Devoir 2

IFT814 – Cryptographie

Chargé de cours : Martin Fiset

Étudiant : Mamadou Senghor

CIP : senm1912

Université de Sherbrooke

Département d'informatique

Automne 2025

Date de remise : 10 octobre 2025

Table des matières

1 Analyse d’un MAC défaillant

Le schéma étudié est un code d’authentification de message (MAC) défini par :

$$\mathrm{MAC}(k, m) = (m \oplus k) \bmod 2^{32}$$

où la clé k et le message m sont des entiers de 64 bits. Seuls les 32 bits de poids faibles du résultat sont conservés comme *tag*.

La vérification du message est donnée par :

$$\mathrm{Verif}(k, m, t) = \begin{cases} 1, & \text{si } t = \mathrm{MAC}(k, m) \\ 0, & \text{sinon.} \end{cases}$$

Ce schéma est volontairement **non sécurisé**, afin d’en démontrer les failles.

1.1 Code source Python

Le code suivant implémente le schéma MAC et simule quatre scénarios entre **Alice**, **Bob** et **Eve**.

Listing 1 – Implémentation complète du MAC et simulation des scénarios.

```
import secrets

MASK32 = (1 << 32) - 1 # 0xFFFFFFFF
MASK64 = (1 << 64) - 1 # 0xFFFFFFFFFFFFFFFF


# ===============================================================
# 1. Fonctions du MAC
# ===============================================================
def Gen():
    """Genere une clé k de 64 bits."""
    return secrets.randbits(64) & MASK64


def MAC(k, m):
    """
    Genere un tag d’authentification de 32 bits.
    MAC(k,m) = (m  k) mod 2^32
    """
    return (m ^ k) & MASK32


def Verif(k, m, tag):
    """
    Vérifie l’authenticité d’un message.
    Retourne 1 si accepté, 0 si rejeté.
    """
    return 1 if MAC(k, m) == (tag & MASK32) else 0


# ===============================================================
# 2. Fonctions d’affichage
# ===============================================================
def b64(x):
```

```
    """Retourne une représentation binaire sur 64 bits."""
    return f"{x & MASK64:064b}"


def b32(x):
    """Retourne une représentation binaire sur 32 bits."""
    return f"{x & MASK32:032b}"


def b64_split(x):
    """Retourne une représentation binaire sur 64 bits, séparée en deux parties de 32 bits."""
    high = (x >> 32) & MASK32
    low = x & MASK32
    return f"{high:032b} {low:032b}"


# ==============================================================
# 3. Messages et modes (alternance PASSIF / ACTIF)
# ==============================================================
def get_messages_and_modes():
    """
    Définit les 4 messages selon l'énoncé:
    - m = 0^64 (tous les bits à0)
    - m = 2^63 (bit de poids fort à1, les autres à0)
    - m = 0^32 puis alternance 101010...10
    - m = alternance de 101010...10 sur 64 bits
    """
    m1 = 0 # tous les bits à0
    m2 = 1 << 63 # bit de poids fort à1, autres à0
    m3 = (1 << 32) - 1 # 32 bits bas à1, 32 bits hauts à0
    m4 = int('10' * 32, 2) # alternance 1010... sur 64 bits

    messages = [
        (m1, "m = 0^64 (tous les bits à0)"),
        (m2, "m = 2^63 (bit de poids fort à1, autres à0)"),
        (m3, "m = 0^32 puis alternance 101010...10 (32 bits)"),
        (m4, "m = alternance 1010... sur 64 bits")
    ]

    modes = ["PASSIF", "ACTIF", "PASSIF", "ACTIF"]
    return messages, modes


# ==============================================================
# 4. Simulation principale
# ==============================================================
def scenario():
    messages, modes = get_messages_and_modes()

    print("\n" + "=" * 80)
    print(" SIMULATION DU SCÉNARIO ALICE  EVE  BOB")
    print(" Alternance : PASSIF / ACTIF / PASSIF / ACTIF")
    print("=" * 80)

    for i, ((m, desc), mode) in enumerate(zip(messages, modes), start=1):
        print(f"\n{'-' * 80}")
        print(f"SCÉNARIO {i}: {desc}")
```

```python
        print(f"MODE : {mode}")
        print(f"{'-' * 80}")

        # Alice génère la clé et le tag
        k = Gen()
        t = MAC(k, m)

        # --- CTÉ ALICE ---
        print("\n CTÉ ALICE")
        print(f" Clé k générée : {b64_split(k)}")
        print(f" Message m : {b64_split(m)}")
        print(f" Tag t calculé : {b32(t)}")
        print(f" (tag = (m  k) mod 2^32, garde 32 bits bas)")

        # --- CTÉ EVE ---
        print("\n CTÉ EVE (interception)")
        print(f" Observe m : {b64_split(m)}")
        print(f" Observe t : {b32(t)}")

        if mode == "ACTIF":
            # Eve modifie les 32 bits hauts du message
            delta_high = secrets.randbits(32)
            modification = (delta_high << 32) & MASK64
            m_prime = m ^ modification

            print("\n EVE MODIFIE le message (ATTAQUE ACTIVE)")
            print(f" Modification  (32 bits hauts) : {b32(delta_high)} 00000000000000000000000000000000")
            print(f" Message original m : {b64_split(m)}")
            print(f" Message modifié m' : {b64_split(m_prime)}")
            print(f" Tag transmis (INCHANGÉ) : {b32(t)}")

            # Analyse de la modification
            diff = m ^ m_prime
            print(f"\n Analyse : m  m' = {b64_split(diff)}")
            print(f"  Seuls les 32 bits HAUTS ont changé!")

        else:
            m_prime = m
            print("\n EVE reste PASSIVE (pas de modification)")
            print(f" Message transmis m' : {b64_split(m_prime)}")
            print(f" Tag transmis : {b32(t)}")

        # --- CTÉ BOB ---
        print("\n CTÉ BOB (réception)")
        v = Verif(k, m_prime, t)
        print(f" Message reçu m' : {b64_split(m_prime)}")
        print(f" Tag reçu t : {b32(t)}")

        # Calcul du tag attendu par Bob
        t_bob = MAC(k, m_prime)
        print(f" Tag calculé : {b32(t_bob)}")
        print(f" Tags égaux ? : {t == t_bob}")
        print(f" Vérification : {' ACCEPTÉ' if v == 1 else ' REJETÉ'} (v = {v})")

        # --- DÉTAIL DU CALCUL ---
        print("\n DÉTAIL DU CALCUL")
        xor_val = m_prime ^ k
```

```
        xor_high = (xor_val >> 32) & MASK32
        xor_low = xor_val & MASK32

        print(f" m'  k = {b64_split(xor_val)}")
        print(f"  32 bits hauts  32 bits bas")
        print(f" Partie haute (IGNORÉE par mod 2^32) : {b32(xor_high)}")
        print(f" Partie basse (= tag vérifié) : {b32(xor_low)}")

        # --- CONCLUSION DU SCÉNARIO ---
        print("\n CONCLUSION")
        if mode == "ACTIF":
            if v == 1:
                print(" PROBLME CRITIQUE : Bob accepte un message MODIFIÉ!")
                print(" Le MAC est CASSÉ : Eve peut modifier les 32 bits hauts")
                print(" sans que le tag ne change!")
            else:
                print(" Bob rejette le message modifié (comportement normal)")
        else:
            if v == 1:
                print(" Transmission passive réussie (comportement normal)")
            else:
                print(" Problème inattendu : message légitime rejeté!")



# ============================================================
# 5. Exécution principale
# ============================================================
if __name__ == "__main__":

    scenario()

    print("\n" + "=" * 80)
    print("FIN DE LA SIMULATION")
    print("=" * 80 + "\n")
```

1.2 Résultats expérimentaux

L'exécution du script affiche les échanges entre Alice, Eve et Bob, pour quatre scénarios alternés : deux passifs, deux actifs.

Les captures ci-dessous montrent les sorties du terminal.

```
--------------------------------------------------------------------------------
SCÉNARIO 1: m₁ = 0^64 (tous les bits à 0)
MODE : PASSIF
--------------------------------------------------------------------------------

 CÔTÉ ALICE
  Clé k générée : 00111000100001110101010001110111 10110100111000111110001011000001
  Message m     : 00000000000000000000000000000000 00000000000000000000000000000000
  Tag t calculé : 10110100111000111110001011000001
  (tag = (m ⊕ k) mod 2^32, garde 32 bits bas)

 CÔTÉ EVE (interception)
  Observe m : 00000000000000000000000000000000 00000000000000000000000000000000
  Observe t : 10110100111000111110001011000001

  EVE reste PASSIVE (pas de modification)
  Message transmis m' : 00000000000000000000000000000000 00000000000000000000000000000000
  Tag transmis : 10110100111000111110001011000001

 CÔTÉ BOB (réception)
  Message reçu m' : 00000000000000000000000000000000 00000000000000000000000000000000
  Tag reçu t      : 10110100111000111110001011000001
  Tag calculé     : 10110100111000111110001011000001
  Tags égaux ?    : True
  Vérification    :  ACCEPTÉ (v = 1)

 DÉTAIL DU CALCUL
  m' ⊕ k = 00111000100001110101010001110111 10110100111000111110001011000001
           ↓ 32 bits hauts    ↓ 32 bits bas
  Partie haute (IGNORÉE par mod 2^32) : 00111000100001110101010001110111
  Partie basse (= tag vérifié)        : 10110100111000111110001011000001

 CONCLUSION
  Transmission passive réussie (comportement normal)
```

FIGURE 1 – Scénario 1 — Eve passive (message inchangé).

```
--------------------------------------------------------------------------------
SCÉNARIO 2: m₂ = 2^63 (bit de poids fort à 1, autres à 0)
MODE : ACTIF
--------------------------------------------------------------------------------

 CÔTÉ ALICE
  Clé k générée : 10001010100000000111100001001100 00111101100000010010110011100010
  Message m     : 10000000000000000000000000000000 00000000000000000000000000000000
  Tag t calculé : 00111101100000010010110011100010
  (tag = (m ⊕ k) mod 2^32, garde 32 bits bas)

 CÔTÉ EVE (interception)
  Observe m : 10000000000000000000000000000000 00000000000000000000000000000000
  Observe t : 00111101100000010010110011100010

  EVE MODIFIE le message (ATTAQUE ACTIVE)
  Modification δ (32 bits hauts) : 00111000000000000100001001010101 00000000000000000000000000000000
  Message original m  : 10000000000000000000000000000000 00000000000000000000000000000000
  Message modifié m'  : 10111000000000000100001001010101 00000000000000000000000000000000
  Tag transmis (INCHANGÉ) : 00111101100000010010110011100010

  Analyse : m ⊕ m' = 00111000000000000100001001010101 00000000000000000000000000000000
  → Seuls les 32 bits HAUTS ont changé!

 CÔTÉ BOB (réception)
  Message reçu m' : 10111000000000000100001001010101 00000000000000000000000000000000
  Tag reçu t      : 00111101100000010010110011100010
  Tag calculé     : 00111101100000010010110011100010
  Tags égaux ?    : True
  Vérification    :  ACCEPTÉ (v = 1)

 DÉTAIL DU CALCUL
  m' ⊕ k = 00110010100000000011101000011001 00111101100000010010110011100010
           ↓ 32 bits hauts   ↓ 32 bits bas
  Partie haute (IGNORÉE par mod 2^32) : 00110010100000000011101000011001
  Partie basse (= tag vérifié)        : 00111101100000010010110011100010

 CONCLUSION
  PROBLÈME CRITIQUE : Bob accepte un message MODIFIÉ!
  Le MAC est CASSÉ : Eve peut modifier les 32 bits hauts
  sans que le tag ne change!
```

FIGURE 2 – Scénario 2 — Eve active (message modifié accepté).

```
-----------------------------------------------------------------------------
SCÉNARIO 3: m₃ = 0^32 puis alternance 101010...10 (32 bits)
MODE : PASSIF
-----------------------------------------------------------------------------

 CÔTÉ ALICE
  Clé k générée : 11110010010110001100101101011010 11100001010100110010101101010010
  Message m     : 00000000000000000000000000000000 11111111111111111111111111111111
  Tag t calculé : 00011110101011001101010010101101
  (tag = (m ⊕ k) mod 2^32, garde 32 bits bas)

 CÔTÉ EVE (interception)
  Observe m : 00000000000000000000000000000000 11111111111111111111111111111111
  Observe t : 00011110101011001101010010101101

  EVE reste PASSIVE (pas de modification)
  Message transmis m' : 00000000000000000000000000000000 11111111111111111111111111111111
  Tag transmis : 00011110101011001101010010101101

 CÔTÉ BOB (réception)
  Message reçu m' : 00000000000000000000000000000000 11111111111111111111111111111111
  Tag reçu t      : 00011110101011001101010010101101
  Tag calculé     : 00011110101011001101010010101101
  Tags égaux ?    : True
  Vérification    :  ACCEPTÉ (v = 1)

 DÉTAIL DU CALCUL
  m' ⊕ k = 11110010010110001100101101011010 00011110101011001101010010101101
          ↓ 32 bits hauts   ↓ 32 bits bas
  Partie haute (IGNORÉE par mod 2^32) : 11110010010110001100101101011010
  Partie basse (= tag vérifié)        : 00011110101011001101010010101101

 CONCLUSION
  Transmission passive réussie (comportement normal)
```

FIGURE 3 – Scénario 3 — Eve passive.

```
--------------------------------------------------------------------------------
SCÉNARIO 4: m₄ = alternance 1010... sur 64 bits
MODE : ACTIF
--------------------------------------------------------------------------------

 CÔTÉ ALICE
  Clé k générée : 00010011001000010010101111100110 01101011010011010101010000110011
  Message m     : 10101010101010101010101010101010 10101010101010101010101010101010
  Tag t calculé : 11000001111001111111111010011001
  (tag = (m ⊕ k) mod 2^32, garde 32 bits bas)

 CÔTÉ EVE (interception)
  Observe m : 10101010101010101010101010101010 10101010101010101010101010101010
  Observe t : 11000001111001111111111010011001

  EVE MODIFIE le message (ATTAQUE ACTIVE)
  Modification δ (32 bits hauts) : 10000011100011001101000111000110 00000000000000000000000000000000
  Message original m  : 10101010101010101010101010101010 10101010101010101010101010101010
  Message modifié m'  : 00101001001001100111101101101100 10101010101010101010101010101010
  Tag transmis (INCHANGÉ) : 11000001111001111111111010011001

  Analyse : m ⊕ m' = 10000011100011001101000111000110 00000000000000000000000000000000
  → Seuls les 32 bits HAUTS ont changé!

 CÔTÉ BOB (réception)
  Message reçu m' : 00101001001001100111101101101100 10101010101010101010101010101010
  Tag reçu t      : 11000001111001111111111010011001
  Tag calculé     : 11000001111001111111111010011001
  Tags égaux ?    : True
  Vérification    :  ACCEPTÉ (v = 1)

 DÉTAIL DU CALCUL
  m' ⊕ k = 00111010000000111010100000100010 11000001111001111111111010011001
           ↓ 32 bits hauts   ↓ 32 bits bas
  Partie haute (IGNORÉE par mod 2^32) : 00111010000000111010100000100010
  Partie basse (= tag vérifié)        : 11000001111001111111111010011001

 CONCLUSION
  PROBLÈME CRITIQUE : Bob accepte un message MODIFIÉ!
  Le MAC est CASSÉ : Eve peut modifier les 32 bits hauts
  sans que le tag ne change!
```

FIGURE 4 – Scénario 4 — Eve active : Bob accepte un message altéré.

2 Analyse mathématique et démonstrations

2.1 Analyse de la probabilité de réussite d'Eve

Le schéma MAC considéré est défini par :

$$\mathrm{MAC}(k, m) = (m \oplus k) \bmod 2^{32}$$

où $k, m \in \{0,1\}^{64}$.

On observe que l'opération $\bmod 2^{32}$ ne conserve que les **32 bits de poids faibles** du résultat de $(m \oplus k)$. Ainsi, les 32 bits de poids forts de m **n'ont aucune influence** sur la valeur du tag t.

Soit $m' = m \oplus (\Delta \ll 32)$, c'est-à-dire une modification portant uniquement sur la partie haute du message. Alors :

$$\begin{aligned}
\mathrm{MAC}(k, m') &= (m' \oplus k) \bmod 2^{32} \\
&= ((m \oplus (\Delta \ll 32)) \oplus k) \bmod 2^{32} \\
&= ((m \oplus k) \oplus (\Delta \ll 32)) \bmod 2^{32} \\
&= (m \oplus k) \bmod 2^{32} \quad \text{(car les bits modifiés sont au-delà du bit 32)} \\
&= \mathrm{MAC}(k, m)
\end{aligned}$$

Conséquence : Eve peut **modifier arbitrairement** les 32 bits de poids forts de m sans changer le tag t. Ainsi :

$$\boxed{\Pr[\mathrm{MAC}(k, m') = \mathrm{MAC}(k, m)] = 1}$$

De plus, après avoir observé plusieurs paires (m_i, t_i), Eve peut forger un nouveau message m_e tel que :

$$m_e = m_i \oplus (\Delta \ll 32)$$

et obtenir un tag valide t_i sans connaître la clé k. Cette attaque réussit encore avec une probabilité de 1.

Conclusion : Le schéma MAC échoue totalement à assurer l'intégrité :

$$\mathrm{MAC}(k, m') = \mathrm{MAC}(k, m) \quad \forall\, m' \text{ qui diffère de } m \text{ sur les 32 bits hauts.}$$

Eve peut donc falsifier ou forger des messages valides de façon déterministe.

2.2 Étude de la pseudo-aléatoirité de la fonction $F(k, x)$

On considère la fonction :

$$F : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^{n-16}, \quad F(k, x) = (x \oplus k) \bmod 2^{n-16}$$

c'est-à-dire que l'on tronque les 16 bits de poids forts du résultat.

Observation : Deux valeurs d'entrée x_1 et x_2 telles que :

$$x_1 \oplus x_2 = 2^{n-16}$$

produisent la même sortie :

$$F(k, x_1) = F(k, x_2)$$

pour toute clé k.

Cette dépendance structurelle est **incompatible** avec une fonction pseudo-aléatoire, puisque les collisions ne devraient pas être prévisibles.

Construction d'un distingueur : On définit le distingueur $\mathcal{D}$ comme suit :

1. Tire deux entrées x_1, x_2 aléatoires telles que $x_1 \oplus x_2 = 2^{n-16}$.
2. Envoie (x_1, x_2) à l'oracle.
3. Si $F(x_1) = F(x_2)$, alors $\mathcal{D}$ conclut qu'il s'agit de $F(k, \cdot)$.
4. Sinon, $\mathcal{D}$ conclut qu'il s'agit d'une fonction aléatoire.

Analyse de l'avantage : Pour la fonction $F(k, \cdot)$, on a :

$$\Pr[F(k, x_1) = F(k, x_2)] = 1$$

alors que pour une fonction aléatoire :

$$\Pr[\text{collision}] = 2^{-(n-16)}$$

Ainsi, l'avantage du distingueur est :

$$\text{Adv}_F(\mathcal{D}) = 1 - 2^{-(n-16)} \approx 1$$

Conclusion : $F(k, x)$ **n'est pas une fonction pseudo-aléatoire** : il existe un distingueur efficace $\mathcal{D}$ capable de la reconnaître avec probabilité presque certaine.

$$\boxed{F(k, x) \text{ n'est pas PRF (Pseudo-Random Function).}}$$

2.3 Conclusion

Le MAC étudié montre que tronquer la sortie d'une fonction cryptographique supprime des bits d'entropie essentiels à la sécurité. La perte d'information dans les bits hauts rend la construction vulnérable : Eve peut modifier un message sans invalider le tag. De plus, la fonction dérivée $F(k, x)$ n'est pas pseudo-aléatoire, car elle présente des collisions déterministes et prévisibles.

2.4 Limites de l'implémentation et améliorations possibles

Limites du schéma MAC.

— Le MAC ne conserve que les 32 bits de poids faibles du résultat $(m \oplus k)$, ce qui permet à Eve de modifier librement les 32 bits hauts sans altérer le tag.

— L'opération XOR est linéaire et ne fournit aucune diffusion cryptographique.

— La taille du tag (32 bits) est trop faible pour résister aux collisions ou attaques par force brute.

— Aucune gestion de clé n'est prévue (rotation, renouvellement, stockage sécurisé).

Limites de l'implémentation Python.

— Le générateur de clé utilise `secrets.randbits` sans validation de l'uniformité ni gestion de l'état aléatoire.

— Le code ne gère pas d'erreurs ou de vérifications d'entrée (ex. : taille de message incorrecte).

— Le script n'intègre pas de comparaison en temps constant (*timing attack possible*).

— La simulation est purement illustrative : elle n'utilise pas de canaux réseau, de protocoles ou de stockage sécurisé.

— Les impressions (`print`) exposent les clés et tags, ce qui serait inacceptable dans une application réelle.

Améliorations possibles.
- Remplacer le schéma par une construction sûre : **HMAC-SHA256** ou **CMAC-AES**.
- Étendre le tag à au moins **128 bits**.
- Ajouter une gestion robuste des erreurs et des entrées.
- Utiliser une **comparaison en temps constant** pour la vérification.
- Simuler un environnement plus réaliste : communication Alice–Bob via un canal intercepté par Eve.

Conclusion. L'implémentation actuelle est suffisante pour démontrer la faiblesse du schéma, mais non pour un usage réel. Une approche standardisée et sécurisée (HMAC/C-MAC) serait indispensable pour garantir l'intégrité et l'authenticité des messages.

3 Cryptosystème RSA et extensions

Soit la sortie $(221, 13, 17)$ d'un algorithme GenModulus pour RSA.

3.1 Identification de N, p, q et primalité

En RSA, `GenModulus` génère typiquement (N, p, q) où :
- N est le module (produit de deux nombres premiers) ;
- p et q sont les facteurs premiers.
- Donc : $N = 221$, $p = 13$, $q = 17$

Vérifions :
- $13 \times 17 = 221$ ✓
- 13 est premier ? (divisible seulement par 1 et 13)
- 17 est premier ? (divisible seulement par 1 et 17)

3.2 Calcul de $\varphi(N)$ et taille de $\mathbb{Z}_N^*$

Pour deux premiers distincts p, q,

$$\varphi(N) = \varphi(pq) = (p-1)(q-1) = (13-1)(17-1) = 12 \cdot 16 = 192.$$

La taille du groupe multiplicatif est $|\mathbb{Z}_{221}^*| = \varphi(221) = 192$.

3.3 Choix de $e = 11$: coprimalité et exposant privé d

Vérifions $\gcd(e, \varphi(N)) = \gcd(11, 192) = 1$. Si oui. On cherche $d \equiv e^{-1} \pmod{\varphi(N)}$, i.e. $11d \equiv 1 \pmod{192}$.

Algorithme d'Euclide. On souhaite vérifier que $\gcd(e, \varphi(N)) = 1$. Pour $e = 11$ et $\varphi(N) = 192$, appliquons l'algorithme d'Euclide :

$$\begin{aligned} 192 &= 11 \times 17 + 5, \\ 11 &= 5 \times 2 + 1, \\ 5 &= 1 \times 5 + 0. \end{aligned}$$

Ainsi, $\gcd(192, 11) = 1$.

$$\boxed{e = 11 \text{ est valide car il est premier avec } \varphi(N).}$$

Algorithme d'Euclide étendu. Nous cherchons d tel que :

$$e \times d \equiv 1 \pmod{\varphi(N)}.$$

Autrement dit :

$$11 \times d \equiv 1 \pmod{192}.$$

Appliquons l'algorithme d'Euclide étendu :

$$\begin{aligned}
1 &= 11 - 5 \times 2, \\
&= 11 - (192 - 11 \times 17) \times 2, \\
&= 11 \times (1 + 34) - 192 \times 2, \\
&= 11 \times 35 - 192 \times 2.
\end{aligned}$$

Ainsi :

$$11 \times 35 \equiv 1 \pmod{192}.$$

$$\boxed{d = 35.}$$

3.4 Clés et schémas

On considère les paramètres suivants :

$$N = 221, \quad e = 11, \quad d = 35, \quad p = 13, \quad q = 17.$$

3.4.1 Chiffrement RSA "textbook" (basique)

— Clé publique : $(N, e) = (221, 11)$
— Clé privée : $(N, d) = (221, 35)$

Opérations :

$$c = m^e \bmod N = m^{11} \bmod 221,$$
$$m = c^d \bmod N = c^{35} \bmod 221.$$

Contrainte : $m \in \mathbb{Z}_N^*$, c'est-à-dire $0 < m < 221$ et $\gcd(m, N) = 1$.

3.4.2 Signature RSA "textbook"

— Clé publique (vérification) : $(N, e) = (221, 11)$
— Clé privée (signature) : $(N, d) = (221, 35)$

Opérations :

$$\sigma = m^d \bmod N = m^{35} \bmod 221,$$
$$m' = \sigma^e \bmod N = \sigma^{11} \bmod 221.$$

Vérification : $m' = m \Rightarrow$ signature valide.

Remarque : Les rôles des clés sont inversés par rapport au chiffrement :
— Pour chiffrer : on utilise la clé publique (e).
— Pour signer : on utilise la clé privée (d).

3.4.3 Chiffrement RSA-OAEP (PKCS#1)

— Clé publique : $(N, e) = (221, 11)$
— Clé privée : $(N, d) = (221, 35)$

Différence majeure : le schéma RSA-OAEP ajoute un **remplissage aléatoire (padding)** avant le chiffrement, assurant la sécurité contre de multiples attaques.

Structure du schéma OAEP :

$$\text{Message original } m \xrightarrow{\text{Padding OAEP}} m' \xrightarrow{\text{RSA}} c = (m')^e \bmod N$$

Étapes du padding :

1. Générer une chaîne aléatoire r.
2. Calculer :
$$X = (m \,||\, 0^{k_1}) \oplus G(r),$$
$$Y = r \oplus H(X),$$
où G et H sont des fonctions de hachage.
3. Construire : $m' = X||Y$.

Chiffrement RSA classique :

$$c = (m')^e \bmod N$$

Paramètres typiques :

G, H basés sur SHA-256, $k_0 = 256$ bits, k_1 : padding de zéros, Label : optionnel (souvent vide).

3.4.4 Tableau comparatif

Aspect	**RSA Chiffrement**	**RSA Signature**	**RSA-OAEP**
Clé publique	(N, e)	(N, e)	(N, e)
Clé privée	(N, d)	(N, d)	(N, d)
Usage clé publique	Chiffrer	Vérifier	Chiffrer
Usage clé privée	Déchiffrer	Signer	Déchiffrer
Padding	Aucun	Aucun	OAEP (aléatoire)
Déterministe ?	Oui	Oui	Non (aléatoire)
Sécurité	Faible	Faible	Forte

3.4.5 Pourquoi RSA-OAEP est plus sécurisé ?

Faiblesses du RSA "textbook" :

— **Déterministe** : le même message produit toujours le même chiffré.
— **Malléabilité** : si $c = m^e$, alors $c' = c \times 2^e = (2m)^e$.
— **Pas de protection contre les petits exposants.**
— **Attaques par dictionnaire** possibles.

Améliorations apportées par RSA-OAEP :

— **Randomisation** : le padding OAEP rend le chiffrement non-déterministe.

— **Intégrité** : les fonctions H et G détectent les modifications.

— **Sécurité prouvée** : RSA-OAEP est *IND-CCA2 sécurisé*.

3.4.6 Exemple conceptuel (avec nos valeurs)

RSA Textbook :

$$m = 42, \quad c = 42^{11} \bmod 221 = 168.$$

RSA-OAEP :

$$m = 42, \quad r = \text{aléatoire}, \quad m' = \mathrm{OAEP}(42, r), \quad c = (m')^{11} \bmod 221.$$

Même message $m = 42 \rightarrow$ **chiffrés différents à chaque fois.**

3.5 Analyse de sécurité

3.5.1 Partie 1 — Attaque par dictionnaire

Contexte. Un adversaire intercepte un chiffré :

$$c = m^e \bmod N$$

Il sait que le message original $m \in M$, où M est un ensemble de 1000 messages possibles, et il connaît la clé publique (N, e).

Principe de l'attaque. RSA *textbook* étant déterministe, un même message m produit toujours le même chiffré c. Un adversaire peut donc construire un dictionnaire de correspondance entre les chiffrés et leurs messages.

Algorithme de l'attaque.

1. **Pré-calcul :** Pour chaque message $m_i \in M$, calculer :

   $$c_i = m_i^e \bmod N$$

   puis stocker la paire (c_i, m_i) dans un dictionnaire D.
2. **Interception :** Lorsqu'un chiffré c est observé, chercher c dans D.
3. **Résultat :** Si $c = c_i$, alors le message est m_i.

Complexité.

$$\text{Pré-calcul : } O(|M|) = O(1000), \quad \text{Attaque : } O(1) \text{ (recherche dans un hash table).}$$

Exemple concret. Avec nos paramètres RSA :

$$N = 221, \quad e = 11$$

et un petit espace de messages $M = \{2, 3, 5, 7, 11, 13, \dots\}$:

$$\begin{aligned} m = 2 &\Rightarrow c = 2^{11} \bmod 221 = 160, \\ m = 3 &\Rightarrow c = 3^{11} \bmod 221 = 200, \\ m = 5 &\Rightarrow c = 5^{11} \bmod 221 = 72, \\ m = 42 &\Rightarrow c = 42^{11} \bmod 221 = 168. \end{aligned}$$

Dictionnaire obtenu :

$$D = \{160 \mapsto 2,\ 200 \mapsto 3,\ 72 \mapsto 5,\ 168 \mapsto 42, \dots\}$$

Si l'adversaire intercepte $c = 168$, il trouve immédiatement $m = 42$.

Pourquoi cette attaque fonctionne-t-elle ?

- **RSA textbook est déterministe** : le même message produit toujours le même chiffré.
- **Espace de messages limité** : seulement 1000 messages possibles.
- **Clé publique connue** : n'importe qui peut chiffrer sans la clé privée.
- **Absence de padding** : aucune randomisation ne protège le message.

3.5.2 Partie 2 — Contre-mesures

1. Utiliser RSA-OAEP (recommandé).

- Ajoute un **padding aléatoire** avant le chiffrement :

$$c = (OAEP(m, r))^e \bmod N$$

  où r est une valeur aléatoire différente à chaque chiffrement.
- Le même message produit des chiffrés différents :

$$m = 42 \Rightarrow \begin{cases} c_1 = (OAEP(42, r_1))^e \bmod N, \\ c_2 = (OAEP(42, r_2))^e \bmod N \end{cases} \Rightarrow c_1 \neq c_2.$$

L'adversaire ne peut plus pré-calculer de dictionnaire, car r est inconnu.

2. Ajouter un sel (*salt*) aléatoire. Avant chiffrement :

$$m' = m \,||\, s, \quad s \text{ aléatoire (128 bits par exemple).}$$

Deux chiffrés pour le même message diffèrent :

$$Enc(m||s_1) \neq Enc(m||s_2).$$

3. Chiffrement hybride (RSA + AES).

1. Générer une clé symétrique aléatoire k.
2. Chiffrer le message avec AES : $c_{\text{msg}} = AES_k(m)$.
3. Chiffrer la clé avec RSA : $c_{\text{key}} = k^e \bmod N$.
4. Transmettre : $(c_{\text{key}}, c_{\text{msg}})$.

Avantages :

— Chaque chiffrement génère une clé aléatoire unique.
— Adapté aux messages longs.
— Utilisé dans des protocoles sécurisés comme TLS ou PGP.

4. Agrandir l'espace de messages. Inclure du hasard ou un horodatage :

$$m' = m \,||\, \text{random}(128) \,||\, \text{timestamp}.$$

L'espace des messages devient pratiquement infini, ce qui empêche les attaques par dictionnaire.

5. Utiliser une signature plutôt qu'un chiffrement. Si le but est l'authentification :

$$\sigma = Sign(m) = m^d \bmod N,$$

et on envoie (m, σ) au lieu de chiffrer m.

3.5.3 Tableau comparatif des contre-mesures

Contre-mesure	**Efficacité**	**Complexité**	**Standard**
RSA-OAEP	Excellente	Moyenne	PKCS#1 v2.x
Sel aléatoire	Bonne	Faible	Non standard
Chiffrement hybride (RSA + AES)	Excellente	Moyenne	TLS, PGP
Agrandir l'espace de messages	Acceptable	Faible	Ad-hoc
Signature au lieu de chiffrement	Bonne*	Faible	Standards variés

*Si le but est l'authentification, non la confidentialité.

3.5.4 Recommandation finale

Pour un usage en production :

— Toujours utiliser **RSA-OAEP** (ou RSA-KEM).
— Ne jamais utiliser RSA textbook :
  — pour chiffrer des messages courts ou prévisibles,
  — sans padding,
  — ou dans des applications réelles de sécurité.
— Utiliser des bibliothèques éprouvées :
  — Python : `cryptography`, `PyCryptodome`,
  — Java : `javax.crypto` avec `RSA/ECB/OAEPPadding`,
  — OpenSSL : option `-oaep`.

3.6 Extension au cas multi-premier

3.6.1 Partie 1 — Expression de $\varphi(N)$ pour $N = pqr$

Dans RSA standard, $N = pq$ avec p, q premiers ; la fonction d'Euler est

$$\varphi(pq) = \varphi(p)\,\varphi(q) = (p-1)(q-1).$$

Pour le cas multi-premier, on considère $N = pqr$ avec p, q, r *premiers distincts*. La multiplicativité de φ (lorsque les facteurs sont premiers entre eux) donne

$$\varphi(N) = \varphi(pqr) = \varphi(p)\,\varphi(q)\,\varphi(r) = (p-1)(q-1)(r-1).$$

Plus généralement, si $N = \prod_{i=1}^{k} p_i$ avec p_i premiers distincts,

$$\varphi(N) = \prod_{i=1}^{k}(p_i - 1).$$

3.6.2 Partie 2 — Exemple numérique ($p = 7,\ q = 11,\ r = 13$)

Module.

$$N = pqr = 7 \cdot 11 \cdot 13 = 1001.$$

Fonction d'Euler.

$$\varphi(N) = (p-1)(q-1)(r-1) = 6 \cdot 10 \cdot 12 = 720.$$

Clés RSA possibles. On souhaite e tel que $\gcd(e, \varphi(N)) = 1$. Par exemple $e = 7$ convient. On cherche $d \equiv e^{-1} \pmod{\varphi(N)}$:

$$7d \equiv 1 \pmod{720} \quad \Rightarrow \quad d = 103 \quad (\text{car } 7 \cdot 103 = 721 = 720 + 1).$$

Clés :

$$\text{pk} = (N, e) = (1001, 7), \qquad \text{sk} = (N, d) = (1001, 103).$$

Propriété (rappel). Pour tout $m \in \mathbb{Z}_N^*$ et tout entier $k \geq 0$,

$$m^{k\varphi(N)+1} \equiv m \pmod{N}, \quad \text{donc ici} \quad m^{720k+1} \equiv m \pmod{1001}.$$

3.6.3 Partie 3 — Implications sur la sécurité et la performance

Idée générale. À *taille de module N fixée*, utiliser plus de facteurs premiers distincts $(p, q, r, \dots)$ implique des *facteurs individuels plus petits*, ce qui **facilite la factorisation** par des algorithmes modernes. En contrepartie, le déchiffrement/signature peut être **accéléré** via le CRT appliqué à davantage de sous-modules.

Comparaison qualitative (même taille N).

Aspect	RSA à 2 facteurs	RSA à 3 facteurs
Module	$N = pq$	$N = pqr$
$\varphi(N)$	$(p-1)(q-1)$	$(p-1)(q-1)(r-1)$
Taille des facteurs (à $\|N\|$ fixé)	Plus grands	Plus petits
Sécurité (facto. à $\|N\|$ fixé)	Plus élevée	Plus faible
Performance du déchiffrement (CRT)	$\sim 4\times$ plus rapide que sans CRT	Légèrement plus rapide que pq
Standards (NIST / PKCS#1)	Recommandé	Non recommandé en général

Remarques.

— Avec k facteurs et $|N|$ fixé, chaque p_i est plus petit : certains algorithmes de factorisation (p. ex. ECM) deviennent plus efficaces.

— Pour maintenir le *même niveau de sécurité* que RSA à deux facteurs, il faut *augmenter la taille de* N en multi-premier, ce qui annule l'avantage de performance.

— Historiquement, le RSA multi-premier a été envisagé pour des gains de performance (smartcards), mais il est **peu recommandé** dans les standards modernes.

3.6.4 Tableau récapitulatif (exemple $p = 7,\ q = 11,\ r = 13$)

Critère	RSA à 2 facteurs	RSA à 3 facteurs (exemple)
N	pq	$7 \times 11 \times 13 = 1001$
$\varphi(N)$	$(p-1)(q-1)$	$6 \times 10 \times 12 = 720$
$\|\mathbb{Z}_N^*\|$	$\varphi(N)$	720
Sécurité (à $\|N\|$ identique)	Plus élevée	Plus faible
Performance (CRT)	$\sim 4\times$ plus rapide que sans CRT	Légèrement supérieure à pq
Standardisation	Conforme NIST / PKCS#1	Généralement non recommandée

3.6.5 Conclusion

Pour $N = pqr$, on a

$$\varphi(N) = (p-1)(q-1)(r-1).$$

L'exemple $(7, 11, 13)$ donne $N = 1001$ et $\varphi(N) = 720$. À *taille de module identique*, RSA multi-premier offre une performance CRT légèrement meilleure mais **réduit la sécurité** (facteurs plus petits donc factorisation facilitée). Les standards actuels recommandent le RSA à **deux facteurs** pour les applications de sécurité.

4 Conclusion générale

Ce travail pratique nous a permis d'explorer plusieurs aspects essentiels de la cryptographie asymétrique et symétrique, en mettant en lumière les failles de conception et les bonnes pratiques à adopter.

Dans la première partie, l'analyse du schéma MAC défaillant a montré qu'une simple troncature ou un mauvais choix d'opérations peut compromettre totalement l'intégrité d'un message : Eve pouvait modifier les bits de poids forts sans changer le tag, révélant une **faille structurelle critique**.

Les sections suivantes ont approfondi la sécurité du chiffrement RSA. L'étude du *RSA textbook* a mis en évidence son caractère déterministe et sa vulnérabilité aux attaques par dictionnaire, tandis que le schéma **RSA-OAEP** s'est révélé beaucoup plus robuste grâce à son padding aléatoire et sa sécurité prouvée (IND-CCA2).

L'exploration du cas **multi-premier** a permis de comprendre que, malgré un léger gain de performance, la sécurité diminue fortement car les facteurs premiers sont plus petits et donc plus faciles à factoriser. Cette configuration est d'ailleurs déconseillée par les standards modernes (NIST, PKCS#1).

En résumé, ce TP a permis de :

— Identifier les erreurs de conception d'un MAC non sécurisé ;
— Comprendre les limites du RSA classique et ses vulnérabilités pratiques ;
— Évaluer les avantages du RSA-OAEP et des schémas hybrides (RSA + AES) ;
— Analyser l'impact de la structure du module sur la sécurité globale.

Conclusion finale : La sécurité cryptographique ne dépend pas seulement des formules mathématiques, mais surtout des **détails d'implémentation et des protocoles utilisés**. Seules les versions standardisées comme RSA-OAEP doivent être utilisées en pratique, et toute simplification du schéma RSA conduit à une perte significative de sécurité.

5 Références

Références

[1] M. Fiset, *Notes de cours et diapositives du cours IFT814 – Cryptographie*, Université de Sherbrooke, 2025.

[2] Cryptoool 2, *Outil éducatif pour la cryptographie*. Disponible sur : https://www.cryptool.org/en/ct2

[3] GeeksforGeeks, *RSA Algorithm (Cryptography)*. Disponible sur : https://www.geeksforgeeks.org/computer-networks/rsa-algorithm-cryptography/

[4] Wikipédia, *Code d'authentification de message (MAC)*. https://fr.wikipedia.org/wiki/Code_d%27authentification_de_message

[5] Wikipédia, *Optimal Asymmetric Encryption Padding*. https://fr.wikipedia.org/wiki/Optimal_Asymmetric_Encryption_Padding