Bitcoin démocratisé.

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

Loïc Morel

Pandul

---

2 Le portefeuille Bitcoin.

Pandul

Sommaire.

Remerciements.

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.

Dans ce **tome 2** nous allons entrer dans le vif du sujet. Nous allons étudier en profondeur le portefeuille Bitcoin ou wallet.

L'objectif va être de pouvoir visualiser comment se construit un portefeuille sur Bitcoin, comment il se décompose et à quoi servent les différentes informations qui le constituent. Cette compréhension des mécanismes du portefeuille vous permettra par la suite d'améliorer votre utilisation de Bitcoin en termes de sécurisation et de confidentialité.

Entropie, phrase mnémonique, graine, xpub, passphrase ou encore adresses, nous allons étudier et vulgariser tous ces concepts dont vous entendez parler mais dont vous ne comprenez pas forcément le fonctionnement technique.

Mais avant de commencer, je pense qu'il est important de rappeler ce qu'est un portefeuille Bitcoin et quelle est son utilité.

Comme je l'ai évoqué dans le **tome 1**, le mot "portefeuille" est finalement assez mal choisi. En effet, des bitcoins ne peuvent pas être stockés de la même façon que l'on stockerait des pièces dans un portefeuille en cuir. Au lieu de cela, les unités de compte bitcoin sont représentées sur le réseau par des UTXO (*Unspent Transaction Output*), qui sont tout simplement des morceaux de bitcoins.

Ces UTXO sont associés à un utilisateur à travers sa clé publique. Pour pouvoir dépenser ses bitcoins, l'utilisateur devra produire une signature à partir de la clé privée associée à cette clé publique. C'est ce que je décrivais dans le premier tome de la série.

Les informations qui doivent être conservées précieusement par l'utilisateur sont donc **les clés privées**.

Le portefeuille Bitcoin permet à l'utilisateur de conserver facilement ses clés privées en un seul lieu. Le fonctionnement ressemble donc plus à celui d'un porte-clés qu'à celui d'un portefeuille.

Les premiers portefeuilles utilisés sur Bitcoin étaient simplement des logiciels regroupant des clés privées, déterminées de manières pseudo-aléatoire, qui n'avaient aucun lien entre elles. C'est ce que l'on appelle un portefeuille JBOK (*Just a Bunch Of Keys*).

Ces premiers portefeuilles sont quelque peu laborieux à utiliser puisqu'ils obligent l'utilisateur à réaliser une nouvelle sauvegarde pour toute nouvelle paire de clés générée.

En 2012, Pieter Wuille publie le BIP32, une proposition visant à construire des portefeuilles Bitcoin déterministes hiérarchiques. L'idée derrière ce nouveau format de portefeuille est de ne plus déterminer aléatoirement les clés privées, mais de toutes les dériver de façon déterministe et hiérarchique depuis une information unique : **la graine** (*seed*).

Ainsi, l'utilisateur n'a plus besoin de réaliser une nouvelle sauvegarde pour toute nouvelle paire de clés générée, comme dans le cas des portefeuilles JBOK. Une unique sauvegarde de la graine permettra de retrouver de manière déterministe l'ensemble des paires de clés du portefeuille.

Le portefeuille HD BIP32 sera amélioré avec le BIP39 qui a introduit une façon standardisée d'encoder la graine afin de la rendre lisible plus aisément par un utilisateur et d'en faciliter la sauvegarde : C'est la fameuse phrase mnémonique, également nommée phrase de récupération ou encore phrase de 24 mots.

Aujourd'hui, l'extrême majorité des utilisateurs de Bitcoin disposent d'un portefeuille de type HD. C'est donc le format que l'on va étudier ici.

Gardez simplement à l'esprit que l'utilisation du protocole Bitcoin peut se faire autrement, avec d'autres standards de portefeuilles.

L'entropie.

Comme nous l'avons vu en introduction avec les portefeuilles JBOK, et comme je l'avais expliqué dans le **tome 1**, une clé privée Bitcoin doit être un nombre pseudo-aléatoire. Au plus ce nombre sera aléatoire et désordonné, au plus la paire de clé sera sécurisée.

En effet, nous avions vu ensemble que le nombre de clés privées possibles sur Bitcoin est presque de 2^{256}, un nombre extrêmement grand proche du nombre d'atomes dans l'univers observable. Le nombre de possibilités est tellement grand que si vous déterminez une clé privée de façon aléatoire, il est impossible qu'un autre utilisateur tombe sur la même clé ou trouve votre clé.

En revanche, si le caractère aléatoire n'est pas assez prononcé, il y aura statistiquement plus de chances que votre clé soit trouvée.

Le problème avec la génération aléatoire des clés privées, c'est que l'utilisateur devra réaliser une nouvelle sauvegarde pour toute nouvelle clé générée. Les portefeuilles HD ne génèrent donc plus les clés privées de façon aléatoire, mais de façon déterministe à partir d'une information unique, de telle sorte que l'utilisateur n'ait plus qu'une seule information à sauvegarder.

Pour conserver un caractère pseudo-aléatoire lors de la détermination des clés privées d'un portefeuille HD, l'information unique utilisée à la base du portefeuille sera un nombre aléatoire : c'est ce que l'on appelle l'entropie.

A la base de tout portefeuille Bitcoin HD se trouve donc une entropie, un nombre pseudo-aléatoire d'une taille au choix. Au plus ce nombre est grand et désordonné, au plus le portefeuille qui en découle est sécurisé.

Ce nombre est ensuite ramené à une taille standard, entre 128 bits et 256 bits.

⇨ La plupart du temps, ce nombre est généré directement par le logiciel qui héberge le portefeuille grâce à un PRNG (*Pseudo Random Number Generator*). Néanmoins, il est tout à fait possible de générer ce nombre soi-même puis de l'importer dans un logiciel par la suite. Cela permet d'avoir la main sur la génération de l'entropie et d'en maîtriser le caractère pseudo-aléatoire ainsi que la taille.

La taille de ce nombre déterminera par la suite la taille de la phrase mnémonique. Les standards les plus utilisés sont ainsi les entropies de 128 bits donnant une phrase mnémonique de 12 mots, et les entropies de 256 bits donnant une phrase mnémonique de 24 mots.

Une bonne façon de générer une entropie pseudo-aléatoire de 256 bits est de réaliser 256 lancés de dés afin de disposer d'une suite binaire de 256 caractères. J'ai rédigé un article vous détaillant étape par étape comment générer soi-même son entropie, et comment l'utiliser pour générer un portefeuille HD. Vous pouvez le retrouver en cliquant ici : Comment générer soi-même sa phrase mnémonique Bitcoin ?

Une autre façon de générer une bonne entropie est de déterminer aléatoirement un nombre très grand, supérieur à 256 bits, et de venir réduire ce nombre à 256 bits en utilisant la fonction de hachage SHA256. C'est le système que vous retrouvez notamment dans les Coldcard avec l'option "*Dice Roll*".

Maintenant que nous avons défini ce qu'est l'entropie à la base du portefeuille HD, voyons ensemble comment passer de ce nombre pseudo-aléatoire à une phrase mnémonique.

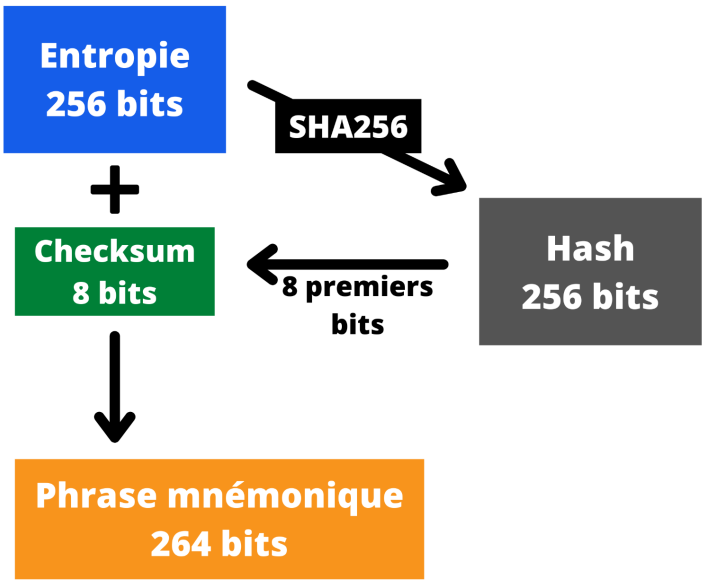
La phrase mnémonique.

Également nommée “seed phrase”, “phrase de récupération”, “24 mots” ou encore “phrase secrète”, la phrase mnémonique est un encodage de la graine du portefeuille permettant d’en faciliter la sauvegarde. Introduite en 2013 avec BIP39, elle représente aujourd’hui un standard utilisé pour l'extrême majorité des portefeuilles Bitcoin.

Pour passer d’une entropie à une phrase mnémonique, il suffit de calculer la checksum de l’entropie, et de concaténer entropie et checksum.

La checksum (ou “somme de contrôle” en français) représente le début du hash de l’entropie utilisant SHA256. Pour une entropie de 256 bits, la checksum sera de 8 bits.

On va donc passer notre entropie dans la fonction de hachage SHA256 pour en obtenir un hash. On va récupérer les 8 premiers bits de ce hash. Et enfin, on va mettre bout à bout l’entropie de 256 bits et la checksum de 8 bits, ce qui nous donne une phrase mnémonique de 264 bits.



La taille de la checksum et de la phrase mnémonique dépendent de la taille de l’entropie (`ENT`) initiale. La taille de la checksum (`CS`) est déterminée en divisant la taille de l’entropie par 32.

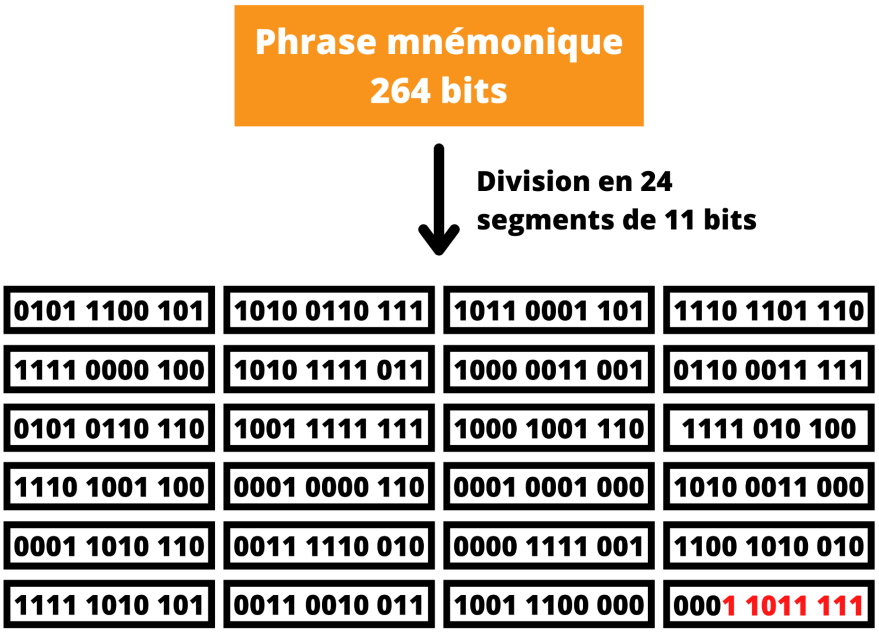
Par exemple, pour une entropie de 256 bits :

```
CS = ENT / 32
CS = 256 / 32 = 8
```

Voici un tableau de correspondance entre la taille de l'entropie et la taille de la phrase qui en découle :

Entropie (bits)	Checksum (bits)	ENT || CS (bits)	Phrase (mots)
128	4	132	12
160	5	165	15
192	6	198	18
224	7	231	21
256	**8**	**264**	**24**

Pour passer de la concaténation `ENT || CS` à une phrase de 24 mots, nous allons diviser `ENT || CS` en 24 segments de 11 bits chacun.



Vous remarquerez sur ce schéma que les **8 derniers bits** en rouge correspondent à la checksum.

Dans notre exemple, avec `ENT || CS` d'une taille de 264 bits, nous disposerons de 24 segments de 11 bits. Chaque segment va permettre de déterminer le rang d'un mot au sein d'[une liste de 2048 mots décrite dans BIP39].

Par exemple, si mon premier segment de 11 bits est `00101101110`. En base 10 cela donnera `366`. Comme la liste de mots est numérotée de 1 à 2048, il y a un décalage d'un rang avec ce segment pouvant aller de 0 à 2047. J'additionne donc `1` ce qui donne `367`.

Je n'ai plus qu'à récupérer le mot n°367 sur la liste, qui est en l'occurrence : `column`.

Nous procéderons de même pour les 23 autres segments de bits afin de disposer de notre phrase de 24 mots.

⇨ Une des caractéristiques de cette liste de mots est qu'aucun mot ne dispose des mêmes quatre premières lettres dans le même ordre.

Ainsi, il suffit de noter les quatre premières lettres de chaque mot de notre phrase mnémonique pour disposer d'une sauvegarde fonctionnelle. Cela permet de gagner de la place, notamment si vous utilisez un support en métal.

La passphrase BIP38/BIP39.

La passphrase est un sel cryptographique optionnel d'une taille arbitraire.

Elle permet d'améliorer la sécurité d'un portefeuille HD en ajoutant un mot de passe libre qui, une fois aggloméré à la phrase mnémonique, permettra de calculer la graine.

La passphrase est parfois également nommée : "two-factor seed phrase", "password", "seed extension", "extention word" ou encore "13ème ou 25ème mot".

La passphrase est un mot de passe déterminé librement par l'utilisateur sans lequel il est impossible de dériver les clés d'un portefeuille. Elle est proposée pour la première fois sur Bitcoin avec le BIP38. Dans cette amélioration, la passphrase permet alors simplement de sécuriser une clé privée.

Avec BIP39 et l'arrivée des phrases mnémoniques, elle est reprise et adaptée au format du portefeuille HD. Ainsi, elle ne sert non plus à sécuriser seulement une clé privée, mais bien l'intégralité d'un portefeuille.

Parfois appelée à tort 25ème mot, elle ne fait pas partie de la phrase mnémonique. Contrairement à celle-ci, la passphrase est une donnée libre, qui ne sera généralement pas stockée sur le logiciel qui stocke la phrase mnémonique.

Elle permet à l'utilisateur d'un portefeuille HD d'ajouter une couche supplémentaire de sécurité. Un attaquant en possession de la phrase mnémonique d'un portefeuille, ou du matériel qui héberge la phrase mnémonique, ne sera pas en capacité de dériver les paires de clés (et donc d'accéder aux bitcoins) tant qu'il n'aura pas trouvé la passphrase. Cela ajoute donc une garantie supplémentaire en cas de vol du hardware wallet ou de la phrase mnémonique.

En revanche, cette passphrase est à double tranchant. Si l'utilisateur lui-même oublie sa passphrase, il ne pourra pas retrouver l'accès à ses bitcoins.

Nous verrons dans la partie suivante comment cette information s'inclut dans le processus de dérivation d'un portefeuille HD.

La graine.

Une fois la phrase mnémonique obtenue, nous allons pouvoir commencer la dérivation d'un portefeuille Bitcoin. La prochaine étape de dérivation est la **graine** ou **seed** en anglais.

Son standard BIP39 la décrit comme une suite alphanumérique de 512 bits qui constitue la base d'un portefeuille HD (déterministe hiérarchique). A partir de la graine, il sera possible de dériver l'intégralité des paires de clés d'un portefeuille Bitcoin.

Quelle que soit la taille de votre phrase mnémonique, la taille de votre graine sera toujours de 512 bits si vous respectez le standard BIP39.

Pour dériver une graine depuis une phrase mnémonique, on utilise la fonction PBKDF2 : *Password-Based Key Derivation Function 2*.

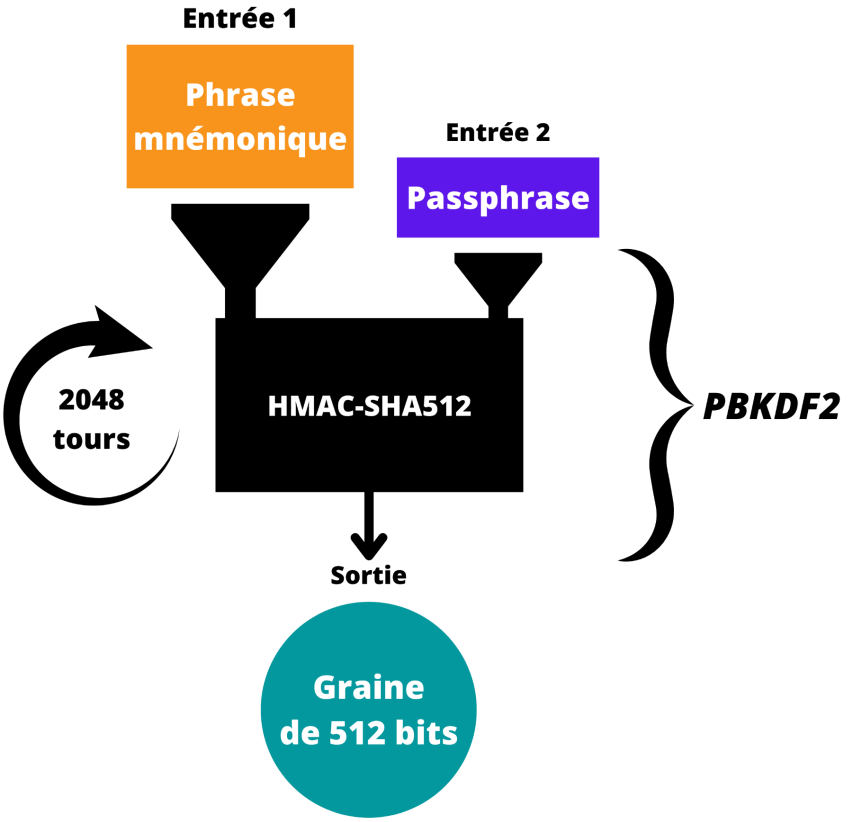
Comme expliqué dans le **tome 1**, c'est un algorithme de dérivation de clé qui applique une fonction choisie par l'utilisateur à un message de taille arbitraire avec un sel cryptographique, et répète l'opération plusieurs fois.

Sur le protocole Bitcoin, les paramètres de cette fonction seront :

- Pour le message nous prendrons la **phrase mnémonique**.
- Pour le sel nous prendrons la **passphrase**.
- La fonction choisie est **HMAC-SHA512**.
- Le nombre d'itération est de **2048**.

Si le portefeuille en question ne dispose pas de passphrase, le champ du sel sera laissé vide.

Voici la représentation schématique du fonctionnement de PBKDF2 lors de la dérivation de la graine d'un portefeuille HD :



⇨ Pour rappel, HMAC est un algorithme d'authentification de message, utilisé ici avec la fonction de hachage SHA512. Je décris leurs fonctionnements en détail dans le **tome 1**.

Une fois que le portefeuille dispose de la graine (*seed*), nous allons pouvoir commencer la dérivation des clés.

La clé maîtresse.

Après avoir dérivé la graine, la prochaine étape dans la dérivation d'un portefeuille HD va être de déterminer la clé privée maîtresse et le code de chaîne maître.

Pour obtenir ces deux informations essentielles au processus de dérivation des paires de clés, il faut appliquer la fonction HMAC-SHA512 à la graine. Le condensat de cette opération sera un nombre de 512 bits.

Pour rappel, la fonction HMAC-SHA512 utilise en entrée un message et une clé. Dans la cas de la dérivation de la clé maîtresse, ces deux informations seront :

- Message : les 512 bits de la graine.
- Clé : identique pour tout le monde, les deux mots "*Bitcoin Seed*".

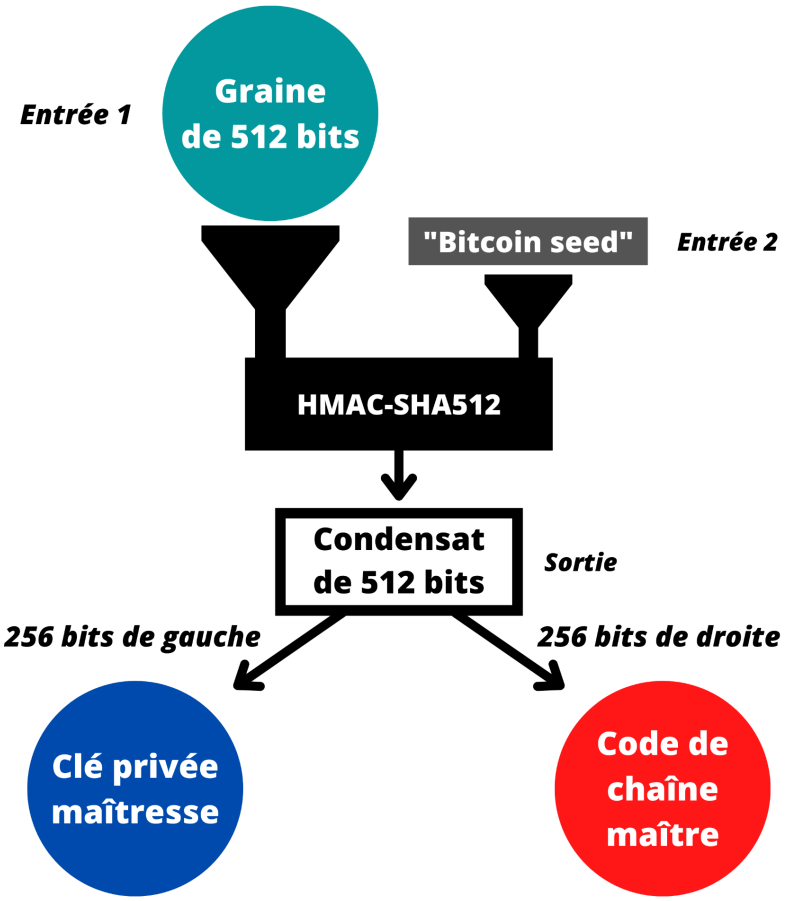
La raison pour laquelle la clé est identique pour chaque utilisateur est que HMAC-SHA512 nécessite forcément une deuxième entrée. Les développeurs n'ont pas utilisé simplement SHA512 car la fonction n'était pas implémentée sur Bitcoin contrairement à HMAC-SHA512. De plus, l'ajout de la clé "*Bitcoin seed*" permet de générer une clé maîtresse unique à Bitcoin plutôt que d'utiliser n'importe quel ancien hash SHA512.

Le condensat de 512 bits en sortie sera séparé en deux :

- Les 256 bits de gauche donneront la clé privée maîtresse du portefeuille.
- Les 256 bits de droite donneront le code de chaîne maître.

⇨ En réalité, on appliquera le format $\boldsymbol{parse_{256}}$ aux 256 bits de gauche pour obtenir la clé privée maîtresse. On interprète cette séquence comme un nombre de 256 bits, l'octet le plus significatif en premier.

Voici la représentation schématique de la génération de la clé privée maîtresse et du code de chaîne maître, à partir de la graine :



La clé maîtresse d'un portefeuille est en quelque sorte la clé parent de toutes les autres clés enfants obtenues par dérivation. Elle représente la profondeur zéro du portefeuille HD, c'est-à-dire l'information de base.

Le code de chaîne intervient dans la dérivation des clés enfants. Il est impossible de dériver des clés sans en avoir la connaissance. Il permet d'introduire une source d'entropie dans le processus de dérivation.

A partir de ces deux informations que sont la **clé privée maîtresse** et le **code de chaîne maître**, nous allons pouvoir dériver des clés enfants.

Chaque paire enfant se décompose en trois partie :

- La clé privée.
- La clé publique.
- Le code de chaîne.

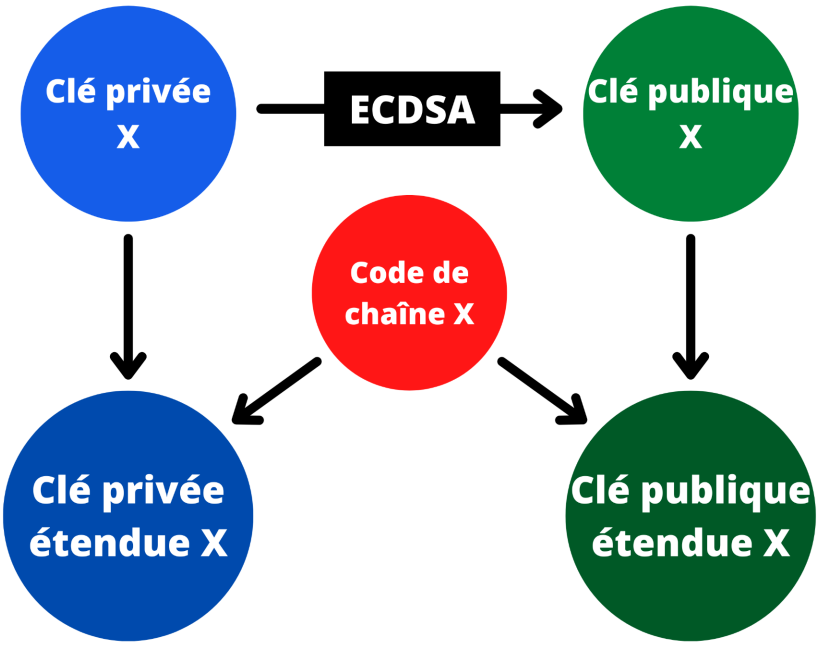
Les clés étendues.

Souvent confondues avec la clé maîtresse, les clés étendues, également nommées clés extensibles, sont pourtant bien différentes de celle-ci.

En théorie, une clé étendue représente simplement la concaténation de n'importe quelle clé (publique ou privée) et de son code de chaîne associé. En réalité, le terme de clés étendues est aujourd'hui utilisé seulement pour désigner la xpub et la xprv : la paire parent d'un compte.

Comme expliqué précédemment, une clé privée parent seule ou une clé publique parent seule ne permettent pas de dériver les clés enfants descendantes de cette paire. Pour ce faire, il faut nécessairement disposer du code de chaîne associé à la paire de clé parent en question.

Étant donné que les clés étendues agrègent ces deux informations (la clé et le code de chaîne), elles permettent donc de rassembler en une seule suite l'intégralité des informations nécessaires pour dériver des clés enfants.



Lorsqu'elles sont mises en forme, ces clés étendues disposent d'un préfixe qui permet de les différencier. Ainsi les clés privées étendues commencent toujours par : `xprv`, `yprv` ou `zprv`. Et les clés publiques étendues commencent toujours par : **`xpub`**, `ypub` ou `zpub`.

La clé privée étendue permet de dériver l'intégralité des clés privées enfants du compte, et donc par multiplication sur les courbes elliptiques, l'intégralité des clés publiques enfants.

La clé publique étendue **`xpub`** permet de dériver l'intégralité des clés publiques enfants du compte, sauf les clés publiques enfants endurcies (voir partie suivante).

Contrairement à la clé privée étendue, la clé publique étendue ne permettra pas d'accéder aux clés privées enfants.

Une personne en possession d'une **`xpub`** d'un compte sur un portefeuille HD pourra donc observer toutes les clés publiques enfants du compte (sauf les endurcies). Il pourra également voir les fonds qui y transitent et dériver des adresses, mais il ne pourra jamais débloquer les bitcoins associés à ces clés publiques.

⇨ Le lien mathématique irréversible entre clé privée et clé publique est expliqué en détail dans le **tome 1** de la série.

Pour résumer : On utilise la multiplication sur les courbes elliptiques pour dériver une clé publique unique depuis une clé privée. Cette multiplication est une fonction à sens unique, ce qui veut dire qu'il est facile de déterminer une clé publique en sachant sa clé privée mais qu'il est impossible de faire l'inverse.

Pour obtenir la clé privée étendue d'un compte, il faut concaténer la clé privée choisie pour être parent, et son code de chaîne associé. Cela nous donne une suite de 512 bits.

Pour obtenir la clé publique étendue d'un compte, il faut concaténer la clé publique choisie pour être parent, et son code de chaîne associé. Cela nous donne une suite de 520 bits.

On a donc une clé privée étendue représentée par **`k || c`**, où **`k`** est la clé privée choisie, et **`c`** est le code de chaîne associé.

On a également une clé publique étendue représentée par **`K || c`**, où **`c`** est le même code de chaîne et où **`K = point(k)`**.

La fonction **`point`** représente la dérivation de clé publique sur les courbes elliptiques.

En théorie, seuls la clé (publique ou privée) et le code de chaîne sont nécessaires pour dériver les clés enfants. En réalité, d'autres informations vont être ajoutées aux clés étendues afin d'identifier la clé parent et d'indiquer sa place dans la hiérarchie du portefeuille.

Premièrement, un préfixe de 1 octet va être ajouté à la clé privée afin qu'elle soit d'une taille similaire à la clé publique : 33 octets (pour rappel : 1 octet = 8 bits).

Les autres informations qui composeront une clé étendue seront :

Version	Permet de placer "xprv" ou "xpub" au début de la clé étendue.	4 octets
Profondeur	Nombre de dérivations par rapport à la clé maîtresse (je vous explique à quoi cela correspond dans la partie suivante).	1 octet
Empreinte parent	Quatre premiers octets du HASH160 de la clé parent : `HASH160 (p) = RIPEMD160(SHA256(p))`	4 octets
Numéro index	Numéro de cette paire enfant parmi ses sœurs.	4 octets
Code de chaîne	Code de chaîne (voir partie précédente).	32 octets
Clé	La clé privée + un préfixe de 1 octet, ou la clé publique seule. La clé publique étant plus longue que la clé privée de l'ordre de 1 octet, l'octet supplémentaire ajouté à la clé privée permet d'égaliser sa taille. Cet octet supplémentaire est `0x00`.	33 octets
Somme de contrôle	Checksum des lignes précédentes utilisant HASH256 (double SHA256).	4 octets

Au total, une clé étendue fait donc 78 octets sans la checksum, et 82 octets avec la checksum. Une fois la checksum ajoutée, on convertit ces informations en Base58 afin de disposer d'un format lisible plus aisément par un utilisateur.

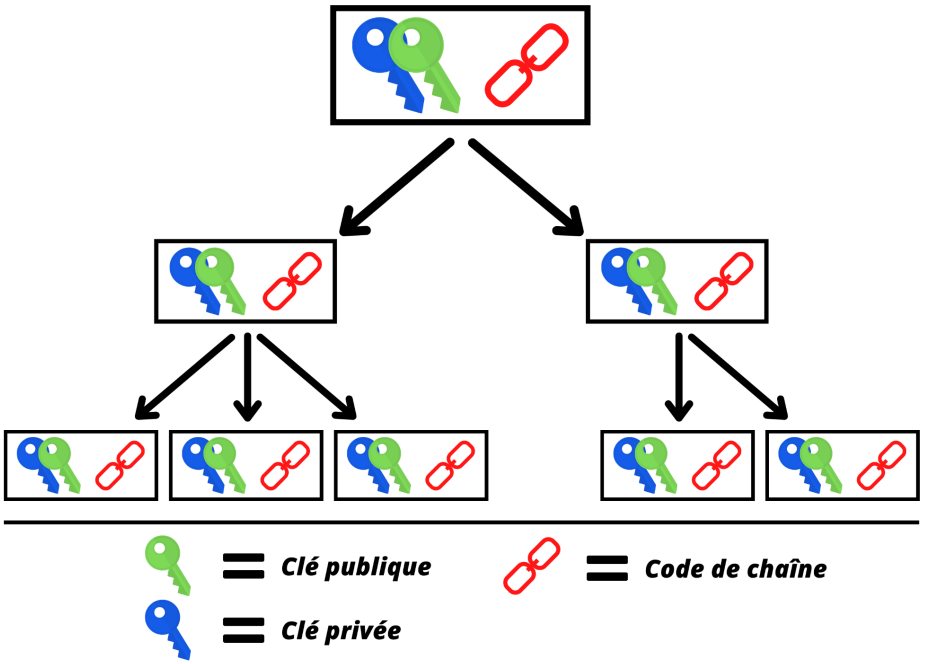
Voici à quoi ressemble une clé publique étendue, avec les informations supplémentaires décrites ci-dessus, et les couleurs correspondantes :

```
En HEX (base 16) :

0488B21E036D5601AD80000000C605DF9FBD77FD6965BD02B77831E
C5C78646AD3ACA14DC3984186F72633A89303772CCB99F4EF346078
D167065404EED8A58787DED31BFA479244824DF50658051F067C3A


En Base 58 :

xpub6CTNzMUkzpurBWaT4HQoYzLP4uBbGJuWY358Rj7rauiw4rMHCyq
3Rfy9w4kyJXJzeFfyrKLUar2rUCukSiDQFa7roTwzjiAhyQAdPLEjqH
T
```

Dérivation de paires de clés.

A partir de la clé maîtresse et du code de chaîne maître, nous allons être capables de dériver un certain nombre de paires de clés enfants. Ces clés enfants pourront soit être utilisées comme telles pour des échanges de bitcoins, soit servir à leur tour de clés parents ce qui donnera des paires de clé petits-enfants.

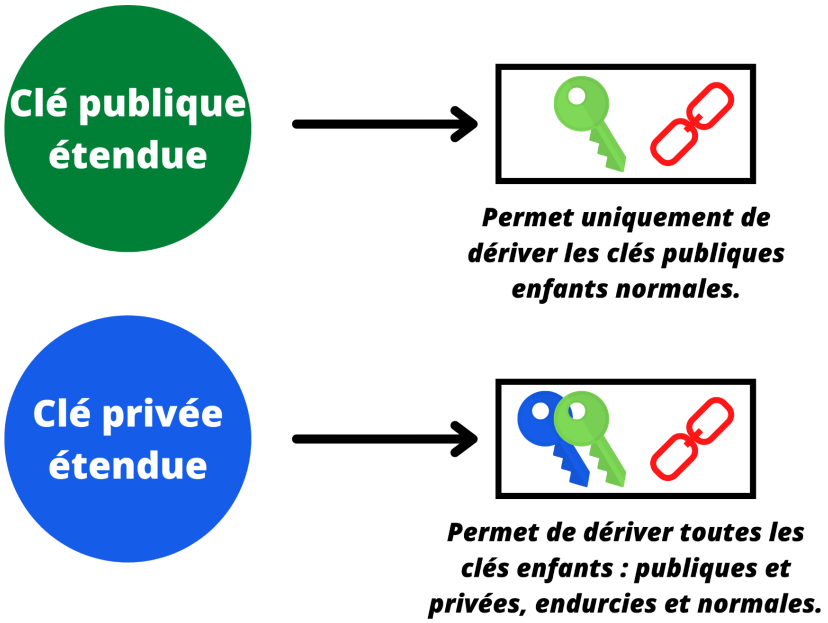
Ce fonctionnement hiérarchique permet de segmenter des branches et d'offrir de nombreuses options de gestion des clés et de structures de portefeuille.



Il existe alors deux types de paires de clés enfants : les clés enfants endurcies (ou renforcées) et les clés enfants non endurcies (également nommées clés enfants normales).

Les clés publiques enfants normales peuvent être dérivées soit à partir de la clé publique étendue parent, soit à partir de clé privée étendue parent.

En revanche, les clés enfants endurcies et les clés privées enfants normales ne peuvent être dérivées que depuis la clé privée étendue parent. En conséquence, il sera impossible de tracer ces clés endurcies à partir de la clé publique étendue parent, contrairement aux clés normales.



Chaque paire de clés étendues dispose de 2^{31} paires de clés enfants normales et de 2^{31} paires de clés enfants endurcies. Chacune de ces paires dispose d'un numéro d'indice (index), un nombre entier de 32 bits permettant de différencier chaque paire de clés de ses autres sœurs.

Les paires de clés enfants normales disposent d'un **index** compris entre 0 et $2^{31}-1$. Et les paires de clés enfants endurcies disposent d'un index compris entre 2^{31} et $2^{32}-1$.

Le processus de dérivation des clés enfants fait de nouveau intervenir la fonction HMAC-SHA512. Pour rappel, cet algorithme applique une fonction de hachage à un message et à une clé en entrée, pour donner un condensat de 512 bits en sortie.

⇨ Attention ! La "clé" utilisée dans la fonction HMAC n'a rien à voir avec les paires de clés (publiques et privées) dont nous parlions jusqu'ici. Dans l'utilisation de HMAC, le mot "clé" désigne simplement une variable utilisée en input de la fonction.

Dans le cas de la dérivation de clés enfants, la "clé" de la fonction HMAC sera le code de chaîne parent, et le message sera une concaténation de la clé parent et de l'index choisi.

Voici ci-dessous les différents processus de dérivation en fonction des catégories de clés enfants souhaitées, et du type de clé parent en entrée.

Pour chaque processus :

```
→ C_par = Code de chaîne parent
→ k_par = Clé privée parent
→ K_par = Clé publique parent

→ C_enf = Code de chaîne enfant
→ k_enf = Clé privée enfant
→ K_enf = Clé publique enfant

→ h = Hash résultant de la fonction HMAC-SHA512
→ H = Résultat de h * G, où G est le point générateur
      (multiplication sur les courbes elliptiques).

→ n = Ordre de la courbe elliptique sachant le point
d'origine (voir tome 1).
```

❖ Clé privée parent → clé privée enfant.

Nous allons dériver k_{enf} et C_{enf} à partir de k_{par} et de C_{par}.

Premièrement on choisit un nombre `i` de 32 bits pour l'index de la clé.
Si `i` $< 2^{31}$, alors nous calculerons une clé normale. Si `i` $\geq 2^{31}$, alors nous calculerons une clé endurcie.

- Pour k_{enf} endurcie, la formule sera :

```
h = HMAC-SHA512 (Clé = C_par, Message = 0X00 || k_par || i )
```

⇨ Pour rappel, `0X00` est ici le préfixe permettant d'amener la clé privée à la même taille que la clé publique, à savoir 264 bits ou 33 octets.

`||` est le symbole pour une concaténation, cela consiste simplement à mettre bout à bout deux opérandes.

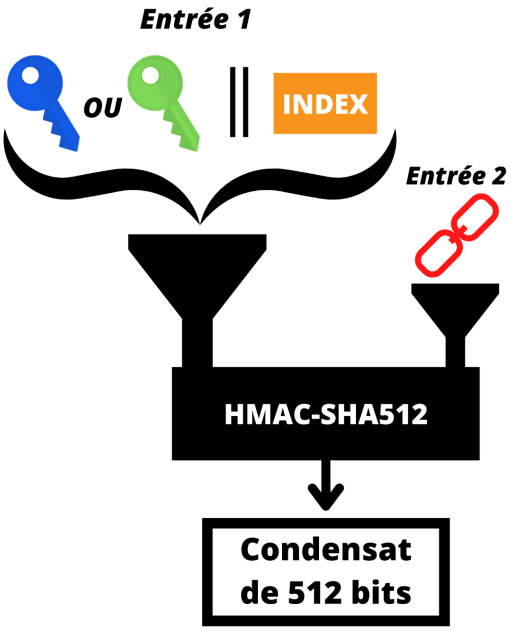
- Pour k_{enf} normale la formule sera :

$$h = \text{HMAC-SHA512 (Clé = } C_{par}\text{, Message = point}(k_{par}) \text{ || } i \text{)}$$

⇨ Pour rappel, la fonction `point(p)` permet de calculer le point `P` (la clé publique de `p`) sur la courbe elliptique tel que `P = G * p`

On a donc : `point(p) = P`

Schématiquement, ces opérations ressemblent à cela :



Deuxièmement, on récupère h (le condensat en sortie de la fonction HMAC) et on le sépare en deux séquences de 32 octets :

- les 32 premiers octets seront h_1,
- les 32 derniers octets seront h_2.

La clé privée enfant k_{enf} sera alors égale à :

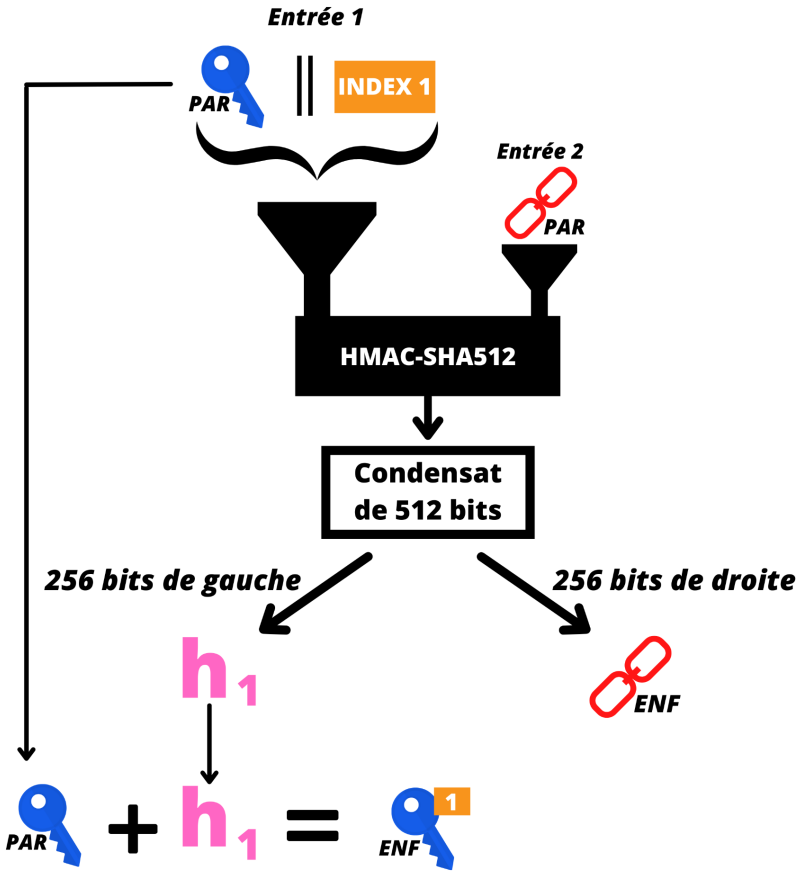
$$k_{enf} = parse_{256}(h_1) + k_{par} \ [mod\ n]$$

Dans la formule précédente, nous avons additionné h_1 mis en forme $parse_{256}$, et k_{par}. Pour maintenir cette nouvelle clé privée enfant dans le rang de notre courbe elliptique, on la module avec `n`.

Le code de chaîne enfant sera : $c_{enf} = h_2$.

Si $parse_{256}(h_1) \geq n$ ou si $k_{enf} = 0$, alors la clé enfant est invalide. Il faudra alors réessayer l'opération avec le prochain `index`.

Ce schéma illustre la dérivation d'une clé privée enfant endurcie :



Pour disposer d'une clé privée enfant normale, le schéma serait similaire mis à part que la **clé privée parent** (bleu) en entrée 1 serait remplacée par la **clé publique parent** (verte).

❖ Clé publique parent → clé publique enfant.

Nous allons dériver K_{enf} et C_{enf} à partir de K_{par} et de C_{par}.

> ⇨ Pour rappel, “k” minuscule désigne une clé privée et “K” majuscule désigne la clé publique associée.

Premièrement on choisit un nombre `i` de 32 bits pour l’index de la clé.

A partir d’une clé publique parent, nous ne pourrons calculer qu’une clé publique enfant normale (pas endurcie) il faut donc que : $i < 2^{31}$

Ensuite on applique la fonction HMAC :

$$h = \text{HMAC-SHA512 (Clé} = C_{par}, \text{ Message} = K_{par} \,||\, i \text{)}$$

Puis on récupère h (le condensat en sortie de la fonction HMAC) et on le sépare en deux séquences de 32 octets :

- les 32 premiers octets seront h_1,
- les 32 derniers octets seront h_2.

La clé publique enfant K_{enf} sera :

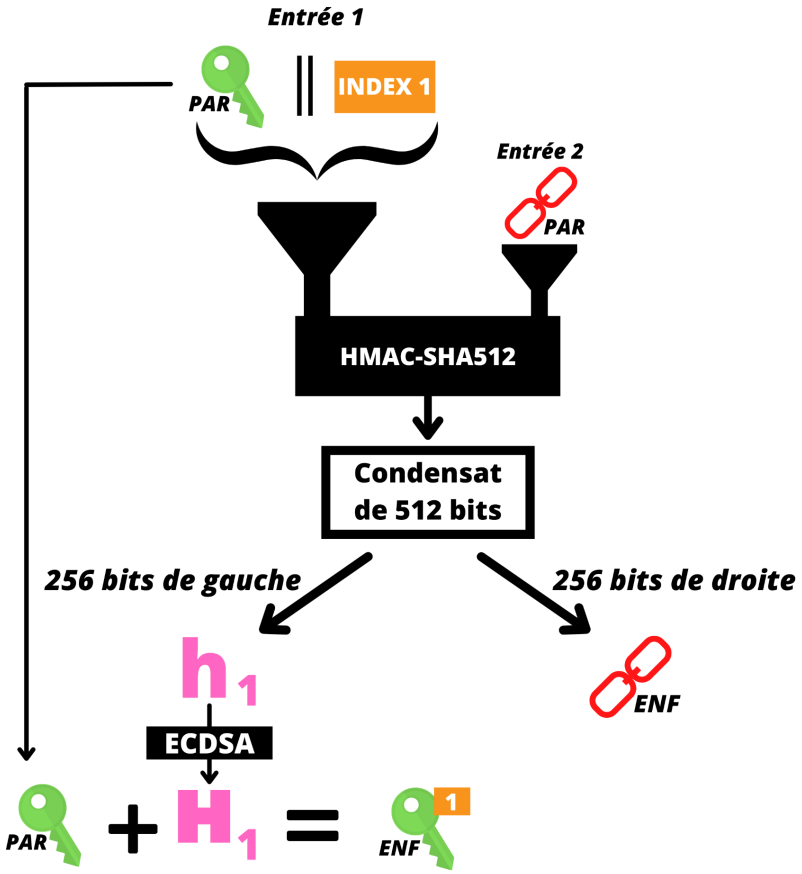
$$K_{enf} = \text{point}(parse_{256}(h_1)) + K_{par}$$

Le code de chaîne enfant sera :

$$C_{enf} = h_2$$

Si $parse_{256}(h_1) \geq n$, ou si K_{enf} est le point à l’infini, alors la clé enfant est invalide. Il faudra alors réessayer avec le prochain `index`.

Ci-dessous, le schéma de la dérivation d'une clé publique enfant à partir de la clé publique parent. En admettant que : `point(`$parse_{256}(h_1)$`) = `H_1



❖ Clé privée parent → clé publique enfant.

Nous allons dériver K_{enf} et C_{enf} à partir de k_{par} et de C_{par}. Deux méthodes différentes se présentent à nous :

- Soit on calcule la clé publique parent K_{par} à partir de k_{par} en utilisant la multiplication sur les courbes elliptiques. Puis, nous pourrons ensuite dériver K_{enf} comme dans la partie "**Clé publique parent → clé publique enfant**".

  Cette méthode ne fonctionne que pour avoir une clé publique enfant normale (non endurcie).

- Soit on dérive k_{enf} à partir de k_{par} comme dans la partie "**Clé privée parent → clé privée enfant**". Puis, nous pourrons ensuite calculer K_{enf} à partir de k_{enf} en utilisant la multiplication sur les courbes elliptiques.

  Cette méthode ne fonctionne que pour avoir une clé publique enfant endurcie.

❖ Clé publique parent → clé privée enfant.

Dériver une clé privée enfant en sachant seulement sa clé publique parent et son code de chaîne parent est impossible.

Seule la clé privée parent donne accès aux clés privées enfants (normales ou endurcies).

Nous pouvons alors nous demander, dans le cas des paires de clés enfants normales : Comment est-ce possible qu'une clé publique enfant, dérivée d'une clé publique étendue, corresponde à une clé privée enfant, dérivée de la clé privée étendue associée ?

Comme vu précédemment, la fonction HMAC aura exactement les mêmes entrées dans le cas de la dérivation d'une paire de clés normales, que ce soit pour calculer k_{enf} ou pour calculer K_{enf}. À savoir :

```
Message = Kpar || i
Clé = Cpar
```

Le hash en sortie sera donc le même. La seule différence réside dans le traitement de ce hash. Pour dériver une clé privée enfant, les 32 premiers octets de ce hash seront ajoutés à la clé privée parent. Pour dériver une clé publique enfant, ces mêmes 32 octets seront passés dans ECDSA pour obtenir un point, qui sera lui-même ajouté à la clé publique parent.

En faisant ce calcul, grâce aux propriétés de la multiplication sur les courbes elliptiques (voir tome 1), la clé publique enfant obtenue en utilisant ECDSA sur la clé privée enfant associée sera exactement le même point que la clé publique enfant obtenue par dérivation depuis la clé publique parent.

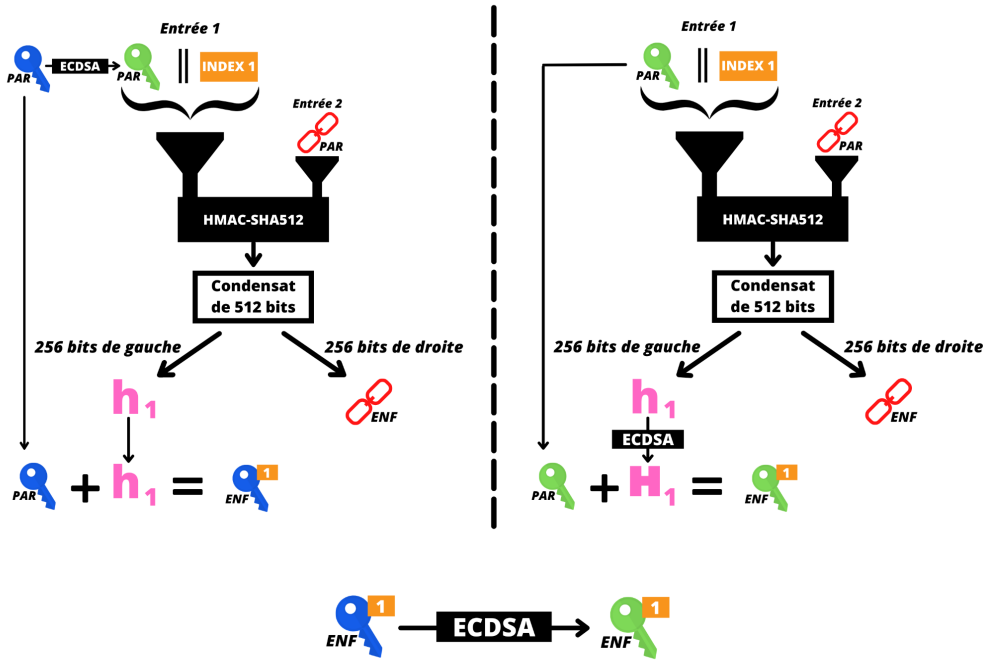
Cela donne cela pour un calcul depuis k_{par} :

```
h = HMAC-SHA512 (Clé = Cpar, Message = point(kpar) || i )

kenf = parse256(h1) + kpar [mod n]

Kenf = kenf x G

G : Le point générateur.
```

Et cela donne cela pour un calcul depuis K_{par} :

```
h = HMAC-SHA512 (Clé = Cpar, Message = Kpar || i )

Kenf = point(parse256(h1)) + Kpar
```

Schématiquement cela ressemblera à cela :



Structure du portefeuille.

Maintenant que nous savons dériver des paires de clés enfants à partir d'une paire de clés parents et du code de chaîne, nous allons voir comment s'organise la structure de cet arbre de dérivation au sein d'un portefeuille HD (déterministe hiérarchique).

L'idée est de dériver depuis la clé privée maîtresse et le code de chaîne maître plusieurs couches de profondeur. L'ajout d'une couche de profondeur correspond à une dérivation d'une paire de clés enfants depuis une paire de clés parents.

Les différents BIP ont défini à travers le temps des normes sur ces chemins de dérivation afin de standardiser leur utilisation au travers des différents logiciels.

Ainsi, voici à quoi correspond chaque couche de dérivation sur la majorité des portefeuilles HD actuels :

La profondeur 0 est celle de la **clé maîtresse** (BIP 32). A cette profondeur se trouve les informations de base de notre portefeuille HD, c'est-à-dire la clé privée maîtresse et le code de chaîne maître que nous avons définis précédemment.

La notation de cette profondeur sera : `m/`

La profondeur 1 est celle de l'**objectif** (BIP 43). L'objectif définit une certaine structure logique pour la suite de la dérivation du portefeuille. Cela permet d'harmoniser la compatibilité des différents logiciels.

Par exemple, une adresse Segwit aura dans son chemin de dérivation en profondeur 1 : `/84'/`. Une adresse P2TR aura : `/86'/`. Une adresse multisig aura `/48'/`. Vous remarquerez que le numéro d'un objectif correspond au numéro du BIP qui le définit.

La profondeur 2 est celle du **type de crypto-monnaie** (BIP 44). Cette profondeur permet de différencier les comptes Bitcoin des comptes des autres crypto-monnaies. Il est ainsi attribué à chaque crypto un index unique que l'on retrouve à cette profondeur de portefeuille. Le but est de conserver une compatibilité sur des portefeuilles multi-coins, tout en séparant bien chaque crypto-monnaie.

Par exemple, l'index attribué à Bitcoin est `0x80000000`. Une adresse Bitcoin a donc en profondeur 2 de son chemin : `/0'/`.

Les index sont déterminés dans l'ordre en partant de 2^{31} (`=0x80000000`). On commence seulement à partir de cet index étant donné que l'on souhaite avoir une dérivation endurcie. Pour rappel, les dérivations endurcies se font seulement grâce à des index compris entre 2^{31} et $2^{32}-1$.

La profondeur 3 est celle du compte (BIP 32). Cette profondeur nous permet de différencier et d'organiser facilement notre portefeuille en différents **comptes**. Ces comptes sont numérotés à partir de `0`.

Par exemple, mon premier compte sur mon portefeuille dispose d'une notation en profondeur 3 : `/0'/`.

En réalité, les clés étendues (**xpub** et **xprv)** telles que vous les connaissez, se trouvent à ce niveau de profondeur.

La profondeur 4 est celle de la chaîne (BIP 32). Chaque compte tel que défini en profondeur 3 disposera de **deux chaînes** en profondeur 4 : une chaîne externe et une chaîne interne (également appelée "change").

La chaîne externe dérive des adresses destinées à être communiquées publiquement, c'est-à-dire les adresses que l'on nous propose lorsque l'on clique sur "*recevoir*" dans notre logiciel de portefeuille.

La chaîne interne dérive les adresses destinées à ne pas être échangées publiquement, c'est-à-dire principalement les adresses de change.

⇨ Pour rappel, une transaction Bitcoin dispose d'inputs et d'outputs. Un UTXO ne peut pas être divisé, il doit impérativement être consommé en intégralité.

Imaginez que vous souhaitiez acheter une baguette pour 4000 sats, et que vous disposiez d'un UTXO de 10000 sats. L'input de votre transaction sera l'UTXO de 10000 sats. Les outputs seront d'une part un UTXO de 4000 sats, qui ira sur l'adresse du boulanger, et d'autre part un UTXO de 6000 sats qui ira vers une autre adresse qui vous appartient. Cette adresse est appelée adresse de change : elle accueille la monnaie de votre transaction.

Ainsi, la chaîne externe aura en profondeur 4 : `/0/`. Et la chaîne interne aura en profondeur 4 : `/1/`.

Notons que ce niveau de profondeur 4 est dédié au type de script dans le cas d'un portefeuille multisig (voir BIP48). Le niveau dédié aux chaînes est alors décalé d'une profondeur.

La profondeur 5 est celle de l'index de l'adresse (BIP 32). Ce niveau indique simplement le numéro de l'adresse et de sa paire de clés afin de les différencier de leurs sœurs.

Par exemple, la première adresse dispose de l'index `/0/`, la deuxième adresse dispose de l'index `/1/`...

Maintenant que nous savons en détail ce que représente chaque profondeur, étudions la notation de ce chemin de dérivation :

Pour noter un chemin de dérivation, chaque niveau de profondeur sera séparé d'une barre oblique `/` et indiquera l'index utilisé. La notation apostrophe `'` signale que la paire de clés indiquée est endurcie.

Un chemin de dérivation d'une adresse Bitcoin ressemblera donc à cela :

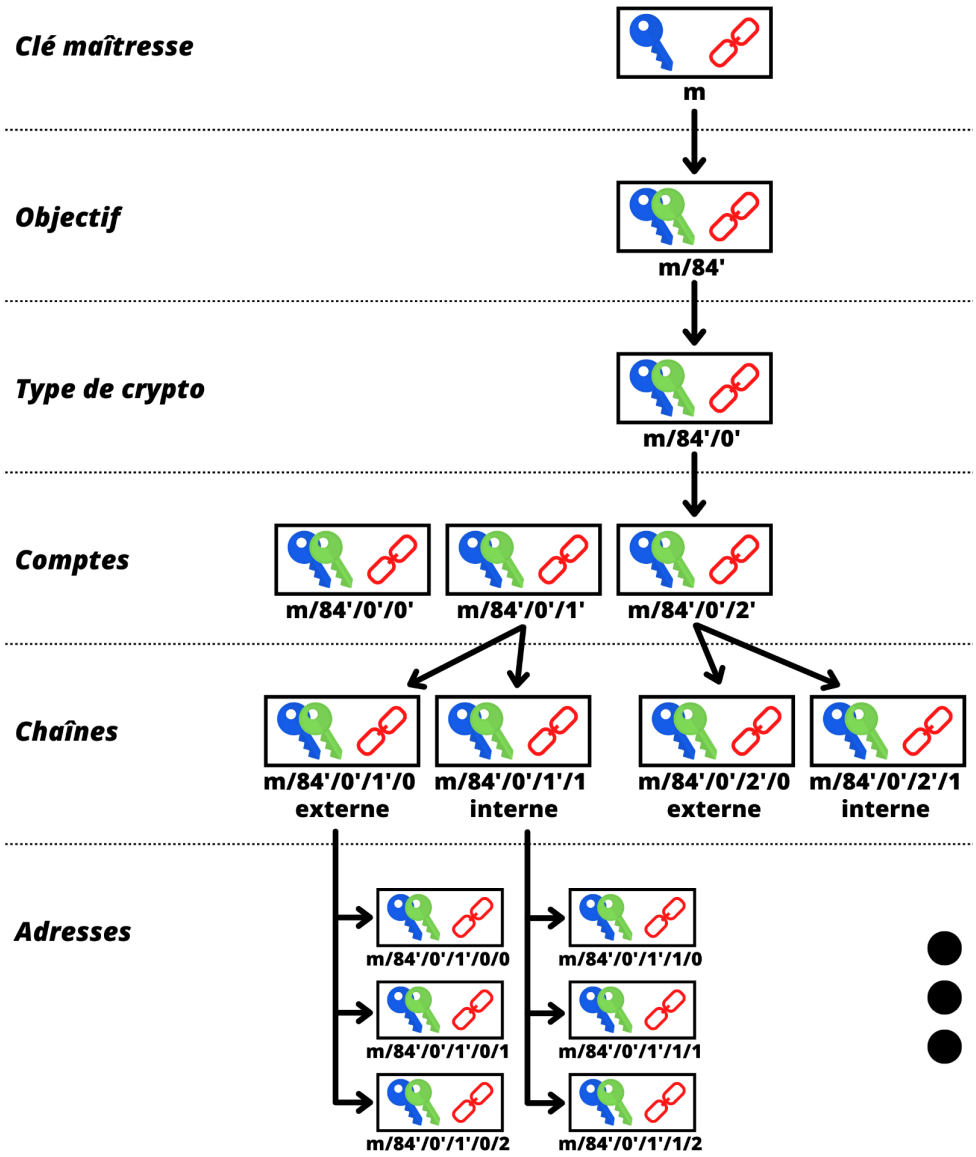
```
m / 84' / 0' / 1' / 0 / 7
```

Dans cet exemple nous pouvons analyser que :

- `84'` indique un standard Segwit.
- `0'` indique que c'est une adresse Bitcoin.
- `1'` indique que j'utilise le deuxième compte du portefeuille.
- `0` indique que l'on dérive une adresse externe.
- `7` indique que cette adresse est la 8ème parmi ses sœurs externes.

⇨ Vous remarquerez que l'apostrophe qui indique l'utilisation d'une dérivation endurcie disparaît à partir de la profondeur 4. C'est grâce à cela que l'on peut dériver l'intégralité des clés publiques enfants d'un compte à partir de la **xpub**.

Voici une représentation schématique des chemins de dérivation au sein d'un portefeuille HD :



Les plus attentifs d'entre-vous auront remarqué que parfois, dans le chemin de dérivation, il est noté `/0'/`. Cela est censé correspondre à un index `0` sur une clé enfant endurcie. Mais comme nous l'avons vu précédemment, les index des clés endurcies ne peuvent être compris qu'entre 2^{31} et $2^{32}-1$.

Ainsi, lorsque vous voyez dans un chemin de dérivation un index avec une apostrophe, il faut lui ajouter 2^{31} pour obtenir le véritable nombre utilisé comme index.

Par exemple, si j'ai **`/44'/`** sur mon chemin de dérivation, étant donné qu'il y a une apostrophe indiquant une dérivation endurcie, je dois ajouter **2^{31}** (soit **`2 147 483 648`**) à l'index indiqué pour obtenir le véritable index.

Dans mon exemple :

```
/44'/ → index = 44 + 2^31

= 44 + 2 147 483 648

= 2 147 483 692

Format hexadécimal de l'index : 0x8000002C
```

Pour résumer, voici à quoi correspond chaque niveau d'un chemin de dérivation dans le cas d'un portefeuille HD lambda :

```
clé maîtresse / objectif' / type de crypto' / compte' /
chaîne / numéro d'adresse
```

Dérivation d'une adresse.

En toute fin de dérivation d'un portefeuille nous avons les adresses de réception.

Pour rappel, l'objectif final de la dérivation d'un portefeuille Bitcoin est de disposer de paires de clés soeurs. Chaque paire de clé est composée d'une clé privée et d'une clé publique.

La clé privée est déterminée par dérivation depuis sa clé privée parent. La clé publique associée sera simplement dérivée par multiplication sur les courbes elliptiques comme expliqué dans le **tome 1**.

Avec une paire clé privée / clé publique, on peut déjà recevoir des bitcoins. Il existe un script P2PK (*Pay to Public Key*) qui permet de bloquer des bitcoins sur une clé publique. Cette méthode était notamment celle utilisée par Satoshi Nakamoto pour recevoir ses premiers bitcoins minés.

Pour débloquer les bitcoins bloqués sur cette clé publique par le script P2PK, le propriétaire devra fournir une signature numérique en utilisant la clé privée associée (voir **tome 1**).

Aujourd'hui, très peu d'utilisateurs emploient encore cette méthode. Au lieu de cela, la majorité d'entre eux utilise des adresses.

Le fonctionnement est alors similaire à celui susmentionné, à la différence près que le destinataire ne donne pas à l'émetteur sa clé publique, mais une adresse de réception.

Une adresse est tout simplement un hash d'une clé publique, modulé dans un format spécifique. L'émetteur enverra donc les bitcoins vers l'adresse du destinataire au lieu de les envoyer directement à sa clé publique. Une adresse est donc une destination à usage unique.

Pour rappel, une fonction de hachage n'est pas réversible, il est donc impossible de déterminer une clé publique en sachant seulement son adresse.

Pourquoi utilise-t-on une adresse si l'on peut envoyer des bitcoins directement à une clé publique ?

Il existe plusieurs raisons, mais la principale semble être le gain de place. Une clé publique dans sa forme non-compressée fait 512 bits alors que RIPEMD160 (la fonction utilisée pour les adresses) produit un condensat de 160 bits seulement.

A la création de Bitcoin, on ne savait visiblement pas qu'une clé publique pouvait être compressée dans un format de 256 bits + 8 bits : en conservant uniquement la valeur de $\boldsymbol{x}$ + le préfixe indiquant quel $\boldsymbol{y}$ choisir (voir ci-dessous). On a donc opté pour un hash de cette clé publique : l'adresse.

Le fait que cette information soit plus petite ne permet pas du tout de gagner de la place dans les blocs. En effet, un script P2PK nécessite légèrement plus de place en output mais beaucoup moins de place en input.

Finalement, cette information a plutôt été réduite afin de faciliter la tâche à l'utilisateur. Une fois encodée en base 58, une adresse fait seulement 34 caractères alors qu'une clé publique ferait probablement 51 caractères.

L'utilisateur n'a donc qu'à noter 34 caractères pour recevoir des bitcoins.

Au-delà du gain de taille, les adresses ont d'autres avantages par rapport à une simple clé publique :

- Elles incluent une checksum permettant de détecter très facilement une faute de frappe lorsque l'utilisateur entre son adresse quelque part pour recevoir des fonds.
- Elles permettent d'ajouter une seconde couche de sécurité entre l'information publique (l'adresse) et l'information privée (la clé privée). L'utilisation de fonctions de hachages permet notamment à une paire de clés d'être résistante à une attaque d'un potentiel futur calculateur quantique. Cela est vrai tant que la clé publique n'est pas révélée.

Maintenant, voyons comment dériver une adresse à partir d'une clé publique.

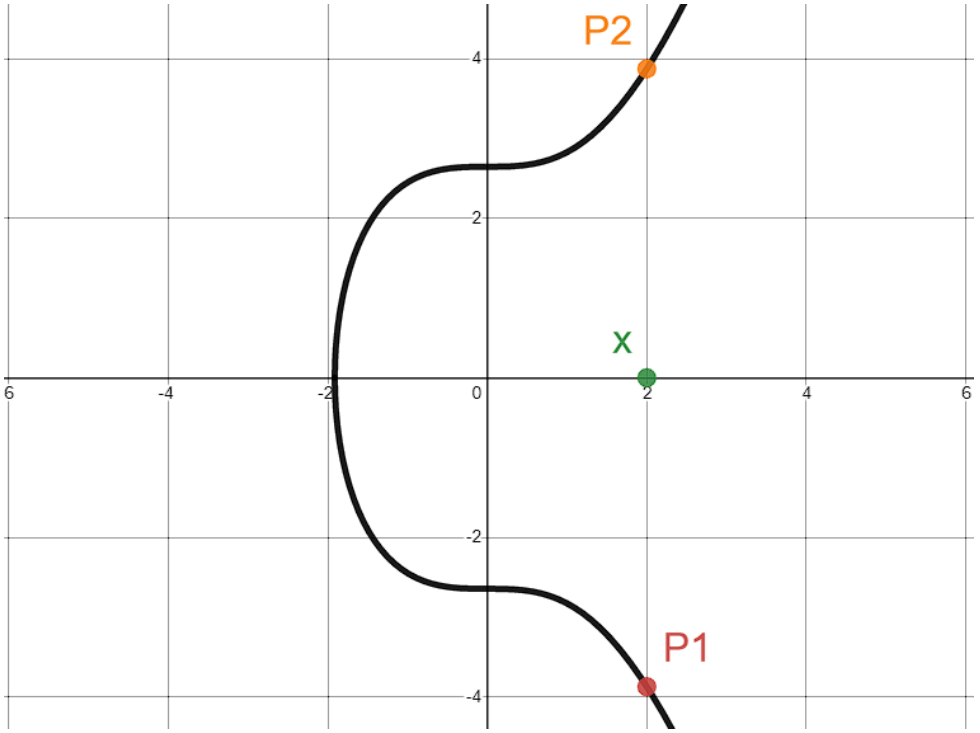
Nous prendrons ici l'exemple d'une adresse Segwit en format Bech32 (BIP173). Néanmoins, le processus sera à peu près similaire pour les autres types d'adresses.

Étape 1 : Il faut d'abord disposer d'une paire de clés enfants. On récupère la clé publique qu'il va falloir compresser.

Pour rappel, une clé publique est simplement un point sur la courbe elliptique (voir [tome 1]). Une clé publique fera donc 520 bits : 8 bits pour un préfixe qui sera `04`, 256 bits pour la valeur x et 256 bits pour la valeur y.

La courbe elliptique étant symétrique par rapport à l'axe des abscisses, pour tout point (x, y) sur la courbe elliptique, il existe un point $(x, -y)$ qui sera également sur la courbe elliptique. Cela veut dire que pour tout x, il ne peut y avoir que 2 valeurs de y sur la courbe elliptique : soit positive, soit négative.

Par exemple, voici une abscisse x. Il n'existe que les points `P1` et `P2` qui sont sur la courbe elliptique et qui disposent de cette même abscisse x :



Pour désigner une clé publique, il suffit donc d'indiquer son abscisse x. Pour savoir lequel des deux seuls points qui ont cette abscisse est bien notre clé publique, nous ajoutons à x un préfixe permettant d'indiquer quel y il faut prendre.

De cette façon, on peut réduire la taille d'une clé publique de 520 bits à 264 bits (`préfixe : 8 bits || x : 256 bits`). Cette forme réduite de la clé publique originelle s'appelle forme **compressée**.

Comme indiqué dans le **tome 1**, en réalité nous ne travaillons pas sur le corps des réels mais sur un corps fini d'ordre p (nombre premier). En conséquence, le signe de y sera équivalent à sa parité. Nous devrons donc simplement indiquer dans notre préfixe si la valeur du y de la clé publique est paire ou impaire.

Ainsi, le préfixe `02` sera utilisé pour indiquer un y pair, et le préfixe `03` sera utilisé pour indiquer un y impair.

Maintenant que nous savons compresser une clé publique, prenons un exemple.

Soit une clé publique, un point sur la courbe elliptique, décrite en base 16 :

```
K =
04678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e
0ea1f61deb649f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b
8d578a4c702b6bf11d5f
```

On peut déjà séparer le préfixe, x et y :

```
Préfixe = 04

x =
678afdb0fe5548271967f1a67130b7105cd6a828e03909a67962e0e
a1f61deb6

y =
49f6bc3f4cef38c4f35504e51ec112de5c384df7ba0b8d578a4c702
b6bf11d5f
```

On va maintenant déterminer si y est pair ou impair. La parité d'un nombre se calcule sur le dernier chiffre. Par exemple : `654425541` est impair car le dernier chiffre `1` est impair.

On va donc prendre simplement `f`, le dernier caractère hexadécimal de notre y, et déterminer si celui-ci est pair ou impair.

```
→ Base 16 (HEX) : f
→ Base 10 (décimal) : 15
→ y est donc impair.
```

Le préfixe de notre clé publique compressée sera donc `03`. Notre clé publique compressée sera finalement en base 16 :

```
K =
03678AFDB0FE5548271967F1A67130B7105CD6A828E03909A67962E
0EA1F61DEB6
```

Étape 2 : On passe ensuite la clé publique compressée dans la fonction de hachage SHA256. Il en ressortira un hash de 256 bits.

Dans notre exemple cela donne en base 16 :

```
SHA256(K) =
C489EBD66E4103B3C4B5EAFF462B92F5847CA2DCE0825F4997C7CF5
7DF35BF3A
```

Étape 3 : Puis on passe le hash de l'étape précédente dans la fonction de hachage RIPEMD160. Cette étape permet principalement de réduire la taille de l'empreinte à 160 bits.

Dans notre exemple cela donnera :

```
RIPEMD160(SHA256(K)) =

En base 16 : 9F81322CC88622CA4CCB2A52A21E2888727AA535

En base 2 :
10011111100000010011001000101100110010001000011000100010
01100101001001100110010110010101001010010101000100001111
10001010001000100001110010011111010101001010011010 1
```

⇨ Ce double hachage SHA256 + RIPEMD160 est également nommé : HASH160.

Étape 4 : On encode ensuite ce condensat en format binaire par groupes de 5 bits.

Ce qui nous donne :

```
Base 2 (binaire) :
10011 11110 00000 10011 00100 01011 00110 01000 10000
11000 10001 01100 10100 10011 00110 01011 00101 01001
01001 01010 00100 00111 10001 01000 10001 00001 11001
00111 10101 01001 01001 10101

Base 16 (HEX) :
131e0013040b06081018110c1413060b0509090a040711081101190
715090915

Base 10 (décimal) :
19 30 00 19 04 11 06 08 16 24 17 12 20 19 06 11 05 09
09 10 04 07 17 08 17 01 25 07 21 09 09 21
```

Ce sera ce que l'on appelle le "**Payload**" : la charge utile de l'adresse.

Étape 5 : On ajoute ensuite les données nécessaires pour calculer la somme de contrôle.

Pour le calcul de la somme de contrôle, les premiers standards Bitcoin en Base58check utilisent un double hachage (HASH256). Dans le cas de Bech32, une autre fonction est utilisée : La checksum est calculée par un code **BCH** (***B**ose, Ray-**C**haudhuri et **H**ocquenghem*) qui est un code de correction d'erreur.

Son utilisation ici permet de garantir une détection des erreurs pour des caractères mal saisis lorsque l'utilisateur copie son adresse de réception.

Le calcul de cette checksum est assez complexe, il est basé sur l'arithmétique de champ fini polynomial. Je ne vais donc pas détailler ici le calcul de la checksum.

Pour ceux qui souhaitent en découvrir plus sur le fonctionnement des codes BCH, je vous mets un lien vers un article traitant ce sujet à la fin de l'ebook.

Ce programme BCH aura besoin de plusieurs informations en entrée (inputs) pour calculer la checksum :

- **Le HRP** (Human Readable Part - Partie lisible par l'utilisateur) : **`bc`** pour une adresse Segwit.

Cet HRP nous allons l'étendre. Pour ce faire, il faut d'abord encoder chaque lettre en base 2. On va ensuite prendre les 3 premiers bits de chaque caractère que nous convertirons en base 10. On va mettre un séparateur `0`. Et on va concaténer les 5 derniers bits de chaque caractère que nous convertirons également en décimal :

```
b (ASCII) = 01100010 (base 2)
c (ASCII) = 01100011 (base 2)

011 (base 2) = 3 (base 10)
011 (base 2) = 3 (base 10)
00010 (base 2) = 2 (base 10)
00011 (base 2) = 3 (base 10)

HRP étendu en base 10 = 03 03 00 02 03
```

Le fait d'étendre le HRP permet d'isoler les 5 derniers bits de chaque caractère qui sont plus exposés à une modification. Cela permet de renforcer la puissance de la checksum.

- **La version** : dans le cas de Segwit ce sera `00`.
- **Le Payload** : le résultat de l'étape 4.
- **L'espace** réservé pour la somme de contrôle. Nous voulons une checksum à 6 caractères, nous prendrons donc six `0` : `00 00 00 00 00 00`.

Finalement, notre entrée de programme BCH sera en base 10 :

```
03 03 00 02 03 00 19 30 00 19 04 11 06 08 16 24 17 12
20 19 06 11 05 09 09 10 04 07 17 08 17 01 25 07 21 09
09 21 00 00 00 00 00 00
```

Étape 6 : On calcule ensuite la somme de contrôle grâce au code BCH avec les informations de l'étape 5.

Dans notre exemple cela nous donnera la checksum suivante :

```
En base 16 :
0a100b040d12

En base 10 :
10 16 11 04 13 18
```

Étape 7 : On peut maintenant construire notre adresse. On va commencer par concaténer dans l'ordre :

- La version de Segwit : `00`
- Le résultat de l'étape 4 : le **Payload**.
- La checksum : `10 16 11 04 13 18`

Ce qui nous donne en base 10 :

```
00 19 30 00 19 04 11 06 08 16 24 17 12 20 19 06 11 05
09 09 10 04 07 17 08 17 01 25 07 21 09 09 21 10 16 11
04 13 18
```

On va maintenant mapper chaque valeur à son caractère Bech32 en fonction du tableau de conversion suivant :

	0	1	2	3	4	5	6	7
+0	q	p	z	r	y	9	x	8
+8	g	f	2	t	v	d	w	0
+16	s	3	j	n	5	4	k	h
+24	c	e	6	m	u	a	7	l

Pour mapper une valeur en un caractère Bech32, il suffit de trouver les valeurs qui s'additionnent en première colonne et en première ligne, et de récupérer le caractère qui leur correspond.

Par exemple, si nous avons une valeur décimale de `6`, son caractère Bech32 sera `x` car `6 = 6 + 0`. Si nous avons une valeur décimale de `27`, son caractère Bech32 sera `m` car `27 = 3 + 24` :

	0	1	2	3	4	5	6	7
+0	q	p	z	r	y	9	x	8
+8	g	f	2	t	v	d	w	0
+16	s	3	j	n	5	4	k	h
+24	c	e	6	m	u	a	7	l

La particularité de l'alphabet Bech32 est qu'il intègre l'ensemble des caractères alphanumériques à l'exception de `1`, `b`, `i` et `o`. C'est une variante de la Base 32 classique.

Si nous reprenons notre exemple, notre concaténation donnera en caractère Bech32 :

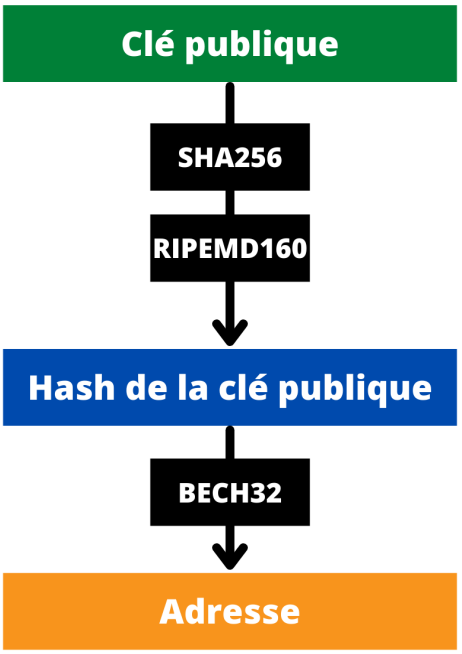
```
qn7qnytxgsc3v5nxt9ff2y83g3pe84ff42stydj
```

Notre adresse est presque prête, il suffit d'y ajouter le HRP `bc` et le séparateur `1` :

```
bc1qn7qnytxgsc3v5nxt9ff2y83g3pe84ff42stydj
```

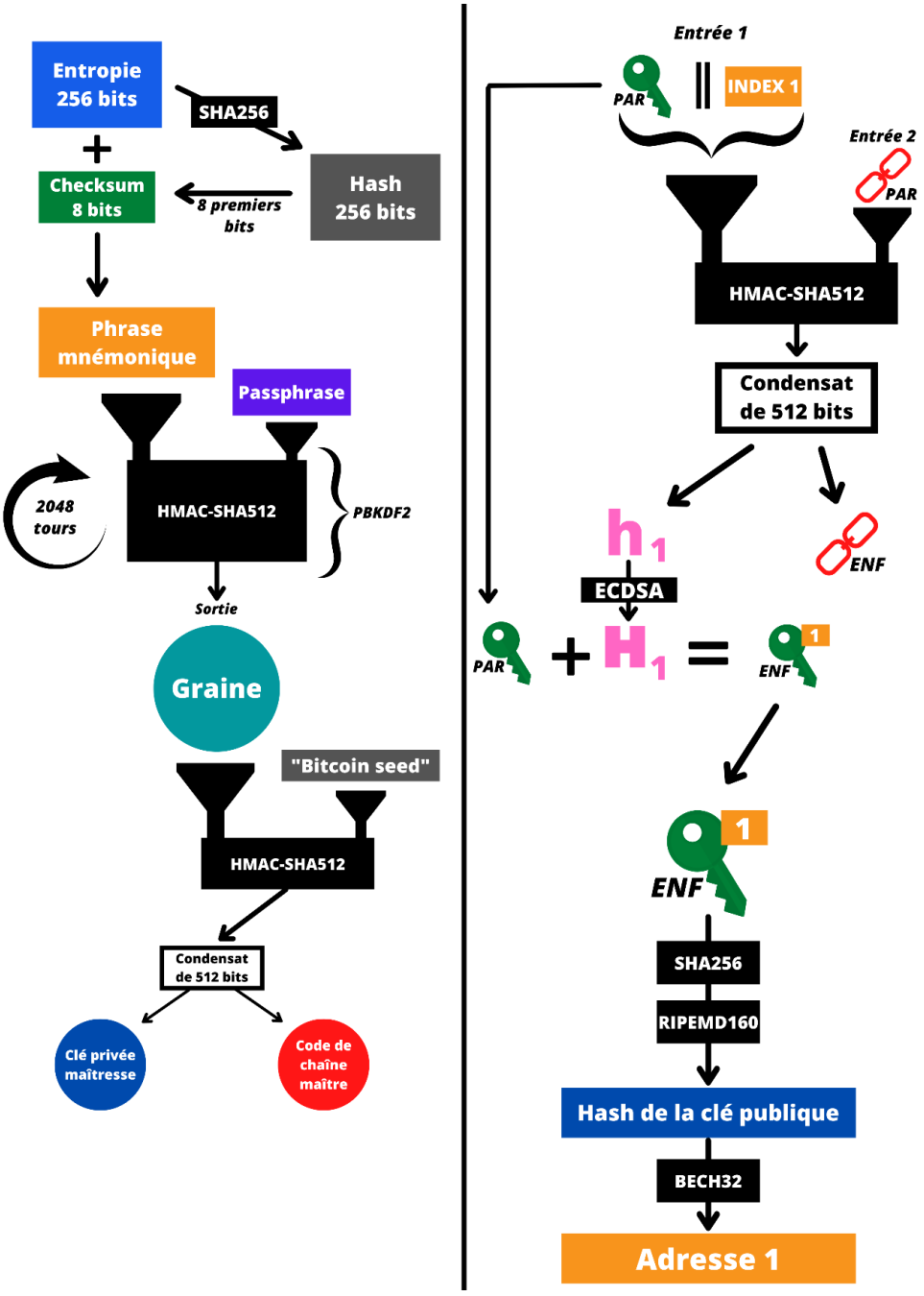
Notre adresse est enfin prête. Nous pouvons la transmettre à un émetteur qui souhaite nous envoyer des bitcoins.

Pour résumer, voici schématiquement le chemin que nous avons réalisé, depuis la clé publique, afin d'obtenir l'adresse :



Vision globale.

Pour terminer cet ebook portant sur les portefeuilles HD, je vous propose un schéma offrant une vision globale de la construction d'un portefeuille HD, depuis l'entropie, jusqu'aux adresses :



Conclusion.

Nous avons pu découvrir en détail le fonctionnement et les caractéristiques d'un portefeuille déterministe hiérarchique.

Le portefeuille est selon moi une des parties les plus importantes à comprendre du protocole Bitcoin étant donné que c'est un passage obligatoire pour tout utilisateur. Ces connaissances seront indispensables pour aborder plus tard les stratégies de sécurisation et de préservation de la confidentialité. J'aborderai ces points en détail dans un prochain ouvrage.

Pour mieux comprendre tous les concepts évoqués, je vous conseille de pratiquer, et de comparer les actions que vous faites sur votre portefeuille avec les informations théoriques que vous avez lues dans cet ebook.

Les connaissances essentielles qui ressortent de cet ebook sont :

- Un portefeuille permet de déterminer et de gérer les clés donnant accès aux bitcoins représentés par des UTXO.

- Un portefeuille HD est basé sur une entropie, une phrase mnémonique et une éventuelle passphrase. La graine représente l'information unique qui permet de dériver l'ensemble des clés d'un portefeuille et donc de débloquer les bitcoins associés.

- Une passphrase optionnelle peut être ajoutée à un portefeuille pour augmenter son niveau de sécurisation.

- Les paires de clés enfants sont le résultat d'un chemin de dérivation de plusieurs profondeurs.

- Les adresses sont générées à partir d'une clé publique en utilisant une fonction de hachage et en appliquant un format spécifique.

Dans les prochains tomes de la série ***Bitcoin Démocratisé***, nous étudierons sûrement en détail les mécanismes des transactions, de la preuve de travail ou encore du réseau pair-à-pair.

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.

Articles sur la génération d'une phrase mnémonique :
https://armantheparman.com/dicev1/
https://learnmeabitcoin.com/technical/mnemonic

Les différents BIP étudiés :
https://github.com/bitcoin/bips/blob/master/bip-0032.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0038.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0039.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0043.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0044.mediawiki
https://github.com/satoshilabs/slips/blob/master/slip-0044.md
https://github.com/bitcoin/bips/blob/master/bip-0048.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0084.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0086.mediawiki
https://github.com/bitcoin/bips/blob/master/bip-0173.mediawiki
https://en.bitcoin.it/wiki/BIP_0032
https://en.bitcoin.it/wiki/BIP_0044

Ressource sur l'encodage Base58 :
https://en.bitcoin.it/wiki/Base58Check_encoding

Articles sur les portefeuilles HD, la dérivation et BIP32 :
https://medium.com/vault0x/hierarchically-deterministic-wallets-the-concepts-3aa487e71235
https://blog.fearcat.in/a?ID=00550-29885170-f73b-4603-9cf7-d21383273297
https://learnmeabitcoin.com/technical/extended-keys
https://medium.com/@robbiehanson15/the-math-behind-bip-32-child-key-derivation-7d85f61a6681
https://learnmeabitcoin.com/technical/derivation-paths

Articles et ressources sur Bech32 et les codes BCH :
https://github.com/f2pool/bech32-elixir/blob/master/lib/bech32.ex
https://hexdocs.pm/bech32/Bech32.html
https://medium.com/@meshcollider/some-of-the-math-behind-bech32-addresses-cf03c7496285
https://blog.costan.ro/post/2020-02-10-bitcoin-bech32-segwit-address/

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