

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	8
2	Contexte du stage et organisation de l'entreprise	8
2.1	Groupe Thales	8
2.2	Thales Service Numérique (TSN)	9
2.3	Direction Technique (DT)	10
2.4	Le positionnement du stage dans les travaux de l'entreprise	11
3	Travaux réalisé	11
3.1	Chiffrement	11
3.1.1	AES	11
3.1.2	AES CBC	15
3.1.3	AES GCM	21
3.1.4	Programme de chiffrement/déchiffrement	23
3.2	HSM	25
3.3	Signature	27
3.3.1	RSA	27
3.3.2	Courbes elliptiques avec Ed25519	28
3.3.3	Serveur signature	33
3.4	Wireguard	34
3.4.1	Configuration	35
3.4.2	Mise en place	36
3.5	Port knocking ¹	36
3.5.1	Problème et solution	36
3.5.2	Mon implémentation	37
3.6	Signal	38
3.6.1	Extended Triple Diffie-Hellman (X3DH)	38
3.6.2	Double ratchet	40
3.7	Algorithmes post-quantiques	40
4	Conclusion	43

1. En réalité, je ne fais pas du Port knocking mais du SPA (Single Packet Authorization) qui, à la différence du port knocking inclu de la cryptographie et est bien plus sécurisé

Table des figures

1	Ping pong entre le client et le serveur avec un protocole TCP	6
2	Chiffres clés de 2023 du groupe Thales	9
3	Axes majeurs du développement à Thales	9
4	fonctionnement d'AES	12
5	Réorganisation d'un bloc de 128 bits	13
6	Effet de SubBytes sur un bloc, avec $\tilde{s} = S(s)$	13
7	Effet de ShiftRows sur un bloc	14
8	réorganisation des clefs en suite de mots de 32 bits	15
9	Fonctionnement d'AES ECB, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k	15
10	Fonctionnement d'AES CBC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k	16
11	Déchiffrement d'un bloc (en rouge : ce qui ne nous est pas accessible) .	18
12	Illustration de l'effet de ZIV_n sur le message déchiffré par le serveur(en rouge : ce qui ne nous est pas accessible)	19
13	Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible) .	21
14	Déchiffrement d'AES CBC avec en rouge ce qui ne nous est pas accessible	21
15	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.	22
16	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 $H = AES_k(0^{128})$	22
17	Fonctionnement d'AES GCM avec en vert CTR et en rouge GMAC	23
18	HSM de Thales	25
19	Courbe $y^2 = x^3 - x + 9$ sur \mathbb{R}	29
20	$P + Q$ sur $y^2 = x^3 - x + 9$ sur \mathbb{R} avec $P = (-1 - 3)$ et $Q = (1, 3)$. .	30
21	Courbe $y^2 = x^3 - 6x + 2$ sur $\mathbb{Z}/59\mathbb{Z}$	31
22	architecture du serveur signature	34
23	Algorithmes finalistes du round 3 à standardiser (voir [9])	41
24	Problème simple : connaissant (A, z) , trouver y	41
25	Problème dur : connaissant (A, z) , trouver (y, e)	42
26	Illustration d'un treillis issu du choix de (y, e)	42

Liste des tableaux

1	table de vérité du XOR	5
2	Variante d'AES	12

Glossaire

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
4. **American Standard Code for Information Interchange (ASCII)** : C'est un code de caractères standard utilisé pour représenter les textes en informatique et en télécommunication. L'ensemble des caractères ASCII sont codé sur 7 bits ce qui laisse donc $2^7 = 128$ caractères possibles (donc on est très limité avec l'ASCII)
5. **UTF-8** : Comme [ASCII](#), c'est un code de caractères standard utilisé pour représenter les textes en informatique et en télécommunication. Mais par contre il code ses caractères sur 1 à 4 [bytes](#) selon la rareté d'utilisation du caractère.
6. **Galois Field (GF)** : En mathématiques, un corps fini ou corps de Galois est un corps qui contient un nombre fini d'éléments. L'exemple le plus courant de corps finit est $\mathbb{Z}/p\mathbb{Z}$.
7. **Diffie Hellman (DH)** : L'échange Diffie-Hellman est un protocole qui permet à deux parties, Alice et Bob, de partager un secret commun K . Le protocole repose sur la difficulté de résoudre le problème du logarithme discret.

Avec $p \in \mathbb{P}$ et $g \in \mathbb{Z}/p\mathbb{Z}$, Alice et Bob choisissent chacun un nombre qu'ils gardent secret et calculent de leur côté

$$A = g^a[n], \quad B = g^b[n]$$

Alice envoie A à Bob et réciproquement, ensuite Alice (resp Bob) calcule $K = B^a[n]$ (resp $K = A^b[n]$)

En effet on a bien $A^b = g^{ab} = B^a[n]$ et comme a (resp b) n'est connu que de Alice (resp Bob) il n'y a que Alice et/ou Bob qui peuvent retrouver ce secret K .

On remarque que ce protocole à besoin de la présence des 2 parties pour permettre l'échange (échange synchrone)

8. **Keyed-Hash Message Authentication Code (HMAC)** : HMAC est un code d'authentification de message qui utilise une fonction de hachage
9. **Key Derivation Function (KDF)** : une KDF permet de générer une clef à partir d'une donnée arbitraire. En plus de ça, une KDF doit garantir que la sortie est indépendante de l'entrée et qu'elle est "difficile" à deviner.
10. **HMAC-based Key Derivation Function (HKDF)** : HKDF est une **KDF** qui utilise la fonction **HMAC**.
11. **Concaténation (||)** : On ajoute bout à bout les deux suites de bits
12. **Transmission Control Protocol (TCP)** : Le TCP est un protocole de communication qui permet de transmettre des données de manière fiable et ordonnée sur un réseau.

Lors d'une communication TCP il y a un ping pong entre le client et le client

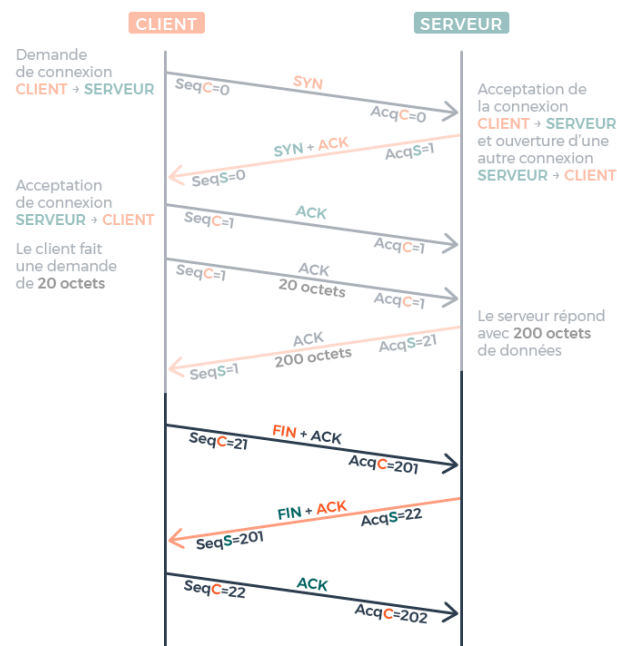


FIGURE 1 – Ping pong entre le client et le serveur avec un protocole TCP

13. **User Datagram Protocol (UDP)** : Comme pour [TCP](#), UDP est un protocole de communication qui permet de transmettre des données mais cette fois ci de manière non fiable et sans connexion.
Une caractéristique importante de l'UDP dans le cadre de mon stage est qu'il n'y a pas de détection d'erreur. Ainsi, un attaquant n'aura aucun retour sur les paquets qu'il envoie au serveur UDP.
14. **Algorithmes post-quantique** : Les algorithmes post-quantiques sont des algorithmes de cryptographie qui sont conçus pour résister aux attaques d'un ordinateur quantique.
15. **National Institute of Standards and Technology (NIST)** : C'est un institut de recherche et de normalisation américain. Dans le domaine de la cryptographie, le NIST joue un rôle important en définissant les standards et les recommandations pour les algorithmes et les protocoles de cryptographie utilisés pour protéger les informations sensibles. Les standards cryptographiques du NIST sont largement adoptés et respectés à l'échelle internationale, et ils ont un impact significatif sur la sécurité des systèmes d'information et des communications.
16. **Treillis** : On appelle treillis tout ensemble ordonné dans lequel deux éléments quelconques ont toujours une borne supérieure et une borne inférieure

1 Introduction

Dans le cadre de ma deuxième année d'études à l'Ecole Centrale de Lyon, j'ai eu l'opportunité de réaliser un stage d'application de six mois au sein de Thales, de mai à novembre 2024. Sous la tutelle de M. Mahmoud Chilali, j'ai occupé le poste de cryptographe/développeur cryptographique.

Au cours de ce stage, j'ai approfondi mes connaissances en cryptographie en apprenant à maîtriser divers algorithmes de chiffrement et de signature numérique, tels que AES, RSA ou encore Ed25519. J'ai également eu l'occasion de mettre certains algorithmes à l'épreuve, en démontrant notamment la vulnérabilité du chiffrement AES en mode CBC. Parallèlement, j'ai découvert d'autres aspects de la cryptographie, tels que la protection des données sensibles avec les modules de sécurité matériels (HSM) et d'autres sujets passionnants.

L'objectif principal de mon stage était de préparer le terrain pour une refonte du logiciel Sakyena en Rust, j'ai pu réécrire une grande partie de ces algorithmes dans ce nouveau langage passionnant.

Ce rapport présente l'entreprise qui m'a accueilli et le contexte dans lequel j'ai effectué mon stage. Ensuite, je détaille les résultats de mes travaux et les connaissances que j'ai acquises au cours de ce stage, en mettant l'accent sur les algorithmes que j'ai étudiés et programmé en Rust. Enfin, je montre l'impact de ce stage sur la construction de mon projet professionnel

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

Les informations qui m'ont permis d'écrire cette sous section sont tirées de [6].

« *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.

Les clients de Thales sont de grandes organisations - États, administrations, entreprises, etc. Ses clients sont responsables des opérations, des services et des in-



FIGURE 2 – Chiffres clés de 2023 du groupe Thales

frastructures 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 des ses clients Thales opère selon 3 points clés



FIGURE 3 – 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).

Safekeyna 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.

Safekeyna 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 Safekeyna.

3 Travaux réalisés

3.1 Chiffrement

3.1.1 AES

Toutes les informations qui m'ont permis d'écrire cette sous section sont directement tirées de [2].

Face au manque de standard de chiffrement, le [NIST](#) a lancé un concours en 1997 pour créer un nouveau standard : 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 variantes² :

2. à l'origine cet algorithme pouvait supporter des blocs de 128, 192 et 256 bits. Cette fonctionna-

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)

Le fonctionnement d’AES est relativement simple en principe. Il traite son entrée (un bloc de 128 bits) en appliquant une permutation Π répétée plusieurs fois, à l’exception du dernier tour où une permutation légèrement différente, notée $\hat{\Pi}$, est utilisée. La particularité de ces deux permutations, Π et $\hat{\Pi}$, réside dans leur inversibilité, ce qui signifie qu’il est possible de retrouver l’entrée d’origine en appliquant l’opération inverse. Cette propriété permet de déchiffrer le message en remontant étape par étape le processus d’AES.

En plus de ces permutations, le résultat de chacune est soumis à une opération XOR³ avec une clé k_i , où i correspond au numéro du round⁴. Ces clés sont calculées à partir d’une unique clé secrète de 128 bits (voir 3.1.1).

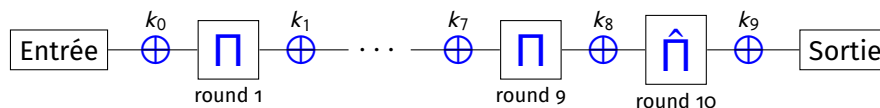


FIGURE 4 – fonctionnement d’AES

Permutation Π

Cette permutation ne dépend pas des paramètres de l’algorithme et peut donc être précalculé⁵. Elle se décompose en trois sous-permutations (toutes inversibles) : SubBytes, ShiftRows et MixColumns. Pour plus facilement comprendre ces 3 sous-permutations, il faut s’imaginer qu’un bloc de 128 bits⁶ est une matrice 4×4 où chaque cellule contient un byte

lité n’a malheureusement pas été conservé par NIST. Cette fonction aurait permis à AES d’être post quantique.

3. L’avantage du XOR est qu’il est auto-inverse, c’est-à-dire $\text{XOR}^2 = \text{id}$, donc $\text{XOR}^{-1} = \text{XOR}$. Ainsi, il suffit de réappliquer le XOR pour remonter le processus d’AES

4. Un round est le passage dans une permutation

5. Cela permet une exécution très rapide

6. i.e 4 bytes

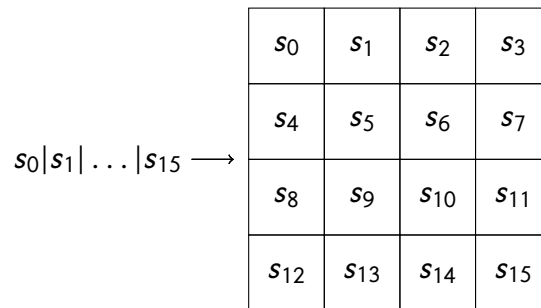


FIGURE 5 – Réorganisation d'un bloc de 128 bits

Une fois ce passage au format matriciel admis, on peut facilement expliquer les sous-permutations :

1. SubBytes

On applique à chaque élément de la matrice une permutation $S : \{0, 1\}^8 \rightarrow \{0, 1\}^8$ (donc d'un **byte** vers un autre).

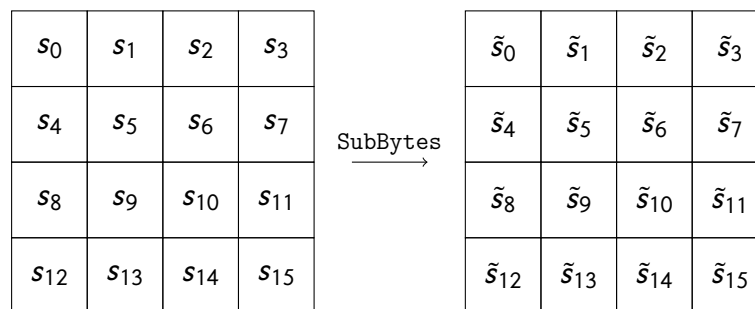


FIGURE 6 – Effet de SubBytes sur un bloc, avec $\tilde{s} = S(s)$

RAJOUTER DEF S

2. ShiftRows

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

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

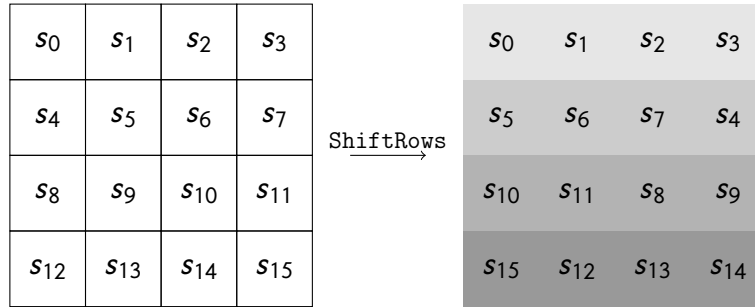


FIGURE 7 – Effet de ShiftRows sur un bloc

3. MixColumns

Pour cette permutation, on calcule dans $GF(2^8)$ ⁷ le produit matriciel (à gauche) de notre bloc par :

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

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

Donc, en résumé :

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

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 telle que⁸ :

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

7. muni du polynôme irréductible $x^8 + x^4 + x^3 + x + 1$ ie 100011011

8. C'est exactement Π mais sans la permutation MixColumns

Création des clefs k_i

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

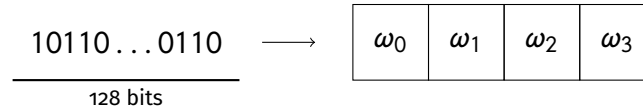


FIGURE 8 – réorganisation des clefs en suite de mots de 32 bits

On pose la première clef $k_0 = k = \boxed{\omega_{0,0}|\omega_{0,1}|\omega_{0,2}|\omega_{0,3}}$. Ensuite, on calcule $k_i = \boxed{\omega_{i,0}|\omega_{i,1}|\omega_{i,2}|\omega_{i,3}}$ une fonction de k_{i-1} avec :

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

Et $g : \{0, 1\}^{32} \rightarrow \{0, 1\}^{32}$ une fonction tirée des standards d'AES (voir [1]).

3.1.2 AES CBC

Toutes les informations qui m'ont permis d'écrire cette sous-section sont directement tirées de [7].

À 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 (ça correspond à l'utilisation d'AES en mode ECB).

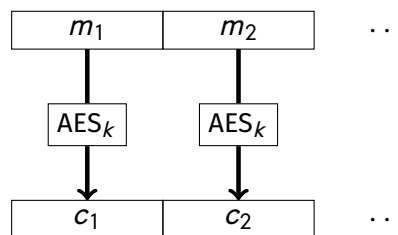
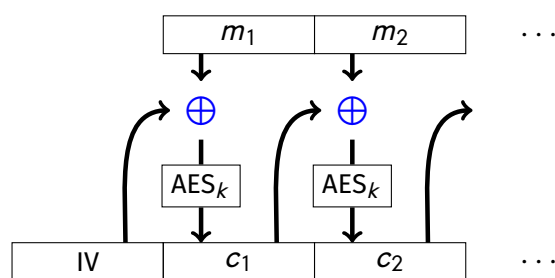


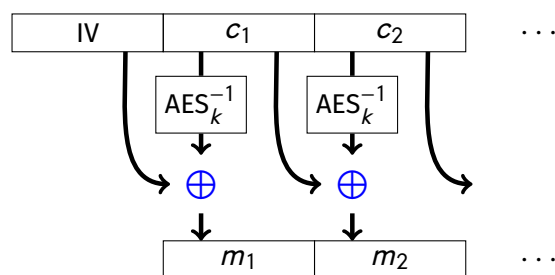
FIGURE 9 – Fonctionnement d'AES ECB, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

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

tout pratique. Il faut donc trouver une autre manière d'utiliser AES de manière sécurisée sans pour autant se limiter à des données d'un bloc de 128 bits⁹. Une solution proposée est d'utiliser AES en mode CBC (Cipher Block Chaining).



(a) Chiffrement



(b) Déchiffrement

FIGURE 10 – Fonctionnement d'AES CBC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k

Le premier bloc du chiffré est un Initialization Vector (IV), c'est un bloc choisi aléatoirement.

Ensuite, un deuxième point important est qu'il faut que la taille du message en clair soit exactement un multiple de 128 bits¹⁰. La solution à ce problème¹¹ 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 bits¹². En pratique, le padding peut prendre plein de forme différente, on peut compléter avec des 0, des bits aléatoires, ect...

La seule règle importante est que le dernier **byte** du bloc doit contenir le nombre de

9. dans la pratique, on ne chiffre quasiment jamais qu'un bloc, il faut donc un algorithme pour permettre à AES de chiffrer des données de plusieurs blocs

10. c'est aussi vrai pour toutes les autres méthodes de chiffrement par bloc

11. C'est un problème, car en pratique les données que l'on chiffre n'ont aucune raison d'avoir une taille multiple de 128

12. si le message est déjà un multiple de 128 bits on rajoute 128 bits de padding (i.e un bloc)

byte de padding qui ont été rajoutés (entre 0 et 15)¹³.

Pour ma part, mon padding suivra la règle suivante : tous les bytes du padding ont la même valeur (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

`15|15|15|15|15|15|15|15|15|15|15|15|15|15|15|15`.

J'ai aussi essayé d'implémenter un padding plus classique¹⁴ mais j'ai du mal le faire, car ce type de padding me posais des problèmes dans 3.1.2 alors que la théorie m'assure que ça ne devrait pas être le cas.

Mon implémentation

Dans le cadre de ma mission, je dois implémenter AES CBC en RUST. J'ai choisi de ne pas réimplémenter AES, car j'avais du mal à implémenter des éléments relatif à *g* (voir 3.1.1) mais aussi car une implémentation d'AES est très régulièrement inutilisable en pratique car trop facile à attaquer. Rien que des informations telles que la consommation électrique de l'ordinateur, le temps qu'il met à répondre ou encore le champ électromagnétique qu'il émet sont suffisantes pour casser une implémentation "normal"¹⁵.

J'ai donc implémenté le mode CBC avec le padding énoncé à la fin de 3.1.2. J'ai choisi pour simplifier la tâche de me restreindre à chiffrer des chaînes de caractères¹⁶. De là est apparu un pseudo-problème, le type Char en RUST peut faire de 1 à 4 byte (ce type est codé selon le standard UTF-8). Mais, au déchiffrement, ne connaissant pas le nombre de byte associé à chaque caractère, j'ai été dans l'obligation de faire comme si chaque caractère ne faisait qu'un byte. Donc, si mon message n'est pas uniquement écrit avec des caractères ASCII, j'aurais de la perte d'information. Par exemple, si je chiffre et déchiffre le message : "Je vais à l'école" j'obtiens un message du type "Je vais \$£ l'&ùcole" car les caractères "à" et "é" ne font pas partie du tableau ASCII car ils sont codés sur plusieurs bytes. J'ai choisi d'ignorer ce problème, étant donné qu'il n'est dû qu'au format de l'entrée.

13. Sinon on n'a aucun moyen de savoir ce qui faisait partie ou non du message original

14. On remplit de 0 (par exemple pour un padding de 3 bytes cela donne `00|00|02`)

15. La seule solution pour régler ce problème est d'utiliser directement les instructions du processeur

16. C'est une des premières choses que j'ai faites dans mon stage. À l'époque, cette approche me semblait la plus intuitive et la plus simple. En fait, je me compliquais la vie sans le savoir. C'est bien plus simple de ne pas se préoccuper du type d'entrée et juste de tout convertir au format binaire

Padding oracle attack

Dans cette sous-sous-section, j'explique comment j'ai attaqué AES CBC et par extension tous les modes de chiffrement par bloc non authentifié (qui ne fournissent pas une preuve d'authenticité du chiffré).

L'attaque que j'ai menée repose sur un principe simple. Pour chaque algorithme de chiffrement qui utilisent un padding, l'attaquant trouvera toujours un moyen d'avoir un padding oracle ¹⁷ (pas encore de contre-exemples). Je vais montrer ci-dessous qu'à partir du moment où on a accès à ce genre d'information, on peut déchiffrer le message.

Déchiffrer un bloc

En reprenant la figure 10 et en l'adaptant pour un bloc c on obtient la figure 11.

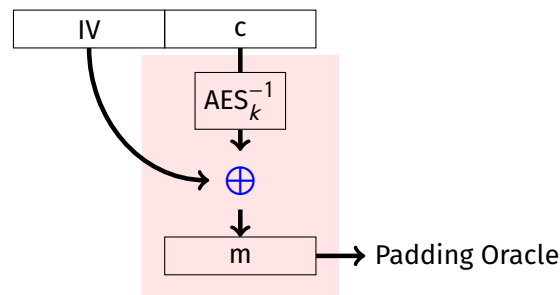


FIGURE 11 – Déchiffrement d'un bloc (en rouge : ce qui ne nous est pas accessible)

Le but de l'attaque est de trouver un IV tel que $m = \boxed{15|\dots|15}$ car si on y arrive alors, on a :

$$IV \oplus AES_k^{-1}(c) = \boxed{15|\dots|15} \Rightarrow IV \oplus \boxed{15|\dots|15} = AES_k^{-1}(c)$$

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

On pose $IV = \boxed{00|\dots|00|b}$ 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 situations 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 !

17. Avoir accès à un padding oracle, c'est pouvoir déduire du comportement du serveur si le padding du message que je lui demande de déchiffrer est correct (on peut par exemple déduire cette information du temps que met le serveur à nous répondre).

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, on a 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 $IV = ZIV = ZIV_0$ ^{18 19}.

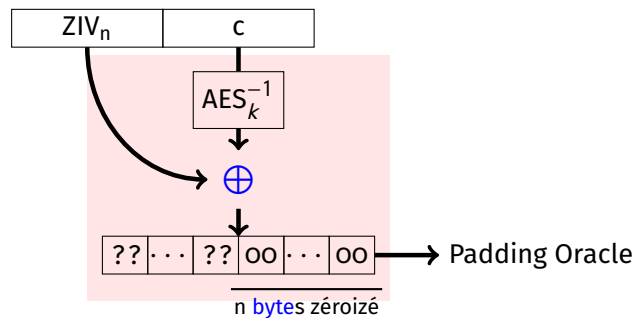


FIGURE 12 – Illustration de l'effet de ZIV_n sur le message déchiffré par le serveur(en rouge : ce qui ne nous est pas accessible)

On sait donc créer ZIV_0 . Supposons que l'on soit capable créer ZIV_n , essayons de trouver ZIV_{n+1} . On pose

$$IV = ZIV_n \oplus \boxed{00|\dots|00|b|n|\dots|n} \quad \text{avec } n \text{ bytes } \boxed{n}$$

et b, un **byte** quelconque (256 possibilités). Comme $0 \oplus x = x$ ²⁰, alors on a

$$m = \boxed{??|??|\dots|??|n|\dots|n} \quad \text{avec } n \text{ bytes } \boxed{n}$$

Comme à la première étape (celle qui permet de trouver ZIV_0), on fait varier b (256 possibilités) jusqu'à avoir une réponse positive du padding oracle.

18. Car on a fait en sorte que notre IV produise un m du type $\boxed{??|??|\dots|??|00}$

19. On note ZIV_n l'IV qui impose que les n derniers bytes de m valent 00

20. 0 est l'élément neutre de \oplus dans $\mathbb{Z}/2\mathbb{Z}$

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 $n+1$ bytes \boxed{n} . Alors

$$\Leftrightarrow \underbrace{\boxed{??|??|\dots|??|n|\dots|n|}}_{\text{avec } n+1 \text{ bytes } \boxed{n}} \oplus \underbrace{\boxed{oo|\dots|oo|n|\dots|n|}}_{\text{avec } n+1 \text{ bytes } \boxed{n}} = \underbrace{\boxed{??|\dots|??|oo|\dots|oo}}_{\text{avec } n+1 \text{ bytes } \boxed{oo}}$$

Donc

$$\begin{aligned} \text{IV} \oplus \underbrace{\boxed{oo|\dots|oo|n|\dots|n|}}_{\text{avec } n+1 \text{ bytes } \boxed{n}} \oplus \text{AES}_k^{-1}(c) &= m \oplus \underbrace{\boxed{oo|\dots|oo|n|\dots|n|}}_{\text{avec } n+1 \text{ bytes } \boxed{n}} \\ &= \underbrace{\boxed{??|\dots|??|oo|\dots|oo}}_{\text{avec } n+1 \text{ bytes } \boxed{oo}} \end{aligned}$$

Et on cherche ZIV_{n+1} tel que

$$\text{ZIV}_{n+1} \oplus \text{AES}_k^{-1}(c) = \underbrace{\boxed{??|\dots|??|oo|\dots|oo}}_{\text{avec } n+1 \text{ bytes } \boxed{oo}}$$

Donc par identification

$$\begin{aligned} \text{ZIV}_{n+1} &= \text{IV} \oplus \boxed{oo|\dots|oo|n|\dots|n|} \\ &= \text{ZIV}_n \oplus \underbrace{\boxed{oo|\dots|oo|b|n|\dots|n|}}_{\text{avec } n \text{ bytes } \boxed{n}} \oplus \underbrace{\boxed{oo|\dots|oo|n|\dots|n|}}_{\text{avec } n+1 \text{ bytes } \boxed{n}} \end{aligned}$$

D'où l'hérédité. On a donc montré par récurrence que $\forall n^{21}$, on peut trouver ZIV_n . Or, on sait que (voir 13)

$$\text{AES}^{-1}(c) = \text{ZIV}_{15}$$

Donc, on a déchiffré le bloc c en trouvant ZIV_{15}

Généralisation à n blocs

On rappelle le schéma de déchiffrement d'AES CBC sur la figure 14. On remarque alors qu'on peut très facilement généraliser à n blocs ce que l'on a fait pour un bloc, car pour trouver c_i il suffit de faire comme si c_{i-1} est l'IV et on remonte comme ça jusqu'à avoir trouvé tous les c_i .

21. dans notre cas comme on a des blocs de 16 bytes, $\forall n \in \llbracket 0; 15 \rrbracket$

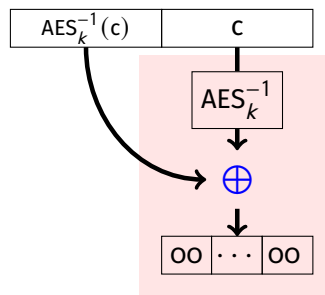


FIGURE 13 – Déchiffrement d'un bloc, (en rouge : ce qui ne nous est pas accessible)

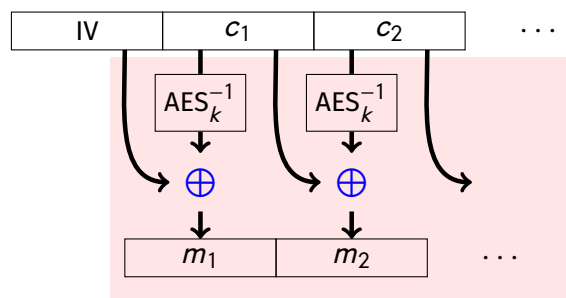


FIGURE 14 – Déchiffrement d'AES CBC avec en rouge ce qui ne nous est pas accessible

3.1.3 AES GCM

Toutes les informations qui m'ont permis d'écrire cette sous-section sont directement tirées de [7].

On s'est intéressé dans la sous-section précédente au mode CBC en montrant qu'AES CBC était facilement attaquable et donc pas fiable. On s'intéresse ici au mode GCM (Galois Counter Mode), le mode d'utilisation le plus courant d'AES. Comme pour AES CBC, AES GCM fait des opérations sur des blocs de 128 bits pour permettre de chiffrer de grands fichiers. Mais, à la différence d'AES CBC, AES GCM utilise un algorithme d'authentification en parallèle du chiffrement, ce qui permet d'assurer que le chiffré n'a pas été changé. Cette deuxième partie est essentielle, car il a été montré qu'un algorithme de chiffrement non authentifié est quasiment systématiquement attaquable.

CTR

Comme CBC, CTR (Counter) est un mode de manipulation des blocs qui permet de chiffrer des données avec des algorithmes de chiffrement par blocs (comme AES) tout en garantissant un certain niveau de sécurité. CTR, c'est GCM mais sans l'authentifi-

cation. Il est assez similaire à CBC mais à comme gros avantage de ne pas obliger de rajouter un padding à la fin des données claires. Son fonctionnement est décrit sur la figure 15.

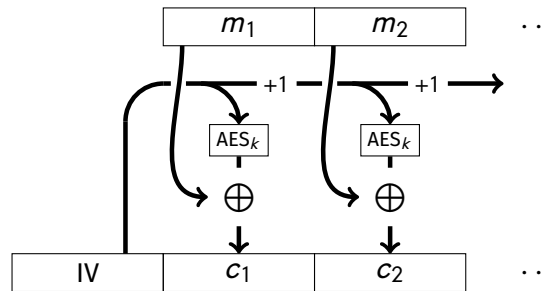


FIGURE 15 – Fonctionnement d'AES CTR, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , Le $+1$ signifie qu'on incrémente de 1 l'IV généré aléatoirement.

GMAC

Pour obtenir GCM il faut rajouter une étape d'authentification à CTR. Elle est faite par un algorithme incrémental GMAC (Galois Message Authentication Code). Cet algorithme produit au cours du chiffrement des blocs un Tag qui résume de manière sécurisée toutes les informations sur la donnée à chiffrer. L'intérêt de cela est que chaque entrée produira un Tag unique. Donc, si quelqu'un a modifié le chiffré, alors il y aura une erreur lors du déchiffrement. Cela permet de garantir que la donnée que vous déchiffrez est bien celle que l'on souhaite récupérer.

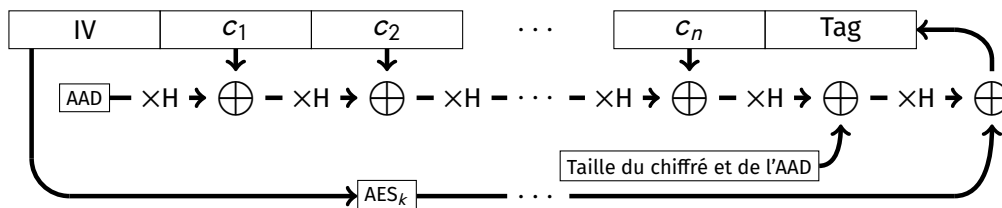


FIGURE 16 – Fonctionnement du GMAC, avec AES_k le chiffrement d'un bloc par AES avec la clef secrète k , et $\times H$ la multiplication dans $GF(2^{128})$ par $H = AES_k(0^{128})$.

Dans la figure 16, une nouvelle entrée, appelée AAD (Additional Authenticated Data), est introduite. L'AAD est une donnée que l'on souhaite authentifier, mais pas 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 garantir que l'adresse n'a pas été altérée."

GCM

GCM est la combinaison du mode CTR (voir 3.1.3) et de la création d'un Tag avec GMAC (voir 3.1.3). Combiné à AES, cela nous donne l'algorithme de chiffrement authentifié le plus largement utilisé. Son fonctionnement est décrit sur la figure 17.

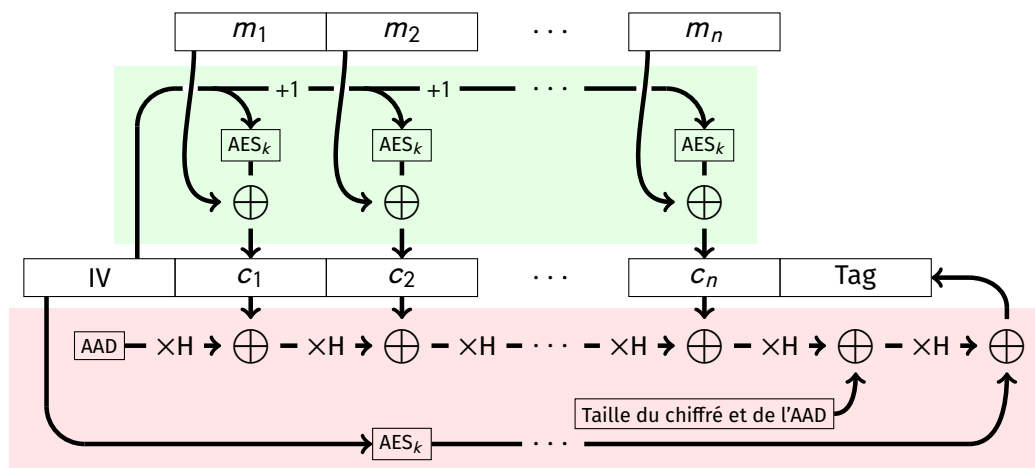


FIGURE 17 – Fonctionnement d'AES GCM avec en vert CTR et en rouge GMAC

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).
2. une clef (de préférence générée aléatoirement)
3. des données à chiffrer et authentifier
4. des données à authentifier (on peut aussi ne pas en fournir)

et fournit 2 sorties :

1. Le chiffré de la donnée à chiffrer
2. Le tag associé aux entrées (mis à la fin du chiffré)

3.1.4 Programme de chiffrement/déchiffrement

Contexte

Maintenant que j'ai pris connaissance des algorithmes pour chiffrer des données, je dois mettre en place une solution écrite en Rust qui permettra de chiffrer et de déchiffrer des fichiers. Cette solution devra permettre d'utiliser soit AES-GCM, soit ChaCha20-Poly1305 (un autre algorithme de chiffrement non détaillé précédemment). Une optimisation que je pourrais envisager serait de proposer une version qui chiffrerait "au fil de l'eau", c'est-à-dire de manière progressive, bloc par bloc, à mesure que

les données sont lues ou écrites. Cette méthode présente l'avantage de maintenir un coût constant en espace mémoire, car elle ne nécessite pas de charger l'intégralité du fichier en mémoire avant de le chiffrer.

Pour exécuter ce programme de chiffrement, il suffira d'utiliser les commandes suivantes²² :

```
rucrypt [options] < input > output  
rudecrypt [options] < input > output
```

Les options permettront de choisir l'algorithme utilisé²³.

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 [8]. Pour AES, 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.

$\text{rucrypt}(\text{data}) \rightarrow \text{RSA}(\text{clef symétrique}) \mid \text{nonce} \mid \text{data chiffré}$

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

23. 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

3.2 HSM

Le but du cryptographe est de créer des problèmes faciles à résoudre quand on a la clefs, mais beaucoup trop longs à casser sinon. 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 sécurisé pour garder nos clefs, i.e, il faut un coffre-fort moderne.

Pour cela, il existe les HSM (Hardware Security Module) qui sont des dispositifs physiques dédiés à la gestion, au stockage et à la protection des clefs 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.

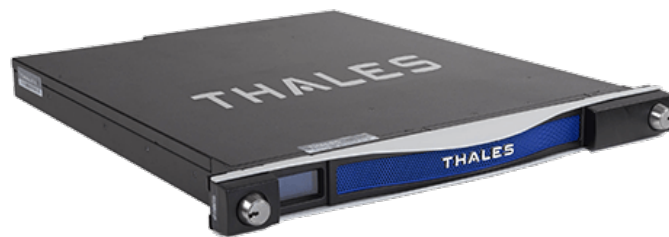


FIGURE 18 – HSM de Thales

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, de 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, de les documenter et de les tester (via un SoftHSM).

Pour communiquer avec un SoftHSM j'ai dû me plonger dans la documentation 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 [8], 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 tests, j'ai eu une grosse difficulté : je ne peux pas imposer la valeur d'une clé via mon programme (c'est une donnée protégée donc le softsm me l'interdit). Pour contourner ce problème, je lance un script bash dans mon code Rust qui crée la clé 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/softsm/libsoftsm2.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 tous les tests, et donc, grâce à mon tutoriel, je suis maintenant capable de remplacer OpenSSL par un programme "maison" en Rust.

3.3 Signature

Pour compléter mes connaissances en cryptographie, j'ai eu à implémenter deux algorithmes de signature. Comme pour le Tag dans AES GCM (voir [3.1.3](#)), 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 clé publique et une clé privée. Comme seul l'expéditeur connaît la clé privée, il est le seul à pouvoir signer le document.

3.3.1 RSA

Génération des clefs

Pour créer la clé publique et la clé 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 grands (au moins 3072 bits) mais e peut être petit. On remarque alors le grand défaut de RSA : les clefs sont extrêmement grande. 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 telle 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 cela, il faut 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.

3.3.2 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 vue de la taille gigantesque des clefs. Une alternative bien plus performante que RSA serait un cryptosystème s'appuyant sur les courbes elliptiques.

Courbe elliptique

Pour un corps 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 symétriques par rapport à l'axe des abscisses (donc si P est sur la courbe alors $-P$ aussi).

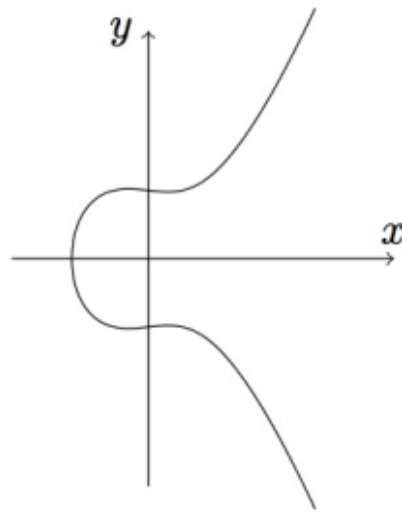


FIGURE 19 – Courbe $y^2 = x^3 - x + 9$ sur \mathbb{R}

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 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.

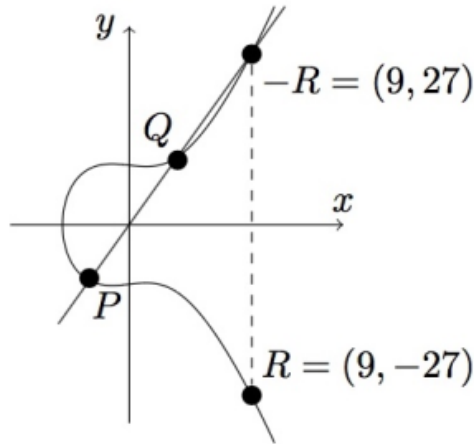


FIGURE 20 – $P + Q$ sur $y^2 = x^3 - x + 9$ sur \mathbb{R} avec $P = (-1, -3)$ et $Q = (1, 3)$

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 définie 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 3.3.2)
 2. Sinon, si $P = Q$ on utilise la méthode de la tangente (voir 3.3.2)
 3. Sinon ($P = -Q$) alors, on pose

$$P + Q = O$$

Voir [2] si vous voulez voir les formules complètes.

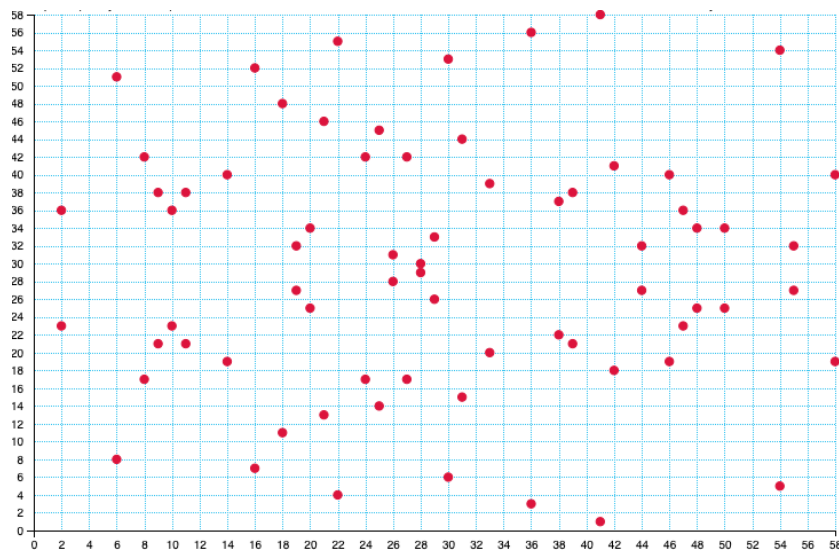


FIGURE 21 – Courbe $y^2 = x^3 - 6x + 2$ sur $\mathbb{Z}/59\mathbb{Z}$

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.

À 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 généré 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 ²⁴ un point de Ed25519 donné dans [3]

On a alors la clef privée k et la clef publique $\text{compress}(A)$ ²⁵. On remarque bien que contrairement à RSA, EdDSA permet d'avoir des clefs bien plus petites (256 bits chacune).

Signer avec Ed25519

Pour signer des données avec les courbes d'Edward on utilise l'algorithme EdDSA décrit plus précisément dans [3]. Il y a des paramètres généraux propres à l'algorithme EdDSA sur Ed25519, mais je ne les présente pas. Pour les voir, je vous invite à lire la section 5.1 de [3].

L'idée principale de cet algorithme est de générer à partir du message et de la clef privée un point $R = rB$. Pour calculer r , il faut calculer le hash modulo L ²⁶ des 256 derniers bits du hash de la clef privée $h = \text{hash}(k)$ concaténé 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 :

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

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

24. Pour des raisons de simplicité, calculatoire, on utilise une convention différente que celle présentée dans ici 3.3.2 mais elle est détaillée dans [3]

25. $\text{compress}()$ 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

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

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 cela, il doit décompresser²⁷ 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

$$\begin{aligned} xB = R + \tilde{h}A &\Rightarrow rB + \tilde{h}sB = R + \tilde{h}A \\ &\Rightarrow R + \tilde{h}A = R + \tilde{h}A \\ &\Rightarrow (\tilde{h} - \tilde{h})A = 0 \\ &\stackrel{28}{\Rightarrow} \tilde{h} = \tilde{h} \\ &\Rightarrow m = \tilde{m} \end{aligned}$$

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

3.3.3 Serveur signature

Pour proposer ces services de signature, il est nécessaire de déployer un serveur qui recevra des données et renverra leurs signatures correspondantes. En pratique, c'est ainsi que les algorithmes de signature sont utilisés pour permettre aux entreprises de contrôler l'accès aux clés sensibles. De plus, j'ai intégré à ce serveur la possibilité de connexion à un HSM (cf 3.2) si nous en disposons, afin de renforcer la sécurité. Conformément à la demande de mon responsable, les données doivent être au format JSON et présenter le format suivant :

```
{
  "id": "Vincent",
  "hash": "b16ed7d24b3ecb...",
  "methode": "sha512"
}
```

27. Algorithme assez intuitif, mais avec des astuces de calcul détaillées dans [3]. Il consiste à récupérer la coordonnée y , à vérifier qu'il existe bien des points avec cette coordonnée appartenant à la courbe, puis à choisir le bon point grâce au signe

Afin de simplifier l'utilisation et de réduire le nombre de signatures nécessaires auprès du serveur (ce qui peut être lent avec un HSM), nous envoyons au serveur une liste ordonnée de données (l'ordre permet de garantir l'unicité de la signature pour chaque liste). Cela permet d'éviter la signature individuelle de chaque donnée tout en garantissant l'authenticité de l'ensemble des données de la liste.

```
{
  "obj": [
    {
      "id": "Vincent",
      "hash":
        ↪ "b16ed7d24b3ecb...",
      "methode": "sha512"
    },
    {
      "id": "Alice",
      "hash":
        ↪ "6d201beeefb589...",
      "methode": "sha512"
    },
    {
      "id": "Bob",
      "hash":
        ↪ "cb872de2b8d250...",
      "methode": "sha512"
    }
  ]
}
```

En réalité, les données seront transmises à un client tiers, qui les hachera avant de les relayer au serveur. Cette approche offre une grande flexibilité, permettant ainsi à chaque partie prenante de configurer et d'utiliser son client de manière autonome.

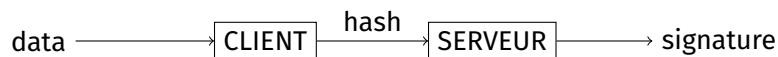


FIGURE 22 – architecture du serveur signature

3.4 Wireguard

Maintenant, on veut créer à l'aide de Wireguard un flux chiffré entre plusieurs machines pour permettre à ces machines de communiquer de manière simple et sécurisée.

3.4.1 Configuration

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

- une qui commence par `[Interface]`, elle contient les informations de la machine, comme la clef privée et l'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/21U4kYpS5KZlQfjOjddlvs7bgr0Z3ZHHBqF8lU=
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+tmfka8ukXcEV3vz5wtYinhZpLOEiA0e+gwhke28=
Address = 10.8.8.1

[Peer]
PublicKey = GignedMPFFe+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.

3.4.2 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 n'en a plus besoin une fois qu'elles sont dans les fichiers de configuration) et à restreindre l'accès aux fichiers de configuration.

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

3.5 Port knocking²⁹

3.5.1 Problème et solution

Un des problèmes avec un canal comme on les utilise avec WireGuard est qu'ils sont ouverts en permanence. Ainsi, une personne qui a accès à un canal l'aura toujours. Une solution à ce problème peut être le port knocking.

En effet, le port knocking est une manière d'ouvrir temporairement un canal. Son fonctionnement est le suivant :

29. En réalité, je ne fais pas du Port knocking mais du SPA (Single Packet Authorization) qui, à la différence du port knocking inclu de la cryptographie et est bien plus sécurisé

1. Le port de communication est fermé par défaut pour empêcher les accès non autorisés.
2. L'utilisateur envoie un paquet à un serveur UDP contenant une liste d'informations. Si le serveur arrive à identifier un utilisateur qui a le droit d'accéder au canal grâce à cette liste d'informations, alors il ouvre le port de communication.
3. Le port se referme automatiquement après un laps de temps défini.

3.5.2 Mon implémentation

J'ai dû mettre en place le port knocking sur un canal WireGuard. Pour faire ça, j'ouvre un canal sans aucun Peer (personne ne peut communiquer sur ce canal). Le fichier de configuration de ce canal ressemble donc à ça

```
[Interface]
PrivateKey = 6GK6HC9adKr00wGxU8jldcTTh+A29clxBB+D1vUzXU=
Address = 10.0.0.0/24
ListenPort = 51820
```

Ensuite, sur un serveur UDP je regarde si les paquets que je reçois contiennent une clef publique qui fait partie de la liste des clefs publiques autorisées (C'est comme ça que j'identifie un utilisateur). Si c'est le cas, j'édite le fichier de configuration pour rajouter le peer qui correspond à la clef publique et je mets à jour le canal. La configuration devient alors temporairement la suivante³⁰

```
[Interface]
PrivateKey = 6GK6HC9adKr00wGxU8jldcTTh+A29clxBB+D1vUzXU=
Address = 10.0.0.0/24
ListenPort = 51820

[Peer]
PublicKey = 4N2H/21U4kYpS5KZlQfj0jddlvs7bgr0Z3ZHHBqF8lU=
AllowedIPs = 10.0.0.1/32
```

Ensuite, j'ai rajouté un compteur qui supprime automatiquement le nouveau peer au bout d'un moment.

Dans la pratique, pour envoyer un paquet au serveur, je dois rentrer

```
nc -u -w 1 localhost 3000 < input.json
```

30. Par contre, le fichier de configuration lui est toujours le même et si je redémarre le canal, alors on retourne à la configuration initiale.

Avec un paquet au format json selon le standard suivant³¹

```
{
  "allowed_ip":
    ↪ "10.0.0.0/24",
  "key_pub": "GigedMPFFe+..."
}
```

Ensuite, le serveur regarde de temps en temps si l'utilisateur a toujours le droit d'utiliser le canal. Si ce n'est pas le cas, il le retire.

3.6 Signal

Dans l'idée de se débarrasser définitivement de RSA³², mon tuteur m'a chargé d'étudier et de recréer le protocole d'échange des clés de l'application Signal. Ce protocole a pour énorme avantage de permettre d'échanger les clefs sans avoir besoin au même moment des deux parties. On le qualifie de protocole d'échange asynchrone³³.

3.6.1 Extended Triple Diffie-Hellman (X3DH)

Toutes les informations qui m'ont permis d'écrire cette sous-sous-section sont directement tirées de [5].

X3DH est un protocole d'échange de clef asynchrone qui met en jeu trois parties : Alice, Bob et un serveur.

Dans notre exemple, Alice (A) veut discuter avec Bob (B). Pour faire ça, ils doivent avoir un secret commun. Notre problème est que Bob n'est pas forcément là quand Alice veut générer ce secret. Pour tenter de résoudre ce problème, le serveur servira à Alice et à Bob d'intermédiaire.

Clefs

Pour permettre cet échange, le serveur gardera pour chaque utilisateur un lot de clefs publiques^{34 35}. Pour l'utilisateur X (dans notre cas X = A (resp B) pour Alice (resp

31. Comme c'est un serveur UDP, on peut envoyer ce que l'on veut, mais on ne saura jamais s'il y a eu une erreur, c'est l'avantage de ce type de serveur.

32. On est capable de rendre RSA robuste à toute sorte d'attaque, mais cette robustesse à un prix énorme. Comme remarqué ici 3.3.1. En effet, la taille des clefs ne cessant d'augmenter, son utilisation est de plus en plus contraignante.

33. En plus d'être asynchrone, ce protocole n'utilise pas RSA qui était à peu près la seule solution pour faire de l'asynchrone avant

34. Des clefs pour courbes elliptiques comme Ed25519 3.3.2

35. les clefs secrètes correspondantes sont gardées sur la machine de l'utilisateur

Bob))

- $IK_{X, pub}$ la clef d'identité de X. Cette clef a une longue durée de vie (n'a pas besoin d'être changée souvent)
- $EK_{X, pub}$ une clef éphémère que X régénère à chaque fois qu'il utilise X3DH.
- $SPK_{X, pub}$ une clef signée (avec $IK_{X, priv}$) qui peut changer de temps en temps. La signature est aussi sur le serveur. Cette clef est surtout importante dans 3.6.2.
- $OPK_{X, pub}$ une clef à utilisation unique. Dans la pratique, X3DH n'a pas réellement besoin d'OPK mais cela rajoute une couche de sécurité. Chaque utilisateur envoie au serveur une grande liste d'OPK la renouvelle assez souvent.

À tout moment, n'importe qui peut demander au serveur de lui donner ce lot de clefs.

Pour notre exemple (où Alice commence l'échange), on utilisera les clefs IK_A , EK_A , IK_B , SPK_B et OPK_B

Premier message

Pour faire un échange de clefs avec X3DH, Alice demande au serveur les clefs publiques de Bob. Ensuite, elle vérifie que $SPK_{B, pub}$ correspond bien à la signature reçue. Si la vérification réussit, elle calcule

$$\begin{aligned} DH_1 &= DH (IK_{A, priv}, SPK_{B, pub}) \\ DH_2 &= DH (EK_{A, priv}, IK_{B, pub}) \\ DH_3 &= DH (EK_{A, priv}, SPK_{B, pub}) \\ DH_4 &= \begin{cases} \emptyset & \text{si Alice n'a pas } OPK_{B, pub} \\ DH (EK_{A, priv}, OPK_{B, pub}) & \text{sinon} \end{cases} \end{aligned}$$

et puis elle calcule le secret commun

$$SK = HKDF (DH_1 || DH_2 || DH_3 || DH_4)$$

Enfin, elle envoie à Bob le premier message qui contient :

- $IK_{A, pub}$
- $EK_{A, pub}$
- l'identifiant de $SPK_{B, pub}$ (et de $OPK_{B, pub}$ si elle en a utilisé une)
- le message chiffré³⁶ avec SK et avec comme associated data³⁷

$$AD = \text{compress}(IK_{A, pub}) || \text{compress}(IK_{B, pub})$$

36. typiquement avec AES GCM

37. la fonction compress est la même que celle utilisée dans 3.3.2

Réception du premier message

À l'aide des clefs dans le premier message et de ses propres clefs, Bob peut calculer SK. En effet, en calculant

$$\begin{aligned}DH_1 &= \text{DH}(\text{SPK}_{B, \text{priv}}, \text{IK}_{A, \text{pub}}) \\DH_2 &= \text{DH}(\text{IK}_{B, \text{priv}}, \text{EK}_{A, \text{pub}}) \\DH_3 &= \text{DH}(\text{SPK}_{B, \text{priv}}, \text{EK}_{A, \text{pub}}) \\DH_4 &= \begin{cases} \emptyset & \text{si Alice n'a pas utilisé OPK}_{B, \text{pub}} \\ \text{DH}(\text{OPK}_{B, \text{priv}}, \text{EK}_{A, \text{pub}}) & \text{sinon} \end{cases}\end{aligned}$$

On retrouve bien

$$SK = \text{HKDF}(DH_1 \parallel DH_2 \parallel DH_3 \parallel DH_4)$$

3.6.2 Double ratchet

Maintenant qu'Alice et Bob partagent un secret, ils peuvent communiquer de manière sécurisée. Pour garantir cette sécurité, l'application Signal utilise l'algorithme du double ratchet.³⁸ J'ai aussi programmé cet algorithme³⁹ mais il n'a pas énormément d'intérêt dans le cadre du stage. Je me passe donc de l'expliquer, mais tout est très bien expliqué ici [10]

3.7 Algorithmes post-quantiques

Pour finir mon stage, j'ai pu m'intéresser aux algorithmes post-quantiques. Ces nouveaux algorithmes sont vitaux, car nos algorithmes asymétriques, tels que RSA et ceux s'appuyant sur les courbes elliptiques, sont basés sur des problèmes mathématiques difficiles à résoudre pour les ordinateurs classiques, mais facile pour un ordinateur quantique⁴⁰. Par contre, on sait aussi que ces ordinateurs quantiques ne peuvent pas casser les algorithmes cryptographiques symétriques tels qu'AES car ils sont basés sur des opérations de blocs de bits, et que les ordinateurs quantiques ne peuvent pas effectuer ces opérations de manière significativement plus efficace que les ordinateurs classiques⁴¹.

38. Cela permet de ne pas compromettre toute la conversation et la conversation future si une clef a été récupérée par un attaquant malveillant

39. pas totalement fini

40. l'algorithme de Shor, découvert en 1994, permet à un ordinateur quantique de factoriser des nombres très grands en temps polynomial

41. On sait même que ces ordinateurs ne peuvent pas aller plus de deux fois plus vite que nos ordinateurs classiques sur ce type de problèmes

En ce moment, le [NIST](#) est en train de choisir le standard de la cryptographie post quantique. Le troisième round a déjà permis de fournir 4 algorithmes qui seront standardisés

<u>Public-Key Encryption/KEMs</u>	<u>Digital Signatures</u>
CRYSTALS–KYBER	CRYSTALS–Dilithium
	FALCON
	SPHINCS ⁺

FIGURE 23 – Algorithmes finalistes du round 3 à standardiser (voir [\[9\]](#))

Bien qu’il reste des candidats à la standardisation dans le round 4, il semblerait que la cryptographie reposant sur les [treillis](#) soit le futur. En effet, Kyber et Dilithium, les deux algorithmes de la société CRYSTALS qui seront standardisés reposent sur la difficulté à résoudre un problème dans les treillis algébriques.

Dans le papier que CRYSTALS a envoyé au NIST (voir [\[4\]](#)) on nous présente vaguement le problème qui viendra remplacer le problème du logarithme discret (obsolète avec l’algorithme de Shor) dans les algorithmes post-quantiques.

En partant de ce problème simple où il suffit d’inverser A et de multiplier par z pour trouver le résultat (voir [24](#))

$$\begin{pmatrix} A \end{pmatrix} \begin{pmatrix} y \end{pmatrix} = \begin{pmatrix} z \end{pmatrix} \bmod p$$

FIGURE 24 – Problème simple : connaissant (A, z) , trouver y

Ils ont trouvé un problème bien plus dur à résoudre qui, s’il était facile, aurait plein d’applications non cryptographiques majeures (voir [25](#)). Ce problème est en effet un problème sur les treillis, car pour chaque paire (y, e) on crée un nouveau treillis.

$$\begin{pmatrix} A \end{pmatrix} \begin{pmatrix} y \end{pmatrix} + \begin{pmatrix} e \end{pmatrix} = \begin{pmatrix} z \end{pmatrix} \bmod p$$

FIGURE 25 – Problème dur : connaissant (A, z) , trouver (y, e)

Le problème illustré figure 25 en termes de treillis revient à trouver le point le plus proche de l'origine⁴².

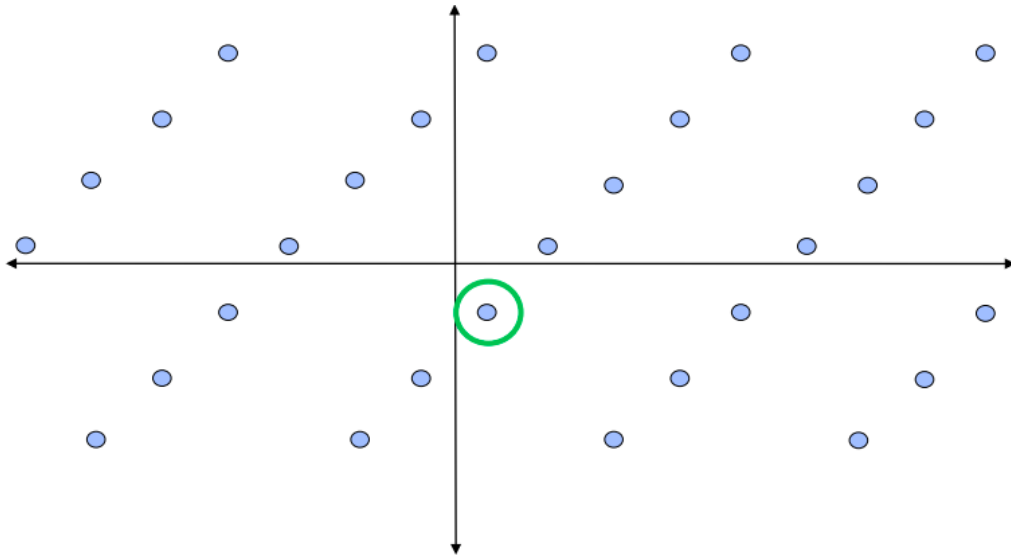


FIGURE 26 – Illustration d'un treillis issu du choix de (y, e)

L'intérêt de ce problème est que l'on ne connaît pas encore d'algorithme (comme celui de Shor pour le problème du logarithme discret) qui permettrait de résoudre ce problème avec un ordinateur quantique. Par contre, on n'a pas non plus de preuve que ce genre de problème est réellement dur pour un ordinateur quantique (contrairement aux algorithmes symétriques).

⁴². Ce problème est aussi appelé problème BDD (Bilinear Diffie-Hellman)

4 Conclusion

En conclusion, ce stage d'application au sein de Thalès a été une expérience extrêmement enrichissante pour moi. J'ai pu approfondir mes connaissances en cryptographie, découvrir de nouveaux algorithmes et mettre en pratique mes compétences en programmation en Rust.

Ce stage m'a également permis de mieux définir mon projet professionnel. J'ai pu prendre conscience de la manière dont fonctionne un grand groupe et, plus particulièrement, de ce à quoi ressemble le métier d'ingénieur informatique dans ce contexte. J'ai aussi pu me rendre compte que mon penchant pour la recherche me fait préférer d'autres domaines que la cryptographie, et où les mathématiques sont plus présentes^{43 44}.

Enfin, je tiens à remercier M. Mahmoud Chilali, mon tuteur, pour son accompagnement et ses conseils tout au long de ce stage. Je remercie également l'ensemble des employés de Thalès que j'ai pu rencontrer pour leur accueil chaleureux et leur disponibilité.

Références

- [1] « Advanced Encryption Standard (AES) ». In : (2001). URL : <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>.
- [2] Dan BONEH et Victor SHoup. *A Graduate Course In Applied Cryptography*. 2023.
- [3] S. JOSEFSSON et I. LIUSVAARA. « Edwards-Curve Digital Signature Algorithm (EdDSA) ». In : (2017).
- [4] « Lattice Cryptography ». In : (2018). URL : <https://csrc.nist.gov/CSRC/media/Presentations/Crystals-Dilithium/images-media/CRYSTALS-Dilithium-April2018.pdf>.
- [5] Trevor Perrin (editor) MOXIE MARLINSPIKE. *The X3DH Key Agreement Protocol*. 2016. URL : <https://signal.org/docs/specifications/x3dh/>.
- [6] *Présentation du Groupe*. 2024. URL : <https://www.thalesgroup.com/sites/default/files/2024-09/20240930%20Thales%20Group%20overview%202024%20-%20FR.pdf>.

43. comme la physique quantique par exemple

44. Les mathématiques y sont la base de tout en cryptographie, mais en pratique et d'après l'expérience que j'ai eue, il n'y a qu'un petit nombre de chercheurs qui vont à la base des modèles et utilisent réellement les outils qui m'intéressent

- [7] Mike ROSULEK. *The Joy of Cryptography*. 2021.
- [8] *Showcase of notable cryptography libraries developed in Rust*. URL : [cryptography.rs](#).
- [9] « Status Report on the Third Round of the NIST Post-Quantum Cryptography Standardization Process ». In : (2022). URL : <https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413-upd1.pdf>.
- [10] Moxie Marlinspike TREVOR PERRIN (EDITOR). *The Double Ratchet Algorithm*. 2016. URL : <https://signal.org/docs/specifications/doubleratchet/>.