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.

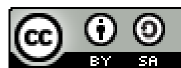
Adapter, remixer, transformer et construire sur le contenu texte et illustrations, à quelque fin que ce soit, même commercialement, à l'exception des logos de Pandul seuls qui sont strictement protégés.

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.

Sommaire.

Remerciements. **2**

Préambule. **3**

Introduction. **4**

1- Les fonctions de hachage. **5**
- Caractéristiques d'une fonction de hachage. 5
- Les fonctions de hachage dans Bitcoin. 9
- Les rouages de SHA256. 10
- Les algorithmes utilisés pour la dérivation. 22

2- Les signatures numériques. **26**
- La courbe elliptique. 28
- La clé privée. 32
- La clé publique. 33
- Addition et doublement de points. 35
- Fonction à sens unique. 38
- Signature numérique. 40
- Vérification de la signature. 43
- Vulgarisation. 44

Conclusion. **46**

Références. **47**

Contacts. **48**

Remerciements.

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

- Grittoshi → https://twitter.com/Grittoshi
- 200KEKS → https://twitter.com/200KEKS
- Fanis Michalakis → https://twitter.com/FanisMichalakis
- Sosthène → https://twitter.com/Sosthene___

Merci également à tous ceux qui m'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 des vulgarisations, 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 ouvrages, un glossaire avec des définitions est publié sur cette page : Glossaire série d'ebooks - Bitcoin Démocratisé.

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.

Introduction.

Ce premier tome est en quelque sorte préparatoire pour la suite de la série d'ebooks. Nous allons y étudier les différents algorithmes liés à la cryptographie 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 omniprésente dans 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 est donc initialement 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. Aujourd'hui, on confond souvent ces deux termes en un seul. Ainsi, selon le NIST[1], la cryptographie moderne est une discipline qui incarne les principes, les moyens et les méthodes de transformation des données afin de masquer leur contenu sémantique, d'empêcher leur utilisation non autorisée ou d'empêcher leur modification non détectée.

Il est intéressant de remarquer que cette discipline 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 sont les fonctions de hachage et les signatures numériques. Voyons ensemble ce en quoi elles consistent.

[1] NIST : ***N****ational* ***I****nstitute of* ***S****tandards and* ***T****echnology.*

1- Les fonctions de hachage.

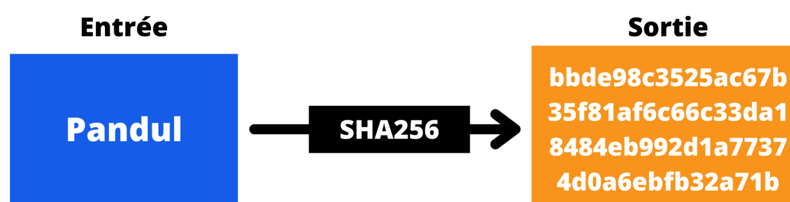
Caractéristiques d'une fonction de hachage.

Le premier type d'algorithmes cryptographiques utilisé dans Bitcoin regroupe les fonctions de hachage. Elles sont utilisées à de nombreuses reprises dans le protocole.

Le hachage est un procédé permettant de mapper une chaîne de bits[2] de longueur arbitraire à une chaîne de bits de longueur fixe grâce à une fonction de hachage cryptographique. En d'autres termes, la fonction de hachage prend en entrée (input) une information de taille arbitraire pour la convertir en une empreinte de taille fixe appelée "hash" en sortie (output).

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

Par exemple, la fonction SHA256[3] produira un hash d'une taille fixe de 256 bits. Ainsi, si l'on met en entrée de cette fonction "`Pandul`", un message de taille arbitraire, le hash en sortie est : "`bbde98…`", une empreinte de 256 bits au format hexadécimal[4].



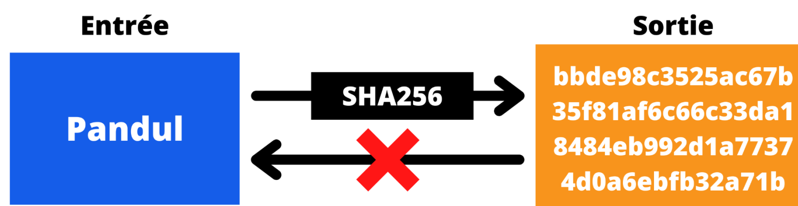
Ces fonctions de hachage cryptographiques sont utilisées dans Bitcoin car elles disposent de 3 caractéristiques intéressantes.

[2] Bit : Unité d'information pouvant prendre une parmi deux valeurs : 0 ou 1. C'est le diminutif de "***bi**nary*" et "*dig**it***". Il est utilisé au sein du système de numération binaire, également appelé base 2.
[3] SHA256 : ***S**ecure **H**ash **A**lgorithm **256***. Cette fonction fait partie de la famille des SHA-2.
[4] Hexadécimal : Système de numération en base 16. Les 16 symboles utilisés sont les chiffres de *0* à *9* et les lettres de *A* à *F*.

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

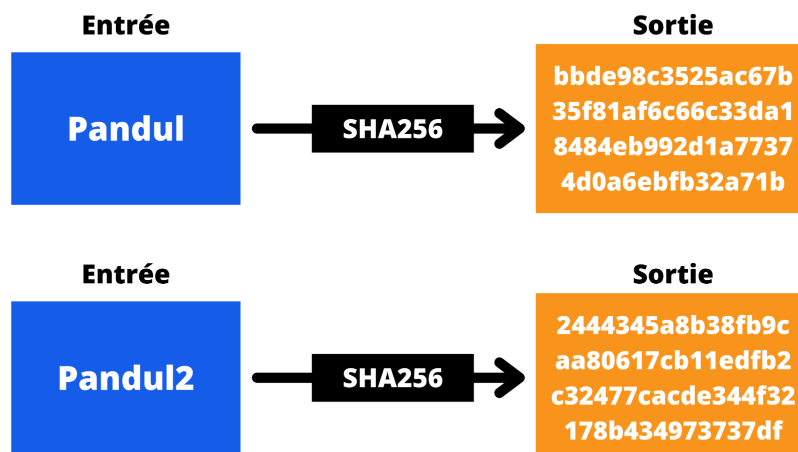
L'irréversibilité[5] se vérifie lorsque le calcul du hash sachant l'information en entrée peut être réalisé facilement, mais le calcul inverse, c'est-à-dire le calcul de l'information en entrée sachant le hash, est impossible. C'est ce que l'on appelle une fonction à sens unique ou "*trap door function*".

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



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

Cette caractéristique est vérifiée uniquement si la moindre petite modification sur le message en entrée résulte en 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 :



[5] En réalité, le terme technique utilisé pour décrire cette caractéristique d'irréversibilité est la "Résistance à la préimage".

Grâce à cette caractéristique, les fonctions de hachage sont utilisées pour vérifier l'intégrité d'une information en mettant en évidence la moindre modification de l'entrée originelle.

➔ La troisième caractéristique d'une fonction de hachage cryptographique est la résistance à la collision.

Cette caractéristique se vérifie s'il est difficile de trouver deux entrées différentes qui donnent le même condensat en sortie.

La 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[6]. Ainsi, une bonne fonction de hachage cryptographique est une fonction pour laquelle le risque de collision entre deux valeurs est tellement faible qu'il peut être considéré comme nul.

Pour que cette caractéristique soit attribuée à une fonction, il ne faut pas qu'une collision puisse être trouvée sur celle-ci plus rapidement qu'en essayant simplement des entrées aléatoires une à une, comme avec l'attaque des anniversaires. Cette attaque illustre une limite à la résistance à la collision d'une fonction de hachage égale à $2^{n/2}$, où n est la taille du condensat en bits.

Par exemple, la fonction de hachage SHA256 a une résistance aux collisions attendue limitée à 2^{128}. Concrètement, cela veut dire que si un attaquant essaie 2^{128} entrées aléatoires différentes, il est très probable qu'il puisse trouver une collision en sortie de la fonction.

On peut donc admettre que pour deux valeurs distinctes en entrée d'une fonction de hachage cryptographique, il est presque impossible d'obtenir exactement le même hash en sortie.

Cette caractéristique n'est par exemple plus vérifiée sur les algorithmes SHA-0 et SHA-1, prédécesseurs des SHA-2, pour lesquels des collisions ont été trouvées. Ces fonctions sont donc aujourd'hui déconseillées et souvent considérées comme désuètes.

[6] Ce phénomène est appelé le principe des tiroirs ("*Pigeonhole principle*" en anglais).

- ➔ Il existe une quatrième caractéristique que l'on peut observer sur une fonction de hachage cryptographique qui est la résistance à la seconde préimage.

Je ne vais pas la détailler ici puisqu'une résistance aux collisions implique forcément une résistance à la seconde préimage. La différence entre les deux tient aux informations imposées à un attaquant.

Pour la résistance à la seconde préimage, un attaquant dispose d'un message imposé m_1 et il doit trouver un message m_2 donnant le même hash que m_1. Pour la résistance à la collision, un attaquant peut choisir librement m_1 et m_2.

Ces trois caractéristiques que je viens de décrire permettent de déterminer de nombreux cas d'usages pour ces primitives cryptographiques que représentent les fonctions de hachage.

Les fonctions de hachage dans Bitcoin.

La fonction de hachage la plus utilisée dans le protocole Bitcoin est **SHA256**. Comme nous l'avons vu précédemment, 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 employée au sein du protocole Bitcoin à différents niveaux. Elle est notamment utilisée pour hacher l'entête d'un bloc dans le mécanisme de Proof-of-Work. On la retrouve également dans le processus de dérivation des adresses de réception, dans la structuration des transactions en Arbre de Merkle au sein des blocs, ou encore dans le calcul de la somme de contrôle d'une phrase de récupération.

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

On utilise aussi dans Bitcoin sa variante SHA512, qui fait partie de la même famille des SHA2, mais qui produit un condensat de 512 bits. Elle est notamment utilisée dans la dérivation d'un portefeuille HD[7], processus que nous étudierons en détail dans le **tome 2**. Notons que cette fonction de hachage cryptographique n'est pas implémentée directement sur le protocole Bitcoin, mais on la retrouve au sein d'autres algorithmes cryptographiques implémentés.

Enfin, la troisième fonction de hachage utilisée dans le protocole est RIPEMD160[8]. Comme son nom l'indique, elle produit un condensat de 160 bits. On la retrouve notamment dans le processus de dérivation des adresses de réception.

Lorsque l'on évoque HASH256 dans Bitcoin, cela décrit un double hachage successif avec la fonction SHA256[9]. Lorsque l'on évoque HASH160, cela décrit un double hachage successif utilisant d'abord SHA256 puis RIPEMD160[10].

[7] HD : ***H**ierarchical **D**eterministic*. En français : déterministe hiérarchique.
[8] RIPEMD : ***R**ACE **I**ntegrity **P**rimitives **E**valuation **M**essage **D**igest*.
RACE : ***R**esearch and Development in **A**dvanced **C**ommunications Technologies in **E**urope*.
[9] Cette double fonction est notamment utilisée dans les arbres de Merkle et dans le protocole de preuve de travail.
[10] Cette double fonction est notamment utilisée dans le processus de dérivation d'une adresse de réception, et pour générer les empreintes de clés.

Les rouages de SHA256.

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

Les fonctions SHA256 et SHA512 appartiennent à la famille des *Secure Hash Algorithm 2*. Le mécanisme de ces deux fonctions de hachage est très similaire. Il est basé sur une construction spécifique appelée Merkle-Damgard. RIPEMD160 est également basée sur le même type de construction. Dans cet ebook, nous allons donc étudier uniquement 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.

Ce processus se décompose en plusieurs étapes :

1- Les bits de 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.

Les bits de rembourrage consistent à ajouter des bits à notre message initial afin qu'il atteigne premièrement une taille égale au multiple de 512 bits supérieur le plus proche, moins 65 bits.

On a donc :

```
M + 1 + P + 64 = n * 512
```

où :

- `M` est la taille de notre message initial (arbitraire).
- `P` est la taille de nos bits de rembourrage sans le séparateur.
- `n*512` est le premier multiple de 512 bits supérieur à `M+64+1`.
- `1` est le bit réservé au séparateur "**1**" entre le message et les bits de rembourrage.
- `64` est le nombre de bits réservés pour le rembourrage avec la taille (voir étape 2).

Les bits de rembourrage commencent donc par un séparateur **`1`**, suivi par le nombre de **`0`** nécessaires pour avoir une taille de `n*512-64` bits.

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

```
Le premier multiple de 512 supérieur à M+64+1 est :
1024 = 2 * 512
Nous avons donc :
→ M + 1 + P + 64 = 1024
→ 950 + P = 1024 - 1 - 64
→ P = 1024 - 1 - 64 - 950
→ P = 9
```

Dans cet exemple, notre première partie de rembourrage doit être de 9 bits + le bit réservé au séparateur `1`. On commence donc par un “**1**”, puis on complète les bits de rembourrage avec 9 bits égaux à “**0**”.

Dans notre exemple, cela nous donne les bits de rembourrage avec séparateur suivants : `1 000 000 000`. Nous ajoutons ces bits de rembourrage à la suite de notre message initial.

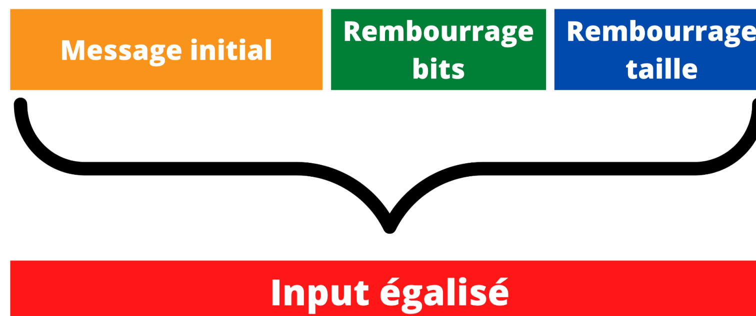
2- Le rembourrage avec la taille (pré-traitement).

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.

Ces 64 bits sont la représentation binaire de la taille du message initial en bits. Si on reprend notre exemple avec un message initial de 950 bits, on va convertir le nombre décimal *950* en nombre binaire ce qui nous donne **11 1011 0110**. On complète ce nombre avec des zéros à la base pour faire 64 bits au total. Dans notre exemple, cela donne :

```
0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0011 1011 0110
```

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 et vecteurs d'initialisation.

La fonction SHA256 est configurée avec plusieurs valeurs de 32 bits par défaut dont nous aurons besoin à l'étape suivante. Tout d'abord, il existe 8 vecteurs d'initialisation constants associés aux lettres `A à H`[11] :

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

Ces vecteurs d'initialisation représentent chacun les 32 premiers bits des parties décimales des racines carrées des 8 premiers nombres premiers.

On définit également 64 constantes associées à la lettre K_i :

```
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
```

Ces constantes représentent chacune les 32 premiers bits des parties décimales des racines cubiques des 64 premiers nombres premiers. Toutes ces informations seront utiles à l'étape suivante.

---

[11] **0x** est un préfixe noté avant une nombre pour signifier que celui-ci est au format hexadécimal (base 16). Par exemple, *0x1F* désigne le nombre *1F* en hexadécimal.

4- La compression.

On va maintenant pouvoir s'attaquer au traitement. 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, nous divisons notre input 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 chaque morceau.

Avant d'étudier les rouages de la fonction de compression, il est important de comprendre le fonctionnement des opérations utilisées au sein de celle-ci. Nous allons utiliser des calculs basées sur l'algèbre de Boole afin de réaliser des opérations au niveau du bit. Ce sont des opérations logiques qui peuvent être décrites par trois opérations de base :

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

A partir de ces opérations logiques, nous allons pouvoir déterminer des fonctions plus complexes comme le *XOR* (*OU* exclusif), une opération très utilisée en cryptographie. Le but de toutes ces opérations est de traiter les opérandes au niveau du bit.

Ces calculs permettent donc de modéliser des raisonnements logiques à partir d'un état et en fonction de conditions. Chaque opération dispose d'une table de vérité produisant un résultat de façon déterministe à partir des deux opérandes de celle-ci.

Au sein de la fonction de compression des algorithmes de hachage de la famille SHA-2, on utilise les opérations suivantes :

- *XOR* (⊕)
- *AND* (∧)
- *NOT* (¬)

Les tables de vérité de ces opérations sont les suivantes pour toute opérande *p* et *q* :

p	*q*	*XOR* (⊕)	*AND* (∧)	*NOT p* (¬p)	*NOT q* (¬q)
1	1	0	1	0	0
1	0	1	0	0	1
0	1	1	0	1	0
0	0	0	0	1	1

Par exemple, si on *XOR* (⊕) le nombre de 6 bits `101100` avec le nombre `001000` cela nous donnera cela :

```
101100 ⊕ 001000 = 100100
```

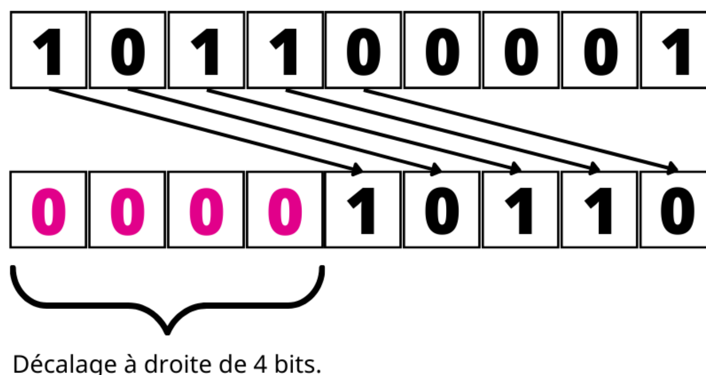
Dans cet exemple, nous avons simplement traité chaque bit de la première opérande avec son bit de même rang sur la deuxième opérande en appliquant la table de vérité du *XOR* présentée ci-dessus.

En plus de ces opérations logiques, la fonction de compression des SHA-2 utilise des opérations de décalage de bits. En l'occurrence, nous allons utiliser l'opération de décalage à droite *ShR* (***Sh**ift **R**ight*) et l'opération de décalage circulaire à droite *RotR* (***Rot**ate **R**ight Shift*).

Leur fonctionnement est très simple. A partir d'une chaîne de bits, l'opération ShR^n va décaler tous les bits à droite de ***n*** rangs, et va compléter les ***n*** trous réalisés sur la chaîne avec ***n*** zéros. Par exemple :

```
ShR⁴ (101100001) = 000010110
```

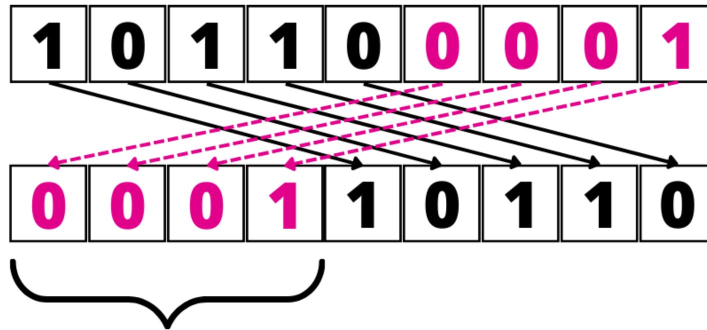
Schématiquement, l'opération ShR^4 sur notre exemple ressemble à cela :



Quant à l'opération $RotR^n$, à partir d'une chaîne de bits, elle va simplement prendre les ***n*** derniers bits de la suite pour les placer en premiers, décalant ainsi les autres bits de ***n*** rangs à droite. Par exemple :

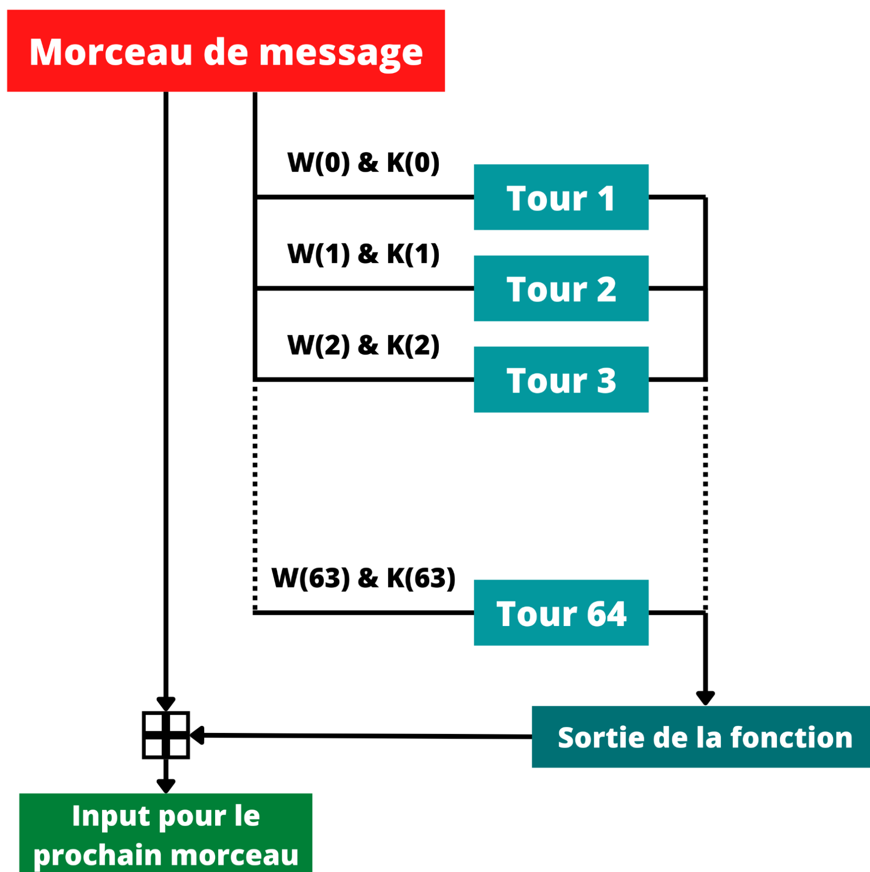
```
RotR⁴ (101100001) = 000110110
```

Schématiquement, l'opération *RotR⁴* sur notre exemple ressemble à cela :



Toutes ces opérations au niveau du bit sont utilisées au sein de la fonction de compression de SHA256. Maintenant que nous avons compris comment fonctionnent ces opérations, étudions ensemble les rouages de celle-ci.

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}, sont constitués par notre morceau de message.

Pour rappel, chaque morceau fait 512 bits. Ses 32 premiers bits seront donc 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}, il existe une formule qui consiste à appliquer certaines des opérations que nous venons de voir sur les bits du message pour générer les nombres de W_{16} à W_{63} :

$$W_i = W_{i-16} + \sigma_0^{\{256\}} + W_{i-7} + \sigma_1^{\{256\}}$$

où :

$$\sigma_0^{\{256\}} = RotR^7(W_{i-15}) \oplus RotR^{18}(W_{i-15}) \oplus ShR^3(W_{i-15})$$

$$\sigma_1^{\{256\}} = RotR^{17}(W_{i-2}) \oplus RotR^{19}(W_{i-2}) \oplus ShR^{10}(W_{i-2})$$

L'addition (+) est effectuée *modulo* 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` est égal à `1`. On l'utilise ici afin de ramener la taille d'un message donné à 32 bits.

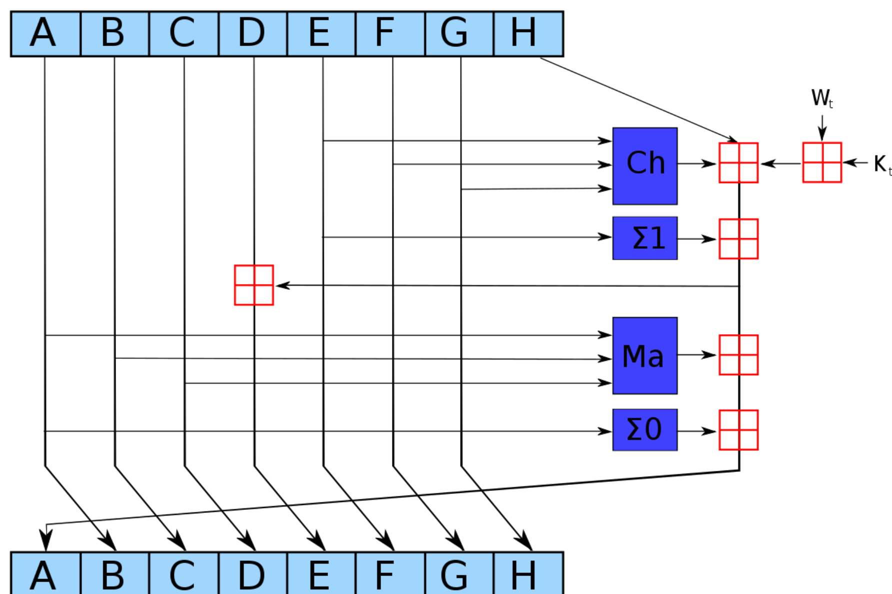
On sait donc maintenant comment déterminer chaque entrée W_i.

Les entrées K_i, c'est-à-dire K_0, K_1, [...], K_{63}, correspondent simplement aux 64 constantes décrites à l'étape 3. Elles vont être notre deuxième entrée.

Enfin, la troisième entrée de notre fonction de compression est 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 vecteurs d'initialisation est utilisée en input pour le premier tour du premier morceau 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 entrées, regardons plus en détail comment fonctionnent ces 64 tours :



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

Ce schéma représente un seul tour de notre fonction de compression. On peut y reconnaître nos 3 entrées de fonction :

- 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, et on continue de même jusqu'au 64ème tour.

Le symbole **[+]** sur le schéma ci-dessus correspond à une addition *modulo* 2^{32}. Les autres calculs du tour se décomposent comme cela :

$$Ch(E, F, G) = (E \wedge F) \oplus ((\neg E) \wedge G)$$

$$Maj(A, B, C) = (A \wedge B) \oplus (A \wedge C) \oplus (B \wedge C)$$

$$\Sigma_0^{\{256\}}(A) = RotR^2(A) \oplus RotR^{13}(A) \oplus RotR^{22}(A)$$

$$\Sigma_1^{\{256\}}(E) = RotR^6(E) \oplus RotR^{11}(E) \oplus RotR^{25}(E)$$

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 $[A_S, B_S, C_S, D_S, E_S, F_S, G_S, H_S]$[12] 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 :

```
Pour i de 0 à 63 :

Σ1{256}(EE) = RotR6(EE) ⊕ RotR11(EE) ⊕ RotR25(EE)

Ch(EE, FE, GE) = (EE ∧ FE) ⊕ ((¬ EE) ∧ GE)

temp1 = HE + Σ1{256}(EE) + Ch(EE, FE, GE) + Ki + Wi

Σ0{256}(AE) = RotR2(AE) ⊕ RotR13(AE) ⊕ RotR22(AE)

Maj(AE, BE, CE) = (AE ∧ BE) ⊕ (AE ∧ CE) ⊕ (BE ∧ CE)

temp2 = Σ0{256}(AE) + Maj(AE, BE, CE)
```

$$\Sigma_1^{\{256\}}(E_E) = RotR^6(E_E) \oplus RotR^{11}(E_E) \oplus RotR^{25}(E_E)$$

$$Ch(E_E, F_E, G_E) = (E_E \wedge F_E) \oplus ((\neg E_E) \wedge G_E)$$

$$temp1 = H_E + \Sigma_1^{\{256\}}(E_E) + Ch(E_E, F_E, G_E) + K_i + W_i$$

$$\Sigma_0^{\{256\}}(A_E) = RotR^2(A_E) \oplus RotR^{13}(A_E) \oplus RotR^{22}(A_E)$$

$$Maj(A_E, B_E, C_E) = (A_E \wedge B_E) \oplus (A_E \wedge C_E) \oplus (B_E \wedge C_E)$$

$$temp2 = \Sigma_0^{\{256\}}(A_E) + Maj(A_E, B_E, C_E)$$

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

Le symbole (+) correspond encore à une addition *modulo* 2^{32}.

Voici donc comment se décompose un seul tour. Une fois que l'on a la sortie de ce tour (la nouvelle suite de $[A_S, B_S, C_S, D_S, E_S, F_S, G_S, H_S]$), on l'utilise comme entrée pour le tour suivant. On continue ainsi de suite jusqu'à arriver au 64ème tour du morceau. L'output du 64ème tour sera additionné *modulo* 2^{32} à l'input du premier tour de ce morceau, soit la suite $[A_0, B_0, C_0, D_0, E_0, F_0, G_0, H_0]$ initiale. Chaque lettre sera additionnée *mod* 2^{32} à sa même lettre en entrée :

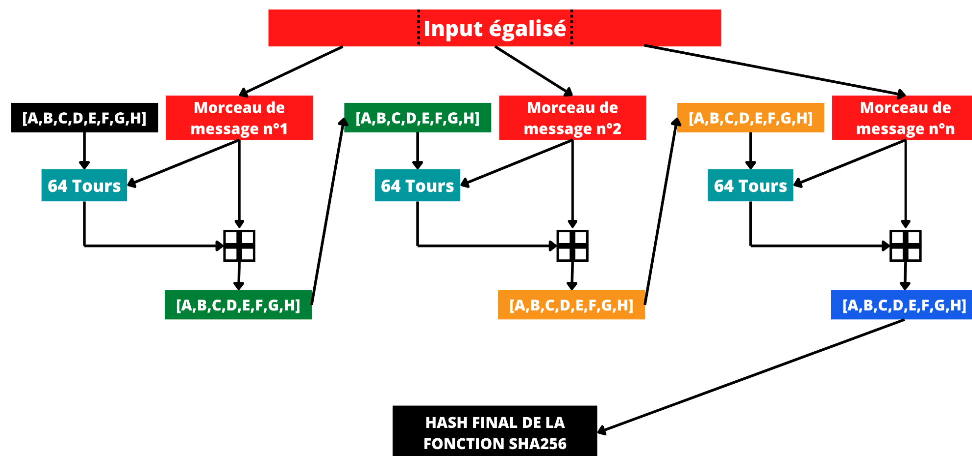
$A = A_0 + A_{63}$ | $B = B_0 + B_{63}$
$C = C_0 + C_{63}$ | $D = D_0 + D_{63}$
$E = E_0 + E_{63}$ | $F = F_0 + F_{63}$
$G = G_0 + G_{63}$ | $H = H_0 + H_{63}$

[12] Ici, la couleur orange permet de mettre en évidence la suite de sortie et la couleur bleu indique la suite d'entrée d'un seul tour de la fonction de compression.

Cette nouvelle suite **`[A,B,C,D,E,F,G,H]`** est concaténée[13] dans l'ordre alphabétique pour ne donner plus qu'un nombre de 256 bits. Ce nombre est le hash intermédiaire du premier morceau. Ce condensat intermédiaire servira d'input pour le premier tour du second morceau de 512 bits. Ce second morceau va également passer ses 64 tours. Et on continue ainsi de suite jusqu'à avoir traité tous les morceaux de 512 bits de notre message initial égalisé.

La sortie finale de notre fonction de hachage SHA256 est l'output du 64ème tour du dernier morceau additionné *modulo* 2^{32} à l'input du 1er tour du dernier morceau. C'est-à-dire que la sortie de la fonction de hachage est le hash intermédiaire du dernier morceau.

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



Mais alors, en quoi cette fonction est-elle irréversible, résistante aux collisions et résistante à la falsification ?

Pour la résistance à la falsification c'est assez simple à comprendre. Il y a tellement de calculs effectués en cascade, qui dépendent à la fois de l'entrée et des constantes, que la moindre modification du message initial change complètement le chemin parcouru, et donc change complètement le hash de sortie.

Cette propriété est en partie assurée par le mélange des états intermédiaires avec les états initiaux pour chaque morceau.

[13] L'opération de concaténation consiste à mettre bout à bout les opérandes.

Ensuite, lorsque l'on parle d'une fonction de hachage cryptographique, le terme "irréversibilité" n'est généralement pas utilisé. À la place, on parle de "résistance à la préimage" qui spécifie que pour tout y donné, il est difficile de trouver un x tel que $\mathbf{h}(x) = y$. Cette résistance à la préimage, quant à elle, est garantie par les opérations utilisées dans la fonction de compression et par la perte de certaines informations essentielles dans le processus. Par exemple, pour un résultat donné à une addition *modulo* il existe plusieurs opérandes possibles :

```
3+2 mod 10 = 5
7+8 mod 10 = 5
5+10 mod 10 = 5
```

On voit bien dans cet exemple qu'en sachant uniquement le *modulo* utilisé (10) et le résultat (5) on ne peut pas déterminer avec certitude quelles sont les deux bonnes opérandes utilisées dans l'addition. On dit qu'il existe plusieurs congrus *modulo* 10.

Pour l'opération *XOR* on est confronté au même problème. Rappelez-vous de la table de vérité de cette opération : toute sortie de 1 bit peut être déterminée par deux configurations différentes en entrées qui ont exactement la même probabilité d'être les bonnes valeurs. On ne peut donc pas déterminer avec certitude les opérandes d'un *XOR* en sachant uniquement son résultat. Si on augmente la taille des opérandes du *XOR*, le nombre de possibles entrées en sachant uniquement le résultat augmente de façon exponentielle. De plus, le *XOR* est souvent utilisé aux côtés d'autres opérations au niveau du bit, comme l'opération *RotR*, qui viennent ajouter encore plus d'interprétations possibles au résultat.

On utilise également au sein de la fonction de compression l'opération *ShR*. Celle-ci vient supprimer une partie de l'information de base qui est donc impossible à retrouver par la suite. Il n'y a encore une fois pas de moyen algébrique pour inverser cette opération. Toutes ces opérations à sens unique et ces opérations de perte d'information sont utilisées à de très nombreuses reprises dans les fonctions de compression. Le nombre de possibles entrées pour une sortie donnée est donc presque infini, et chaque tentative de calcul inverse mènerait à des équations avec un nombre d'inconnus très élevé qui augmenterait exponentiellement à chaque étape.

Enfin, pour la caractéristique de résistance aux collisions, plusieurs paramètres entrent en compte. Le pré-traitement du message d'origine tient un rôle essentiel. Sans ce pré-traitement, il pourrait être plus facile de trouver des collisions sur la fonction.
Cette caractéristique est aussi assurée par les propriétés de résistance à la préimage et de résistance à la falsification.

En effet, pour qu'il y est une résistance aux collisions, il faut que toute sortie de la fonction soit imprévisible. Toute prévisibilité peut être utilisée pour chercher des collisions plus rapidement que par une attaque des anniversaires (brute force). L'objectif étant que tout bit en sortie ne puisse être deviné à l'avance avec une probabilité supérieure à 0,5. Généralement, les cryptographes qui construisent de telles fonctions déterminent d'abord les meilleures attaques possibles pour trouver une collision, puis ils adaptent leurs paramètres afin de rendre ces attaques désuètes.

Comme vu précédemment, la collision est impossible à éviter au vu du principe des tiroirs. Si l'on néglige le fait que l'étape du rembourrage vient limiter la taille maximum de l'entrée à environ 2 exaoctets, alors pour toute valeur donnée en entrée il existe une infinité d'autres valeurs qui produisent une collision sur la fonction.

Néanmoins, si le résultat de la fonction est imprévisible, on peut alors admettre que les hash ont une distribution pseudo-aléatoire par rapport au message d'origine. Si l'ensemble des valeurs réalistes en entrée est inférieur à l'ensemble des sorties possibles, et si la distribution est pseudo-aléatoire, alors le risque de collision est très faible sur une fonction qui produit un hash suffisamment grand.

➩ Cette structure de la fonction de hachage SHA256 est une construction dite de Merkle-Damgard. Il existe d'autres types de constructions comme la construction de l'éponge utilisée pour la famille de fonctions de hachage SHA3[14] ou pour Keccak-256[15], une des fonctions de hachage utilisées sur Ethereum.
Ce type de construction de Merkle-Damgard rend certaines fonctions de hachage vulnérables aux attaques par extension de longueur. Ce type d'attaque peut notamment être réalisée sur un hachage qui utilise en entrée une concaténation de deux informations.
C'est probablement pour cette raison que Satoshi Nakamoto a implémenté un double hachage SHA256[16] à plusieurs endroits dans le protocole, au lieu d'un simple hachage SHA256.

[14] Keccak est la fonction initiale, conçue par Guido Bertoni, Joan Daemen, Michaël Peeters, Seth Hoffert, Ronny Van Keer et Gilles Van Assche, gagnante du concours du NIST pour le standard SHA-3. Cette fonction fut légèrement modifiée par le NIST pour définir le standard SHA-3 final. Aujourd'hui, on différencie Keccak qui est l'algorithme original et SHA-3 qui est l'algorithme modifié par le NIST. Les deux fonctions sont similaires mais elles ne produisent pas les mêmes condensats.
[15] Contrairement à d'autres fonctions de hachage, le nom de "Keccak" n'est pas un acronyme. D'après le cryptographe Jean-Philippe Aumasson, inventeur de la fonction de hachage BLAKE, le nom viendrait d'une danse traditionnelle balinaise nommée "Kecak".
[16] Le double hachage successif SHA256 est nommé HASH256 dans Bitcoin et SHA256d de manière plus générale.

Les algorithmes utilisés pour la dérivation.

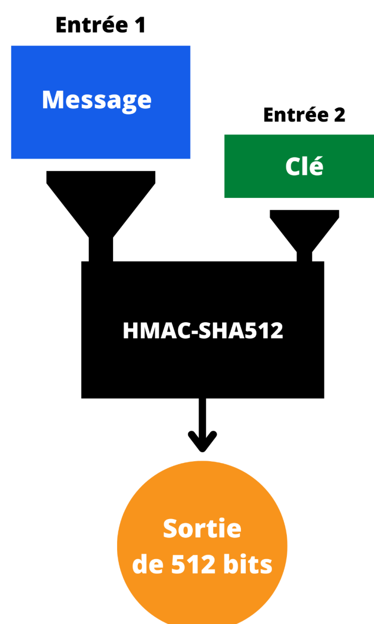
À partir de ces primitives cryptographiques que représentent les fonctions de hachage, le protocole Bitcoin utilise également des algorithmes cryptographiques pour de la dérivation. Ces fonctions de dérivation sont basées sur des fonctions de hachage, elles sont donc comparables à ces dernières. La principale différence entre une fonction de hachage et un algorithme utilisé pour de la 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 en partie les mêmes caractéristiques que celle-ci à savoir : l'irréversibilité, la résistance à la falsification et la résistance à la collision.

On en utilise deux sur le protocole Bitcoin : HMAC[17] et PBKDF2[18].

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.

Dans Bitcoin on utilise HMAC-SHA512, c'est-à-dire l'algorithme HMAC intégrant la fonction de hachage SHA512[19].

Voici son schéma de fonctionnement général :



[17] HMAC : ***H**ash-based **M**essage **A**uthentication **C**ode.*
[18] PBKDF2 : ***P**assword **B**ased **K**ey **D**erivation **F**unction **2**.*
[19] SHA512 : Fonction de la famille des SHA-2 similaire à SHA256 dans ses mécanismes mais qui produit un condensat de 512 bits.

Étudions plus en détail ce qu'il se passe dans cette boîte noire HMAC-SHA512. Soit la fonction HMAC-SHA512 avec :

- `m` : message de taille arbitraire choisi par l'utilisateur (entrée 1).
- `K` : clé arbitraire choisie par l'utilisateur (entrée 2).
- `K'` : la clé `K` ajustée à la taille ***B*** des blocs.
- `SHA512` : La fonction de hachage SHA512.
- `⊕` (*XOR*) : Fonction ou exclusif.
- `||` : Opération de concaténation, c'est-à-dire de mettre les deux opérandes l'une à la suite de l'autre.
- *`opad`* : constante définie par l'octet `0x5c` répété ***B*** fois.
- *`ipad`* : constante définie par l'octet `0x36` répété ***B*** fois.
- ***B*** : La taille des morceaux de la fonction de hachage utilisée.

Avant de commencer, il existe une étape de prétraitement qui consiste à ajuster la taille de la clé et des constantes. Ces informations seront ajustées à la taille ***B*** qui est égale à la taille des morceaux (ou blocs) utilisés dans la fonction de hachage choisie.

Par exemple, si nous avions choisi d'utiliser la fonction SHA256 au sein de la fonction HMAC, nous aurions eu un ***B*** égal à 512 bits, soit 64 octets. Dans Bitcoin on utilise SHA512 dans la fonction HMAC, nous avons donc un ***B*** égal à 128 octets (1024 bits).

L'étape de prétraitement consiste premièrement à égaliser la clé de taille arbitraire choisie par l'utilisateur. Pour ce faire, si la clé est d'une taille inférieure à ***B***, on la rembourre avec des zéros à la fin de la chaîne de bits jusqu'à atteindre la taille ***B***. Si la clé est d'une taille supérieure à ***B***, on la hache avec la fonction de hachage choisie, puis on rembourre le condensat avec des zéros à la fin de la chaîne de bits jusqu'à atteindre la taille ***B***.

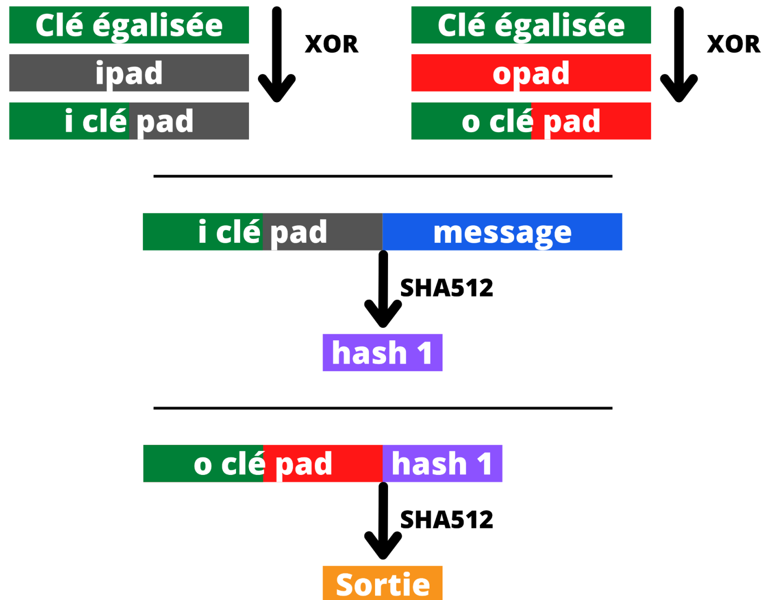
Pour les constantes *`opad`* et *`ipad`*, on va simplement répéter leur octet le nombre de fois nécessaire pour atteindre la taille ***B***. Par exemple, si nous avons un ***B*** égal à 128 octets, *`opad`* sera une concaténation de 128 `0x5c` :

```
opad = 0x5c5c5c5c5c5c[...]5c5c5c5c5c
→ 128 fois 5c
```

La fonction HMAC-SHA512 est alors définie par l'équation suivante :

```
HMAC-SHA512_K(m) =
  SHA512 ((K' ⊕ opad) || SHA512 ((K' ⊕ ipad) || m))
```

Schématiquement, cela représenterait cela :



Voici les étapes décomposées de cette fonction :

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

Cette fonction **HMAC** est utilisée dans 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**.

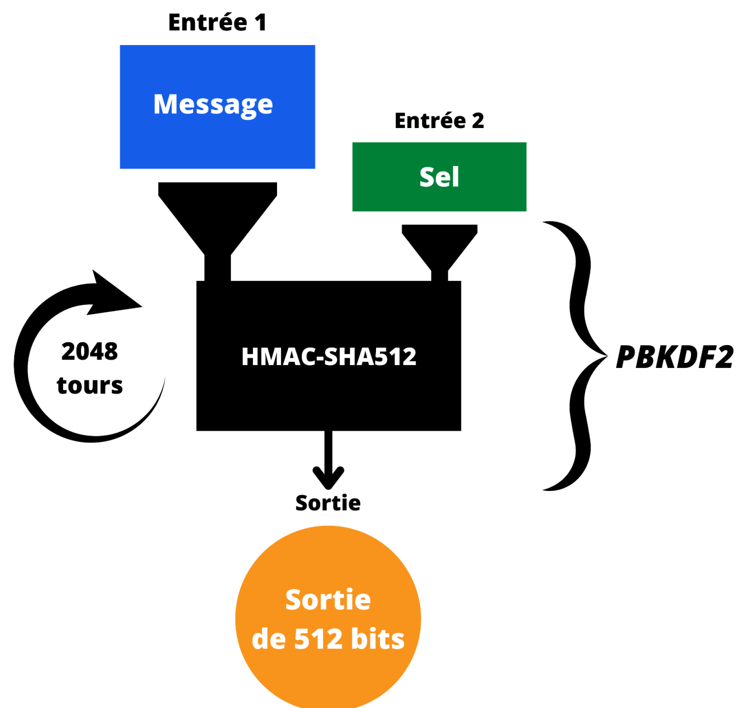
PBKDF2, pour ***Password-Based Key Derivation Function 2***, est une fonction de dérivation de clé. Cet algorithme va simplement appliquer une fonction choisie par l'utilisateur à un message de taille arbitraire avec un sel cryptographique, et il va répéter cette opération un certain nombre de fois pour sortir une clé comparable à un condensat.

Étant donné que cet algorithme utilise une fonction de hachage en son sein, il dispose en partie 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 dans Bitcoin et les algorithmes utilisés pour de la dérivation intégrant ces mêmes fonctions.

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 dès le premier tome afin de ne pas avoir à y revenir dans chaque tome suivant.

Dans la partie suivante, nous allons étudier et vulgariser la deuxième grande catégorie de méthodes cryptographiques utilisées dans le protocole Bitcoin : Les signatures numériques.

2- Les signatures numériques.

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

Le mot “portefeuille” (wallet) dans 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[20], qui sont 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 la propriété des UTXO 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 remplir des conditions de dépenses qui avaient été créées lors de la transaction précédente. Généralement, la condition de dépense consiste à prouver au reste du réseau que l’utilisateur est bien le propriétaire légitime de la clé publique associée à l’UTXO qu’il souhaite dépenser.

Il va donc devoir établir une preuve irréfutable qu’il connaît la clé privée associée à cette clé publique, sans pour autant dévoiler ladite clé privée au reste du réseau.

[20] UTXO : *Unspent **Transaction** **O**utput*. En français : sortie de transaction non-dépensée.

Pour ce faire, un utilisateur qui initie une transaction doit 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 doit 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 on utilise deux algorithmes de signature : **ECDSA**[21] et le Protocole de **Schnorr**[22].

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

Ces deux algorithmes sont assez similaires dans leurs mécanismes. Ils sont tous deux basés sur la cryptographie sur les courbes elliptiques. La différence majeure entre ces deux protocoles réside dans la structure de la signature.

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.

---

[21] ECDSA : ***E****lliptic* ***C****urve* ***D****igital* ***S****ignature* ***A****lgorithm*. En français : Algorithme de signature digitale sur courbe elliptique.

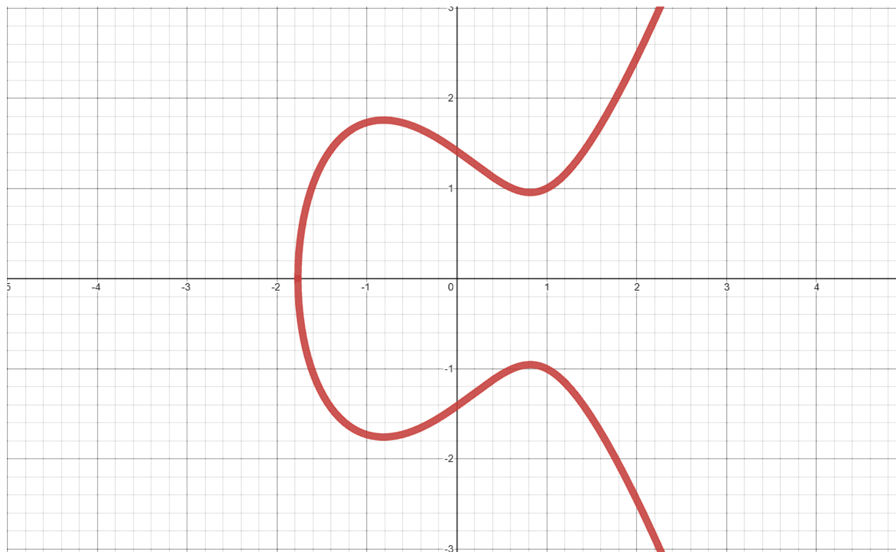
[22] Le nom du protocole de Schnorr vient de son inventeur Claus-Peter Schnorr.

La courbe elliptique.

La cryptographie sur les courbes elliptiques est un ensemble d'algorithmes qui utilisent une courbe elliptique pour ses différentes propriétés mathématiques et géométriques dans un objectif cryptographique, et dont la sécurité se base sur la difficulté du calcul du logarithme discret. Les courbes elliptiques sont notamment utilisées pour réaliser des échanges de clés, du chiffrement asymétrique, ou encore pour réaliser des signatures numériques.

Une des propriétés importantes de ces courbes est qu'elles sont 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 : $y^2 = x^3 + ax + b$.

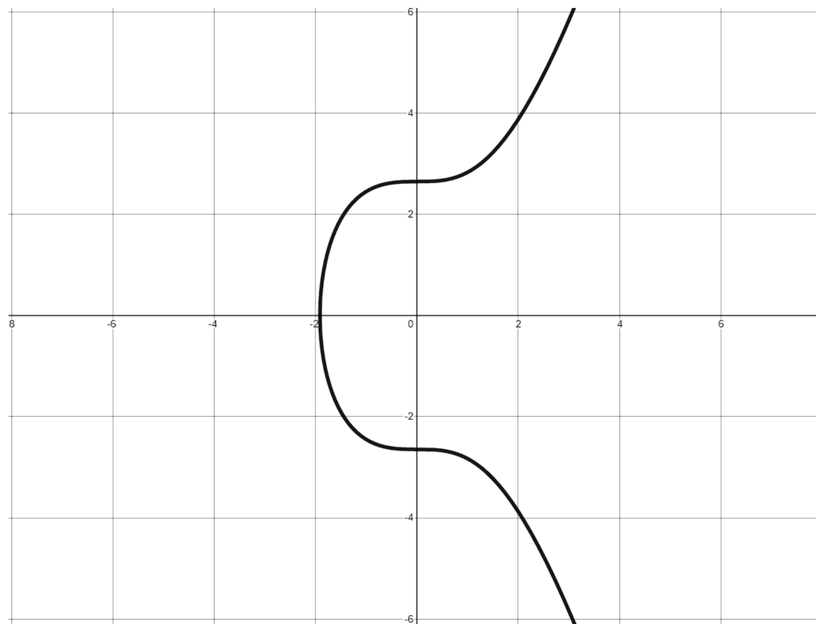
Pour utiliser ECDSA, il faudra donc choisir les paramètres de la courbe elliptique, c'est-à-dire les valeurs de `a` et de `b` dans l'équation de cette courbe.

Il existe ainsi différents standards de courbes elliptiques réputées cryptographiquement sûres. La plus connue est la courbe *secp256r1*, définie et conseillée par le NIST.

Malgré cela, Satoshi Nakamoto, l'inventeur de Bitcoin, a choisi de ne pas utiliser cette courbe. La raison de ce choix est inconnue, mais certains pensent qu'il a préféré trouver une alternative car les paramètres de cette courbe contiennent potentiellement une backdoor. À la place, le protocole Bitcoin utilise le standard ***secp256k1***[23].

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 $\mathbb{Z}\boldsymbol{p}$ 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.

[23] **SEC** désigne "***S**tandards for **E**fficient **C**ryptography*". **P256** désigne le fait que la courbe est définie sur un corps $\mathbb{Z}\boldsymbol{p}$ où ***p*** est un nombre premier de **256** bits. **K** désigne le nom de son inventeur Neal **K**oblitz et **1** désigne la première version.

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

2^{256} n'est pas un nombre premier, ECDSA utilise donc un nombre premier inférieur et relativement proche de ce nombre[24]. La représentation hexadécimale de ce nombre premier est :

```
p = 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFE FFFFFC2F
```

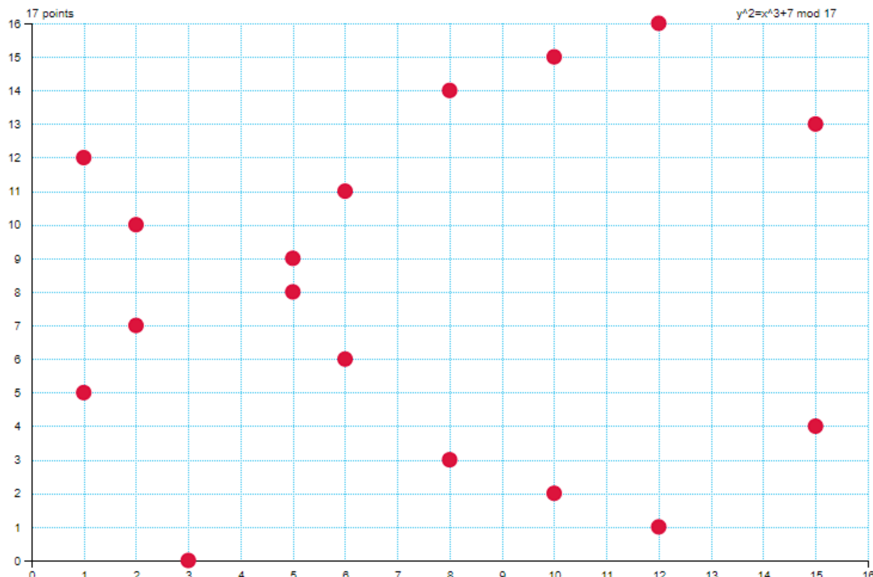
En format décimal, nous aurons :

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

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

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

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.

[24] Notons que ce n'est pas le nombre premier le plus proche existant. Il a été choisi pour différents facteurs.

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

On utilise un corps fini d'entiers positifs afin d'assurer la précision de la courbe. En effet, les courbes elliptiques sur le corps des réels sont imprécises. Si l'on effectue de nombreux ajouts de points, les erreurs d'arrondis vont s'accumuler à chaque étape et le résultat final n'aura aucun sens et sera difficilement reproductible. L'utilisation exclusive d'entiers positifs permet d'assurer une précision parfaite des ajouts de points et donc une reproductibilité du calcul.

L'utilisation de ce corps fini d'ordre premier est régie 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, mais infiniment plus grand.

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 est simplement un nombre pseudo-aléatoire. Dans le cas de Bitcoin, ce nombre est d'un taille 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 est un nombre de 256 bits compris entre `1` et `n-1`, en sachant que `n` est un nombre relativement plus petit que $\mathbf{2^{256}}$. Il existe donc certains nombres de 256 bits qui ne sont pas valides pour devenir clé privée dans Bitcoin.

Si la génération du nombre pseudo-aléatoire mène à une clé privée entre `n` et $\mathbf{2^{256}}$, celle-ci est considérée comme invalide et il faudra de nouveau réaliser la génération aléatoire.

Le nombre de possibilités pour une clé privée Bitcoin est donc presque de $\mathbf{1{,}158 * 10^{77}}$. 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 dans 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 dans 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 est appelée "`k`" minuscule.

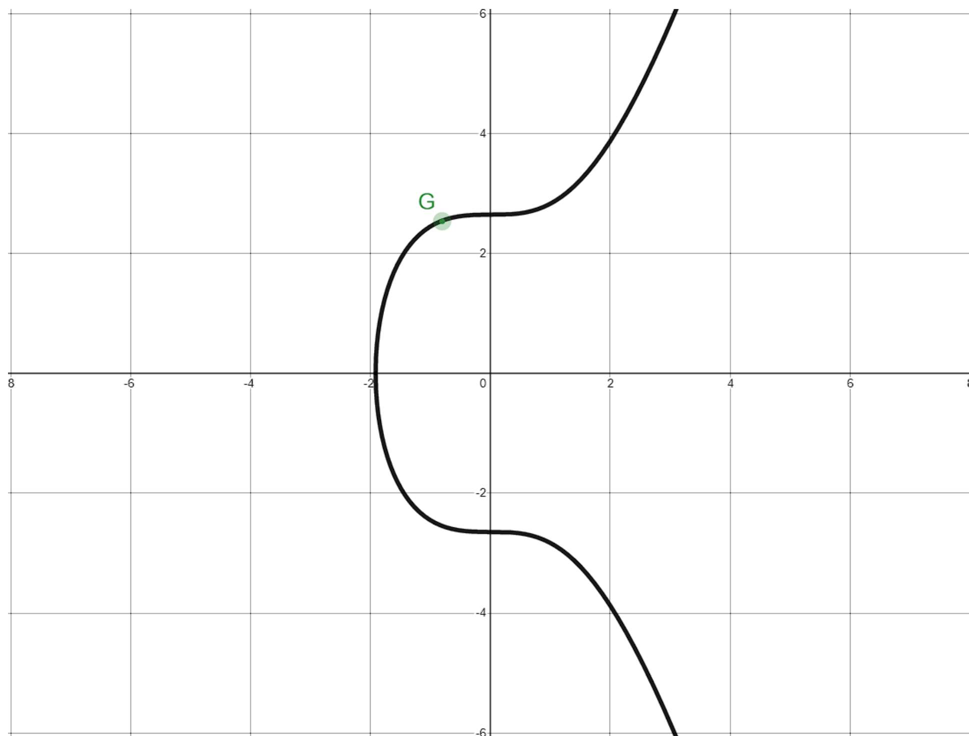
La clé publique.

La clé publique est un nombre de 512 bits[25] qui est généré à partir de la clé privée. Ce nombre est un point sur la courbe elliptique que nous nommons "`K`" majuscule.

➪ Comme nous le verrons dans le **tome 2**, une clé publique de 512 bits peut être compressée en un nombre de 264 bits conservant pourtant les mêmes informations. C'est ce que l'on appelle une clé publique compressée.

Pour calculer `K` nous allons utiliser l'addition et le doublement de points sur les courbes elliptiques[26] 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 toutes les parties prenantes, utilisé pour calculer l'intégralité des clés publiques de l'algorithme.

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.



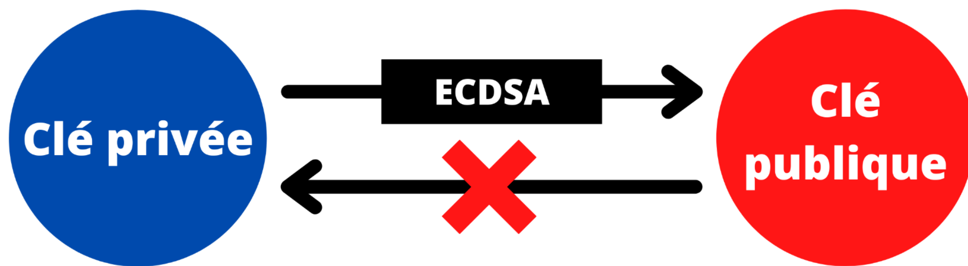
[25] 512 bits est la taille de la charge utile de la clé publique, c'est-à-dire les valeurs concaténées du ***x*** et du ***y*** indiquant le point sur la courbe elliptique. Si l'on ajoute le préfixe, une clé publique dispose alors d'une taille de 520 bits.
[26] Je vous explique ce concept dans la partie suivante.

La principale caractéristique de l'addition et du doublement de points sur les courbes elliptiques est qu'ils ne sont pas réversibles. Ce sont des "*trapdoor functions*".

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.

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.



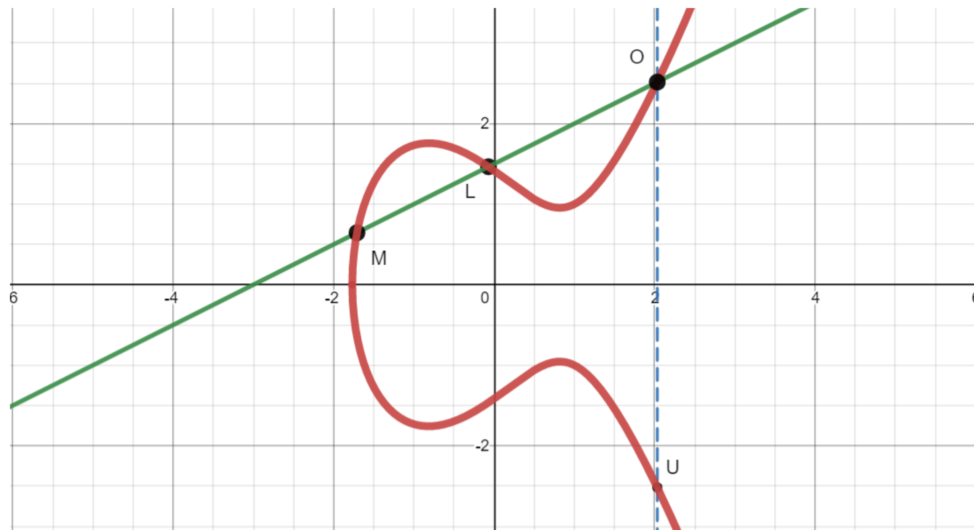
Addition et doublement de points.

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`.

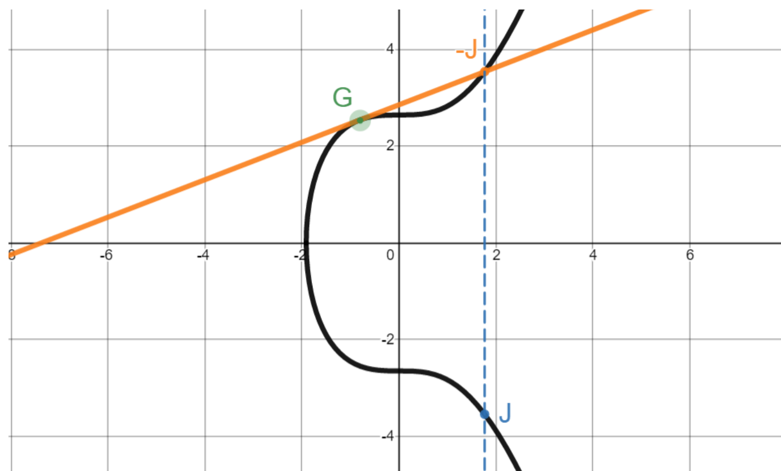
Nous aurons donc :

`➔ M + L = U`

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



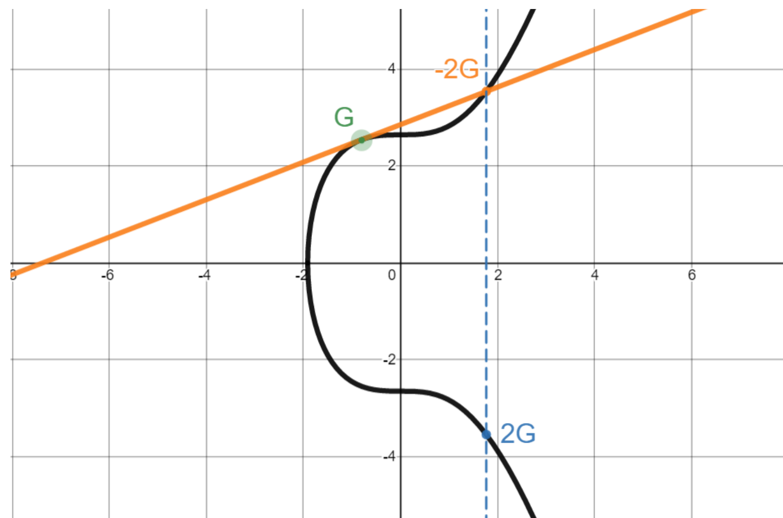
Maintenant si nous voulons ajouter un même point à lui même, c'est-à-dire réaliser un doublement de point, cela reviendrait à tracer la tangente à la courbe en ce point, et à récupérer l'opposé du point d'intersection entre la tangente et la courbe elliptique. Graphiquement, cela donnerait cela :



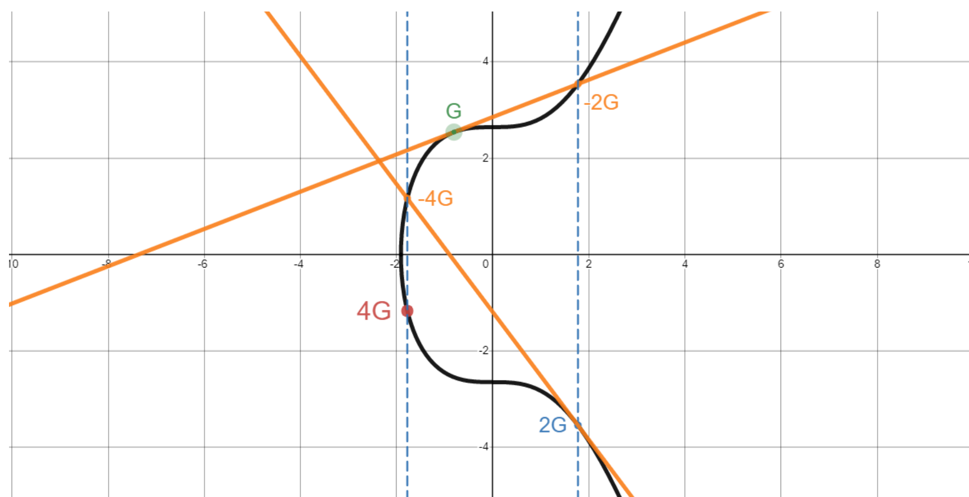
Dans cet exemple pour calculer le point `J` tel que `J = G + G`, nous avons simplement tracé la tangente à la courbe elliptique en le point `G` (droite orange). Cette tangente coupe une nouvelle fois la courbe elliptique en un point appelé ici `-J`. L'opposé de ce point nous donne notre résultat `J`.

Maintenant que nous savons faire l'addition sur les courbes elliptiques et le doublement à partir d'un point générateur (`G`), nous pouvons également réaliser le produit scalaire sur les courbes elliptiques. La produit scalaire d'un point par *n* revient à ajouter ce point à lui-même *n* fois.

Si nous reprenons notre exemple, calculer `G + G` revient à calculer `2G`. Nous pouvons renommer nos points en conséquence :

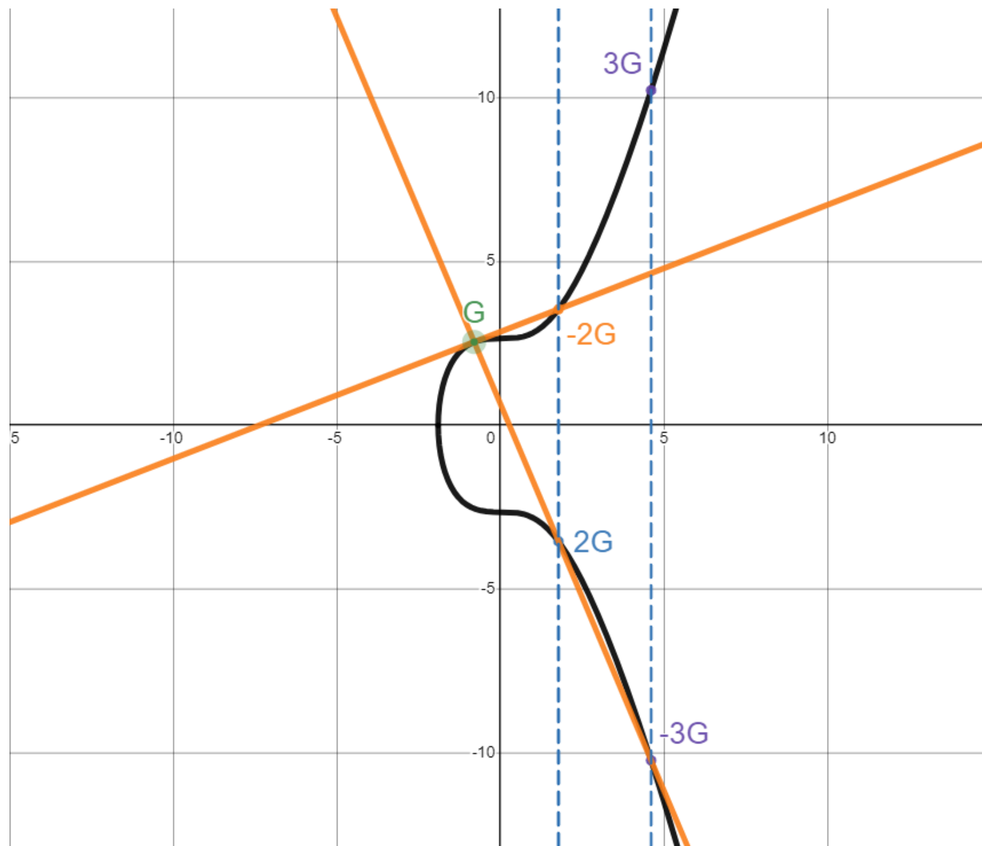


Nous pouvons ainsi continuer et calculer `4G` en traçant la tangente de la courbe elliptique en `2G` et en prenant le point opposé par rapport à l'axe des abscisses. On a alors effectué un doublement du point `2G` :



Si l'on souhaite par exemple calculer le point `3G`, nous devons d'abord calculer le point `2G` en doublant le point `G`, puis nous additionnons `G` et `2G`.

Graphiquement cela représenterait cela :



Pour additionner `G` et `2G` il suffit de tracer la droite reliant ces deux points (droite orange), de récupérer le point unique `-3G` à l'intersection entre cette droite et la courbe elliptique, et de déterminer `3G` tel que l'opposé à `-3G`.

Nous aurons donc :

- ➔ `G + G = 2G`
- ➔ `2G + G = 3G`

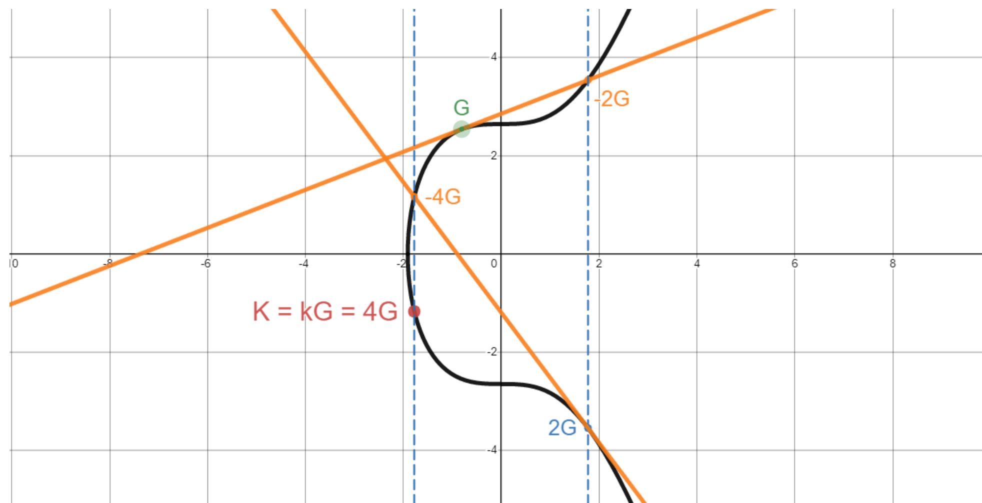
Fonction à sens unique.

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 `4`. Nous avons donc `k = 4`.

Pour calculer la clé publique associée à notre clé privée, nous réalisons l'opération `K = k · G`. Dans notre exemple, cela donne `K = 4 · G = 4G`.

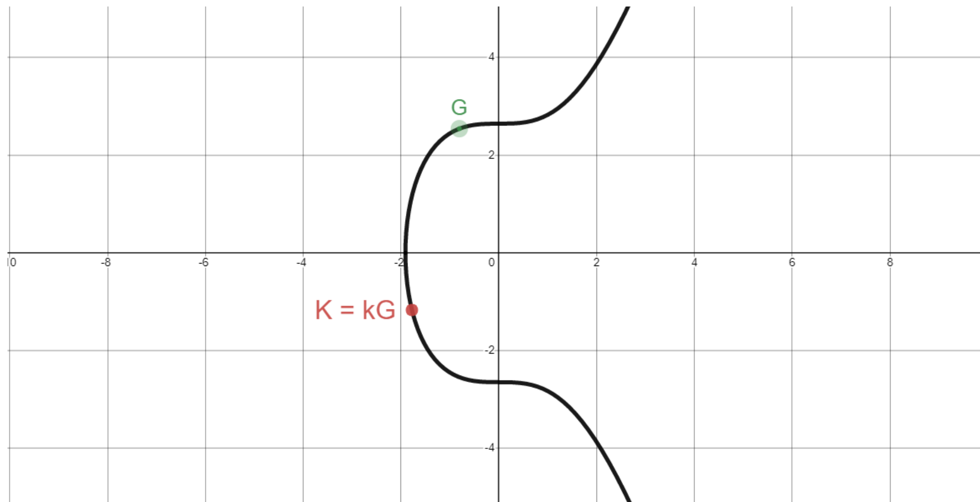
Graphiquement, cela ressemble à cela :



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 courbe, vous serez dans l'incapacité de calculer `k`, c'est-à-dire le nombre de fois que l'on a doublé `G` 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 en partie grâce à ce principe qu'il est impossible de déterminer une clé privée Bitcoin en connaissant uniquement sa clé publique.

Évidemment, 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 `4`. 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 en utilisant cette paire de clé comme condition de dépense. 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 utilisée. Dans le cas de Bitcoin, les paramètres de ***secp256k1*** sont :

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

```
p = 2^256 - 2^32 - 977

= 0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFF
FFFFFFFF FFFFFFFE FFFFFC2F
```

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

- La courbe $y^2 = x^3 + ax + b$ 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
```

Ce nombre est la forme compressée 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 = 0x FFFFFFFF 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 Bitcoin 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 des bitcoins. Ils sont bloqués sur sa paire de clés et demandent de produire une signature numérique à l'aide de la clé privée pour remplir leurs conditions de dépense. Alice 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, ni tout autre participant au réseau 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 est partagée en deux parties que nous nommons `S1` et `S2` tels que `S1` et `S2` concaténés donnent `S`, la signature numérique.

Voici comment créer ces deux valeurs :

`S1 :`

Pour trouver `S1` il va falloir générer un nombre pseudo-aléatoire que nous nommerons `v`. Ce nombre doit être un nonce[27] 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 le produit scalaire 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` est la valeur de `S1` recherchée :

```
V = v · G = (x,y)
S1 = x mod n
```

`S2 :`

Pour trouver `S2` Alice commence par réaliser un hash de sa transaction Bitcoin non-signée. Ce hash est 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 ) mod n
```

[27] Nonce (***N****umber* ***o****nly* ***u****sed* ***once***) : Nombre déterminé de façon arbitraire, généralement pseudo-aléatoire, destiné à être utilisé une seule fois puis changé.

Vérification de la signature.

Alice envoie donc sa transaction Bitcoin avec sa signature `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 changeront de conditions de dépenses en dépendant dorénavant de la paire de clés 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 sympathiques, 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 pseudo-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 du produit scalaire sur les courbes elliptiques expliqué 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 est `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 = v · G`. 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 le produit scalaire sur les courbes elliptiques.

Si `T = T'` alors Bob a la preuve 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 dans le calcul de la signature. 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 hachage et des signatures numériques.

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 hachage produisent une empreinte de taille fixe et 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 clés 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.

Outil 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://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf

Rapport FIPS 198-1 : *The Keyed-Hash Message Authentication Code (HMAC)* :
https://csrc.nist.gov/csrc/media/publications/fips/198/1/final/documents/fips-198-1_final.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

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

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

Email : loic@pandul.fr

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

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