MID£COIN

Projet

Participants : **Ali, Brandon, Sergio, Youssef**

Date : Décembre 2023

Table des matières

1 INTRODUCTION

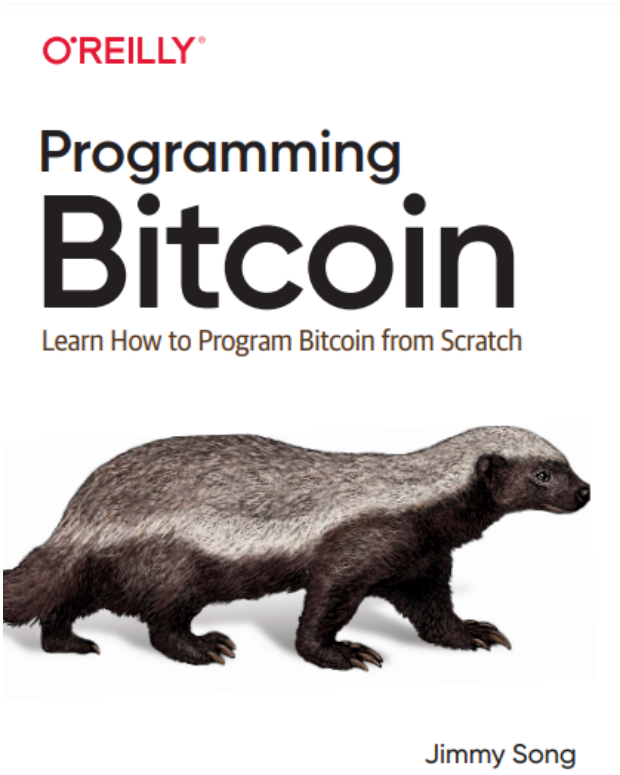
Ce projet nous a permis de mieux comprendre comment fonctionnent les cryptomonnaies, ce domaine qui nous paraissait obscure et complexe. Grâce à cet exercice instructif combinant mathématiques et informatiques nous avons pu acquérir quelques bases dans ce vaste sujet qu'est la cryptomonnaie.

Pour commencer ce projet nous serons obliger de revenir sur la soutenance du 17 novembre qui nous a permis de relever quelques erreurs commises, ainsi que sur les ambiguïtées concernant la signature et la vérification d'un message.

Ensuite, nous nous dirigerons au coeur du sujet et expliquerons notre réprésentation d'un bloc et plus généralement la blockchain, une structure de données particulière adaptée à notre problème.

Et pour finir, notre travail est loin d'être parfait donc nous apporterons un point de vu critique, pour comprendre les limites et ce qui peut-être amélioré dans notre projet.

Notre projet python est disponible à cet adresse. On s'est inspiré de la vidéo qui résume très bien les blockchains et du livre "Programming Bitcoin de Jimmy Song".



2 SIGNATURE

Objectif : Dans cette partie, la problématique est la suivante : Comment arrive-t-on à développer un moyen puissant qui permet de valider aux yeux du monde la validité d'un message. En clair, l'auteur (Bob) signe son message avec une signature que seul Bob peut reproduire. Et dans un second temps TOUT le monde peut vérifier que ce message est bien écrit par Bob et TOUT le monde sera convaincus si la vérification est réussi. Dans cette démarche Bob est représenté par une clé publique (visible au monde entier) et une clé privée (visible à Bob uniquement).

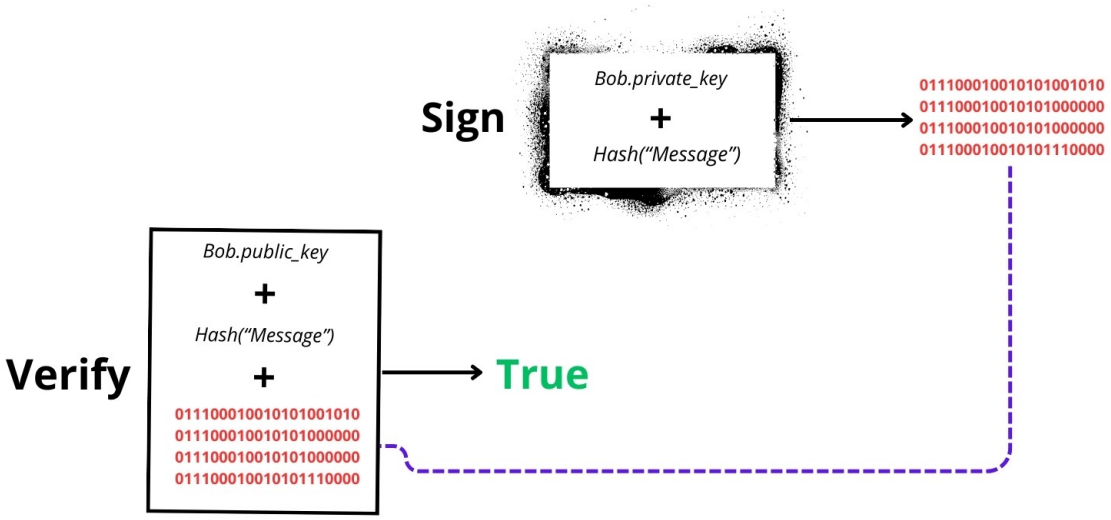


FIGURE 1 – Idée sur le fonctionnement des fonctions Verify et Sign

Ceci illustre bien les deux grandes fonctions de la première partie, et ils vont servir non pas à signer des messages quelconques mais signer des transactions. Dans cette partie nous allons dans un premier temps voir comment la fonction de hashage fonctionne et dans un second temps l'utilisation de notre courbe elliptique pour réussir à signer et vérifier un message.

2.1 Fonction de Hashage

Les fonctions de hachage jouent un rôle essentiel dans la blockchain et la sécurité de nombreux systèmes informatiques. Une fonction de hachage prend en entrée des données de taille variable et génère une empreinte numérique de taille fixe, souvent une séquence alphanumérique unique.

Fonctionnement Une fonction de hachage doit satisfaire plusieurs propriétés pour être efficace dans le contexte de la blockchain :

1. **Déterministe :** Pour une entrée donnée, la fonction de hachage doit toujours produire la même sortie.
2. **Rapide à calculer :** Le processus de hachage doit être rapide, même pour de grandes quantités de données.
3. **Résistant aux collisions :** Il doit être improbable que deux ensembles de données différents produisent la même empreinte (collision).
4. **Effet avalanche :** Une petite modification dans l'entrée doit entraîner une modification significative dans la sortie (principe de l'effet avalanche).

2.2 Courbe Elliptique Cryptographie

Comme dit dans le livre "Programming Bitcoin- Jimmy Song" Pour vérifier un message il suffit de combiner notre courbe elliptique bien choisie et la notion d'un corps fini dans ce rapport le but n'est pas de revenir sur les deux notions (c'est très bien expliquer dans le livre) mais de voir l'intérêt qu'elles ont pour notre projet et comment on s'en est servi.

2.2.1 Choix des paramétres de la courbe elliptique et Le domaine du corps fini

Premièrement, l'équation de la courbe doit être choisi, pour accélerer les calculs nous avons décider de fixer les entiers relatifs $a = 0$ et $b = 7$, l'équation de la courbe est donc la suivante : $y^2 = x^3+7$. Nous avions ensuite besoin pour l'implémentation en 256bits de 3 éléments, à savoir le P qui représente l'élement maximale du corps, le G qui représente le point générateur de la courbe (c-à-d que $\forall x \in K$ un corps, $\exists n \in \mathbf{N}$, tq $x = n \times G$), et enfin N qui est la première valeur telle que $N \times G = 0$. A remarquer que P est très proche de 2^{256} permettant alors que la plupart des valeurs se retrouvant inférieur à 2^{256} appartiennt au corps K et que nous ayons autant de points sur la coube qu'il n'y a de nombres encodables en 256bits. Le N est également très proche de 2^{256}, afin de permettre à la multiplication d'être exprimée en 256bits. Dans ce projet, nous obtenons $P = 2^{256} - 2^{32} - 977$, $G = (5.5066263022277344e + 76, 3.2670510020758816e + 76)$ et enfin $N = 1.157920892373162e + 77$. En fait c'est la courbe cryptographique secp256k1 utilisé pour le bitcoin.

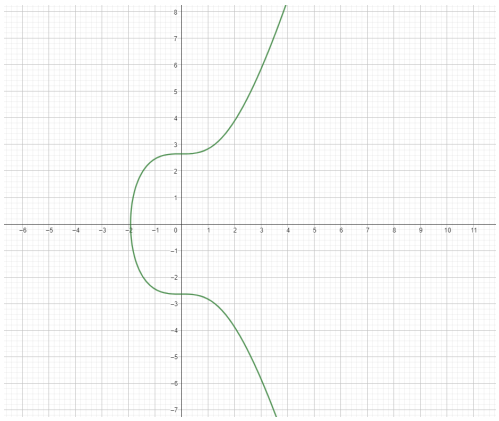


FIGURE 2 – Graphique de la courbe elliptique $y^2 = x^3 + 7$

2.2.2 Implémentation de la fonction sign

Revenons à notre objectif initial qui était de réussir à signer un message de façon unique en fonction de la clé privée.

1. **Choisir k aléatoire :** La première étape est de choisir un nombre totalement aléatoire dans notre domaine de définition [0, N], comme dit dans le livre la fonction randint de python n'est pas aléatoire donc il faut créer une fonction qui génére un nombre complétement aléatoire (voir code).
2. **Calculer l'inverse de k :** ensuite on doit calculer l'inverse de nombre aléatoire dans notre corps, et pour ceci on utilise le fait que N soit un nombre premier et donc le théorème du petit Fermat $k^{-1} = k^{N-2}[N]$.
3. **On calcule r :** r c'est tout simplement la coordonnée x du point $k * G$, G étant le point générateur présenté ci-dessus.
4. **Calculer s :** $s = (z + re) * k^{-1}$ z représente le message hashé entré en paramètre de la fonction, et e représente la private key avec quoi on veut signer.

la signature c'est tout simplement le couple (r, s) en réalité c'est seulement le s nous intéresse c'est lui la signature en 256 bits mais le r va nous servir pour vérifier.

2.2.3 Implémentation de la fonction verify

1. **Calculer s^{-1} :** Encore une fois le théorème du petit Fermat nous sauve car on a $s^{-1} = s^{N-2}[N]$
2. **Calculer u :** $u = s^{-1} * z$, z etant le message hashé à vérifier.
3. **Calculer v :** $v = s^{-1} * r$ le r est renvoyer par la signature du message
4. **Calculer T :** $T = u * G + v * P$ avec $P = eG$ c'est le point qui repésente la clé publique.

Conclusion : si la coordonnée x de T est égal à r alors le message est bien écrit par la personne avec cette clé publique. Sinon la signature n'est pas valide et donc l'identité de l'auteur de ce message est fausse.

```
data_to_sign = b"my_message!"
secret_key = b"my_secret_key"

data_sign = hash256(data_to_sign)
secret_key_sign = hash256(secret_key)
sk = PrivateKey(secret_key_sign)
signature = sk.sign(data_sign)
P = sk.secret * G
print(P.verify(data_sign, signature))
```

FIGURE 3 – exemple test verify renvoyant Vrai

3 BLOCKCHAIN

Avant de parler d'une chaîne de blocs, il fautd'abord savoir ce qu'est un bloc.

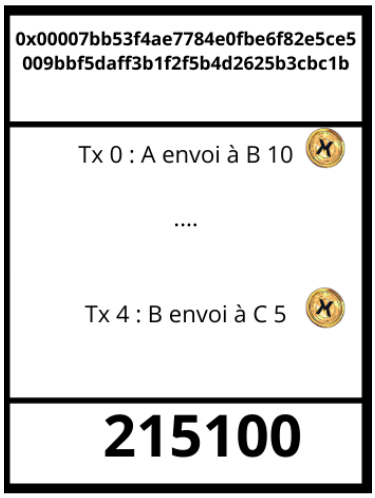


FIGURE 4 – exemple de notre représentation d'un bloc

3.1 Elements dans un Bloc

3.1.1 Transaction

Chaque bloc contient un nombre fixe de transactions, défini à 5 dans notre configuration. Ces transactions constituent les échanges enregistrés sur la blockchain, formant ainsi l'historique complet des activités. Chaque transaction est composée de plusieurs éléments essentiels qui permettent de garantir la validité et la sécurité du processus.

— Tout d'abord, la transaction inclut la quantité de cryptomonnaie transférée, spécifiant le montant échangé entre les parties impliquées. Ce paramètre est crucial pour suivre et vérifier les échanges de valeur sur la blockchain.

— En plus de la quantité, la transaction identifie clairement celui qui effectue le transfert et celui qui le reçoit. Ces données sont essentielles pour suivre l'origine et la destination des fonds, assurant ainsi une traçabilité complète des transactions.

— La sécurité de la transaction est garantie par la présence d'une signature numérique. Chaque partie impliquée dans la transaction utilise une clé privée pour signer électroniquement les détails de la transaction. Cette signature permet aux nœuds du réseau de vérifier l'authenticité de la transaction et d'assurer que seules les parties autorisées peuvent effectuer des échanges.

— Enfin, chaque transaction est associée à un numéro unique d'identification. Ce numéro de transaction permet de référencer de manière unique chaque échange sur la blockchain. Il contribue à éviter toute confusion ou duplication au sein du réseau, renforçant ainsi l'intégrité du système.

3.1.2 Proof of Work

Un autre élément crucial est la *Proof of Work* (preuve de travail), un nombre calculé de manière intensive et ajouté au bloc. Ce nombre est choisi de manière à ce que le hachage résultant du bloc, lorsqu'il est concaténé avec la *Proof of Work*, satisfasse une condition spécifique. Dans notre cas, cette condition exige que l'entier obtenu soit inférieur à 2^{236}, garantissant ainsi une caractéristique spécifique. en clair, on cherche un entier telle que le hash du bloc avec les transactions additionner à cet entier nous donne 256 bits, avec les 20 premiers bits à 0. Et chercher ce nombre revient à miner un bloc.

```
def proofOfWork(self):
    self.pW = 0
    while not self.verify_block():
        self.pW += 1
    return self.pW
```

FIGURE 5 – Fonction de minage

tant que cette fonction ne trouve pas la proof of work pour laquelle le hash du bloc satisfait les conditions nécessaires (5 transaction par bloc + 20 zéros) on test avec l'entier suivant. la proof of work est représentée par l'entier en bas du bloc dans la figure 4

3.1.3 Prev Code

Dans la figure 4 le previous code est la partie en hexadécimal en haut du bloc et cela représente tout simplement le hash du bloc précédent dans la chaîne

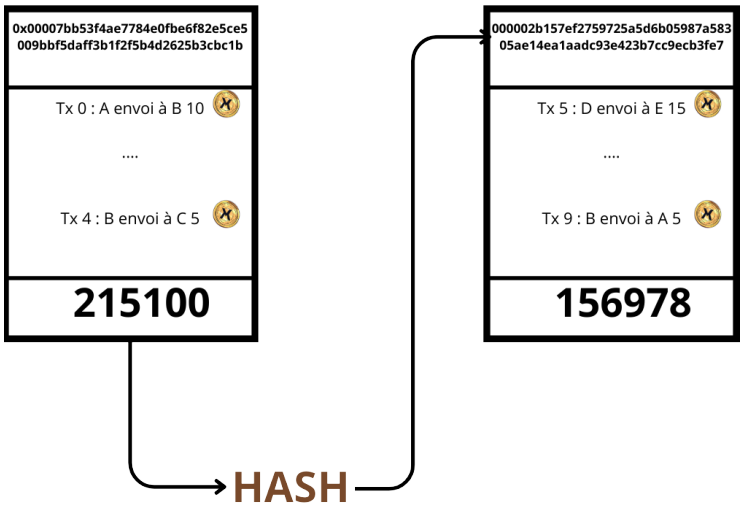


FIGURE 6 – Obtention du Prev Code

3.2 Implémentation de la Blockchain

Après avoir présenté le concept de blocs, nous allons maintenant expliquer le fonctionnement de la Blockchain en commençant par une petite introduction et en poursuivant avec notre implémentation.

3.2.1 Introduction à la Blockchain

La blockchain : Une innovation majeure dans le domaine des technologies de l'information, elle offre une approche décentralisée et sécurisée pour enregistrer et vérifier des informations. Fondamentalement, elle peut être comprise comme une structure de données en forme de chaîne, composée de blocs successifs.

Décentralisation sécurisée : La blockchain est un registre numérique décentralisé, partagé et sécurisé, où chaque morceau d'information est stocké dans un bloc. Ces blocs sont liés entre eux de manière linéaire, formant ainsi une chaîne continue. Leur liaison est rendue possible grâce à la présence du précédent bloc dans chaque maillon de la chaîne, assurant ainsi la continuité et l'intégrité de la blockchain.

Proof of Work : Au cœur du fonctionnement de la blockchain se trouve le concept de la *Proof of Work*. Cela garantit que l'ajout d'un bloc à la chaîne nécessite un effort significatif.

Intégrité de la chaîne : Si un bloc est modifié, la *Proof of Work* devient invalide. La recherche d'une nouvelle *Proof of Work* pour le bloc modifié change son hachage, impactant le bloc suivant. Ce mécanisme assure l'intégrité de l'ensemble de la chaîne, rendant toute tentative de manipulation coûteuse et peu pratique.

Sécurité et confiance : La blockchain offre ainsi une solution robuste pour créer un registre immuable et de confiance, éliminant le besoin d'une autorité centrale pour valider les transactions et assurant la sécurité des données enregistrées.

3.2.2 Création de la Blockchain, ajout de blocs et vérification de la validité de la Blockchain

La Blockchain étant une suite de blocs, on a décidé de implémenter la Blockchain comme une classe ayant comme attributs sa taille et une liste des blocs qui la constituent.

```
class BlockChain:
    def __init__(self):
        self.size = 0
        self.chain = []
```

FIGURE 7 – Initialisation Class BlockChain

Pour rajouter un bloc à notre blockChain il faut d'abord qu'il contienne bien 5 transactions si c'est le cas on le rajoute à la fin de chaîne son previous code sera équivalent au hash code du dernier bloc présent dans la chaîne afin de miner le bloc pour trouver la proof of work. Un seul autre cas à gérer est lorsque il n'y a pas de bloc présent dans la chaîne est qu'on doit ajouter le premier bloc, alors on va juste dire que son prev code est 0.

3.3 Minage de Bloc

Les mineurs sont ceux qui crée les blocs sans eux, le système de la blockchain ne fonctionnerait pas, dans notre cas c'est facile de calculer la proof of work d'un bloc puisqu'on demande "seulement" 20 zéros au début mais plus le nombre de zéros est grand plus la proof of work sera longue à calculer donc utilisant plus de ressource, mais cela garantit plus de sécurité puisque les calculs seront plus long si jamais on veut falsifier des blocs. On a décidé de présenter un mineur comme étant la propriété d'une personne.

Lorsque le mineur en compétition avec les autres réussit à miner un bloc avant ses concurrents il reçoit une récompense en MC qui est directement transféré dans le portefeuille du propriétaire.

```
Block 00000d3c918c2f1ee43ac00f037aa03b4cdedfe7f3cc2b5078091ee2609cb4cc
--------------------------------------------------
Mineur Rtx4080 a été trop lent
Mineur Rtx4060 a été trop lent
Mineur Rtx4090 a été trop lent
Mineur Rtx4060ti a été le plus performant, Mugsy99 remporte 50MC
Mineur Rtx4070 a été trop lent
Mineur Gtx1650 a été trop lent
--------------------------------------------------
```

FIGURE 8 – Exemple des mineurs en compétition

3.3.1 Halving

Pour notre simulation on a 5 personnes, chaque personne possède des mineurs et peut faire des transactions. Au début chaque personne à 100 MC dans son portefeuille, donc à l'état 0 on 500 MC en circulation, limitons notre stock de MC pour que cette monnaie ne soit pas illimité, on a décidé de limiter cette monnaie à 1500 unitées.

Objectif : Décider d'une récompense pour les mineurs tel qu'on dépasse jamais 1500 MC en circulation. Si on décide de récompenser 50 MC par bloc miné, on se rend compte qu'on dépasse le stock au bout de 20 blocs, le Halving consiste à diviser par deux la récompense tout les x blocs minés pour ne jamais atteindre 1500 MC.

On cherche x tel que $\sum_{i=0}^{\infty} \frac{50x}{2^n} = 50x * \sum_{i=0}^{\infty} \frac{1}{2^n} = 100x = 1500 - 500 = 1000$ on a alors $x = 10$ Tout les 10 blocs minés il faut diviser la récompense par 2. Cela prend un peu de temps mais on peut le simuler sur notre programme.

4 SIMULATION

Afin de pouvoir simuler le fonctionnement de la Blockchain dans un cadre plus pratique, nous avons crée deux classes Personne et Mineur pour pouvoir réaliser les transactions et miner d'une façon un peu plus réaliste les blocs crées.

4.1 Quelques modifications

Lors de l'implémentation de la simulation, nous avons remarqué quelques erreurs d'implémentation nottament dans les blocs et dans la Blockchain. Voici les changements accompagnés des explications.

— Classe Block : on ne calcule plus la *Proof of Work* lors de l'initialisation, c'est le mineur qui s'en occupera lors du minage du bloc. On l'initialise donc à 0.

— Classe Blockchain : dans la fonction add(block), on ne recalcule plus la *Proof of Work*, c'est le mineur qui s'en occupera lors du minage du bloc. Idem pour la mise à jour du prédecesseur du bloc ajouté.

4.2 Classes Personne et Mineur

Nous avons défini une personne comme quelqu'un avec un nom, un portefeuille, une clé privée et une clé publique. On peut modifier son portefeuille avec la fonction modifyMC(amount). Cela est utilisé dans la classe Transaction comme vous pouvez observez dans le point 3.1.1.

```
class Personne:
    def __init__(self, name, wallet, mdp):
        self.name = name
        self.wallet = wallet
        self.sk = PrivateKey(hash256(mdp.encode()))
        self.pk = self.sk.secret*G

    def __repr__(self):
        return "{} possède {}MC".format(self.name, self.wallet)

    def modifyMC(self, amount):
        self.wallet += amount
```

FIGURE 9 – Classe Personne

Un mineur est intialisé comme ayant un nom et un propriétaire (pour les éventuelles récompenses). Pour miner un bloc, il possède une fonction minage(bloc, bc) qui prend comme paramètre le bloc à miner et la blockchain où on va ajouter ce bloc. Le mineur mettra à jour le prédecesseur du bloc, calculera la *Proof of Work* et changera le hash.

```
def minage_bloc(self, block, bc):
    if bc.size > 0:
        block.prev = bc.chain[-1].hash
    block.pW = block.proofOfWork()
    block.hash = block.get_hash()
    return block
```

FIGURE 10 – Fonction minage

5 CONCLUSION

On a réussi à signer et vérifier des transactions, en s'aidant de la courbe elliptique utilisée par le bitcoin, en travaillant sur le sujet on se rend vite compte que c'est impossible d'imiter une signature à moins de connaitre la clé privée de la personne. La validité d'une transaction est définie par la signature et si l'envoi de monnaie est possible (on ne peut pas envoyer plus de MC qu'on en possède). Ensuite nous avons présenté notre version et compréhension d'un bloc (Prev code, 5 transactions, proof of work). Et à la fin nous avons présenté le moteur de la blockchain : le mineur.
Sur ce dernier point il y a une limite : lorsqu'on récompense un mineur on doit noter cette "transaction" dans un bloc afin de tracer la récompense. Ce n'est pas le cas dans ce projet on a décidé de se faciliter la tâche.
Pour continuer sur les limites on peut parler de la simulation peu réaliste, pour simuler des transactions au lieu de laisser les personnes choisir les montant et le destinataire on le génére de façon aléatoire pour ne pas perdre beaucoup de temps, et pour voir l'effet du halving il faut miner 20-30 blocs donc il faut écrire plus de 100 transactions c'est pour cela qu'on a décider de générer les transactions aléatoirement et automatiquement.
Pour conclure, ce projet était enrichissant et n'est pas limité, on se rend bien compte que notre rendu est une version très simplifiée et qu'on peut pousser la théorie et la programmation encore plus loin. On s'est amusés !