
DÉVELOPPEMENT D'UNE BLOCKCHAIN POST-QUANTUM RESISTANT

PIR / ADL

MASTER 1 CRYPTIS - Université de Limoges

Baptiste COUPRY

Anthony FRAGA

Eléonore VAN DER PLOEG

2023-2024

Table des matières

0.1	Introduction	2
1	La blockchain	3
1.1	Rappel du fonctionnement d'une blockchain	3
1.1.1	Généralités	3
1.1.2	Le wallet	4
1.1.3	Le consensus	5
2	Construire le wallet	6
2.1	Quel algorithme de signature ?	6
2.1.1	Au revoir les signatures de Lamport	6
2.1.2	Passage à MQOM	7
2.1.3	Passage à Dilithium	11
2.2	Construire le wallet	19
2.2.1	Générer les clés et l'adresse	19
2.2.2	Stocker les données	22
3	Construire le consensus	38
3.1	De la <i>Proof of Work</i> aux VDFs	39
3.1.1	Qu'est ce qu'une VDF ?	40
3.1.2	Avantages & inconvénients	41
3.1.3	L'exemple de Solana	42
3.1.4	Une VDF basée sur les isogénies	42
3.1.5	Notre choix : Une VDF basée sur les fonctions de hachage	44
3.1.6	Fonctionnement de la VPoRS	50
3.1.7	Devenir staker	52
4	Conclusion	54
4.1	Ce qu'il reste à faire	54
4.1.1	Construire les blocs	54
4.1.2	Construire le réseau	55

0.1 Introduction

Pour faire suite à notre rapport du premier semestre détaillant notre projet et les problématiques que nous avons pour mission de résoudre, nous vous présentons aujourd'hui notre solution à la réalisation d'une blockchain post-quantum résistant. Ce premier rapport détaillait les premières solutions que nous avions eu, celles-ci ont été revues en profondeur afin d'obtenir un résultat davantage qualitatif sur bien des aspects de notre blockchain. Ainsi, nous n'utiliserons finalement pas les signatures de Lamport ainsi que le consensus de la Proof of Work. Ces choix vous seront présentés à travers les différentes parties de ce document.

Nous rappellerons tout d'abord quelques généralités sur le fonctionnement de la blockchain avant d'étudier les étapes traversées pour travailler avec un algorithme de signature correspondant à nos attentes. Ce second semestre étant en partie consacrée au développement, nous présenterons ensuite notre wallet par ses différentes versions avant de poursuivre sur l'élaboration de notre consensus. N'ayant pu clore le développement informatique de notre blockchain, les parties suivantes sur la construction des blocs, des transactions ainsi que du réseau resteront des parties théoriques. L'objectif de ce rapport est de présenter nos réponses face aux obstacles rencontrés mais aussi de fournir le raisonnement détaillé derrière le fonctionnement de notre blockchain PQR que nous avons nommé "Qrypto". Ce nom a été trouvé pour faire référence à la partie post-Quantique dans notre blockchain. Chaque blockchain ayant un token associé, soit, une cryptomonnaie, ce nom nous paraissait être le bon pour faire comprendre rapidement notre objectif derrière le projet.

Note : Notre projet se situe sur le lien Github ci-contre : <https://github.com/Baptoo10/qrypto.git>

Chapitre 1

La blockchain

1.1 Rappel du fonctionnement d'une blockchain

Cette partie est utile afin de rappeler rapidement le fonctionnement d'une blockchain par ses concepts clés. Le rendu du premier semestre détaillait davantage son fonctionnement et son utilité.

1.1.1 Généralités

La blockchain, littéralement la "chaîne de blocs" est une base de données distribuée reposant sur un réseau décentralisé. Les utilisateurs d'une blockchain vont y ajouter des données en envoyant ce qu'on appelle des transactions. Ces blocs correspondent à l'espace où sont stockées ces données. Une transaction correspond donc à l'envoi de données d'un utilisateur A à un utilisateur B.¹ Cette transaction est avant tout transmise à un certain type de noeud du réseau qui va ajouter cette transaction dans un bloc qu'il crée. Enfin, lorsque son bloc est prêt, qu'il est considéré complet par le noeud, celui-ci va obéir à un consensus qui définit les paramètres de la blockchain. Une fois que celui-ci a répondu à ces règles, le bloc peut être transmis aux autres noeuds du réseau via un mécanisme P2P (pair à pair). Le consensus est l'essence de la blockchain, c'est grâce à lui que le protocole est dessiné.

La blockchain étant une base de données distribuée, elle est aussi transparente, tout le monde peut y lire les informations saisies et ainsi vérifier les données incluses ou encore le bon respect des règles du consensus par le noeud ayant créé le bloc. En fonction du type de consensus, ce noeud pourra être appelé "mineur" (chaîne avec un consensus du type Proof of Work/PoW) ou "validateur" (chaîne avec un consensus du type Proof of Stake/PoS).

Ces blocs sont interdépendants, de fait, modifier le contenu d'un bloc x modifiera la validité des blocs supérieurs dans le rang. Puisque le contenu de la blockchain est

1. Sous certaines conditions et sous certaines architectures, A peut s'auto-envoyer des transactions. On retrouve ce fonctionnement avec le protocole Ordinal utilisé sur la blockchain Bitcoin.

visible par tous, alors ces changements seront remarqués et la communauté prendra en conséquence une décision face au problème d'intégrité qui se pose². Il est ainsi possible de comprendre la blockchain comme un registre distribué, soit accessible à tous et à tout moment, immuable et sans intermédiaire autre que le mineur/validateur.

La figure 1.1 présente une illustration de la représentation que l'on peut se faire de la blockchain.

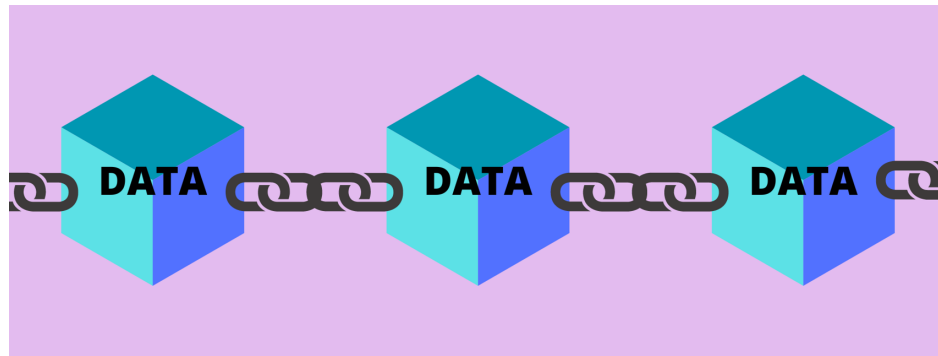


FIGURE 1.1 – Représentation visuelle d'une blockchain

1.1.2 Le wallet

Pour envoyer et recevoir des transactions, il vous faudra une adresse d'émission et de destination. Cette adresse est obtenue grâce à une dérivation de la clé publique de l'utilisateur, elle même générée à partir d'un algorithme de signature qui va créer la clé privée associée. Cette adresse est unique et différentes versions co-existent sur un réseau. Pour bien reconnaître le réseau ainsi que la version, ces adresses ont bien souvent un préfixe qui détaille ces informations. Pour illustrer nos dires, l'adresse Bitcoin "bc1p5d7rjq7g6rdk2yhzks9smlaqtedr4dekq08ge8ztwac72sfr9rusxg3297" représente une adresse de type P2TR, soit, une adresse créée à la suite de la mise à jour (on parlera ici de soft-fork) Taproot. Celle-ci se reconnaît par le préfixe "bc1p".

Cette adresse est codée en base58 pour éviter toutes mauvaises copies, dans le cas où l'utilisateur souhaiterait recopier à la main son adresse (déconseillé). Effectivement, ce type d'encodage élimine les lettres et chiffres trop semblables (le "l" et le "I", le "0" et le "O"). Afin de rester dans cette norme d'usage, nous générons nos adresses avec le même encodage, nous y reviendrons dans la partie dédiée de ce rapport.

Ces adresses sont donc obtenues à la suite de chemins de dérivation à partir d'une clé publique. Cette clé publique et sa clé privée associée sont donc générées à partir d'un algorithme qui sera aussi utilisé pour la signature. Comme développé dans notre précédent rapport, Bitcoin utilisait ECDSA avant de se tourner vers SCHNORR.

Enfin, ces données (clés et adresse(s)) sont stockées dans le wallet. C'est ici que

2. Un hard-fork peut être suggéré, c'est à dire diviser en 2 parties la chaîne à partir du bloc corrompu pour repartir sur de nouvelles bases.

toutes les actions d'un utilisateur seront sauvegardées. Si des bitcoins sont envoyées ou reçus, alors certains scripts (propres au type de l'adresse) sont exécutés³.

1.1.3 Le consensus

Vous l'aurez compris, sans consensus une blockchain ne répondrait à aucune règle et ne pourrait donc pas fonctionner. Le consensus correspond ainsi au travail que doit réaliser un noeud pour partager au réseau le bloc qu'il crée. Le consensus et l'ensemble des éléments gravitant autour permettant à la blockchain de fonctionner définissent le protocole. Dans notre cas nous avons nommé notre blockchain "Qrypto", ainsi, le nom du protocole que nous avons développé est aussi "Qrypto".

Pour illustrer son fonctionnement, prenons la Proof of Work (que nous abrégons désormais PoW) qui est le consensus utilisé sur le réseau Bitcoin.

Le mécanisme du PoW consiste au calcul d'un hash de bloc débutant par un certain nombre 0 ou, plus précisément, un hash ayant une valeur, en base 10, inférieure ou égale à une valeur cible nommée "Target". Puisqu'une information ne peut avoir qu'une seule empreinte, on retrouve dans le Blockheader d'un bloc un "Nonce". Ce nonce est un nombre à usage unique qui est l'unique valeur que le mineur peut faire varier afin d'obtenir une empreinte de bloc valide. C'est donc ce "Nonce" que fait varier, par brute force, le mineur pour miner son bloc.

Le premier mineur ayant obtenue une empreinte valide pour son bloc partage celui-ci au réseau par une technique de flooding, ce qui permet de répandre les données sur l'intégralité du réseau.

3. Citons le scriptSig et le PublicKeyScript sur Bitcoin

Chapitre 2

Construire le wallet

2.1 Quel algorithme de signature ?

2.1.1 Au revoir les signatures de Lamport

Notre choix premier pour générer une signature et qui puisse résister à une attaque quantique réalisée avec l'algorithme de Shor a été d'utiliser les signatures de Lamport. Cependant, ce choix nous contraignait pour 2 raisons principales. Premièrement, la taille des signatures produites est bien trop importante. Or, dans une blockchain, afin que son réseau soit le plus décentralisé possible, il est nécessaire que quiconque puisse télécharger son contenu pour prendre le rôle de nœud. Si la blockchain est trop lourde, tout le monde ne pourra pas la faire tourner chez soi et seuls les parties avec le plus d'espace de stockage le pourront, centralisant alors le pouvoir. La centralisation étant en opposition avec les principes fondamentaux de la blockchain. En effet, pour un message m de 256 bits et une fonction de hachage qui renvoie en sortie une empreinte de 256 bits, comme le fait la fonction SHA-256, alors la taille des signatures est de $256 \times 256 = 65536$ bits.

Aussi, les signatures produites sont à usage uniques, ce qui ajoutait de la complexité dans l'architecture de notre blockchain.

Nous voulions donc trouver un algorithme de signature qui pouvait correspondre à nos attentes en proposant :

- Un schéma PQR prouvé
- Des tailles de clés acceptables
- De la rapidité dans la génération des clés et des signatures

C'est de par ces réflexions que nous nous sommes ainsi tourné, dans un premier temps, vers un algorithme de signature PQR actuellement proposé pour la compétition du NIST visant à standardiser de nouveaux algorithmes de signatures PQR¹.

1. La compétition "Post-Quantum Cryptography: Digital Signature Schemes" en est actuellement à son premier tour.

2.1.2 Passage à MQOM

Dans ce premier temps, nous nous sommes tournés sur l'algorithme de signature MQOM (pour *Multivariate Quadratic On-my-Mind*), un algorithme développé et proposé au concours du NIST par des chercheurs français. Nous verrons dans cette partie sur quoi cet algorithme de signature est basé et son fonctionnement. Mais tout d'abord, nous avons besoin de quelques rappels sur les polynômes multivariés et le domaine cryptographique associé.

La cryptographie multivariée

Définition 1. Soit K un corps, et X_1, \dots, X_n des indéterminées. Un polynôme $f \in K[X_1, \dots, X_n]$ est une combinaison linéaire de monômes de la forme $X_1^{\alpha_1} X_2^{\alpha_2} \dots X_n^{\alpha_n}$ à coefficient dans K .

Exemple 1. L'anneau de polynômes à deux indéterminées $K[x, y]$ est souvent utilisé pour la représentation de courbes planes sur le plan K^2 .

$$f(x, y) = x^3 + 3x^2y - 2xy + 3x - 8y \quad g(x, y) = x^4 + y^4 \quad h(x, y) = x^2 + xy + y^2$$

Ces polynômes sont des exemples d'éléments de cet anneau.

Dans le cadre de l'algorithme MQOM, nous nous plaçons sur un corps fini $K = \mathbb{F}_q$, et nous nous intéresserons aux monômes de degré 2 (de la forme x^2, xy, y^2).

Proposition 1. Soit $X = (X_1, \dots, X_n)$, et $f \in K[X_1, \dots, X_n]$ un polynôme constitué de monômes de degré 2. Il existe une matrice $A \in \mathcal{M}_n(K)$ telle que $f(X) = X^T A X$.

Exemple 2. Dans $\mathbb{R}[x, y]$, le polynôme $x^2 + 4xy + y^2$ peut être représenté par la matrice $A = \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix}$:

$$\begin{aligned} X^T A X &= \begin{pmatrix} x & y \end{pmatrix} \begin{pmatrix} 1 & 3 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x + y & 3x + y \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\ &= (x^2 + xy + 3xy + y^2) = (x^2 + 4xy + y^2) \end{aligned}$$

De manière équivoque, nous pouvons construire des polynômes de degré 1 avec simplement un vecteur de coefficient et un vecteur d'indéterminés, que l'on notera plus tard $b^T X$.

L'algorithme est basé sur un problème compliqué du domaine des polynômes multivariés, qui est le suivant :

Le problème quadratique multivarié (*MQ problem*)

Nous noterons $q = p^s$ la puissance d'un nombre premier, et n la dimension du \mathbb{F}_q -espace vectoriel.

Problème MQ

Étant donné $\{A_i\}_{1 \leq i \leq m}$, $\{b_i\}_{1 \leq i \leq m}$, x and y tel que :

1. $x \in \mathbb{F}_q^n$ généré aléatoirement ;
2. $\forall i \in \llbracket 1, m \rrbracket, A_i \in \mathbb{F}_q^{n \times n}$ généré aléatoirement ;
3. $\forall i \in \llbracket 1, m \rrbracket, b_i \in \mathbb{F}_q^n$ généré aléatoirement ;
4. $\forall i \in \llbracket 1, m \rrbracket, y_i := x^\top A_i x + b_i \top x$

QUESTION : A partir de $(\{A_i\}, \{b_i\}, y)$, trouver x .

Théorème 1. *MQP est un problème de type NP-complet.*

Le problème réside sur le fait que l'on ne sache pas résoudre un système d'équations quadratiques, que ce soit avec un ordinateur classique (car ce n'est pas linéaire), ni avec les capacités d'un ordinateur quantique

$$\begin{cases} y_1 &= \left(\sum_{i=1}^n \sum_{j=1}^n \alpha_{i,j}^{(1)} x_i x_j \right) + \left(\sum_{i=1}^n \beta_i^{(1)} x_i \right) \\ y_2 &= \left(\sum_{i=1}^n \sum_{j=1}^n \alpha_{i,j}^{(2)} x_i x_j \right) + \left(\sum_{i=1}^n \beta_i^{(2)} x_i \right) \\ \vdots & \quad \quad \quad \vdots \\ y_m &= \left(\sum_{i=1}^n \sum_{j=1}^n \alpha_{i,j}^{(m)} x_i x_j \right) + \left(\sum_{i=1}^n \beta_i^{(m)} x_i \right) \end{cases}$$

L'algorithme MQOM

Avant de passer à l'algorithme MQOM, qui nous le verrons est une suite de protocoles, il semble nécessaire de rappeler certaines notions et principes des protocoles utilisés.

MPC pour *Multi-Party Computation* (ou calcul multipartite) est un protocole cryptographique permettant à plusieurs parties de calculer une fonction ensemble par rapport à leurs entrées, mais tout en faisant en sorte qu'elles restent secrètes.

En notant les participants p_1 jusqu'à p_N , avec d_1, \dots, d_N leur donnée privée respective, le but est de calculer l'image d'une fonction F en prenant en entrée toutes les données. Autrement dit, on veut calculer $F(d_1, \dots, d_N)$, sans que l'on puisse retrouver une donnée d_i privée. C'est là tout l'enjeu des protocoles MPC, qui doivent ruser pour contourner ce problème.

MPCitH pour *MPC in the Head* est une méthode utilisée pour passer d'un protocole de MPC à un protocole à divulgation nulle de connaissance (ZKP, pour *Zero Knowledge Proof*).

Un protocole ZKP demande l'engagement de deux individus, un prouveur (qui connaît la donnée) et un vérifieur (qui cherche à se persuader que le prouveur possède

la donnée). Le but de cette méthode est, par l'itération de protocoles défi/réponse, de montrer au vérifieur que le prouveur est bien en connaissance de la donnée, sans pour autant donner d'information sur celle-ci.

Dans le contexte de la MPCitH, le prouveur doit simuler le protocole MPC avec une partie de taille N , qui vérifie que w est un témoin (*witness* en anglais, qui varie selon les modèles) correct pour la donnée x , qui est solution du problème MQ. A la suite de se premier échange, une suite de défis/réponses est mise en route, à la méthode d'une ZKP.

A noter que l'on peut se retrouver fasse à un témoin faussement positif, avec une probabilité de $\frac{1}{N}$ par défi, où N est le nombre de parties. Pour faire baisser cette probabilité, nous augmentons le nombres de challenges.

Heuristique de Fiat-Shamir ou transformation de Fiat Shamir, permettant de transformer un protocole de ZK à un protocole de signature (ou une preuve non interactive à divulgation nulle de connaissance).

Heuristique de Fiat-Shamir

Soit \mathcal{E} un protocole de ZK, pour prouver la connaissance de $(x, w) \in X \times W$. On notera l'ensemble d'engagement \mathcal{A} (éléments que le prouveur peut envoyer au vérifieur); l'ensemble des challenges \mathcal{C} , que le vérifieur renvoie au prouveur; et l'ensemble des réponses de prouveur \mathcal{R} .

La preuve non-interactive se déroule de la manière suivante :

- le prouveur fabrique une fonction de hachage $\mathcal{H} : \mathcal{A} \times \mathcal{X} \rightarrow \mathcal{C}$;
- le prouveur génère un élément $a \in \mathcal{A}$ tout comme dans la version interactive. Ensuite, il calcule $\mathcal{H}(a, x) := c$ un haché et donne une réponse r correspondante au challenge. Il envoie $\pi = (a, r)$ comme preuve.
- pour la vérification, il suffit de haché (a, x) pour obtenir c , et vérifier que r est une réponse adéquate au couple engagement-défi (a, c) .

L'algorithme MQOM est un combiné de tous ces protocoles cryptographique, dans l'ordre, permettant de passer d'un protocole MPC à un protocole de signature. Voici comment cela se produit :

- Le protocole MPCitH (de base) est le suivant. Le prouveur est en connaissance de la solution x du problème MQ. Le vérifieur demande à passer une série de trois défis au prouveur, si bien que cela revient finalement à effectuer le protocole MPC.
- La transformée de Fiat Shamir transforme le protocole ci-dessus en un protocole de signature, de la même manière que dans le bloc ci-dessus. Nous obtenons un premier hash, dérivé du premier partage correspondant au témoin, que nous concaténons avec le premier défi. Le second et le troisième hash sont obtenus de manière analogue, avec la même concaténation. La vérification se déroule de la

manière dont nous l'avons détaillé ci-dessus.

Avantages & inconvénients de l'algorithme

Avantages de MQOM en comparaison avec les algorithmes de signatures classiques, et des protocoles proposés dans le concours du NIST.

Tailles de clés relativement petites en comparaison avec les algorithmes déjà existant et ceux proposés dans l'actuelle compétition du NIST. Pour le niveau 3 de sécurité, nous pouvons compter sur une taille² comprise entre 73 et 92 octets pour la clé publique et entre 78 et 102 octets pour la clé secrète.

Fortement parallélisable comme les protocoles basées sur MPCitH. La parallélisation est effectuée sur les calculs des parties, au cours des défis donnés et des répétitions de ceux-ci, ainsi que sur la communication des données.

Inconvénients

Taille de clés relativement grandes en comparaison avec les protocoles basées sur les réseaux euclidiens. Même si d'autres protocoles produisent des tailles encore plus grandes, il faut se rendre à l'évidence que ce ne sont pas les meilleurs sur cet aspect. Nous pouvons compter une taille moyenne de 15000 octets pour le niveau 3 de sécurité demandé par le NIST.

Augmentation quadratique des tailles de signatures pour l'augmentation de la sécurité, comme les transformations de Fiat-Shamir sur les répétitions de ZKP.

Relativement lent en comparaison avec les autres protocoles basés sur MPCitH, et aux protocoles de signatures basés sur les réseaux euclidiens. C'est en grande partie à cause de l'utilisation des systèmes de cryptographie symétriques, tels que les fonctions de hachages. En pratique, le temps de signature et de vérification se situe entre $10 \cdot 10^6$ et $45 \cdot 10^6$ cycles pour la catégorie 1 du NIST.

Conclusion

Les inconvénients de cet algorithme sont plutôt contraignant dans notre cas puisque nous souhaitons avoir des tailles de signatures petites, ainsi qu'un algorithme plutôt rapide.

Si l'on observe les comparaisons effectuées avec d'autres protocoles soumis à la même compétition, nous remarquons que certains sont plus adaptés à notre besoin.

2. Il existe plusieurs dérivées de l'algorithme, comme dans le choix du corps de base (GF_{31} et GF_{251}), ainsi que dans le choix de notre priorité (entre rapidité et taille courte).

C'est principalement pour cette raison que nous nous sommes tournés finalement vers l'algorithme de signature Dilithium, qui est basé sur les réseaux euclidiens.

2.1.3 Passage à Dilithium

Dilithium, tout comme MQOM, est un algorithme de signature qui a été soumis au NIST lors d'un concours de PQC. A la différence de MQOM, Dilithium a été finaliste du précédent concours et va donc être standardisé. Cet aspect nous était important puisqu'il prouvait la sécurité de l'algorithme, là où MQOM ne l'a pas (encore) été. Ainsi, il permet tout comme son précédent d'assurer une protection face à la menace quantique. Le plus de Dilithium, face à ses adversaires, se porte sur la taille des signatures, qui sont relativement petite pour un protocole PQR.

Avant d'analyser comment fonctionne l'algorithme, nous allons introduire quelques notions sur les réseaux euclidiens.

Les réseaux euclidiens

Définition 2. Soit $(A, +, \cdot)$ un anneau commutatif unitaire. On appelle A -module, ou module sur A , tout ensemble M tel que :

1. $(M, +)$ est un groupe abélien ;
2. Pour $n, m \in M, \lambda, \mu \in A$, on a :

$$\begin{aligned} 1 \cdot n &= n \\ \lambda \cdot (n + m) &= \lambda \cdot n + \lambda \cdot m \\ (\lambda + \mu) \cdot n &= \lambda \cdot n + \mu \cdot n \\ (\lambda \cdot \mu) \cdot n &= \lambda \cdot (\mu \cdot n) \end{aligned}$$

Exemple 3. L'exemple d'anneau le plus parlant, et qui est utilisé dans les réseaux euclidiens, est l'anneau \mathbb{Z} . Pour tout entier naturel n , l'ensemble \mathbb{Z}^n est un \mathbb{Z} -module.

Nous utiliserons dans la suite \mathbb{Z} comme anneau, si bien que nous pouvons définir les réseaux euclidiens.

Définition 3. Un réseau euclidien de dimension n est un \mathbb{Z} -module de rang n dans \mathbb{R}^n .

En notant $(e_i)_{1 \leq i \leq n}$ une base, linéairement indépendante dans \mathbb{R}^n , d'un réseau euclidien L , on a :

$$L = \left\{ \sum_{i=1}^n a_i e_i, \forall i = 1, \dots, n, a_i \in \mathbb{Z} \right\}$$

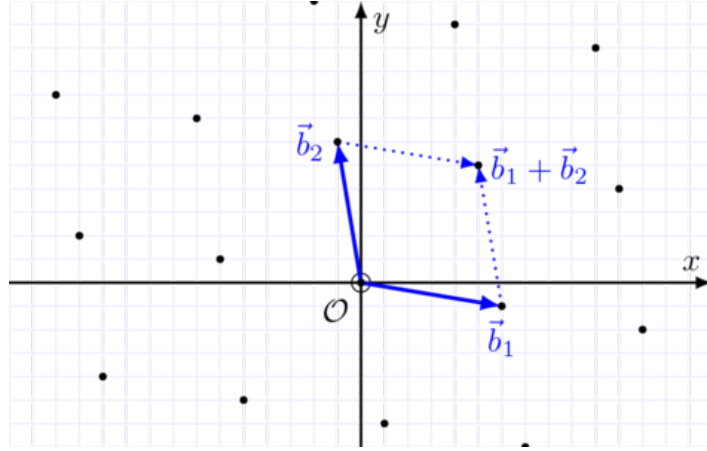


FIGURE 2.1 – Réseaux euclidien $L = \{\alpha \vec{b}_1 + \beta \vec{b}_2, \alpha, \beta \in \mathbb{Z}\}$, représenté par les points noirs.

Base orthogonale

Définition 4. Une base $\mathcal{B} = \{e_1, \dots, e_n\}$ d'un espace vectoriel est dite orthogonale lorsque pour tout couple $i, j \in \llbracket 1, n \rrbracket, i \neq j$, on a :

$$\langle e_i, e_j \rangle = 0$$

Pour un K -espace vectoriel E de dimension n , il existe une méthode pour orthogonaliser une base : l'orthogonalisation de Gramm-Schmidt.

En revanche, en ce qui concerne un \mathbb{Z} -module L de dimension $n > 2$, il n'existe pas de méthode efficace pour obtenir le même résultat. Cela provient du fait que nous ne nous situons pas dans un corps, nous ne pouvons alors pas être certain qu'il existe des éléments orthogonaux 2 à 2. Sur le graphique 2.1, les vecteurs \vec{b}_1 et \vec{b}_2 ne sont pas orthogonaux.

C'est de ce résultat que se construit la cryptographie sur les réseaux euclidiens. Il est en effet à l'origine du problème du vecteur le plus court (SVP, pour *Short Vector Problem* en anglais), un problème fondamental de ce domaine.

La cryptographie basée sur les réseaux euclidiens

Pour un réseau L , nous noterons $\lambda(L) := \min\{\|\vec{u}\|_N, \vec{u} \in L \setminus \{\vec{0}\}\}$ le vecteur avec la plus courte distance selon une norme N .

Initialement, le problème du vecteur le plus court, qui est pilier de la cryptographie sur les réseaux, s'annonce ainsi :

Problème du vecteur le plus court

Étant donné une base d'un réseau euclidien L muni d'une norme N , trouver le vecteur $\vec{v} \in L$ le plus petit non nul au sens de la norme. Autrement dit, trouver le vecteur $\vec{v} \in L$ tel que :

$$\|\vec{v}\|_N = \lambda(L)$$

En général, la norme N représente la norme euclidienne. Pour des raisons d'optimisation, Dilithium utilise la norme infinie.

Théorème 2. *Le SVP est un problème NP-difficile, ce qui veut dire que tout problème NP peuvent être réduit en temps polynomial à ce problème.*

Il existe un problème plus générique et plus faible, appelé SVP_γ (qui prend un certain $\gamma \leq 1$ en compte), qui est le suivant :

 SVP_γ

Étant donné une base d'un réseau euclidien L muni d'une norme N , trouver un vecteur $\vec{v} \in L$ tel que :

$$\|\vec{v}\|_N \leq \gamma \cdot \lambda(L)$$

Toutefois, nous nous intéresserons à un problème sous-jacent de ce dernier, appelé problème de la solution courte entière (SIS, ou *Short Integer Solution*), sur les réseaux modulo q . Dans le cas de Dilithium, les réseaux considérés sont des modules de dimensions d d'anneaux de polynômes, que nous noterons ainsi :

$$R = \mathbb{Z}[X]/(X^n - 1) \quad R_q \simeq \mathbb{Z}_q[X]/(X^n - 1)$$

 $SIS_{n,m,d,q,\beta}$

Étant donné $a = (a_1, \dots, a_m)$ généré aléatoirement, avec $a_i \in R_q^d$ pour tout $i \in \llbracket 1, m \rrbracket$, trouver un vecteur $\vec{x} = (x_1, \dots, x_m) \in R^m$ tel que :

- $0 < \|\vec{x}\|_N \leq \beta$;
- $\vec{a} \cdot \vec{x} \langle a_i, x_i \rangle = 0_{R_q^d}$.

Ces problèmes sont à la base de la cryptographie sur les réseaux, et sont à l'origine de la sécurité de l'algorithme Dilithium.

Proposition 2. *En reprenant les notations du problème, on considère $h : \vec{x} \rightarrow \vec{a} \cdot \vec{x}$ comme fonction de hachage. Cette fonction est résistante à la collision.*

Démonstration. Soit $D_\beta^m := \{\vec{x} \in R_q^m, \|\vec{x}\| \leq \beta\}$. On suppose que l'on définit la fonction $h : D_\beta^m \rightarrow R_q$.

Supposons que l'on trouve \vec{x}_1 et \vec{x}_2 deux vecteurs formant une collision, c'est à dire $\vec{a} \cdot \vec{x}_1 \equiv \vec{a} \cdot \vec{x}_2$. Alors $\vec{x}_1 - \vec{x}_2$ est une solution du problème $SIS_{n,m,d,q,2b}$.

En effet, en supposant que l'on ait $\vec{a} \cdot \vec{x}_1 - \vec{a} \cdot \vec{x}_2 = \vec{a} \cdot (\vec{x}_1 - \vec{x}_2) \equiv \vec{0}$, avec $\|x_1\|$ et $\|x_2\|$ court. Alors, par l'inégalité triangulaire, on obtient

$$\|\vec{x}_1 - \vec{x}_2\| \leq \|\vec{x}_1\| + \|\vec{x}_2\| \leq 2b$$

Nous obtenons une solution du problème $SIS_{n,m,d,q,2b}$. □

L'algorithme Dilithium

Nous avons vu que la sécurité de Dilithium reposait sur la difficulté que l'on a à résoudre le problème SIS. Désormais, nous allons voir comment nous obtenons un algorithme de signature.

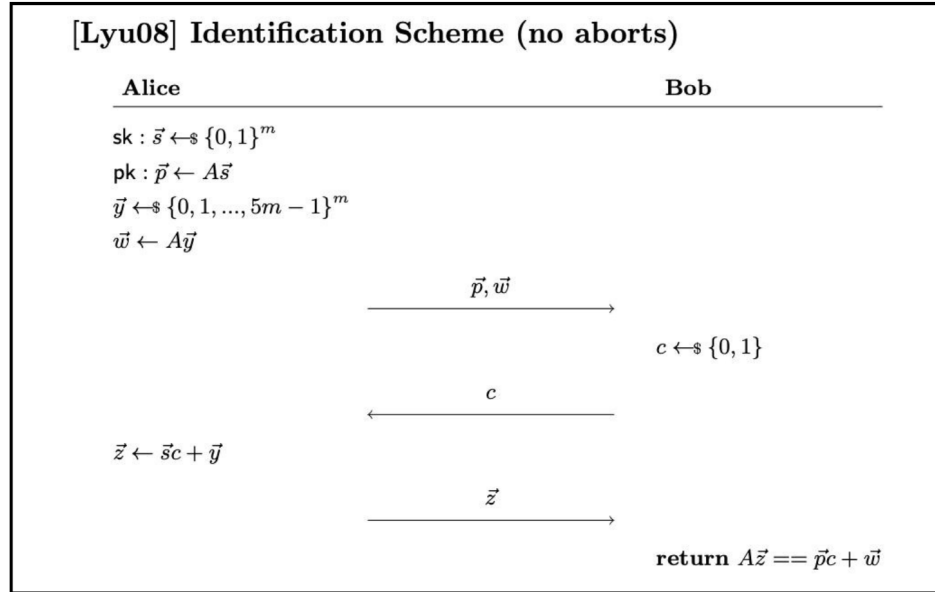


FIGURE 2.2 – Schéma d'algorithme de signature avec la transformation de Fiat-Shamir (Schéma repris du site web ici)

Dans la partie 2.1.2, nous avons vu que la transformée de Fiat-Shamir permet, à partir d'un protocole de ZKP, de faire un protocole de signature (2.2). Le problème avec le schéma qui est proposé ci-dessus, est qu'il est possible que le vérificateur récupère des informations sur la clé secrète. Par exemple, si $c = 1$, il est possible de récupérer des informations sur la clé privée \vec{p} (avec un peu de travail).

Pour résoudre ce problème, la solution qui a été trouvée est de changer \vec{z} en *None* lorsque celui-ci peut faire fuiter des informations sur \vec{p} . On répète autant de fois que nécessaire jusqu'à ce que l'on obtienne un vecteur sans informations sensibles.

Finalement, l'algorithme Dilithium est basé sur l'idée du schéma affiché ci-dessous.

```

Gen
01  $\mathbf{A} \leftarrow R_q^{k \times \ell}$ 
02  $(\mathbf{s}_1, \mathbf{s}_2) \leftarrow S_\eta^\ell \times S_\eta^k$ 
03  $\mathbf{t} := \mathbf{A}\mathbf{s}_1 + \mathbf{s}_2$ 
04 return  $(pk = (\mathbf{A}, \mathbf{t}), sk = (\mathbf{A}, \mathbf{t}, \mathbf{s}_1, \mathbf{s}_2))$ 

Sign( $sk, M$ )
05  $\mathbf{z} := \perp$ 
06 while  $\mathbf{z} = \perp$  do
07    $\mathbf{y} \leftarrow S_{\gamma_1 - 1}^\ell$ 
08    $\mathbf{w}_1 := \text{HighBits}(\mathbf{A}\mathbf{y}, 2\gamma_2)$ 
09    $c \in B_\tau := H(M \parallel \mathbf{w}_1)$ 
10    $\mathbf{z} := \mathbf{y} + c\mathbf{s}_1$ 
11   if  $\|\mathbf{z}\|_\infty \geq \gamma_1 - \beta$  or  $\|\text{LowBits}(\mathbf{A}\mathbf{y} - c\mathbf{s}_2, 2\gamma_2)\|_\infty \geq \gamma_2 - \beta$ , then  $\mathbf{z} := \perp$ 
12 return  $\sigma = (\mathbf{z}, c)$ 

Verify( $pk, M, \sigma = (\mathbf{z}, c)$ )
13  $\mathbf{w}'_1 := \text{HighBits}(\mathbf{A}\mathbf{z} - c\mathbf{t}, 2\gamma_2)$ 
14 if return  $\llbracket \|\mathbf{z}\|_\infty < \gamma_1 - \beta \rrbracket$  and  $\llbracket c = H(M \parallel \mathbf{w}'_1) \rrbracket$ 

```

FIGURE 2.3 – Algorithme Dilithium, simplifié (Algorithme tiré du document CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation).

Notre choix : avx2-Dilithium3-AES-R

Pour les raisons évoquées ci-dessus, notre choix s'est naturellement tourné sur Dilithium. Toutefois, il existe diverses versions standardisées de Dilithium : ³

- dilithium2
- dilithium2-AES
- dilithium2-AES-R
- avx2-dilithium2-AES-R
- dilithium3
- dilithium3-AES
- dilithium3-AES-R
- avx2-dilithium3-AES-R
- dilithium5
- dilithium5-AES
- dilithium5-AES-R
- avx2-dilithium5-AES-R

Pour faire notre choix, nous avons dû comprendre ce que chaque version apportait. Ce que nous avons donc pu comprendre c'est qu'avant tout la version 3 était davantage

3. Le fichier "dilithium_methodes.txt" fournit dans l'archive de notre projet explique de façon succincte l'utilité et le contenu des différents fichiers présents dans Dilithium.

recommandée par l'équipe de développement : "We recommend using the Dilithium3 parameter set, which—according to a very conservative analysis—achieves more than 128 bits of security against all known classical and quantum attacks."⁴. La version dilithium3 était ainsi celle que nous devions sélectionner.

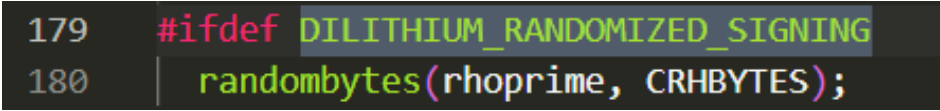
La version Dilithium3-AES utilise AES-256 en mode compteur (CTR) en tant que XOF afin de remplacer l'utilisation de SHAKE-128 en tant que XOF dans la version Dilithium de base (autrement appelée Dilithium-SHA). Pour rappel, une fonction XOF est une fonction à sortie extensible (Extendable-output function). La version AES-256 permet d'étendre la matrice A , générée au début de la partie de l'algorithme générant les clés. Ainsi, les clés générées seront aussi davantage étendue puisque les vecteurs générés aléatoirement s_1 et s_2 correspondent à la clé privée et la clé publique sont calculés à l'aide de la matrice A telle que : $pk = A \times s_1 + s_2$. En ce sens, choisir la version de Dilithium implémentant AES-256 nous paraissait être la bonne solution à adopter.

Par défaut, le mode de signature de Dilithium est déterministe. Pour le remplacer par le mode de signature aléatoire, il faut utiliser la version Dilithium3-AES-R, où le `#define DILITHIUM_RANDOMIZED_SIGNING` est ajouté dans le fichier `config.h`. Si cette directive est ajoutée, alors dans le fichier `sign.c`, lors du calcul de la signature (par la fonction `crypto_sign_signature()`), la condition `#ifdef DILITHIUM_RANDOMIZED_SIGNING` est vérifiée et donc la fonction `randombytes()` est exécutée. La signature générée devient alors aléatoire. Les figure 2.4 et 2.5 présentent la définition de l'aléatoire dans le code source de `avx2_Dilithium3-AES-R`.

```
C config.h
1  #ifndef CONFIG_H
2  #define CONFIG_H
3
4  #define DILITHIUM_MODE 3
5  #define DILITHIUM_USE_AES
6  #define DILITHIUM_RANDOMIZED_SIGNING
7
8  #define CRYPTO_ALGNAME "Dilithium3-AES-R"
9  #define DILITHIUM_NAMESPACE(s) pqcrystals_dilithium3aes_r_avx2##s
10
11 #endif
12
```

FIGURE 2.4 – Capture d'écran d'une partie du code source de `avx2_Dilithium3-AES-R`, dans le fichier `config.h`

4. Citation tirée du site web de Dilithium : <https://pq-crystals.org/dilithium/index.shtml>



```
179  #ifdef DILITHIUM_RANDOMIZED_SIGNING
180  |    randombytes(rhoprime, CRHBYTES);
```

FIGURE 2.5 – Capture d’écran d’une partie du code source de `avx2_Dilithium3-AES-R`, dans le fichier `sign.c`

En effet, nous avons finalement utilisé la version `avx2_Dilithium3-AES-R` puisque celle-ci permet de réaliser des opérations parallèles ou vectorielles sur des architectures SIMD telle qu’ici, AVX2. AVX2 permet d’élargir les opérations sur plusieurs éléments de données simultanément de 128 bits à 256 bits. AVX2 permet donc de traiter plus de données à la fois, ce qui peut largement accélérer les opérations qui peuvent être parallélisées ou vectorialisées. Ainsi, dans le cas de Dilithium, ses performances sont améliorées lors de l’exécution de l’algorithme.

Le tableau ci-dessous résume les performances en fonction des différentes versions de Dilithium⁵ :

5. Skylake est une génération de microprocesseurs. Les parties "LWE Hardness (Core-SVP and refined)" et "SIS Hardness (Core-SVP)" font référence à la résistance des problèmes de "Learning with Errors" (LWE) et de "Short Integer Solution" (SIS). "Core-SVP" référence une version particulière ou simplifiée du problème SVP.

NIST Security Level	2	3	5
Output Size			
public key size (bytes)	1312	1952	2592
signature size (bytes)	2420	3293	4595
LWE Hardness (Core-SVP and refined)			
BKZ block-size b (GSA)	423	624	863
Classical Core-SVP	123	182	252
Quantum Core-SVP	112	165	229
BKZ block-size b (simulation)	433	638	883
\log_2 Classical Gates (see App. C.5)	159	217	285
\log_2 Classical Memory (see App. C.5)	98	139	187
SIS Hardness (Core-SVP)			
BKZ block-size b	423 (417)	638 (602)	909 (868)
Classical Core-SVP	123 (121)	186 (176)	265 (253)
Quantum Core-SVP	112 (110)	169 (159)	241 (230)
Performance (Unoptimized Reference Code, Skylake)			
Gen median cycles	300,751	544,232	819,475
Sign median cycles	1,081,174	1,713,783	2,383,399
Sign average cycles	1,355,434	2,348,703	2,856,803
Verify median cycles	327,362	522,267	871,609
Performance (AVX2, Skylake)			
Gen median cycles	124,031	256,403	298,050
Sign median cycles	259,172	428,587	538,986
Sign average cycles	333,013	529,106	642,192
Verify median cycles	118,412	179,424	279,936
Performance (AVX2+AES, Skylake)			
Gen median cycles	70,548	153,856	153,936
Sign median cycles	194,892	296,201	344,578
Sign average cycles	251,144	366,470	418,157
Verify median cycles	72,633	102,396	151,066

FIGURE 2.6 – Capture d'écran du tableau "Output sizes, security, and performance of Dilithium" tirée du document CRYSTALS-Dilithium Algorithm Specifications and Supporting Documentation

2.2 Construire le wallet

Pour notre wallet, nous voulions que celui-ci soit développé de façon à ce qu'il :

- soit structuré
- y est une possibilité de le chiffrer
- soit rapide d'exécution
- soit léger
- génère des clés uniques

Pour ce faire, nous avons privilégié le développement avec le langage C. Ce langage était, dans notre cas, le choix idéal pour une raison de rapidité d'exécution, de sécurité puisqu'étant bas niveau nous pouvons contrôler les accès mémoires mais aussi pour pouvoir utiliser Dilithium sans soucis de compatibilité. Dilithium étant développé en C et le wallet ne pouvant fonctionner sans cet algorithme, ce fut une raison supplémentaire d'utiliser ce langage.

Le développement du wallet a connu 4 phases majeures de développement. Ces 4 versions vont être détaillées ici afin de comprendre les obstacles rencontrés et la façon dont nous avons dû les gérer pour parvenir à nos fins. Cet objectif final devait être atteint pour pouvoir développer les autres segments de la blockchain, mis à part le consensus qui n'a pas un rôle directement lié au wallet.

2.2.1 Générer les clés et l'adresse

La première étape de la construction du wallet fût la génération des clés. Pour ce faire, nous devons donc comprendre avant toute chose le code source de Dilithium. Comme expliqué plus haut, nous voulions contrôler l'entièreté de notre projet afin d'optimiser les performances et la sécurité de notre blockchain. C'est pourquoi, bien qu'une librairie C⁶ puisse nous permettre d'implémenter simplement Dilithium, nous nous sommes refusés à cette pratique. Dans notre cas, la compréhension de nos actions pourrait probablement avoir un important impact sur la qualité du résultat final. Il a donc fallu comprendre le fonctionnement de son code source pour savoir ce qu'il était ou non possible de réaliser. La figure ci-dessous présente la méthode permettant de générer les clés dans le code source de notre wallet, à savoir le fichier "gen_key.c". Les figures 2.7 et 2.8 présentent la partie code de notre wallet sur la génération des clés à l'aide de Dilithium.

6. <https://libpqcrypto.org/index.html>

```

int gen_keys(uint8_t pk[], uint8_t sk[], uint8_t seed[]) {
    // Gen of the keys (pk & sk (or mk))
    crypto_sign_keypair(pk, sk, seed);

    //printf("CRYPTO_PUBLICKEYBYTES = %d\n", CRYPTO_PUBLICKEYBYTES);
    //printf("CRYPTO_MASTERSECRETKEYBYTES = %d\n", CRYPTO_MASTERSECRETKEYBYTES);
    //printf("SEEDBYTES = %d\n", 3 * SEEDBYTES);

    //printf("\nClé publique : %s\n", showhex(pk, CRYPTO_PUBLICKEYBYTES));
    //printf("\nClé privée : %s\n", showhex(mk, CRYPTO_MASTERSECRETKEYBYTES));
    //printf("\nSeed : %s\n", showhex(seed, 3 * SEEDBYTES));

    return 0;
}

```

FIGURE 2.7 – Capture d’écran d’une partie du code source de notre wallet, dans le fichier `gen_key.c`. Cette image montre la méthode de génération des clés.

```

uint8_t sk[CRYPTO_MASTERSECRETKEYBYTES];
uint8_t pk[CRYPTO_PUBLICKEYBYTES];
uint8_t seed[3 * SEEDBYTES];

```

FIGURE 2.8 – Capture d’écran d’une partie du code source de notre wallet, dans le fichier `gen_key.c`. Cette image montre les paramètres utilisés dans la méthode `int gen_keys(uint8_t pk[], uint8_t sk[], uint8_t seed[])`.

La génération des clés est permise grâce à la méthode `"int crypto_sign_keypair(uint8_t *pk, uint8_t *sk, uint8_t *seed)"` du fichier `sign.c` du code source de Dilithium.

Vous remarquerez sur les captures d’écrans précédentes que par moments (notamment dans les commentaires de code), la clé secrète a été renommée "Master key" ou "Master secret key". La raison est simple, dans l’idée où nous voudrions mettre à jour notre blockchain pour ajouter de nouvelles clés publiques afin d’obtenir de nouvelles adresses pour un même wallet, une master key permettrait de dériver ces nouvelles clés. Aujourd’hui, notre clé privée seule ne peut fournir qu’une adresse, ce qui est amplement suffisant pour notre cas d’usage.

Afin de générer nos adresses avec un chemin de dérivation précis et sécuritaire, nous voulions ainsi reprendre en partie ce qui se fait déjà pour diverses blockchain, notamment Bitcoin. A savoir, construire les adresses à la suite de multiples passages de la clé publique dans des fonctions de hachage et l’usage de certains bits fixes en préfixe de sortie de l’adresse pour définir le type d’adresse et sur quel réseau elle repose⁷. Afin que ce chemin défini dans la méthode `"char* gen_address(uint8_t pk[])"` soit clair, la figure 2.9 présente ce chemin :

7. Les fonctions de hachage utilisées sont SHA256, dSHA256 (ou double SHA256) ainsi que RIPEMD160. Ici, l’utilisation de SHA512, comme mentionné dans notre premier rapport, n’aurait que trop peu d’intérêt puisque la clé publique ne permet pas de délivrer les fonds associés.

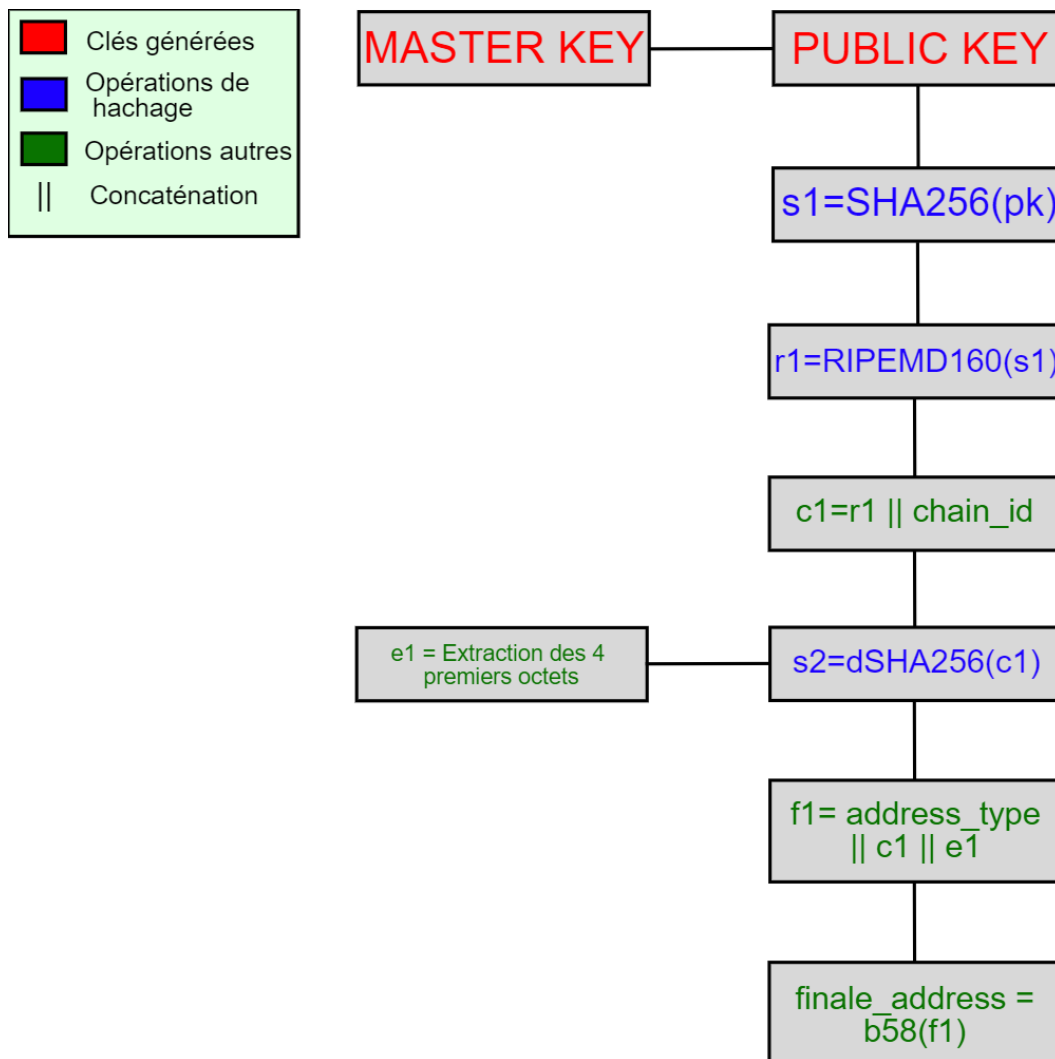


FIGURE 2.9 – Schéma représentant le chemin de dérivation jusqu'à obtenir l'adresse finale en base 58.

Pour nos adresses générées, nous voulions tout d'abord réutiliser l'encodage en base 58. Celui-ci nous paraissait être la solution simple d'usage et pratique pour l'utilisateur, soit, la solution qui convient le plus à nos besoins. Pour ce faire nous avons utilisé le code source de Dmitrii Pichulin sous licence MIT. Vous pourrez le retrouver dans le dossier `./src/base58` de notre archive. L'encodage en base 58 de notre adresse finale ($f1$ si l'on se réfère à la figure 2.9) est réalisé à l'aide de la méthode développée pour, `"char* encodageb58(unsigned char *chainid_ripemd160_fb, size_t chainid_ripemd160_fb_len, const uint16_t addr_type)"` et présente dans le fichier `gen_key.c`.

Un exemple d'adresse générée par notre algorithme est le suivant : `"Cq1GUq8D2vFEh4hxBdVSyQHiqeUtnKKUGf3XVsr6nWGs"`. On reconnaît ici que

l'on est sur une adresse classique de version 1 sur la blockchain qrypto (Cq1, à comprendre comme Classical Address Version 1). De plus, cette adresse a été générée pour fonctionner sur le MAINNET⁸ comme le laisse comprendre la suite du préfixe "GUq8" qui n'est autre que l'encodage base 58 de "MAIN" après concaténation à la suite de l'adresse générée. Si aucun type d'adresse n'est défini dans le fichier "config.h", alors le préfixe sera donné par défaut et sera "11". Ci-contre une adresse de type défaut valable sur le réseau de test, le tesnet : "11HtN9KPpYQfUPSb39jEanPSdRGUvn98VbdRnzZM5A8".

2.2.2 Stocker les données

Le wallet correspond donc au stockage des informations de l'utilisateur dans un fichier stocké sur sa machine. Notre wallet contient donc les informations suivantes :

- Clé publique au format brut
- Clé privée au format brut
- Clé publique au format hexadécimal
- Clé privée au format hexadécimal
- L'adresse dérivée de la clé publique

L'extension choisie pour notre wallet est le `.dat`, soit le format *bitkeys*, actuellement utilisé dans le logiciel Bitcoin Core. Ce choix s'est fait afin de référencer Bitcoin dans notre projet par simple "hommage" et n'a pas d'impact sur la façon dont sont stockées et gérées nos données.

Nous avons choisi de stocker les adresses sous format brut et hexadécimal afin de rendre à l'utilisateur un maximum d'informations, notamment pour qu'il puisse lui-même vérifier l'intégrité de ses clés et de son adresse sous format hexadécimal et base 58.

Pour stocker ces données, nous avons suivi plusieurs étapes avant d'arriver à une version finale fonctionnelle et optimisée.

Un wallet sans moteur de base de données relationnelle

Dans un premier temps, nous avons décidé de réaliser une première version fonctionnelle et sans optimisation. Pour ce faire, nous avons simplement choisi d'écrire dans un fichier nouvellement créé, les données voulues. La figure 2.10 présente la méthode `void walletdat(uint8_t pk[], uint8_t sk[])` permettant de construire le wallet de notre blockchain que nous avons appelé `qptwallet.dat`. Le "qpt" correspond à l'acronyme du nom de notre blockchain, "qrypto".

8. Une blockchain a toujours un réseau principale, là où les transactions sont réelles, c'est le mainnet. Une blockchain a toujours aussi un réseau de test, appelé testnet, correspondant dans notre cas au préfixe "HtN9K".

```

void walletdat(uint8_t pk[], uint8_t sk[]) {
    FILE *fPtr = fopen("./qptwallet.dat", "wb");
    if (fPtr == NULL) {
        printf(stderr, "Error: Cannot open the wallet.dat file for writing (wb).\n");
        exit(1);
    }
    // State of the file
    fprintf(fPtr, "unlock\n");
    // Writing bruts values
    fprintf(fPtr, "brut values : \n");
    fprintf(fPtr, "brut public key : \n");
    fwrite(pk, sizeof(uint8_t), CRYPTO_PUBLICKEYBYTES, fPtr);
    fprintf(fPtr, "\n\nbrut secret key : \n");
    fwrite(sk, sizeof(uint8_t), CRYPTO_SECRETKEYBYTES, fPtr);
    //////////////////////////////////////
    // Writing hex values
    fprintf(fPtr, "\n\nhex values : \n");
    fprintf(fPtr, "public key : %s\n", showhex(pk, CRYPTO_PUBLICKEYBYTES));
    fprintf(fPtr, "secret key : %s\n", showhex(sk, CRYPTO_SECRETKEYBYTES));
    fprintf(fPtr, "address : %s\n", addr_cat_crf);
    fclose(fPtr);
    // Free memory
    free(addr_cat_crf);
}

```

FIGURE 2.10 – Capture d’écran d’une partie du code source de l’une des premières versions de notre wallet, dans le fichier `gen_key.c`. Cette image montre la méthode de création du wallet `qptwallet.dat`.

Vous pouvez visualiser le résultat d’un wallet généré avec cette méthode avec le fichier `qptwallet_v1.dat` présent dans l’archive rendue.

Bien que le fichier généré puisse être considéré par son propriétaire comme étant un wallet fonctionnel, ce qui est le cas, il manquait de sécurité. Rappelons le, le wallet stocke toutes les informations de l’utilisateur, notamment sa clé privée. Si celle-ci venait à être volée, cryptographie post-quantique utilisée ou non, elle donnerait ainsi accès à la clé publique et par extension à l’adresse. De cette façon, l’attaquant pourrait aisément voler les fonds associés à la clé privée. Dans un soucis de vouloir éviter ce type d’attaque au maximum, nous voulions donc ajouter un élément essentiel, le chiffrement.

Comment chiffrer ?

Dans la première version de notre wallet implémentant la possibilité de chiffrer son wallet, nous avons choisi de gérer cette partie avec le langage C++. De cette façon le développement serait plus simple, plus flexible et pour autant nous aurions un chiffrement qui resterait assez rapide pour notre cas d’usage.

Nous sommes alors partis sur l’utilisation d’AES 256 en mode CBC. Ce choix s’explique puisqu’AES est un algorithme de chiffrement symétrique largement implémenté à travers les infrastructures de sécurité, témoignant de sa robustesse. Robustesse que

nous voulions davantage renforcée en utilisant le mode de clé de 256 bits. Le mode CBC permet de créer une interdépendance entre les blocs de données grâce à l'opération XOR réalisée entre le bloc chiffré $n - 1$ et le bloc de données n à l'état brut. Ainsi, chaque bloc chiffré dépend du bloc de données brut précédent. De cette façon, si des données sont altérées, cela affectera l'ensemble des blocs de données suivants. La figure 2.11

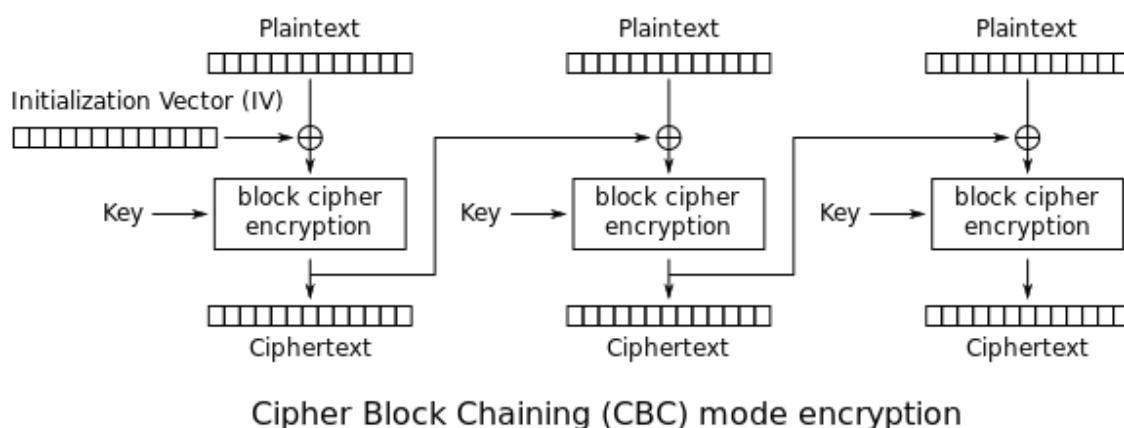


FIGURE 2.11 – Schéma représentatif du mode de fonctionnement CBC dans AES. (Image tirée du site web educative.io)

Pour réaliser ce chiffrement nous avons donc utiliser les outils fournis par openssl, à savoir l'entête `<openssl/evp.h>`.

Ce fichier d'entête nommé "EVP" correspond au terme "EnVeloPe". Cette interface permet d'encapsuler différents algorithmes cryptographiques (AES, DES, RSA, etc.) sous une API commune. Dans notre programme nous utilisons les fonctions utiles à l'implémentation d'AES, dans notre cas, dans un programme C++. L'usage de cet outil permet d'utiliser ces algorithmes sans pour autant prêter attention aux détails d'implémentations de ces derniers. Sa facilité d'usage en fait donc une force dans le développement d'outils.

Pour chiffrer le wallet nouvellement créé nous avons donc créé un fichier à part et lié à celui de la génération des clés, le fichier "walletdat_aes_cpp.cpp". Celui ci se compose de 2 méthode principale, la méthode de dérivation de la clé générée à partir du password choisit par l'utilisateur (voir figure 2.12 présentant cette méthode) et la seconde permettant de gérer toutes les opérations de chiffrement et de déchiffrement avec AES-256-CBC (voir figure 2.13 présentant le début de cette méthode).

```
void deriveKeyFromPassword(const string &password, unsigned char *key, unsigned char *iv){
    if (EVP_BytesToKey(EVP_aes_256_cbc(), EVP_sha256(), nullptr,
        reinterpret_cast<const unsigned char *>(password.c_str()), password.length(), 1, key, iv) != KEY_SIZE) {
        err();
    }
}
```

FIGURE 2.12 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "walletat_aes_cpp.cpp". Cette image montre la méthode de dérivation de la clé générée à partir du password choisit par l’utilisateur.

```
void aes_file(const string &inputFilename, const string &outputFilename, const string &password) {
    ifstream inputFile(inputFilename, ios::binary);
    ofstream outputFile(outputFilename, ios::binary);
    if (!inputFile || !outputFile) {
        err();
    }
    string firstLine;
    getline(inputFile, firstLine);
    cout << "first line " << firstLine << endl;
    bool crypttype;
    if (firstLine.find("unlock") != string::npos) {
        crypttype = true; // Must encrypt the file
    } else if (firstLine.find("lock") != string::npos) {
        crypttype = false; // Must decrypt the file
    } else {
        cerr << "Invalid mode found in the file." << endl;
        exit(1);
    }
}
```

FIGURE 2.13 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "walletat_aes_cpp.cpp". Cette image présente une partie de la méthode permettant de savoir si le wallet est déjà chiffré ou non (par la lecture de la première ligne du fichier).

La figure 2.13 présente donc le début de la méthode de chiffrement/déchiffrement. Cette capture d’écran expose la lecture de la première ligne du fichier entré en paramètre (dans notre cas, le fichier wallet). Si cette ligne contient le terme "lock", alors le fichier est chiffré et le booleen "crypttype" est mis à la valeur "false", indiquant alors au reste du programme de déchiffrer le wallet avec le password alors entré par l’utilisateur. La figure 2.14 présente la façon dont nous chiffons et déchiffrons le wallet avec cette première version de chiffrement pour notre wallet.

```

while ((bytesRead = inputFile.readsome(reinterpret_cast<char *>(buffer), sizeof(buffer))) > 0) {
    if (crypttype) {
        if (EVP_EncryptUpdate(ctx, cipherText, &cipherTextLength, buffer, bytesRead) != 1) {
            err();
        }
        // Writing the ciphertext in the outputFile
        outputFile.write(reinterpret_cast<const char *>(cipherText), cipherTextLength);
    } else if (!crypttype) {
        if (EVP_DecryptUpdate(ctx, clearText, &clearTextLength, buffer, bytesRead) != 1) {
            err();
        }
        // Writing the cleartext in the outputFile
        outputFile.write(reinterpret_cast<const char *>(clearText), clearTextLength);
    }
}
if (crypttype) {
    // Ending the encryption
    if (EVP_EncryptFinal_ex(ctx, cipherText, &cipherTextLength) != 1) {
        err();
    }
    // Writing the last part of the ciphertext in the outputFile
    outputFile.write(reinterpret_cast<const char *>(cipherText), cipherTextLength);
    remove(inputFilename.c_str());
} else if (!crypttype) {
    // Ending the decryption
    if (EVP_DecryptFinal_ex(ctx, clearText, &clearTextLength) != 1) {
        err();
    }
    // Writing the last part of the deciphertext in the outputFile
    outputFile.write(reinterpret_cast<const char *>(clearText), clearTextLength);
    remove(inputFilename.c_str());
}
}

```

FIGURE 2.14 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "walletdat_aes_cpp.cpp". Cette image présente une partie de la méthode permettant de chiffrer ou déchiffrer le wallet en fonction du contenu du booleen "crypttype".

Enfin, nous avons construit une méthode permettant de rendre le wallet en mode lecture seule pour éviter les modifications manuelles sur le fichier, diminuant ainsi la possibilité de sabotage d’un wallet.

Dans une version supérieure de ce wallet, nous avons décidé de hacher le password de l’utilisateur par SHA-256. Ce choix afin de vérifier que le password fourni était correct dans le but d’éviter que l’utilisateur se trompe de mot de passe et rechiffre alors son wallet (déjà chiffré) avec une nouvelle clé dérivée du mauvais mot de passe. Pour pouvoir travailler avec le fichier "sha256.c" contruit pour le projet et ainsi bénéficier de sa rapidité d’exécution, nous avons finalement choisi de retranscrire notre programme C++ en un fichier C. Le programme C++ "walletdat_aes_cpp.cpp" était alors réécrit dans le fichier "wal.c" lui même appelé par le fichier "encryptwallet.c" permettant ici de gérer la partie interface utilisateur lui proposant de chiffrer/déchiffrer son wallet avec un mot de passe haché et écrit sur la première ligne du fichier du wallet. La figure 2.15 illustre une partie de ce fonctionnement.

```

if(!HasAlreadyBeenCipher) {
    printf("Do you want to encrypt your wallet file with AES (recommended) ? [Y/n] ");
    scanf(" %s", &userResponse);

    if (userResponse == 'Y' || userResponse == 'y' || userResponse == 'Yes' || userResponse == 'yes' ||
        userResponse == 'YES') {
        response = true;
        printf("You have chosen to encrypt the wallet.dat file.\n");

        char userPassword[100];

        do {
            printf("Choose a password (100 characters max) : ");
            scanf(" %s", &userPassword);

        } while (!isPswdGood(userPassword));

        sha256_fun((uint8_t *) userPassword, sha256_hash, 1, strlen(userPassword));

        char *hashpassword = showhex2(sha256_hash, SHA256_DIGEST_LENGTH);

        aes_file("wallet.dat", hashpassword);
        free(hashpassword);
    }
}

```

FIGURE 2.15 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "encryptwallet.c". Cette image présente une partie de la méthode "void encryptfile(bool HasAlreadyBeenCipher)" permettant de laisser le choix à l’utilisateur de chiffrer ou non son wallet.

Ce fonctionnement était valable et nous permettait désormais de concevoir des wallets avec les données nécessaires stockées et une possibilité de chiffrer et déchiffrer celui-ci. Cependant, sa construction tout comme la recherche d’éléments dans le fichier n’était pas optimale. Nous devions ainsi trouver une nouvelle manière de générer nos wallets. Pour ce faire, nous avons étudié ce qui est actuellement proposé sur le marché et après de nombreuses recherches il nous fallait un système de gestion de base de données relationnelle. Celui qui correspondait le plus à nos attentes a été SQLite.

Implémentation d’SQLite

Pour structurer notre wallet, nous sommes donc partis sur l’option la plus cohérente, un SGBD relationnel. Le choix d’SQLite s’explique par les critères suivants que nous avons définis pour notre cas d’usage :

- Autonomie dans le stockage : Un fichier seul permet de stocker l’intégralité des tables et attributs de la base de données, aucun serveur n’est requis. Dans notre cas, si un serveur était utilisé, nous perdriions tous les avantages offerts par la blockchain, il y aurait une centralisation des services, ce qui était une option impensable.
- Une légèreté d’implémentation : SQLite est une bibliothèque C adaptée aux environnements aux ressources limitées.
- Une légèreté des bases de données : Le poids des bases de données créées reste relativement faible et donc acceptable dans nos cas d’usages (wallet et autres

fichiers que nous développerons dans les prochaines parties de ce rapport).

- Une facilité d'implémentation : Il est possible d'utiliser SQLite avec une amalgamation, c'est à dire qu'un seul fichier va répertorier l'intégralité du code source. Dans notre cas, ce sont les fichiers "sqlite3.c" et "sqlite3.h" qui le permettent.⁹
- Rapidité : SQLite est optimisé pour des performances élevées, les requêtes sont généralement plus rapides puisqu'elles se réalisent directement sur le fichier de base de données, ici le wallet.
- Une facilité d'usage : Utilisant le langage SQL, un langage familier, nous connaissons au préalable le fonctionnement des requêtes de ce type de bases de données.
- Open Source : SQLite est un logiciel open source et du domaine public ne requérant pas de licence pour son usage.

Les contraintes développées ci-dessus étant toutes prises en charge par SQLite, ce choix était donc naturellement celui que nous devions choisir.

```
void walletdat(uint8_t pk[], uint8_t sk[]){
    sqlite3 *db;

    // Ouvrir la base de données SQLite
    int rc = sqlite3_open("qptwallet.dat", &db);
    if (rc) {
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }

    // Créer la table si elle n'existe pas déjà
    char *sql = "CREATE TABLE IF NOT EXISTS wallet (public_key TEXT, secret_key TEXT, address TEXT);";
    rc = sqlite3_exec(db, sql, NULL, 0, NULL);
    if (rc != SQLITE_OK) {
        fprintf(stderr, "Can't create table: %s\n", sqlite3_errmsg(db));
        sqlite3_close(db);
        exit(1);
    }
}
```

FIGURE 2.16 – Capture d'écran d'une partie du code source de la seconde version de notre wallet, dans le fichier "gen_keys.c". Cette image présente une partie de la méthode créant le wallet avec SQLite.

La figure 2.16 présente le début de la méthode créant le fichier "qptwallet.dat". On reconnaît une requête SQL de création de la table "wallet" avec les colonnes "public_key", "secret_key" et "address". A noter que cette version a été la première version fonctionnelle de notre wallet SQLite. Cette version a été mise à jour à de multiples reprises afin de fonctionner convenablement dans un environnement adapté. Les majeures modifications effectuées ensuite ont été¹⁰ :

9. Le fichier "sqlite3ext.h" fournit une interface standardisée pour développer des extensions pour SQLite en tant que bibliothèques partagées, permettant ainsi aux développeurs d'étendre les fonctionnalités de SQLite de manière modulaire et cohérente, ce qui n'est pas utile dans notre cas.

10. Nous développerons ces modifications dans la partie suivante de ce chapitre.

- Changement des types de certaines colonnes
- Changement du nom de la table
- Ajouts de colonnes

```
// Préparer la requête d'insertion
sql = "INSERT INTO wallet (public_key, secret_key, address) VALUES (?, ?, ?);";
sqlite3_stmt *stmt;
rc = sqlite3_prepare_v2(db, sql, -1, &stmt, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't prepare statement: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
}

// Binder les valeurs et exécuter la requête
rc = sqlite3_bind_text(stmt, 1, showhex(pk, CRYPTO_PUBLICKEYBYTES), -1, SQLITE_STATIC);
rc = sqlite3_bind_text(stmt, 2, showhex(sk, CRYPTO_SECRETKEYBYTES), -1, SQLITE_STATIC);
rc = sqlite3_bind_text(stmt, 3, addr_cat_crf, -1, SQLITE_STATIC);
rc = sqlite3_step(stmt);
if (rc != SQLITE_DONE) {
    fprintf(stderr, "Can't execute statement: %s\n", sqlite3_errmsg(db));
    sqlite3_close(db);
    exit(1);
}

// Finaliser et fermer la base de données
sqlite3_finalize(stmt);
sqlite3_close(db);
}
```

FIGURE 2.17 – Capture d'écran d'une partie du code source de la seconde version de notre wallet, dans le fichier "gen_keys.c". Cette image présente une partie de la méthode insérant des données dans le wallet avec SQLite.

La figure 2.17 présente la première version fonctionnelle pour insérer des données dans le fichier "qptwallet.dat" à l'aide de SQLite. On reconnaît aussi une requête SQL d'insertion de données "INSERT INTO wallet (public_key, secret_key, address) VALUES (?, ?, ?);".

Pour compléter notre wallet et avoir une version chiffrable, nous devons donc trouver une façon de le faire qui soit davantage optimale que notre première version. Pour ce faire, nous avons étudié les extensions à SQLite permettant de chiffrer et déchiffrer une base de données. C'est ainsi, qu'après de nombreuses recherches sur le sujet, nous avons convenu que l'extension "SQLCipher" conviendrait.

Implémentation d'SQLCipher

SQLCipher est un standalone de la bibliothèque de base de données SQLite qui permet d'ajouter le chiffrement AES-256 en mode CBC sur des fichiers de base de données. Nous avons choisi de développer avec la version open source et gratuite du projet, soit,

"SQLCipher Community Edition". Son installation et utilisation fût est assez périlleuse puisque tous les modules n'étaient pas mis à jour pour fonctionner correctement entre eux. De fait nous n'avons pas pu utiliser directement des méthodes du code source. Un problème auquel on a finalement pu trouver une alternative. Pour installer et pouvoir utiliser convenablement SQLCipher en ligne de commande, la démarche à suivre est la suivante :

```
$ wget https://github.com/sqlcipher/sqlcipher/archive/master.zip
$ unzip master.zip
$ cd sqlcipher-master
$ sudo apt-get update
$ sudo apt-get install libssl-dev
$ ./configure --enable-tempstore=yes CFLAGS=
"-DSQLITE_HAS_CODEC" LDFLAGS="-lcrypto"
$ make
$ sudo make install DESTDIR=
```

Une fois l'installation réalisée, nous avons donc pu utiliser SQLCipher. Pour ce faire, nous avons avant tout repris la création de notre wallet avec SQLite comme la figure 2.18 le présente, en rajoutant les données brutes de nos clés à notre wallet ainsi qu'en modifiant le nom de la table. Nous ne présentons ici que la partie d'insertion des données dans notre wallet de la méthode "void sql_walletdat(uint8_t pk[], uint8_t sk[], char *userpswd, bool mustencrypt, char *finale_address)" du fichier "wallet_enc_dec.c".

```

// Creation of the table qptwallet
char *sql = "CREATE TABLE IF NOT EXISTS qptwallet (idwallet INTEGER PRIMARY KEY AUTOINCREMENT,
raw_public_key BLOB, raw_secret_key BLOB,
        hex_public_key TEXT, hex_secret_key TEXT, address TEXT);";
rc = sqlite3_exec(dec_db, sql, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't create table: %s\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

// Insert data into qptwallet table
sql = "INSERT INTO qptwallet (raw_public_key, raw_secret_key, hex_public_key, hex_secret_key, address)
VALUES (?, ?, ?, ?, ?);";
sqlite3_stmt *stmt;
rc = sqlite3_prepare_v2(dec_db, sql, -1, &stmt, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't prepare statement: %s\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

rc = sqlite3_bind_blob(stmt, 1, pk, CRYPTO_PUBLICKEYBYTES, -1);
rc = sqlite3_bind_blob(stmt, 2, sk, CRYPTO_SECRETKEYBYTES, -1);
rc = sqlite3_bind_text(stmt, 3, showhex(pk, CRYPTO_PUBLICKEYBYTES), -1, SQLITE_STATIC);
rc = sqlite3_bind_text(stmt, 4, showhex(sk, CRYPTO_SECRETKEYBYTES), -1, SQLITE_STATIC);
rc = sqlite3_bind_text(stmt, 5, finale_address, -1, SQLITE_STATIC);
rc = sqlite3_step(stmt);
if (rc != SQLITE_DONE) {
    fprintf(stderr, "Can't execute statement: %s\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

```

FIGURE 2.18 – Capture d'écran d'une partie du code source de la seconde version de notre wallet, dans le fichier "wallet_enc_dec.c". Cette image présente une partie de la méthode insérant des données dans le wallet avec SQLite.

La partie de la méthode présentée en figure 2.18, permet l'insertion des données dans le wallet, alors nommé "dec_qptwallet.dat". Le préfixe "dec" référence le fait que le wallet créé est avant tout déchiffré, aucun password n'y est assigné.

En fonction du choix de l'utilisateur, le wallet peut être chiffré et alors la méthode "void enc_walletdat(char *userpswd)" est appelée. Dans cette méthode, nous copions la base de données déchiffrée "dec_qptwallet.dat" et la collons dans une version chiffrée "enc_qptwallet.dat".


```
void enc_walletdat(char *userpswd) {  
  
    sqlite3 *dec_db;  
    int rc;  
    char attach_db[200];  
  
    // Open the non cipher database  
    rc = sqlite3_open("dec_qptwallet.dat", &dec_db);  
  
    if (rc != SQLITE_OK) {  
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(dec_db));  
        sqlite3_close(dec_db);  
        exit(1);  
    }  
  
    // Attach the cipher bdd  
    snprintf(attach_db, sizeof(attach_db), "ATTACH DATABASE 'enc_qptwallet.dat' AS encrypted KEY '%s';", userpswd);  
    rc = sqlite3_exec(dec_db, attach_db, NULL, 0, NULL);  
    if (rc != SQLITE_OK) {  
        fprintf(stderr, "Can't attach database because your password is probably wrong (or %s)\n", sqlite3_errmsg(dec_db));  
        sqlite3_close(dec_db);  
        exit(1);  
    }  
}
```

FIGURE 2.19 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "wallet_enc_dec.c". Cette image présente une partie de la méthode créant la base de données chiffrée avec SQLCipher.

La figure 2.19 présente la création de la base de données/du wallet chiffré avec SQLCipher en attachant à cette base de données un password, celui fourni par l’utilisateur. Une version améliorée pourrait fournir non pas directement le password de l’utilisateur mais la version SHA-256 de celui-ci. Par manque de temps, nous n’avons pas développé cette fonctionnalité.

```

// Creation of the table qptwallet
char *sql = "CREATE TABLE IF NOT EXISTS encrypted.qptwallet (idwallet INTEGER PRIMARY KEY AUTOINCREMENT,
raw_public_key BLOB, raw_secret_key BLOB,
        \" hex_public_key TEXT, hex_secret_key TEXT, address TEXT);";
rc = sqlite3_exec(dec_db, sql, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't create table: %s\\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

// Insert data from non cipher db into cipher one (from qptwallet table)
sql = "INSERT INTO encrypted.qptwallet SELECT * FROM qptwallet;";
rc = sqlite3_exec(dec_db, sql, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't insert data: %s\\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

// Detach the cipher db
rc = sqlite3_exec(dec_db, "DETACH DATABASE encrypted;", NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't detach database: %s\\n", sqlite3_errmsg(dec_db));
    sqlite3_close(dec_db);
    exit(1);
}

sqlite3_close(dec_db);

// remove dec_qptwallet.dat file
shell_command("rm -f dec_qptwallet.dat");

```

FIGURE 2.20 – Capture d'écran d'une partie du code source de la seconde version de notre wallet, dans le fichier "wallet_enc_dec.c". Cette image présente une partie de la méthode copiant la base de données non chiffrée vers sa version chiffrée avec SQLCipher.

La figure 2.20 présente l'insertion des données dans le fichier "enc_qptwallet.dat" et la suppression du fichier en clair (pour ne conserver dans les fichiers que la version chiffrée) par l'appel d'une commande système. La méthode "void shell_command(char* commande)" étant définie dans le fichier d'entête "config.h".

Enfin, la méthode "void dec_walletdat(char *userpswd)" permet de déchiffrer un wallet reconnu comme chiffré. Cette reconnaissance se fait par 2 moyens, notamment par la méthode "bool IsWalletEncrypted(bool enc)" définie dans le fichier "gen_key.c". La figure 2.21 expose le programme de définition de la clé de déchiffrement du wallet.

```
void dec_walletedat(char *userpswd) {  
  
    sqlite3 *enc_db;  
    char *err_msg = 0;  
    int rc;  
    char attach_db[200];  
    char key[100];  
  
    // Open the cipher database  
    rc = sqlite3_open("enc_qptwallet.dat", &enc_db);  
  
    if (rc != SQLITE_OK) {  
        fprintf(stderr, "Can't open database: %s\n", sqlite3_errmsg(enc_db));  
        sqlite3_close(enc_db);  
        exit(1);  
    }  
  
    // Define the cipher key  
    snprintf(key, sizeof(key), "PRAGMA key = '%s';", userpswd);  
    rc = sqlite3_exec(enc_db, key, NULL, 0, NULL);  
    if (rc != SQLITE_OK) {  
        fprintf(stderr, "Can't set PRAGMA key: %s\n", sqlite3_errmsg(enc_db));  
        sqlite3_close(enc_db);  
        exit(1);  
    }  
}
```

FIGURE 2.21 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "wallet_enc_dec.c". Cette image présente une partie de la méthode permettant de définir la clé de déchiffrement, correspondant au password de l’utilisateur.

Enfin, la figure 2.22 permet de finaliser le déchiffrement du wallet à partir du password utilisateur.

```

// Attach the non cipher bdd
snprintf(attach_db, sizeof(attach_db), "ATTACH DATABASE 'dec_qptwallet.dat' AS decrypted KEY ''");
rc = sqlite3_exec(enc_db, attach_db, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't attach database because your password is probably wrong (or %s)\n", sqlite3_errmsg(enc_db));
    sqlite3_close(enc_db);
    exit(1);
}

// Creation of the table qptwallet
char *sql = "CREATE TABLE IF NOT EXISTS decrypted.qptwallet (idwallet INTEGER PRIMARY KEY AUTOINCREMENT,
raw_public_key BLOB, raw_secret_key BLOB, hex_public_key TEXT, hex_secret_key TEXT, address TEXT);";
rc = sqlite3_exec(enc_db, sql, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't create table: %s\n", sqlite3_errmsg(enc_db));
    sqlite3_close(enc_db);
    exit(1);
}

// Insert data from non cipher db into cipher one (from qptwallet table)
sql = "INSERT INTO decrypted.qptwallet SELECT * FROM qptwallet;";
rc = sqlite3_exec(enc_db, sql, NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't insert data: %s\n", sqlite3_errmsg(enc_db));
    sqlite3_close(enc_db);
    exit(1);
}

// Detach the cipher db
rc = sqlite3_exec(enc_db, "DETACH DATABASE decrypted;", NULL, 0, NULL);
if (rc != SQLITE_OK) {
    fprintf(stderr, "Can't detach database: %s\n", sqlite3_errmsg(enc_db));
    sqlite3_close(enc_db);
    exit(1);
}

sqlite3_close(enc_db);

// remove enc_qptwallet.dat file
shell_command("rm -f enc_qptwallet.dat");
}

```

FIGURE 2.22 – Capture d’écran d’une partie du code source de la seconde version de notre wallet, dans le fichier "wallet_enc_dec.c". Cette image présente une partie de la méthode permettant de créer le wallet déchiffrer "dec_qptwallet.dat".

Résultats

Ce sous-chapitre permet de présenter les principales fonctionnalités de notre programme. Nous ne rentrerons pas dans les détails des gestions d’erreurs.

Pour créer l’exécutable du programme, veuillez suivre les étapes décrites dans le fichier "readme.md" fourni dans l’archive du projet.

La figure 2.23 présente le programme "gen_key_mode3" tel que s’il était exécuté pour la première fois sur votre machine. Une adresse en base 58 est générée et votre wallet est créé. Vous avez ensuite le choix de le chiffrer ou non. Pour que le password soit valide, il doit contenir au moins 12 caractères, dont au moins une lettre minuscule, une lettre majuscule, un chiffre et un caractère spécial.

```
src/wallet$ ./gen_key_mode3
ADDRESS : b58_addr||chainid_ripemd160_fb (base58): Cq1GUq8CzNCmpoMmLmbptLgri5STWkNo429YQ4yaxwq5
Do you want to encrypt your wallet file with AES (recommended) ? [Y/n] Y
You have chosen to encrypt your wallet, becoming enc_qptwallet.dat file.
Choose a password (100 characters max) : Password_123!
Command has been run successfully.
```

FIGURE 2.23 – Capture d'écran de l'exécution du programme gérant le wallet.

La figure 2.24 prouve que le chiffrement a bien fonctionné. Effectivement, on remarque qu'avec SQLite, l'erreur "Parse error : file is not a database (26)" est levée, indiquant à l'utilisateur que le programme ne reconnaît pas le format SQLite dans le fichier. De fait, il est conclu que ce fichier n'est pas une base de données (database), or, elle est simplement chiffrée. La commande "PRAGMA key = password;" permet de le déverrouiller en ligne de commande pour avoir un accès direct aux données stockées. Dans notre cas, nous choisissons de retourner l'adresse de l'utilisateur.

```
src/wallet$ sqlcipher enc_qptwallet.dat
SQLite version 3.44.2 2023-11-24 11:41:44 (SQLCipher 4.5.6 community)
Enter ".help" for usage hints.
sqlite> SELECT address FROM qptwallet;
Parse error: file is not a database (26)
sqlite> PRAGMA key = "Password_123!";
ok
sqlite> SELECT address FROM qptwallet;
Cq1GUq8CzNCmpoMmLmbptLgri5STWkNo429YQ4yaxwq5
```

FIGURE 2.24 – Capture d'écran de l'ouverture du wallet avec le standalone SQLCipher.

La figure 2.25 présente une capture d'écran de l'exécution de notre programme où nous choisissons de déchiffrer notre fichier "enc_qptwallet.dat".

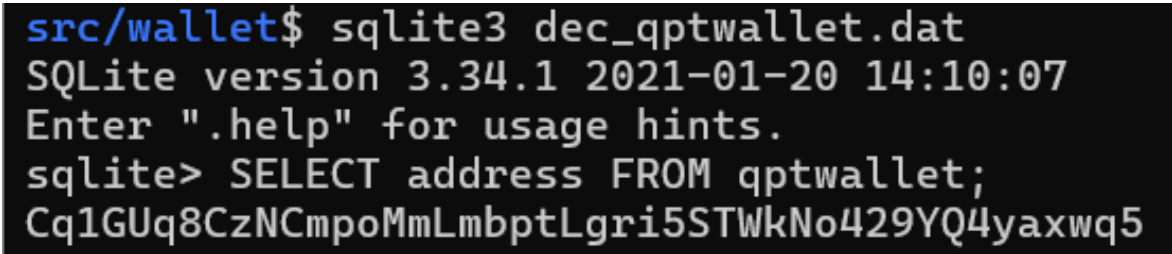
```
src/wallet$ ./gen_key_mode3
You already have a wallet (cipher one).
Database is encrypted

Do you want to decipher it ? [Y/n] y

Please, enter your password (max 100 charac) : Password_123!
Command has been run successfully.
```

FIGURE 2.25 – Capture d'écran du déchiffrement du wallet avec le standalone SQLCipher.

On comprend via la commande présentée dans la figure 2.26 que le fichier est bien déchiffré. Notons que nous avons exécuté la commande avec SQLite mais la même commande aurait pu être faite via SQLCipher, le résultat aurait été similaire, SQLCipher étant une extension à SQLite.

A terminal window with a black background and white text. The prompt is 'src/wallet\$'. The command 'sqlite3 dec_qptwallet.dat' has been executed. The output shows 'SQLite version 3.34.1 2021-01-20 14:10:07' and 'Enter ".help" for usage hints.' The user has entered 'sqlite> SELECT address FROM qptwallet;' and the output is 'Cq1GUq8CzNCmpoMmLmbptLgri5STWkNo429YQ4yaxwq5'.

```
src/wallet$ sqlite3 dec_qptwallet.dat
SQLite version 3.34.1 2021-01-20 14:10:07
Enter ".help" for usage hints.
sqlite> SELECT address FROM qptwallet;
Cq1GUq8CzNCmpoMmLmbptLgri5STWkNo429YQ4yaxwq5
```

FIGURE 2.26 – Capture d’écran de l’ouverture du wallet avec SQLite.

Chapitre 3

Construire le consensus

Le consensus définit les règles qui vont permettre à la blockchain de grandir. Chaque bloc est ajouté suivant des principes définis dès le départ qui empêchent aussi bien que possible les mineurs/validateurs de tricher. Ce contrôle se fait évidemment sans une entité supérieure qui ferait office d'autorité. De fait, tout est régi en fonction d'un travail qui doit être effectué par le mineur/validateur puis vérifié par les autres nœuds du réseau. Ce travail, pour la PoW (Proof of Work) constitue le fait de trouver un hash valide à une target donnée pour le bloc créé. Lorsque le bloc est partagé au réseau, il est facile pour quiconque de vérifier que le hash du bloc correspond bien à la target définie. Ayant expliqué en profondeur l'algorithme de la PoW dans notre précédent rapport, nous ne reviendrons pas sur son fonctionnement ici. Cependant, bien que ce consensus fonctionne, nous voulions une solution davantage performante et qui réponde à certains critères que nous nous sommes fixés.

Question économique et énergétique Un problème survient dès lors qu'une PoW est implémentée, le problème de la consommation importante des ressources. Aujourd'hui, les mineurs les plus importants possèdent des fermes de minages qui consomment d'importantes quantités d'électricité et d'eau. L'objectif étant d'être le premier à trouver le hash valide, c'est une réelle course. Pour pouvoir y participer et être au moins rentable, d'importantes infrastructures sont à mettre en place. Le coût de celles-ci associées aux ressources énergétiques dépensées peut rapidement grimper. De fait, le système, bien que décentralisé sur le papier, est en réalité plus centralisé qu'il n'y paraît. De plus, en raison de la situation écologique qui préoccupe la population, rares se font les nouvelles blockchains en PoW qui performant.

Quantum era S'ajoute à cette problématique une autre menace que devra affronter la cryptographie moderne ; la cryptanalyse quantique. Il semble aujourd'hui raisonnable de se préparer à son arrivée en déployant des blockchains d'un nouveau genre, des blockchains PQR. Bien que la PoW soit résistante à la cryptanalyse quantique, l'algorithme de Grover pourrait toutefois permettre de trouver un hash valide plus vite

que la moyenne¹. Cet algorithme offrirait à qui peut l'exécuter, un avantage sur la plupart des autres mineurs et ainsi centraliser les gérants de la blockchain.

Notre objectif Nous voulions donc aller plus loin en proposant un schéma qui permettrait à notre blockchain d'être :

- Plus économique ;
- Davantage quantum-resistant qu'une PoW ;
- Plus équitable entre les participants, soit offrir davantage de décentralisation.
- Plus rapide ;
- Plus eco-friendly ;

3.1 De la *Proof of Work* aux VDFs

Depuis plusieurs années maintenant, les chercheurs et développeurs collaborent pour fournir sans cesse de nouveaux modèles de consensus. En fonction des équipes de R&D, certaines privilégient la sécurité, d'autres la scalabilité ou encore la décentralisation. L'un des exemples de développement les plus parlants serait celui de la preuve d'enjeu ou PoS (*Proof of Stake*). À l'inverse de la PoW, les mineurs (qui sont ici appelés des validateurs) n'ont plus à fournir une importante puissance de calcul mais simplement à mettre à disposition du réseau une certaine quantité de tokens. Ce principe, c'est le staking. Une fois ses tokens stakés, l'utilisateur est éligible à devenir un leader, soit, un potentiel validateur. En devenant leader, sa mission va être de créer un bloc et de le propager sur le réseau. Si le bloc proposé est accepté, alors le leader a validé un bloc (et devient par conséquent un validateur) et peut emporter sa récompense (ou transaction COINBASE). Dans le cas inverse où le leader a tenté de tricher, celui-ci est "slashé", soit puni par le réseau. Différentes punitions peuvent être prévues, notamment celle de burn² les tokens stakés de l'utilisateur.

L'objectif recherché par rapport à la PoW est notamment la rapidité d'exécution, une diminution de la demande de consommation (donc une meilleure décentralisation des validateurs en théorie) mais aussi plus d'équité entre les validateurs du réseau (ici aussi, une meilleure décentralisation des validateurs en théorie).

Dans notre processus de développement de notre blockchain, nous proposons une innovation, un consensus d'un nouveau genre. Notre objectif serait de coupler à une PoS ce que l'on appelle une VDF (*Verifiable Delay Function*). Depuis leur création en 2018, les VDF sont devenues un nouvel objet cryptographique réputé. Il en existe aujourd'hui plusieurs dont certaines sont PQR. Vous l'aurez probablement compris mais c'est à ce dernier type de VDF que nous allons désormais nous intéresser.

1. Nous avons détaillé cette partie dans notre précédent rapport.

2. Comprendre détruire les tokens de l'utilisateur en les envoyant sur un dead wallet, soit sur une adresse pour laquelle personne n'a la clé privée associée.

Nous expliquerons davantage notre consensus dans la partie dédiée mais retenez que nous utilisons le principe de la PoS couplé à une VDF.

3.1.1 Qu'est ce qu'une VDF ?

Définition 5 (Verifiable Delay Function). *Une fonction de délai vérifiable (VDF en anglais) est une primitive cryptographique conçue de manière à ce que le calcul prenne un certain temps t choisi, même avec une parallélisation des calculs. Néanmoins, la vérification du calcul doit se faire en temps bien inférieur à t .*

Une VDF $V = (\text{Setup}, \text{Eval}, \text{Verify})$ est un regroupement des trois algorithmes :

1. $\text{Setup}(\lambda, t) \rightarrow pp = (e_k, v_k)$ un algorithme "aléatoire", qui produit les paramètres publics pour l'évaluation e_k et la vérification v_k . Cette algorithme doit être polynomial en λ ;
2. $\text{Eval}(pp, x) \rightarrow (y, \pi)$ prend en entrée un $x \in \mathcal{X}$ et renvoie une sortie $y \in \mathcal{Y}$ (le domaine et codomaine sont dans le couple pp), et une preuve π (optionnelle). Ce calcul demandera t étapes ;
3. $\text{Verify}(pp, x, y, \pi) \rightarrow \{\text{oui}, \text{non}\}$ si $\text{Eval}(pp, x) = (y, \pi)$, on renvoie oui, et non dans le cas contraire.

Cette définition correspond aux VDFs dites faibles. Il existe également des VDFs dites incrémentés, VDF décodables, et *trapdoor* VDF. Nous ne rentrerons pas dans les détails de ceux-ci, car celles-ci ne nous intéressent pas ici.

Le principe même d'une VDF est de pouvoir donner une évaluation à effectuer en un temps t . En l'occurrence, c'est pour cette raison que nous nous sommes dirigé vers l'usage de ces fonctions, car elles nous permettent d'avoir un certain contrôle sur le temps entre chaque blocs.

Exemple 4. *Pour comprendre l'idée principale, nous allons prendre comme exemple une version simplifiée d'un protocole existant. La VDF en question se porte sur le calcul de la racine carré modulo p , avec $p \equiv 3 \pmod{4}$. A partir de x , on souhaite calculer $y = \sqrt{x} \pmod{p}$.*

Or, le groupe $\mathbb{Z}/p\mathbb{Z}$ a la bonne particularité que pour un carré x , sa racine est égale à $x^{\frac{p+1}{4}}$. Ce qui nous ramène à un calcul de puissance, plus pratique à calculer et demande $\log_2(\frac{p+1}{4})$ opérations.

En revanche, lorsque l'on a une racine carré de x potentielle y , il est beaucoup plus efficace de le vérifier. En effet, il suffit de calculer $y \times y = y^2$ et vérifier si l'on retombe sur notre x . Cette vérification se fait une seule opération.

Dans ce cas, la fonction *Setup* renvoie le nombre premier p et la classe x ; *Eval* correspond à la fonction d'exponentiation et *Verify* à celle du passage au carré.

En plus de ça, une VDF doit respecter les propriétés suivantes :

Définition 6 (Exactitude). *Une VDF V est correcte si pour toutes évaluations effectuée correctement, la vérification renvoie bien oui.*

Autrement dit, une VDF qui vérifie l'exactitude ne peut pas donner de faux négatif (retourner *non* alors que le calcul a été fait correctement).

Définition 7 (Solidité). *Une VDF V est solide si la probabilité pour une fausse évaluation y' retourne oui pour la vérification est négligeable devant λ .*

Désormais, nous allons voir qu'est-ce qu'une VDF peut nous apporter.

3.1.2 Avantages & inconvénients

Premièrement, l'utilisation des VDFs dans un consensus comporte plusieurs avantages.

Choix du validateur Dans l'idée de notre consensus, la VDF serait présente afin de remplir la tâche principale de la sélection pseudo-aléatoire du prochain leader. Lors du calcul d'une VDF, un utilisateur malveillant ne peut trafiquer le résultat puisqu'il n'a aucun impact sur celui-ci³.

Contrôle La VDF est ainsi comprise dans le calcul à effectuer du validateur pour ajouter son bloc au réseau. C'est le délai du calcul qui permettra de réguler le temps entre chaque bloc, permettant ainsi aux autres participants la vérification de la VDF et donc de la décision pseudo-aléatoire du prochain leader.

Économique Le calcul d'une VDF ne pouvant être parallélisable et ne demandant pas d'importantes ressources pour être réalisée dans un temps raisonnable, si les paramètres sont correctement définis, nous respecterions nos engagements en terme d'économie et donc d'écologie. De fait, notre blockchain serait davantage décentralisée, car plus accessible sur la question du coût matériel.

Il reste cependant un inconvénient à prendre en compte.

Stockage Notre consensus serait logiquement plus lourd et moins rapide qu'une simple PoS.

L'idée d'utiliser une VDF dans une blockchain a déjà été expérimentée avec Solana qui en utilise une dans son consensus. Nous la présentons dans la section suivante.

3. Ce qui n'est pas le cas du DRB (*Distributed Randomness Beacon*) pour lequel le dernier utilisateur peut avoir un effet sur le résultat, et donc le guider à son avantage.

3.1.3 L'exemple de Solana

Solana (2017) est une blockchain assez nouvelle dans l'écosystème si comparaison est faite avec l'aîné Bitcoin. Notamment grâce à son tps (transactions per second) plus important que ne pouvait l'être les premières blockchains, elle a su conquérir le monde du Web3.0.

Preuve d'historique

Solana utilise une PoS couplée d'une PoH (*Proof of History*), qui permet d'ordonner les blocs dans le bon ordre. Le principe est de hacher (avec ici SHA-256) l'hora-date directement dans les transactions. Cela permet d'augmenter les performances et le débit sur le réseau. Ainsi, la PoH est utilisée pour ordonner de manière efficace les événements (ici les blocs) écoulés.

Fonction de hachage séquentielle

C'est là qu'intervient la VDF, qui est donc ici basée sur les fonctions de hachages.

Cette VDF consiste à calculer en premier lieu le hash de l'entrée donnée un certain nombre de fois. À noter que ce calcul est impossible à paralléliser puisque chaque résultat/chaque hash dépend des précédents.

En ce qui concerne la vérification, il est tout à fait possible de paralléliser les calculs lorsque les résultats intermédiaires sont aussi enregistrés. Si chaque hash est correctement vérifié, alors le bloc est validé.

Bien que ce système fonctionne, nous avons voulu directement travailler avec une autre VDF, une basée sur les isogénies.

3.1.4 Une VDF basée sur les isogénies

Post Quantum Resistance

La VDF en tant que telle n'est pas forcément PQR : le calcul qui est censé être long pourrait s'effectuer de manière plus rapide avec un ordinateur quantique (si, par exemple, l'on demandait la factorisation d'un grand nombre, ce calcul pourrait rapidement être réalisé avec l'algorithme de Shor). Avec le développement de cryptosystèmes PQR (avec la compétition du NIST notamment), on voit émerger toutes sortes de primitives cryptographiques qui sont conçues pour résister à la cryptanalyse quantique : c'est le cas de la VDF que nous avons choisi d'implémenter dans notre blockchain.

C'est une VDF⁴ basée sur les isogénies, qui utilise des notions de courbes supersingulières ainsi que des notions de courbes de Montgomery et de correspondances de Richelot.

Nous avons choisi d'utiliser cette VDF, pour deux raisons principales :

4. Voici le lien GitHub du repo : <https://github.com/pq-vdf-isogeny/pq-vdf-isogeny.git>.

- C'est la seule VDF PQR trouvée nous paraissant fiable qui était open-source : nous avons ainsi pu en récupérer le code.
- Elle est développée en Sagemath, un langage avec lequel nous sommes familier.

L'algorithme

Setup La fonction prend en entrée 3 arguments :

- λ : entier qui correspond au niveau de sécurité ;
- μ : entier qui correspond à la taille de premier p ;
- t : entier correspond à la complexité souhaité.

et en sortie retourne deux clés :

- $e_k = (p, b, l, j)$: la clé d'évaluation, avec l un premier dans un voisinage de t , p un premier de taille μ vérifiant certaines conditions et $b > \lambda$ tel que $2^b = c^2l + d^2$, pour $c, d \in \mathbb{N}$ premiers entre eux. Il y a aussi l'entier j , qui est obtenu comme étant j-invariant de la courbe elliptique obtenue par l'opération `E0.isogenies_prime_degree(3)[1]`, avec $E0 : y^2 = x^3 + 1 \pmod{p}$.
- $v_k = (p, b, l, c, d, j)$: la clé de vérification, comportant la clé d'évaluation, ainsi que les deux entiers c et d tels que $2^b = c^2l + d^2$.

Eval prend en entrée la clé d'évaluation $e_k = (p, b, l, j)$, et retourne en sortie :

- P_1, Q_1, P_1p, Q_1p : points de Montgomery de la forme $(X_1 \cdot i + X_0 : Y_1 \cdot i + Y_0 : 1)$, avec X_1, X_0, Y_1, Y_0 des entiers et i une indéterminée ;
- E_1, E_1p : des courbes elliptiques de la forme : $y^2 = x^3 + ax^2 + bx + c$, avec $a, b, c \in \mathbb{F}_{p^2}$

Verify prend en entrée la clé de validation v_k ainsi que la sortie de l'évaluation et sort retourne un booléen.

Les résultats

Nous avons testé la VDF sur quelques machines pour analyser, à notre échelle, la variabilité des temps de calculs. Les résultats que nous affichons dans le tableau ci-dessous sont ceux obtenus avec les paramètres de base fournis par la VDF que nous utilisons. Ainsi, nous obtenons en moyenne :

Processeurs	setup	eval	verif
Intel64 Family 6 Model 142 Stepping 11 GenuineIntel 1600 MHz	0.40 sec	130 sec	13 sec
Intel64 Family 6 Model 142 Stepping 10 GenuineIntel 1600 MHz	40 sec	375 sec	32 sec
Intel 64 Family 6 Model 186 Stepping 3 GenuineIntel 1300MHz	0.35 sec	117 sec	12 sec
13th Gen Intel(R) Core(TM) i7-13700 @ Max 5200 MHz	0.39 sec	41.87 sec	5.04 sec

TABLE 3.1 – Temps moyen pour différentes machines

L'implémentation

Après avoir compris la structure globale du programme (développé en SageMath), nous avons débuté l'implémentation de cette VDF basée sur les isogénies dans notre projet. Pour ce faire nous devons séparer le setup et l'évaluation de la vérification. Cette idée dans l'objectif qu'un validateur doit pouvoir effectuer le setup et l'évaluation puis, un vérificateur (comprendre, un noeud éligible à être validateur du réseau, soit, un staker) doit pouvoir effectuer la vérification afin de valider ou invalider le bloc créé. Bien que cela semble possible, nous n'avons pu parvenir à un tel objectif. Effectivement, nous avons une erreur lors de la vérification si celle-ci était réalisée indépendamment des étapes de setup et d'évaluation. N'ayant pu résoudre le problème posé, nous avons ouvert une "issue", sur le repo github du projet, toujours sans réponse aujourd'hui (voir figure 3.1.



FIGURE 3.1 – Capture d'écran de l'issue ouverte sur le repo github du projet "pq-vdf-isogeny". Pour en savoir plus sur le problème rencontré, nous vous redirigeons vers le lien de l'issue : <https://github.com/pq-vdf-isogeny/pq-vdf-isogeny/issues/1>

De ce problème, nous devons donc réfléchir à choisir ou développer une nouvelle VDF. Nous nous sommes donc inspiré de la PoH de Solana et avons développé notre propre version.

3.1.5 Notre choix : Une VDF basée sur les fonctions de hachage

LevelDB

LevelDB est un système de gestion de base de données (SGBD) libre de type NoSQL, basé sur un LSM-Tree (Log Structured Merge Tree ou Arbre de Fusion Structuré en Journaux). Un LSM-Tree est une structure de données qui gère une correspondance clé-valeur fonctionnant sur au moins deux niveaux : en mémoire et sur le disque. Les données sont d'abord stockées en mémoire, et si la quantité de données dépasse un certain seuil, elles sont finalement écrites sur le disque. La partie sur disque est constituée de tables de chaînes triées immuables (SSTable). Les SSTable stockent les paires clé-valeur triées dans l'ordre des clés..

Étant donné que les LSM-Trees ont été initialement conçus pour gérer des charges de travail d'écriture importantes, leur fonctionnement a été optimisé pour les écritures séquentielles volumineuses plutôt que pour les petites écritures aléatoires. Cette approche permet de réduire le temps de recherche et la latence des accès disque (HDD/SSD).

LevelDB permet donc d'insérer, de supprimer et d'ordonner des données sous forme de paires clé (type : string) - valeur (type : string) de manière persistante, sans nécessiter de relation client-serveur, car tout est géré par la bibliothèque du même nom sur la machine cliente.

Son utilité dans notre blockchain est ainsi primordiale. LevelDB nous servirait alors pour :

- Stocker l'index blocs en fonction de leur profondeur
- Stocker le vecteur de hash créé par le validateur pour chaque bloc (voir ci-dessous)
- Stocker l'UTXOset⁵
- Possiblement le fichier StakerHash.db (si nous utiliserions SQLite3, nous le nommerions StakerHash.dat)

Pour tester et comprendre son utilisation, nous avons opté pour l'utilisation d'un script JavaScript. Ce choix s'explique par la simplicité d'utilisation de JavaScript par rapport au C ou au C++. Étant donné qu'une bibliothèque (level) permet d'interagir avec LevelDB en JavaScript, il nous semblait plus pratique de tester directement notre SGBD de cette manière. La figure 3.2 présente notre script tandis que la figure 3.3 présente le résultat obtenu. La figure 3.4 présente le dictionnaire créé via la commande 'leveldbutil'.

5. L'UTXOset permet de stocker la quantité de tokens relatives à chaque adresse.

```
const { Level } = require('level')

async function main() {
  // Create a database
  const db = new Level('dbtest', { valueEncoding: 'json' })

  try {
    // Add an entry with key 'a' and value 1
    await db.put('a', 1);
    await db.put('b', 2);
    await db.put('c', 3);
    console.log('Entry added successfully');

    // Synchronize the data to disk
    await db.close();
    await db.open();

    // Get the value associated to the key
    const valuea = await db.get('a');
    const valueb = await db.get('b');
    const valuec = await db.get('c');

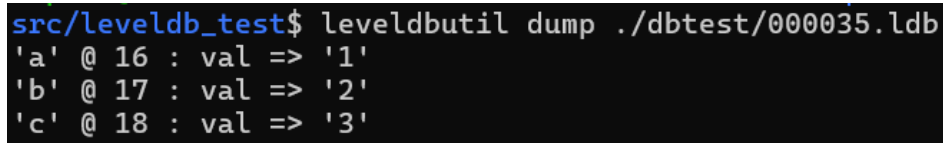
    console.log("Value for key 'a':", valuea);
    console.log("Value for key 'b':", valueb);
    console.log("Value for key 'c':", valuec);
  } catch (error) {
    console.error('Error:', error);
  }
}

main().catch(console.error);
```

FIGURE 3.2 – Capture d'écran d'une partie du script "leveldb_test.js".

```
src/leveldb_test$ node leveldb_test.js
Entry added successfully
Value for key 'a': 1
Value for key 'b': 2
Value for key 'c': 3
```

FIGURE 3.3 – Capture d'écran du résultat obtenu via l'exécution du script "leveldb_test.js".



```
src/leveldb_test$ leveldbutil dump ./dbtest/000035.ldb
'a' @ 16 : val => '1'
'b' @ 17 : val => '2'
'c' @ 18 : val => '3'
```

FIGURE 3.4 – Capture d’écran de l’exécution de la commande ‘leveldbutil’ pour connaître le dictionnaire enregistré dans une database créée par LevelDB.”.

Fonctionnement de notre VDF

Notre VDF (désormais développée en C++) comprend deux méthodes principales : une pour l’évaluation et une autre pour la vérification, pouvant être exécutées indépendamment.⁶

Évaluation : L’évaluation implique le hachage répété d’un fichier de block généré par le validateur, résultant en un vecteur contenant un indice de thread et le hash associé à cet indice. Ce vecteur est ensuite stocké dans un fichier créé par LevelDB et intégré au block proposé. Les vérificateurs extraient ce fichier LevelDB du block, qui contient les paires clé-valeur à vérifier. Ils exécutent alors des opérations de hachage en parallèle pour assurer l’intégrité du block et valider sa soumission.

Vérification : Chaque thread exécute 128 hashes (ou 78125, définition probablement à revoir) sur le block soumis et compare son empreinte finale avec le hash de la valeur associée à la prochaine clé. Chaque clé correspond à un thread, ce qui signifie que chaque thread récupère sa valeur associée, effectue un certain nombre d’opérations de hachage sur cette valeur (ici 128), puis compare son résultat avec la valeur associée à la prochaine clé. Si une assertion échoue, cela signifie que l’intégrité du block n’est pas respectée. Dans ce cas, le vérificateur transmettra sa réponse (False) aux autres nœuds, et cette information sera stockée dans un fichier nommé "vempool.dat". Ce fichier peut être conçu avec SQLite sans nécessiter l’utilisation de SQLCipher dans ce cas précis. Ici, l’intérêt du parallélisme est d’aller toujours plus vite, de concevoir une blockchain qui puisse donner la possibilité à tous les nœuds de vérifier et ce, rapidement, quelque soit leur infrastructure hardware. Effectivement, nous souhaitons utiliser OpenMP afin que seul un CPU puisse effectuer les calculs.

Setup : Le setup est lui fournit en dur dans le code source de notre blockchain et consiste en la définition du nombre de threads et de hash à opérer.

6. Pour différencier les blocs de la blockchain et les blocs utilisés dans des opérations de parallélisme, nous écrirons "block" pour ceux de la blockchain et "bloc" pour ceux associés au parallélisme.

Les figures 3.5 et 3.6 présente le code écrit pour réaliser l'étape d'évaluation. Nous omettrons de présenter pour ce rapport le fichier "sha256_file.cpp" permettant de réaliser les opérations de hachage. Vous le retrouverez cependant dans l'archive de notre projet.

```
// in CUDA a block is 128 threads. Then 78125*128 = 10000000 rounds, calcul is also working with OpenMP even if there is no block
const int nb_rounds = 10000000;
const int nb_work_by_thread = 78125;
unsigned char sha256_hash[SHA256_DIGEST_LENGTH];

map<int, vector<char>> consensus_evaluation(const char* block){

    bool isDataIsFile = true;
    map<int, vector<char>> hash_map;
    int i = 0;

    while(i < nb_work_by_thread) {

        vector<char> hash_vector;

        if (isDataIsFile) {
            //only the first part is hashed (nb_rounds/nb_work_by_thread)
            if (sha256_file_fun(block, sha256_hash, nb_rounds/nb_work_by_thread, isDataIsFile) == 0) {
                // Adding the hash to hash_vector
                for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
                    hash_vector.push_back(sha256_hash[i]);
                }
            } else {
                cerr << "Error in the sha256 calcul of the block" << endl;
            }
        }
        // Won't hash n times the file rn but n times FROM the nb_rounds/nb_work_by_thread ^th hash of the file (for the first loop), and so
        else {
            if (hash_map.find(i-1) != hash_map.end()) {
                // Get the previous hash from dictionary
                const vector<char>& previous_hash = hash_map[i-1];
                if (sha256_file_fun(&previous_hash[0], sha256_hash, nb_rounds/nb_work_by_thread, isDataIsFile) == 0) {
                    for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
                        hash_vector.push_back(sha256_hash[i]);
                    }
                } else {
                    cerr << "Error in the sha256 calcul of the block" << endl;
                }
            }
        }
    }
}
```

FIGURE 3.5 – Capture d'écran d'une partie du code source de notre consensus, dans le fichier "consensus_main.cpp". Cette image présente la première partie de la méthode permettant de réaliser l'évaluation. Les paramètres globaux "nb_rounds" et "nb_work_by_thread" définissent le setup.

```

// Won't hash n times the file rn but n times FROM the nb_rounds/nb_blocks ^th hash of the file (for the first loop), and so on
else {
    if (hash_map.find(i-1) != hash_map.end()) {
        // Get the previous hash from dictionary
        const vector<char>& previous_hash = hash_map[i-1];

        if (sha256_file_fun(&previous_hash[0], sha256_hash, nb_rounds/nb_blocks, isDataIsFile) == 0) {
            for (int i = 0; i < SHA256_DIGEST_LENGTH; ++i) {
                hash_vector.push_back(sha256_hash[i]);
            }
        }
        else {
            cerr << "Error in the sha256 calcul of the block" << endl;
        }
    }
    else {
        cerr << "Error, no previous hash calculated" << endl;
    }
}

// add the hash_vector to the map/dictionary
hash_map.insert({i, hash_vector});
// We're no longer working with the file but with the hash
isDataIsFile = false;
i++;
}
return hash_map;
}

```

FIGURE 3.6 – Capture d'écran d'une partie du code source de notre consensus, dans le fichier "consensus_main.cpp". Cette image présente la seconde partie de la méthode permettant de réaliser l'évaluation.

La méthode de vérification n'ayant été développée nous ne pourrions vous la présenter. Cependant, nous savons que nous l'aurions développé avec OpenMP afin de rendre les étapes de parallélisme accessibles à tous. La première idée que nous avions était d'utiliser CUDA, cependant, son installation peut être lourde et fastidieuse, ce qui réduirait considérablement l'accès à notre blockchain. De par cette volonté, OpenMP semble être la version la plus accessible. La méthode "void display_op(map<int, vector<char>> hash_map)" peut être dispensée lors de l'exécution de l'évaluation, elle permet simplement d'afficher en console les résultats. Les résultats ont été vérifiés par un script Python réalisant lui aussi le hachage d'un fichier donné en entrée.

Résultats

La figure 3.7 présente l'exécution de l'étape d'évaluation de notre VDF ainsi que l'affichage de sa dernière paire clé-valeur.

```
src/consensus$ ./run_consensus
Last value from the last key (hex) :
key : 78124
value : f792a6a9fe5e41994fe9507d8ff115193c870aeeb1e4d0c593a59785099317f7
```

FIGURE 3.7 – Capture d’écran de l’exécution du programme réalisant l’étape d’évaluation du consensus sur un fichier donné. Ici le fichier donné est "new_blk03802.dat" (retrouvez ce fichier dans le path "blocks" de l’archive), soit, une copie factice d’un block Bitcoin).

3.1.6 Fonctionnement de la VPoRS

Enfin, nous vous présentons le coeur de notre blockchain, notre consensus : la Verifiable Proof of Random Staker (VPoRS).

Notre schéma complet est disponible dans l’archive de notre projet sous format .png. Nous décrirons dans cette partie du rapport le fonctionnement global de la VPoRS, pour davantage de détails, veuillez vous reporter sur le schéma fourni dans l’archive du projet.

1. Alice initie l’envoi de 5QPT (devise de notre blockchain) à Bob. Cette transaction tx_A comprend naturellement des frais et une taille (exprimée en kB).
2. La transaction tx_A est incluse dans la mempool, fichier répertoriant toutes les transactions en attente d’inclusion dans un bloc.
3. Un staker s_i du réseau est désigné (nous expliquerons la phase de désignation).
4. Le staker s_i sélectionne les transactions de la mempool les plus intéressantes selon le rapport poids/frais des transactions, dont tx_A .
5. Notre staker s_i génère aléatoirement k_i avec un PRNG, pour le prochain staker s_{i+1} .
6. Une fois son bloc b_i construit, s_i procède à l’étape d’évaluation de la VDF vu précédemment.
7. Pour s’assurer que son évaluation est correcte, s_i effectue la vérification de b_i avant son envoi sur le réseau.
8. Si la vérification a fonctionné, s_i hash k_{i-1} fois (où k_{i-1} est déterminé par s_{i-1} durant son étape 5) le résultat du hash de son bloc. Il obtient alors le résultat noté $Hvdf_i$.
9. Chaque noeud possède un fichier "StakerHash.dat/db" contenant les adresses des autres stakers du réseau. Ces adresses sont hashées elles aussi k_{i-1} fois au moins (on notera une de ces adresses $Haddr$). Chaque adresse possède, au lancement de notre blockchain, 5 empreintes différentes dans le fichier avec un minimum de 1 et un maximum de 10 versions hashées différentes. Ces paramètres pourraient s’auto moduler selon le nombre de stakers dans le réseau, de la même

manière que Bitcoin module la difficulté de minage. Le nombre d'empreintes dans le fichier "StakerHash.dat/db" dépend du score scr_{addr} associé à l'adresse. Ainsi, une adresse est hashée une fois de plus à chaque itération, jusqu'à ce que la valeur de l'itération atteigne le score correspondant à l'adresse : scr_{addr} . À chaque itération, l'adresse hashée est ajoutée au fichier "StakerHash.dat/db". Par exemple, une adresse donnée $Haddr$ à un score de 3, alors 3 versions hashées de cette adresse subsisteront dans le fichier "StakerHash.dat/db", à savoir :

- Une version hashée k_{i-1} fois
- Une version hashée $k_{i-1} + 1$ fois
- Une version hashée $k_{i-1} + 2$ fois

10. La valeur en base 10 de l'empreinte dans le fichier "StakerHash.dat/db" étant la plus proche de l'empreinte $Hvdf_i$ sélectionnera l'adresse associée comme prochain leader de la blockchain. C'est ici un simple calcul de distance qui s'opère.
11. Une fois ceci fait, le leader s_i hash une fois son bloc.
12. s_i partage ensuite son bloc b_1 , contenant tc_A , au réseau par flood.
13. Chaque staker⁷ doit procéder à la vérification du bloc b_i proposé en réalisant l'étape "Verification" de la VDF (pour s'assurer de l'aléatoire et de l'intégrité du prochain leader sélectionné).
14. Chaque staker procède au calcul de l'arbre de Merkle fournit dans le bloc b_i afin de s'assurer de l'intégrité des transactions en comparant le résultat obtenu à l'arbre de Merkle qu'ils calculent de leur côté sur les transactions de la mempool choisies pour le bloc b_i proposé.
15. Chaque staker procède au hash unique du bloc b_i en question pour s'assurer de l'intégrité de l'intégralité des données transmises. L'opération est : $H(b_i)$ où $H()$ est une fonction de hachage.
16. Chaque staker transmet aux autres noeuds du réseau leur validation ou invalidation du bloc (en fonction de s'ils trouvent des erreurs ou non dans les vérifications réalisées). Ce booléen est inclus dans le fichier "vempool.dat", contenant les résultats des vérifications (vempool = mempool des vérification).
17. Si $\geq 50\%$ des stakers invalident le bloc, alors le leader s_i est "slashé", c'est à dire puni. Une partie de ses tokens stakés seront brûlés/détruits et son score $scr_{(addr(s_i))}$ sera de 1 seulement. Les stakers ayant transmis l'information jugée comme correcte (ici un booléen=False) seront récompensés en voyant leur score croître tandis que les autres seront eux aussi "slashés" en ne perdant cependant que des points de score, leur tokens ne seront pas brûlés ici.
Dans ce cas de figure, le leader devant construire le bloc b_i venant d'être invalidé sera alors le validateur précédent, s_{i-1} .

7. Soit, de potentiels futurs leader. Les stakers doivent ici être des noeuds du réseau. Un noeud du réseau n'est pas nécessairement staker.

18. Si $\geq 50\%$ des stakers valident le bloc, alors le leader s_i devient validateur et est récompensé par la transaction COINBASE⁸. Son score ne varie cependant pas. Les stakers ayant transmis l'information jugée comme correcte (ici un booléen=True) seront récompensés en voyant leur score croître tandis que les autres seront "slashés" en ne perdant cependant que des points de score. Leur tokens ne seront pas brûlés.
19. Bob reçoit enfin la transaction d'Alice (tx_A) et l'UTXOset est mis à jour.
20. Si b_i a été validé par le réseau, le prochain leader s_{i+1} peut désormais débiter la création du bloc b_{i+1} en incluant le fichier "vempool.dat" dans son bloc afin de garder une trace des blocs validés ou non et quelles sont les adresses ayant répondu.
21. Les transactions sélectionnées dans le bloc précédent sont retirées de la mem-pool. Le fait que ces transactions ne soient supprimées que par le leader s_{i+1} permet à tous les noeuds du réseau de vérifier l'arbre de Merkle du bloc b_i .

Si un bloc n'a pas été proposé dans un temps imparti (que nous définirions en dur dans le protocole de notre blockchain, soit le protocole Qrypto), alors le prochain leader est le dernier validateur en date.

3.1.7 Devenir staker

Pour devenir staker, soit potentiel leader et par extension, potentiel validateur, l'utilisateur devrait, dans une blockchain normalement constituée permettant la Proof of Stake (PoS) déposer ses tokens/cryptomonnaies, dans un smart contract. Pour rappel, un smart contract est un simple programme permettant d'effectuer des actions sous certaines conditions vérifiées par la blockchain. Dans le cas du staking, l'utilisateur dépose ses tokens à une adresse correspondant à l'adresse d'un smart contract et ainsi, s'il a staké/déposé assez, il devient éligible à devenir leader. Ce cas de figure est

le plus adapté, cependant cela requiert d'ajouter de la programmabilité à notre blockchain ce qui n'est pas, dans un premier temps, notre objectif. Ainsi, pour palier ce problème nous proposons aux utilisateurs souhaitant staker, d'envoyer leurs fonds directement sur un dead wallet (adresse pour laquelle la clé privée associée n'est connue de personne). Pour reconnaître cet envoi, la transaction serait enregistrée dans le fichier "UTXOStakingSet.db". Ce fichier LevelDB permettrait ainsi de répertorier les UTXOs des stakers, notamment la transaction leur ayant permis de devenir staker. Enfin, l'utilisateur serait enregistré dans le fichiers "stakers.dat" répertoriant tous les stakers. Le nouveau staker aurait donc un score initial associé qui correspondrait à la moitié du score maximum atteignable. Par exemple si le score maximum est de 10, alors un nouveau staker aurait un score initial de 5. Le fichier "stakers.dat" prendrait donc l'adresse du staker, dans quel bloc est son UTXO (ou transaction) ainsi que son

8. Permettant la création de tokens qui sont ici directement envoyés au validateur.

score. Pour rappel, le score évolue en fonction du comportement du validateur (voir fonctionnement de la VPoRS).

Chapitre 4

Conclusion

4.1 Ce qu'il reste à faire

N'ayant pu terminer le développement informatique de notre blockchain, cette section de notre rapport présente ce qu'il reste actuellement à développer.

4.1.1 Construire les blocs

Quelle données ?

Une blockchain sans donnée n'est pas une blockchain. Alors qu'allons nous rajouter de différent des autres blockchains dans nos blocs ? Le listing ci-contre le présente.

Block fields

- Magic Number \implies Permet de spécifier le réseau (mainnet ou testnet dans notre cas).
- Blockheader \implies Correspond à l'entête d'un bloc. Nous détaillons son contenu à la suite de la section "Block fields".
- Blocksize \implies Correspond à la taille du bloc (exprimée en Mo).
- Transaction counter \implies Correspond au nombre total de transactions dans le bloc.
- Transactions \implies Correspond aux transactions.
- Merkle Root \implies Correspond à la racine de l'arbre de Merkle sur les transactions.
- k \implies Correspond à la valeur pseudo-aléatoire générée.
- VectorVDF \implies Correspond au vecteur résultant de l'évaluation de la VDF.
- $Addr_{nextleader}$ \implies Correspond à l'adresse du leader sélectionnée par le dernier hash de la VDF et k .

Block header

- Magic Number \implies Permet de spécifier le réseau (mainnet ou testnet dans notre cas).

- $nb_{rounds} \implies$ Correspond au nombre de rounds qui doivent être effectués pour réaliser le calcul de la VDF.
- $timestamp$ et son certificat \implies Correspond au moment précis auquel le leader a finalisé la construction de son bloc b_i .
- $Hvempool \implies$ Correspond au hash de la vempool du bloc précédent b_{i-1} pour garder une trace de la validité du bloc + pour créer la relation d'interdépendance entre les blocs.
- $HB_{i-1} \implies$ Correspond au hash du bloc précédent b_{i-1} pour créer la relation d'interdépendance entre les blocs.

Le problème de l'horodatage évité

L'horodatage (ou timestamp) est une problématique dans les blockchains aujourd'hui. Par exemple, dans Bitcoin est utilisé le timestamp délivré par Unix (Unix Time Stamp). En utilisant celui-ci, Bitcoin fait face à 2 obstacles :

- L'Unix Time Stamp ne fournit des périodes de temps que jusqu'à l'année 2038.
- Sujet à la Time Warp Attack qui permet à un mineur de modifier le timestamp d'un block afin d'altérer le protocole de la blockchain (dans notre exemple, le protocole Bitcoin), soit, le consensus de celle-ci. Par exemple, sur Bitcoin, réduire la difficulté de minage lors de sa mise à jour (tous les 2016 blocs).

Pour résoudre la capacité d'Unix à gérer les timestamps, Satoshi Nakamoto a programmé le système d'horodatage pour éviter cette défaillance et l'a retardé jusqu'à l'année 2106. Cependant, la Time Warp Attack est toujours un problème d'actualité pour Bitcoin.

Notre solution pour répondre à ces problématiques serait l'usage d'une autorité d'horodatage délivrant un certificat de validité pour chaque requête. Nous pensions à sélectionner au moins 2 autorités afin d'en avoir une de secours si la première venait à rencontrer des soucis. Pour l'une d'entre elle, nous avons pensé à utiliser "Free TSA"¹. Ce choix s'explique par sa simplicité d'usage (de simples requêtes curl) mais aussi puisque nous sommes familier avec cette autorité pour l'avoir déjà utilisé dans un projet.

4.1.2 Construire le réseau

Les données sur disque

Afin que notre réseau soit effectif, il est nécessaire que les noeuds aient certaines données à minima sur leur disque où tourne notre blockchain "Qrypto". Ces données sont présentées ci-dessous. Cette présentation fait office de résumé des données stockées par chaque noeud.²

1. Free TSA : https://freetlsa.org/index_en.php

2. Rappel : les .dat font références aux fichiers SQLite et les .db aux fichiers LevelDB.

- Le logiciel "Qrypto" permettant au protocole "Qrypto" (soit le consensus VPoRS et tout ce qui peut graviter autour de son fonctionnement) de fonctionner.
- mempool.dat \Rightarrow Répertoriant les transactions en attente d'être placées dans un bloc.
- vempool.dat \Rightarrow Répertoriant les validités ou non du bloc courant proposé.
- stakers.dat \Rightarrow Répertoriant tous les stakers.
- StakerHash.dat/db \Rightarrow Répertoriant les hashes des adresses des stakers afin qu'ils soient potentiellement sélectionnés comme prochain leader.
- UTXOSTakingSet.db \Rightarrow Répertoriant les UTXOs des stakers, notamment la transaction leur ayant permis de devenir staker.

Mécanisme pair-à-pair (P2P)

Pour construire le réseau nous utiliserions de la programmation Python qui nous permettrait de créer aisément le réseau pair-à-pair (P2P) prévu. Pour rappel, la blockchain est un réseau pair-à-pair construit pour la création d'une base de données distribuée. La figure 4.1 présente les différents types réseaux. Nous devrions donc développer un réseau décentralisé pour construire une base de données distribuée.

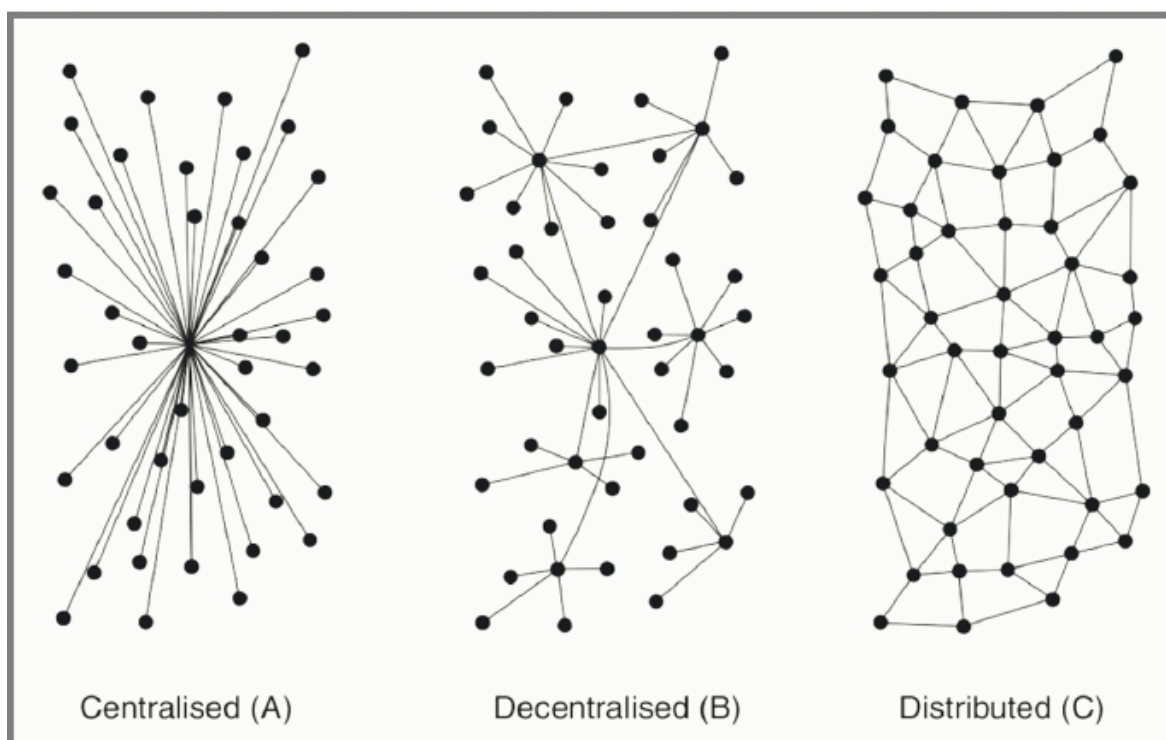


FIGURE 4.1 – Les différents réseaux schématisés par Paul Baran. A gauche, un réseau centralisé, au centre un réseau décentralisé (le réseau sur lequel repose la blockchain), à droite un réseau distribuée (ce qui peut donc représenter la base de données qu'est la blockchain).