REPUBLIQUE DU CAMEROUN
Paix-Travail-Patrie

UNIVERSITÉ DE YAOUNDÉ I

ECOLE NATIONALE SUPERIEURE POLYTECHNIQUE DE YAOUNDE

DEPARTEMENT DE GENIE INFORMATIQUE

REPUBLIC OF CAMEROON
Peace-Work-Fatherland

UNIVERSITY OF YAOUNDE I

NATIONAL ADVANCED SCHOOL OF ENGINEERING OF YAOUNDE

DEPARTMENT OF COMPUTER ENGINEERING

EXPOSE DANS LE CADRE DU COURS DE SCIENCE DE L'INFORMATION

THEME

FONCTIONS DE HACHAGE ET SIGNATURES NUMERIQUES

Fait par :
DOBA CHRISTIAN NDOUYANG
FOPA NDÉ MILENE LÉA
DJOKAM MFOMO FRANCK CHARES
NDAM MBOMBO RAMINE DELYAN
MBELLE MASSENGUE BOSTIN IVAN
EYENGA ZAMBO ANDY MIGUEL
YOGO KAM CALVIN KARIS

Sous l'encadrement de :
Dr HERVÉ TALE KALACHI, UY1

Année académique 2025-2026

Table des matières

Table des figures

INTRODUCTION GENERALE

À l'ère du numérique, la sécurité des données et l'authentification des échanges sont devenues des enjeux majeurs dans les systèmes d'information. Les fonctions de hachage et les signatures numériques constituent deux piliers fondamentaux de la cryptographie moderne, permettant d'assurer l'intégrité, l'authenticité et la non-répudiation des informations échangées.

Une fonction de hachage transforme un message de taille arbitraire en une empreinte numérique de taille fixe, appelée condensé. Lorsqu'elle est cryptographique, elle doit satisfaire des propriétés de sécurité strictes : résistance aux collisions, à la préimage et à la seconde préimage. Ces fonctions sont utilisées dans de nombreux contextes : protection des mots de passe, vérification d'intégrité, construction de structures de données vérifiables (arbres de Merkle), et surtout comme composant essentiel des signatures numériques.

Une signature numérique permet à un expéditeur d'attacher une preuve cryptographique à un message, vérifiable par quiconque dispose de la clé publique correspondante. Elle garantit à la fois l'identité de l'émetteur et l'intégrité du contenu. Des algorithmes comme RSA et DSA, reposant sur la cryptographie asymétrique et les infrastructures à clés publiques (PKI), mettent en œuvre ces mécanismes.

Ce document présente une étude détaillée des fonctions de hachage cryptographiques – incluant les constructions de Merkle-Damgård et éponge, ainsi que les algorithmes MD5 et SHA-256 – puis aborde les principes, techniques et enjeux des signatures numériques. Une application illustrative développée dans le cadre pédagogique vient concrétiser ces concepts.

I. FONCTIONS DE HACHAGE

1. Définition

Une fonction de hachage est un procédé qui transforme une donnée (ou message) de longueur arbitraire en une valeur de longueur fixe, appelée empreinte ou haché. Elle doit être rapide à calculer et, selon l'usage, respecter certaines propriétés supplémentaires.
On note généralement une fonction de hachage par h. Formellement :

$$h : \{0,1\}^* \longrightarrow \{0,1\}^n,$$

où $\{0,1\}^*$ désigne l'ensemble des chaînes binaires de longueur arbitraire et $\{0,1\}^n$ l'ensemble des chaînes binaires de longueur fixe n.

2. Motivations

Les fonctions de hachage sont utilisées dans divers contextes pour leurs propriétés :

— **Tables de hachage (structures de données).** Les tables de hachage servent à indexer et retrouver rapidement des clés en calculant une adresse à partir d'une clé. L'objectif est la performance (complexité moyenne proche de $O(1)$) ; les collisions sont tolérées et gérées par des stratégies (chaînage, open addressing). Ce rôle est purement algorithmique et diffère des exigences cryptographiques (non-inversibilité n'est pas requise).

— **Intégrité des données.** Une empreinte change de manière significative si le message est modifié. Les hachages permettent de détecter altérations ou corruptions lors du stockage ou de la transmission des données.

— **Authentification de messages (MAC).** En combinant un hachage et une clé secrète on obtient un code d'authentification de message. Cela prouve que le message vient d'une source possédant la clé et qu'il n'a pas été modifié.

— **Signature électronique.** Pour gagner en efficacité, on signe l'empreinte d'un document plutôt que le document entier. Le hachage doit garantir que toute modification du message invalide la signature.

— **Protection et stockage sécurisé des mots de passe.** Stocker des empreintes (souvent salées) au lieu des mots de passe en clair limite l'impact d'une fuite de la base de données. La résistance au calcul d'antécedent rend difficile la récupération des mots de passe originaux.

— **Dérivation et diversification de clés.** Les fonctions de hachage servent à dériver plusieurs clés à partir d'un secret maître de façon à limiter les conséquences d'une compromission partielle. Elles évitent que la compromise d'une clé révèle d'autres clés ou le secret initial.

— **Protocoles d'engagement.** Hacher une valeur permet de s'engager sur elle sans la révéler ; l'engagement est lié à la valeur mais reste caché jusqu'à la révélation. Cette propriété est utile dans les protocoles où l'on doit prouver ultérieurement l'authenticité d'un engagement.

— **Génération de nombres pseudo-aléatoires.** En itérant une fonction de hachage sur une graine ou en combinant une graine et un compteur, on obtient des séquences de bits utilisables comme source pseudo-aléatoire pour des usages cryptographiques.

— **Construction de structures vérifiables (ex. arbres de Merkle).** Les empreintes permettent de construire des preuves compactes d'appartenance et d'intégrité pour de grands ensembles de données, facilitant les vérifications distribuées et l'immutabilité (ex. blockchain, systèmes distribués).

— **Résistance aux attaques (préimage, seconde préimage, collisions).** Les propriétés cryptographiques classiques (préimage, seconde préimage, collision) forment la base des garanties de sécurité : elles empêchent respectivement la récupération d'un message depuis son haché, la construction d'un autre message ayant le même haché, et la recherche efficace de paires de messages identiques en empreinte.

3. Fonctions de hachage cryptographiques

Une fonction de hachage est dite cryptographique lorsqu'elle possède certaines propriétés de sécurité.

3.1. Propriétés classiques

Trois propriétés, dites classiques, sont exigées de toute fonction de hachage cryptographique : résistance aux collisions, résistance au calcul d'antécédent et résistance au calcul de second antécédent.
Pour définir ces propriétés, posons r $\in \mathbb{N}^*$ et considérons la fonction de hachage cryptographique h : $\{0,1\}^r \to \{0,1\}^n$.

3.1.1. Résistance aux collisions (Collision Resistance)

La fonction h résiste de façon optimale aux collisions si on ne possède pas d'algorithme capable de produire un couple de messages $(x, x') \in \{0,1\}^r\{0,1\}^r$, tel que $x' \neq x$ et $h(x') = h(x)$ avec une complexité meilleure que $O(2^{n/2})$ opérations. Un tel couple forme une **collision** pour la fonction de hachage h.

3.1.2. Résistance au calcul d'antécédent (Preimage Resistance)

Soit un message m tiré aléatoirement dans $\{0,1\}^r$ (ensemble de définition de la fonction h) on pose $y = h(m)$. La fonction h résiste de façon optimale au calcul

d'antécédent si on ne possède pas d'algorithme capable de produire un antécédent $x \in \{0,1\}^r$ tel que $h(x) = y$, avec une complexité meilleure que $O(2^n)$ opérations. Pour cette propriété, on parle aussi de **fonction à sens unique**(One Way).

3.1.3. Résistance au calcul de second antécédent (Second Preimage Resistance)

Soit un message x tiré aléatoirement dans $\{0,1\}r$. La fonction h résiste de façon optimale au calcul de second antécédent si on ne possède pas d'algorithme capable de produire un second antécédent $x' \in \{0,1\}^r$ tel que $x' \neq x$ et $h(x') = h(x)$, avec une complexité meilleure que $O(2^n)$ opérations.

3.2. Construction des fonctions de hachage

Une approche naturelle pour construire les fonctions de hachage consiste à définir une fonction qui **opère sur un domaine de taille fixe** puis d'étendre ce domaine grâce à un **extenseur de domaine**. Dans le cadre de cette approche s'inscrivent les fonctions de hachage cryptographiques dites *itératives*, qui sont aussi les plus répandues et qui nous intéressent ici. Celles-ci combinent une fonction ou primitive, opérant sur un domaine de taille fixe, et un extenseur pour étendre ce domaine. Dans les pages qui suivent, nous présentons l'extenseur de domaine d'après la construction de Merkle-Damgård, puis celui appelé éponge cryptographique. Par la suite viennent des descriptions de la construction des fonctions de hachage MD5, SHA-1 et SHA-256.

3.2.1. Extenseur Merkle-Damgård

La construction de Merkle-Damgård ou construction MD a été découverte indépendamment par Merkle et Damgård en 1989. La plupart des fonctions de hachage connues, telles que MD4, MD5, SHA-0, SHA-1, RIPEMD-160, etc., suivent la méthode itérative MD.
Un élément fondamental dans la construction MD est la *fonction de compression*, constituant la primitive qui opère sur un domaine de taille fixe et qui prend donc une entrée de taille fixe et produit une sortie de taille fixe également.
Une fonction de compression f prend en entrée un bloc de message de taille r et une *variable de chaînage* de taille n ; elle se définit alors comme : $f : \{0,1\}^r \ 0,1^n \rightarrow \{0,1\}^n$. une variable de chaînage est une variable contenant successivement des chaînes de bits fournies à f lors des itérations.
Soit $h : \{0,1\}^k \rightarrow \{0,1\}^n$, où $k \in \mathbb{N}^*$ (quelconque), une fonction de hachage construite sur la méthode MD par itération de la fonction de compression f, et M le message à hacher (représenté en binaire) et $|M|$ sa longueur. La construction se fait comme suite :

- **Étape 1 - Bourrage du message (padding en anglais)** : il consiste à ajouter (par concaténation) à M le bit 1 suivi d'un nombre m de bit 0, puis ajouter (toujours par concaténation) la représentation binaire de la longueur de M, un nombre de bits est réservé d'avance pour cela. m est choisi de telle sorte que le message M' (représenté en binaire) obtenu après bourrage ait une longueur *multiple* de r, taille d'un bloc traité par f. C'est le principe de **bourrage de Merkle-Damgård**.

- **Étape 2** : on subdivise M' en t blocs de taille r : $t = |M'|/r$. Il faut noter qu'à l'étape précédente, si $|M|$ est déjà multiple de r, on ajoute ici à la fin

un nouveau bloc pour faire le bourrage.

- **Etape 3 - calcul itératif de h(M), haché de M** : ce calcul est décrit par

$$\begin{cases} VC_0 = VI \\ VC_i = f(VC_{i-1}, M_{i-1}), i = 1, ..., t \\ h(M) = VC_t \end{cases}$$

où f est la fonction de compression de h, VC_i est la variable de chaînage pour l'étape i, avec VC_0 une valeur prédéfinie ou valeur initiale VI intrinsèquement liée à une implémentation algorithmique spécifique. **La dernière variable de chaînage constitue le haché de** M.

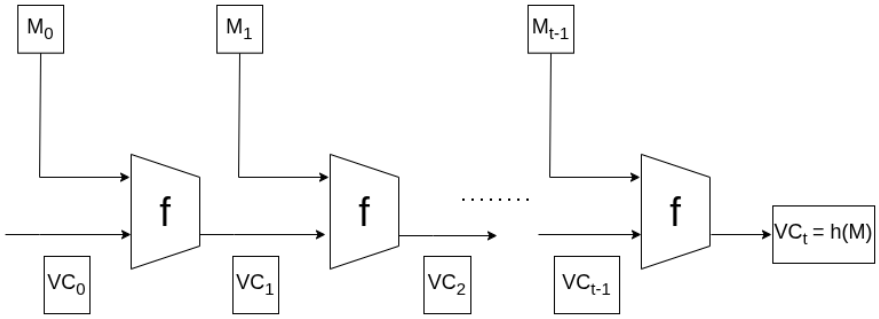

FIGURE 1 – Illustration de l'extenseur de Merkle-Damgård

Il est montré que pour une construction MD, une fonction de hachage hérite les propriétés de sécurité de la fonction de compression f utilisée.

3.2.2. Extenseur éponge

A. Motivation

L'algorithme de Merkle-Damgård, bien qu'efficace, présente certaines vulnérabilités (attaques par extension, multi-collisions). Les éponges cryptographiques ont été conçues pour :

— Éviter ces vulnérabilités structurelles
— Offrir plus de flexibilité (sortie de longueur variable)
— Fournir une construction proche d'un oracle aléatoire

B.Métaphore de l'éponge

Le nom "éponge" provient de l'analogie avec une éponge physique :
— **Phase d'absorption** : l'éponge "absorbe" le message
— **Phase d'essorage** : l'éponge est "pressée" pour produire la sortie

C. État interne

L'extenseur éponge dispose d'un état interne divisé en deux parties distinctes :
— **La partie externe (rate)** : Cette partie de l'état interagit directement avec les données d'entrée. Elle a une taille de r bits et sert d'interface entre le message et l'état interne.
— **La partie interne (capacity)** : Cette partie de l'état ne touche jamais directement les données d'entrée ou de sortie. Elle a une taille de c bits et garantit la sécurité de la construction en maintenant une information cachée qui ne peut pas être directement manipulée par un attaquant.

La taille totale de l'état est donc $b = r + c$ bits.

D. Construction

D.1. Fonction de permutation

Au cœur de la construction se trouve une fonction de permutation **p** qui opère sur l'état complet de b bits :

$$p : \{0,1\}^b \to \{0,1\}^b$$

Cette permutation doit être :
— Bijective (inversible)
— Pseudo-aléatoire
— Efficace à calculer

D.2. Phase d'absorption

La phase d'absorption est la première étape du processus éponge. Elle consiste à intégrer progressivement le message d'entrée dans l'état interne de la fonction.

a) Initialisation

Au début de la phase d'absorption, l'état interne est initialisé à zéro. Tous les bits de l'état, qu'ils appartiennent à la partie externe ou à la partie interne, sont mis à zéro. Cette initialisation uniforme garantit que le processus est déterministe et reproductible.

b) Préparation du message

Le message d'entrée doit d'abord être préparé pour le traitement. Cette préparation comprend l'ajout d'un padding (bourrage) au message si nécessaire, afin que sa longueur soit un multiple exact de r bits. Le message paddé est ensuite découpé en blocs de taille r bits chacun. Si le message original fait par exemple 250 bits et que $r = 64$ bits, on ajoutera du padding pour arriver à 256 bits, ce qui donnera 4 blocs de 64 bits.

c) Traitement itératif des blocs

Pour chaque bloc du message, le processus suivant est répété :
— **Étape 1 : Combinaison XOR** Le bloc courant est combiné avec la partie externe de l'état interne en utilisant l'opération XOR (ou exclusif bit à bit). Cette opération mélange les données du message avec l'état actuel. Seule la partie externe de l'état est modifiée lors de cette étape ; la partie interne (capacity) reste inchangée directement.
— **Étape 2 : Application de la permutation** Une fois le bloc "XORé" avec la partie externe, une fonction de permutation est appliquée à la totalité de l'état interne. Cette fonction de permutation est une transformation bijective qui mélange et diffuse les bits à travers tout l'état, incluant à la fois la partie externe et la partie interne. C'est cette permutation qui assure que la partie capacity est indirectement influencée par les données d'entrée, tout en restant cachée.
La fonction de permutation doit posséder de bonnes propriétés cryptographiques : elle doit assurer une diffusion et une confusion importantes pour que chaque bit de sortie dépende de manière complexe de tous les bits d'entrée.

d) Itération complète

Ce processus de XOR suivi d'une permutation est répété pour chacun des blocs du message. À la fin du traitement du dernier bloc, l'état interne contient une empreinte cryptographique du message complet, et la phase d'absorption est terminée.

D.3. Phase d'essorage (Squeezing)

a) Principe général

La phase d'essorage est la seconde étape du processus éponge. Elle consiste à extraire de l'état interne la sortie de la longueur désirée. Cette phase tire également son nom de l'analogie avec l'éponge : après avoir absorbé le message, on "essore" l'éponge pour en extraire le condensat qui constituera le haché.

b) Point de départ

La phase d'essorage commence avec l'état interne tel qu'il se trouve à la fin de la phase d'absorption. Cet état contient toute l'information nécessaire pour produire la sortie, encodée de manière diffusée à travers les parties externe et interne.

c) Extraction de la sortie

Le processus d'extraction se déroule comme suit :

Première extraction : on extrait les r premiers bits de l'état, c'est-à-dire la totalité de la partie externe. Ces bits constituent le premier bloc de la sortie. Comme pour l'absorption, seule la partie externe est accessible ; la partie capacity reste cachée et n'est jamais directement lue.

Vérification de la longueur : si la longueur de sortie désirée est inférieure ou égale à r bits, alors ces r premiers bits suffisent (éventuellement tronqués à la longueur exacte demandée), et la phase d'essorage est terminée.

Extractions supplémentaires : si plus de r bits sont nécessaires, le processus continue. On applique à nouveau la fonction de permutation à l'ensemble de l'état interne. Cette permutation transforme l'état et prépare les prochains bits de sortie. Après cette permutation, on extrait à nouveau r bits de la partie externe. Ces nouveaux bits constituent le deuxième bloc de sortie.

Itération : ce processus se répète autant de fois que nécessaire : permutation de l'état complet, puis extraction de r bits, jusqu'à ce que la longueur de sortie totale désirée soit atteinte. Si la longueur finale désirée n'est pas un multiple exact de r, le dernier bloc extrait est simplement tronqué pour obtenir exactement le nombre de bits demandé.

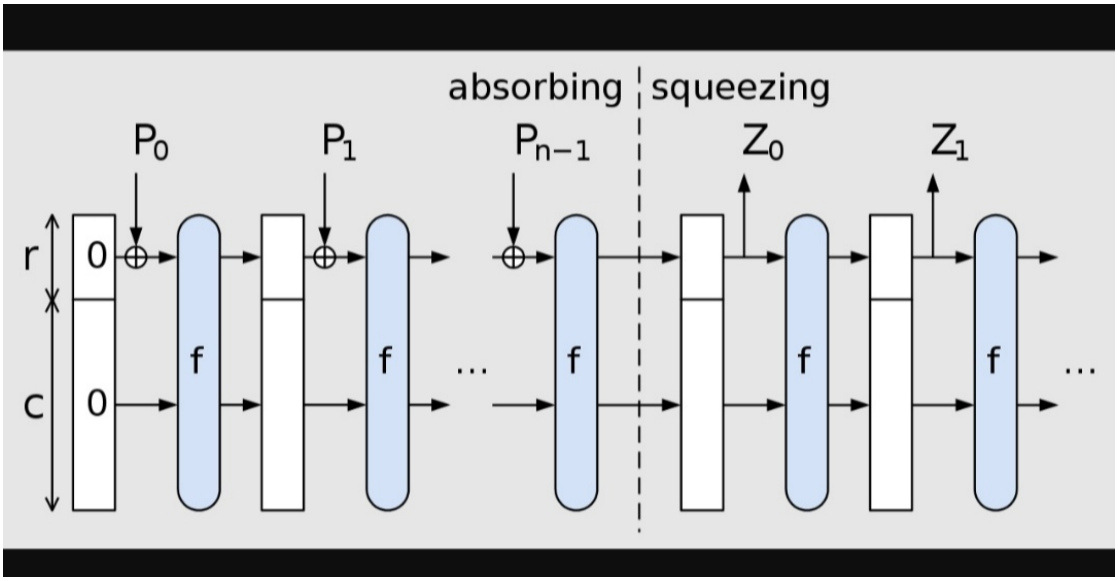

FIGURE 2 – Illustration de l'extenseur éponge

3.2.3. Fonction MD5

La fonction MD5 (Message Digest Algorithm 5), standardisée dans la RFC 1321, produit un condensé de 128 bits à partir d'un message de longueur arbitraire. Elle suit la construction de Merkle-Damgård et opère sur des mots de 32 bits. Les données sont traitées par blocs de 512 bits.

A. Terminologie : un mot est une chaîne de 32 bits, représentable en hexadécimal par 8 chiffres. On utilise la convention *little-endian* : le mot de poids faible est placé à gauche dans les représentations internes.

Les opérations logiques se font bit à bit sur les mots de 32 bits.

B. Opérations sur les mots : soient X, Y, Z trois mots de 32 bits. MD5 utilise les opérations suivantes :

- $X \wedge Y$: "ET" logique bit-à-bit.
- $X \vee Y$: "OU" logique bit-à-bit.
- $X \oplus Y$: "OU exclusif" logique.
- $\neg X$: négation bit-à-bit.
- $X + Y = (x + y) \bmod 2^{32}$ où x, y sont les entiers représentés par X, Y.
- $\mathrm{ROTL}_n(X)$: rotation à gauche de n bits.

MD5 définit également quatre fonctions logiques non linéaires :

$$\begin{aligned} F(X,Y,Z) &= (X \wedge Y) \vee (\neg X \wedge Z), \\ G(X,Y,Z) &= (X \wedge Z) \vee (Y \wedge \neg Z), \\ H(X,Y,Z) &= X \oplus Y \oplus Z, \\ I(X,Y,Z) &= Y \oplus (X \vee \neg Z). \end{aligned}$$

C. Constantes utilisées : MD5 utilise une table de 64 constantes $T[i]$ définies par :

$$T[i] = \lfloor 2^{32} \cdot |\sin(i)| \rfloor, \qquad 1 \leq i \leq 64,$$

où l'indice i est exprimé en radians.

Chaque itération utilise également une rotation cyclique à gauche d'un nombre de bits dépendant du tour :

$$s = \begin{cases} \{7, 12, 17, 22\} & \text{(tour 1)}, \\ \{5, 9, 14, 20\} & \text{(tour 2)}, \\ \{4, 11, 16, 23\} & \text{(tour 3)}, \\ \{6, 10, 15, 21\} & \text{(tour 4)}. \end{cases}$$

D. Étapes de calcul du haché

a) Bourrage du message MD5 applique un bourrage similaire à celui de Merkle-Damgård :

1. concaténer le bit 1 ;
2. concaténer k zéros, où k est choisi pour que

$$(L + 1 + k) \equiv 448 \pmod{512};$$

3. concaténer la longueur initiale L du message sur 64 bits *little-endian*.

Le message obtenu est découpé en blocs M_i de 512 bits.

b) Initialisation de la variable de chaînage La variable de chaînage initiale est constituée de quatre mots de 32 bits :

$$\begin{aligned} A_0 &= \texttt{67452301}, \\ B_0 &= \texttt{efcdab89}, \\ C_0 &= \texttt{98badcfe}, \\ D_0 &= \texttt{10325476}. \end{aligned}$$

c) Traitement itératif des blocs M_i Pour chaque bloc de 512 bits, on définit d'abord 16 mots de 32 bits :

$$M_i = (X_0, X_1, \ldots, X_{15}),$$

extraits en *little-endian*.
Les quatre mots de travail sont initialisés ainsi :

$$A = A_{i-1}, \quad B = B_{i-1}, \quad C = C_{i-1}, \quad D = D_{i-1}.$$

Le traitement consiste en 64 itérations divisées en quatre tours.

Tour 1 : $0 \leq j \leq 15$

$$A \leftarrow B + \mathrm{ROTL}_{s_j}(A + F(B, C, D) + X_j + T[j]).$$

Tour 2 : $16 \leq j \leq 31$

$$A \leftarrow B + \mathrm{ROTL}_{s_j}(A + G(B, C, D) + X_{(5j+1) \bmod 16} + T[j]).$$

Tour 3 : $32 \leq j \leq 47$

$$A \leftarrow B + \mathrm{ROTL}_{s_j}(A + H(B, C, D) + X_{(3j+5) \bmod 16} + T[j]).$$

Tour 4 : $48 \leq j \leq 63$

$$A \leftarrow B + \mathrm{ROTL}_{s_j}(A + I(B, C, D) + X_{7j \bmod 16} + T[j]).$$

Après chaque opération, les registres sont permutés cycliquement :

$$(A, B, C, D) \leftarrow (D, A, B, C).$$

d) Mise à jour de la variable de chaînage À la fin des 64 itérations :

$$\begin{aligned} A_i &= A_{i-1} + A, \\ B_i &= B_{i-1} + B, \\ C_i &= C_{i-1} + C, \\ D_i &= D_{i-1} + D. \end{aligned}$$

e) Production du haché Le haché final MD5 est la concaténation :

$$\mathrm{MD5}(M) = A_n \,\|\, B_n \,\|\, C_n \,\|\, D_n$$

exprimée en hexadécimal, selon la convention *little-endian*.

E. Aspects de sécurité :

- **Etat de la sécurité :** MD5 est cassé pour la collision. Des techniques de cryptanalyse différentielle permettent de trouver des collisions en un temps très faible comparé à $2^{128/2}$.
- **Vulnérabilités notables :**
  - collisions pratiques et rapides (exploitées historiquement pour forger des certificats, créer des binaires différents partageant le même MD5, etc.) ;
  - susceptibilité aux attaques de *chosen-prefix* — facilitation de forges ciblées ;
  - vulnérable à l'extension de longueur (Merkle–Damgård).
- **Conséquence :** MD5 *ne doit plus* être utilisé pour des garanties d'intégrité ou d'authenticité ; il est obsolète pour les signatures numériques et certificats.

3.2.4 Fonction SHA-256

Les fonctions de hachage SHA (Secure Hash Algorithm) font partie de la norme fédérale de traitement de l'information des Etats-unis d'Amérique.
La construction de la fonction SHA-2 est basée sur l'extenseur de Merkle-Damgård. Commençons par définir la terminologie et certaines opérations nécessaires à cette construction et que nous utiliserons dans la suite.
La fonction SHA-256 calcule un condensé ou haché de 256 bits pour un message de longueur L avec $0 \leq L < 2^{64}$. Elle opère sur des mots de 32 bits et traite les données par blocs de 512 bits.

A. Terminologie

Un **bit** est un chiffre binaire, élément de $\{0, 1\}$.
Un **chiffre hexadécimal** est un élément de $\{0, 1, ..., 9, A, ..., F\}$. On peut le voir comme la représentation d'une chaîne de 4 bits.

Un **mot** est une chaîne de 32 bits ou 64 bits pouvant respectivement être représentées par une suite de 8 ou 16 chiffres hexadécimaux.

La **convention "big-endian"** est celle qui exige que le bit de plus grand poids soit écrit à l'extrême gauche dans une chaîne de bits. C'est elle que nous utiliserons dans la suite.

On peut représenter un entier $\in [0, 2^{32} - 1]$ comme un mot de 32 bits. Il en est de même pour un entier $\in [0, 2^{64} - 1]$, représentable comme mot de 64 bits.

Un **bloc** est une chaîne de 512 bits ou 1024 bits. Il peut être représenté comme une suite de mots.

B. Opérations sur les mots

Soit X, Y, Z des mots de 32 bits.

- $X \wedge Y$ est l'opération bit-à-bit du "ET" logique de X et Y.
- $X \vee Y$ est l'opération bit-à-bit du "OU" logique de X et Y.
- $X \oplus Y$ est l'opération bit-à-bit du "OU exclusif" logique de X et Y.
- $\neg X$ est l'opération bit-à-bit du "NON" logique de X.
- $X + Y$: si x et y sont les entiers représentés par X et Y respectivement, alors poser $z = (x + y)\ mod\ 2^{32}$
  et considérer Z sa représentation sous forme de mot. Ainsi on définit $X+Y = Z$.
- $SHR^n(X) = X >> n$ et $SHL^n(X) = X << n$ sont respectivement les opérations **décalage (logique) à droite de n bits de X**, et **décalage (logique) à gauche de n bits de X**, où $0 \leq n < w$.
- $ROTR^n(X) = (X >> n) \ \vee \ (X << (w - n))$ c'est l'opération **rotation à droite de n bits de X**.
- $ROTL^n(X) = (X << n) \ \vee \ (X >> (w - n))$ c'est l'opération **rotation à gauche de n bits de X**.

C. Fonctions et Constantes utilisées

Dans l'algorithme du SHA-2, les fonctions logiques suivantes seront utilisées :

- $\mathrm{CH}(x, y, z) = (x \wedge y) \oplus (\neg x \wedge z)$
- $\mathrm{MAJ}(x, y, z) = (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)$
- $\mathrm{BSIG}_0(x) = \mathrm{ROTR}^2(x) \oplus \mathrm{ROTR}^{13}(x) \oplus \mathrm{ROTR}^{22}(x)$
- $\mathrm{BSIG}_1(x) = \mathrm{ROTR}^6(x) \oplus \mathrm{ROTR}^{11}(x) \oplus \mathrm{ROTR}^{25}(x)$
- $\mathrm{SSIG}_0(x) = \mathrm{ROTR}^7(x) \oplus \mathrm{ROTR}^{18}(x) \oplus \mathrm{SHR}^3(x)$
- $\mathrm{SSIG}_1(x) = \mathrm{ROTR}^{17}(x) \oplus \mathrm{ROTR}^{19}(x) \oplus \mathrm{SHR}^{10}(x)$

Aussi, SHA-256 utilise une table de 64 constantes de 32-bits $K_0, \ldots, K_{63}$. Elles sont obtenues en prenant les premiers 32 bits des parties fractionnaires des racines cubiques des 64 premiers nombres premiers. Ci-après leur table en hexadécimal, de gauche à droite :

```
428a2f98 71374491 b5c0fbcf e9b5dba5
3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3
72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc
2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7
c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13
650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3
d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5
391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208
90befffa a4506ceb bef9a3f7 c67178f2
```

FIGURE 3 – Table des 64 constantes du SHA-256

D. Etapes de calcul du haché

La variable de chaînage VC se présente comme un ensemble de 8 mots de 32 bits VC[0], VC[1], ..., VC[7], pour un total de 256 bits. Celle-ci voit sa valeur VC_i se mettre à jour au cours de plusieurs itérations d'ordre i. Chacune de ces valeurs, à part la première et la dernière, est appelée haché intermédiaire. La dernière valeur produit le haché par concaténation.

a) Bourrage ou padding

Pour le SHA-256, on bourre le message M de longueur L bits, d'après le principe donné par l'extenseur Merkle-Damgard :

1. concaténer le bit `1`,
2. concaténer K zéros, où K est le plus petit entier ≥ 0 tel que $(L + 1 + K) \bmod 512 = 448$; on réserve en fait 64 bits.
3. concaténer la représentation binaire sur 64 bits de L (encodé big-endian).

Après bourrage on découpe le message obtenu en n blocs M_i de 512 bits.

b) Initialisation de la variable de chaînage

Les mots de la variable de chaînage initiale VC_0, de taille 32 bits et en hexadécimal sont :

$$VC_0[0] = \texttt{6a09e667}, \quad VC_0[1] = \texttt{bb67ae85},$$
$$VC_0[2] = \texttt{3c6ef372}, \quad VC_0[3] = \texttt{a54ff53a},$$
$$VC_0[4] = \texttt{510e527f}, \quad VC_0[5] = \texttt{9b05688c},$$
$$VC_0[6] = \texttt{1f83d9ab}, \quad VC_0[7] = \texttt{5be0cd19}.$$

Ces mots sont obtenus en prenant les premiers 32 bits des parties fractionnaires des racines carrées des 8 premiers nombres premiers.

c) Traitement itératif des n blocs M_i

On considère successivement les blocs $M_i, 1 \leq i \leq n$. Le processus utilise 8 mots de travail de 32 bits, soit a, b, c, d, e, f, g *et* h, une suite de 64 mots de 32 bits, la variable de chaînage et enfin deux mots tampon T_1 et T_2.
Considérons un bloc M_i.

— **Etape 1 - Construction d'une suite de mots** $(W_t)_{0 \leq t \leq 63}$
Le bloc M_i de 256 bits est découpé en 16 mots de 32 bits à partir du bit de plus fort poids, constituant les 16 premiers termes $W_0, \cdots, W_{15}$ de la suite (W_t).
Et pour t allant de 16 à 63 :

$$W_t = \big(\mathrm{SSIG}_1(W_{t-2}) + W_{t-7} + \mathrm{SSIG}_0(W_{t-15}) + W_{t-16}\big) \bmod 2^{32}.$$

— **Etape 2 - Initialisation des mots de travail**

$$\begin{aligned} a &= VC_{i-1}[0], \quad b = VC_{i-1}[1], \\ c &= VC_{i-1}[2], \quad d = VC_{i-1}[3], \\ e &= VC_{i-1}[4], \quad f = VC_{i-1}[5], \\ g &= VC_{i-1}[6], \quad h = VC_{i-1}[7]. \end{aligned}$$

— **Etape 3 - calcul du haché intermédiaire** VC_i
Pour $t = 0$ **à** 63 :

$$\begin{aligned} T_1 &= h + \mathrm{BSIG}_1(e) + \mathrm{CH}(e, f, g) + K_t + W_t \\ T_2 &= \mathrm{BSIG}_0(a) + \mathrm{MAJ}(a, b, c) \\ h &\leftarrow g \\ g &\leftarrow f \\ f &\leftarrow e \\ e &\leftarrow d + T_1 \\ d &\leftarrow c \\ c &\leftarrow b \\ b &\leftarrow a \\ a &\leftarrow T_1 + T_2. \end{aligned}$$

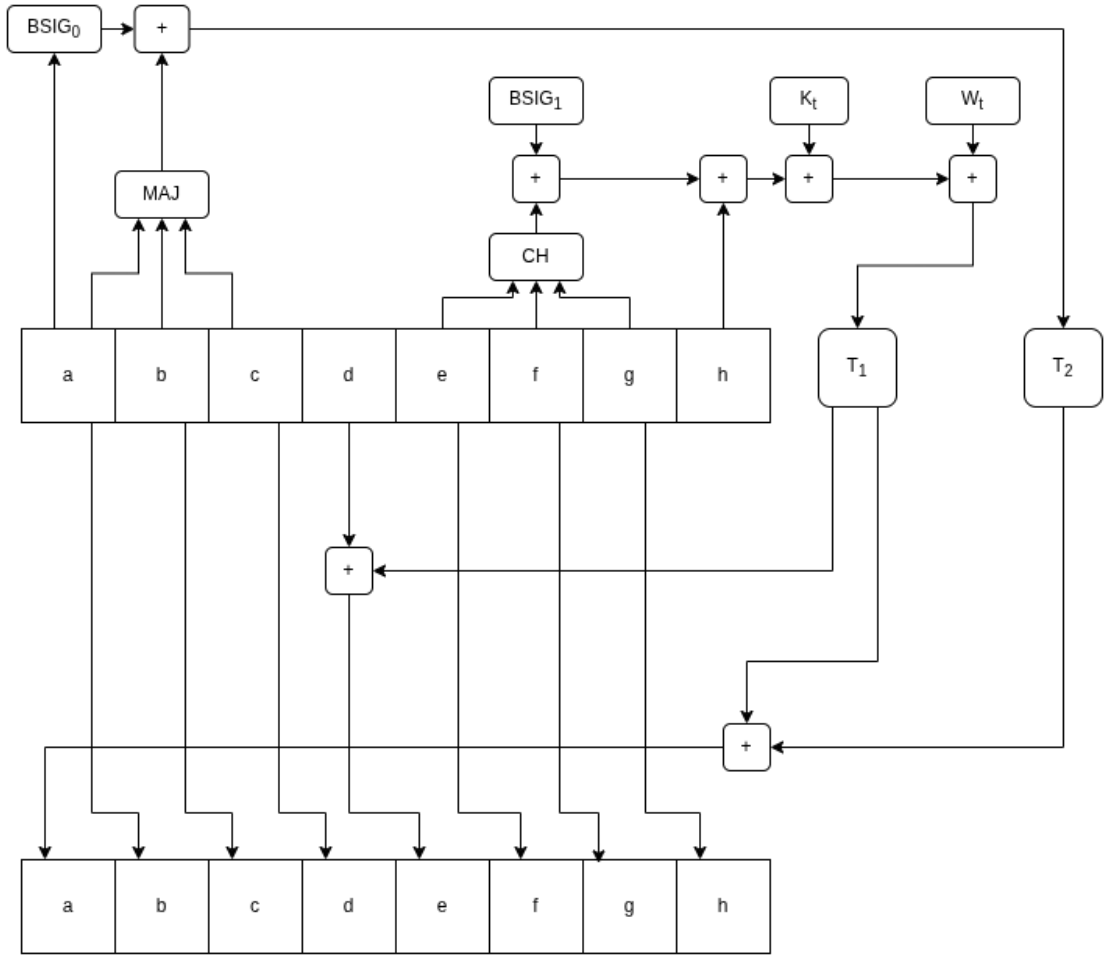

FIGURE 4 – Exécution d'une des 64 itérations pour le bloc M_i

Après les 64 itérations, on met à jour la valeur de hachage intermédiaire ou variable de chaînage :

$$\begin{aligned} VC_i[0] &= VC_{i-1}[0] + a, \quad VC_i[1] = VC_{i-1}[1] + b \\ VC_i[2] &= VC_{i-1}[2] + c, \quad VC_i[3] = VC_{i-1}[3] + d \\ VC_i[4] &= VC_{i-1}[4] + e, \quad VC_i[5] = VC_{i-1}[5] + f \\ VC_i[6] &= VC_{i-1}[6] + g, \quad VC_i[7] = VC_{i-1}[7] + h. \end{aligned}$$

Après traitement du dernier bloc, le haché SHA-256 est la concaténation des huit mots $VC_n[0] \parallel VC_n[1] \parallel \ldots \parallel VC_n[7] = h(M)$, en hexadécimal, convention big-endian.

E. Aspects de sécurité :

Propriétés de sécurité attendues

- **Résistance aux collisions :** complexité théorique $\approx 2^{128}$ (attaque du *birthday*).
- **Préimage / seconde-préimage :** complexités attendues $\approx 2^{256}$ (préimage) et $\approx 2^{256}$ (seconde-préimage).
- **Robustesse pratique :** à ce jour, aucune attaque pratique ne compromet la sécurité effective de SHA-256 pour les usages courants (intégrité, signatures).

Vecteurs d'attaque pertinents

Collision : attaques théoriques réduisant légèrement les marges existent, mais aucune collision pratique n'a été démontrée publiquement pour SHA-256.

Préimage / seconde-préimage : pas d'attaques pratiques connues réduisant significativement 2^{256}.

Extension de longueur : la construction Merkle–Damgård permet l'attaque d'extension de longueur : privilégier des constructions (ex. HMAC) qui la mitigent.

Attaques ciblées / chosen-prefix : plus coûteuses que pour MD5/SHA-1 ; aucune méthode pratique bien établie pour SHA-256 à ce jour.

Usage inapproprié : utilisation directe pour le hachage de mots de passe (trop rapide) ou inclusion naïve dans des protocoles peut créer des failles opérationnelles.

II. SIGNATURES NUMERIQUES

1. Définition et généralités

Une signature numérique est définie comme un code cryptographique qu'un émetteur attache à un message numérique et permettant d'authentifier l'origine et garantir l'intégrité du message.
Le processus implique qu'un expéditeur peut signer un message en utilisant un algorithme de génération de signature, tandis que tout destinataire peut vérifier cette signature en utilisant un algorithme public de vérification qui prend en entrée le message reçu.

1.1. Définition formelle

Un schéma de signature est défini mathématiquement comme un quintuplet $(P, S, K, \text{sig}, \text{ver})$ où :

- P : espace des messages possibles
- S : espace des signatures possibles
- K : espace des clés possibles
- $\text{sig}_K : P \to S$: algorithme de signature (secret)
- $\text{ver}_K : P \times S \to \{\text{vrai}, \text{faux}\}$: algorithme de vérification (public)

La propriété essentielle s'exprime par :

$$\forall x \in P, \forall K \in K, \text{ver}_K(x, \text{sig}_K(x)) = \text{vrai}$$

1.2. Caractéristiques fondamentales des signatures numériques

a) Propriétés essentielles

Une signature numérique digne de confiance doit posséder les propriétés suivantes :

- Authentification de l'auteur ainsi que de la date et l'heure de la signature
- Vérification de l'intégrité du contenu au moment de la signature
- Vérifiabilité par des tiers pour résoudre les litiges potentiels
- Dépendance du motif de bits au message signé, garantissant l'unicité de chaque signature
- Utilisation d'informations uniques à l'expéditeur pour prévenir la falsification
- Facilité de production et de vérification
- Impossibilité computationnelle de falsification
- Possibilité pratique de conservation en stockage

b) prerequis-cryptographiques

Les exigences fondamentales pour un système de signature numérique robuste incluent :

— La signature doit être un motif de bits dépendant du message, assurant qu'elle ne peut être réutilisée ou transférée
— La production et la vérification doivent être efficaces d'un point de vue computationnel
— La falsification doit être informatiquement impossible, même avec des ressources substantielles
— Le système doit résister aux diverses catégories d'attaques cryptographiques

2. Motivations

Les signatures numériques jouent un rôle central en sécurité informatique. Elles permettent d'authentifier l'émetteur d'un message, d'en garantir l'intégrité et d'établir des preuves vérifiables dans des systèmes ouverts ou distribués. Les motivations suivantes résument les besoins essentiels auxquels répondent ces mécanismes.

— **Authentification de l'émetteur.** Une signature numérique permet d'établir de manière cryptographiquement vérifiable que le message a été produit par le détenteur légitime de la clé privée. L'identité de l'émetteur ne repose alors plus sur des éléments facilement falsifiables (adresse réseau, adresse e-mail, etc.), mais sur une opération mathématiquement liée à un secret que seul l'émetteur possède. Elle constitue ainsi un mécanisme d'authentification robuste face à l'usurpation d'identité.

— **Intégrité du message.** Les signatures lient de manière indissociable le message et la clé privée de l'émetteur : toute modification, même minime, modifie l'empreinte du document et invalide la signature lors de la vérification. Elles fournissent ainsi une garantie d'intégrité beaucoup plus forte qu'une simple valeur de contrôle non authentifiée.

— **Non-répudiation.** Puisque seule la clé privée peut produire une signature valide, il est irréfutable, dans un cadre cryptographique, que l'émetteur en est l'auteur. Cette propriété est indispensable dans les systèmes juridiques, administratifs, bancaires ou contractuels, où il est nécessaire de pouvoir établir la responsabilité d'une action.

— **Vérification indépendante et publique.** La vérification d'une signature ne requiert aucun secret : seule la clé publique associée est nécessaire. Toute personne ou tout système peut donc vérifier l'authenticité du message sans solliciter l'émetteur. Cela rend les signatures adaptées aux environnements distribués, ouverts et transparents tels que Internet, les architectures à clés publiques et les blockchains.

— **Sécurisation de la diffusion numérique.** Les signatures garantissent que les logiciels, mises à jour, documents officiels ou certificats émanent bien de leur auteur légitime et n'ont pas été modifiés. Elles constituent un rempart essentiel contre la distribution de contenus falsifiés, malveillants ou altérés, et sont aujourd'hui au cœur du code signing, des systèmes de mises à jour sécurisées et des autorités de certification.

— **Preuve d'engagement ou d'accord.** Dans les protocoles distribués, les transactions financières, les systèmes de vote électronique ou les accords contractuels, une signature représente une preuve vérifiable qu'une partie a approuvé une opération. Elle matérialise un engagement cryptographiquement solide, vérifiable a posteriori, et opposable en cas de litige ou de contestation.

— **Engagement à divulgation nulle de connaissance.** Certaines constructions cryptographiques utilisent des signatures comme briques pour produire des engagements permettant de prouver qu'une affirmation est vraie sans révéler aucune information sur le secret sous-jacent. Les signatures servent alors à garantir l'authenticité de l'engagement et à empêcher toute falsification. Couplées à des protocoles de type *zero-knowledge*, elles permettent d'attester une propriété (connaissance d'un secret, validité d'une donnée, appartenance à un ensemble) sans en divulguer le contenu, renforçant ainsi la confidentialité et la robustesse des protocoles interactifs et non interactifs.

3. Principes de base de la cryptographie asymétrique et PKI

La cryptographie asymétrique repose sur l'utilisation de paires de clés distinctes, l'une privée et l'autre publique, permettant d'assurer des fonctions fondamentales telles que la confidentialité, l'authentification et la signature numérique. La gestion correcte et sécurisée de ces clés est indispensable pour garantir la fiabilité des mécanismes cryptographiques, en particulier dans les environnements ouverts où les participants ne se connaissent pas directement. Les infrastructures de gestion des clés publiques (PKI) fournissent alors un cadre organisationnel et technique pour établir la confiance entre entités, certifier les clés et structurer leur utilisation.

3.1. Cryptographie asymétrique

La cryptographie asymétrique s'appuie sur une paire de clés mathématiquement liées :
une clé privée, gardée secrète, et une clé publique, librement diffusable. La clé privée permet d'effectuer des opérations sensibles, notamment signer ou déchiffrer, tandis que la clé publique sert à vérifier ou chiffrer selon le mécanisme utilisé. Cette séparation des rôles rend possible la communication sécurisée entre entités n'ayant aucun secret partagé réalable. Elle permet aussi de produire des signatures numériques vérifiables par tout tiers connaissant la clé publique correspondante. Le modèle repose sur la difficulté pratique d'inférer la clé privée à partir de la clé publique.

3.2. PKI

Une infrastructure à clés publiques (PKI) fournit un ensemble de services, d'entités et de règles permettant de gérer le ycle de vie des clés et des certificats numériques. Elle repose notamment sur les autorités de certification, chargées d'attester le lien entre une clé publique et une identité, et sur les registres et listes de révocation, qui assurent le suivi de la validité des certificats. La PKI organise ainsi la confiance au sein d'un réseau en offrant un mécanisme fiable pour vérifier l'authenticité

d'une clé publique avant de l'utiliser. Elle constitue un élément fondamental dans les systèmes sécurisés, les communications électroniques et les services nécessitant une vérification d'identité robuste.

4. Techniques de signature numérique

4.1. RSA

L'algorithme de signature RSA est une construction de cryptographie à clé publique permettant à un détenteur d'une clé privée de produire une *signature* attestant l'authenticité et l'intégrité d'un message. La vérification de la signature s'effectue à l'aide de la clé publique correspondante. La sécurité pratique repose sur des hypothèses arithmétiques (notamment la difficulté de factoriser un grand entier composé de deux premiers) et sur l'emploi d'un hachage et d'un schéma d'encodage/padding adaptés.

4.1.1. Principe mathématique

Soient p et q deux nombres premiers distincts et

$$n = p \cdot q, \qquad \varphi(n) = (p-1)(q-1).$$

On choisit un exposant public e tel que $\gcd(e, \varphi(n)) = 1$. L'exposant privé d est alors l'inverse multiplicatif de e modulo $\varphi(n)$:

$$d \equiv e^{-1} \pmod{\varphi(n)}, \quad \text{c.-à-d.} \quad ed \equiv 1 \pmod{\varphi(n)}.$$

L'arithmétique se déroule dans l'anneau $\mathbb{Z}_n$. Pour signer on transforme d'abord le message m en une valeur entière x avec $0 \leq x < n$ (typiquement via un hachage $H(m)$ puis un encodage/padding conforme aux standards, afin d'assurer unicité et sécurité).

4.1.2. Génération des clés

1. Choisir deux grands premiers secrets p et q.
2. Calculer $n = pq$ et $\varphi(n) = (p-1)(q-1)$.
3. Choisir e tel que $1 < e < \varphi(n)$ et $\gcd(e, \varphi(n)) = 1$ (valeur courante : $e = 65537$).
4. Calculer d tel que $ed \equiv 1 \pmod{\varphi(n)}$ (par exemple via l'algorithme d'Euclide étendu).
5. La clé publique est (n, e) ; la clé privée est d (et, pour des optimisations, les facteurs p, q sont conservés).

4.1.3. Algorithme de signature

Soit m le message à signer et x la valeur entière dérivée de m après hachage et encodage (on suppose $0 \leq x < n$).

$$\textbf{Signature :} \qquad s \equiv x^d \pmod{n}.$$

La valeur s est la signature qui accompagne m. En pratique l'exponentiation modulaire est réalisée par des méthodes rapides (exponentiation binaire) et l'utilisation du CRT (avec p, q) permet d'accélérer le calcul.

4.1.4. Algorithme de vérification

Étant donnée la signature s et la clé publique (n, e) :

$$\textbf{Vérification :} \qquad x' \equiv s^e \pmod{n}.$$

La vérification accepte la signature si et seulement si $x' = x$ (ou, après décodage, si l'empreinte obtenue coïncide avec $H(m)$). En pratique on compare les chaînes d'encodage / digest, non l'entier brut.

4.1.5. Preuve de correction

Supposons d'abord que $x \in \mathbb{Z}_n^\times$ (i.e. $\gcd(x, n) = 1$). Par définition de d il existe un entier k tel que

$$ed = 1 + k\varphi(n).$$

On a alors

$$s^e \equiv (x^d)^e = x^{ed} = x^{1+k\varphi(n)} = x \cdot \left(x^{\varphi(n)}\right)^k.$$

Par le théorème d'Euler, pour $x \in \mathbb{Z}_n^\times$ on a $x^{\varphi(n)} \equiv 1 \pmod{n}$. Ainsi $\left(x^{\varphi(n)}\right)^k \equiv 1$ et donc

$$s^e \equiv x \pmod{n},$$

ce qui montre que l'opération de vérification récupère bien x : la signature est correcte.

Si $\gcd(x, n) \neq 1$, la démonstration ci-dessus n'est pas directement applicable via Euler ; on raisonne alors modulo p et modulo q séparément. Étant donné $ed \equiv 1 \pmod{p-1}$ et $ed \equiv 1 \pmod{q-1}$, on obtient

$$s^e \equiv x \pmod{p} \quad \text{et} \quad s^e \equiv x \pmod{q},$$

puis, par le théorème chinois des restes, $s^e \equiv x \pmod{n}$. Cette variante montre que, même lorsque x partage un facteur avec n, la relation algébrique de correction tient en travaillant sur les composantes premières, ce qui motive le soin porté au choix de l'encodage/padding pour éviter de telles valeurs pathologiques en pratique.

4.1.6. Exemple concret

Supposons que Alice veut envoyer le message M = 4 à Bob avec une signature. Prenons des petits nombres pour faciliter.

Paramètres d'Alice

$$p = 3, \qquad q = 11$$
$$n = p \cdot q = 33, \qquad \varphi(n) = (p-1)(q-1) = 2 \cdot 10 = 20$$
$$e = 3 \quad \text{(exposant public)}, \qquad d = 7 \quad \text{(clé privée, car } 3 \times 7 \equiv 1 \pmod{20})$$

Signature d'un message

— Message à signer : $M = 4$.
— Alice calcule la signature :

$$S \equiv M^d \pmod{n} \quad \Rightarrow \quad S \equiv 4^7 \pmod{33}.$$
$$4^7 = 16384.$$
$$16384 \div 33 = 496 \text{ avec reste } 16.$$
$$\text{Ainsi :} \quad S = 16.$$

Vérification par Bob

— Bob reçoit $(M = 4,\ S = 16)$.
— Avec la clé publique d'Alice, il calcule :

$$M' \equiv S^e \pmod{n} \quad \Rightarrow \quad M' \equiv 16^3 \pmod{33}.$$

$$16^3 = 4096.$$

$$4096 \div 33 = 124 \text{ avec reste } 4.$$

$$M' = 4 = M \quad \Rightarrow \quad \text{Signature validée.}$$

La vérification confirme qu'Alice est bien l'auteur du message.

4.2. DSA

L'algorithme DSA (Digital Signature Algorithm) est un algorithme de signature numérique standardisé par le National Institute of Standards and Technology (NIST) en 1991 comme alternative à l'algorithme RSA standard. Il offre le même niveau de sécurité et de performance que le RSA, mais utilise une formule mathématique différente et moins répandue pour la signature et le chiffrement.

La sécurité du DSA repose sur la difficulté du problème du logarithme discret dans un groupe fini. Ce problème est considéré comme calculatoirement difficile, ce qui garantit la robustesse de l'algorithme contre les attaques par force brute.

4.2.1. Aperçu du DSA

Le processus se fait en trois étapes :
— Génération des clés
— Signature du document
— Vérification du document signé

4.2.2. Processus de signature

A. Génération des clés

1. Choisir des longueurs L et N avec L divisible par 64. Ces longueurs définissent directement le niveau de sécurité de la clé. NIST 800-57 recommande de choisir $L = 3072$ et $N = 256$ pour une sécurité équivalente à 128 bits.
2. Choisir un nombre premier p de longueur L
3. Choisir un nombre premier q de longueur N, de telle façon que $p - 1 = qz$ avec z un entier
4. Choisir h, avec $1 < h < p - 1$ de manière que :

$$g = h^z \bmod p > 1$$

5. Générer aléatoirement un x, avec $0 < x < q$
6. Calculer :

$$y = g^x \bmod p$$

Clés résultantes
— Clé publique : (p, q, g, y)
— Clé privée : x

B. Signature du document

Pour signer un message m, on procède comme suit :

1. Choisir un nombre aléatoire s tel que $1 < s < q$
2. Calculer :
$$s_1 = (g^s \bmod p) \bmod q$$
3. Si $s_1 = 0$, recommencer avec un autre s
4. Calculer :
$$s_2 = (H(m) + s_1 \cdot x) \cdot s^{-1} \bmod q$$
où $H(m)$ est le résultat d'un hachage cryptographique
5. Si $s_2 = 0$, recommencer avec un autre s
6. La signature est (s_1, s_2)

C. Vérification du document signé

Pour vérifier une signature (s_1, s_2) d'un message m :

1. Rejeter la signature si $0 < s_1 < q$ ou $0 < s_2 < q$ n'est pas vérifiée
2. Calculer :
$$w = s_2^{-1} \bmod q$$
3. Calculer :
$$u_1 = H(m) \cdot w \bmod q$$
4. Calculer :
$$u_2 = s_1 \cdot w \bmod q$$
5. Calculer :
$$v = (g^{u_1} \cdot y^{u_2} \bmod p) \bmod q$$
6. La signature est valide si $v = s_1$

4.2.3. Validité de l'algorithme

Ce principe de signature est correct dans le sens où le vérificateur acceptera toujours des signatures authentiques.

Justification mathématique

Démontrons que pour une signature valide, on a bien $v = s_1$:
À partir de l'équation de signature :

$$s_2 = (H(m) + s_1 \cdot x) \cdot s^{-1} \bmod q$$

On peut déduire :

$$s = (H(m) + s_1 \cdot x) \cdot s_2^{-1} \bmod q$$

Donc :

$$s = H(m) \cdot w + s_1 \cdot x \cdot w \bmod q$$
$$s = u_1 + s_1 \cdot x \cdot w \bmod q$$

Or, par définition, $s_1 = (g^s \bmod p) \bmod q$, donc :

$$\begin{aligned} v &= (g^{u_1} \cdot y^{u_2} \bmod p) \bmod q \\ &= (g^{u_1} \cdot g^{x \cdot u_2} \bmod p) \bmod q \\ &= (g^{u_1 + x \cdot u_2} \bmod p) \bmod q \\ &= (g^{u_1 + x \cdot s_1 \cdot w} \bmod p) \bmod q \\ &= (g^s \bmod p) \bmod q \\ &= s_1 \end{aligned}$$

CONCLUSION

Les fonctions de hachage cryptographiques et les signatures numériques sont des outils indispensables pour sécuriser les communications et les données dans les environnements numériques ouverts. Ce travail a permis d'en explorer les fondements théoriques, les constructions techniques et les applications pratiques.

Nous avons vu que les fonctions de hachage, notamment à travers les constructions de Merkle-Damgård et éponge, évoluent pour répondre à des exigences de sécurité toujours plus fortes. Alors que MD5 est aujourd'hui considéré comme obsolète en raison de ses vulnérabilités aux collisions, SHA-256 reste un standard robuste et largement déployé.

Les signatures numériques, quant à elles, s'appuient sur la cryptographie asymétrique et les infrastructures PKI pour offrir authentification, intégrité et non-répudiation. Les algorithmes RSA et DSA illustrent deux approches complémentaires, fondées respectivement sur la factorisation d'entiers et le problème du logarithme discret.

Enfin, le développement d'une application démonstrative a montré comment intégrer ces briques cryptographiques dans un système concret, depuis le hachage et la signature jusqu'à la vérification via des certificats X.509.

Dans un monde de plus en plus interconnecté et exposé aux menaces cybernétiques, la maîtrise de ces technologies reste essentielle pour concevoir des systèmes fiables, vérifiables et résistants aux attaques.

ANNEXE

I. DEVELOPPEMENT D'UN SYSTEME LOGICIEL ILLUSTRATIF

Cette section présente les activités de développement de d'une application de signature numérique développée dans le cadre de notre l'exposé.

I.1. Contexte et objectifs

L'objectif principal est de démontrer un système simple permettant :

- le chiffrage et déchiffrement de documents ;
- la signature numérique garantissant l'authenticité et l'intégrité d'un document ;
- l'utilisation de certificats X.509 pour vérifier l'identité des expéditeurs.

Ces objectifs servent à illustrer des primitives cryptographiques dans le cadre d'un exposé (flux Alice/Bob/Oscar).

I.2. Description des besoins

Besoins fonctionnels

1. Charger un document (PDF, texte) et en générer le haché ;
2. Signer le haché avec la clé privée de l'utilisateur ;
3. Chiffrer le document si nécessaire pour la confidentialité ;
4. Vérifier la signature avec la clé publique et le certificat X.509 ;
5. Gérer des clefs (génération RSA, import/export) et certificats.

Besoins non fonctionnels

- Sécurité des clefs (gestion serveur, stockage protégé) ;
- Simplicité d'interface pour une démonstration en salle ;
- Déploiement continu sur une plateforme accessible.

I.3. Conception de la solution

Vue fonctionnelle

Le processus principal est le suivant :

1. Lútilisateur A (Alice) charge un document M ;
2. L'application calcule le haché $h(M)$ (SHA-256) ;

3. A signe le haché avec sa clé privée (RSA-PSS + SHA-256) pour obtenir la signature S_M ;
4. Optionnellement, A chiffre le document/signature (AES-256-GCM / RSA-OAEP) ;
5. Le destinataire B (Bob) vérifie le certificat X.509 de A et la signature ;
6. Si la vérification est correcte, le document est accepté comme intégral et authentique.

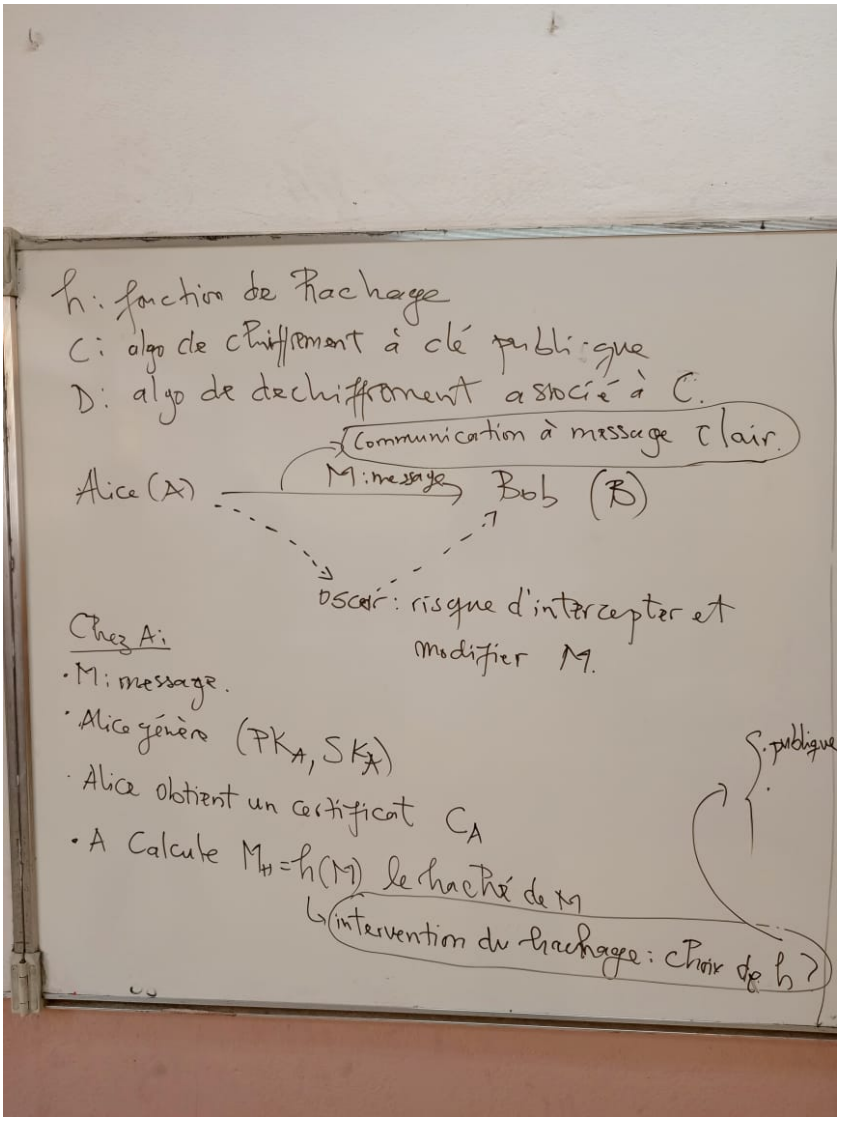

FIGURE 5 – Illustration du flux fonctionnelle 1

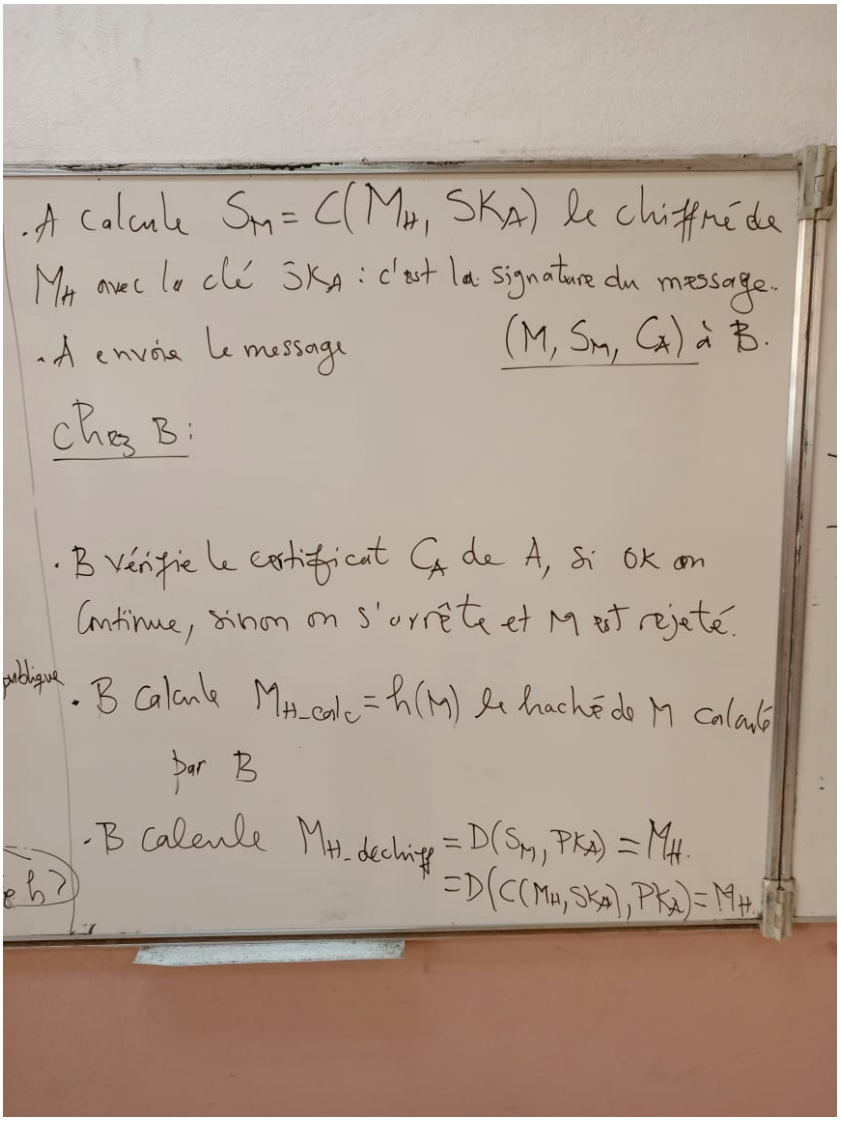

FIGURE 6 – Illustration du flux fonctionnelle 2

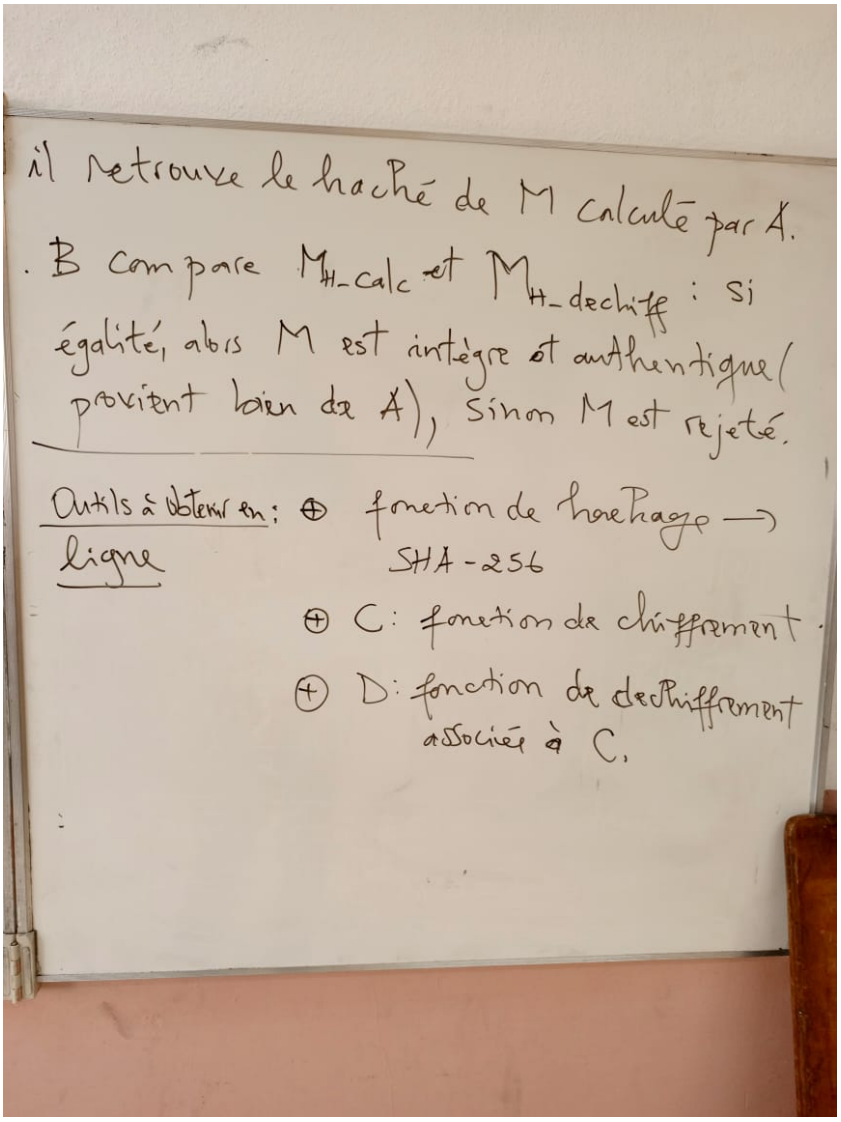

FIGURE 7 – Illustration du flux fonctionnelle 3

Composants principaux

- Frontend Web (interface de chargement et vérification) ;
- Backend (API Django) gère les opérations cryptographiques et le stockage temporaire ;
- Module cryptographique (Python Cryptography library) : gestion RSA, AES, PSS, OAEP, X.509 ;
- Base minimale de données ou stockage de fichiers pour journaux / certificats.

I.4. Choix techniques et justification

- Hachage : SHA-256 (performance, standardisation) ;
- Signature asymétrique : RSA-PSS avec `SHA-256` pour la robustesse probabiliste ;
- Chiffrement asymétrique : RSA-OAEP pour l"echangement de clefs ou petites charges ;
- Chiffrement symétrique : AES-256-GCM pour la confidentiality et l'intégrité (auth tag) ;
- Certificats : X.509 v3 pour références d'identité standard ;

— Langage/back-end : Python + Django (rapidité de prototypage, bibliothèques crypto) ;
— Déploiement : plateforme Railway (application publique de démonstration).

Partie 1 : Expéditeur (Signature et Chiffrement)

```
# Signer le document (hash SHA-256) avec cle privee de l'
    expediteur (RSA-PSS)
sender_private = serialization.load_pem_private_key(
    sender.private_key.encode(), password=None
)

# Calculer le hash du document
doc_hash = hashlib.sha256(document_bytes).digest()

# Signer le hash
signature = sender_private.sign(
    doc_hash,
    asym_padding.PSS(
        mgf=asym_padding.MGF1(hashes.SHA256()),
        salt_length=asym_padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)

# Generer une cle symetrique AES-256-GCM
aes_key = AESGCM.generate_key(bit_length=256)
nonce = os.urandom(12)  # 96-bit nonce pour GCM

# Chiffrer le document avec AES-256-GCM
aesgcm = AESGCM(aes_key)
ciphertext = aesgcm.encrypt(nonce, document_bytes, None)

# Chiffrer la cle AES avec la cle publique du destinataire (RSA-
    OAEP)
recipient_public = serialization.load_pem_public_key(
    recipient.public_key.encode()
)
```

Listing 1 – Processus côte expediteur: signer et chiffrer

Partie 2 : Destinataire (Vérification et Déchiffrement)

```
# Verifier le certificat X.509 de l'expediteur
sender_cert = x509.load_pem_x509_certificate(sender_cert_pem.
    encode())
cert_subject = sender_cert.subject.get_attributes_for_oid(NameOID.
    COMMON_NAME)[0].value
cert_valid_from = sender_cert.not_valid_before_utc
cert_valid_to = sender_cert.not_valid_after_utc
cert_is_valid = cert_valid_from <= datetime.now(cert_valid_from.
    tzinfo) <= cert_valid_to

# Dechiffrer la cle AES avec la cle privee RSA (RSA-OAEP)
recipient_private = serialization.load_pem_private_key(
    current_user.private_key.encode(), password=None
```

```
)
aes_key = recipient_private.decrypt(
    encrypted_key,
    asym_padding.OAEP(
        mgf=asym_padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Dechiffrer le document avec AES-256-GCM
aesgcm = AESGCM(aes_key)
decrypted_bytes = aesgcm.decrypt(nonce, ciphertext, None)

# Verifier la signature RSA-PSS avec la cle publique de l'
    expediteur
sender_public = sender_cert.public_key()
doc_hash = base64.b64decode(doc_hash_b64) if doc_hash_b64 else
    hashlib.sha256(decrypted_bytes).digest()

try:
    sender_public.verify(
        signature,
        doc_hash,
        asym_padding.PSS(
            mgf=asym_padding.MGF1(hashes.SHA256()),
            salt_length=asym_padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    signature_valid = True
except InvalidSignature:
    signature_valid = False

# Marquer comme lu
document.is_read = True
document.save()
```

Listing 2 – Processus côte destinataire: verifier et dechiffrer

I.5. Architecture déploiement

La solution est déployée en mode web : front-end servi par Django (templates ou mini SPA), backend exposant des endpoints REST pour : génération de haché, signature, chiffrement/déchiffrement, vérification de certificats.

I.6. Quelques captures d'ecran de l'interface de l'application devéloppée

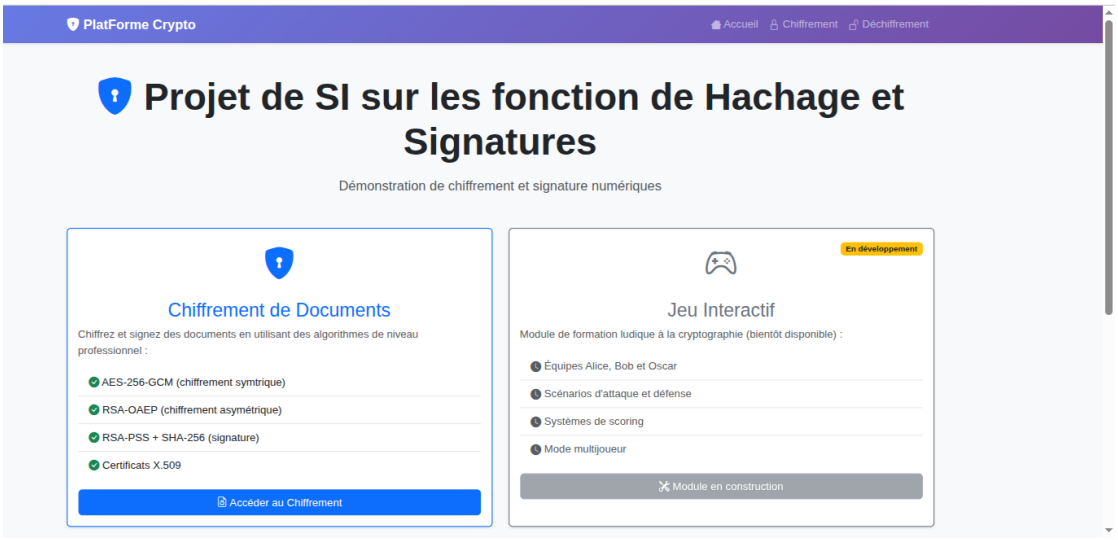

FIGURE 8 – Page d'accueil de la plateforme

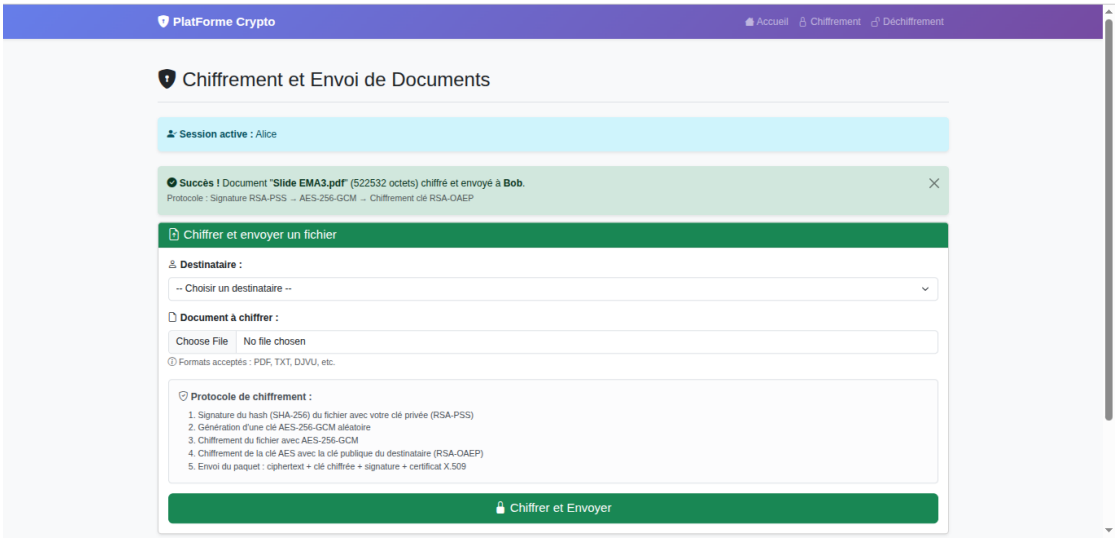

FIGURE 9 – Page permettant de signer(eventuellement chiffrer) un fichier et l'envoyer

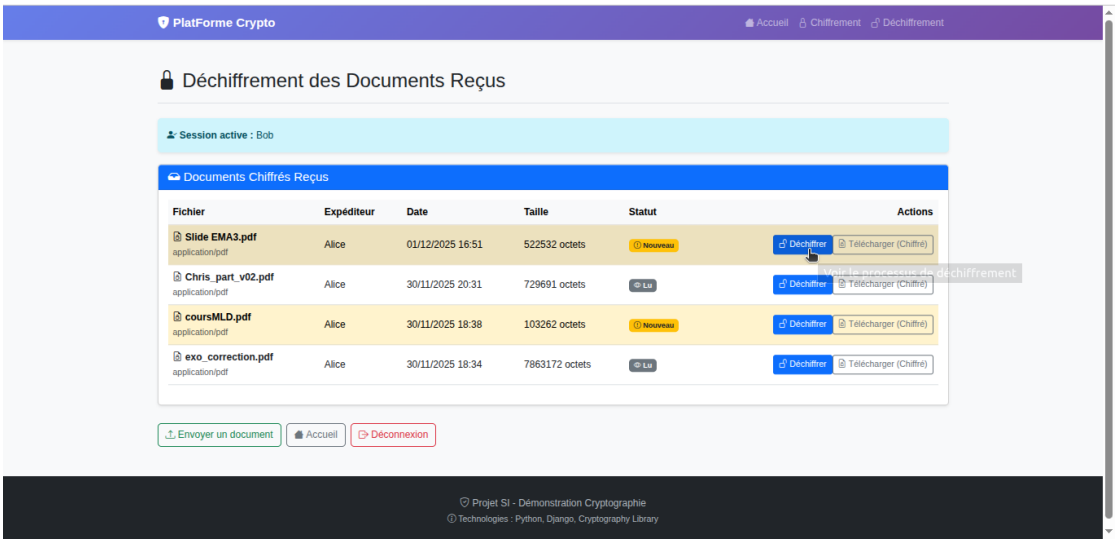

FIGURE 10 – Page pour le choix d'un fichier reçu qu'on veut consulter

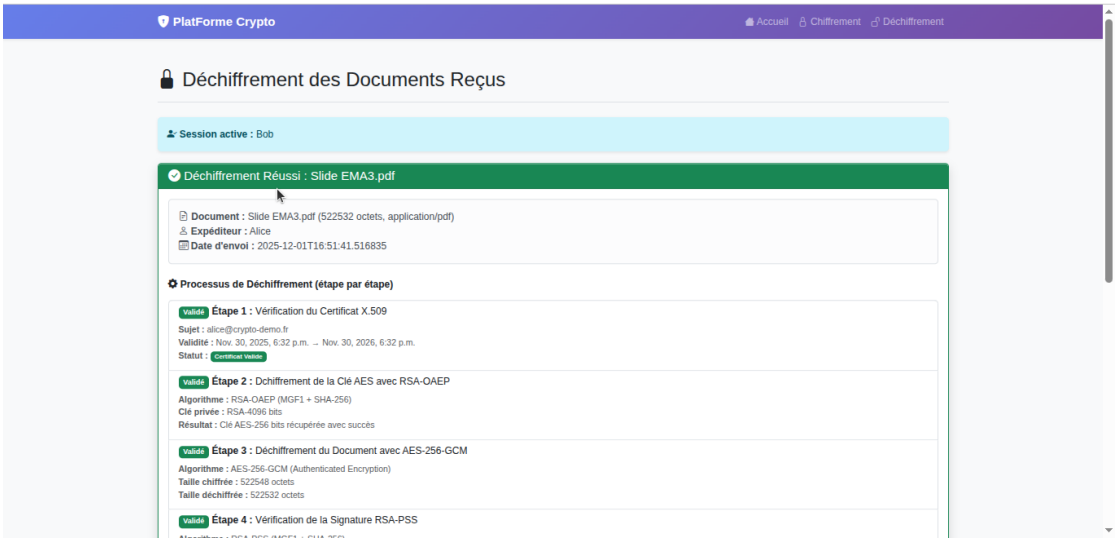

FIGURE 11 – Vérification de la signature réussie

II. Quelques exercices

Exercice 1 : Propriétés des fonctions de hachage cryptographiques

Énoncé :

Soit h une fonction de hachage cryptographique produisant une empreinte de 256 bits.

1. Expliquez la différence entre la **résistance à la préimage** et la **résistance aux collisions**.
2. Quelle est la complexité théorique attendue pour trouver :
   (a) une préimage ?
   (b) une collision ?
3. Pourquoi la résistance aux collisions est-elle plus difficile à atteindre que la résistance à la préimage sur le plan théorique ?

Corrigé :

1. **Résistance à la préimage** : étant donné une empreinte y, il doit être impossible de trouver un message x tel que $h(x) = y$.
   Résistance aux collisions : il doit être impossible de trouver deux messages distincts x_1 et x_2 tels que $h(x_1) = h(x_2)$.
2. Complexité théorique (pour une fonction idéale) :
   (a) Préimage : $O(2^{256})$
   (b) Collision : $O(2^{128})$ (grâce à l'attaque des anniversaires)
3. La recherche de collision bénéficie de l'**effet du paradoxe des anniversaires** : on cherche une paire parmi N messages, ce qui réduit la complexité à $\sqrt{N}$, alors que pour une préimage, on cherche un message spécifique correspondant à une empreinte donnée, d'où une complexité en N.

Exercice 2 : Attaque par extension de longueur sur Merkle-Damgård

Énoncé :

Soit une fonction de hachage H construite selon la méthode de Merkle-Damgård, avec une fonction de compression f, une valeur initiale IV, et un bourrage standard MD. On connaît un message M (non bourré) et son haché $h = H(M)$.

1. Expliquez comment un attaquant peut produire un message $M' = M \| \text{padding}(M) \| X$ (où X est un texte arbitraire) et calculer $H(M')$ sans connaître M, mais seulement $|M|$ et h.
2. Quelle est la conséquence de cette vulnérabilité pour l'utilisation de H dans des contextes comme les codes d'authentification de message (MAC) naïfs ?
3. Proposez une contre-mesure simple pour se prémunir contre cette attaque.

Corrigé :

1. L'attaque procède ainsi :
   (a) L'attaquant connaît $|M|$ et peut donc reconstruire $\text{padding}(M)$.
   (b) Il forme $M_{\text{pad}} = M \| \text{padding}(M)$.
   (c) Dans la construction MD, h est exactement la valeur de la variable de chaînage après traitement du dernier bloc de M_{pad}.

(d) Pour étendre avec X, l'attaquant commence avec h comme valeur initiale et continue le processus itératif sur les blocs de $X' = \text{padding}(X)$.

(e) Il obtient ainsi $H(M\|\text{padding}(M)\|X')$ sans connaître M.

2. Cette vulnérabilité rend dangereux l'usage naïf de $H(k\|m)$ comme MAC, car un attaquant peut étendre le message sans connaître la clé secrète k.
3. Contre-mesure : utiliser HMAC ou une construction éponge qui n'est pas vulnérable à l'extension de longueur. Sinon, dans un protocole maison, on peut utiliser $H(k\|m\|k)$ ou des méthodes similaires qui brisent la structure linéaire.

Exercice 3 : Signature RSA avec padding PKCS#1 v1.5

Énoncé :

Dans le schéma de signature RSA avec padding PKCS#1 v1.5, le message encodé avant signature a la forme :

$$00\ 01\ FF\ FF\ ...\ FF\ 00\|\text{hash\_alg\_identifiant}\|\text{hash}(m)$$

Soit un module RSA n de 2048 bits. On signe un message m avec SHA-256.

1. Calculez la taille maximale du champ de padding (les octets FF) sachant que :
   — L'en-tête occupe 3 octets (00 01 FF)
   — Le séparateur : 1 octet (00)
   — L'identifiant SHA-256 : 19 octets (codé ASN.1)
   — Le haché SHA-256 : 32 octets
2. Pourquoi ce padding est-il important pour la sécurité ?
3. Quel est l'inconvénient principal de PKCS#1 v1.5 par rapport à PSS ?

Corrigé :

1. Taille totale du message encodé : 256 octets (2048 bits).
   Taille fixe occupée : $3 + 1 + 19 + 32 = 55$ octets.
   Taille du padding : $256 - 55 = 201$ octets, soit 200 octets FF après le premier FF.
2. Le padding remplit plusieurs rôles :
   — Il garantit que le message à signer est de la taille du module.
   — Il ajoute une redondance structurée qui permet de détecter les falsifications.
   — Il empêche certaines attaques comme celle de la recherche de messages de forme spéciale.
3. PKCS#1 v1.5 est déterministe (même message $\rightarrow$ même signature) et a été vulnérable à des attaques par oracle de padding dans le passé. PSS (Probabilistic Signature Scheme) est probabiliste (ajout d'un sel aléatoire) et offre une sécurité prouvée sous l'hypothèse RSA, ce qui en fait un choix plus robuste.

BIBLIOGRAPHIE/WEBOGRAPHIE

WEBOGRAPHIE

— https ://www.rfc-editor.org/

BIBLIOGRAPHIE

— Cours de sciences de l'information du Dr Hervé TALE KALACHI, ENSPY, UY1
— Cryptographic Hash Functions Add-on, Network Security course, Technische Universität München.
— Encyclopedia of Cryptography, Security and Privacy, 3rd (Sushil Jajodia, Pierangela Samarati, Moti Yung, ...)

— Analyse et conception de fonctions de hachage cryptographiques, Thèse de doctorat de Stéphane Manuel à l'Ecole Polytechnique de Paris (X).