sont en fait des échecs complets d'un point de vue monétaire. Ceci les rend peut-être encore plus intéressants à étudier, et souligne le fait que ce chapitre ne doit pas être utilisé comme un guide pour l'investissement.

Avec de nouvelles altcoins apparaissant tous les jours, il serait impossible de ne pas rater certaines d'entre elles, peut-être celle qui changera l'histoire. Le rythme de l'innovation est ce qui fait que cet environnement est si excitant ; il garantit que ce chapitre sera incomplet et périmé dès sa publication.

Une Classification des Monnaies et Chaînes alternatives

Bitcoin est un projet open source, et son code a été utilisé comme base pour de nombreux autres projets logiciels. La forme la plus commune de logiciel dérivé du code source de bitcoin est la création de monnaies décentralisées alternatives, ou *altcoins*, qui utilisent les même briques de base pour implémenter des monnaies numériques.

Il y a plusieurs couches de protocole implémentées au-dessus de la blockchain bitcoin. Ces *metacoins*, *metachaînes*, ou *applications blockchain* utilisent la blockchain comme une platforme applicative ou étendent le protocole bitcoin en lui ajoutant des couches supplémentaires. Parmi les exemples, on trouve les Colored Coins, Mastercoin, NXT et Counterparty.

Dans ce paragraphe nous allons examiner quelques altcoins notables, comme Litecoin, Dogecoin,

Freicoin, Primecoin, Peercoin, Darkcoin, et Zerocoin. Ces altcoins sont notables pour des raisons historiques ou parce qu'elles constituent un bon exemple d'un type particulier d'innovation, pas parce qu'elles ont la plus grande valeur ou qu'elles sont les "meilleures" altcoins.

Outre les altcoins, il existe également un certain nombre d'implémentations alternatives de blockchain qui ne sont pas réellement des "coins", et que j'appelle des *altchains*. Ces altchains implémentent un algorithme de consensus ainsi qu'un livre de compte distribué comme plateforme pour des contrats, systèmes d'enregistrement de noms ou d'autres applications. Les altchains se basent sur les mêmes briques élémentaires, et utilisent parfois aussi une monnaie ou autre token comme mécanisme de paiement, mais leur objectif principal n'est pas d'être une monnaie. Nous étudierons Namecoin et Ethereum comme exemple de ces altchains.

Enfin, un certain nombre de concurrents de bitcoin proposent une monnaie numérique ou un réseau de paiement, sans utiliser un livre de compte partagé ou un mécanisme de consensus basé sur une preuve de travail ; c'est le cas de Ripple et consorts. Ces technologies, n'étant pas basées sur la blockchain, sortent du cadre de ce livre et ne seront donc pas abordées dans ce chapitre.

Plateformes Meta Coin

Les meta coins et meta chaînes sont des couches logicielles implémentées au dessus de bitcoin, que ce soit pour implémenter une monnaie-dans-la-monnaie ou une surcouche à la plateforme/protocole à l'intérieur du système bitcoin. Ces couches fonctionnelles étendent le coeur du protocole bitcoin et ajoutent des fonctionnalités en codant des données supplémentaires à l'intérieur des transactions et adresses bitcoin. Les premières implémentations de meta coins utilisaient différents hacks pour ajouter des métadonnées dans la blockchain, comme le fait d'utiliser des adresses pour encoder des données, ou encore des champs inutilisés des transactions (par exemple le champ sequence). Depuis l'introduction de l'opcode OP_RETURN dans les scripts de transaction, les meta coins ont eu la possibilité d'enregistrer des métadonnées directement dans la blockchain, et la plupart sont en train de migrer en ce sens.

Colored Coins

Les *Colored coins* sont un meta protocole qui encapsule des informations sur de petites quantités de bitcoin. une coin "colorée" est une somme de bitcoin réassignée pour représenter un actif différent. Imaginez, par exemple, prendre un billet de 1 dollar, et mettre un coup de tampon dessus indiquant, "Ce billet constitue 1 action de la société Acme Inc." Maintenant, le billet peut servir à deux choses : c'est un billet de banque et aussi un certificat d'action. Parce que sa valeur en tant qu'action est supérieure, vous ne voudriez pas vous en servir pour acheter des bonbons, donc il ne sera en pratique plus utilisé comme unité de monnaie. Les colored coins fonctionnent de manière similaire, en convertissant une certaine valeur, faible, de bitcoin en un certificat représentant un autre actif. Le terme "coloré" fait référence à l'idée de donner une signification spéciale à travers un attribut particulier comme une couleur — c'est une métaphore : il n'y a pas de couleurs dans les colored coins.

Les colored coins sont gérés par des porte-monnaie spécialisés qui enregistrent et interprètent les métadonnées associées aux bitcoin colorés. En utilisant un tel wallet, l'utilisateur va colorer une

somme de bitcoins en leur ajoutant un label doté d'une signification spéciale. Par exemple, un label pourrait représenter des certificats d'action, des obligations, des propriétés immobilières, des matières premières, ou des articles de collection. Les utilisateurs de colored coins sont totalement libres de l'attribution et l'interprétation de la "couleur" associée à des bitcoins. Pour colorer des pièces, l'utilisateur définit une métadonnée associée, comme par exemple un mode d'émission, le fait de pouvoir ou non les subdiviser, un symbole et une description, ainsi que d'autres informations. Une fois colorées, les pièces peuvent être achetées et vendues, subdivisées, agrégées, et peuvent faire l'objet de dividendes. Les colored coins peuvent aussi être "décolorées" en leur retirant leur attribut spécial, et peuvent être rachetées pour leur valeur faciale en bitcoin.

Pour montrer un exemple d'utilisation des colored coins, nous avons créé un ensemble de 20 colored coins portant le symbole "MasterBTC", qui représentent un bon pour un exemplaire gratuit de ce livre, comme le montre la figure Le profil de métadonnées des colored coins, configuré comme bon pour un exemplaire gratuit du livre. Chaque unité de MasterBTC représentée par ces colored coins peut maintenant être vendue ou donnée à n'importe quel utilisateur doté d'un wallet compatible avec les colored coins ; ce dernier pourra à son tour les transférer à d'autres, ou bien les échanger auprès de l'émetteur contre un exemplaire du livre. Cet exemple peut être vu ici.

Example 1. Le profil de métadonnées des colored coins, configuré comme bon pour un exemplaire gratuit du livre

```
{
  "source_addresses": [
    "3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD"
  ],
  "contract_url":
"https://www.coinprism.info/asset/3NpZmvSPLmN2cVFw1pY7gxEAVPCVfnWfVD",
  "name_short": "MasterBTC",
  "name": "Free copy of \"Mastering Bitcoin\"",
  "issuer": "Andreas M. Antonopoulos",
  "description": "This token is redeemable for a free copy of the book \"Mastering
Bitcoin\"",
  "description_mime": "text/x-markdown; charset=UTF-8",
  "type": "Other",
  "divisibility": 0,
  "link_to_website": false,
  "icon_url": null,
  "image_url": null,
  "version": "1.0"
}
```

Mastercoin

Mastercoin est une couche de protocole se superposant à bitcoin, et qui constitue une plateforme pour

une variété d'applications étendant le système bitcoin. Mastercoin utilise la devise MST pour l'établissement de transactions, mais elles n'a pas pour but principal d'être une monnaie. Il s'agit plutôt d'une plateforme pour construire d'autres choses, comme une monnaie personnalisée, des tokens de "propriété intelligente", des places de marché d'actifs décentralisés, et des contrats. Dites vous que Mastercoin est un protocole applicatif superposé à la couche de transport de transactions financières qu'est bitcoin, tout comme HTTP est situé au-dessus de TCP.

Mastercoin fonctionne principalement par le biais de transactions envoyées et reçues vers et depuis des adresses bitcoin spéciales appelées adresses "exodus" (1EXoDusjGwvnjZUyKkxZ4UHEf77z6A5S4P), exactement comme HTTP utilise un port TCP spécifique (port 80) pour se différencier du reste du traffic TCP. Le protocole Mastercoin est en train d'évoluer progressivement, de l'utilisation de ces adresses exodus avec multi-signatures à celle de l'opérateur OP_RETURN pour encoder les métadonnées de transaction.

Counterparty

Counterparty est une autre couche de protocole implémentée au-dessus de bitcoin. Counterparty permet le développement de monnaies personnalisées, de tokens échangeables, d'instruments financiers, de places de marché d'actifs décentralisés, et davantage. Counterparty est implémenté principalement via l'opérateur op_RETURN dans le langage de script bitcoin, pour encoder les métadonnées qui étendent la signification des transactions bitcoin. Counterparty utilise la devise XCP comme token pour effectuer des transactions.

Les Alt Coins

La grande majorité des altcoins sont dérivées du code source de bitcoin, c'est ce qu'on appelle des "forks". Certaines sont implémentées "depuis zéro", en se basant sur le modèle de la blockchain, mais sans utiliser le code source de bitcoin. Les altcoins et altchains (cf. chapitre suivant) sont toutes les deux des implémentations distinctes de la technologie blockchain, et les deux utilisent leur propre blockchain. La dénomination différente sert à indiquer que les altcoins servent principalement de monnaie, alors que les altchains sont utilisées à d'autres fins.

Pour être précis, le premier fork alternatif majeur du code de bitcoin n'était pas un altcoin, mais l'altchain *Namecoin*, que nous étudierons dans un prochain paragraphe.

Sur la base de la date de son annonce, le premier altcoin résultant d'un fork de bitcoin est apparu en Août 2011 ; il s'appelait *IXCoin*. IXCoin modifiait quelques un des paramètres originaux de bitcoin, en particulier il accélérait la création monétaire en fixant la récompense à 96 coins par bloc.

En Septembre 2011, _*Tenebrix* fut lancé. Tenebrix était la première cryptomonnaie à implémenter un algorithme de preuve de travail alternatif, nommé *scrypt*, conçu à l'origine pour l'extension de mot de passe (résistance à la force brute). L'objectif affiché de Tenebrix était de faire une monnaie résistante au minage sur GPU et ASIC, en utilisant un algorithme gourmand en mémoire vive. Tenebrix échoua comme monnaie, mais servit de base pour Litecoin, qui a connu un un grand succès et à donner lieu à des centaines de clones.

Litecoin, en plus d'utiliser scrypt comme algorithme de preuve de travail, implémenta également un temps de génération de bloc plus rapide, en ciblant 2.5 minutes contre 10 minutes pour bitcoin. La monnaie résultante est vendue comme étant à bitcoin ce que l'argent est à l'or, et est destinée à être une monnaie alternative légère. Grâce au temps de confirmation plus rapide et à la limite totale de 84 millions, beaucoup d'adhérents de Litecoin pensent qu'il est plus adapté aux transactions dans les commerces que bitcoin.

Les altcoins ont continué à proliférer en 2011 et 2012, qu'ils soient basés sur bitcoin ou sur Litecoin. Au début de 2013, il y avait 20 altcoins s'affrontant pour se faire une place sur le marché. Avant la fin de 2013, ce nombre avait explosé pour atteindre 200, 2013 devenant rapidement "l'année des altcoins". La croissance des altcoins à continué en 2014, avec plus de 500 altcoins en existence à au moment d'écrire ce livre. Plus de la moitié des altcoins aujourd'hui sont des clones de Litecoin.

Créer un altcoin est facile, c'est pourquoi il en existe maintenant plus de 500. La plupart des altcoins diffèrent peu de bitcoin et n'offrent rien qui vaille la peine de les étudier. Beaucoup sont en fait des tentatives pour enrichir leurs créateurs. Parmi ces copies et les fraudes de type "pump-and-dump", il y a toutefois quelques exceptions notables et des innovations très importantes. Ces altcoins prennent des approches radicalement différentes, ou ajoutent une innovation significative au modèle de conception de bitcoin. Il y a trois domaines principaux dans lesquels ces altcoins se différencient de bitcoin :

- Une politique monétaire différente
- Un mécanisme de preuve de travail ou de consensus différent
- Des fonctionnalités spécifiques, telles qu'une anonymat fort

Pour plus d'information, référez-vous à ce frise chronologique des altcoins et altchains.

Évaluer un altcoin

Avec tant d'altcoins en circulation, comment décide-t-on lesquels sont dignes d'attention ? Certains altcoins visent à se propager largement et être utilisées comme monnaie. D'autres sont des laboratoires pour expérimenter différentes fonctionnalités et modèles monétaires. D'autres sont justes des arnaques pour enrichir rapidement leurs créateurs. Pour évaluer les altcoins, je regarde leur caractéristiques principales et leurs métriques marché.

Voici quelques questions à se poser pour évaluer à quel point un altcoin se différencie de bitcoin :

- L'altcoin introduit-il une innovation significative ?
- Y a-t-il une différence suffisamment intéressante pour attirer des utilisateurs de bitcoin ?
- L'altcoin s'adresse-t-il à un marché de niche ou une application intéressants ?
- L'altcoin peut-il attirer suffisamment de mineurs pour être protégé contre les attaques de consensus ?

Voici certaines des métriques financières et marché les plus importantes à considérer :

- Quelle est la capitalisation totale de l'altcoin ?

- Quelle est l'estimation du nombre d'utilisateurs/wallets de l'altcoin ?
- Combien de marchands acceptent l'altcoin ?
- Combien de transactions journalières (en volume) sont exécutées sur l'altcoin ?
- Quelle valeur est négociée quotidiennement ?

Dans ce chapitre, nous allons nous concentrer principalement sur les caractéristiques techniques et le potentiel d'innovation des altcoins représentées par la première série de questions.

Paramètres Monétaires Alternatifs : Litecoin, Dogecoin, Freicoin

Bitcoin est doté de paramètres monétaires lui conférant les caractéristiques particulières d'une monnaie déflationniste à émission plafonnée. Elle est limitée à 21 millions d'unités de monnaie majeures (ou 21 billiards d'unités mineures), son taux d'émission diminue de façon géométrique, et un nouveau bloc est créé toutes les 10 minutes, ce qui contrôle la vitesse de confirmation des transactions et la génération de monnaie. Beaucoup d'altcoins ont modifié ces paramètres principaux pour aboutir à une politique monétaire différente. Les exemples suivants sont certains des plus notables parmi les centaines d'altcoins qui existent.

Litecoin

Apparu en 2011, Litecoin est un des premiers altcoins, et la deuxième monnaie numérique après bitcoin. Ses innovations principales étaient l'utilisation de *scrypt* comme algorithme de preuve de travail (hérité de Tenebrix), et ces paramètres monétaires plus légers et rapides.

- Temps de génération de bloc : 2.5 minutes
- Masse monétaire totale : 84 millions de pièces d'ici 2140
- Algorithme de consensus : Preuve de travail Scryppt
- Capitalisation totale : 260 millions de dollars à la mi-2014

Dogecoin

Dogecoin est sorti en Décembre 2013, et est basé sur un fork de Litecoin. Dogecoin est remarquable en ce qu'il possède une politique d'émission rapide, et une masse monétaire très élevée, afin d'encourager le fait de dépenser et donner des pourboires. Dogecoin est également notable parce qu'il a débuté comme une plaisanterie mais est devenu assez populaire, avec une communauté importante et active, avant de décliner rapidement en 2014.

- Temps de génération de bloc : 60 secondes
- Masse monétaire totale : 100 000 000 000 (100 milliards) de Doge d'ici 2015
- Algorithme de consensus : Preuve de travail Scryppt
- Capitalisation totale : 12 millions de dollars à la mi-2014.

Freicoin

Freicoin est apparu en Juillet 2012. C'est une *monnaie fondante*, ce qui signifie que son taux d'intérêt est négatif pour la monnaie stockée. Un TAEG de 4.5% est associé à la valeur stockée dans Freicoin, afin d'encourager la consommation et décourager l'accumulation d'argent. Freicoin est notable en ce qu'il implémente une politique monétaire à l'exact opposé de la déflation propre à Bitcoin. Freicoin n'a pas eu de succès en tant que monnaie, mais est un exemple intéressant de la variété de politiques monétaires qui peuvent être mises en oeuvre par les altcoins.

- Temps de génération de bloc : 10 minutes
- Masse monétaire totale : 100 millions de pièces en 2140
- Algorithme de consensus : Preuve de travail SHA256
- Capitalisation totale : 130 000 dollars à la mi-2014

Innovation dans l'algorithme de consensus : Peercoin, Myriad, Blackcoin, Vericoin, NXYT

Le mécanisme de consensus de Bitcoin est basé sur une preuve de travail qui utilise l'algorithme SHA256. Les premiers altcoins ont introduit scrypt comme algorithme de preuve de travail alternatif, permettant au minage d'être davantage tourné vers les CPU, et moins susceptible d'être centralisés avec des ASICs. Depuis lors, les innovations dans le mécanisme de consensus ont continué à un rythme effréné. Plusieurs altcoins ont adopté un ensemble d'algoithmes tels que scrypt, , scrypt-N, Skein, Groestl, SHA3, X11, Blake, et d'autres. Certains altcoins ont combiné plusieurs algorithmes de preuve de travail. En 2013, nous avons vu l'invention d'une alternative à la preuve de travail, appelée *preuve de participation*, qui est à la base de beaucoup d'altcoins modernes.

La preuve de participation est un système par lequel les détenteurs d'une monnaie peuvent mettre en jeu de la monnaie comme collatéral associé à un intérêt. Un peu à la façon d'un certificat de dépôt, les participants peuvent mettre en réserve une portion de leurs fonds, et voir leur investissement rémunéré sous la forme d'émission de monnaie (par le biais de taux d'intérêts), et de commissions de transactions.

Peercoin

Peercoin est apparu en Août 2012, et est la première altcoin à utiliser une hybridation de preuve de travail et de preuve de participation pour émettre de la monnaie.

- Temps de génération de bloc : 10 minutes
- Masse monétaire totale : Illimitée
- Algorithme de consensus : (Hybride) preuve de participation avec une preuve de travail initiale.
- Capitalisation totale : 14 millions de dollars à la mi-2014

Myriad

Myriad est sorti en Février 2014 et est notable pour avoir utilisé cinq algorithmes de preuve de travail différents (SHA256d, Scrypt, Qubit, Skein, ou Myriad-Groestl) simultanément, avec une difficulté variant pour chaque algorithme en fonction de la participation au minage. Le but est d'immuniser Myriad contre le minage par ASIC la centralisation, et de le rendre beaucoup plus résistant face à une attaque de consensus, parce que de multiples algorithmes devraient être attaqués simultanément.

- Temps de génération de bloc : 30 secondes en moyenne (cible de 2.5 minutes par algorithme de minage)
- Masse monétaire totale : 2 milliard d'ici 2024
- Algorithme de consensus : Preuve de travail multi-algorithmique
- Capitalisation totale : 120 000 dollars à la mi-2014

Blackcoin

Blackcoin fut présenté en Février 2014 et utilise un algorithme de consensus par preuve de participation. Il est également notable pour introduire des "multipools", un type de pool de minage pouvant choisir entre différents altcoins automatiquement en fonction de la profitabilité.

- Temps de génération de bloc : 1 minute
- Masse monétaire totale : Illimitée
- Algorithme de consensus : Preuve de participation
- Capitalisation totale : 3.7 million de dollars à la mi-2014

VeriCoin

VeriCoin a été lancé en Mai 2014. Il utilise un algorithme de consensus par preuve de participation avec un taux d'intérêt variable qui s'ajuste dynamiquement en fonction du poids relatif de l'offre et de la demande sur le marché. Il est également le premier altcoin à proposer un échange automatisé vers bitcoin lors d'un paiement en bitcoin via le wallet.

- Temps de génération de bloc : 1 minute
- Masse monétaire totale : Illimitée
- Algorithme de consensus : Preuve de participation
- Capitalisation totale : 1.1 million de dollars à la mi 2014

NXT

NXT (prononcé "Next") est un altcoin en preuve de participation "pure", en ce qu'il n'utilise pas de minage par preuve de travail. NXT est une implémentation depuis zéro d'une cryptomonnaie, pas un fork de bitcoin ou d'un autre altcoin. NXT implémente de nombreuses fonctionnalités avancées, dont un registre de nom (similaire à Namecoin), une place de marché d'actifs décentralisée (similaire au

Colored Coins), une messagerie intégrée décentralisée et sécurisée (similaire à Bitmessage), et une délégation de participation (pour déléguer une preuve de participation à un tiers). Pour ses partisans, NXT est la "génération suivante" de cryptomonnaie, ou cryptomonnaie 2.0.

- Temps de génération de bloc : 1 minute
- Masse monétaire totale : Illimitée
- Algorithme de consensus : Preuve de participation
- Capitalisation totale : 30 millions de dollars à la mi-2014

Innovation du minage à double-emploi : Primecoin, Curecoin, Gridcoin

L'algorithme de preuve de travail de Bitcoin n'a qu'un seul objectif : sécuriser le réseau bitcoin. Par comparaison à la sécurité des systèmes de paiement traditionnels, le coût du minage n'est pas très élevé. Cependant, il a été critiqué par beaucoup comme étant un "gâchis". La génération d'altcoin suivante tente de résoudre ce problème. Les algorithmes de preuve de travail à double-emploi résolvent un problème "utile" spécifique, tout en produisant une preuve de travail pour sécuriser le réseau. Le risque d'ajouter un rôle externe à la sécurité de la monnaie est que cela ajoute également une influence sur la courbe offre/demande.

Primecoin

Primecoin a été annoncée en Juillet 2013. Son algorithme de preuve de travail recherche des nombres premiers, en calculant chaînes de Cunningham et de nombres premiers jumeaux. Les nombres premiers sont utiles dans un certain nombre de disciplines scientifiques. La blockchain Primecoin contient les nombres premiers découverts, constituant ainsi une archive publique de découverte scientifique en parallèle du livre public des transactions.

- Temps de génération de bloc : 1 minute
- Masse monétaire totale : Illimitée
- Algorithme de consensus : Preuve de travail par calcul de chaîne de nombres premiers
- Capitalisation totale : 1.3 millions de dollars à la mi-2014

Curecoin

Curecoin a été annoncé en Mai 2013. Il combine un algorithme de preuve de travail par SHA256 avec une recherche de repliement de protéine à travers le projet Folding@Home. Le repliement de protéine est une simulation des interactions biochimiques des protéines, gourmande en puissance de calcul, utilisée pour mettre au point de nouveaux médicaments.

- Temps de génération de bloc : 10 minutes
- Masse monétaire totale : Illimitée
- Algorithme de consensus : Preuve de travail avec recherche de repliement de protéine
- Capitalisation totale : 58 000 dollars à la mi-2014

Gridcoin

Gridcoin est apparu en Octobre 2013. En complément d'une preuve de travail basée sur scrypt, les mineurs participent aussi à la grille de calcul distribuée BOINC. BOINC — Berkeley Open Infrastructure for Network Computing — est un protocole ouvert pour les grilles de calcul de recherche scientifique, qui permet aux participants de mettre à disposition la puissance de calcul non utilisée, à destination d'un large éventail de recherche académique en informatique. Gridcoin utilise BOINC comme une plateforme de calcul générique, plutôt que de résoudre des problèmes scientifiques spécifiques comme les nombres premiers ou le repliement de protéines.

- Temps de génération de bloc : 150 secondes
- Masse monétaire totale : Illimitée
- Algorithme de consensys : Preuve de travail avec participation à la grille de calcul BOINC
- Capitalisation totale : 122 000 dollars à la mi-2014

Altcoins orientées anonymat : CryptoNote, Bytecoin, Monero, Zerocash/Zerocoin, Darkcoin

Bitcoin est souvent décrit à tort comme une monnaie "anonyme". En fait, il est relativement facile de relier des identités à des adresses bitcoin et, en utilisant l'analyse big-data, de relier ces adresses entre elles pour réaliser un schéma complet du profil de dépenses de quelqu'un. Plusieurs altcoins vise à résoudre ce problème directement en se concentrant sur un anonymat fort. La première tentative de ce genre est très certainement *Zerocoin*, un protocole meta-coin au dessus de bitcoin permettant de préserver l'anonymat, introduit par un papier présenté au Symposium de l'IEEE sur la Sécurité et la Vie privée en 2013. Zerocoin sera implémenté comme un altcoin complètement séparé appelé Zerocash, en développement au moment d'écrire ce livre. Une approche alternative à l'anonymat a été lancée avec *CryptoNote* dans un papier publié en Octobre 2013. CryptoNote est une brique technologique qui est implémentée par un certain nombre d'altcoins forkés, étudiés ci-après. En plus de Zerocash et CryptoNotes, il y a plusieurs monnaies anonymes indépendantes, comme Darkcoin, qui utilise des adresses camouflées et un mélange des transactions pour garantir l'anonymat.

Zerocoin/Zerocash

Zerocoin constitue une approche théorique de l'anonymat en matière de monnaies numériques, introduit en 2013 par des chercheurs à l'Université Johns Hopkins. Zerocash en est une implémentation de type altcoin, qui est en développement et n'est pas encore sortie.

CryptoNote

CryptoNote est un altcoin, implémentation de référence apportant les bases d'un cash numérique anonyme. Il a été présenté en Octobre 2013. Il est conçu pour être forké vers différentes implémentations, et a un mécanisme de remise à zéro périodique intégré le rendant lui-même inutilisable comme monnaie. Plusieurs altcoins sont nées de CryptoNote, comme Bytecoin (BCN), Aeon (AEON), Boolberry (BBR), duckNote (DUCK), Fantomcoin (FCN), Monero (XMR), MonetaVerde (MCN), Quazarcoin (QCN). CryptoNote est aussi notable pour être une implémentation depuis zéro d'une

monnaie numérique, et pas un fork de bitcoin.

Bytecoin

Bytecoin a été la première implémentation partant de CryptoNote, offrant une monnaie anonyme viable basée sur la technologie CryptoNote. Bytecoin a été lancée en Juillet 2012. Notez qu'il existait auparavant un altcoin nommé Bytecoin avec le symbole monétaire BTE, alors que le Bytecoin dérivé de CryptoNote possède le symbole BCN. Bytecoin utilise l'algorithme de preuve de travail Cryptonight, qui requiert un accès à un minimum 2MB de RAM par instance, ce qui exclut le mining GPU ou ASIC. Via CryptoNote, Bytecoin hérite des signatures circulaires, des transactions intraçables et d'un anonymat résistant à l'analyse de blockchain.

- Temps de génération de blloc : 2 minutes
- Masse monétaire totale : 184 milliards de BCN
- Algorithme de consensus : Preuve de travail Cryptonight
- Capitalisation totale : 3 millions de dollards à la mi-2014

Monero

Monero est une autre implémentation de CryptoNote. Sa courbe d'émission est légèrement plus plate que celle de Bytecoin, 80% de la monnaie étant générée dans les quatre premières années. Il offre les mêmes propriétés d'anonymat héritées de CryptoNote.

- Temps de génération de bloc : 1 minute
- Masse monétaire totale : 18.4 millions de XMR
- Algorithme de consensus : Preuve de travail Cryptonight
- Capitalisation totale : 5 millions de dollars à la mi-2014

Darkcoin

Darkcoin a été lancé en Janvier 2014. Darkcoin implémente une monnaie anonyme via un protocole de re-mélange pour toutes les transactions, apppelé DarkSend. Darkcoin est également notable pour utiliser 11 passes de fonctions de hash différentes (blake, bmw, groestl, jh, keccak, skein, luffa, cubehash, shavite, simd, echo) pour l'algorithme de preuve de travail.

- Temps de génération de bloc : 2.5 minutes
- Masse monétaire totale : Maximum 22 million de DRK
- Algorithme de consensus : Preuve de travail multi-algorithme à plusieurs passes
- Capitalisation totale : 19 millions de dollars à la mi-2014

Altchains à vocation non monétaires

Les altchains sont des implémentations alternatives du design pattern de la blockchain, qui ne sont pas

utilisées principalement comme monnaie. Beaucoup incluent une monnaie, mais cette dernière est utilisée comme un jeton pour allouer quelque chose d'autre, comme une ressource ou un contrat. La monnaie, en d'autres termes, n'est pas l'objet principal de la plateforme ; elle en est une propriété secondaire.

Namecoin

Namecoin a été le premier fork du noeud bitcoin. Namecoin est une plateforme décentralisée clé-valeur d'enregistrement et de transfert qui utilise la blockchain. Elle propose un registre global de nom de domaine similaire au systèmes des noms de domaine sur Internet. Namecoin est actuellement utilisé comme un service de nom de domaine (DNS) alternatif pour le nom de domaine racine .bit. Namecoin peut aussi être utilisé pour enregistrer des noms et des paires clé-valeur dans d'autres espaces de noms, comme des adresses emails, des clés de cryptage, des certificats SSL, des signatures de fichiers, des systèmes de vote, des certificats d'action, et une multitude d'autres applications.

Le système Namecoin inclue la monnaie Namecoin (symbole NMC), utilisée pour payer les commissions de transactions associées à l'enregistrement et au transfert des noms. Au prix actuel, la commission pour enregistrer un nom est de 0.01 NMC, soit approximativement 1 cent US. Comme dans bitcoin, les commissions sont prélevées par les mineurs namecoin.

Les paramètres de base de Namecoin sont les mêmes que ceux de bitcoin.

- Temps de génération de bloc : 10 minutes
- Masse monétaire totale : 21 millions de NMC d'ici 2140
- Algorithme de consensus : Preuve de travail SHA256
- Capitalisation totale : 10 millions de dollars à la mi-2014

Les espaces de nom de Namecoin ne sont pas limités, et n'importe qui peut utiliser le namespace qu'il souhaite, de la façon où il l'entend. Toutefois, certains namespaces ont une spécification bien définie, de telle sorte que quand ils les lisent depuis la blockchain, les logiciels savent comment les lire et les interpréter S'ils sont malformés, alors le logiciel que vous utilisez vous renverra une erreur. Les namespaces les plus populaires sont :

- d/ est l'espace de nom pour les domaines .bit + id/ est le namespace pour stocker des identifiants de personne, tels que des adresses email, des clés PGP, etc.
- u/ est une spécification supplémentaire plus structurée pour stocker des identités (basée sur openspecs)

Le client Namecoin est très semblable à Bitcoin Core, car il dérive du même code source. Après installation, le client va télécharger la copie complète de la blockchain Namecoin et sera alors prêt à effectuer des requêtes ou enregistrer des noms. Il y a trois commandes principales :

name_new

Teste l'existence ou pré-enregistre un nom

name_firstupdate

Enregistre un nom et publie l'enregistrement

name_update

Modifie ou met à jour un enregistrement de nom.

Par exemple, pour enregistrer un domaine mastering-bitcoin.bit, nous utilisons la commande name_new comme suit :

```
$ namecoind name_new d/mastering-bitcoin
```

```
[
    "21cbab5b1241c6d1a6ad70a2416b3124eb883ac38e423e5ff591d1968eb6664a",
    "a05555e0fc56c023"
]
```

La commande name_new permet de réserver le nom en créant un hash du nom avec un clé aléatoire. Les deux chaînes retournées par name_new sont le hash et la clé aléatoire (a05555e0fc56c023 dans l'exemple qui précède), et sont utilisées pour publier l'enregistrement. Une fois que la demande de réservation a été enregistrée dans la blockchain Namecoin elle peut être convertie en enregistrement public avec la commande name_firstupdate, en fournissant la clé aléatoire :

```
$ namecoind name_firstupdate d/mastering-bitcoin a05555e0fc56c023 "{"map": {"www":
{"ip":"1.2.3.4"}}}}"
b7a2e59c0a26e5e2664948946ebeca1260985c2f616ba579e6bc7f35ec234b01
```

Cet exemple va associer le nom de domaine www.mastering-bitcoin.bit à l'adresse IP 1.2.3.4. Le hash retourné est l'ID de transaction qui peut être utilisé pour suivre l'enregistrement. Vous pouvez voir quels noms vous sont attribués en exécutant la commande name_list :

```
$ namecoind name_list
```

```
[
    {
        "name" : "d/mastering-bitcoin",
        "value" : "{map: {www: {ip:1.2.3.4}}}}",
        "address" : "NCccBXrRUahAGrisBA1BLPWQfSrups8Geh",
        "expires_in" : 35929
    }
]
```

Les enregistrements Namecoin doivent être mis à jour tous les 36 000 blocs (approximativement 200 à 250 jours). La commande name_update n'a pas de commission, d'où il résulte que renouveler un domaine Namecoin est gratuit. Des tierces parties peuvent prendre en charge l'enregistrement, le renouvellement automatique, la mise à jour par interface web, en échange d'une faible commission. Avec un fournisseur tiers, vous éviter de devoir faire tourner un client Namecoin, mais vous perdez l'indépendance qu'offre ce système d'enregistrement de nom décentralisé .

Ethereum

Ethereum est un processeur de contrat Turing-complet, et une plateforme d'exécution basée sur un registre de type blockchain. Ce n'est pas un clone de Bitcoin, mais une conception et une implémentation complètement indépendante. Ehtereum possède une monnaie intégrée, appelée *ether*, qu'il est nécessaire de payer exécuter un contrat. La blockchain Ethereum enregistre des *contrats*, qui sont exprimés dans un langage Turing-complet bas niveau, ressemblant à du byte code. Globalement, un contrat est un programme qui tourne sur tous les noeuds du système Ethereum. Les contrats Ethereum peuvent stocker des données, envoyer et recevoir des paiements en ether, stocker de l'ether, et exécuter un éventail infini d'opérations (d'où le caractère Turing-complet), tels des agents logiciels autonomes et décentralisés.

Ethereum peut implémenter des systèmes relativement complexes, qui sont ailleurs implémentés par des altchains dédiées. Par exemple, ce qui suit est un contrat d'enregistrement de nom similaire à Namecoin, écrit en Ethereum (ou plus précisément, écrit dans un langage de haut-niveau compilable en code Ethereum) :

```
if !contract.storage[msg.data[0]]: # Is the key not yet taken?
    # Then take it!
    contract.storage[msg.data[0]] = msg.data[1]
    return(1)
else:

    return(0) // Otherwise do nothing
```

Futur des Monnaies

Le futur des monnaies cryptographiques, de façon générale, est encore plus prometteur que celui de bitcoin. Bitcoin a introduit une nouvelle forme d'organisation décentralisée et de consensus qui a engendré des centaines d'innovations incroyables. Ces inventions vont probablement affecter des pans entiers de l'économie, depuis la science des systèmes distribués jusqu'à la finance, les monnaies, les banques centrales, et la gestion des entreprises. Beaucoup d'activités humaines, qui jusqu'alors nécessitaient des organisations ou institutions centralisées servant de point de contrôle ou de confiance, peuvent maintenant être décentralisées. L'invention de la blockchain et du système de consensus va réduire significativement le coût de l'organisation et de la coordination des systèmes fonctionnant à grande échelle, tout en supprimant les possibilités de concentration de pouvoir, corruption, et capture règlementaire.

Sécurité du Bitcoin

La sécurisation du bitcoin est un enjeu important car les bitcoins ne sont pas une représentation abstraite d'une valeur, contrairement au solde d'un compte en banque. Bitcoin s'apparente plus à du cash numérique ou de l'or. Vous avez probablement déjà entendu l'expression "La possession représente les neuf dixièmes de la loi". Et bien avec le bitcoin, la possession c'est la loi. La possession de clés permettant de débloquer des bitcoins est équivalente à la possession de billets ou de lingots d'un métal précieux. Vous pouvez les perdre, les déplacer, vous les faire voler, ou donner accidentellement le mauvais montant à quelqu'un. Dans tous ces cas, les utilisateurs n'ont aucun recours, exactement comme si il jetaient des billets de banque dans une rue bondée.

Cependant, bitcoin fournit des options que le cash l'or ou les comptes en banques ne fournissent pas. Un portefeuille bitcoin contenant vos clés, peut être sauvegardé sous forme de copies multiples, il peut même être imprimé sur du papier en tant que sauvegarde physique. Il est impossible de "sauvegarder" du cash de l'or ou un compte en banque. Bitcoin présente assez de différences par rapports aux anciens modèle pour que l'on puisse envisager sa sécurité d'une manière totalement nouvelle.

Principes de Sécurité

Le principe de base du bitcoin est la décentralisation et ce principe possède des implications importante pour sa sécurité. Un modèle centralisé tel qu'une banque traditionnelle ou un réseau de paiement, dépends du contrôle de l'accès et la capacité à garder les voleurs hors du système. en comparaison, un système décentralisé comme bitcoin déplace la responsabilité et le contrôle vers les utilisateurs. Parce que la sécurité du réseau est basée sur la preuve de travail et non le contrôle d'accès, le réseau peut être ouvert et aucun cryptage n'est requis pour le traffic bitcoin.

Sur les réseaux de paiment traditionnels, tels que les systèmes de cartes de crédit, le paiment n'est pas limité car il contient l'identifiant privé de l'utilisateur (le numéro de carte de crédit). Après la dépense initiale, quiconque ayant accès à cet identifiant peut "retirer" des fonds à l'infiini. Ducoup, le réseau de paiement a besoin d'être crypté tout le long et doit s'assurer qu'aucune oreille indiscrète ou intermédiaire ne peut compromettre le trafic de paiement, lors du transit ou lorsqu'il est enregistré. Si un acteur malveillant arrive à accéder au système, il peut compromettre les transactions *et* les informations de paiement pouvant être utilisées pour créer de nouvelles transactions. Pire, quand les donnée d'un client sont compromises, les clients sont exposés à une usurpation d'identité et doivent agir afin d'empêcher l'usage frauduleux des comptes compromis.

Bitcoin est radicalement différent. Une transaction bitcoin n'autorise qu'un montant précis vers un destinataire spécifique, et ne peut être contrefaite ou modifiée. Elle ne révèle aucune information privée telle que l'identité des parties, et ne peut être utilisée pour l'autorisation de paiement additionnels. Par conséquent, un réseau de paiement bitcoin n'a pas besoin d'être crypté ou protégé des oreilles indiscrètes. En fait, vous pouvez diffuser des transaction bitcoin sur un réseau public tel qu'un réseau WiFi ou Bluetooth public, sans aucune perte de sécurité.

Le modèle de sécurité décentralisé de bitcoin donne beaucoup de pouvoir aux utilisateurs. Ce pouvoir

s’accompagne de la responsabilité de maintenir le secret des clés. Pour la plupart des utilisateurs ce n’est pas si facile à réaliser, en particulier sur les terminaux web grand public tels que les smartphones ou les ordinateurs portables. Bien que le modèle de sécurité décentralisé de bitcoin évite les problèmes évoqués pour les cartes de crédit, beaucoup d’utilisateurs sont incapable de sécuriser de façon adéquate leurs clés et se les font tout simplement voler, les unes après les autres.

Construire des systèmes bitcoin sécurisés

Le principe le plus important pour les développeur bitcoin est la décentralisation. La plupart des développeurs sont familiers des modèles de sécurité centralisés et peuvent être tentés de reproduire ces modèles à leurs applications bitcoin, ce qui peut avoir des conséquences désastreuses.

La sécurité Bitcoin repose sur le control décentralisé des clés et sur l’indépendance de la validation des transaction par les mineurs. Si vous souhaitez profiter de la sécurité Bitcoin, vous devez vous assurer que vous restez dans le cadre de son modèle de sécurité. En d’autres termes : ne privez pas les utilisateurs du contrôle de leur clés et ne traitez pas des transaction en dehors de la blockchain.

Par exemple, jusqu’a récemment, beaucoup d’échanges bitcoin concentraient tous les fonds des utilisateurs dans des "hot wallets" (des porte-monnaies connectés) avec les clés enregistrées sur un serveur unique. Une telle architecture dépossède les utilisateur de tout contrôle et centralise le contrôle des clés sur un système unique. Beaucoup de ces échanges ont fini par se faire hacker, avec des conséquences désastreuse pour leurs clients.

Une autre erreur très courante est de traiter les transaction en dehors de la blockchain dans une tentative de réduire les frais de transaction ou d’accélérer leur traitement. Un système "off blockchain" enregistrera les transaction au sein d’un registre interne centralisé et les synchronisera de façon périodique avec la blockchain. Cette pratique substitue encore la sécurité décentralisée de bitcoin avec une approche propriétaire et centralisée. Quand les transactions sont en dehors de la blockchain, des registres mal sécurisés et centralisés peuvent être falsifiés sans que personne s’en aperçoive.

A moins que vous ne soyez prêts à investir lourdement dans le design d’un architecture propre sécurisés contenant de multiples couches de contrôle et des audits réguliers (comme le font les banques traditionnellement) vous devriez faire très attention avant de sortir les fonds du contexte de sécurité décentralisé offert par bitcoin. Même si vous avez les moyens et la discipline nécessaires pour implémenter un modèle de sécurité robuste, une telle architecture reproduit le modèle fragile des réseaux financiers traditionnels qui sont empoisonnés par des problèmes récurrent d’usurpation d’identité, de corruption et de détournements. Pour profiter du modèle de sécurité décentralisé unique de Bitcoin, vous devez fuir la tentation d’architecture centralisées qui malgré le fait qu’elle vous soit plus familières pervertissent la sécurité de bitcoin au final.

La racine de confiance

L’architecture de sécurité traditionnelle est basée sur le concept de *racine de confiance*, qui est un coeur de confiance utilisé comme fondation de la sécurité pour tout système ou application. L’architecture de sécurité est développée autour de cette racine de confiance comme une série de cercles concentriques, comme les couches d’un oignon, étendant la confiance en s’éloignant de son centre. Chaque couche de

construit au dessus d'une couche interne offrant plus de confiance en utilisant les contrôles d'accès, les signature numériques, le cryptage et d'autres principes de sécurité. Comme les système logiciels deviennent plus complexes, ils deviennent de fait plus sujets aux bugs qui peuvent eux mêmes engendrer des failles de sécurité. Au final plus un système devient complexe, plus il est difficile de le sécuriser. Le concept de la racine de confiance permet de s'assurer que la confiance est placée au sein de la partie la moins complexe du système, et donc la moins vulnérable, tandis que les parties plus complexes prennent place au sein des couches supérieures qui l'entourent. L'architecture de sécurité est répétée à différente échelle, établissant en premier lieu une racine de confiance au niveau hardware pour ensuite l'élargir au niveau du système d'exploitation, s'établir ensuite dans les couches de services de haut niveau, pour finalement se retrouver dans les différents serveurs formant des cercles concentrique de confiance allant en décroissant.

L'architecture de sécurité de Bitcoin est différente. Avec Bitcoin, le système de consensus est à l'origine de la création d'un registre public qui est complètement décentralisé. Une blockchain correctement validée utilise le genesis block comme racine de confiance, et construit une chaîne de confiance jusqu'au bloc actuel. Les systèmes Bitcoin peuvent et se doivent d'utiliser la blockchain comme racine de confiance. Lors du design d'une application bitcoin complexe composée de services tournant sur des systèmes différents, vous devez examiner attentivement l'architecture de sécurité afin de bien vous assurer de l'endroit où vous placez la confiance. En fin de compte, la seule chose à laquelle vous devez explicitement accorder votre confiance est une blockchain entièrement validé. Si votre application place la confiance ailleurs que dans la blockchain, cela doit être fait avec précaution car cela introduit de la vulnérabilité. Une bonne méthode pour évaluer l'architecture de votre application est de considérer chacun de ses composants et d'évaluer un scénario hypothétique ou ce composant est complètement compromis et sous le contrôle d'un acteur malveillant. Prenez chaque composant de votre application, l'un après l'autre, et évaluez les impacts globaux sur la sécurité si ce composant se trouve compromis. Si votre application se trouve ne plus être sécurisée lorsqu'un de ses composants est compromis, cela démontre que vous avez malencontreusement placé de la confiance au sein de ce composant. Une application bitcoin sans vulnérabilités ne devrait être vulnérable que si le mécanisme de consensus bitcoin lui-même est compromis, cela signifierait que sa racine de confiance est basée sur la partie la plus forte de l'architecture de sécurité bitcoin.

Les nombreux exemples d'échanges bitcoin piratés servent à souligner ce point car leur architecture de sécurité et leur design ne supportent pas un examen des plus basiques. Ces implémentations centralisées ont placé la confiance explicitement dans plusieurs composants en dehors de la blockchain, tels que dans des hot wallets, des registres centralisés dans des base de données, des clés de cryptage vulnérable ou d'autres composants similaires.

Bonnes pratiques de sécurité

L'Homme utilise de contrôles de sécurité physiques depuis des milliers d'années. En comparaison, notre expérience dans la sécurité numérique est de moins de 50 ans. Les systèmes d'exploitation modernes ne sont pas très sécurisés et pas particulièrement adaptés au stockage d'argent numérique. Nos ordinateurs sont continuellement exposés à des menaces extérieures via des connexions permanentes à Internet. Ils font tourner des milliers de composants logiciels écrits par des centaines d'auteurs différents, qui ont souvent un accès illimité aux fichiers de l'utilisateur. Un seul petit bout de

code, parmi les milliers installés sur votre ordinateur,peut compromettre votre clavier et vos fichiers, et voler les bitcoins contenus dans vos porte-monnaies. Le niveau de compétence en maintenance informatique requis pour garder votre ordinateur sans virus n'est possédé que par une petite minorité d'utilisateurs.

Malgré les dizaines d'années de recherche en sécurité informatique, les biens numériques sont toujours vulnérables faces à un adversaire déterminé. Même les systèmes hautement protégés et restreints, tels que ceux des entreprises financières, des agences de sécurité nationales ou des organismes de défense, sont fréquemment déjoués. Bitcoin crées des biens numériques ayant une valeur intrinsèque qui peuvent être volés et détournés de leurs nouveaux propriétaires instantanément et de manière irrévocable. Cela encourage grandement les hackers. Jusqu'à présente, les hackers devaient convertir des informations d'identité ou des informations de comptes —tels que les cartes de crédits et les comptes bancaires— en valeur après les avoir compromis. En dépit de la difficulté de recel et de blanchiment des données financières, nous avons pu remarquer une escalade dans le nombre de vols.

Heureusement, bitcoin encourage aussi d'une certaine manière l'amélioration de la sécurité des ordinateurs. Alors que les risques de compromission précédents étaient vagues et indirects, bitcoin rend ces risques clairs et évidents. Garder des bitcoins sur un ordinateur à pour résultat de mettre dans l'esprit de l'utilisateur le besoin de sécurité accru. Avec la prolifération et l'adoption sans cesse grandissante de bitcoin et des autres crypto-monnaies, nous avons vu apparaitre des moyens de plus en plus sophistiqués de hacking et avec eux de nouvelles solutions de sécurité. En d'autres termes, les hackers disposent maintenant d'une nouvelle cible juteuse et les utilisateurs possèdent désormais une vraie raison de se défendre en assurant la sécurité de leurs systèmes.

Durant ces trois dernières années, conséquemment à l'adoption du bitcoin, nous avons vus une grande innovation dans le domaine de la sécurité informatique qui a pris la forme du cryptage matériel, du stockage des clés, des porte-monnaies physiques, de la technologie multi-signature et des séquestres numériques. Dans les sections suivantes nous examineront plusieurs bonnes pratiques pour assurer la sécurité des utilisateurs.

Stockage physique de bitcoins

Parce que la plupart des utilisateur sont beaucoup plus à l'aise avec la sécurisation physique, une méthode très efficace pour sécuriser les bitcoins consiste à les convertir dans une forme physique. Les clés bitcoin ne sont rien d'autres que des grands nombres. Cela veut dire qu'elles peuvent être stockées sous forme physique, par exemple imprimées sur du papier ou gravées dans une pièce de métal. Sécuriser les clés revient dans ce cas à simplement sécuriser la copie physique des clés bitcoin. Un ensemble de clés bitcoin imprimées sur du papier est appellé "paper wallet" ou porte-monnaie papier, et il existe beaucoup d'outils gratuits permettant de les créer. Je conserve personnellement la majorité (99% ou plus) de mes bitcoins sur des porte monnaie papier cryptés avec BIP0038, avec de multiples copies conservées dans des coffres. Le fait de garder ses bitcoins hors ligne est désigné par le terme "cold starage" (littéralement "stockage à froid") et est considéré comme la technique de sécurisation la plus efficace. Un système de cold storage consiste en la génération des clés sur un système déconnecté (qui n'est jamais connecté à internet) et stocké hors ligne soit sur du papier, soit sur un support

numérique te qu'une clé USB.

Les Hardware Wallets

Sur le long terme, la sécurité bitcoin prendra peu a peu la forme de porte-monnaies physiques inviolable. Contrairement à un smartphone ou un ordinateur de bureau, un porte-monnaie physique n'a qu'un seul rôle : sécuriser des bitcoins. Sans aucun logiciel capable de le compromettre et une interface limitée, les hardware wallets peuvent fournir un niveau de sécurité quasi infaillible pour les utilisateurs non experts. Je m'attends à voir les hardware wallets devenir la méthode prédominante de stockage de bitcoin. Trezor est un exemple de hardware wallet disponible sur le marché.

Equilibrer le risque

Bien que la plupart des utilisateurs ne craignent que le vol de bitcoin, il existe un risque encore plus grand. Les fichiers de données sont très souvent perdus. Si ils contiennent des bitcoins, cette perte est d'autant plus douloureuse. Dans un effort de sécurisation de leur porte-monnaie bitcoin, les utilisateurs doivent faire très attention à ne pas aller trop loin et finir par perdre leurs bitcoins. En Juillet 2011, un projet de sensibilisation et d'éducation au bitcoin à perdu près de 7000 bitcoins. Dans leur effort d'empêcher tout vol, les propriétaires ont implémenté une série complexe de sauvegardes cryptées. Au final il ont accidentellement perdu les clés de cryptage, rendant les sauvegarde inutilisables avec comme résultat le perte d'une petite fortune. Comme les fait de cacher de l'argent en l'enterrant dans le désert, si vous sécurisez trop vos bitcoins il est possible que vous ne les retrouviez plus vous-même.

Diversification du risque

Est-ce que vous transporteriez la totalité de votre argent en cash dans votre porte-feuille ? Beaucoup de gens considéreraient cela comme une folie, pourtant, les utilisateurs bitcoin gardent souvent tous leurs bitcoin dans un unique wallet. Il vaut mieux à la place, disperser le risque dans de multiples wallets bitcoin. Les utilisateur prudents ne garderont qu'une petite portion de leur bitcoins, peut-être moins de 5%, dans un porte-monnaie web ou mobile comme "argent de poche". Le reste doit être séparé entre plusieurs moyens de stockage différents tels que des portemonnaies sur ordinateur ou hors-ligne (stockage à froid).

Multi-sig et Gouvernance

Quand une entreprise ou un individu stocke de grands montants de bitcoin, ils doivent considérer l'option d'adresses bitcoin multi-signature. Les adresses multi-signature sécurisent les fonds et nécessitant plus d'une signature pour effectuer un paiement. Les clés doivent être stockées dans plusieurs endroits et sous le contrôle de personnes différentes. Dans une entreprise par exemple, les clés doivent être générées indépendamment et gardées par plusieurs dirigeants, pour s'assurer qu'une personne seule ne puisse compromettre les fonds. Les adresses multi-signature peuvent également permettre la redondance, quand une seule personne possède différentes clés qui sont stockées dans différents endroits.

Survie

Un autre aspect de la sécurité souvent négligé est la disponibilité des bitcoins, particulièrement dans un contexte d'incapacité ou de décès du possesseur des clés. Les utilisateurs bitcoin sont encouragés à utiliser des mots de passe complexes et à garder leurs clés secrètes, sans les faire connaitre à quiconque. Malheureusement, dans la pratique, cela rend quasiment impossible pour la famille d'un utilisateur de récupérer les fonds si le possesseur n'est plus là pour les déverrouiller. Dans la plupart des cas, les familles des possesseurs de bitcoin ne sont tout simplement pas au courant de l'existence de fonds en bitcoins.

Si vous possédez beaucoup de bitcoins, vous devriez considérer le partage de l'accès à ces bitcoin à un proche de confiance ou un avocat. Un dispositif plus complexe de survie peut être la mise en place d'un accès multi-signature et d'une planning de succession au travers d'un avocat spécialisé qui fera office d'"exécutant testamentaire numérique".

Conclusion

Bitcoin est un technologie nouvelle, sans précédent et complexe. Avec le temps nous développerons de meilleurs outils de sécurité et des pratiques plus accessibles aux non-experts. Pour l'instant les utilisateurs bitcoin peuvent utiliser les conseils prônés ici pour profiter d'une expérience bitcoin sûre et sans problèmes.

Appendix A: Commandes de l’explorateur Bitcoin (bx)

```
Usage: bx COMMANDE [--help]

Info: Les commandes bx sont:

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
```

Pour plus d'informations, rendez vous sur the la page d'accueil Bitcoin explorer et le manuel utilisateur de Bitcoin Explorer.

Exemples de commande bx utilisé

Regardons quelques exemples de commandes Bitcoin Explorer pour expérimenter avec les clés et adresses:

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 BIP0032. 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 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
xprv9s21ZrQH143K2BEhMYpNQoUvAgiEjArAVaZaCTgsaGe6LsAnwubeiTcDzd23mAoyizm9cApe51gNfLMkBqkYo
WWMCRwzfuJk8RwF1SVEpAQ
```

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
xprv9vkDLt81dTKjwHB8fsVB5QK8cGnzveChzSrtCfvu3aMWvQaThp59ueufuyQ8Qi3qpjk4aKsbmbfxwcgS8PYbg
oR2NWHeLyvg4DhoEE68A1n

$ bx hd-private --index 0 < account
xprv9xHfb6w1vX9xgZyPNXVgAhPxSsEkeRcPHEUV5iJcVEsuUEACvR3NRY3fpGhcnBiDbvG4LgndirDsia1e9F3DW
PkX7Tp1V1u97HKG1FJwUpU

$ bx hd-private --index 1 < account
xprv9xHfb6w1vX9xjc8XbN4GN86jzNAZ6xHEqYxzbLB4fzHFd6VqCLPGRZFsdjsuMVERadbgDbziCRJru9n6tzEWr
ASVpEdrZrFidt1RDfn4yA3
```

Next we use the hd-public command to generate the corresponding sequence of two public keys.

```
$ bx hd-public --index 0 < account
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz
5wzaSW4FiGrseNDR4LKqTy

$ bx hd-public --index 1 < account
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQD
hohEcDFTEYMvYzwRD7Juf8
```

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
xpub6BH1zcTuktiFu43rUZ2gXqLgzu5F3tLEeTQ5t6iE3aQtM2VMTxMcyLN9fYHiGhGpQe9QQYmqL2eYPFJ3vezHz
5wzaSW4FiGrseNDR4LKqTy

$ bx hd-private --index 1 < account | bx hd-to-public
xpub6BH1zcTuktiFx6CzhPbGjG3UYQ13WR16CmtbPiagEKpEVtpyjshWyMaMV1cn7nUPUkgQHPVXJVqsrA8xWbGQD
hohEcDFTEYMvYzwRD7Juf8
```

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.

Propositions d'Amélioration du Bitcoin (Bitcoin Improvement Proposals ou BIP en anglais)

Les propositions d'amélioration du Bitcoin constitue un dossier de conception qui a pour but de fournir des informations à la communauté Bitcoin ou de décrire au fur et à mesure une fonctionnalité relative au Bitcoin, à ses différents processus ainsi qu'à son environnement.

Conformément à l'objet et aux principes directeurs du BIP0001_BIP_, il existe trois types de propositions ou BIP :

_La_BIP _standard

Concerne les changements qui affectent la plupart ou toutes les implémentations de bitcoin, comme une modification du protocole de communication, des règles de validité des blocs ou des transactions, ou encore tout changement ou ajout qui impacte l'interopérabilité des applications utilisant le bitcoin.

_La_BIP _informationnelle

Décrit un problème dans la conception du Bitcoin, ou bien donne des consignes générales ainsi que des informations à la communauté Bitcoin, sans proposer de nouvelle fonctionnalité. Les BIPs informationnelles ne représentent pas nécessairement un consensus ou une recommandation de la communauté ; ce qui, par conséquent, laisse le choix aux utilisateurs et aux développeurs de les ignorer ou de s'y conformer.

_La_BIP_de_processus

Décrit un processus Bitcoin, ou propose une modification (ou un évènement) au niveau d'un processus. Les BIPs de type processus ressemblent aux BIPs standards, mais ne concernent pas le protocole Bitcoin lui-même. Elles peuvent proposer une implémentation, mais pas dans le codebase de Bitcoin. Elles nécessitent souvent un consensus communautaire et, contrairement aux BIPs informationnelles, elles valent plus que de simples recommandations, d'autant plus les utilisateurs ne sont pas libres de les ignorer en général. Procédures, consignes, changements dans le processus décisionnel, modifications apportées aux outils ou à l'environnement relatif au développement du Bitcoin. N'importe quelle métadonnée est considérée comme une BIP de processus.

Les propositions d'amélioration du Bitcoin sont enregistrées dans un répertoire versionné au GitHub. Le tableau Aperçu des BIPs montre un aperçu des BIPs datant de l'automne 2014. Consultez le répertoire officiel pour une mise à jour sur les BIPs existantes et sur leur contenu.

Table 1. Aperçu des BIPs

BIP#	Lien	Titre	Auteur	Type	Statut
1	https://github.com/bitcoin/bips/blob/master/bip-0001.mediawiki	BIP Purpose and Guidelines	Amir Taaki	Standard	Active
10	https://github.com/bitcoin/bips/blob/master/bip-0010.mediawiki	Multi-Sig Transaction Distribution	Alan Reiner	Informationnelle	Brouillon
11	https://github.com/bitcoin/bips/blob/master/bip-0011.mediawiki	M-of-N Standard Transactions	Gavin Andresen	Standard	Acceptée
12	https://github.com/bitcoin/bips/blob/master/bip-0012.mediawiki	OP_EVAL	Gavin Andresen	Standard	Supprimée
13	https://github.com/bitcoin/bips/blob/master/bip-0013.mediawiki	Address Format for pay-to-script-hash	Gavin Andresen	Standard	Final
14	https://github.com/bitcoin/bips/blob/master/bip-0014.mediawiki	Protocol Version and User Agent	Amir Taaki, Patrick Strateman	Standard	Acceptée
15	https://github.com/bitcoin/bips/blob/master/bip-0015.mediawiki	Aliases	Amir Taaki	Standard	Supprimée

BIP#	Lien	Titre	Auteur	Type	Statut
16	https://github.com/bitcoin/bips/blob/master/bip-0016.mediawiki	Pay To Script Hash	Gavin Andresen	Standard	Acceptée
17	https://github.com/bitcoin/bips/blob/master/bip-0017.mediawiki	OP_CHECKHASHVERIFY (CHV)	Luke Dashjr	Supprimée	Brouillon
18	https://github.com/bitcoin/bips/blob/master/bip-0018.mediawikilink:	hashScriptCheck	Luke Dashjr	Standard	Brouillon
19	https://github.com/bitcoin/bips/blob/master/bip-0019.mediawiki	M-of-N Standard Transactions (Low SigOp)	Luke Dashjr	Standard	Brouillon
20	https://github.com/bitcoin/bips/blob/master/bip-0020.mediawiki	URI Scheme	Luke Dashjr	Standard	Remplacée
21	https://github.com/bitcoin/bips/blob/master/bip-0021.mediawiki	URI Scheme	Nils Schneider, Matt Corallo	Standard	Acceptée
22	https://github.com/bitcoin/bips/blob/master/bip-0022.mediawiki	getblocktemplate - Fundamentals	Luke Dashjr	Standard	Acceptée

BIP#	Lien	Titre	Auteur	Type	Statut
23	https://github.com/bitcoin/bips/blob/master/bip-0023.mediawiki	getblocktemplate - Pooled Mining	Luke Dashjr	Standard	Acceptée
30	https://github.com/bitcoin/bips/blob/master/bip-0030.mediawiki	Duplicate transactions	Pieter Wuille	Standard	Acceptée
31	https://github.com/bitcoin/bips/blob/master/bip-0031.mediawiki	Pong message	Mike Hearn	Standard	Acceptée
32	https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki	Hierarchical Deterministic Wallets	Pieter Wuille	Informative	Acceptée
33	https://github.com/bitcoin/bips/blob/master/bip-0033.mediawiki	Stratized Nodes	Amir Taaki	Standard	Brouillon
34	https://github.com/bitcoin/bips/blob/master/bip-0034.mediawiki	Block v2, Height in coinbase	Gavin Andresen	Standard	Acceptée
35	https://github.com/bitcoin/bips/blob/master/bip-0035.mediawiki	mempool message	Jeff Garzik	Standard	Acceptée

BIP#	Lien	Titre	Auteur	Type	Statut
36	https://github.com/bitcoin/bips/blob/master/bip-0036.mediawiki	Custom Services	Stefan Thomas	Standard	Brouillon
37	https://github.com/bitcoin/bips/blob/master/bip-0037.mediawiki	Bloom filtering	Mike Hearn and Matt Corallo	Standard	Acceptée
38	https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki	Passphrase-protected private key	Mike Caldwell	Standard	Brouillon
39	https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki	Mnemonic code for generating deterministic keys	Slush	Standard	Brouillon
40		Stratum wire protocol	Slush	Standard	Numéro de BIP attribué
41		Stratum mining protocol	Slush	Standard	Numéro de BIP attribué
42	https://github.com/bitcoin/bips/blob/master/bip-0042.mediawiki	A finite monetary supply for bitcoin	Pieter Wuille	Standard	Brouillon
43	https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki	Purpose Field for Deterministic Wallets	Slush	Standard	Brouillon

BIP#	Lien	Titre	Auteur	Type	Statut
44	https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki	Multi-Account Hierarchy for Deterministic Wallets	Slush	Standard	Brouillon
50	https://github.com/bitcoin/bips/blob/master/bip-0050.mediawiki	March 2013 Chain Fork Post-Mortem	Gavin Andresen	Informative	Brouillon
60	https://github.com/bitcoin/bips/blob/master/bip-0060.mediawiki	Fixed Length "version" Message (Relay-Transactions Field)	Amir Taaki	Standard	Brouillon
61	https://github.com/bitcoin/bips/blob/master/bip-0061.mediawiki	"reject" P2P message	Gavin Andresen	Standard	Brouillon
62	https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki	Dealing with malleability	Pieter Wuille	Standard	Brouillon
63		Stealth Addresses	Peter Todd	Standard	Numéro de BIP attribué
64	https://github.com/bitcoin/bips/blob/master/bip-0064.mediawiki	getutxos message	Mike Hearn	Standard	Brouillon

BIP#	Lien	Titre	Auteur	Type	Statut
70	https://github.com/bitcoin/bips/blob/master/bip-0070.mediawiki	Payment protocol	Gavin Andresen	Standard	Brouillon
71	https://github.com/bitcoin/bips/blob/master/bip-0071.mediawiki	Payment protocol MIME types	Gavin Andresen	Standard	Brouillon
72	https://github.com/bitcoin/bips/blob/master/bip-0072.mediawiki	Payment protocol URIs	Gavin Andresen	Standard	Brouillon
73	https://github.com/bitcoin/bips/blob/master/bip-0073.mediawiki	Use "Accept" header with Payment Request URLs	Stephen Pair	Standard	Brouillon

Appendix A: pycoin, ku, et tx

La librairie Python pycoin, écrite et maintenue au départ par Richard Kiss, est une librairie Python qui supporte la manipulation de clés et de transactions bitcoin, son niveau de support du langage de script permet même de traiter les transactions non standard.

La biblitothèque pycoin supporte à la fois Python 2 (2.7x) et Python 3 (après 3.3), et est fournie avec certains utilitaires de ligne de commande pratiques, ku et TX.

Key Utility (KU)

L'utilitaire en ligne de commande ku ("key utility") est un couteau suisse pour manipuler des clés. Il supporte les clé BIP32, WIF et les adresses (bitcoin et alt coins). Voici quelques exemples.

Créer un clé BIP32 en utilisant les sources d'entropie par défaut de GPG et de /dev/random:

```
$ ku create

input           : create
network         : Bitcoin
wallet key      : xprv9s21ZrQH143K3LU5ctPZTBnb9kTjA5Su9DcWHvXJemiJBsY7VqXUG7hipgdWaU
                    m2nhnzdvxJf5KJo9vjP2nABX65c5sFsWsV8oXcbpehtJi
public version  : xpub661MyMwAqRbcFpYYiuvZpKjKhnJDZYAkWSY76JvvD7FH4fsG3Nqiov2CfxzxY8
                    DGcpfT56AMFeo8M8KPkFMfLUtvwjwb6WPv8rY65L2q8Hz
tree depth      : 0
fingerprint     : 9d9c6092
parent f'print  : 00000000
child index     : 0
chain code      : 80574fb260edaa4905bc86c9a47d30c697c50047ed466c0d4a5167f6821e8f3c
private key     : yes
secret exponent :
112471538590155650688604752840386134637231974546906847202389294096567806844862
 hex            : f8a8a28b28a916e1043cc0aca52033a18a13cab1638d544006469bc171fddfbe
wif             : L5Z54xi6qJusQT42JHA44mfPVZGjyb4XBRWfxAzUWwRiGx1kV4sP
 uncompressed   : 5KhoEavGNNH4GHKoy2Ptu4KfdNp4r56L5B5un8FP6RZnbsz5Nmb
public pair x   :
764606382405464783648433974782784681018771177678734621270215603682901140160 34
public pair y   :
598078796574697741020401202982722077309212917366332477370774067536768257777 01
 x as hex       : a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 y as hex       : 843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
y parity        : odd
key pair as sec : 03a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
 uncompressed   : 04a90b3008792432060fa04365941e09a8e4adf928bdbdb9dad41131274e379322
                    843a0f6ed9c0eb1962c74533795406914fe3f1957c5238951f4fe245a4fcd625
hash160         : 9d9c609247174ae323acfc96c852753fe3c8819d
 uncompressed   : 8870d869800c9b91ce1eb460f4c60540f87c15d7
Bitcoin address : 1FNNRQ5fSv1wBi5gyfVBs2rkNheMGt86sp
 uncompressed   : 1DSS5isnH4FsVaLVjeVXewVSpfqktdiQAM
```

Créer un clé BIP32 à partir d'une phrase secrète

WARNING	La phrase secrète dans cet exemple est trop facile à deviner

```
$ 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
```

Récupérer les informations en 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":
"04b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f826d8b4d3010aea16ff4c1
c1d3ae68541d9a04df54a2c48cc241c2983544de52",
   "public_pair_x_hex":
"b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f",
   "wallet_key":
"xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFi
dhjFj82pVShWu9curWmb2zy",
   "chain_code": "5eeb1023fd6dd1ae52a005ce0e73420821e1d90e08be980a85e9111fd7646bbc",
   "child_index": "0",
   "hash160_uncompressed": "e5bd3a7e6cb62b4c820e51200fb1c148d79e67da",
   "btc_address": "19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii",
   "fingerprint": "5d353a2e",
   "hash160": "5d353a2ecdb262477172852d57a3f11de0c19286",
   "input": "P:foo",
   "public_pair_x":
"818219827193811040617773492691304190244936166509935893945534043477743931681 91",
   "public_pair_y":
"589942180696054242783207032506897801547850995092776917231263250512004590382 90",
   "key_pair_as_sec":
"02b4e599dfa44555a4ed38bcfff0071d5af676a86abf123c5b4b4e8e67a0b0b13f"
}
```


Clé publique BIP32:

```
$ ku -w -P P:foo
xpub661MyMwAqRbcFVF9ULcqLdsEa5WnCCugQAcgNd9iEMQ31tgH6u4DLQWoQayvtSVYFvXz2vPPpbXE1qpjoUFid
hjFj82pVShWu9curWmb2zy
```

Générer une sous-clé:

```
$ ku -w -s3/2 P:foo
xprv9wTErTSkjVyJa1v4cUTFMFkWMe5eu8ErbQcs9xajnsUzCBT7ykHAwdrxvG3g3f6BFk7ms5hHBvmbdutNmyg6i
ogWKxx6mefEw4M8EroLgKj
```

Sous-clé endurcie:

```
$ ku -w -s3/2H P:foo
xprv9wTErTSu5AWGkDeUPmqBcbZWX1xq85ZNX9iQRQW9DXwygFp7iRGJo79dsVctcsCHsnZ3XU3DhsuaGZbDh8iDk
BN45k67UKsJUXM1JfRCdn1
```

WIF:

```
$ ku -W P:foo
L26c3H6jEPVSqAr1usXUp9qtQJw6NHgApq6Ls4ncyqtsvcq2MwKH
```

Adresse:

```
$ ku -a P:foo
19Vqc8uLTfUonmxUEZac7fz1M5c5ZZbAii
```

Générer un groupe de sous-clés:

```
$ ku P:foo -s 0/0-5 -w
xprv9xWkBDfyBXmZjBG9EiXBpy67KK72fphUp9utJokEBFtjsjiuKUUDF5V3TU8U8cDzytqYnSekc8bYuJS8G3bhX
xKWB89Ggn2dzLcoJsuEdRK
xprv9xWkBDfyBXmZnzKf3bAGifK593gT7WJZPnYAmvc77gUQVej5QHckc5Adtwxa28ACmANi9XhCrRvtFqQcUxt8r
UgFz3souMiDdWxJDZnQxzx
xprv9xWkBDfyBXmZqdXA8y4SWqfBdy71gSW9sjx9JpCiJEiBwSMQyRxan6srXUPBtj3PTxQFkZJAiwoUpmvtrxKZu
4zfsnr3pqyy2vthpkwuoVq
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
xprv9xWkBDfyBXmZv2q3N66hhZ8DAcEnQDnXML1J62krJAcf7Xb1HJwuW2VMJQrCofY2jtFXdiEY8UsRNJfqK6DAd
yZXoMvtaLHyWQx3FS4A9zw
xprv9xWkBDfyBXmZw4jEYXUHYc9fT25k9irP87n2RqfJ5bqbjKdT84Mm7Wtc2xmzFuKg7iYf7XFHKkSsaYKWKJbR5
4bnyAD9GzjUYbAYTtN4ruo
```

Générer l’adresse correspondante:

```
$ ku P:foo -s 0/0-5 -a
1MrjE78H1R1rqdFrmkjdHnPUdLCJALbv3x
1AnYyVEcuqeoVzH96zj1eYKwoWfwte2pxu
1GXr1kZfxE1FcK6ZRD5sqqqs5YfvuzA1Lb
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
1Cz2rTLjRM6pMnxPNrRKp9ZSvRtj5dDUML
1WstdwPnU6HEUPme1DQayN9nm6j7nDVEM
```

Générer les WIFs correspondants:

```
$ ku P:foo -s 0/0-5 -W
L5a4iE5k9gcJKGqX3FWmxzBYQc29PvZ6pgBaePLVqT5YByEnBomx
Kyjgne6GZwPGB6G6kJEhoPbmyjMP7D5d3zRbHVjwcq4iQXD9QqKQ
L4B3ygQxK6zH2NQGxLDee2H9v4Lvwg14cLJW7QwWPzCtKHdWMaQz
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
L2oD6vA4TUyqPF8QG4vhUFSgwCyuuvFZ3v8SKHYFDwkbM765Nrfd
KzChTbc3kZFxUSJ3Kt54cxsogeFAD9CCM4zGB22si8nfKcThQn8C
```

Vérifier si cela fonctionne en choisissant une chaîne BIP32 (celle correspondant à la sous-clé 0/3):

```
$ ku -W
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
L2L2PZdorybUqkPjrmhem4Ax5EJvP7ijmxbNoQKnmTDMrqemY8UF
$ ku -a
xprv9xWkBDfyBXmZsA85GyWj9uYPyoQv826YAadKWMaaEosNrFBKgj2TqWuiWY3zuqxYGpHfv9cnGj5P7e8EskpzK
L1Y8Gk9aX6QbryA5raK73p
116AXZc4bDVQrqmcinzu4aaPdrYqvuiBEK
```

Ouais, cela semble familier.

De l’exposant secret :

```
$ 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
```

Version de Litecoin:

```
$ ku -nL 1

input            : 1
network          : Litecoin
secret exponent  : 1
 hex             : 1
wif              : T33ydQRKp4FCW5LCLLUB7deioUMoveiwekdwUwyfRDeGZm76aUjV
 uncompressed    : 6u823ozcyt2rjPH8Z2ErsSXJB5PPQwK7VVTwwN4mxLBFrao69XQ
public pair x    :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y    :
32670510020758816978083085130507043184471273380659243275938904335757337482424
 x as hex        : 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 y as hex        : 483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
y parity         : even
key pair as sec  : 0279be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
 uncompressed    : 0479be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798
                     483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
 uncompressed    : 91b24bf9f5288532960ac687abb035127b1d28a5
Litecoin address : LVuDpNCSSj6pQ7t9Pv6d6sUkLKoqDEVUnJ
 uncompressed    : LYWKqJhtPeGyBAw7WC8R3F7ovxtzAiubdM
```

Dogecoin WIF:

```
$ ku -nD -W 1
QNcdLVw8fHkixm6NNyN6nVwxKek4u7qrioRbQmjxac5TVoTtZuot
```

De la public pair (sur le Testnet) :

```
$ ku -nT
550662630222773436695787188951685343262506034537775941755001873603 89116729240,even

input                   :
550662630222773436695787188951685343262506034537775941755001873603
                           89116729240,even
network                 : Bitcoin testnet
public pair x           :
55066263022277343669578718895168534326250603453777594175500187360389116729240
public pair y           :
326705100207588169780830851305070431844712733806592432759389043357573374824 24
 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
```

À partir du hash160 :

```
$ ku 751e76e8199196d454941c45d1b3a323f1433bd6

input           : 751e76e8199196d454941c45d1b3a323f1433bd6
network         : Bitcoin
hash160         : 751e76e8199196d454941c45d1b3a323f1433bd6
Bitcoin address : 1BgGZ9tcN4rm9KBzDn7KprQz87SZ26SAMH
```

Comme une adresse Dogecoin:

```
$ ku -nD 751e76e8199196d454941c45d1b3a323f1433bd6

input            : 751e76e8199196d454941c45d1b3a323f1433bd6
network          : Dogecoin
hash160          : 751e76e8199196d454941c45d1b3a323f1433bd6
Dogecoin address : DFpN6QqFfUm3gKNaxN6tNcab1FArL9cZLE
```

Transaction Utility (TX)

L'utilitaire en ligne de commande tx affichera des transactions lisibles par un humain, cherchera les transactions de bases de la cache de transactions de pycoin ou des services web (blockchain.info, blockr.io et biteasy.com sont actuellement pris en charge), fusionnera les transactions, ajoutera ou supprimera les entreés et sorties, et signera les transactions.

Voici quelques exemples

Voir la fameuse transaction "pizza" [PIZZA]:

```
$ tx 49d2adb6e476fa46d8357babf78b1b501fd39e177ac7833124b3f67b17c40c2a
warning: consider setting environment variable PYCOIN_CACHE_DIR=~/.pycoin_cache to
cache transactions fetched via web services
warning: no service providers found for get_tx; consider setting environment variable
PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
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! Nous n'avons pas mis en place les services web. Faisons cela maintenant:

```
$ PYCOIN_CACHE_DIR=~/.pycoin_cache
$ PYCOIN_SERVICE_PROVIDERS=BLOCKR_IO:BLOCKCHAIN_INFO:BITEASY:BLOCKEXPLORER
$ export PYCOIN_CACHE_DIR PYCOIN_SERVICE_PROVIDERS
```

Ce n'est pas fait automatiquement de sorte qu'un outil en ligne de commande ne divulguera pas à une site tiers les informations privées des transactions qui vous intéresse. Si vous ne vous en souciez pas,

vous pouvez mettre ces lignes dans votre *.profile*.

Essayons de nouveau:

```
$ 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
010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a4
93046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e
9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e8000
0001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

** can't validate transaction as source transactions missing
```

La dernière ligne apparaît car pour valider les signatures des transactions, vous avez techniquement besoin des transactions sources. Donc, ajoutons -a pour agrandir les transactions avec l'information source:

```
$ 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 propogate
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

010000000141045e0ab2b0b82cdefaf9e9a8ca9ec9df17673d6a74e274d0c73ae77d3f131e000000004a4
93046022100a7f26eda874931999c90f87f01ff1ffc76bcd058fe16137e0e63fdb6a35c2d78022100a61e
9199238eb73f07c8f209504c84b80f03e30ed8169edd44f80ed17ddf451901ffffffff010010a5d4e8000
0001976a9147ec1003336542cae8bded8909cdd6b5e48ba0ab688ac00000000

all incoming transaction values validated
```

Regardons maintenant les sorties non dépensées pour une adresse spécifique (UTXO). Dans le bloc numéro 1, nous voyons une transaction coinbase de 12c6DSiU4Rq3P4ZxziKxzrL5LmMBrzjrJX. Nous pouvons fetch_unspent pour trouver tous les coins de cette adresse :

```
$ 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
```

Appendix A: Opérateurs du langage de script des transactions, Constantes et Symboles

Place une valeur sur la pilemontre les opérateurs qui placent les valeurs sur la pile.

Table 1. Place une valeur sur la pile

Symbole	Valeur (hex)	Description
OP_0 ou OP_FALSE	0x00	Un tableau vide est placé sur la plie
1-75	0x01-0x4b	Place les N octets suivants sur la pile, avec N compris entre 1et 75 octets
OP_PUSHDATA1	0x4c	Le prochain octet de script contient N, place les N octets suivants sur la pile
OP_PUSHDATA2	0x4d	Les deux prochains octets de script contiennent N, place les N octets suivants sur la plie
OP_PUSHDATA4	0x4e	Les quatres prochains octets de script contiennent N, place les N octets suivants sur la pile
OP_1NEGATE	0x4f	Place la valeur "–1" sur la pile
OP_RESERVED	0x50	Halte - Transaction invalide à moins d'être trouvée dans un clause OP_IF non exécutée
OP_1 ou OP_TRUE	0x51	Place la valeur "1" sur la pile
OP_2 à OP_16	0x52 to 0x60	Pour OP_N, place la valeur "N" sur la pile. Par exemple., OP_2 place la valeur "2"

Contrôle de flux conditionnel montre les opérateurs de contrôle de flux conditionnels.

Table 2. Contrôle de flux conditionnel

Symbole	Valeur (hex)	Description
OP_NOP	0x61	Ne fait rien
OP_VER	0x62	Halte - Transaction invalide à moins d'être trouvée dans un clause OP_IF non exécutée

Symbole	Valeur (hex)	Description
OP_IF	0x63	Exécute les commandes suivantes si la valeur du haut de la pile est différente de 0
OP_NOTIF	0x64	Exécute les commandes suivantes si la valeur du haut de la pile est égale à 0
OP_VERIF	0x65	Halte - Transaction invalide
OP_VERNOTIF	0x66	Halte - Transaction invalide
OP_ELSE	0x67	Exécute la commande si les commande précédentes n'ont pas été exécutées
OP_ENDIF	0x68	Mar la fin des blocs OP_IF, OP_NOTIF, OP_ELSE
OP_VERIFY	0x69	Vérifie le haut de la pile, arrête et invalide la transaction si elle est différente de TRUE
OP_RETURN	0x6a	Arrête et invalide la transaction

Opération de pile montre les opérateurs utilisés pour manipuler la pile.

Table 3. Opération de pile

Symbole	Valeur (hex)	Description
OP_TOALTSTACK	0x6b	Retire l'objet du haut de la pile et le place dans une pile alternative
OP_FROMALTSTACK	0x6c	Retire l'objet du haut de la pile alternative et la place dans la pile
OP_2DROP	0x6d	Retire deux objets de la pile
OP_2DUP	0x6e	Duplique les deux objets du haut de la pile
OP_3DUP	0x6f	Duplique les trois objets du haut de la pile
OP_2OVER	0x70	Copie le troisième et le quatirème objet de la pile vers le haut de la pile

Symbole	Valeur (hex)	Description
OP_2ROT	0x71	Copie le cinquième et le sixième objet de la pile vers le haut de la pile
OP_2SWAP	0x72	Echange les deux premières paires d'objet dans la pile
OP_IFDUP	0x73	Duplique l'objet du haut de la pile si il est différent de 0
OP_DEPTH	0x74	Compte les objets dans la pile et retourne le résultat
OP_DROP	0x75	Retire l'objet du haut de la pile
OP_DUP	0x76	Duplique l'objet du haut de la pile
OP_NIP	0x77	Retire le deuxième objet de la pile
OP_OVER	0x78	Copie le deuxième objet de la pile et le place en haut
OP_PICK	0x79	Retire la valeur N du haut, et la copie N fois l'objet en haut de la pile
OP_ROLL	0x7a	Retire la valeur N du haut, et la déplace N fois vers le haut de la pile
OP_ROT	0x7b	effectue une rotation des trois premières valeurs de la pile
OP_SWAP	0x7c	Echange les trois objets du haut de la pile
OP_TUCK	0x7d	Copie la valeur du haut de la pile et l'insère entre la première et la deuxième valeur de la pile.

Opérations sur les chaines de caractères montre les opérateurs de chaîne de caractères.

Table 4. Opérations sur les chaines de caractères

Symbole	Valeur (hex)	Description
OP_CAT	0x7e	Désactivé (concatène les deux objets du haut)

Symbole	Valeur (hex)	Description
OP_SUBSTR	0x7f	Désactivé (retourne une sous-chaîne)
OP_LEFT	0x80	Désactivé (retourne la sous-chaine de gauche)
OP_RIGHT	0x81	Désactivé (retourne la sous-chaine de droite)
OP_SIZE	0x82	Calcule la longueur de la chaine de caractères du haut de la pile et retourne le résultat

Arithmétique binaire et conditions montre les opérateurs de logique booléenne et d'arithmétique binaire.

Table 5. Arithmétique binaire et conditions

Symbole	Valeur (hex)	Description
OP_INVERT	0x83	Désactivé (inverse les bits de la valeur du haut de la pile)
OP_AND	0x84	Désactivé (effectue un AND booléen des deux premières valeur de la pile)
OP_OR	0x85	Désactivé (effectue un OR booléen des deux premières valeur de la pile))
OP_XOR	0x86	Désactivé (effectue un XOR booléen des deux premières valeur de la pile)
OP_EQUAL	0x87	Place TRUE (1) si les deux valeur du haut sont exactement égales, place FALSE (0) autrement
OP_EQUALVERIFY	0x88	Pareil que OP_EQUAL, mais exécute OP_VERIFY ensuite pour arrêter l'exécution si le résultat est différent de TRUE
OP_RESERVED1	0x89	Arrêt - Transaction invalide à moins que cet opération ne soit placée dans une clause OP_IF non exécutée

Symbole	Valeur (hex)	Description
OP_RESERVED2	0x8a	Arrêt - Transaction invalide à moins que cet opération ne soit placée dans une clause OP_IF non exécutée

Opérateurs numériques shows numeric (arithmetic) operators.

Table 6. Opérateurs numériques

Symbole	Valeur (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

Symbole	Valeur (hex)	Description
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

Opérations cryptographiques et de hashage shows cryptographic function operators.

Table 7. Opérations cryptographiques et de hashage

Symbole	Valeur (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

Symbole	Valeur (hex)	Description
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_NOP as workaround
OP_CHECKMULTISIGVERIFY	0xaf	Same as CHECKMULTISIG, then OP_VERIFY to halt if not TRUE

Non-operators shows nonoperator symbols

Table 8. Non-operators

Symbole	Valeur (hex)	Description
OP_NOP1-OP_NOP10	0xb0-0xb9	Does nothing, ignored

Reserved OP codes for internal use by the parser shows operator codes reserved for use by the internal script parser.

Table 9. Reserved OP codes for internal use by the parser

Symbole	Valeur (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