Clés, Adresses

2022-03-04

# Clés, Adresses

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

## Présentation

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

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

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

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

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

### Cryptographie à clé publique et crypto-monnaie

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

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

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

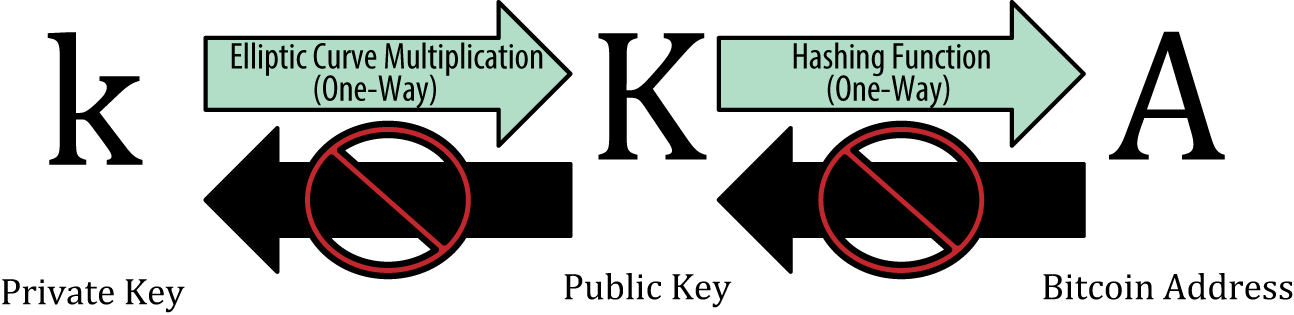
Il existe une relation mathématique entre la clé publique et la clé privée qui permet d'utiliser la clé privée pour générer des signatures sur les messages. Ces signatures peuvent être validées par rapport à la clé publique sans révéler la clé privée.

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

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

### Clés privées et publiques

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



Clé privée, clé publique et adresse Bitcoin

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

### Clés privées

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

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

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

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

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

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

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

1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD

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

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

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

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

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

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

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

### Clés publiques

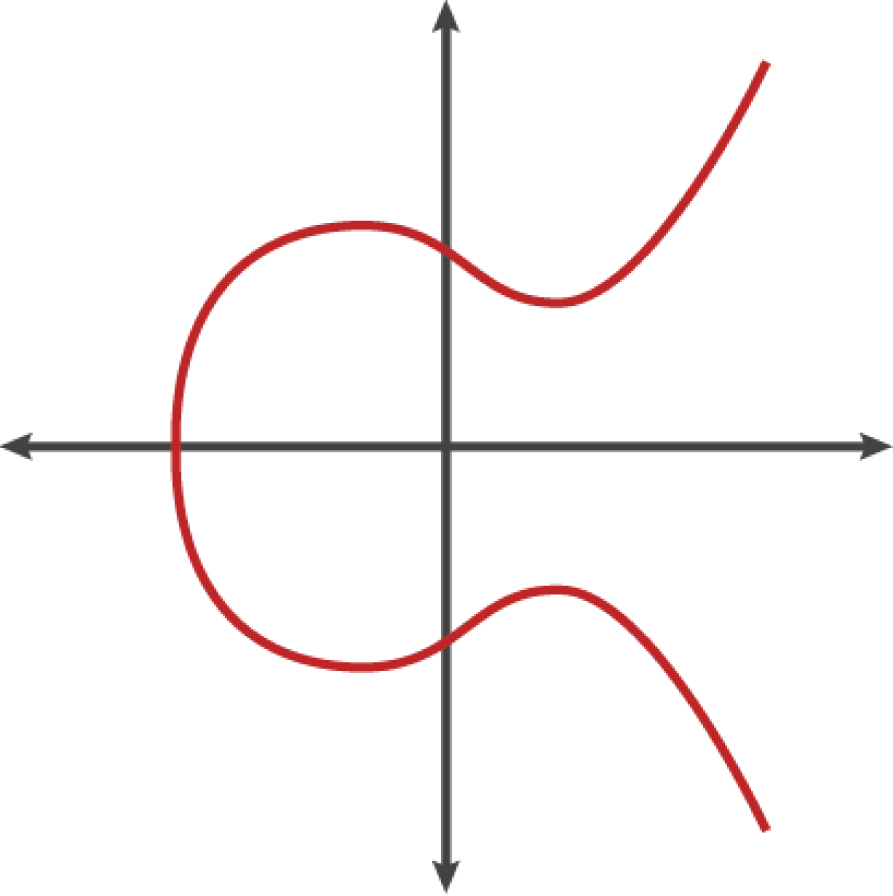
La clé publique est calculée à partir de la clé privée en utilisant la multiplication par courbe elliptique, qui est irréversible : *K* = *k* \* *G*, où *k* est la clé privée, *G* est un point constant appelé *point générateur* et *K* est la clé publique résultante. L'opération inverse, connue sous le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

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

### La cryptographie à courbe elliptique expliquée

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

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



Une courbe elliptique

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

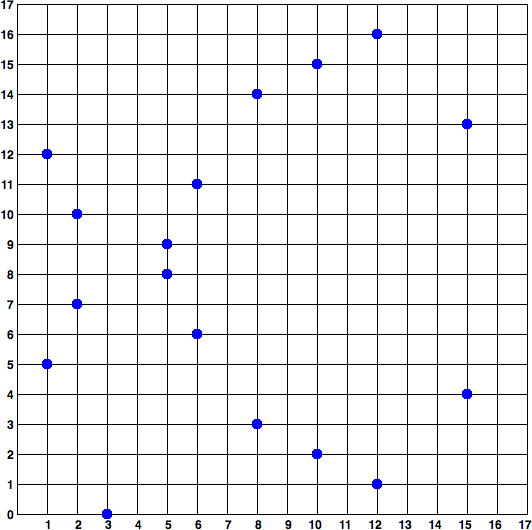
\begin{equation} {y^2 = (x^3 + 7)}~\text{over}~(\mathbb{F}\_p) \end{equation}

ou

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

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

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



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

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

P = (55066263022277343669578718895168534326250603453777594175500187360389116729240, 32670510020733184471273335757337482424)

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

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

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

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

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

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

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

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

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

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

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

### Génération d'une clé publique

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

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

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

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

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

K = 1E99423A4ED27608A15A2616A2B0E9E52CED330AC530EDCC32C8FFC6A526AEDD \* G

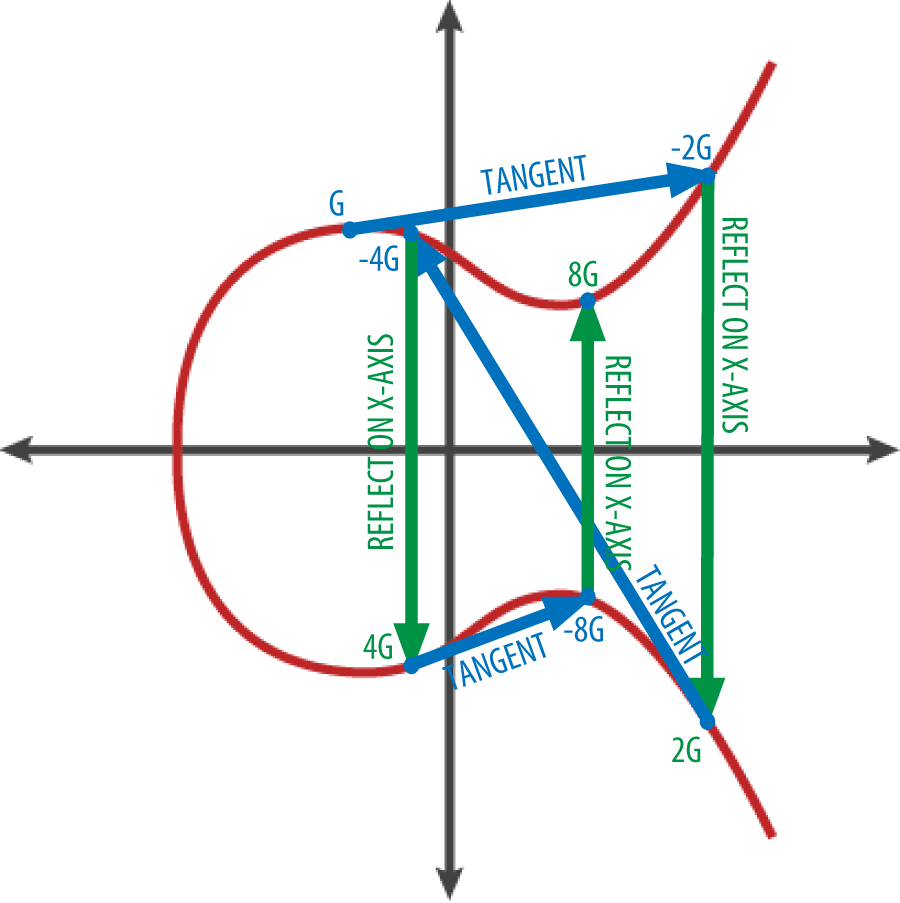
La clé publique *K* est définie comme un point K = (x,y) :

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

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

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

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



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

## Adresses Bitcoin

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

1J7mdg5rbQyUHENYdx39WVWK7fsLpEoXZy

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

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

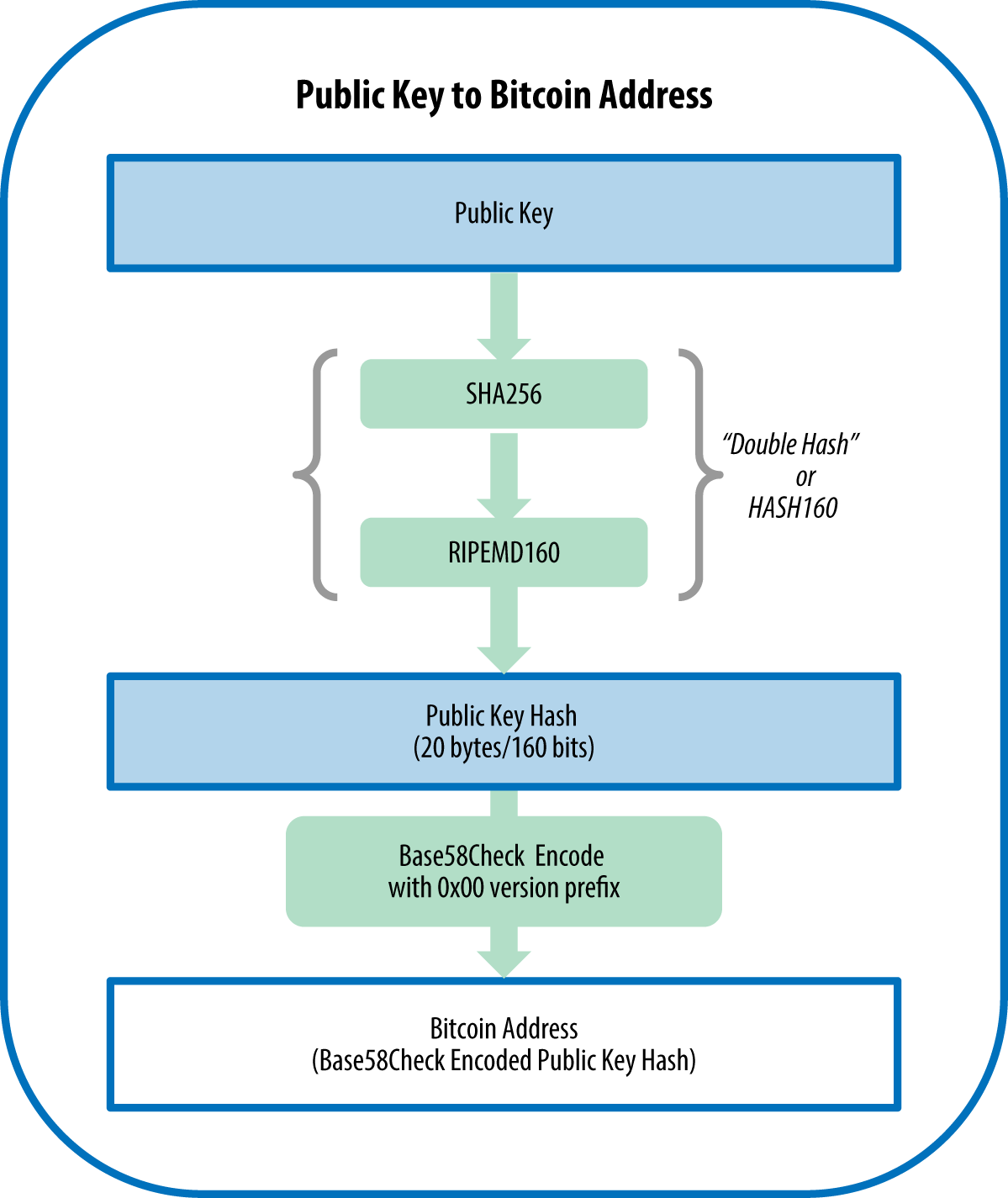
En commençant par la clé publique *K*, nous calculons le hachage SHA256, puis calculons le hachage RIPEMD160 du résultat, produisant un nombre de 160 bits (20 octets) :

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

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

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

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



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

### Encodage Base58 et Base58Check

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

Alphabet Base58 de .Bitcoin

123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz

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

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

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

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

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

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

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

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

Base58Check préfixe de version et exemples de résultats encodés

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

### Formats de clé

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

#### Formats de clé privée

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

Représentations de clés privées (formats d'encodage)

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

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

Exemple : même clé, différents formats

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

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

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

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

#### Décoder à partir de Base58Check

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

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

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

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

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

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

#### Encoder de l'hex vers Base58Check

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

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

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

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

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

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

#### Formats de clé publique

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

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

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

x = F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A  
y = 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

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

K = 04F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A↵ 07CF33DA18BD734C600B96A72BBC4749D5141C90EC8AC328AE52DDFE2E505BDB

#### Clés publiques compressées

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

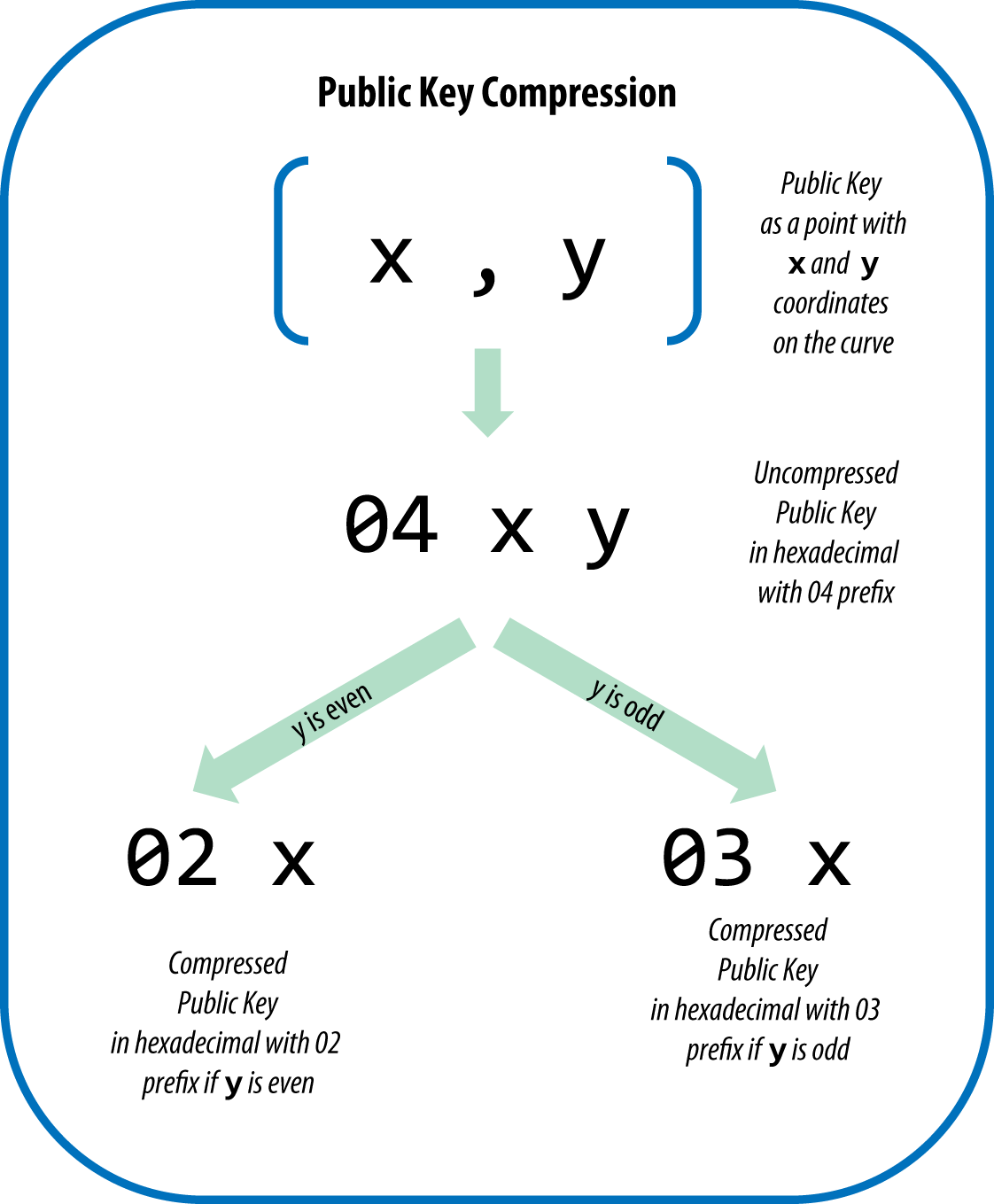
Comme nous l'avons vu dans la section <<pubkey> >, une clé publique est un point (x,y) sur une courbe elliptique. Parce que la courbe exprime une fonction mathématique, un point sur la courbe représente une solution à l'équation et, par conséquent, si nous connaissons la coordonnée *x*, nous pouvons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

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

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

K = 03F028892BAD7ED57D2FB57BF33081D5CFCF6F9ED3D3D7F159C2E2FFF579DC341A

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



Compression de clé publique

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

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

#### Clés privées compressées

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

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

Exemple : même clé, différents formats

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

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

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

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

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

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

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

#include <bitcoin/bitcoin.hpp>  
  
int main()  
{  
 // Private secret key string as base16  
 bc::ec\_secret decoded;  
 bc::decode\_base16(decoded,  
 "038109007313a5807b2eccc082c8c3fbb988a973cacf1a7df9ce725c31b14776");  
  
 bc::wallet::ec\_private secret(  
 decoded, bc::wallet::ec\_private::mainnet\_p2kh);  
  
 // Get public key.  
 bc::wallet::ec\_public public\_key(secret);  
 std::cout << "Public key: " << public\_key.encoded() << std::endl;  
  
 // Create Bitcoin address.  
 // Normally you can use:  
 // bc::wallet::payment\_address payaddr =  
 // public\_key.to\_payment\_address(  
 // bc::wallet::ec\_public::mainnet\_p2kh);  
 // const std::string address = payaddr.encoded();  
  
 // Compute hash of public key for P2PKH address.  
 bc::data\_chunk public\_key\_data;  
 public\_key.to\_data(public\_key\_data);  
 const auto hash = bc::bitcoin\_short\_hash(public\_key\_data);  
  
 bc::data\_chunk unencoded\_address;  
 // Reserve 25 bytes  
 // [ version:1 ]  
 // [ hash:20 ]  
 // [ checksum:4 ]  
 unencoded\_address.reserve(25);  
 // Version byte, 0 is normal BTC address (P2PKH).  
 unencoded\_address.push\_back(0);  
 // Hash data  
 bc::extend\_data(unencoded\_address, hash);  
 // Checksum is computed by hashing data, and adding 4 bytes from hash.  
 bc::append\_checksum(unencoded\_address);  
 // Finally we must encode the result in Bitcoin's base58 encoding.  
 assert(unencoded\_address.size() == 25);  
 const std::string address = bc::encode\_base58(unencoded\_address);  
  
 std::cout << "Address: " << address << std::endl;  
 return 0;  
}

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

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

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

## Implémenter des clés et des adresses en Python

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

import bitcoin  
  
# Generate a random private key  
valid\_private\_key = False  
while not valid\_private\_key:  
 private\_key = bitcoin.random\_key()  
 decoded\_private\_key = bitcoin.decode\_privkey(private\_key, 'hex')  
 valid\_private\_key = 0 < decoded\_private\_key < bitcoin.N  
  
print "Private Key (hex) is: ", private\_key  
print "Private Key (decimal) is: ", decoded\_private\_key  
  
# Convert private key to WIF format  
wif\_encoded\_private\_key = bitcoin.encode\_privkey(decoded\_private\_key, 'wif')  
print "Private Key (WIF) is: ", wif\_encoded\_private\_key  
  
# Add suffix "01" to indicate a compressed private key  
compressed\_private\_key = private\_key + '01'  
print "Private Key Compressed (hex) is: ", compressed\_private\_key  
  
# Generate a WIF format from the compressed private key (WIF-compressed)  
wif\_compressed\_private\_key = bitcoin.encode\_privkey(  
 bitcoin.decode\_privkey(compressed\_private\_key, 'hex'), 'wif')  
print "Private Key (WIF-Compressed) is: ", wif\_compressed\_private\_key  
  
# Multiply the EC generator point G with the private key to get a public key point  
public\_key = bitcoin.fast\_multiply(bitcoin.G, decoded\_private\_key)  
print "Public Key (x,y) coordinates is:", public\_key  
  
# Encode as hex, prefix 04  
hex\_encoded\_public\_key = bitcoin.encode\_pubkey(public\_key,'hex')  
print "Public Key (hex) is:", hex\_encoded\_public\_key  
  
# Compress public key, adjust prefix depending on whether y is even or odd  
(public\_key\_x, public\_key\_y) = public\_key  
if (public\_key\_y % 2) == 0:  
 compressed\_prefix = '02'  
else:  
 compressed\_prefix = '03'  
hex\_compressed\_public\_key = compressed\_prefix + bitcoin.encode(public\_key\_x, 16)  
print "Compressed Public Key (hex) is:", hex\_compressed\_public\_key  
  
# Generate bitcoin address from public key  
print "Bitcoin Address (b58check) is:", bitcoin.pubkey\_to\_address(public\_key)  
  
# Generate compressed bitcoin address from compressed public key  
print "Compressed Bitcoin Address (b58check) is:", \  
 bitcoin.pubkey\_to\_address(hex\_compressed\_public\_key)

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

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

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

import ecdsa  
import os  
from ecdsa.util import string\_to\_number, number\_to\_string  
  
# secp256k1, http://www.oid-info.com/get/1.3.132.0.10  
\_p = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEFFFFFC2FL  
\_r = 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFEBAAEDCE6AF48A03BBFD25E8CD0364141L  
\_b = 0x0000000000000000000000000000000000000000000000000000000000000007L  
\_a = 0x0000000000000000000000000000000000000000000000000000000000000000L  
\_Gx = 0x79BE667EF9DCBBAC55A06295CE870B07029BFCDB2DCE28D959F2815B16F81798L  
\_Gy = 0x483ada7726a3c4655da4fbfc0e1108a8fd17b448a68554199c47d08ffb10d4b8L  
curve\_secp256k1 = ecdsa.ellipticcurve.CurveFp(\_p, \_a, \_b)  
generator\_secp256k1 = ecdsa.ellipticcurve.Point(curve\_secp256k1, \_Gx, \_Gy, \_r)  
oid\_secp256k1 = (1, 3, 132, 0, 10)  
SECP256k1 = ecdsa.curves.Curve("SECP256k1", curve\_secp256k1, generator\_secp256k1, oid\_secp256k1)   
ec\_order = \_r  
  
curve = curve\_secp256k1  
generator = generator\_secp256k1  
  
def random\_secret():  
 convert\_to\_int = lambda array: int("".join(array).encode("hex"), 16)  
   
 # Collect 256 bits of random data from the OS's cryptographically secure random generator  
 byte\_array = os.urandom(32)  
   
 return convert\_to\_int(byte\_array)  
  
def get\_point\_pubkey(point):  
 if point.y() & 1:  
 key = '03' + '%064x' % point.x()  
 else:  
 key = '02' + '%064x' % point.x()  
 return key.decode('hex')  
  
def get\_point\_pubkey\_uncompressed(point):  
 key = '04' + \  
 '%064x' % point.x() + \  
 '%064x' % point.y()  
 return key.decode('hex')  
  
  
# Generate a new private key.  
secret = random\_secret()  
print "Secret: ", secret  
  
# Get the public key point.  
point = secret \* generator  
print "EC point:", point  
  
print "BTC public key:", get\_point\_pubkey(point).encode("hex")  
  
# Given the point (x, y) we can create the object using:  
point1 = ecdsa.ellipticcurve.Point(curve, point.x(), point.y(), ec\_order)  
assert point1 == point

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

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

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

## Clés et adresses avancées

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

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

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

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

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

hachage de script = RIPEMD160(SHA256(script))

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

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

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

#### Adresses multisignatures et P2SH

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

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

### Adresses personnalisées

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

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

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

#### Génération d'adresses personnalisées

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

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

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

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

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

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

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

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

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

Mineur d'adresse .Vanity

#include <random>  
#include <bitcoin/bitcoin.hpp>  
  
// The string we are searching for  
const std::string search = "1kid";  
  
// Generate a random secret key. A random 32 bytes.  
bc::ec\_secret random\_secret(std::default\_random\_engine& engine);  
// Extract the Bitcoin address from an EC secret.  
std::string bitcoin\_address(const bc::ec\_secret& secret);  
// Case insensitive comparison with the search string.  
bool match\_found(const std::string& address);  
  
int main()  
{  
 // random\_device on Linux uses "/dev/urandom"  
 // CAUTION: Depending on implementation this RNG may not be secure enough!  
 // Do not use vanity keys generated by this example in production  
 std::random\_device random;  
 std::default\_random\_engine engine(random());  
   
 // Loop continuously...  
 while (true)  
 {  
 // Generate a random secret.  
 bc::ec\_secret secret = random\_secret(engine);  
 // Get the address.  
 std::string address = bitcoin\_address(secret);  
 // Does it match our search string? (1kid)  
 if (match\_found(address))  
 {  
 // Success!  
 std::cout << "Found vanity address! " << address << std::endl;  
 std::cout << "Secret: " << bc::encode\_base16(secret) << std::endl;  
 return 0;  
 }  
 }  
 // Should never reach here!  
 return 0;  
}  
  
bc::ec\_secret random\_secret(std::default\_random\_engine& engine)  
{  
 // Create new secret...  
 bc::ec\_secret secret;  
 // Iterate through every byte setting a random value...  
 for (uint8\_t& byte: secret)  
 byte = engine() % std::numeric\_limits<uint8\_t>::max();  
 // Return result.  
 return secret;  
}  
  
std::string bitcoin\_address(const bc::ec\_secret& secret)  
{  
 // Convert secret to payment address  
 bc::wallet::ec\_private private\_key(secret);  
 bc::wallet::payment\_address payaddr(private\_key);  
 // Return encoded form.  
 return payaddr.encoded();  
}  
  
bool match\_found(const std::string& address)  
{  
 auto addr\_it = address.begin();  
 // Loop through the search string comparing it to the lower case  
 // character of the supplied address.  
 for (auto it = search.begin(); it != search.end(); ++it, ++addr\_it)  
 if (\*it != std::tolower(\*addr\_it))  
 return false;  
 // Reached end of search string, so address matches.  
 return true;  
}

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

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

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

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

#### Sécurité de l'adresse personnalisée

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

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

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

Génération d'adresses personnalisées pour correspondre à une adresse aléatoire

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

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

### Portefeuilles en papier

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

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

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

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

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

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



Un exemple de portefeuille en papier simple

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

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



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