Bitcoin démocratisé.

La série d'ebooks gratuits pour comprendre les rouages techniques de Bitcoin.

Loïc Morel

Pandul

Cet ouvrage est mis à disposition selon les termes de la Licence Creative Commons : Attribution - Partage dans les Mêmes Conditions 4.0 International (CC BY-SA 4.0), à l'exception des logos de Pandul seuls qui demeurent la propriété intellectuelle de l'Ei Loïc Morel.

Pour en savoir plus, cliquez ici :
https://creativecommons.org/licenses/by-sa/4.0/

Pour résumer, vous avez le droit de :

Partager, copier et redistribuer le contenu texte et illustrations sur n'importe quel support ou format, à l'exception des logos de Pandul seuls qui sont strictement protégés.

Adapter, remixer, transformer et construire sur le contenu texte et illustrations, à quelque fin que ce soit, même commercialement, à l'exception des logos de Pandul seuls qui sont strictement protégés.

En respectant les termes suivants :

Attribution - Vous devez donner le crédit approprié (Pandul et Loïc Morel), fournir un lien vers la licence et indiquer si des modifications ont été apportées. Vous pouvez le faire de toute manière raisonnable, mais en aucun cas suggérant que le concédant vous approuve ou approuve votre utilisation.

Partage dans les mêmes conditions - Si vous copiez, utilisez, remixez, transformez ou développez le contenu, vous devez distribuer vos contributions sous la même licence que l'original.

Pour disposer de cet ebook en format .odt, merci de bien vouloir me contacter par mail à loic@pandul.fr.

L'intégralité des informations contenues dans ce livre ne constitue ni un conseil en investissements, ni une sollicitation à investir, ni une offre quelconque d'achat ou de vente, ni un conseil en systèmes et logiciels informatiques.

Le lecteur demeure entièrement propriétaire et responsable de ses décisions et de ses éventuels actifs numériques à tout moment.

Aucune responsabilité ne saurait donc être engagée.

1/ Bitcoin et la cryptographie.

Pandul

Sommaire.

Remerciements.

Je voudrais remercier sincèrement l'ensemble des Bitcoiners qui m'ont apporté leur aide, leurs conseils d'experts et leurs encouragements sur cette série d'ebooks :

- Grittoshi → https://twitter.com/Grittoshi
-

Merci également à tous ceux qui ont apporté leur aide sur ce projet, mais qui ont préféré rester anonymes.

Préambule.

La série d'ebooks ***Bitcoin Démocratisé*** a pour objectif d'apporter une base de connaissances techniques sur Bitcoin en français, à travers un format qui permet le développement et la liberté de rédaction.

Elle s'adresse aux personnes qui souhaitent découvrir comment fonctionnent les rouages derrière Bitcoin, soit dans le but de progresser dans son utilisation, soit par simple curiosité. Vous y trouverez beaucoup de vulgarisations techniques, avec des schémas et des illustrations, mais également des astuces très concrètes et pratiques.

Que ce soit sur les réseaux sociaux, lors des rencontres avec mes clients ou quand je participe à des évènements liés à Bitcoin, j'ai pu constater qu'une question revient constamment : Quel contenu francophone me conseillerez-vous pour approfondir mes connaissances sur le sujet ?

L'objectif de ces humbles ouvrages est d'essayer d'apporter une réponse à cette question. Ils seront publiés sous forme d'une série non exhaustive, afin de pouvoir traiter des points précis sur Bitcoin en entrant à chaque fois dans le détail pour chaque sujet.

J'ai souhaité que cette série soit entièrement gratuite et sans contrepartie afin que chacun puisse disposer de ces informations à but pédagogique. J'espère sincèrement que ces ebooks pourront aider le plus grand nombre possible d'utilisateurs francophones de Bitcoin.

Chaque ouvrage est disponible sur mon site web www.pandul.fr.

Si vous ne comprenez pas certains mots techniques utilisés dans mes ebooks, un glossaire avec des définitions est disponible sur cette page.

Je vous souhaite une excellente lecture et reste à votre entière disposition pour toute demande complémentaire. Mes coordonnées professionnelles et réseaux sociaux sont disponibles à la fin de chaque ouvrage.

Loïc Morel

Introduction.

Ce premier tome de la série est en quelque sorte préparatoire pour la suite de la série d'ebooks. Nous allons y étudier les différents algorithmes de chiffrement qui interviennent au sein du protocole Bitcoin. L'objectif sera de vulgariser et d'entrer dans le détail, mais promis, je ne vais pas vous faire un cours de mathématiques pures. Il y en aura un petit peu, puisqu'elles sont à la base de la cryptographie, mais le but sera que quiconque puisse comprendre l'idée derrière chaque mécanisme.

Toute la sécurité inhérente à Bitcoin est basée sur cette cryptographie. On la retrouve à tous les étages du protocole. La cryptographie est dans les wallets, dans les blocs, dans la communication… Elle est de partout sur Bitcoin. Comprendre comment elle fonctionne, et pourquoi elle est utilisée dans le protocole, représente selon moi un premier pas indispensable pour étudier par la suite les rouages techniques de Bitcoin.

Mais finalement, **c'est quoi la cryptographie ?**

Etymologiquement, le mot "cryptographie" vient des mots en grec ancien "kruptos" qui veut dire "caché", et "graphein" qui veut dire "écriture". La cryptographie c'est donc la science de cacher un message.

On la classe parmi les sciences de la cryptologie qui englobent également l'authentification, la non-répudiation ou encore la cryptanalyse. Dans le langage courant, on confond souvent ces deux termes en un seul.

Il est intéressant de remarquer que cette science existe au moins depuis l'antiquité avec notamment le chiffre de César, une méthode de chiffrement d'un message par simple décalage de lettres.

Dans le protocole Bitcoin, les deux principales applications de la cryptographie et de la cryptologie sont les fonctions de hachage et les signatures numériques. Voyons ensemble en quoi elles consistent ? Qu'est-ce qu'un hash ? Ou encore comment fonctionnent les paires de clés publiques et clés privées ?

1- Les fonctions de hachage.

Caractéristiques d'une fonction de hachage.

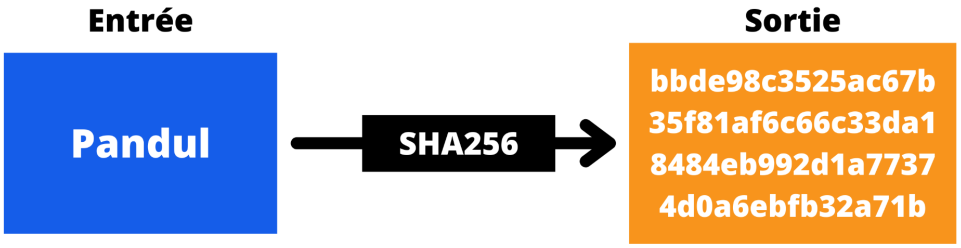
La première famille d'algorithmes cryptographiques utilisée sur Bitcoin regroupe les fonctions de hachage. Elles sont utilisées à de nombreuses reprises dans le protocole.

Le hachage est un procédé permettant de chiffrer une information grâce à une fonction de hachage. Cette fonction va prendre en input (entrée) une information de taille arbitraire et va la convertir en une empreinte de taille fixe appelée "hash" en output (sortie).

⇨ Ce hash est parfois également nommé : "digest", "condensat", "condensé" ou encore "haché".

Par exemple, la fonction SHA256 sortira toujours un hash d'une taille de 256 bits en sortie.

Ainsi, si je mets en entrée `Pandul`, un message de taille arbitraire, le hash en sortie sera alors : `bbde98…`, une empreinte de 256 bits.

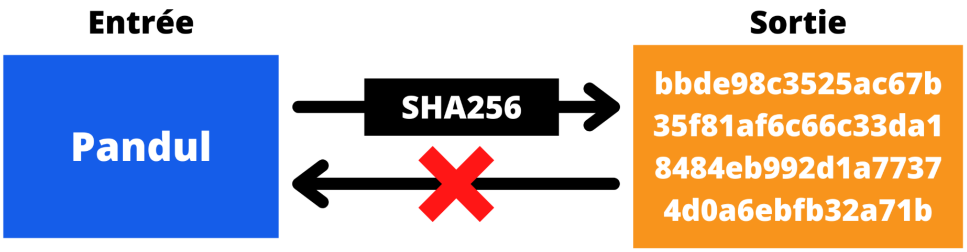


Ces fonctions de hachage sont utilisées dans Bitcoin car elles disposent de 3 caractéristiques intéressantes :

- La première caractéristique d'une fonction de hachage est l'irréversibilité.

Cela veut dire que le calcul du hash sachant l'information en entrée peut être réalisé facilement. En revanche le calcul inverse, c'est-à-dire le calcul de l'information en entrée sachant le hash doit être impossible.

C'est ce que l'on appelle une fonction irréversible, fonction à sens unique ou encore "*trap door function*".

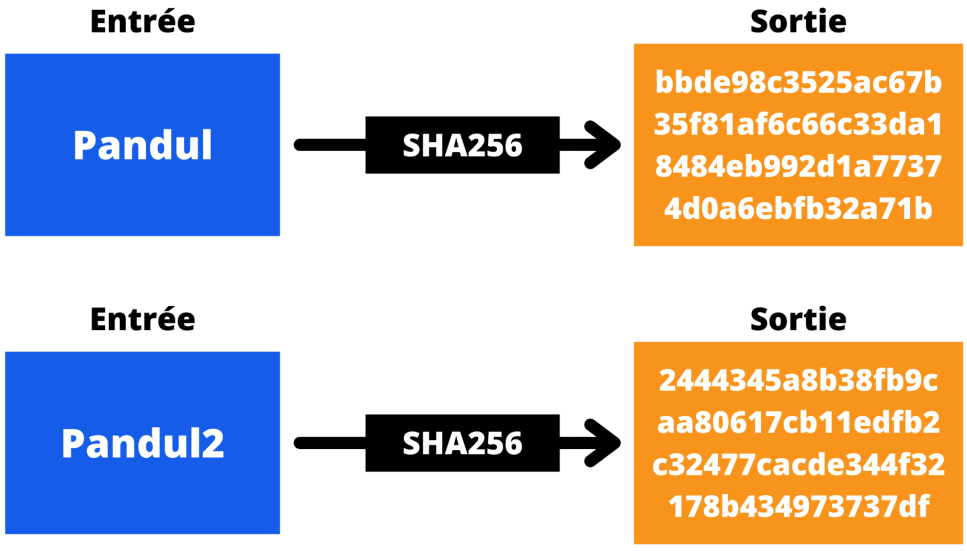


Dans notre exemple, obtenir le hash `bbde98...` en connaissant le message en entrée `Pandul` est très facile. En revanche, il sera impossible de trouver `Pandul` en ayant connaissance uniquement du hash `bbde98...`.

- La deuxième caractéristique d'une fonction de hachage est la résistance à la falsification.

Cette caractéristique est vérifiée uniquement si la moindre petite modification sur le message en entrée résulte sur un hash profondément différent en sortie.

Ainsi, si nous reprenons notre exemple, nous pouvons voir que le simple ajout d'un chiffre sur le message en entrée modifie complètement le hash en sortie :



Grâce à cette spécificité, les fonctions de hachage sont utilisées pour vérifier l'authenticité d'une information et pour détecter la moindre petite modification de l'information originelle.

- Enfin la troisième caractéristique des fonctions de hachage est la résistance à la collision.

Une collision se produit lorsqu'un couple de données distinctes en entrée d'une fonction de hachage donne exactement le même hash en sortie.

Cette situation de collision est mathématiquement impossible à éviter au vu de la limitation de la taille de la sortie et de la différence de taille entre les entrées et les sorties.

Ainsi, une bonne fonction de hachage sera une fonction pour laquelle le risque de collision est tellement faible qu'il peut être considéré comme nul.

On admet donc que pour deux valeurs distinctes en entrée de la fonction, il est impossible d'obtenir exactement le même hash en sortie.

Ces trois caractéristiques permettent de déterminer de nombreux cas d'usages pour les fonctions de hachage.

Les fonctions de hachage dans Bitcoin.

La fonction de hachage la plus souvent utilisée sur le protocole Bitcoin est **SHA256** (*Secure Hash Algorithm*). C'est une fonction de hachage qui produit en sortie un condensat de 256 bits.

Conçue au début des années 2000 par la NSA, elle est aujourd'hui un standard fédéral de traitement des données aux États-Unis.

Elle est utilisée au sein du protocole Bitcoin notamment pour :

- Obtenir le hash de l'entête d'un bloc candidat. Hash qui sera ensuite comparé avec la cible de difficulté afin de valider un bloc.

- Obtenir la racine de Merkle en agrégeant les transactions au sein d'un arbre de Merkle. Dans l'arbre de Merkle on utilise un double **SHA256** également nommé **HASH256**.

- Obtenir un hash d'une clé publique afin d'en faire une adresse. Dans ce cas, il sera appliqué une deuxième fonction de hachage sur **SHA256** appelée **RIPEMD160** (*RACE Integrity Primitives Evaluation Message Digest*), qui donne un condensat de 160 bits.
  L'utilisation de **RIMPEMD160** + **SHA256** est également nommée **HASH160**.

- Obtenir un hash dont les premiers bits serviront de checksum (somme de contrôle) à partir de l'entropie afin de calculer la phrase mnémonique.

Je reviendrai évidemment sur toutes ces utilisations en détail dans les tomes suivants.

On retrouve également dans Bitcoin sa variante SHA512 qui est notamment utilisée dans le processus de dérivation d'un portefeuille HD.

Les rouages de SHA256.

Nous avons vu précédemment que les fonctions de hachages disposent de caractéristiques intéressantes qui justifient un usage au sein du protocole Bitcoin. Voyons maintenant ensemble les rouages de ces fonctions de hachage.

Les fonctions SHA256 et SHA512 appartiennent à la famille des *Secure Hash Algorithm*, et plus précisément à la famille des SHA-2. Le mécanisme de ces deux fonctions de hachage est similaire. J'expliquerai donc simplement ici le fonctionnement de SHA256.

Pour rappel, nous disposons d'un message de taille arbitraire en entrée. L'objectif est de le passer dans la fonction SHA256 afin de disposer d'un hash de 256 bits en sortie. Le processus se décompose en plusieurs étapes :

1- Le rembourrage (pré-traitement).

Pour commencer, il va falloir égaliser notre message en entrée afin qu'il dispose d'une taille d'un multiple de 512 bits.

Le rembourrage consiste à ajouter des bits à notre message en input afin qu'il atteigne premièrement une taille égale au multiple de 512 bits supérieur le plus proche, moins 64 bits.

On a donc :

```
M + P + 64 = n*512
```

où :

- `M` est notre message en input de taille arbitraire.
- `P` est notre rembourrage.
- `n*512` est le premier multiple de 512 bits supérieur à `M+64`.

Le rembourrage commence toujours par un `1` puis le nombre de `0` nécessaires pour arriver à `n*512-64`.

Par exemple, si le message en entrée de SHA256 fait 950 bits, nous aurons :

```
Le premier multiples de 512 supérieur à M+64 est 1024
(2 * 512).

→ M + P + 64 = 1024
→ 950 + P = 1024 - 64
→ P = 1024 - 64 - 950
→ P = 10
```

Dans cet exemple, notre rembourrage devra donc être de 10 bits. Comme expliqué, on commence par un `1`, puis on complète avec le nombre de `0` nécessaires. Ici cela nous donne un rembourrage égal à : `1000000000`.

2- Le modulo.

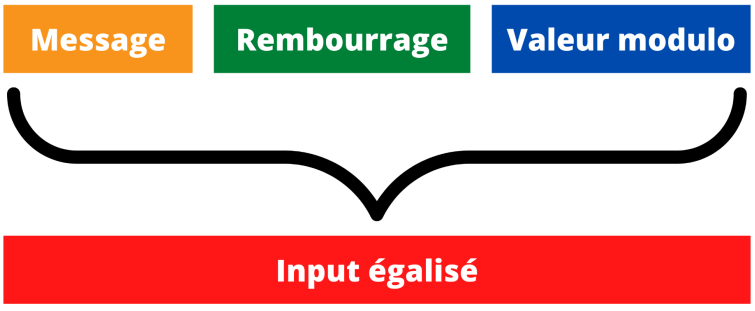
Pour terminer d'égaliser notre input, nous allons ajouter les 64 bits manquants afin d'atteindre une taille totale égale à un multiple de 512 bits.

Pour ce faire, nous allons calculer le modulo du message initial avec 2^{32}.

```
Valeur modulo = M mod 2^32
```

⇨ "*mod*", pour modulo, est simplement une opération mathématique qui permet, entre deux nombres entiers, de renvoyer le reste de la division euclidienne du premier par le deuxième nombre. Par exemple, `16 mod 5` sera égal à `1`.

Puis nous ajoutons ces 64 bits au résultat de l'étape 1 pour arriver à notre input égalisé sur un multiple de 512 bits.



3- Les variables.

La fonction SHA256 est configurée avec des valeurs de 32 bits par défaut dont nous aurons besoin à l'étape suivante.

Tout d'abord, il existe 8 variables associées aux lettres `A à H` telles que :

```
A = 0x6a09e667
B = 0xbb67ae85
C = 0x3c6ef372
D = 0xa54ff53a
E = 0x510e527f
F = 0x9b05688c
G = 0x1f83d9ab
H = 0x5be0cd19
```

Et il existe 64 constantes qui agiront comme clé dans le processus :

```
K[0..63] =
0x428a2f98,     0x71374491,     0xb5c0fbcf,
0xe9b5dba5,     0x3956c25b,     0x59f111f1,
0x923f82a4,     0xab1c5ed5,     0xd807aa98,
0x12835b01,     0x243185be,     0x550c7dc3,
0x72be5d74,     0x80deb1fe,     0x9bdc06a7,
0xc19bf174,     0xe49b69c1,     0xefbe4786,
0x0fc19dc6,     0x240ca1cc,     0x2de92c6f,
0x4a7484aa,     0x5cb0a9dc,     0x76f988da,
0x983e5152,     0xa831c66d,     0xb00327c8,
0xbf597fc7,     0xc6e00bf3,     0xd5a79147,
0x06ca6351,     0x14292967,     0x27b70a85,
0x2e1b2138,     0x4d2c6dfc,     0x53380d13,
0x650a7354,     0x766a0abb,     0x81c2c92e,
0x92722c85,     0xa2bfe8a1,     0xa81a664b,
0xc24b8b70,     0xc76c51a3,     0xd192e819,
0xd6990624,     0xf40e3585,     0x106aa070,
0x19a4c116,     0x1e376c08,     0x2748774c,
0x34b0bcb5,     0x391c0cb3,     0x4ed8aa4a,
0x5b9cca4f,     0x682e6ff3,     0x748f82ee,
0x78a5636f,     0x84c87814,     0x8cc70208,
0x90befffa,     0xa4506ceb,     0xbef9a3f7,
0xc67178f2
```

Toutes ces informations seront utiles à l'étape suivante.

4- La compression.

On va maintenant pouvoir s'attaquer aux calculs. Pour commencer, nous allons diviser notre input égalisé (résultat de l'étape 2) en morceaux de 512 bits chacun. Etant donné que notre input égalisé est d'une taille de `n * 512` bits, alors nous divisons notre input égalisé en `n` morceaux, chacun d'une taille de 512 bits. Chaque morceau va ensuite passer dans la fonction de compression. Cette fonction consiste à effectuer 64 opérations sur le morceau.

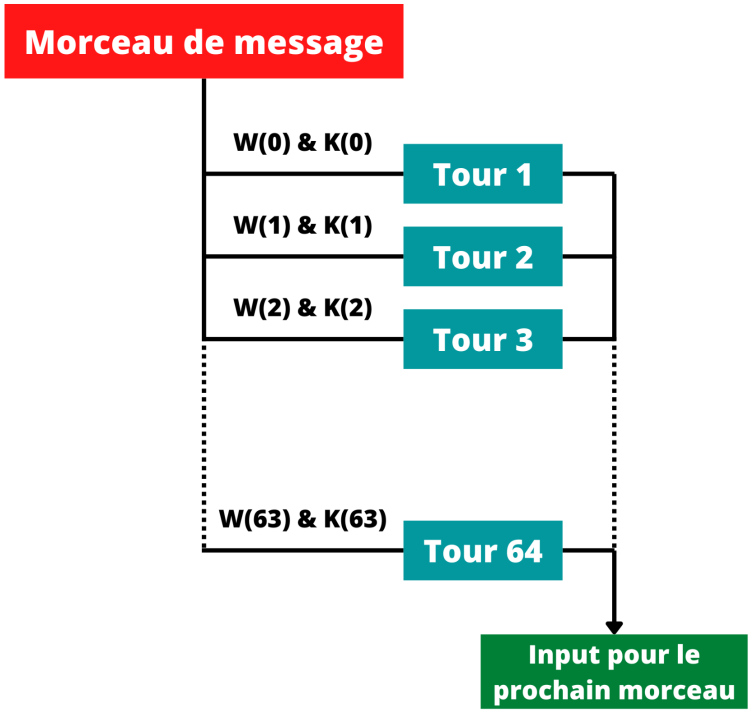
Ces opérations sont basées sur l'algèbre de Boole. Ce sont des opérations logiques qui peuvent être décrites par trois opération de base :

- La disjonction (*OR*).
- La conjonction (*AND*).
- La négation (*NOT*).

A partir de ces opérations logiques de base, nous allons pouvoir déterminer des fonctions plus complexes comme le *XOR* (*OU* exclusif), une opération très utilisée en cryptographie.

Je ne vais pas plus détailler les mécanismes de l'algèbre de Boole dans cet ebook car cela s'éloigne trop de notre thème. Retenez simplement que ce sont des opérations mathématiques, qui à partir de deux informations en entrée, nous donneront un résultat en sortie.

Schématiquement, la fonction de compression ressemblera à cela :



Dans ce schéma, nous pouvons voir que nous avons pour chaque tour 3 inputs : `W(i)`, `K(i)` et une suite que nous verrons plus tard.

`W(i)` est un nombre de 32 bits. Les 16 premiers `W(i)`, c'est-à-dire `W(0)`, `W(1)`, `W(2)`, [...], `W(15)`, représentent notre morceau de message. Pour rappel notre morceau fait 512 bits, ses 32 premiers bits seront `W(0)`, les 32 bits suivants seront `W(1)`, les 32 bits suivants `W(2)`... Jusqu'à `W(15)` qui prendra les 32 derniers bits de notre morceau de message.

Pour déterminer les `W(i)` après `W(15)`, c'est-à-dire `W(16)`, `W(17)`, [...], `W(63)`, une formule existe :

```
W(i) = W(i-16) + σ⁰ + W(i-7) + σ¹

où :

σ⁰ = (W(i-15) RotR 7) ⊕ (W(i-15) RotR 18) ⊕ (W(i-15) ShR 3)

σ¹ = (W(i-2) RotR 17) ⊕ (W(i-2) RotR 19) ⊕ (W(i-2) ShR 10)

RotR : Décalage circulaire à droite.

ShR : Décalage à droite.

⊕ : XOR
```

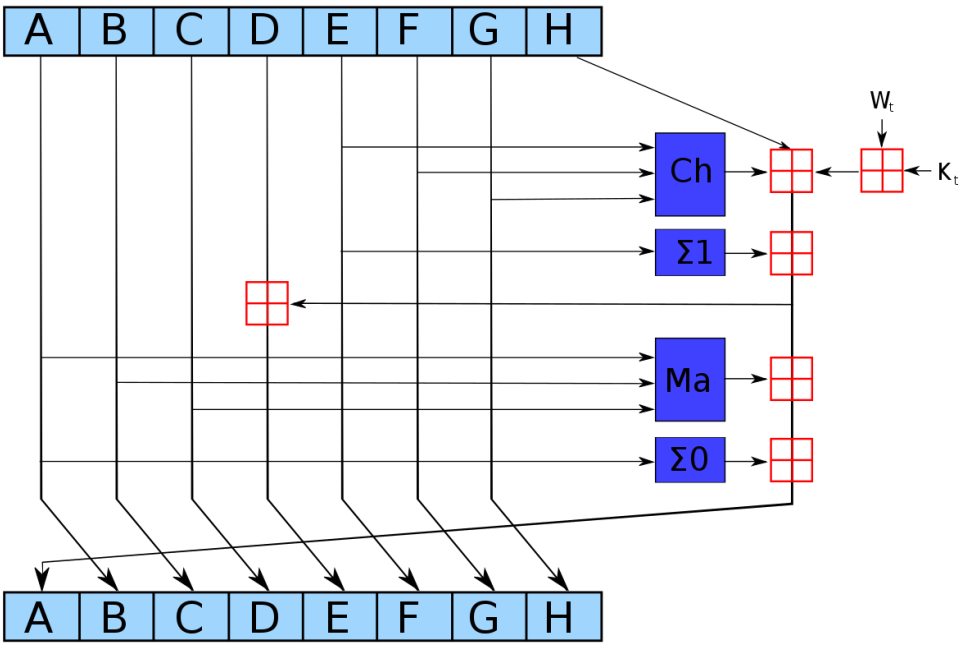
On sait donc maintenant comment déterminer chaque entrée `W(i)`.

Les entrées `K(i)`, c'est-à-dire `K(0)`, `K(1)`, ..., `K(63)`, correspondent simplement aux 64 constantes utilisées comme clé, décrites à l'étape 3.

Enfin, la troisième entrée de notre fonction de compression sera la suite `[A,B,C,D,E,F,G,H]` décrite à l'étape 3, où chaque lettre correspond à un nombre de 32 bits. Cette suite de variables sera utilisée en input pour le premier tour seulement.

L'output du premier tour donnera une autre suite `[A,B,C,D,E,F,G,H]`. Cette nouvelle suite sera utilisée en input pour le deuxième tour. L'output du deuxième tour deviendra l'input du troisième tour, et ainsi de suite.

Maintenant que nous disposons de nos 3 inputs, regardons plus en détail comment fonctionnent ces 64 tours :



Crédit : https://commons.wikimedia.org/wiki/User:Kockmeyer.

Ce schéma représente un tour de notre fonction de compression. On peut y reconnaître nos inputs :

- La suite `[A,B,C,D,E,F,G,H]`.
- W_t que nous avions nommé `W(i)`.
- K_t que nous avions nommé `K(i)`.

On peut déjà observer que ce tour nous donne en sortie une nouvelle suite `[A,B,C,D,E,F,G,H]`. Comme expliqué précédemment, cette nouvelle suite servira d'entrée pour le tour suivant.

Le symbole **[+]** correspond simplement à une addition *modulo* 2^{32}.

Les autres calculs se décomposent comme cela :

```
Ch(E, F, G) = (E AND F) ⊕ ((NOT E) AND G)

Maj(A, B, C) = (A AND B) ⊕ (A AND C) ⊕ (B AND C)

Σ0(A) = RotR(A,2) ⊕ RotR(A,13) ⊕ RotR(A,22)

Σ1(E) = RotR(E,6) ⊕ RotR(E,11) ⊕ RotR(E,25)

⊕ : XOR
```

L'objectif d'un tour est donc d'obtenir en sortie une nouvelle suite `[A,B,C,D,E,F,G,H]`, où chaque lettre est un nouveau nombre de 32 bits.

Voici en détail comment obtenir notre sortie $[A_S, B_S, C_S, D_S, E_S, F_S, G_S, H_S]$ à partir de notre entrée $[A_E, B_E, C_E, D_E, E_E, F_E, G_E, H_E]$ et des 2 autres inputs connus : `W(i)` et `K(i)` :

La couleur orange permet de mettre en évidence la suite de sortie et la couleur bleu indique la suite d'entrée.

Comme expliqué précédemment, le signe `+` symbolise une addition modulo 2^{32}.

```
Pour i de 0 à 63 :

Σ1 = (E_E RotR 6) ⊕ (E_E RotR 11) ⊕ (E_E RotR 25)
Ch = (E_E AND F_E) ⊕ ((NOT E_E) AND G_E)
temp1 = H_E + Σ1 + Ch + K(i) + W(i)
Σ0 = (A_E RotR 2) ⊕ (A_E RotR 13) ⊕ (A_E RotR 22)
Maj = (A_E AND B_E) ⊕ (A_E AND C_E) ⊕ (B_E AND C_E)
temp2 = Σ0 + Maj
```

```
A_S = temp1 + temp2
B_S = A_E
C_S = B_E
D_S = C_E
E_S = D_E + temp1
F_S = E_E
G_S = F_E
H_S = G_E
```

Voici comment se décompose un tour.

Une fois que l'on a l'output de ce tour (la nouvelle suite de `[A,B,C,D,E,F,G,H]`), on utilise cet output comme input pour le tour suivant. On continue ainsi de suite jusqu'à arriver au 64ème tour.

L'output du 64ème tour servira d'input pour le premier tour du second morceau de 512 bits. On continue ainsi de suite jusqu'à avoir traité tous les morceaux de 512 bits de notre message initial.

La sortie finale de notre fonction de hachage SHA256 sera l'output du 64ème tour du dernier morceau.

Dans cette suite `[A,B,C,D,E,F,G,H]`, chaque lettre fait 32 bits. Nous aurons donc bien une sortie finale de la fonction de hachage d'une taille fixe de 256 bits.

Les algorithmes de dérivation dans Bitcoin.

Sur le protocole Bitcoin sont également utilisés des algorithmes de dérivation basés sur des fonctions de hachages et donc assimilables à ces dernières.

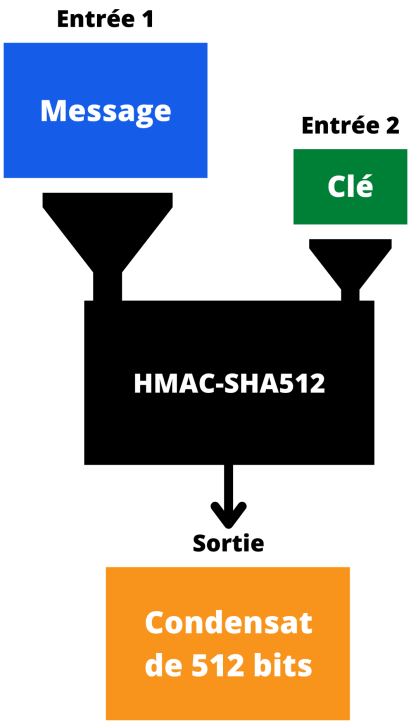
La principale différence entre une fonction de hachage et un algorithme de dérivation est le nombre d'entrées et le modèle de sécurité.

Étant donné que ces algorithmes sont basés sur une fonction de hachage, ils conservent les mêmes caractéristiques que cette dernière à savoir : l'irréversibilité, la résistance à la falsification et la résistance à la collision.

On en utilise deux sur le protocole Bitcoin : PBKDF2 (*Password Based Key Derivation Function 2*) et HMAC (*Hash-based Message Authentication Code*).

HMAC est un algorithme cryptographique permettant de calculer un code d'authentification en utilisant une combinaison d'une fonction de hachage et d'une clé secrète. C'est une fonction dérivée de l'algorithme NMAC.

Sur Bitcoin on utilise HMAC-SHA512, c'est-à-dire l'algorithme HMAC intégrant la fonction de hachage SHA512 (qui est similaire à SHA256 mais qui donne une sortie de 512 bits). Voici son schéma de fonctionnement :



Maintenant, étudions plus en détail ce qu'il se passe dans cette boîte noire HMAC-SHA512.

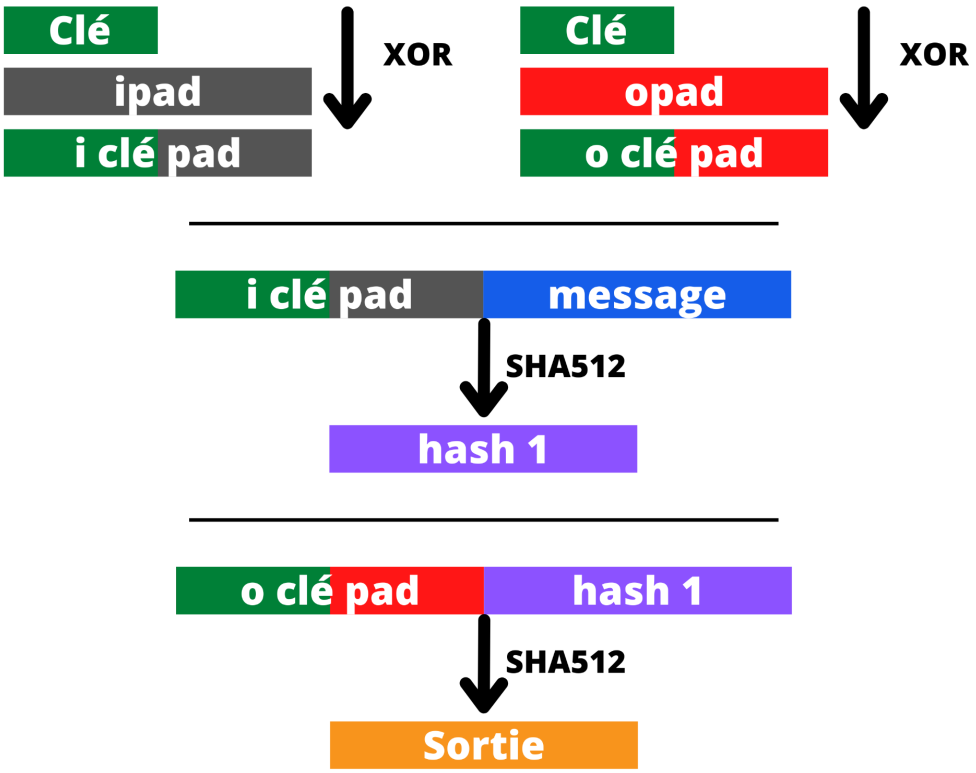
Soit la fonction HMAC-SHA512 avec :

- `K` = clé arbitraire choisie par l'utilisateur (entrée 2).
- `m` = message de taille arbitraire choisi par l'utilisateur (entrée 1).
- `SHA512` : La fonction de hachage SHA512.
- `⊕` (*XOR*) : Fonction "**OU exclusif**" basée sur les opérations Booléennes qui permet à partir de deux opérandes qui peuvent avoir chacun la valeur de "vrai" ou de "faux", de donner un résultat qui a lui-même la valeur "vrai" uniquement si les deux opérandes ont des valeurs différentes.
- `||` : Opération de **concaténation**, c'est-à-dire de mettre les deux opérandes l'une à la suite de l'autre.
- *opad* et *ipad* : deux constantes décidées arbitrairement par l'auteur de la fonction. La seule nécessité sera que ces deux valeurs soient différentes.

Définie par l'équation :

$$HMAC\text{-}SHA512_K(m) = SHA512\ ((K \oplus opad)\ ||\ SHA512((K \oplus ipad)\ ||\ m))$$

Schématiquement, cela représenterait cela :



Voici les étapes décomposées de cette fonction :

- On *XOR* la `clé` (entrée 2) avec `ipad` ce qui nous donne `i clé pad`.
- On *XOR* la `clé` (entrée 2) avec `opad` ce qui nous donne `o clé pad`.
- On concatène `i clé pad` avec le `message` (entrée 1).
- On hache ce résultat avec `SHA512` ce qui nous donne `hash 1`.
- On concatène `o clé pad` avec `hash 1`.
- On hache ce résultat avec `SHA512` ce qui nous donne le `hash de sortie`.

Cette fonction **HMAC** est utilisée sur Bitcoin à la fois dans les chemins de dérivation de clés sur les portefeuilles HD, et également au sein d'une autre fonction nommée **PBKDF2**.

PBKDF2, pour *Password-Based Key Derivation Function 2*, est une fonction de dérivation de clé.

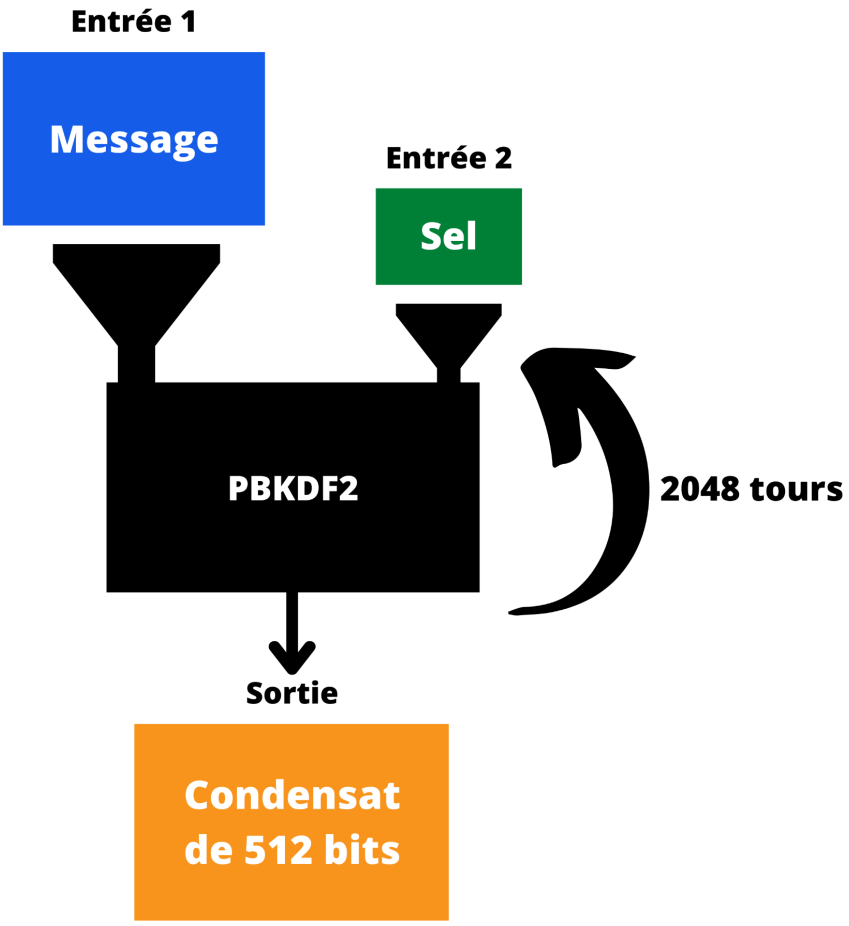
Cet algorithme va simplement appliquer une fonction choisie par l'utilisateur à un message de taille arbitraire avec un sel cryptographique, et il va répéter cette opération un certain nombre de fois pour sortir une clé comparable à un condensat.

Étant donné que cet algorithme utilise une fonction de hachage en son sein, il dispose des mêmes caractéristiques que cette dernière : irréversibilité, résistance à la falsification et résistance à la collision.

Dans le protocole Bitcoin, on utilise PBKDF2 pour dériver la graine d'un portefeuille HD (seed) depuis la phrase mnémonique (ou phrase de récupération).

La fonction pseudo-aléatoire utilisée est alors HMAC-SHA512, le sel représente la passphrase et le nombre d'itérations est de 2048.

Schématiquement, PBKDF2 représenterait cela :



Nous avons donc pu découvrir les fonctions de hachage utilisées sur Bitcoin.

Tout cela vous paraît peut-être encore flou, mais ne vous inquiétez pas, nous verrons leurs utilisations concrètes sur le protocole dès le **tome 2**.

Étant donné que ces fonctions sont utilisées à toutes les étapes du protocole Bitcoin, j'ai souhaité les décrire tout de suite dès le premier tome, afin de ne pas avoir à y revenir dans chaque tome suivant.

Dans la deuxième partie, nous allons étudier et vulgariser la deuxième grande catégorie d'algorithmes cryptographiques utilisés sur Bitcoin : Les signatures numériques.

2- Les signatures numériques.

La deuxième application de la cryptographie dans Bitcoin sont les algorithmes de signatures numériques à clé publique. Voyons ensemble en quoi cela consiste et comment cela fonctionne.

Le mot “portefeuille” (wallet) sur Bitcoin a été assez mal choisi selon moi. En effet, ce que l’on appelle “portefeuille” Bitcoin est un logiciel qui ne conserve pas directement vos bitcoins, contrairement à un portefeuille classique qui permet de conserver des pièces.

⇨ Pour rappel, “Bitcoin” avec un “B” majuscule désigne le système de paiement électronique pair à pair, le protocole ou le réseau. “bitcoin” avec un “b” minuscule désigne l’unité de compte de ce système.

Les bitcoins sont simplement des unités de compte natives du réseau de paiement homographe. Cette unité de compte est représentée par des UTXO (*Unspent Transaction Output*), qui sont simplement des sorties de transactions pas encore dépensées. Si ces sorties ne sont pas dépensées, cela veut dire par déduction qu’elles appartiennent à un utilisateur. Les UTXO sont en quelque sorte des fractions de bitcoins, d’une taille variable, appartenant à un utilisateur.

Le protocole Bitcoin est distribué, il fonctionne sans autorité centrale. On ne peut donc pas faire comme sur les livres de comptes bancaires, où les euros qui vous appartiennent sont simplement associés à votre identité personnelle. Sur le réseau Bitcoin, on associe donc la propriété des UTXO (morceaux de bitcoins) à une clé publique, elle-même liée mathématiquement à une clé privée.

Comme leurs noms l’indiquent, la clé publique est connue de tous et la clé privée est uniquement connue par le propriétaire des fonds.

Pour pouvoir dépenser les bitcoins associés à une clé publique, un utilisateur va devoir prouver au reste du réseau qu’il en est bien le propriétaire légitime. Il va donc devoir établir une preuve irréfutable qu’il connaît la clé privée associée à cette clé publique, sans dévoiler ladite clé privée.

Pour ce faire, un utilisateur qui initie une transaction devra établir une signature numérique à l’aide de sa clé privée sur la transaction en question.

La signature pourra être vérifiée par les autres parties prenantes au réseau. Si elle est valide, cela veut dire que l'utilisateur qui initie la transaction est bien le propriétaire de la clé privée, et donc qu'il est bien le propriétaire des bitcoins qu'il souhaite dépenser. Les autres utilisateurs pourront alors exécuter la transaction.

En conséquence, un utilisateur qui possède des bitcoins sur une clé publique devra trouver un moyen de stocker de manière sécurisée ce qui permet de débloquer ses fonds : la clé privée. Un portefeuille Bitcoin est un dispositif qui va vous permettre de conserver facilement toutes vos clés sans que d'autres personnes n'y aient accès. Cela ressemble donc plus à un porte-clés qu'à un portefeuille.

Le lien mathématique évoqué précédemment entre une clé publique et une clé privée, et la possibilité de réaliser une signature pour prouver la possession d'une clé privée sans la dévoiler, sont rendus possible par un algorithme de signature numérique.

Dans le protocole Bitcoin, nous utilisons deux algorithmes de signature : **ECDSA** pour *Elliptic curve Digital Signature Algorithm*, et le Protocole de **Schnorr**, du nom de son inventeur.

ECDSA est le protocole de signatures numériques utilisé sur Bitcoin depuis ses débuts. Schnorr est tout nouveau sur Bitcoin, puisqu'il a été introduit en Novembre 2021 avec la mise à jour Taproot.

Ces deux algorithmes sont très similaires. Ils sont tous deux basés sur la cryptographie sur les courbes elliptiques. La différence majeure entre ces deux protocoles réside dans la signature : les signatures de Schnorr sont dites "linéaires", c'est-à-dire qu'elles supportent nativement les multi-signatures contrairement aux signatures ECDSA.

Étant donné leurs similitudes techniques, nous allons ici étudier simplement l'algorithme de signature ECDSA. La majorité des concepts évoqués pourront également être vérifiés sur le protocole de Schnorr.

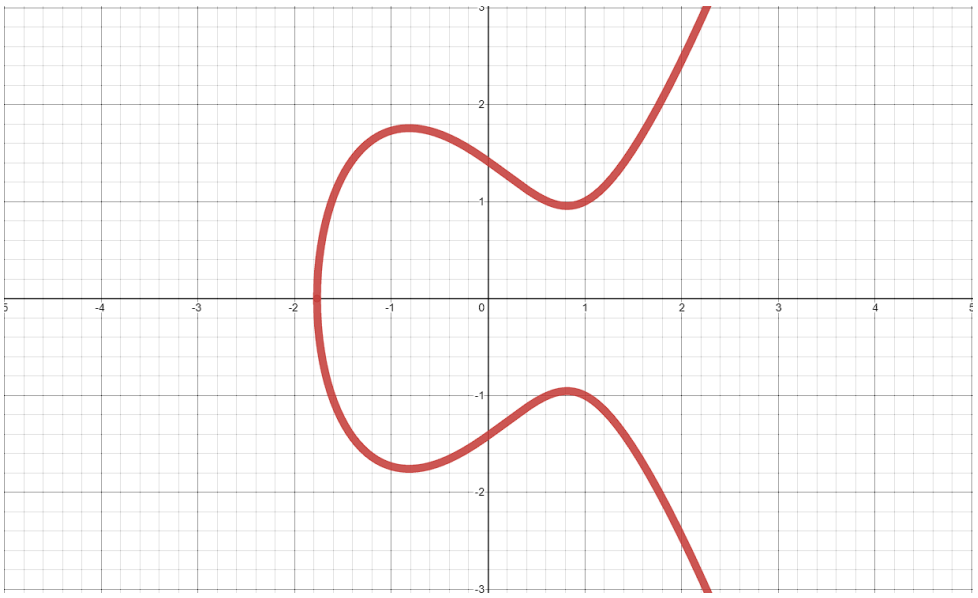
La courbe elliptique.

La cryptographie sur les courbes elliptiques est une famille d'algorithmes de cryptographie asymétrique dont la sécurité se base sur la difficulté du calcul du logarithme discret.

Ces algorithmes utilisent une courbe elliptique pour ses différentes propriétés mathématiques et géométriques. Notamment, ces courbes seront toujours symétriques. Ainsi, toute droite non verticale coupant 2 points sur une courbe elliptique coupera toujours la courbe en un troisième point.

Aussi, toute ligne non verticale et tangente à la courbe en un point coupera toujours la courbe en un deuxième point unique. Ces propriétés nous seront utiles pour la suite.

Voici une représentation d'une courbe elliptique :



Toute courbe elliptique est définie par l'équation : $y^2 = x^3 + ax + b$.

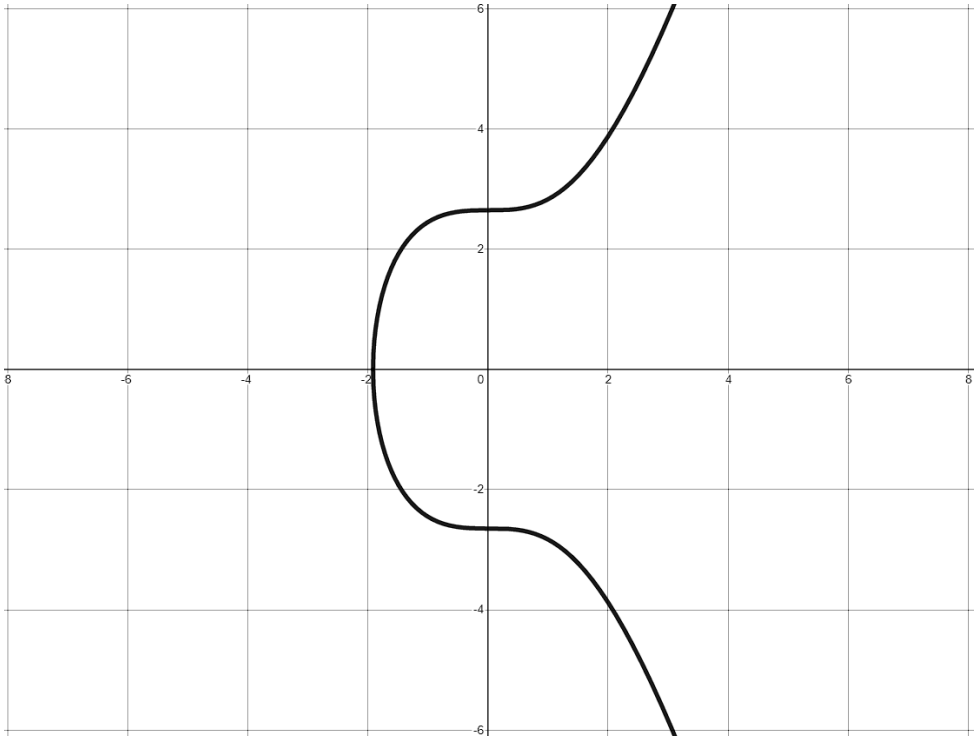
Pour utiliser ECDSA, il faudra donc choisir les paramètres de la courbe elliptique, c'est-à-dire les valeurs de `a` et de `b` dans l'équation.

Il existe ainsi différents standards de courbes elliptiques réputées sûres. La plus connue est la courbe *secp256r1*, définie et conseillée par la NSA. Satoshi Nakamoto, l'inventeur de Bitcoin, a choisi de ne pas utiliser cette courbe. La raison est inconnue, mais certains pensent qu'il a fait ce choix car les paramètres de cette courbe contiennent potentiellement une backdoor.

A la place, le protocole Bitcoin utilise le standard ***secp256k1***. *"SEC"* désigne *"Standards for Efficient Cryptography"*. *"P256"* désigne le fait que la courbe est définie sur un corps $\mathbb{Z}_p$ où p est un nombre premier de 256 bits. *"K"* désigne le nom de son inventeur Neal Koblitz et *"1"* désigne la première version.

Secp256k1 est une courbe définie par les paramètres `a=0` et `b=7`. Son équation est donc : $y^2 = x^3 + 7$.

Sa représentation graphique sur le corps des réels ressemblera à cela :



En réalité, nous ne travaillerons pas sur le corps des réels mais sur le corps $\mathbb{Z}_p$ qui est un corps fini d'entiers positifs *modulo* p, où p est un nombre premier.

⇨ Un nombre premier est simplement un entier naturel qui n'admet que deux diviseurs entiers et positifs : **1 et lui-même**. Par exemple : le chiffre 7 est un nombre premier puisqu'il ne peut être divisé que par deux entiers naturels : 1 et 7.
En revanche, le chiffre 8 n'est pas un nombre premier étant donné qu'il peut être divisé par 1, 2, 4 et 8, il n'admet donc pas que deux diviseurs entiers et positifs mais quatre diviseurs, ce qui l'exclut du groupe des nombres premiers.

La définition de ce corps fini d'ordre premier va nous permettre de travailler exclusivement à partir d'entiers compris dans un rang fini. Sur Bitcoin, ce rang est de 256 bits soit $\mathbf{2^{256}}$.

$\mathbf{2^{256}}$ n'est pas un nombre premier, ECDSA utilise donc un nombre premier inférieur et relativement proche de ce nombre. Notons que ce n'est pas le plus proche existant mais qu'il a été choisi pour différents facteurs.

Ce nombre premier est en hexadécimal :

```
p = FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFE FFFFFC2F
```

En format décimal, nous aurons :

$$p = 2^{256} - 2^{32} - 977$$

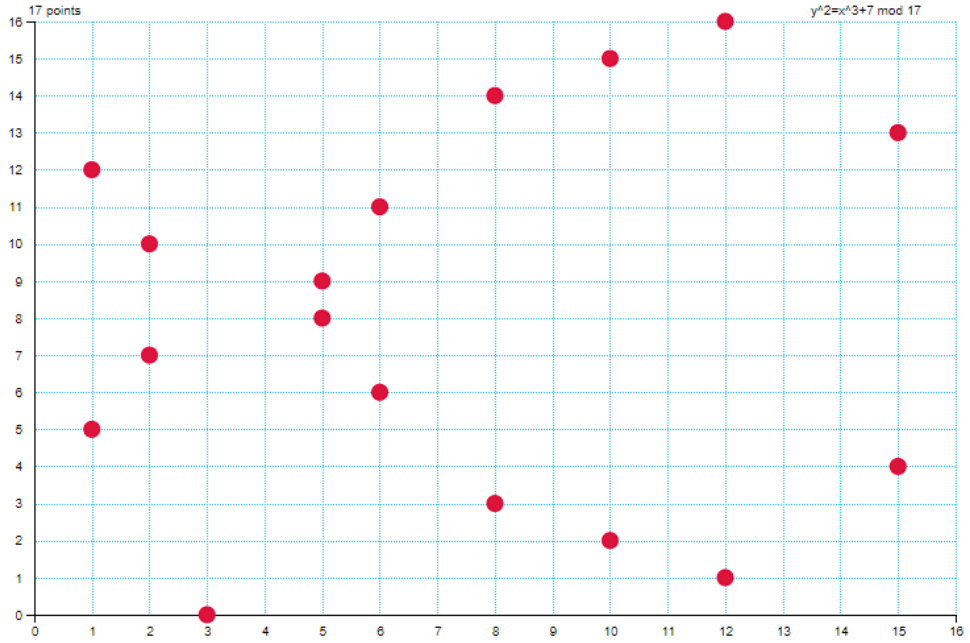
Cela correspond à un nombre long de 78 chiffres dont je vous épargne la lecture.

L'équation de notre courbe elliptique sera donc en réalité :

$$y^2 = (x^3 + 7) \mod (2^{256} - 2^{32} - 977)$$

Étant donné que cette courbe est définie sur le corps $\mathbb{Z}p$, elle ne ressemblera plus vraiment à une courbe mais plutôt à un nuage de points.

Par exemple, voici à quoi ressemble la courbe utilisée dans Bitcoin pour $p = 17$:



Crédit : https://www.graui.de/ Dr. Sascha Grau.

Ici, j'ai intentionnellement limité le corps à *modulo* 17, mais il faut imaginer que celui utilisé sur Bitcoin est infiniment plus grand.

L'utilisation de ce corps fini d'ordre premier est régi par les mêmes mathématiques que l'utilisation d'une courbe sur le corps des réels. Dans un souci de simplification, je vais donc continuer la vulgarisation sur une courbe définie sur des nombres réels.

Néanmoins, vous pouvez garder à l'esprit que cette courbe est en réalité un nuage de points comme sur le schéma ci-dessus.

La clé privée.

Comme vu précédemment, l'algorithme ECDSA est basé sur un couple clé privée / clé publique qui sont liées mathématiquement.

La clé privée sera simplement un nombre pseudo-aléatoire. Dans le cas de Bitcoin, ce nombre pseudo-aléatoire sera de 256 bits.

> ⇨ Un nombre pseudo-aléatoire est un nombre qui dispose de propriétés s'approchant des propriétés idéales d'un nombre aléatoire.

Le nombre de possibilités pour une clé privée Bitcoin est donc théoriquement de $\mathbf{2^{256}}$ possibilités.

En réalité, il existe seulement `n` points sur notre courbe elliptique. Nous verrons plus tard à quoi correspond ce nombre, mais retenez simplement qu'une clé privée valide sera un nombre de 256 bits compris entre `1` et `n−1`, en sachant que `n` est un nombre légèrement plus petit que $\mathbf{2^{256}}$. Il existe donc certains nombres de 256 bits qui ne sont pas valides pour devenir clé privée sur Bitcoin.

Si jamais l'entropie mène à une clé privée entre `n` et 2^{256}, dans ce cas la clé privée sera invalide et il faudra réaliser une autre entropie.

Le nombre de possibilités pour une clé privée Bitcoin est donc presque de $1,158 * 10^{77}$.

C'est un nombre tellement grand que si vous choisissez une clé privée aléatoirement, il est statistiquement presque impossible de tomber sur la clé privée d'un autre utilisateur. Pour vous donner un ordre de grandeur, il y a presque autant de clés privées possibles sur Bitcoin que d'atomes dans l'univers observable.

Comme nous le verrons dans le **tome 2** de la série ***Bitcoin Démocratisé***, aujourd'hui, la majorité des clés privées sur Bitcoin ne sont pas générées aléatoirement mais sont le résultat d'une dérivation déterministe depuis une phrase mnémonique, elle-même pseudo-aléatoire. Cette information ne change rien pour ECDSA, mais elle permet de recentrer ma vulgarisation sur Bitcoin.

Pour la suite de la vulgarisation, la clé privée sera appelée "`k`" minuscule.

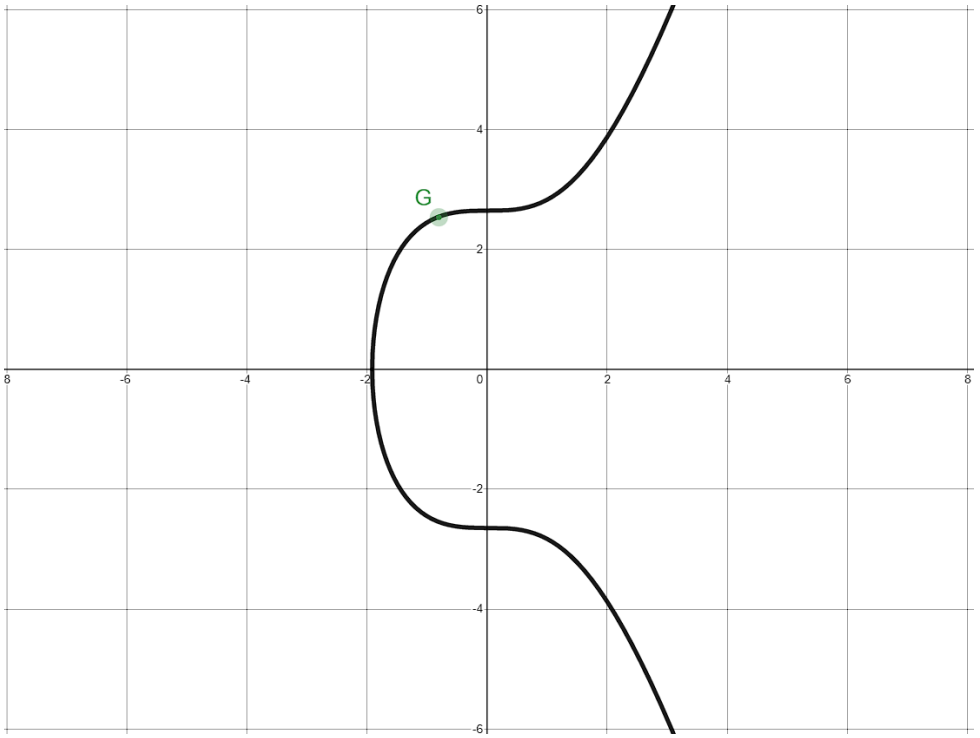
La clé publique.

La clé publique est un nombre de 520 bits qui est généré à partir de sa clé privée. Ce nombre sera un point sur notre fameuse courbe elliptique que nous nommerons "`K`" majuscule.

> ⇨ Comme nous le verrons dans le **tome 2**, une clé publique de 520 bits peut être compressée en un nombre de 264 bits conservant pourtant les mêmes informations. C'est que l'on appelle une clé publique compressée.

Pour calculer `K` nous allons utiliser la multiplication sur les courbes elliptiques tel que : `K = k * G`, où `k` est la clé privée et `G` est le point générateur également parfois appelé point d'origine. C'est un point sur la courbe elliptique, connu par tous les utilisateurs, et utilisé pour calculer toutes les clés publiques.

Le fait que ce point `G` soit commun à toutes les clés publiques Bitcoin nous permet d'être sûr qu'une même clé privée nous donnera toujours la même clé publique.

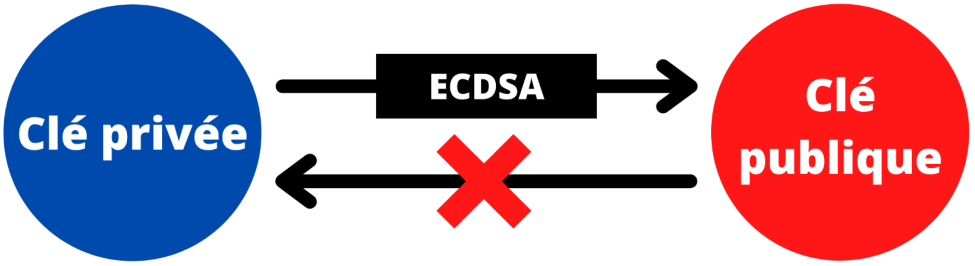


La principale caractéristique de la multiplication sur les courbes elliptiques est qu'elle n'est pas réversible, c'est une "*trap door function*". En conséquence, il est très facile de calculer une clé publique en sachant sa clé privée et le point générateur, mais il est impossible de calculer une clé privée et sachant sa clé publique et le point générateur.

Ainsi, il est facile de calculer `K = k * G` mais il est impossible de calculer `k = K / G`.

Faire ce calcul inverse reviendrait à devoir déterminer le logarithme discret, un calcul aussi difficile que de tenter une attaque par brute force (en essayant une à une toutes les combinaisons possibles).

Même les calculateurs les plus puissants actuels sont pour le moment très loin d'être en capacité de réaliser un tel calcul.



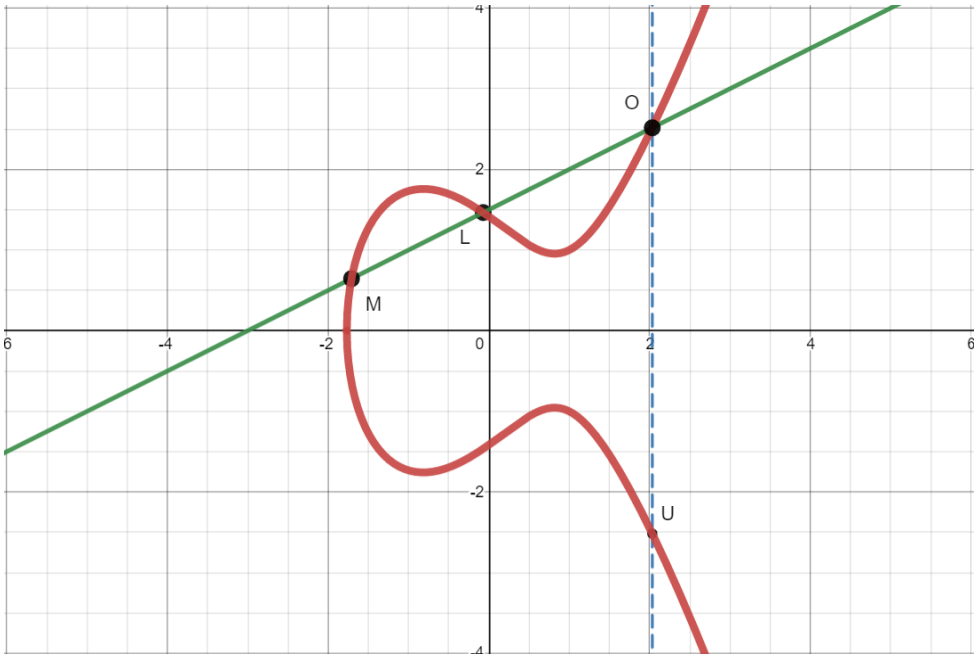
Multiplication sur les courbes elliptiques.

La notion d'addition sur les courbes elliptiques est définie tel que l'addition d'un point `M` sur une courbe elliptique à un autre point `L` sur la même courbe donnera un point `U`, de telle sorte que si l'on trace une droite entre `M` et `L`, elle viendra couper la courbe elliptique en un troisième point `O` qui est l'opposé de `U`.

Nous aurons donc :

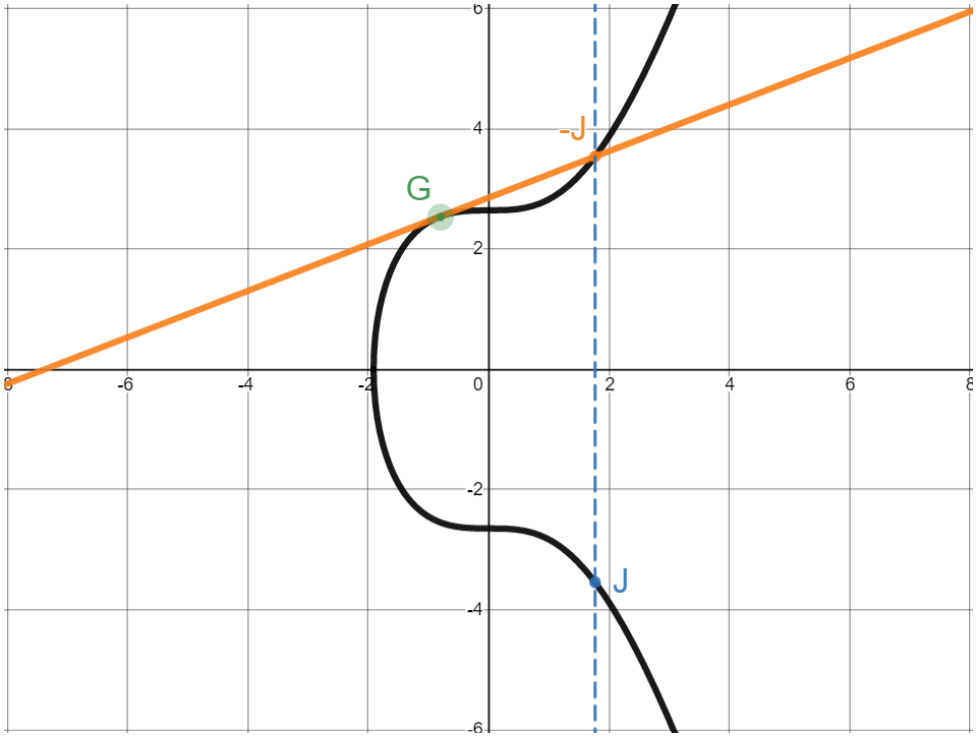
➔ `M + L = U`

On peut le représenter graphiquement avec la courbe ci-dessous :



Maintenant, si nous voulons ajouter un même point à lui même, cela reviendrait à tracer la tangente à la courbe en ce point et à récupérer l'opposé du point d'intersection entre la tangente et la courbe elliptique.

Graphiquement, cela donnerait cela :

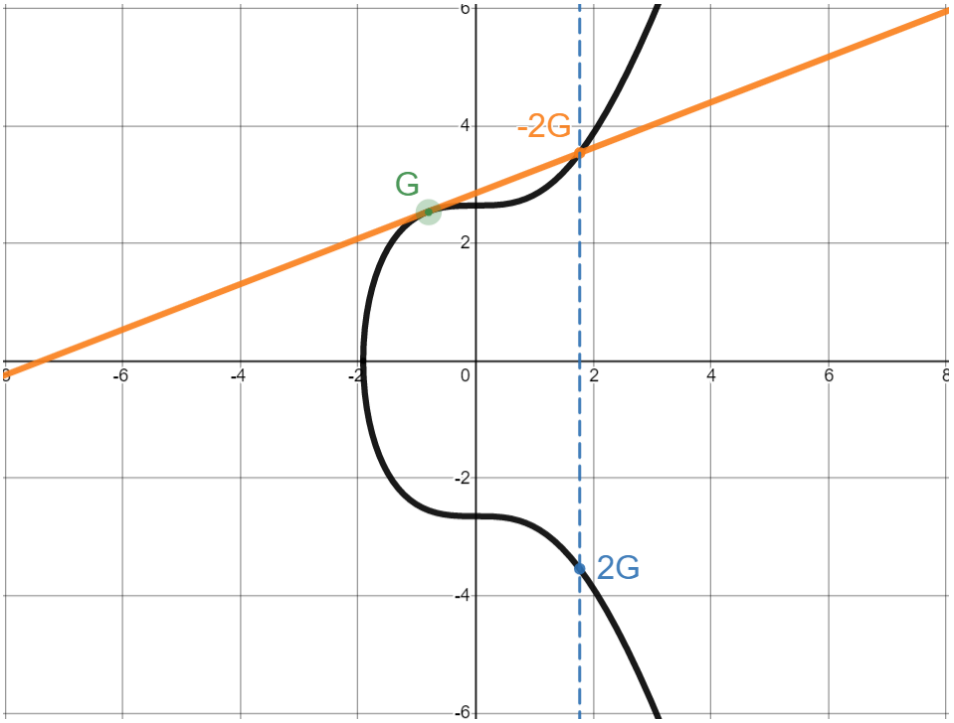


Dans cet exemple pour calculer le point `J` tel que `J = G + G`, nous avons simplement tracé la tangente à la courbe elliptique en le point `G` (courbe orange). Cette tangente coupe une nouvelle fois la courbe elliptique en un point appelé ici `-J`. L'opposé de ce point nous donne notre résultat `J`.

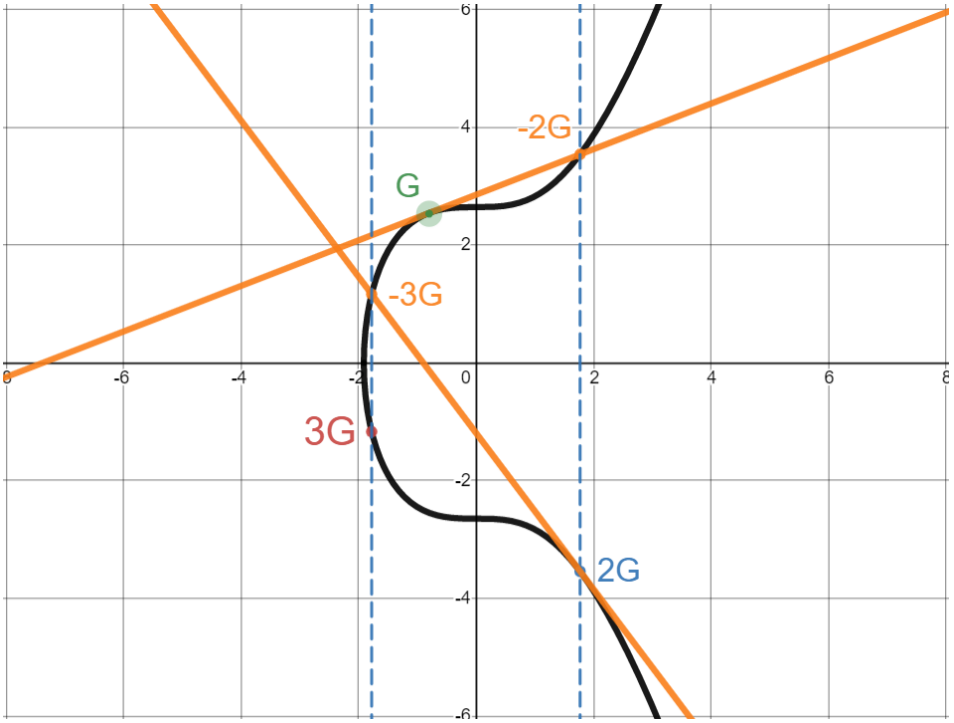
Maintenant que nous savons faire l'addition sur les courbes elliptiques à partir d'un point générateur (`G`), nous pouvons également réaliser la multiplication sur les courbes elliptiques.

Si nous reprenons notre exemple, calculer `G + G` revient à calculer `2 * G`.

Nous pouvons renommer nos points en conséquence :



Nous pouvons ainsi continuer et calculer `3G` en traçant la tangente de la courbe elliptique en `2G` et en prenant le point opposé par rapport à l'axe des abscisses.



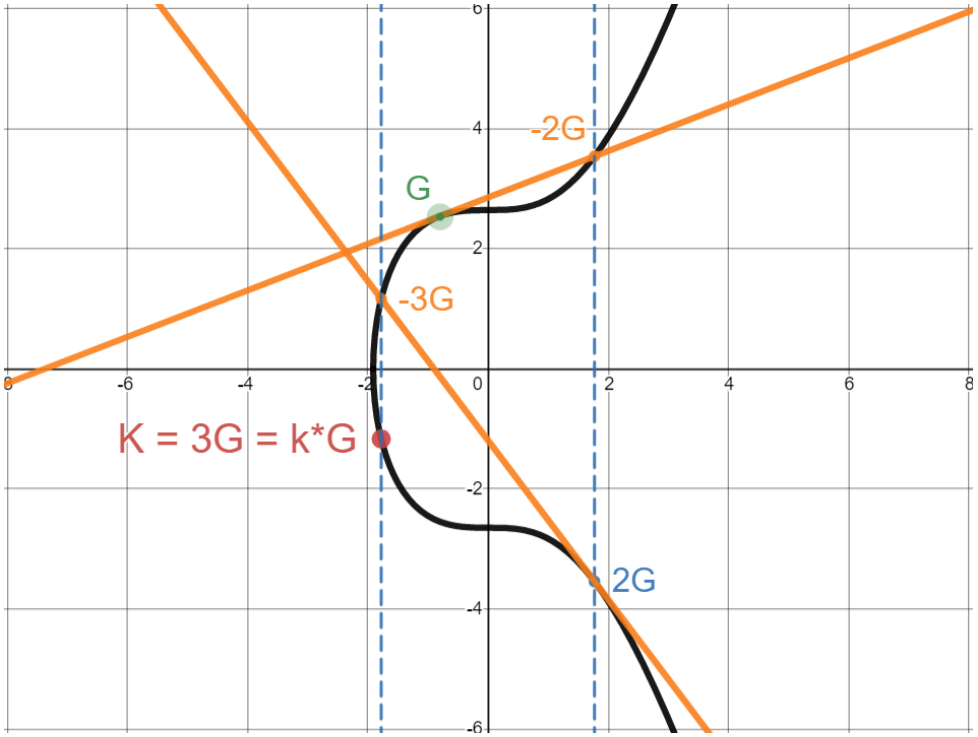
Fonction à sens unique.

Grâce à ces schémas nous pouvons comprendre facilement pourquoi la dérivation d'une clé publique en sachant la clé privée est très facile, mais l'inverse est impossible.

Reprenons notre exemple. Imaginons que nous ayons tiré un nombre aléatoire pour déterminer notre nouvelle clé privée et que ce nombre soit `3`. Nous avons donc `k = 3`.

Pour calculer la clé publique associée à notre clé privée, nous allons réaliser l'opération `K = k * G`. Dans notre exemple cela donne `K = 3 * G = 3G`.

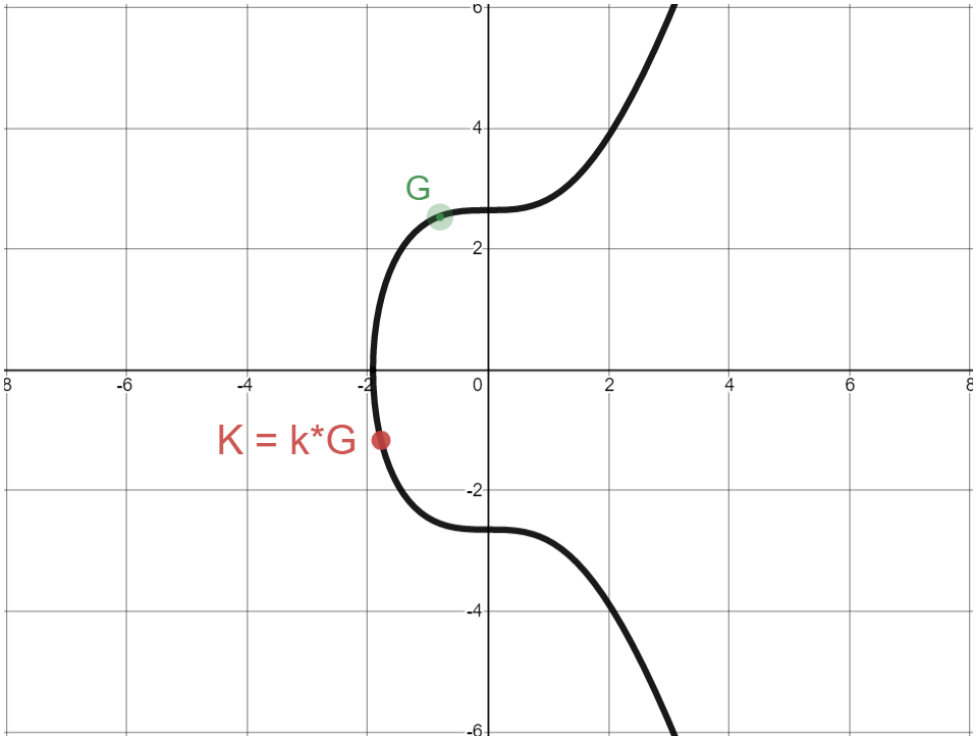
Cela nous donne cela :



Nous avons donc pu facilement calculer la clé publique `K` en sachant `k` et `G`.

Maintenant, si je vous donne uniquement la clé publique `K`, vous serez incapable de déterminer la valeur de la clé privée `k`.

Graphiquement, cela donnerait cela :



En étudiant uniquement cette figure, vous serez dans l'incapacité de calculer `k`, c'est-à-dire le nombre de fois que l'on a ajouté `G` à lui-même pour trouver `K`. Vous pourrez trouver `-K`, le premier opposé du point `K`, mais vous serez ensuite dans l'incapacité de déterminer d'où vient la tangente qui coupe la courbe elliptique en ce point `-K`.

C'est grâce à cela qu'il est impossible de déterminer une clé privée Bitcoin en connaissant uniquement sa clé publique.

Evidemment dans cet exemple vous pourriez calculer `K` par tâtonnement en partant de `G`, étant donné que j'ai intentionnellement choisi une clé privée `k` très petite égale à `3`. Il faut imaginer qu'en réalité ce calcul est infaisable car la clé privée est un nombre avec infiniment plus de possibilités (presque 2^{256}).

C'est sur ce principe qu'est basé la sécurité de l'algorithme ECDSA.

Signature numérique.

Maintenant que nous savons dériver une clé publique à partir d'une clé privée, nous pouvons déjà recevoir des bitcoins sur cette clé publique. Mais comment les dépenser ?

Pour dépenser des bitcoins il va falloir prouver au réseau que vous en êtes bien le propriétaire légitime. Il va donc falloir prouver mathématiquement au réseau que vous êtes en possession de la clé privée associée, sans pour autant la dévoiler.

C'est à ce moment-là qu'intervient la signature numérique, une preuve irréfutable que vous êtes bien en possession de la clé privée associée à la clé publique que vous revendiquez.

Pour réaliser une signature numérique il faut premièrement que tous les participants au réseau connaissent les paramètres de la courbe elliptique. Dans le cas de Bitcoin, les paramètres de *secp256k1* sont :

- Le champ fini $\mathbb{Z}_p$ défini par :

$$p = 2^{256} - 2^{32} - 977$$

$$= \text{0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE FFFFFC2F}$$

p est un nombre premier très grand légèrement inférieur à 2^{256}.

- La courbe $y^2 = x3 + ax + b$ sur $\mathbb{Z}_p$ définie par :

```
a = 0
b = 7
```

- Le point générateur ou point à l'origine `G` :

```
G = 0x02 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798
```

Forme comprimée qui donne uniquement l'abscisse du point `G`. Le préfixe `02` au départ permet de déterminer laquelle des 2 valeurs ayant cette abscisse *x* est à utiliser comme point générateur.

- L'ordre `n` de `G` (le nombre de points existants) et le cofacteur **`h`** :

```
n = 0xFFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6
AF48A03B BFD25E8C D0364141
```

`n` est un nombre très grand légèrement inférieur à ***p***.

```
h = 1
```

`h` est le cofacteur ou le nombre de sous-groupes. Je ne vais pas développer ce que cela représente car c'est assez complexe, et car dans le cas de *secp256k1* nous n'avons pas besoin de le prendre en compte étant donné qu'il est égal à `1`.

Toutes ces informations sont publiques et connues de tous les participants. Grâce à elles, les utilisateurs sont en capacité de réaliser une signature comme cela :

Imaginons qu'Alice souhaite envoyer des bitcoins à Bob. Imaginons également que Eve est un acteur malveillant qui souhaite voler ces bitcoins.

Comme vu précédemment, Alice a déterminé une paire de clés. La clé privée (`k` minuscule) est un nombre aléatoire de 256 bits compris entre `1` et `n - 1` et la clé publique (`K` majuscule) est un point sur la courbe calculé à partir de `k` et de `G` tel que : `K = k * G`.

Alice a déjà reçu une somme de bitcoins sur sa clé publique, elle souhaite désormais les envoyer à Bob.

L'information `K` est donc désormais connue par tous les utilisateurs, mais l'information `k` ne l'est pas. Seule Alice dispose de l'information `k`.

Pour pouvoir débloquer les bitcoins associés à sa clé publique `K`, et donc envoyer les bitcoins à Bob, Alice va devoir fournir une signature à l'aide de sa clé privée `k`. L'objectif est que ni Bob, ni Eve ne puissent calculer la valeur de `k`.

⇨ Pour vous expliquer la construction et la vérification d'une signature ECDSA, je vais d'abord vous donner les formules exactes, puis je vais vulgariser ce que ces formules complexes veulent dire. L'important n'est pas tant de comprendre les mathématiques qui régissent cela mais plutôt de comprendre pourquoi ce mécanisme fonctionne.

La signature **S** d'Alice sera partagée en deux parties que nous nommerons **S1** et **S2** tels que **S1** et **S2** concaténés donnent **S**, la signature ECDSA. Voici comment créer ces deux valeurs.

S1 :

Pour trouver **S1** il va falloir créer un nombre aléatoire très grand que nous nommerons v. Ce nombre doit être un nonce strictement inférieur à **n**. C'est-à-dire que ce nombre doit être différent pour chaque nouvelle signature, sans quoi il sera possible pour Eve de calculer la clé privée et de voler les bitcoins associés.

Alice utilise ensuite la multiplication sur les courbes elliptiques pour déterminer V, un point sur la courbe tel que V = v * G.

L'abscisse ***x*** de ce point V, *modulo* n sera la valeur de **S1** recherchée :

$$V = v * G = (x, y)$$
$$S1 = x \bmod n$$

S2 :

Pour trouver **S2** Alice commence par réaliser un hash de sa transaction Bitcoin non-signée. Ce hash sera nommé H(Tx). La transaction en elle-même étant nommée Tx.

Maintenant Alice peut calculer **S2** en utilisant l'équation suivante :

$$S2 = v^{-1} (H(Tx) + k * S1) \bmod n$$

Vérification de la signature.

Alice envoie donc sa transaction Bitcoin signée **`S (S1 || S2)`** au réseau**.** Matérialisés par les nœuds, les autres utilisateurs du réseau vont la vérifier.

Si la signature est valide, alors la transaction d'Alice pourra être incluse dans un bloc afin d'être confirmée, et les bitcoins seront transférés à la clé publique de Bob.

Imaginons que Bob vérifie la transaction d'Alice, de la même manière que tous les autres utilisateurs la vérifieront.

Bob dispose des paramètres de la courbe *secp256k1* : ***p***, `a`, `b`, `G` et `n`. Il dispose également des informations fournies par Alice, à savoir : `Tx`, **`S1`**, **`S2`** et `K`**.**

Il va commencer par calculer le hash de la transaction `Tx`. Il disposera alors de `H(Tx)`.

Pour vérifier la signature il va calculer un point `P(i,j)` tel que :

$$P = (S2^{-1} * H(Tx) \bmod n) * G + (S2^{-1} * S1 \bmod n) * K$$

Bob détermine ensuite l'abscisse de ce point `P` que l'on nommera *i*.

Si $i \bmod n = S1$, alors la signature est valide.

Ces calculs sont sympas, mais on a du mal à comprendre comment la vérification est possible. Je vous propose donc une vulgarisation de ce mécanisme afin de rendre sa compréhension moins complexe.

Vulgarisation.

Imaginons qu'Alice souhaite prouver à Bob qu'elle connaît un nombre secret `k` (la clé privée) sans pour autant lui révéler ce nombre. De plus, Alice et Bob communiquent via un réseau public surveillé par Eve, une attaquante qui souhaite subtiliser le secret `k`.

Rappelons que :

- La clé privée `k` n'est connue que par Alice. C'est un nombre aléatoire.
- La clé publique `K` est connue par Alice, par Bob et par Eve.
- `K` est déterminé par `k` et `G` (le point générateur) tel que : `K = k * G`.

En raison de l'utilisation de la multiplication sur les courbes elliptiques expliquée précédemment, et de sa caractéristique d'irréversibilité, Bob et Eve ne peuvent pas déterminer `k` en ayant uniquement connaissance de `K` et de `G`.

Voici comment fonctionne le mécanisme :

➔ Alice va déterminer un nouveau nombre secret aléatoire `v`. Elle va ensuite additionner `v` et `k`. La somme de ces deux nombres secrets sera `t` tel que : `t = v + k`.

Il est important que `v` reste secret, sinon `k` pourra être calculé par soustraction.

➔ Alice calcule ensuite un point sur la courbe elliptique nommé `V` tel que : `V = G * v`. Elle va donc ajouter `G` à lui-même `v` fois pour avoir le point `V`.

Ce point `V` va permettre de révéler juste assez d'informations sur `v` sans pour autant le dévoiler.

➔ Alice envoie à Bob `t` et `V`.

On passe ensuite à la vérification de Bob.

➔ Bob additionne la clé publique `K` avec `V`, tel que `T = K + V`.

➔ Bob calcule ensuite un point `T'` tel que `T' = t * G` en utilisant la multiplication sur les courbes elliptiques.

Si `T = T'` alors Bob est sûr qu'Alice connaît bien la valeur de `k`, et donc qu'Alice est bien en possession de la clé privée donnant accès aux bitcoins revendiqués.

En réalité, il existe une faille dans cet exemple de construction d'une signature que je viens de vous décrire. Cette faille pourrait être exploitée par Eve, l'actrice malveillante du réseau, pour essayer de subtiliser les bitcoins d'Alice.

Puisque Eve sait que Bob additionnera `K` avec `V` pour trouver le point `T`, elle peut créer une fausse valeur de `k` et calculer un faux `K`.

Eve va ensuite soustraire `K` du vrai point `V` qu'elle aura elle-même déterminé. Elle aura alors un nouveau point `V` malicieux. La nouvelle valeur du `V` malicieux sera alors `Vm = V - K`.

Le pauvre Bob qui va se faire berner ne pourra pas différencier le `Vm` malicieux du `V` légitime. Il interprètera donc `Vm` comme étant `V`.

Lors de la vérification il procédera donc comme précédemment :

- `T = K + Vm`. En sachant que `Vm = V - K` alors `T = K + V - K`.
- Les `K` s'annulent.
- `T = V`

Eve pourrait alors déterminer `t = v`.

Lors de la vérification de la deuxième partie, Bob aura donc :

```
→ t = v
→ T = v * G
→ T' = t * G = v * G
→ T' = T
```

La signature serait donc considérée comme valide par Bob et les autres vérificateurs, qui débloqueraient alors les bitcoins d'Alice en la faveur d'Eve, alors même que cette dernière n'a pas eu accès à la clé privée légitime.

Pour résoudre cette faille, l'algorithme ECDSA inclut un hash de la transaction. La fonction de hachage permet ici de s’assurer que le multiplicateur utilisé dispose des mêmes propriétés qu’un nombre pseudo-aléatoire, sans devoir faire confiance à l'émetteur pour le caractère aléatoire de ce dernier.

Voici donc comment il est possible de prouver au réseau Bitcoin que vous êtes bien le propriétaire d’un UTXO, sans pour autant rendre publique l’information qui permet de le débloquer.

Conclusion.

Nous avons pu découvrir en détail le fonctionnement, les caractéristiques et l'utilisation des fonctions de hachages et de la cryptographie sur les courbes elliptiques.

Certains concepts sont assez complexes à comprendre et ne sont pas forcément tous nécessaires pour appréhender le fonctionnement technique de Bitcoin. J'ai souhaité tout de même mettre ce tome au début de ma série d'ebooks car ces algorithmes cryptographiques constituent la base technique de Bitcoin. On les retrouve ainsi à tous les niveaux du protocole : minage, portefeuille, transaction…

Selon moi, les informations les plus importantes à retenir sont les caractéristiques de ces algorithmes, au-delà de leur fonctionnement technique. C'est cela qui vous permettra de comprendre pourquoi ils sont utilisés dans le protocole.

Nous avons donc pu voir que les fonctions de hachages disposent de trois caractéristiques principales : l'irréversibilité, la résistance à la falsification et la résistance aux collisions.

Les algorithmes de signatures numériques permettent deux usages principaux : la génération d'une clé publique à partir d'une clé privée, et la signature d'une transaction. La dérivation de la clé publique à partir de la clé privée est également une fonction irréversible.

La signature numérique permet de prouver au réseau que vous êtes bien le propriétaire d'une certaine clé publique en fournissant une preuve mathématique irréfutable que vous êtes en connaissance de la clé privée associée, tout en gardant secrète ladite clé privée.

Dans le prochain tome, nous allons mettre en application tous ces algorithmes car nous allons étudier la construction d'un portefeuille Bitcoin. Nous verrons ce que sont la phrase mnémonique, la graine, les clé étendues, les adresses ou encore les chemins de dérivation. Nous étudierons comment tous ces éléments sont calculés et utilisés au sein du protocole.

Si ce concept de petits ebooks techniques sur Bitcoin en français vous plaît, n'hésitez pas à partager ce contenu auprès de votre entourage et à me suivre sur Twitter où je vous communiquerai les sorties des prochains tomes de la série.

Je publie également du contenu de vulgarisation technique sur mon blog que vous pouvez retrouver en cliquant ici : https://www.pandul.fr/blog

Références.

Outils de visualisation de courbes elliptiques :
https://www.graui.de/

Articles sur les opérations à l'échelle du bit et l'algèbre de Boole :
https://medium.com/walkme-engineering/bitwise-operators-565e3ceb90cd
https://fr.wikipedia.org/wiki/Alg%C3%A8bre_de_Boole_(logique)

Articles et ressources sur le fonctionnement technique des fonctions de hachage :
https://infosecwriteups.com/breaking-down-sha-256-algorithm-2ce61d86f7a3
https://crypto.stackexchange.com/questions/3005/what-do-the-magic-numbers-0x5c-and-0x36-in-the-opad-ipad-calc-in-hmac-do
https://fr.wikipedia.org/wiki/HMAC
https://en.wikipedia.org/wiki/SHA-2

Articles et ressources sur le fonctionnement technique d'ECDSA :
https://cryptobook.nakov.com/digital-signatures/ecdsa-sign-verify-messages
https://suhailsaqan.medium.com/explanation-of-bitcoins-elliptic-curve-digital-signature-algorithm-6603f951863a
https://www.maximintegrated.com/en/design/technical-documents/tutorials/5/5767.html
https://jeremykun.com/2014/02/08/introducing-elliptic-curves/
https://www.johndcook.com/blog/2018/08/14/bitcoin-elliptic-curves/

Rapport FIPS 180-4 : *Secure Hash Standard (SHS)* :
https://csrc.nist.gov/csrc/media/publications/fips/180/4/final/documents/fips180-4-draft-aug2014.pdf

Rapport FIPS PUB 186-4 : *Digital Signature Standard (DSS)* :
https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf

Rapport *SEC 2: Recommended Elliptic Curve Domain Parameters* :
https://www.secg.org/SEC2-Ver-1.0.pdf

Contacts.

Loïc Morel

Email : loic@pandul.fr

Site web : https://www.pandul.fr/

Télécharger la série d'ebooks : https://www.pandul.fr/ressources

Blog : https://www.pandul.fr/blog

Twitter Loïc Morel (@Loic_Pandul) : https://twitter.com/Loic_Pandul