

Rapport de stage d'application - Thales

Vincent CAUJOLLE

Résumé

Ce document présente mon stage d'application (stage ayant eu lieu en fin de deuxième année du cursus ingénieur généraliste de l'École Centrale de Lyon) dans le groupe Thales. La première partie de ce document est consacrée à la présentation du groupe et de son fonctionnement. Ensuite, on aborde le travail que j'ai réalisé au cours des six mois de stage en détaillant les différentes missions qui m'ont été confiées. Enfin, on trouve en fin de rapport un bref passage sur l'impact de ce stage sur la construction de mon projet professionnel (cette partie peut être absente si vous n'êtes pas de l'ECL).

Abstract

This document presents my application internship (internship that took place at the end of the second year of the general engineering curriculum at École Centrale de Lyon) within the Thales group. The first part of this document is dedicated to the presentation of the group and its operation. Then, we discuss the work I carried out during the six months of internship, detailing the various missions given to me. Finally, a brief passage on the impact of this internship on the construction of my professional project can be found at the end of the report (this part might not be present if you are not from the ECL).

Table des matières

| | | |
|----------|--|-----------|
| 1 | Introduction | 6 |
| 2 | Contexte du stage et organisation de l'entreprise | 6 |
| 2.1 | Groupe Thales | 6 |
| 2.2 | Thales Service Numérique (TSN) | 7 |
| 2.3 | Direction Technique (DT) | 7 |
| 2.4 | Le positionnement du stage dans les travaux de l'entreprise | 8 |
| 3 | Travaux réalisé | 9 |
| 3.1 | AES | 9 |
| 3.1.1 | Permutation Π | 10 |
| 3.1.2 | Permutation $\hat{\Pi}$ | 11 |
| 3.1.3 | Création des clefs k_i | 12 |
| 3.2 | AES CBC | 12 |
| 3.2.1 | Mon implémentation | 14 |
| 3.2.2 | Padding oracle attack | 14 |
| 3.3 | AES GCM | 17 |
| 3.3.1 | CTR | 17 |
| 3.3.2 | GMAC | 17 |
| 3.3.3 | GCM | 18 |
| 3.4 | Mise en place d'un programme de chiffrement/déchiffrement | 19 |
| 3.4.1 | Contexte | 19 |
| 3.4.2 | Mise en place | 20 |
| 3.5 | SoftHSM | 20 |
| 3.6 | Signature | 23 |
| 3.6.1 | RSA | 24 |
| 3.6.2 | Courbes elliptiques avec Ed25519 | 25 |
| 3.6.3 | Serveur signature | 29 |
| 3.7 | Wireguard | 29 |
| 3.7.1 | Configuration | 29 |
| 3.7.2 | Mise en place | 31 |
| 3.8 | Signal | 31 |
| 3.9 | Port knocking | 31 |
| 3.10 | Algo post quantique | 31 |
| 4 | Le stage d'application dans la construction de mon projet professionnel | 31 |
| 5 | Conclusion | 31 |

Table des figures

| | | |
|----|--|----|
| 1 | Chiffres clés de 2023 du groupe Thales | 6 |
| 2 | Axes majeurs du développement à Thales | 6 |
| 3 | fonctionnement d'AES | 9 |
| 4 | Réorganisation du bloc | 10 |
| 5 | Effet de SubBytes sur le bloc, avec $\tilde{s} = S(s)$ | 10 |
| 6 | Effet de ShiftRows sur le bloc | 11 |
| 7 | réorganisation des clefs en suite de mots de 32 bits | 12 |
| 8 | Fonctionnement d'AES ECB, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k | 12 |
| 9 | Fonctionnement d'AES CBC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k | 13 |
| 10 | Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible) . | 15 |
| 11 | Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible) . | 16 |
| 12 | Déchiffrement d'AES CBC avec en rouge ce qui ne nous est pas accessible | 17 |
| 13 | Fonctionnement d'AES CTR, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , $Le +1$ signifie qu'on incrémente de 1 l'IV généré aléatoirement. | 18 |
| 14 | Fonctionnement du GMAC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , et $\times H$ la multiplication dans $GF(2^{128})$ par $H = AES_k(0^{128})$ | 18 |
| 15 | Fonctionnement d'AES GCM avec en vert CTR et en rouge GMAC | 19 |
| 16 | HSM de Thales | 21 |
| 17 | Courbe $y^2 = x^3 - x + 9$ sur \mathbb{R} | 25 |
| 18 | $P + Q$ sur $y^2 = x^3 - x + 9$ sur \mathbb{R} avec $P = (-1 - 3)$ et $Q = (1, 3)$. . | 26 |
| 19 | Courbe $y^2 = x^3 - 6x + 2$ sur $\mathbb{Z}/59\mathbb{Z}$ | 27 |

Liste des tableaux

| | | |
|---|----------------------------------|---|
| 1 | table de vérité du XOR | 5 |
| 2 | Variantes d'AES | 9 |

Glosaire

1. GBU : Une Global Business Unit est une division du groupe Thales se concentrant sur un marché spécifique
2. XOR (\oplus) :

| a | b | $a \oplus b$ |
|-----|-----|--------------|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

TABLE 1 – table de vérité du XOR

3. byte : 8 bits

1 Introduction

2 Contexte du stage et organisation de l'entreprise

Dans le cadre de ma deuxième année du cycle ingénieur de l'ECL, j'ai pu faire mon stage d'application du 27/05/24 au 27/11/24 à Thales.

2.1 Groupe Thales

« *Construisons ensemble un avenir de confiance* » est la devise de Thales. Créé en 1893, Thales, sous la direction de Patrice Caine est présent dans 68 pays, compte près de 81000 collaborateurs et affiche un chiffre d'affaire de plus de 16 milliards d'euros.



FIGURE 1 – Chiffres clés de 2023 du groupe Thales

Les clients de Thales sont de grandes organisations - États, administrations, entreprises, etc. Ses clients sont responsables des opérations, des services et des infrastructures critiques de la société telles que : la défense, la sécurité, le transport aérien et ferroviaire, la banque, les télécommunications et bien d'autres. Pour gagner la confiance de ses clients Thales opère selon 3 points clés



FIGURE 2 – Axes majeurs du développement à Thales

Au sein de Thales existent 7 GBUs :

- Avionics (AVS)
- Digital Identity & Security (DIS)
- Secure Communications & Information Systems (SIX)
- Defence Mission Systems (DMS)
- Ground Transportation Systems (GTS)
- Land & Air Systems (LAS)
- Thales Alenia Space (TAS)

Mon stage se déroulait au sein de Thales Services Numériques dans la [GBU SIX](#).

2.2 Thales Service Numérique (TSN)

Thales Services Numériques (TSN), dirigé par Walter Cappilati, est l'ESN du groupe Thales et a pour mission d'assurer la performance, la résilience et la sécurité des systèmes d'information de ses clients. TSN compte plus de 4 300 collaborateurs et propose des solutions pour de nombreux marchés : défense, transport, finances, aéronautique...

TSN travail à 30% pour le groupe et à 70% chez des clients hors Thales. TSN s'est fixé les objectifs suivants :

- Être une entreprise de services numériques faisant preuve d'une croissance rentable plus rapide que le marché, et être reconnue comme une référence dans ses secteurs clés.
- Accentuer l'innovation, la capacité à accompagner la transformation numérique des clients, la maîtrise du développement de solutions sur mesure, la gestion des systèmes d'information les plus critiques et la cybersécurité.
- Être un partenaire de confiance au sein du groupe Thales et auprès des clients privés et publics les plus exigeants, leur garantir la maîtrise d'un bout à l'autre des technologies numériques, au service de leurs enjeux de sécurité et de performance économique et opérationnelle.

Au sein de TSN existent 4 directions d'excellence technique : la direction du conseil numérique sécurisé (DCNS), la direction de l'ingénierie logicielle (DIL), la direction de l'ingénierie IT outsourcing (DIO) et la direction technique (DT). Ma place était au sein de la direction technique, entouré d'architectes sécurité

2.3 Direction Technique (DT)

La fonction première de la Direction Technique consiste à garantir l'alignement technique nécessaire pour répondre aux enjeux de compétitivité des solutions, de bonne exécution technique des projets et de l'approche globale de la cybersécurité

sur le périmètre de Thales Services Numériques La direction technique assure ses fonctions centrées sur le développement et l'animation de l'expertise technique, du domaine de l'infrastructure à celui du développement d'applications logicielles, aussi bien dans les phases de construction que d'opération. Pour assurer ses responsabilités, la direction technique s'appuie sur :

- Une politique technique qui concerne les méthodes, les technologies et les outils à utiliser sur les projets. Elle définit le cadre d'application. La direction technique est donc garante du maintien de la politique technique.
- Une direction de l'innovation qui se charge de développer la partie recherche et développement nommée « Self-Funding Research & Development » (SFRD).
- Une filière expertise comptant plus de 300 personnes. Cette filière a pour objectif d'intervenir en support stratégique et technique pour des équipes projets. Elle est composée d'architectes et experts, de « Strategic Technical Authority » (STA), de « Project Design Authority » (PDA) et de « Design Authority » (DA).
- Une forte synergie avec les autres directions pour identifier les partenariats techniques afin d'assurer la pérennité des solutions déployées.

2.4 Le positionnement du stage dans les travaux de l'entreprise

L'équipe de sécurité de la Direction technique de Thales Services Numérique a pour objectif de garantir des solutions de haute sécurité pour les systèmes fournis en interne ou en externe (qu'ils soient à destination d'administrations Françaises ou de grandes entreprises). Dans ce contexte, un thème d'importance capitale est la protection des données, et notamment la confidentialité et le respect de la vie privée (RGPD).

Takeyna est une suite de composants développés par Thales Services Numériques pour fournir des solutions de chiffrement de données et de signature électronique. Ces composants permettent de « cacher » la complexité des mécanismes cryptographiques en fournissant des interfaces de haut niveau, qui peuvent être facilement intégrées sans nécessiter une expertise cryptographique (qui reste une compétence rare), et qui évite des erreurs que ce soit dans le choix des mécanismes ou dans leur utilisation et implémentation.

Takeyna a été initialement développé en Java, mais mon Tuteur, Mahmoud Chilali, souhaite porter cette application en Rust au vu des performances et du futur prometteur de ce relativement nouveau langage. Ma mission sera donc d'expérimenter avec ce nouveau langage pour fournir des applications pouvant servir de preuve de fonctionnement et simplifier la migration de Takeyna.

3 Travaux réalisé

3.1 AES

Toutes les informations qui m'ont permis d'écrire cette sous section sont directement tirées de [1].

Face au manque de standard de chiffrement par bloc, NIST (National Institute of Standards and Technology) a lancé un concours en 1997 pour créer un nouveau standard de chiffrement : AES (Advanced Encryption Standard). De là naît un nouvel algorithme inspiré de la proposition des cryptographes belge Joan Daemen et Vincent Rijmen. Il permet de chiffrer des blocs de 128 bits et existe sous trois variantes :

| nom | taille de la clef (bits) | taille des blocs (bits) | nombre de rounds |
|---------|--------------------------|-------------------------|------------------|
| AES 128 | 128 | 128 | 10 |
| AES 192 | 192 | 128 | 12 |
| AES 256 | 256 | 128 | 14 |

TABLE 2 – Variantes d'AES

A partir de maintenant, pour fluidifier la lecture, AES 128 = AES)

Remarque : à l'origine cet algorithme pouvait supporter des blocs de 128, 192 et 256 bits. Cette fonctionnalité n'a malheureusement pas été conservé par la NIST. Cette fonction aurait permis à AES d'être post quantique.

AES permute successivement son entrée (un bloc de 128 bits) avec une même permutation Π (sauf au dernier round ou on utilise $\hat{\Pi}$). Cette permutation est inversible, ce qui permet de déchiffrer la sortie en remontant le processus d'AES.

On XOR la sortie de chaque round avec une clef k_i (calculé à partir de la clef secrète de 128 bits). L'avantage du XOR est que $\text{XOR}^2 = \text{id}$. Donc il suffit de réutiliser XOR pour remonter AES.

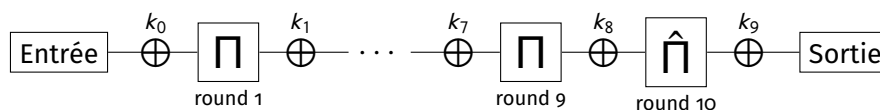


FIGURE 3 – fonctionnement d'AES

3.1.1 Permutation Π

Cette permutation est propre à AES (ne dépend pas de la clef secrète et peut être calculée à l'avance. Cela permet d'avoir un processus de chiffrement très efficace). Elle est le résultat de la composition de 3 sous-permutations (toutes inversible). Pour mieux comprendre comment ces 3 sous-permutations marchent, il faut réorganiser le bloc de 128 bits en une matrice de taille 4×4 où chaque cellule contient un **byte**.

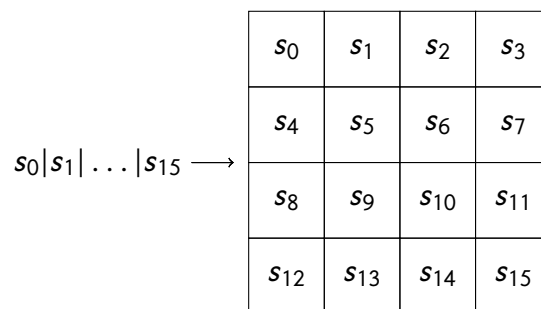


FIGURE 4 – Réorganisation du bloc

1. SubBytes :

On applique à chaque cellule du bloc une permutation $S : \{0, 1\}^8 \rightarrow \{0, 1\}^8$ (d'un **byte** vers un autre).

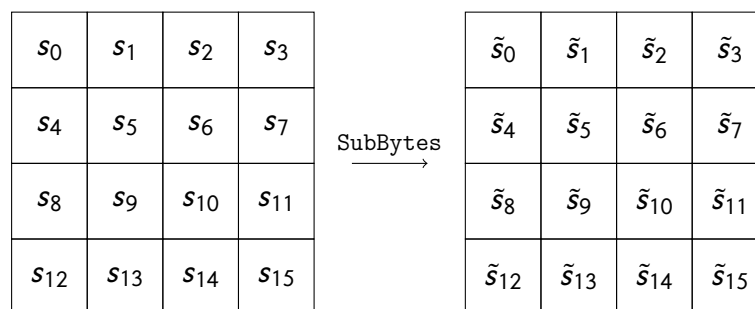


FIGURE 5 – Effet de SubBytes sur le bloc, avec $\tilde{s} = S(s)$

RAJOUTER DEF S

2. ShiftRows :

Cette permutation va pour chaque colonne déplacer de manière cyclique ses éléments de telle manière que la colonne i subira le cycle

$$(0 \quad (1+i)\%4 \quad (2+i)\%4 \quad (3+i)\%4)$$

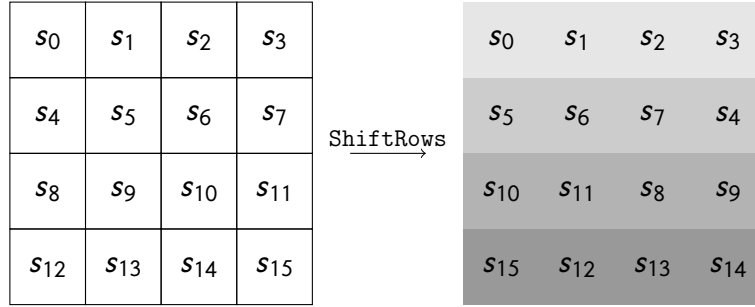


FIGURE 6 – Effet de ShiftRows sur le bloc

3. MixColumns :

Pour cette permutation, on calcule dans $GF(2^8)$ (muni du polynome irréductible $x^8 + x^4 + x^3 + x + 1$ ie 100011011) le produit matriciel (à gauche) de notre bloc par :

$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix}$$

avec les éléments de cette matrices à comprendre comme des éléments de $GF(2^8)$

Donc, en résumé :

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{bmatrix} \xrightarrow{\Pi} \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} \tilde{s}_0 & \tilde{s}_1 & \tilde{s}_2 & \tilde{s}_3 \\ \tilde{s}_5 & \tilde{s}_6 & \tilde{s}_7 & \tilde{s}_4 \\ \tilde{s}_{10} & \tilde{s}_{11} & \tilde{s}_8 & \tilde{s}_9 \\ \tilde{s}_{15} & \tilde{s}_{12} & \tilde{s}_{13} & \tilde{s}_{14} \end{bmatrix}$$

3.1.2 Permutation $\hat{\Pi}$

Généralement on préfère utiliser $\hat{\Pi}$ au lieu de Π au dernier round pour avoir un algorithme de déchiffrement quasiment identique que celui de chiffrement. Avec $\hat{\Pi}$ définie tel que :

$$\begin{bmatrix} s_0 & s_1 & s_2 & s_3 \\ s_4 & s_5 & s_6 & s_7 \\ s_8 & s_9 & s_{10} & s_{11} \\ s_{12} & s_{13} & s_{14} & s_{15} \end{bmatrix} \xrightarrow{\hat{\Pi}} \begin{bmatrix} \tilde{s}_0 & \tilde{s}_1 & \tilde{s}_2 & \tilde{s}_3 \\ \tilde{s}_5 & \tilde{s}_6 & \tilde{s}_7 & \tilde{s}_4 \\ \tilde{s}_{10} & \tilde{s}_{11} & \tilde{s}_8 & \tilde{s}_9 \\ \tilde{s}_{15} & \tilde{s}_{12} & \tilde{s}_{13} & \tilde{s}_{14} \end{bmatrix}$$

C'est exactement Π mais sans la permutation MixColumns.

3.1.3 Création des clefs k_i

A partir d'une clef secrète k (de 128 bits) il faut créer une série de clefs $k_0 \dots k_{10}$. Pour ça on sépare cette clef en 4 mots de 32 bits (4 [byte](#)) chacun.

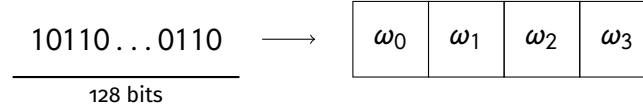


FIGURE 7 – réorganisation des clefs en suite de mots de 32 bits

On définit la première clef $k_0 = \boxed{\omega_{0,0}|\omega_{0,1}|\omega_{0,2}|\omega_{0,3}} = k$ (ie k_0 est égale à la clef secrète). Ensuite, on calcule $k_i = \boxed{\omega_{i,0}|\omega_{i,1}|\omega_{i,2}|\omega_{i,3}}$ en fonction de k_{i-1} tel que :

$$\forall i \in \{1, 2, 3\} \quad \begin{cases} \omega_{i,0} &= \omega_{i-1,0} \oplus g_i(\omega_{i-1,3}) \\ \omega_{i,j} &= \omega_{i-1,j} \oplus \omega_{i,j-1} \quad \forall j \in \{1, 2, 3\} \end{cases}$$

Avec $g : \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}$ une fonction tirée des standards d'AES.

3.2 AES CBC

Toutes les informations qui m'ont permis d'écrire cette sous section sont directement tirées de [3].

A lui seul, AES ne permet de chiffrer que des blocs de 128 bits. On pourrait se dire qu'il suffit d'appliquer AES sur l'ensemble des blocs (c'est le mode ECB).

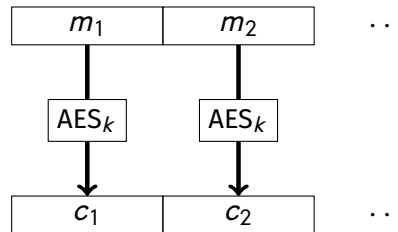


FIGURE 8 – Fonctionnement d'AES ECB, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

Mais, si on utilise la même clef sur plusieurs blocs, le chiffré deviens est très facile à déchiffrer. Il faudrait donc autant de clefs que de blocs pour assurer un chiffrement sécurisé. Cela doublerai la taille du message chiffré et n'est donc pas utilisé.

Il faut donc rajouter une couche supplémentaire pour pouvoir utiliser AES afin de chiffrer ce que l'on veut. On s'intéresse ici au mode CBC (Cipher Block Chaining). Son fonctionnement est décrit sur la figure 9

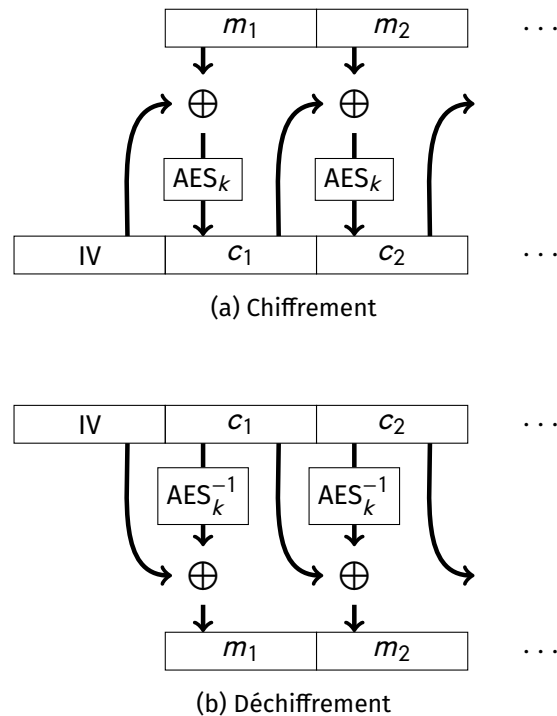


FIGURE 9 – Fonctionnement d'AES CBC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

On remarque que le premier bloc du chiffré est un Initialization Vector (IV). C'est un bloc choisit aléatoirement. Ensuite, un deuxième point important de ce mode d'utilisation d'AES (ou de tout autre méthode de chiffrement par bloc) est qu'il faut que la taille du message en clair soit exactement un multiple de 128 bits. La solution est de rajouter un padding. C'est à dire que l'on complète le message pour que sa taille soit exactement égale au prochain multiple de 128 bits (si le message est déjà un multiple de 128 bits on rajoute 16 **byte** de padding). Il peut prendre plein de forme différente, on peut compléter avec des 0, des bits aléatoires... La seule règle importante est que le dernier **byte** du bloc doit contenir le nombre de **byte** de padding qui ont été rajouté (entre 0 et 15).

Pour ma part mon padding suivra la règle suivante : tous les **bytes** du padding ont la même valeurs (donc celle du dernier **byte**). Par exemple s'il manque 3 **bytes** à mon message pour être un multiple de 16 **byte** (i.e 128 bites) alors je rajoute à la fin du message `02|02|02`. Si c'est déjà un multiple de 16 **byte** je rajoute à la fin

Déchiffrer un bloc

En reprenant la figure 9 et en l'adaptant pour un bloc c on obtien la figure 10.

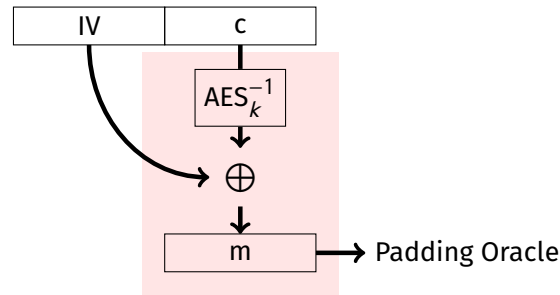


FIGURE 10 – Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible)

Le but de l'attaque est de trouver un IV tel que le padding du bloc déchiffré m est valide. En effet, si on arrive à trouver un IV tel que $m = \boxed{15|\dots|15}$ alors

$$IV \oplus \boxed{15|\dots|15} = \text{AES}_k^{-1}(c)$$

On aura donc réussi à déchiffrer c! Pour arriver à trouver cet IV on procède de la manière suivante :

On pose $IV = \boxed{00|00|00|00|00|00|00|00|00|00|00|00|00|b}$ avec b un **byte** quelconque. On fait varier b (256 possibilités) jusqu'à obtenir une réponse positive du padding oracle. Une fois arrivé à cette étape il existe 2 situation dans lesquelles on peut se trouver :

1. Dans le cas le plus courant on aura $m = \boxed{??|??|\dots|??|00}$. C'est ce qu'on cherche!
2. Mais il se peut aussi que l'avant dernier **byte** permette à plus d'un padding d'être correcte. En effet, si ce **byte** vaut 01 alors il existe un **byte** b tel que l'IV produise un $m = \boxed{??|??|\dots|??|01|01}$.

Ce deuxième cas est gênant mais il est facile de vérifier dans quel cas on se trouve en modifiant l'avant dernier **byte** de l'IV. En effet si en modifiant l'avant dernier **byte** de l'IV produit une réponse favorable du padding oracle cela signifie qu'on est dans le premier cas sinon on est dans le deuxième.

$\boxed{??|??|\dots|42|00} \rightarrow$ padding valide

$\boxed{??|??|\dots|42|01} \rightarrow$ padding invalide

Une fois à cette étape on peut créer un IV qu'on appelle ZIV (zeroing IV). Cet IV permet de mettre à 0 les **bytes** que l'on change dans m. Dans notre cas on a directement

IV = ZIV.

On sait maintenant comment créer le premier ZIV, supposons que l'on soit capable de créer le n-ième ZIV : ZIV_n (cet IV assure que les n derniers bytes de m valent o). Essayons de trouver ZIV_{n+1} .

On pose $IV = ZIV_n + \boxed{oo \dots oo | b | n \dots n}$ avec n bytes \boxed{n} et b, un byte quelconque. Cet IV permet d'avoir $m = \boxed{?? | ?? \dots ?? | n \dots n}$ car o est l'élément neutre de \oplus dans $\mathbb{Z}/2\mathbb{Z}$.

Comme à la première étape (celle qui sert à trouver le premier ZIV), on fait varier b (256 possibilités) jusqu'à avoir une réponse positive du padding oracle. En effectuant la même vérification qu'à la première étape, on est en mesure de trouver b tel que $m = \boxed{?? | ?? \dots ?? | n \dots n}$ avec cette fois ci n+1 byte \boxed{n} . Alors on remarque que $ZIV_{n+1} = IV \oplus \boxed{oo \dots oo | n \dots n}$ car $m \oplus \boxed{oo \dots oo | n \dots n} = \boxed{?? \dots ?? | ?? | oo \dots oo}$.

Ainsi, on est en mesure de trouver un IV (ZIV_{16}) tel que $m = \boxed{oo \dots oo}$ et donc comme on l'a dit en début de section et comme on peut le voir sur la figure 11 on peut déchiffrer le bloc c car si $a \oplus b = 0$ alors $a = b$ donc $AES_k^{-1}(c) = ZIV_{16}$.

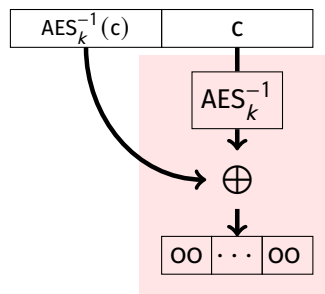


FIGURE 11 – Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible)

Généralisation à n blocs

On rappelle le schéma de déchiffrement d'AES CBC sur la figure 12. On remarque alors que si on est en mesure de trouver $AES_k^{-1}(c_i)$ alors on est capable de trouver $m_i \forall i$. En effet $AES_k^{-1}(c_i) \oplus c_{i-1} = m_i$. Donc en déchiffrant chaque bloc avec une attack par padding oracle on peut déchiffrer l'entièreté du chiffré.

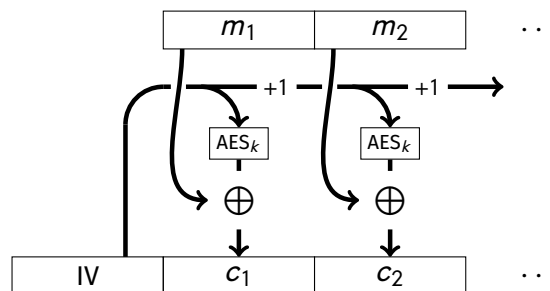


FIGURE 13 – Fonctionnement d'AES CTR, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , Le $+1$ signifie qu'on incrémente de 1 l'IV généré aléatoirement.

lors du déchiffrement. Cela permet de garantir que la donnée que vous déchiffrez est bien celle que l'on souhaite récupérer.

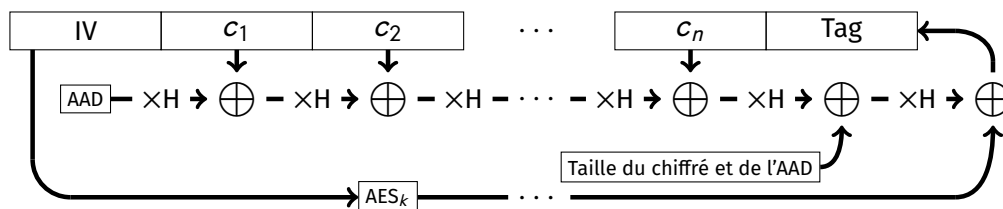


FIGURE 14 – Fonctionnement du GMAC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , et $\times H$ la multiplication dans $GF(2^{128})$ par $H = AES_k(0^{128})$.

Dans la figure 14, une nouvelle entrée, appelée AAD (Additional Authenticated Data), est introduite. L'AAD est une donnée que l'on souhaite authentifier, mais pas nécessairement chiffrer. Par exemple, imaginons que vous envoyez un mot de passe accompagné d'une adresse web. Vous voudriez chiffrer le mot de passe pour le protéger, tout en laissant l'adresse en clair. Cependant, si l'adresse n'est pas authentifiée, un attaquant pourrait la modifier, et vous ne le sauriez pas lors du déchiffrement. L'AAD permet de protéger contre ce risque en garantissant que l'adresse n'a pas été altérée."

3.3.3 GCM

GCM est la combinaison du mode CTR (voir 3.3.1) et de la création d'un Tag avec GMAC (voir 3.3.2). Combiné à AES, cela nous donne l'algorithme de chiffrement authentifié le plus largement utilisé. Son fonctionnement est décrit sur la figure 15.

Pour résumer, AES GCM prend 4 entrées :

1. un nonce (ou IV) généré aléatoirement (que l'on choisit de mettre au début du chiffré pour réduire le nombre d'objets à stocker).

Les options permettront de choisir l'algorithme utilisé. Pour le moment peuvent soit valoir "aes" pour AES-GCM, ou bien "cha" pour ChaCha20-Poly1305 ou encore "aes_stream" pour la version au fil de l'eau d'AES-GCM.

le programme de chiffrement s'appelle rucrypt et celui de déchiffrement rudecrypt

3.4.2 Mise en place

Cette fois-ci, contrairement à Ed25519, je n'ai pas tout reprogrammé. J'ai décidé d'utiliser des crates de confiance, largement éprouvées. Pour les identifier, je me suis aidé d'un site qui référence toutes les solutions couramment utilisées pour la cryptographie en Rust [4]. J'utilise AES 256 (voir le tableau 2). Pour chaque algorithme, je dois donc générer, avant le chiffrement, une clé de 32 octets (256 bits) et un nonce (vecteur d'initialisation) de 12 octets (96 bits). Il est démontré que cacher le nonce n'a aucun intérêt en termes de sécurité. Je place donc le nonce en clair avant le texte chiffré.

On pourrait s'arrêter là, mais il est possible d'ajouter un niveau de sécurité supplémentaire. Actuellement, le chiffrement et le déchiffrement se font avec une seule clé (c'est-à-dire une clé symétrique). Le problème de cette symétrie est que toute personne ayant accès à cette clé peut lire le message en clair. Dans le cadre de ma mission, cela représente une potentielle faille. En effet, le but final de ce programme est de permettre la création de sauvegardes chiffrées de données. Il est donc préférable d'utiliser une clé asymétrique (une pour le chiffrement et une pour le déchiffrement). Dans ce cas de figure, tout le monde peut effectuer une sauvegarde de ses fichiers, mais seules les personnes autorisées pourront les récupérer, cela permet de garder le contrôle sur les flux. Pour cette raison, je génère une clé privée et une clé publique RSA, avec laquelle je chiffre ma clé symétrique. Ainsi, seule la personne en possession de la clé privée pourra déchiffrer le document. Pour plus de simplicité, je place la clé symétrique chiffrée avant le nonce.

`rucrypt(data) →` `RSA(clef symétrique) | nonce | data chiffré`

3.5 SoftHSM

Le but des cryptographe est de créer des problèmes facile à résoudre quand on a la clefs mais beaucoup trop long à casser le cas inverse. Donc, dans beaucoup de cas trouver un moyen de récupérer cette clef est beaucoup plus raisonnable que de chercher une faille dans le cryptosystème. Il faut donc trouver un moyen sécuriser de garder nos clefs, i.e il faut un coffre fort moderne.

Pour ça, il existe les HSM (Hardware Security Module) qui sont des dispositifs physiques dédiés à la gestion, au stockage et à la protection des clés cryptographiques. Ils sont conçus pour exécuter des opérations cryptographiques, telles que le chiffrement, le déchiffrement, la signature numérique et la gestion de clés, de manière sécurisée.



FIGURE 16 – HSM de Thales

Ici on ne s'intéressera pas réellement aux HSM mais plutôt aux SoftHSM, une implémentation logicielle d'un HSM. Ils permettent de développer des programmes, faire des tests en amont d'une utilisation d'un HSM.

Dans un premier temps j'ai dû me familiariser avec ce nouvel objet, puis, j'ai dû regarder les solutions en Rust qui nous permettent de communiquer avec des HSM, les documenter et les tester (via un SoftHSM).

Pour communiquer avec un SoftHSM j'ai dû me plonger dans la doc du logiciel. Avant de pouvoir entamer des opérations cryptographiques il faut créer un environnement dédié dans le SoftHSM.

Pour le créer la commande est la suivante :

```
sudo softhsm2-util --init-token  
  --slot <slot-num>  
  --token <token-label>
```

et pour le supprimer :

```
sudo softhsm2-util --delete-token --label <token-label>
```

Ici il faut bien comprendre qu'un token est une émulation d'un HSM (on peut émuler plusieurs HSM sur un SoftHSM), le slot est un point d'entrée d'un HSM, il simule l'endroit où l'on insère nos informations dans un HSM).

Après avoir initialisé mon SoftHSM, je peux communiquer avec lui en utilisant le standard PKCS11. C'est cette partie plus tard que je devrais programmer en Rust. Pour le

moment, j'utilise openssl, un logiciel qui me permet directement de communiquer avec le SoftHSM. Après avoir étudié la documentation, j'en ai tiré les informations suivantes : Pour créer une clef je dois utiliser :

```
sudo pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so
-l -p <usr-PIN>
--keygen --key-type <enc-mech>
--id <clef-id>
--label <clef-label>
```

Avec <enc-mech> qui prend la forme AES :16, AES :32, ect. Pour la supprimer on utilise la commande :

```
sudo pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so
-l -p <usr-PIN>
-b --type secrkey
--id <clef-ID>
```

Maintenant qu'on a réussi à créer des clefs, on peut les utiliser pour chiffrer et déchiffrer des données. Pour le faire, il faut utiliser :

```
sudo pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so
--login -p <usr-PIN>
--encrypt
--id <key-ID>
-m AES-CBC-PAD
--iv <iv-value>
-i <input file>
-o <output file>
```

```
sudo pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so
--login -p <usr-PIN>
--decrypt
--id <key-ID>
-m AES-CBC-PAD
--iv <iv-value>
-i <input file>
-o <output file>
```

Pour la suite, j'utilise la crate cryptoki de Rust que j'ai trouvée grâce à [\[4\]](#), ce qui me permet de garantir sa fiabilité. La documentation de cette crate n'étant pas encore faite, j'ai créé un fichier tuto dans lequel je réalise toutes les opérations que je pourrais être amené à faire sur le SoftHSM, c'est-à-dire créer et supprimer des clés ainsi

que chiffrer et déchiffrer des données. Pour y arriver, j'ai parcouru le code source de la documentation et je me suis aidé d'outils tels que ChatGPT. Après beaucoup d'essais, j'ai fini par obtenir un tutoriel fonctionnel.

La dernière étape consistait à vérifier que les résultats produits par mon programme correspondaient bien aux attentes. Pour cela, il est nécessaire de tester le programme avec des vecteurs de test. Chaque vecteur de test fournit une clé, un IV (Initialisation Vector), des données à chiffrer, le résultat chiffré attendu, et, si le programme le permet, des données à authentifier ainsi que le TAG correspondant. L'objectif de ces vecteurs est de comparer les sorties théoriques avec celles produites par notre algorithme. Pour que les tests soient validés, il faut que ces dernières soient identiques.

Pour faire ces test j'ai eu une grosse difficulté : je ne peux pas imposer la valeur d'une clef via mon programme (c'est une donnée protégé donc le softhsm me l'interdit). Pour contourner ce problème, je lance un script bash dans mon code rust qui crée la clef avec openssl. Ce script est le suivant :

```
echo -n <key-value> > aes_key.txt
xxd -r -p aes_key.txt > aes_key.bin
sudo pkcs11-tool --module /usr/lib/softhsm/libsofthsm2.so
  -l -p <usr-PIN> --write-object
aes_key.bin
  --type secrkey --key-type <enc-mech>
  --id <clef-id>
  --label <clef-label>
```

Mon programme a passé tout les test, et donc, grâce à mon tutoriel je suis maintenant capable de remplacer openssl par un programme "maison" en rust.

3.6 Signature

Ensuite, pour compléter mes connaissance en cryptographie, j'ai eu à implémenter 2 algorithmes de signature. Comme pour le TAG qu'on crée dans AES GCM (voir [3.3.2](#)), une signature électronique permet d'authentifier une donnée et de garantir son intégrité. Mais, contrairement au Tag, elle peut aussi servir comme une preuve dans un contexte légal. Avec une signature numérique, l'expéditeur ne peut pas nier avoir signé la donnée. En effet on verra plus tard que les algorithmes utilisés nécessitent des clefs asymétriques : une clef publique et une clef privée. Comme seul l'expéditeur connaît la clef privée, il est le seul pouvoir signer le document.

3.6.1 RSA

Génération des clefs

Pour créer la clef publique et la clef privée il faut :

1. Poser $n = pq$ avec p et q deux nombres premiers
2. Calculer $\varphi(n)$ (l'indicatrice d'euler), le nombre d'éléments inversibles de $\mathbb{Z}/n\mathbb{Z}$
i.e $\varphi(n) = (p-1)(q-1)$
3. Choisir e , un entier premier avec $\varphi(n)$
4. Calculer $d = e^{-1}$ dans $\mathbb{Z}/\varphi(n)\mathbb{Z}$ (existe car $e \wedge \varphi(n) = 1$)

On a alors une clef privée $= (n, d)$ et une clef publique $= (n, e)$. Pour des raisons de sécurité, il faut choisir n et d grand (au moins 3072 bits) mais e peut être petit. On remarque alors le grand défaut de RSA : les clefs sont extrêmement grandes. C'est la principale raison pour laquelle on essaye d'éviter d'utiliser RSA quand il faut générer plusieurs clefs. Ce problème ne fera que devenir de plus en plus important avec le temps car il faut constamment adapter la taille des clefs avec l'apparition des nouveaux composants.

Signer avec RSA

Pour une donnée quelconque m on calcule $h = \text{hash}(m)$ avec hash une fonction de hachage tel que

$$\forall m, \text{hash}(m) < n$$

Si cette condition n'est pas vérifiée rien ne garantit que la signature pourra être vérifiée. Enfin pour calculer la signature s associée à m , on chiffre h en calculant

$$s = h^d [n]$$

On remarque bien qu'on utilise la clef privée pour créer la signature.

Vérifier l'intégrité des données

N'importe qui en possession de la clef publique (n, e) est en mesure de dire si le message \tilde{m} qu'il a reçu avec la signature s est authentique (i.e $\tilde{m} = m$). Pour ça il doit calculer $\tilde{h} = \text{hash}(\tilde{m})$ et déchiffrer la signature en calculant $s^e [n]$. Son message est authentique si et seulement si

$$\tilde{h} = s^e [n]$$

Car $s^e = h^{ed} = h [n]$, donc $\tilde{h} = s^e [n] \Leftrightarrow \tilde{h} = h [n]$. Cette dernière égalité modulaire souligne bien l'importance de la condition sur le hash.

3.6.2 Courbes elliptiques avec Ed25519

Nous avons vu qu'il était possible de faire de la signature numérique avec RSA mais que ce cryptosystème n'a qu'un intérêt très limité à la vu de la taille gigantesque des clefs. Une alternative présentée ici permet est bien plus performante que RSA.

Courbe elliptique

Pour un corp K , on peut définir une courbe elliptique par une équation de la forme

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$$

avec $(a_1, \dots, a_6) \in K^6$. On sait que ces courbes sont symétrique par rapport à l'axe des abscisses (donc si P est sur la courbe alors $-P$ aussi).

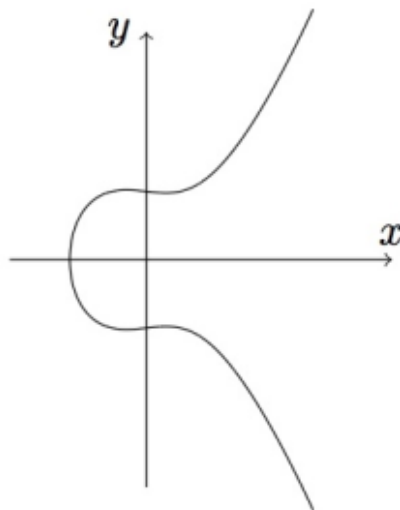


FIGURE 17 – Courbe $y^2 = x^3 - x + 9$ sur \mathbb{R}

Un problème compliqué à résoudre sur ces courbes est de trouver les points rationnels de la courbe. Par contre, on sait qu'avec deux points (U, V) rationnels tels que $U \neq -V$, on peut facilement en trouver deux autres. En effet, la droite qui passe par U et V coupera la courbe en un nouveau point W rationnel (on a le quatrième point par symétrie). De là on définit l'addition sur une courbe elliptique par

$$U + V = -W$$

On peut étendre cette définition en partant d'un point rationnel P de la courbe elliptique avec une ordonnée non nulle. Cette fois ci on prend sa tangente (on fait

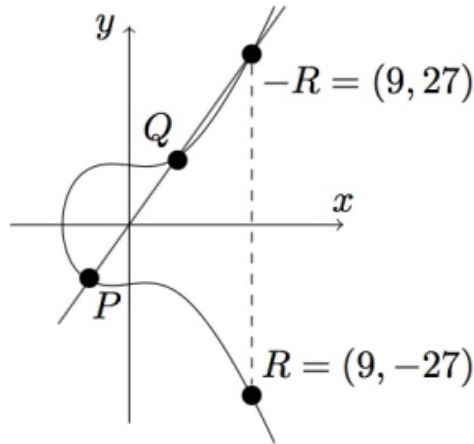


FIGURE 18 – $P + Q$ sur $y^2 = x^3 - x + 9$ sur \mathbb{R} avec $P = (-1, -3)$ et $Q = (1, 3)$

tendre U et V vers P , la droite qui passait par U et V tend bien vers la tangente passant par P). Alors cette tangente va bien couper la courbe en un deuxième point rationnel. Cette extension nous permet de définir $P + P$.

On appelle respectivement ces méthodes la méthode de la corde et de la tangente.

Courbe elliptique sur un espace à dimension finie

En cryptographie, on définit les courbes elliptiques sur des espaces à dimension finie de dimension $p : \mathbb{Z}/p\mathbb{Z}$. Avec p un premier supérieur à 3. la courbe elliptique (E) sur $\mathbb{Z}/p\mathbb{Z}$ vérifiera

$$y^2 = x^3 + ax + b$$

avec $(a, b) \in (\mathbb{Z}/p\mathbb{Z})^2$ et $4a^3 + 27b^2 \neq 0$, cette condition permet d'éviter les singularités.

On définit le groupe $(E(\mathbb{Z}/p\mathbb{Z}), +)$ par :

- $E(\mathbb{Z}/p\mathbb{Z})$ l'ensemble des points rationnels de la courbe E défini sur $\mathbb{Z}/p\mathbb{Z}$
- O l'élément neutre pour l'addition défini comme le point de la courbe à l'infini. Alors

$$\forall P \in E(\mathbb{Z}/p\mathbb{Z}), P + O = O + P = P$$

- Posant $P = (x_1, y_1)$ et $Q = (x_2, y_2)$, on définit l'addition par ces trois règles pour calculer $P + Q$:

1. Si $x_1 \neq x_2$ on utilise la méthode de la corde (voir 3.6.2)

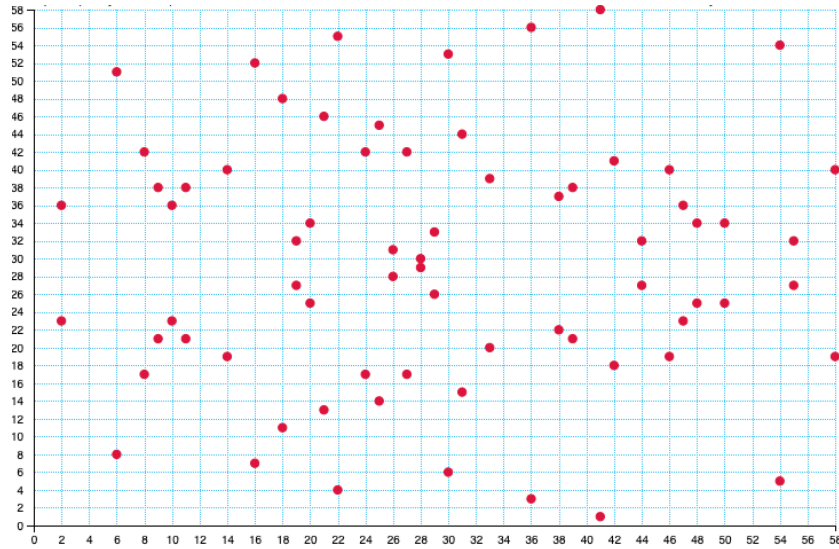


FIGURE 19 – Courbe $y^2 = x^3 - 6x + 2$ sur $\mathbb{Z}/59\mathbb{Z}$

2. Sinon si $P = Q$ on utilise la méthode de la tangente (voir 3.6.2)
3. Sinon ($P = -Q$) alors on pose

$$P + Q = O$$

Voir [1] si vous voulez voir les formules complètes.

Ed25519

On s'intéresse maintenant à un cas particulier des courbes elliptiques (toujours sur $\mathbb{Z}/p\mathbb{Z}$). On définit les courbes d'Edward comme des courbes elliptiques pouvant se mettre sous la forme

$$x^2 + y^2 = 1 + dx^2y^2$$

avec $d \in \mathbb{Z}/p\mathbb{Z}$ et $d \neq 0, 1$. On peut alors définir + par $\forall (P, Q) = ((x_1, y_1), (x_2, y_2)) \in (E(\mathbb{Z}/p\mathbb{Z}))^2$

$$P + Q = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right), \quad O = (0, 1)$$

Cette simplicité d'implémentation permet de limiter certaines attaques.

A partir de maintenant, on se concentre sur la courbe d'Edward avec $d = 37095705934669439343138083508754565189542113879843219016388785533085940283555$ défini sur $\mathbb{Z}/p\mathbb{Z}$ avec $p = 2^{255} - 19$.

Génération des clefs

Pour créer la clef publique et la clef privée il faut :

1. choisir un algorithme de hashage produisant un résultat sur 512 bits (généralement Sha512)
2. créer une chaîne k de 256 bits (de préférence générée aléatoirement).
3. calculer $h = \text{hash}(k)$ et puis s à partir des 256 premiers bits de h avec la convention "little endians".
4. calculer $A = sB$ avec B ¹ un point de Ed25519 donné dans [2]

On a alors la clef privée k et la clef publique $\text{compress}(A)$ ². On remarque bien que contrairement à RSA EdDSA permet d'avoir des clefs bien plus petite (256 bits chacune)

Signer avec Ed25519

Pour signer des données avec les courbes d'Edward on utilise l'algorithme EdDSA décrit plus précisément dans [2]. Il y a de paramètre généraux propre à l'algorithme EdDSA sur Ed25519 mais je ne les présentes pas. Pour les voir, je vous invite à lire la Section 5.1 de [2].

L'idée principale de cet algorithme est de générer à partir et du message et de la clef privée un point $R = rB$. Pour calculer r , il faut calculer le hash modulo L ³ des 256 derniers bits de du hash de la clef privé $h = \text{hash}(k)$ concaténé⁴ à la donnée m à signer.

$$r = \text{hash}(h[32..64] \parallel m) [L]$$

Ensuite on calcule \tilde{h} tel que

$$\tilde{h} = \text{hash}(\text{compress}(R) \parallel \text{compress}(A) \parallel m) [L]$$

Enfin, notant $x = r + \tilde{h}s [L]$ en représentation "little endians" alors la signature est donnée par :

$$\text{signature}_{\text{Ed25519}} = \text{compress}(R) \parallel x$$

1. Pour des raisons de simplicité, calculatoire, on utilise une convention différente que celle présentée dans ici 3.6.2 mais elle est détaillé dans [2]

2. $\text{compress}()$ est l'algorithme de compression de EdDSA. Dans notre cas, c'est une chaîne de 256 bits. les 255 premiers bits sont les 255 premiers bits de la coordonnée y du point en convention "little endians" (comme $y < p$, son bit de poids fort est tout le temps nul). Le dernier bit de la chaîne vaut 1 si le point est négatif, 0 sinon

3. L est une constante introduite dans [2] et vaut $2^{252} + 2774231777372353535851937790883648493$

4. symbolisé par \parallel

On remarque que la signature fait 512 bits et à besoins de la clef privée et de la clef publique (dérive directement de la clef privée) pour la générer.

Vérifier l'intégrité des données

N'importe qui en possession de la clef publique A est en mesure de dire si le message \tilde{m} qu'il a reçu avec la signature est authentique (i.e $\tilde{m} = m$). Pour ça il doit décompresser⁵ le point R en récupérant les 256 premiers bits de la signature. Ensuite il faut récupérer l'entier x écrit en convention "little endian" dans les 256 derniers bits de la signature et calculer

$$\tilde{h} = \text{hash}(\text{compress}(R) \parallel \text{compress}(A) \parallel \tilde{m}) [L]$$

Enfin, le message sera authentique si et seulement si on a l'égalité dans le groupe $(E(\mathbb{Z}/p\mathbb{Z}), +)$

$$xB = R + \tilde{h}A$$

Car

$$\begin{aligned} xB = R + \tilde{h}A &\Rightarrow rB + \tilde{h}sB = R + \tilde{h}A \\ &\Rightarrow R + \tilde{h}A = R + \tilde{h}A \\ &\Rightarrow (\tilde{h} - \tilde{h})A = 0 \\ &\stackrel{6}{\Rightarrow} \tilde{h} = \tilde{h} \\ &\Rightarrow m = \tilde{m} \end{aligned}$$

On remarque qu'on a bien besoins que de la clef publique A pour vérifier la signature.

3.6.3 Serveur signature

3.7 Wireguard

Le but ici est de créer à l'aide de Wireguard un flux chiffré entre plusieurs machines pour permettre à ces machines de communiquer de manière simple et sécurisé.

3.7.1 Configuration

Pour le serveur et les clients il faut créer un fichier de configuration dans `\ect\wireguard\wg0.conf` (On n'est pas obligé de l'appeller wg0). A l'intérieur, le fichier est séparé en au moins 2 sections :

5. Algorithme assez intuitif mais avec des astuces de calcul détaillé dans [2]. Il consiste à récupérer la coordonnée y , vérifier qu'il existe bien des points avec cette coordonnée appartenant à la courbe, puis choisir le bon point grâce au signe

- une qui commence par [Interface], elle contient les informations de la machine, comme la clef privée et adresse.
- une autre (ou plus) qui commence par [Peer] et qui contient les informations des machines (comme la clef publique et les adresses autorisées) avec lesquelles on va communiquer.

Pour chaque machine, il faudra générer une paire de clefs. Pour cela, WireGuard nous fournit un outil. Il permet de créer une clé privée et de dériver la clé publique à partir de celle là.

```
wg genkey > privatekey
wg pubkey < privatekey > publickey
```

Serveur

Pour la configuration du serveur, il faut rajouter le port utilisé pour communiquer (typiquement 51820). Une configuration complète côté serveur pourra ressembler par exemple à

```
[Interface]
PrivateKey = uKlFiRw6goZdHuPNeThISu7sGqr8JH3U+LrQ3VbaBXk=
Address = 10.8.8.254/24
ListenPort = 51820
```

```
[Peer]
PublicKey = 4N2H/21U4kYpS5KZlQfj0jddlvs7bgr0Z3ZHHBqF8lU=
AllowedIPs = 10.8.8.1/32
```

```
[Peer]
PublicKey = NIeEDlBPTJrM7389yaRKzXWJeAEicmrFm0zkfhnDVCo=
AllowedIPs = 10.8.8.2/32
```

Chaque peer est un client avec lequel peut communiquer le serveur. Attention, il faut bien penser à activer l'IP forwarding du serveur pour lui permettre aux clients de communiquer entre eux (en pas uniquement au serveur).

Pour l'activer il faut décommenter la ligne `net.ipv4.ip_forward=1` du fichier `/etc/sysctl.conf`.

Clients

La configuration d'un client est quasiment identique à celle d'un serveur à la différence qu'il n'y a qu'un peer : le serveur. Elle ressemblera typiquement à

```
[Interface]
PrivateKey = uBl+tmfka8ukXcEV3vz5wtYinhZpL0EiA0e+gwhke28=
Address = 10.8.8.1
```

```
[Peer]
PublicKey = GigedMPFFe+r7SB4s48vnEnPREiAXX/TLB07hBuAGm4=
Endpoint = 192.168.218.180:51820
AllowedIPs = 10.8.8.0/24
PersistentKeepalive = 25
```

La dernière ligne permet d'éviter d'avoir des problèmes mais n'est pas obligatoire.

3.7.2 Mise en place

Pour activer la configuration il suffit de lancer `wg-quick up wg0` et pour la désactiver `wg-quick down wg0`. Maintenant, si on veut qu'elle s'active automatiquement quand la machine s'allume il faut lancer `sudo systemctl start wg-quick@wg0` (et `sudo systemctl stop wg-quick@wg0` pour que ça ne soit plus le cas).

Attention, pour l'instant les clefs et les fichiers de configuration sont accessibles à tout le monde. Il faut veiller à supprimer les clefs (on en a plus besoins une fois qu'elles sont dans les fichiers de configurations) et à restreindre l'accès aux fichiers de configuration.

```
sudo mkdir -p /etc/wireguard/
sudo chmod 700 /etc/wireguard
```

3.8 Signal

3.9 Port knocking

3.10 Algo post quantique

4 Le stage d'application dans la construction de mon projet professionnel

5 Conclusion

Références

[1] Dan BONEH et Victor SHOUP. *A Graduate Course In Applied Cryptography*. 2023.

- [2] S. JOSEFSSON et I. LIUSVAARA. “Edwards-Curve Digital Signature Algorithm (EdDSA)”. In : (2017).
- [3] Mike ROSULEK. *The Joy of Cryptography*. 2021.
- [4] *Showcase of notable cryptography libraries developed in Rust*. URL : [cryptography.rs](#).