Nexium

Rapport de soutenance n°2

Jean HERAIL
jean.herail@epita.fr

Milo DELBOS
milo.delbos@epita.fr

Antonin BESSIÈRES
antonin.bessieres@epita.fr

William VALENDUC
william.valenduc@epita.fr

Sommaire

1 Introduction

Dans le cadre du projet de quatrième semestre à l'Epita, notre groupe a pris la décision d'entreprendre la création d'un outil que nous voulions pertinent, utile et ayant un sens pour les gens. C'est de cette conception particulière que naquit l'idée de **Nexium**.

Il s'agit d'un protocole transactionnel pair-à-pair, basé sur la cryptographie, et destiné aux membres d'Epita (étudiants et personnels éducatifs). Basé sur la technologie de **blockchain** [14], et construit selon le **Nakamoto-Consensus** [1] établi avec le **Bitcoin** [2] en 2008, Nexium s'assimile donc comme une réelle crypto-monnaie décentralisée, **trustless** [3], indépendante et résiliente. Le produit final de Nexium se décline sous deux formes binaires : un client, et un serveur.

Nous expliquerons en détail le rôle et l'agencement de ces deux pôles fondamentaux. Conformément aux contraintes techniques requises par le sujet, tout le projet est développé en **Rust** [24]. Ce langage de programmation, reconnu pour ses performances et sa **memory-safety**, [30] est particulièrement propice et adapté dans le cadre d'un projet impliquant des notions de sécurité et cryptographie comme Nexium. La popularité de Rust rend tout aussi appréciable la découverte de son fonctionnement : la richesse de sa **documentation** [22] et l'engagement de sa communauté permettent de résoudre aisément d'éventuels problèmes que l'on rencontre en développant.

En outre, Nexium est un projet aussi ambitieux que captivant : **cryptographie, architecture et communication réseau, développement web, et travail de groupe** sont autant de compétences que chacun d'entre nous développe dans la réalisation de cet objectif commun.

Nous aborderons dans ce rapport la suite des **avancées** que nous avons pu réaliser depuis le début du projet. Les notions techniques les plus importantes du projet seront explorées, et nous détaillerons également la situation globale du groupe, de son organisation, et des perspectives à venir.

[1] Le consensus de Nakamoto fait référence à la manière spécifique dont le réseau décentralisé de Bitcoin parvient à un accord sur l'état de sa blockchain.[19]

[2] La première et la plus capitalisée des cryptomonnaies, créée en 2009. [2]

[3] Le terme "trustless" ("sans confiance" en français) désigne un système ou des transactions où les parties peuvent interagir et échanger de la valeur sans avoir besoin de faire confiance à un tiers. Dans le contexte des cryptomonnaies, la trustless est souvent réalisée grâce à des protocoles cryptographiques basés sur la notion de "preuve de travail".

2 Présentation des membres

2.1 Jean HERAIL

Mon rôle pour ce début de projet est de développer, en tandem avec William, l'architecture de la blockchain et du réseau. Nous écrivons aussi, au fur et à mesure de notre avancement, la documentation théorique de Nexium au format `Markdown`, intégrée dans **le site web officiel de Nexium** [16] Cela implique de comprendre les mécanismes de la **blockchain**, de la **cryptographie**, et de la **communication réseau**.

J'assume également le rôle de chef de projet, en charge de l'organisation du groupe et de la répartition des tâches. Je m'occupe également de l'écriture en `LaTex` des différents documents rendus.

Étant moi même passionné de cryptographie, c'est tout naturellement que j'ai initialement suggéré à mon groupe l'idée d'une cryptomonnaie. Accompagnée par un *synopsis* détaillant les grandes lignes théoriques du projet, l'idée a plu au groupe et a évolué au fil de plusieurs réunions et débats.

Je suis également très intéressé par le langage `Rust`, que j'ai découvert par curiosité en amont de la SPÉ, et que j'ai eu la chance de découvrir plus en profondeur lors des cours du quatrième semestre. Les **TPs hebdomadaires** me permettent régulièrement de me familiariser avec les concepts de Rust, et d'acquérir des automatismes dans l'écriture de code.

Ce projet m'enthousiasme donc tant dans sa dimension **technique** que dans sa dimension **humaine**. Je suis ravi de pouvoir découvrir et travailler sur la blockchain dans un cadre aussi concret. Je dois aussi reconnaître que c'est une chance pour moi d'avoir pu être dans un groupe dont les membres sont aussi motivés, et avec qui je m'entendais déjà très bien avant le début du projet. Cette bonne entente me semble favoriser une communication fluide et efficace.

2.2 Milo DELBOS

Je suis Milo Delbos, étudiant à EPITA et anciennement bachelier spécialité mathématiques et physique. Mon attrait pour les matières scientifiques est ce qui m'a amené à prendre ce chemin d'études et qui m'a mené jusqu'ici. Je me débrouille en mathématiques ainsi qu'en théorie sur les raisonnements algorithmiques.

Mon rôle dans ce projet consistera, avec Antonin, à s'occuper de toute la partie mathématique du projet, par cela j'entends réaliser les fonctions de chiffrement, déchiffrement et les calculs nécessaires pour les changements de type.

Il est aussi évident que mon rôle est d'apporter un avis objectif et utile aux travaux de mes camarades s'ils en demandent afin d'améliorer le rendu global du projet.

Nexium représente une pente à gravir, c'est le premier gros projet de ce type dans lequel je participe, ainsi, il me permet de développer des compétences utiles et meilleures dans tous les domaines touchés (mathématiques, algorithmique, travail de groupe et compétences sociales).

Nous avons tous choisi ce projet car c'est une idée innovante et très intéressante qui n'a encore pas été vue, d'utiliser une blockchain et une crypto monnaie dans un cadre privé qui puisse répondre à beaucoup d'attentes et de requêtes et qui pourrait même être utile à l'école.

Non seulement il y a mon attrait pour ce domaine de la crypto monnaie mais aussi toutes les possibilités derrières qui ne dépendent que de notre imagination, ainsi ce projet nous permettra de transmettre ce que l'on désire tout en améliorant nos compétences.

2.3 Antonin BESSIÈRES

Je m'appelle Antonin Bessieres, étudiant en deuxième année de prépa à l'EPITA, avec un bachelor spécialité Maths et NSI.

J'ai commencé à m'intéresser à l'informatique lors de mon stage de troisième que j'ai eu la chance de faire chez Thales à Paris. Cet intérêt m'a ensuite guidé dans mes choix d'orientation.

Dans le cadre du projet, avec Milo, nous nous focaliserons en particulier sur l'aspect mathématique, en conséquence le chiffrement/déchiffrement et tous les calculs nécessaires à l'avancée du projet. Nexium est un projet que nous développerons sur le langage Rust que j'ai pu découvrir il y a peu de temps, mais que j'apprécie particulièrement.

Ce projet me permet aussi d'aborder le chiffrement et tout un aspect serveur dont je n'avais que vaguement entendu parler avant. En résumé, ce projet est un moyen de développer quelque chose de passionnant et permet une découverte de beaucoup de connaissances ainsi qu'un développement sur l'apprentissage du langage Rust.

2.4 William VALENDUC

Après 3 ans de licence informatique à la faculté des sciences et ingénierie de l'université Paul Sabatier, j'ai rejoint l'EPITA en septembre dernier.

De nature très curieuse, j'ai à coeur de comprendre le fonctionnement de technologies dans de multiples domaines.

Réalisation de la blockchain, de l'architecture et communication réseau ainsi que le serveur web et la documentation du projet.

Ne connaissant pas grand chose du monde des crypto-monnaies avant le début de Nexium, c'est très enrichissant de découvrir cet univers. Possédant un peu d'expérience dans certains domaines que nous avons à utiliser, c'est un plaisir pour moi de pouvoir partager mais également agrandir mes compétences.

3 Organisation du projet

L'idée originelle de Nexium a suscité chez chacun des membres de notre groupe un intérêt et enthousiasme laissant présager un bon processus de développement sur le projet. Afin de pouvoir concrétiser le plus solidement et durablement possible le projet, nous accordons une place importante à l'organisation du travail de groupe.

3.1 Outils utilisés et mis en place

Comme mentionné précédemment, une organisation claire et structurée permet une fluidification de la communication. Aussi avons-nous mis en place plusieurs solutions pour tenir un suivi régulier de l'avancement du projet.

3.1.1 Discord

Nous utilisons un serveur Discord pour communiquer, relever des problèmes, planifier des réunions et partager des ressources. Plusieurs salons textuels sont dédiés aux différentes parties du développement, ainsi qu'à la centralisation des différents liens. Ce serveur Discord est, en outre, la **pierre angulaire** de notre communication, la centralisation de nos échanges et ressources.

3.1.2 Gitlab

Gitlab est une solution d'hébergement de *repositories* Git. Nous avons choisi d'utiliser l'instance Gitlab hébergée par Épita pour Nexium. Cela nous permet une **meilleure gestion des collaborateurs** (avec les logins Épita), et une congruence avec le projet Nexium, puisque c'est de cette instance Gitlab que nous avons tiré l'idée de récupération des `clés publiques GPG`. De plus, Gitlab propose un outil d'hébergement de pages web (**Gitlab Pages**)[10], que nous utilisons pour automatiquement déployer notre site web, sans avoir à gérer de serveur web ou de dépendre d'un prestataire externe.

3.2 Répartition des tâches

Au-delà des réunions hebdomadaires planifiées lors de tous les week-end, notre équipe a pris la décision, pour la première phase de développement de Nexium, de se séparer en **deux groupes** distincts, chacun ayant une mission bien précise.

- **Architecture** : Ce groupe est composé de Jean et William. Ils sont en charge de l'architecture de la blockchain et du réseau, ainsi que du

développement du serveur web. Ils sont également responsables de la rédaction de la documentation théorique de Nexium.

- **Cryptographie** : Ce groupe est composé de Milo et Antonin. Ils sont en charge de la partie mathématique du projet, c'est à dire le chiffrement/déchiffrement et les calculs nécessaires pour les changements de type. Ils ont aussi la charge du `SHA-256`.

4 Pourquoi une cryptomonnaie ?

Bien que l'étendue des notions techniques que recouvre Nexium soit vaste, nous pensons qu'il est important, avant de se lancer dans les détails techniques, de préciser le sens qu'a l'existence des cryptomonnaies de manière générale.

Les cryptomonnaies, particulièrement et en premier lieu le Bitcoin, sont apparues comme une réponse aux limites et faiblesses du **système économique traditionnel**. Une devise devrait avant tout être un vecteur d'échange, une unité de mesure permettant de faciliter les transactions et de stocker la valeur. Or la monnaie moderne repose sur la confiance.

Cette confiance en les institutions financières (telles que les banques centrales) qui émettent et régulent la masse monétaire, font de ce système un système intrinsèquement **faillible**. Il repose sur des entités centralisées, susceptibles de manipulation, de politique monétaire inflationniste ou de crises économiques imprévues[17] [15]. Ces entités privées ou étatiques peuvent, indépendament de la volonté du peuple, imposer des restrictions sur l'accès aux finances et rendre la distribution financière déséquilibrée.

Dans ce contexte apparu le **Bitcoin**. Porté par le mouvement cypherpunk[11] qui cherchait dans les annes 90 à préserver vie privé, liberté d'expression, et garantie de sécurité des données dans un monde qui devenait numérique et dont on ne mesurait pas toujours l'ampleur des enjeux sociétaux à l'époque, le Bitcoin a proposé une solution radicale : une monnaie absolument décentralisée, *open-source*, sans intermédiaire de confiance. Basée sur la technologie de **blockchain**, elle permet de valider, enregistrer et émettre des transactions de manière transparente, immuable et sécurisée, sans faire appel à quelconque autorité centrale. Ainsi, le Bitcoin introduit une forme de **trustless**, où la confiance n'est plus placée en un gouvernement ou une institution, mais dans le **code informatique** qui régit le système : une concrétisation de l'adage "*code is law*"[13].

Cette **décentralisation**, loin d'être un simple détail technique, pose des questions profondes sur les notion de contrôle, d'autonomie et de souveraineté individuelle. Dans un monde où les gouvernements peuvent influencer la monnaie nationale par des politiques économiques [31] et où les institutions bancaires détiennent le pouvoir de décision sur les flux monétaires[3],le Bitcoin a offert un

espace de liberté, où le peuple peut être souverain de sa propre monnaie, sans intervention étatique. Ce modèle a non seulement des implications économiques, mais aussi **philosophiques**. Il remet en question les fondements mêmes de l'organisation de notre société, et offre une alternative radicale à la centralisation du pouvoir économique.

C'est dans cet esprit de **décentralisation**, de **souveraineté individuelle** et de **liberté** que Nexium trouve sa place. Nous voulons de Nexium qu'il incarne l'idée d'un futur où la finance et la monnaie ne seront pas seulement contrôlées par des entités centralisées, mais où chaque citoyen, chaque individu, peut participer, contribuer et décider des règles du monde dans lequel il vit.

Cette contextualisation nous paraissait naturelle à la lumière des **cours d'Éthique** que nous avons pu suivre lors du troisième semestre : le développement d'une technologie doit, tant que possible, s'accompagner d'une réflexion sur ses enjeux sociétaux et humains. Ainsi, Nexium s'inscrit dans une réflexion plus large sur l'évolution des sociétés, et sur l'émergence de nouveaux systèmes de valeurs partagées.

5 Architecture de la blockchain

Comme brièvement mentionné dans la présentation du projet, la cryptomonnaie Nexium (dont le symbole de devise est me `NXM`) est basée sur la blockchain. Définissons donc les grandes lignes de cette technologie, et comment elle est implémentée dans Nexium.

Chaque transaction effectuée par un utilisateur est signée, puis ajoutée à un **bloc** contenant un certain nombre de transactions. Ce bloc est ensuite ajouté à la **blockchain**, une chaîne de blocs contenant l'historique de toutes les transactions effectuées sur le réseau. La blockchain est un registre **public, partagé et immuable**, qui permet de garantir la transparence et la sécurité des transactions.

Chaque bloc est lié au bloc précédent par un **hash** cryptographique, ce qui assure l'intégrité de la chaîne. En effet, si un bloc est modifié, son hash est changé et le hash du bloc suivant ne correspond plus. Cela permet de détecter d'éventuelles tentatives de modification de la blockchain, et de les rejeter.

La méthode de hashage utilisée est celle du **SHA-256**, que nous détaillerons plus loin. Cette fonction de hashage est utilisée pour signer les transactions, générer les blocs, et lier les blocs entre eux.

5.1 Stockage de la blockchain

La blockchain est stockée dans un fichier binaire (`nxm_blckchn.dat`) qui contient les données codées des blocs, du **genesis block** (bloc initial) jusqu'au dernier bloc.

Nous pouvons rechercher n'importe quel bloc de la blockchain en **parcourant successivement** les blocs suivants, en partant du bloc initial.

Pour vérifier l'intégrité ou la validité d'une transaction, nous pouvons vérifier son existence dans un bloc donné à l'aide de la **racine Merkle**, avec une complexité de `O(log n)`.

5.2 Mémoire tampon (`mempool`)

La mémoire tampon est un stockage temporaire pour toutes les transactions qui ne sont pas encore incluses dans un bloc.

Cette mémoire tampon est continuellement **synchronisée** entre tous les nœuds du réseau Nexium.

Lorsque la mémoire tampon est pleine, tous les nœuds commencent à "**miner**" les transactions pour créer le nouveau bloc. Le premier nœud qui trouve le `Nonce`[4] qui satisfait la cible de difficulté diffuse le bloc sur le réseau.

C'est ainsi que la **preuve de travail** est réalisée : le `proof of work` consiste à fournir la preuve cryptographique de minage, c'est à dire le `Nonce` qui satisfait la **cible de difficulté**[5] actuelle.

5.3 Structure d'un bloc

Un bloc dans la blockchain de **Nexium** est structuré de la manière suivante :

- **En-tête du bloc** (*82 octets*)
  - `version` (*2 octets*) : **Version** de la structure du bloc.
  - `previous_block_hash` (*32 octets*) : **Hachage** du bloc précédent.
  - `merkle_root` (*32 octets*) : **Racine Merkle** des transactions du bloc.
  - `timestamp` (*4 octets*) : **Horodatage** du bloc.

[4] Le **Nonce** est une valeur que les mineurs doivent ajuster de telle sorte que le hash du bloc concerné remplisse certaines conditions (*dans le cas du Bitcoin, il doit commencer par un certain nombre de* ***0****.*)

[5] La **cible de difficulté** est un paramètre évoluant au cours du temps selon le consensus du réseau de la blockchain, et qui établit quelles contraintes doivent être remplies pour que le `Nonce` fournisse un hash valide.

  - `difficulty_target` (*4 octets*) : **Cible de difficulté** du bloc.
  - `nonce` (*4 octets*) : **Nonce** du bloc.
  - `transaction_size` (*4 octets*) : **Taille** en octets des transactions du bloc.
- **Transactions** (*variable*) : **Liste** des transactions.
  - `transaction_header` (*73 octets*) : **En-tête** de la transaction.
    * `transaction_size` (*4 octets*) : **Taille** de la transaction.
    * `timestamp` (*4 octets*) : **Horodatage** de la transaction.
    * `fees` (*2 octets*) : **Frais** de la transaction.
    * `emitter` (*64 octets*) : **Login** de l'émetteur de la transaction. (`prenom.nom`).
    * `data_type` (*1 octet*) : **Type** de la transaction.
  - `data` (*de 1 à 1719 octets*) : **Données** de la transaction.
  - `signature` (*256 octets*) : **Signature** de la transaction.

Cette conception, pour laquelle nous nous sommes directement inspirés du **Bitcoin**[26], permet de stocker les informations essentielles de chaque bloc.

Étant donné que les données sont directement écrites en binaire sans label, ni séparation, il est important de respecter scrupuleusement l'ordre de lecture/écriture de chaque champ pour ne pas avoir de problème de données corrompues. Si on prend en exemple le cas de l'en tête des blocs, sur le buffer de 82 octets, on retrouvera sur l'intervalle 0-2 la version du bloc, puis sur 2-34 le hash du bloc précédent, puis sur 34-66 le merkle root, 66-70 le timestamp, et ainsi de suite.

Elle permet aussi un parcours fluide de la blockchain depuis le **genesis block** : en sautant à chaque fois de `n` octets, avec `n` la taille du bloc, on peut aisément accéder à n'importe quel bloc de la blockchain.

L'**identifiant unique d'un bloc** correspond à son *double hash SHA-256*, c'est à dire le hash du hash du bloc. Cela permet de garantir l'**unicité** de chaque bloc, et de faciliter la recherche et la vérification de l'intégrité de la blockchain. Voici donc la formule de l'identifiant d'un bloc :

$$\boxed{\text{id(bloc)} = \text{SHA256(SHA256(bloc))}}$$

Un **Arbre de Merkel** (*merkle root*) est une structure de données arborescente qui permet de résumer efficacement un grand nombre de transactions, assurant ainsi l'intégrité et la vérification rapide des données à travers des hachages successifs. Voici une représentation graphique théorique de l'arbre de Merkle :

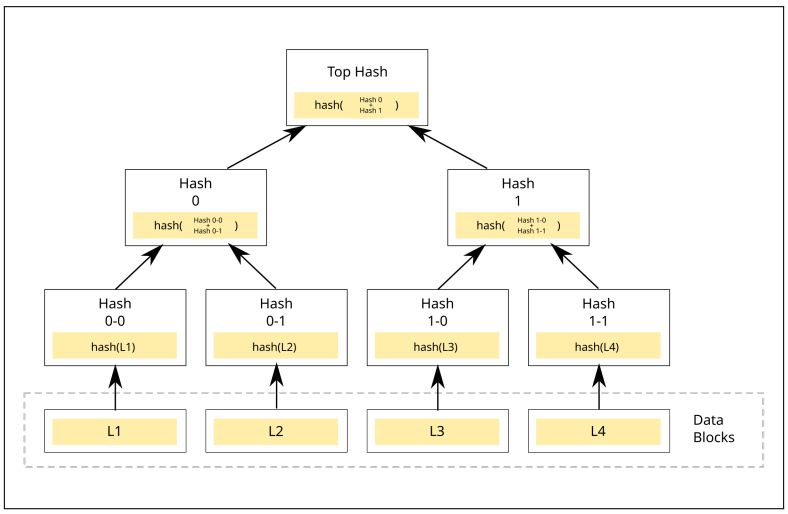

Figure 1: Arbre de Merkle.

Ainsi, à partir de la racine Merkle, on peut vérifier avec une **complexité logarithmique** si une transaction est incluse dans un bloc donné.

6 Architecture du réseau

6.1 Décentralisation

Nous avons choisi une approche **décentralisée** afin de garantir une disponibilité élevée et ainsi assurer un réseau fiable contre les pannes. Chaque noeud est donc **indépendant** des autres, pouvant ainsi continuer comme si de rien n'était en cas de noeud hors ligne ou défaillant.

Afin de garantir la communication entre tous les noeuds et d'optimiser la communication au sein réseau, nous allons nous baser sur la **théorie des graphes** [8],à savoir la *connexité*, la *bi-connexité* et d'autres propriétés. Voici une illustration du concept de **forte-connexité**, que nous utiliserons pour la distribution du réseau.

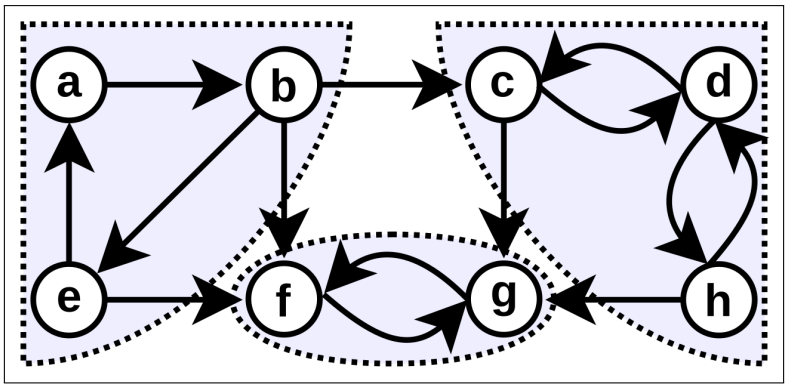

Figure 2: Composantes fortement connexes [29]

6.2 Sécurité

Nous avons songé au problème qui se poserait dans l'éventualité où une personne tenterait de posséder une part au moins égale à `51%` des noeuds du réseau. Nous avons décidé pour cette raison de restreindre les noeuds à un par membre d'Épita, en se basant sur le **login** unique de chaque élève et professeur, que l'on retrouve notamment dans son adresse mail `@epita.fr` et qui se compose par `<prénom>.<nom>`.

Nous pouvons ainsi, par un système dont nous avons théorisé les bases, accéder à la **clé publique** de n'importe quel utilisateur via l'**API de Gitlab** [9]. La requête suivante permet de récupérer la clé publique d'un `login` donné :

GET/`login`/*gpg_keys*/*key_id*

7 Hashage SHA-256

Pour notre projet, nous avons fait le choix d'avoir recours à une **méthode de hachage** [4] [6] pour garantir la **sécurité** et l'**intégrité** des informations/données lors de leur transmission, mais aussi comme *preuve de travail* (*Proof of Work*, PoW) présente dans la blockchain.

7.1 Définition d'une méthode de hachage

Définissons tout d'abord ce qu'est une **méthode de hachage**. C'est une *fonction mathématique* dont le but est de changer une chaîne de caractères avec une taille quelconque en une nouvelle chaîne de **longueur fixe et unique**. Avoir une fonction mathématique, c'est bien, mais alors pourquoi s'en servir ? La particularité d'une fonction de hachage est que la transformation est **irréversible** dûe aux nombreux `décalages de bits`, `XOR` et autres opérations appliquées sur la chaîne de caractères initiale. Mais, en plus d'être irréversible, le moindre changement dans l'entrée initiale change **complètement** la valeur de sortie.

7.2 Utilités des fonctions de hachage

Maintenant que nous savons ce qu'est une méthode de hachage, intéressons-nous à toutes ses utilités, mais surtout à celles qui nous concernent.

Passons d'abord les utilités possibles qui ne sont pas en relation directe avec notre projet. Parmi celles-ci, on retrouve :

- Le **stockage et l'authentification** de mots de passe,
- La **création de signatures numériques**,
- La **détection de logiciels malveillants**.

À présent, regardons les deux points qui nous concernent dans l'utilisation du hachage dans la blockchain.

7.2.1 Intégrité des données

Un premier point déjà évoqué est l'**intégrité des données**, permettant de repérer les données possiblement corrompues ou falsifiées grâce à une simple **comparaison** entre le *résultat attendu* et le *résultat calculé*.

7.2.2 Preuve de travail

Le second point est la **preuve de travail** (`Proof of Work, PoW`) au sein de la blockchain. Ce mécanisme de consensus [12] permet d'obtenir un **accord sur le réseau de blockchains**, pour **confirmer les transactions** et **produire de nouveaux blocs** sur la chaîne.

7.3 Utilisation du SHA256 dans le Bitcoin

Avant de parler de l'implémentation du `SHA256`, voyons son **utilisation dans le Bitcoin** :

- Vérification de l'**intégrité des transactions** : une transaction ne doit pas pouvoir être modifiée. Grâce au hachage, la modification de n'importe quel caractère entraîne une différence de résultat et donc la détection immédiate de la moindre anomalie.
- Lien entre les blocs dans un **Arbre de Merkle**, ce qui permet de condenser et sécuriser la mémoire.
- Utilisation dans le **processus de preuve de travail**, déjà évoqué.

7.4 Implémentation du SHA256

Tous les points et les définitions ayant été abordés, regardons de quelle manière nous avons implémenté le `SHA256`. Pour ce faire, nous avons fait le choix de **séparer en deux** la transformation de la chaîne de caractères :

7.4.1 Prétraitement des données

La fonction `preprocessing` prend une chaîne de caractères en entrée et la convertit en un **vecteur d'octets**. Elle ajoute ensuite un **bit de fin** (`0x80`) et remplit le vecteur avec des **zéros** jusqu'à ce que sa longueur soit un **multiple de 512 bits moins 64 bits**. Enfin, la **longueur initiale** de la chaîne, exprimée en bits, est ajoutée à la fin du vecteur. Cette étape **prépare les données** pour le traitement en assurant qu'elles respectent le format requis par l'algorithme de hachage `SHA256`.

7.4.2 Traitement des données

La fonction `processing` commence par **initialiser des valeurs de hachage** et des **constantes de rondes** spécifiques à l'algorithme `SHA256`. Puis, elle :

- Découpe les données en **blocs de 64 octets** et prépare un **tableau de 64 entiers de 32 bits** pour chaque bloc.
- Effectue une **série d'opérations bit à bit et arithmétiques** pour mettre à jour les valeurs de hachage, en utilisant des **variables temporaires** calculées à partir des données et des constantes de rondes.
- Après avoir traité tous les blocs, la fonction **combine les valeurs de hachage finales** pour produire une **chaîne hexadécimale** représentant le hachage final.

Cet algorithme renvoie une **sortie fixe de 256 bits**.

8 Chiffrement RSA

Notre réseau utilisera le chiffrement RSA [28]. Il s'agit d'un concept de **cryptographie asymétrique** composé d'une clé privée et une clé publique par utilisateur qui leur permettront de s'échanger des *messages* tout en restant secrets et sécurisés.

8.1 Fonctionnement du RSA

Ainsi, un utilisateur **A** pourra envoyer un *message* en le cryptant avec sa clé publique à un utilisateur **B** qui, pour le lire, devra décrypter ce message avec sa clé privée et vice-versa.

Cela rend impossible, sans connaître la clé privée du receveur, d'intercepter un message qui ne vous est pas destiné. Notre travail est donc de créer/générer ces clés et de faire en sorte que chaque personne utilisant Nexium puisse crypter et décrypter des messages grâce à leur propre clé privée et publique.

8.2 Génération de clés

Tout d'abord, la génération de clé se passe sous la forme d'un **algorithme mathématique** [27] qui forme des clés en utilisant cinq nombres, que nous appellerons **p**, **q**, **n**, **e** et **d**, où :

- **p** et **q** sont deux nombres premiers distincts,
- **n** est leur produit, $n = p \times q$,
- **e** est un nombre premier avec la valeur de l'indicatrice d'Euler en **n**,
- **d** est l'inverse modulaire de **e** $\mod(\varphi(n))$ où $\varphi(n) = (p-1)(q-1)$.

La clé publique sera donc le couple (n, e) tandis que la clé privée est le nombre **d**.

Nous avons rencontré des problèmes lors de nos premières générations de clés. Il s'agissait de problèmes **d'optimisation** liés à la taille en bits, en rapport donc avec les types natifs [23] de Rust qui sont limités en taille et qui deviennent donc insuffisants si nous avons besoin de manipuler des nombres très grands. Or, le standard RSA requiert généralement des nombres de l'ordre de **1024 bits** pour être sécurisé [20].

Ces types de taille fixe en mémoire ont donc posé problème pour la génération. Pour résoudre ce problème, nous avons dû utiliser les crates `num-bigint` et `num-primes`.

Ces **crates** nous permettent de générer de grands entiers sans limite de taille. Par exemple, `num_bigint::BigUint` est une structure dynamique qui s'adapte à la taille du nombre, contrairement aux types natifs de Rust (grâce à l'utilisation de vecteurs `u32` ou `u64` pour limiter la taille en mémoire). Nous avons également utilisé `num_primes` pour générer aléatoirement et manipuler des nombres premiers efficacement.

8.3 Chiffrement et Signature

Nous possédions déjà la génération de clé RSA lors de la précédente soutenance. Grâce à cela, nous avons pu implémenter l'ensemble des fonctions nécessaires pour réaliser tous les principes liés au chiffrement RSA, à savoir : la signature et la vérification de signatures, ainsi que le chiffrement et le déchiffrement. Ces fonctions reposent sur les fondements de RSA (nombres premiers, clés publique et privée, n, d, e) tout en intégrant un hachage cryptographique (SHA-256) afin de garantir l'intégrité des messages.

Signature (`sign`)

La première fonction, `sign`, permet de générer une signature numérique pour un message donné. Elle repose sur le principe selon lequel seul le détenteur de la clé privée doit être capable de produire une signature valide. Voici le déroulé de l'opération :

- Le message est tout d'abord transformé en un condensé (hash) grâce à l'algorithme SHA-256 implémenté plus tôt dans le projet. Ce hachage garantit que la signature porte uniquement sur le contenu du message, indépendamment de sa taille.

- Le condensé est ensuite converti en un entier (`BigUint`) pour être manipulé par les fonctions RSA.

- L'entier est signé en le mettant à la puissance d (exposant privé) modulo n :
$$S = H(m)^d \mod n$$

- La fonction renvoie une erreur si le message est vide ou si la taille du haché dépasse celle du module n.

Cette signature peut ensuite être transmise avec le message pour vérification.

Vérification de signature (`check_signature`)

La fonction `check_signature` permet de s'assurer que la signature reçue correspond bien au message. Elle utilise la clé publique du signataire, ce qui garantit que seule la personne possédant la clé privée a pu produire cette signature.

- Le message reçu est haché de la même manière que dans la fonction de signature.
- La signature est déchiffrée en la mettant à la puissance e (exposant public) modulo n :

$$m' = s^e \mod n$$

- Si le résultat du déchiffrement correspond au hash recalculé, alors la signature est considérée comme valide.

Cela confirme que le message n'a pas été modifié ou falsifié, et qu'il provient bien de l'expéditeur déclaré. Ce mécanisme est au cœur de la validation d'identité dans les protocoles sécurisés.

Chiffrement (`crypt`)

La fonction `crypt` permet de chiffrer un message de manière confidentielle. Ce chiffrement est réalisé à l'aide de la clé publique du destinataire.

- Le message est tout d'abord converti en un entier.
- Il est ensuite chiffré par l'opération RSA classique :

$$c = m^e \mod n$$

- Une vérification est effectuée pour s'assurer que la valeur à chiffrer est inférieure au module n, sinon une erreur est renvoyée.

Ce chiffrement garantit que seul le détenteur de la clé privée pourra déchiffrer le message, assurant ainsi la confidentialité, même en cas d'interception du message.

Déchiffrement (`decrypt`)

Le déchiffrement est l'opération inverse du chiffrement. Il s'effectue avec la clé privée.

- Le message chiffré (déjà sous forme d'entier) est déchiffré par l'opération :

$$m = c^d \mod n$$

- On obtient ainsi le message original, à condition que la clé privée utilisée corresponde à la clé publique ayant servi au chiffrement.
- Comme toujours, si le message est vide ou trop grand, une erreur est renvoyée.

8.4 Format PEM

Après avoir généré notre paire de clés RSA (publique et privée), nous avons souhaité les placer dans un format standard, sécurisé, et reconnu par les outils de chiffrement tels que GnuPG. Pour cela, nous les avons soumises à un processus de sérialisation vers le format OpenPGP, puis encapsulées dans un conteneur ASCII Armor au format PEM.

La transformation de la structure RSA en format OpenPGP suit plusieurs étapes, conformément à la RFC 4880 (format des messages OpenPGP).

Paquet de clé publique

En premier lieu, nous construisons un paquet de clé publique contenant :

- La version du format OpenPGP (4) ;
- Un *timestamp* correspondant à la date de création ;
- L'identifiant de l'algorithme (1 pour RSA) ;
- Les deux entiers principaux de la clé RSA : n et e.

Ces derniers sont encodés au format MPI (*Multi Precision Integer*) :

- Une entête de deux octets encode la longueur de l'entier en bits ;
- Une séquence d'octets représente la valeur de l'entier en *Big Endian*.

L'encodage d'un entier se fait donc en deux parties, permettant une taille réduite comparée à la valeur représentée. Par exemple, e (souvent égal à 65537) possède 17 bits significatifs et est représenté sur 5 octets (2 pour la taille, 3 pour la valeur).

Le format MPI est utilisé dans tous les paquets contenant des entiers cryptographiques dans le protocole OpenPGP (paquets de clé publique, clé privée, signature, sous-clés, ou messages chiffrés).

Paquet de clé privée

Le paquet de clé privée suit le même principe que celui de la clé publique, avec en plus les autres entiers RSA (d, p, q, et u). Ces valeurs sont assemblées, puis hachées avec SHA-1 pour vérification. Le tout est ensuite chiffré avec AES-128 en mode CFB, à l'aide d'un mot de passe fourni par l'utilisateur.

Ce chiffrement utilise un mécanisme appelé S2K (*String to Key*) qui dérive une clé AES à partir :

- du mot de passe ;
- d'un *salt* de 8 octets ;
- d'un nombre d'itérations ;
- d'un vecteur d'initialisation aléatoire.

L'ensemble du bloc chiffré est ensuite inclus dans le paquet avec les métadonnées nécessaires. Ce paquet possède un tag différent de celui de la clé publique.

Paquet d'identifiant utilisateur

Un paquet d'identifiant utilisateur est également présent, sous forme de chaîne UTF-8 au format :

`prénom.nom <email@epita>`

Paquet de signature

Il existe aussi un paquet de signature dont le rôle est de lier cryptographiquement la clé publique à un identifiant utilisateur. Il est composé :

- d'un en-tête ;
- de sous-paquets hachés et non hachés ;
- d'une empreinte SHA-256 ;
- d'une structure ASN.1 suivie d'une signature RSA faite avec la clé privée.

Comme pour les autres entiers cryptographiques, cette signature est encodée au format MPI.

Encapsulation PEM

Une fois tous les paquets assemblés pour la clé publique (ou la clé privée), ils sont :

- concaténés en binaire ;
- encodés en base64 ;
- complétés par un `CRC24` (contrôle d'intégrité sur 3 octets).

Enfin, le tout est entouré d'une balise ASCII, différente selon qu'il s'agit d'une clé publique ou d'une clé privée.

Ce format est lisible par les outils OpenPGP, tout en étant transportable sous forme de texte. Le processus complet, conforme à la RFC 4880, assure aux clés une compatibilité avec les outils tels que GPG.

Vous trouverez ci-dessous un exemple de clé RSA transformée par ce processus de formatage PEM :

public_key.pem

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xsBNBGfyzYYBB/sGc+nlGYhmO6KVxZ8ii+ajK+HQ5jD0J9U+ZXzM3f02FBkMFzoj
rZYvPtwV7UJ2QeF255SMCZJAYfPexdfznLn5MbbPgb7KxMdfQ1bnYFnjOTU3FzZT
7lWx1klP5er4Qht7OvuMs2KdfoJLBGm9AyCwh2setZ9aoKJq0zqit1Q4ofC+r6Xb
m3GmFy3+H6xOn5+591KGfnfxTHwCKEgbFnPKPHHb68eD9+z+eDzgHSo5KxINlLfH
shZFWVelP2x4nKi5awElO5ARH2TgZfOXLAaWfgyXg364fhZrjryovrOLOBN5r4IR
UQHZNAA1AFnghD5T4UK4fqQibfTiWNqryZqLABEBAAHNImplYW4uaGVyYWlsIDxq
ZWFuLmhlcmFpbEBlcGl0YS5mcj7CwFwEEwEIABAFAmfyzYYJEPzyLMmr8iq8AABI
ugf5AY+c/ECk/A2YonfSfFO7hTVP7Jbhv0FgwovVIy5BwZrILBea4dQ4spq/ovd2
qItm1/tQdmj17ncoduKBQsLcnr+GDMA6GzgOi6wadGD2i3InfxTqRhftxj4d3auk
bcujcTzMVKC59x64PVRR2ed4pLeLaTu1gZJQLPdUWra1ZGPEKUini8bK6HVktr93
aRxx59giJHEh6QIdT6l1x9aS1HufEkjY9ZYylb42ERr0tORrvPqPqfNI8xsAUMak
OjdkKHI7RldUCPfwiLk9DDuQRz+yG7LrvI+eNzfe2x9cIEl2klcgBu1cBg8UCzyQ
GImS1ic0Je1gXGOnaPhZalb0kA==
=XWDw
-----END PGP PUBLIC KEY BLOCK-----
```

9 Site Web

Astro permet une approche simple et rapide pour la création d'une documentation web.

Utilisant `Vite`, Astro offre une rapidité de développement accrue avec le **SSR** (*Server Side Rendering*), permettant une actualisation dynamique des pages à chaque modification.

Prenant en charge `Markdown`, il nous est possible d'écrire la documentation en respectant la syntaxe Markdown, puis de laisser Astro faire le *build*, générant les fichiers `html`, `css` et `js` nécessaires pour les navigateurs web.

La dernière étape étant de servir les fichiers générés, nous utilisons **GitLab Pages** pour l'hébergement. Nous avons également configuré sur GitLab un déploiement automatique qui met à jour la documentation à chaque nouveau commit.

Le **site web officiel de Nexium** est disponible à l'adresse https://jean.herail.pages.epita.fr/nexium

Afin d'illustrer la **facilité d'utilisation** et de développement propre à `Astro`, voici un exemple de code `Markdown` suffisant à la génération d'une page Web:

web/contents/blockchain.md

```
---
title: La blockchain Nexium
description: Architecture des blocs.
---

# L'architecture de la blockchain Nexium
## Stockage de la blockchain
Nous stockons la **blockchain** dans un fichier binaire ‘
   nxm_blckchn.dat‘.
```

Astro se charge de convertir ce code `Markdown` en `HTML` statique, prêt à être servi par un serveur web. C'est ensuite **GitLab Pages** [10] qui se charge du **build**, et de l'**exposition** automatique du site à chaque nouveau `git push`.

10 Gitlab

Dans le cadre de notre projet, nous avons développé une API Gitlab en Rust, afin d'interfacer facilement notre application avec l'instance Gitlab d'Épita (gitlab.cri.epita.fr). Cette API maison encapsule plusieurs fonctionnalités essentielles, telles que :

- l'authentification OAuth2 (via le flux d'autorisation PKCE),
- la vérification de la validité d'un jeton d'accès,
- la récupération des identifiants utilisateurs et de leurs clés GPG,
- l'ajout d'une clé GPG à un compte utilisateur.

Ce module est principalement utilisé pour interagir avec les utilisateurs via leurs identifiants EPITA, ce qui permet de garantir la sécurité et la traçabilité des actions au sein du réseau Nexium.

10.1 Authentification OAuth2

Pour les clients, l'authentification se fait via un navigateur web ouvert localement. Une fois l'utilisateur connecté, un petit serveur HTTP local récupère le `code` d'autorisation retourné par Gitlab, puis échange ce code contre un *token* d'accès API. Cette opération est encapsulée dans la méthode suivante :

Listing 1: Obtention d'un token Gitlab via OAuth2

```
pub fn get_token() -> Result<String, GitlabError>
```

10.2 Structure principale

La structure `GitlabClient` centralise l'état de la connexion :

```
pub struct GitlabClient {
    api_url: String,
    token: String,
}
```

Elle est instanciée à l'aide d'un jeton d'accès :

```
pub fn new(token: String) -> Self
```

10.3 Fonctions principales

Voici un aperçu des principales fonctionnalités de l'API :

- **Vérification de la validité d'un token** :

```
    pub fn check_token(&self) -> Result<bool,
       GitlabError>
```

Cette fonction vérifie que le token fourni donne bien accès aux informations de l'utilisateur.

- **Récupération des clés GPG d'un utilisateur** :

```
    pub fn get_gpg_keys(&self, login: &str) -> Result<
       Vec<String>, GitlabError>
```

Elle récupère les clés GPG associées au compte Gitlab d'un utilisateur donné.

- **Ajout d'une clé GPG** :

```
    pub fn add_gpg_key(&self, gpg_key: &str) -> Result
       <(), GitlabError>
```

Cette fonction permet à l'utilisateur d'ajouter une nouvelle clé GPG à son compte.

10.4 Erreurs possibles

Nous avons défini une énumération `GitlabError` pour capturer proprement les erreurs liées à l'authentification ou à la communication avec l'API :

```
pub enum GitlabError {
    InvalidToken,
    NetworkError,
    UserNotFound,
    UnknownError,
    NoGPGKeys,
    BadGPGFormat,
    NoWebBrowser,
}
```

10.5 Utilisation dans le projet

L'API est utilisée pour authentifier les utilisateurs de Nexium avec leur compte Gitlab EPITA, récupérer leur clé GPG et ainsi leur permettre de signer les transactions ou blocs avec une preuve d'identité vérifiable.

Cette intégration permet un couplage fort entre l'identité réelle d'un utilisateur et son identité sur la blockchain Nexium, renforçant la sécurité et la confiance du système.

11 Gestion de la configuration serveur

Le serveur Nexium nécessite plusieurs informations de configuration au démarrage, telles que les chemins vers la base de données ou la clé privée, les informations Gitlab de l'utilisateur, ainsi que le port d'écoute. Ces informations sont regroupées dans une structure `Config`, définie dans le module `config.rs`. Ce module permet de générer un fichier de configuration, de le lire depuis un fichier JSON, ou encore de sauvegarder les paramètres courants.

11.1 Structure de configuration

Les différents paramètres sont encapsulés dans la structure suivante :

Listing 2: Structure de configuration

```
pub struct Config {
    pub database_filepath: String,
    pub key_filepath: String,
    pub port: u16,
    pub user_login: String,
    pub gitlab_token: String,
    pub gitlab_api_url: String,
}
```

11.2 Génération interactive

La configuration peut être générée de manière interactive à l'aide de la méthode `generate`, qui demande à l'utilisateur les différentes informations via l'entrée standard. Des valeurs par défaut sont proposées, et certains champs (comme le login) sont validés :

```
pub fn generate(path: &Path) -> Config
```

Les valeurs par défaut sont les suivantes :

- Port : `4242`
- Fichier base de données : `blockchain.db`
- Clé privée : `private-key.pem`
- URL API Gitlab : `https://gitlab.cri.epita.fr/api/v4`

Le `token Gitlab` peut également être lu depuis la variable d'environnement `GITLAB_TOKEN`, si elle est présente.

11.3 Lecture et sauvegarde

Une fois générée, la configuration peut être sauvegardée sous forme de fichier JSON lisible :

```
pub fn to_file(&self, path: &Path)
```

À l'inverse, un serveur peut être initialisé à partir d'un fichier JSON existant :

```
pub fn from_file(path: &Path) -> Config
```

Le fichier doit contenir les champs suivants :

```
{
  "database": "blockchain.db",
  "key": "private-key.pem",
  "port": 4242,
  "user_id": "prenom.nom",
  "gitlab_token": "XYZ",
  "gitlab_api_url": "https://gitlab.cri.epita.fr/api/v4"
}
```

11.4 Validation et ergonomie

Pour améliorer l'ergonomie, la méthode `ask` permet de poser des questions à l'utilisateur avec une gestion simple des retours clavier. Un validateur de login simple est aussi inclus :

```
fn _check_login_syntax(login: String) -> bool
```

Ce système permet de garantir que l'identifiant utilisateur respecte bien le format `prenom.nom` imposé par l'infrastructure Gitlab de l'Épita.

11.5 Intégration dans le projet

Cette gestion centralisée de la configuration rend le serveur Nexium facile à déployer, portable, et cohérent avec le reste du projet. Elle permet notamment d'assurer que chaque instance est bien connectée à une identité Gitlab valide, et qu'elle dispose de toutes les ressources nécessaires à son fonctionnement dès son démarrage.

12 Conclusion

Nexium nous permet d'explorer des concepts cryptographiques avancés, tel que la **blockchain**, l'**architecture réseau** et le **développement web**.

En appliquant des principes de sécurité éprouvés comme le **SHA-256** pour le hachage et le chiffrement **RSA** pour l'authentification, nous avons pu établir les bases d'un système robuste et fiable que nous comptons concrétiser le plus fidèlement dans les prochaines étapes du développement.

Nous avons mis en place au fil de cette première étape du projet les fondations essentielles de la **structure théorique** et de la **cryptographie**, en s'inspirant du Bitcoin et en l'adaptant aux besoins spécifiques dans notre cas d'usage.

Le **site web** et la **documentation** permettront une transparence et une accessibilité accrues pour nos travaux.

Nexium, au-delà d'être un exercice technique, est une expérimentation réelle des enjeux modernes liés à la **décentralisation** et à la **souveraineté numérique**, autant de thématiques qui nous paraissent importantes et que nous approfondissons en nous questionnant sur l'impact des cryptomonnaies dans notre société.

Les prochaines étapes du projet seront tout aussi importantes : **optimisation des performances**, **sécurisation structurelle du réseau** et **interface utilisateur intuitive** sont dans notre ligne de mire.

Bibliographie

[1] EITCA Academy. *Comment fonctionne l'algorithme de signature numérique RSA et quels sont les principes mathématiques qui garantissent sa sécurité et sa fiabilité ? - Académie EITCA*. fr. June 2024. URL: https://fr.eitca.org/la-cyber-s%C3%A9curit%C3%A9/eitc-est-une-cryptographie-classique-avanc%C3%A9e/signatures-num%C3%A9riques/signatures-num%C3%A9riques-et-services-de-s%C3%A9curit%C3%A9/examen-examen-signatures-num%C3%A9riques-et-services-de-s%C3%A9curit%C3%A9/comment-fonctionne-l%27algorithme-de-signature-num%C3%A9rique-rsa-et-quels-sont-les-principes-math%C3%A9matiques-qui-garantissent-sa-s%C3%A9curit%C3%A9-et-sa-fiabilit%C3%A9/ (visited on 04/06/2025).

[2] Bitcoin. *Bitcoin Whitepaper*. URL: https://bitcoin.org/bitcoin.pdf.

[3] Bertrand Blancheton. *Les improvisations financière de la guerre de 1914-1918 en France. Les enjeux de la liquidité*. Tech. rep. Groupe de Recherche en Economie Théorique et Appliquée (GREThA), 2014.

[4] Anes Bukhdir. *Le rôle du hachage dans la technologie de la blockchain*. Consulté le 23 février 2025. 2024. URL: https://www.morpher.com/fr/blog/hashing-in-blockchain.

[5] *Chiffrement RSA*. fr. Page Version ID: 220851654. Dec. 2024. URL: https://fr.wikipedia.org/w/index.php?title=Chiffrement_RSA&oldid=220851654 (visited on 04/06/2025).

[6] Tahiti Cryptomonnaies. *Les petits secrets de Bitcoin : Le SHA-256 expliqué*. Consulté le 23 février 2025. 2024. URL: https://tahiti-cryptomonnaies.com/actualites/les-petits-secrets-de-bitcoin-le-sha-256-explique/.

[7] Hal Finney et al. *OpenPGP Message Format*. Request for Comments RFC 4880. Internet Engineering Task Force, Nov. 2007. URL: https://datatracker.ietf.org/doc/rfc4880/ (visited on 04/06/2025).

[8] RA Freeman et al. "Decentralized estimation and control of graph connectivity in mobile sensor networks". In: *2008 American Control Conference, Seattle, WA*. 2008, pp. 2678–2683.

[9] Gitlab. *Gitlab GPG*. URL: https://docs.gitlab.com/api/user_keys/.

[10] Gitlab. *Gitlab Pages*. URL: https://docs.gitlab.com/user/project/pages/.

[11] Craig Jarvis. "Cypherpunk ideology: objectives, profiles, and influences (1992–1998)". In: *Internet Histories* 6.3 (2022), pp. 315–342.

[12] Ledger. *Qu'est-ce que la preuve de travail ?* Consulté le 23 février 2025. 2019. URL: https://www.ledger.com/fr/academy/quest-ce-que-la-preuve-de-travail.

[13] Lawrence Lessig. "Code is law". In: *Harvard magazine* 1.2000 (2000).

[14] Commission Nationale de l'Informatique et des Libertés. *CNIL - Blockchain*. URL: https://www.cnil.fr/fr/definition/blockchain.

[15] Fotios V Mitsakis. "The impact of economic crisis in Greece: key facts and an overview of the banking sector". In: *Business and Economic Research* 4.1 (2014), pp. 250–267.

[16] Nexium. *Site officiel*. URL: https://jean.herail.pages.epita.fr/nexium.

[17] Burgi Noëlle. *La grande régression. La Grèce et l'avenir de l'Europe*. Bord de l'eau (Le), 2014.

[18] *OpenPGP.PublicKeyPacket — OpenPGP v0.6.1*. URL: https://hexdocs.pm/open_pgp/OpenPGP.PublicKeyPacket.html (visited on 04/06/2025).

[19] Ling Ren. "Analysis of nakamoto consensus". In: *Cryptology ePrint Archive* (2019).

[20] MJB Robshaw. "Security Estimates for 512-bit RSA". In: *WESCON CONFERENCE RECORD*. WESTERN PERIODICALS COMPANY. 1995, pp. 409–412.

[21] *RSA et signatures numériques – StackLima*. fr-FR. July 2022. URL: https://stacklima.com/rsa-et-signatures-numeriques/ (visited on 04/06/2025).

[22] Rust. *La documentation Rust*. URL: https://prev.rust-lang.org/fr-FR/documentation.html.

[23] Rust. *Le langage de programmation Rust - Les types de données*. URL: https://jimskapt.github.io/rust-book-fr/ch03-02-data-types.html.
[24] Rust. *Rust Programming Language*. URL: https://www.rust-lang.org/.
[25] *Schéma de signature numérique RSA utilisant Python – StackLima*. fr-FR. July 2022. URL: https://stacklima.com/schema-de-signature-numerique-rsa-utilisant-python/ (visited on 04/06/2025).
[26] Grégory Soutadé. *Bitcoin en interne*. Aug. 20, 2015. URL: https://blog.soutade.fr/post/2015/08/bitcoin-en-interne.html.
[27] Webia. *Cours 5 : Algorithmes*. Consulté le 23 février 2025. 2015. URL: https://webia.lip6.fr/~spanjaard/cours/Algo_cours5.pdf.
[28] Wikipédia. *Chiffrement RSA*. Consulté le 23 février 2025. 2024. URL: https://fr.wikipedia.org/wiki/Chiffrement_RSA.
[29] Wikipedia. "Strongly connected component". In: URL: https://en.wikipedia.org/wiki/Strongly_connected_component.
[30] Hui Xu et al. "Memory-safety challenge considered solved? An in-depth study with all Rust CVEs". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31.1 (2021), pp. 1–25.
[31] Erica York. "Trump Tariffs: The Economic Impact of the Trump Trade War". In: (2025).