Devoir 3

IFT814 – Cryptographie

Chargé de cours : Martin Fiset

Groupe 1

Nom	CIP
Mamadou Senghor	senm1912
Clément Dumas	dumc4449
Rafael Dhaene	dhar2976
Ricardo Beaujour	bear4830

Université de Sherbrooke
Département d'informatique

Automne 2025

Date de remise : 31 Octobre 2025

Table des matières

Résumé

Ce rapport présente le **Devoir 3** du cours **IFT814 – Cryptographie**, portant sur l'étude et l'implémentation de trois grands volets fondamentaux de la cryptographie moderne : **l'échange de clé Diffie–Hellman**, **le chiffrement RSA "textbook"**, et **le mécanisme de signature RSA classique**.

Première partie — Diffie–Hellman Nous avons développé et exécuté un **script Python** simulant deux scénarios :
— un **scénario passif**, où l'attaquant Eve se limite à observer les échanges entre Alice et Bob ;
— un **scénario actif (MITM)**, où Eve intercepte et modifie les valeurs transmises.
Les traces d'exécution démontrent que dans le cas passif, Alice et Bob obtiennent une clé commune ($k_A = k_B$), alors qu'en présence d'une attaque MITM, les clés diffèrent ($k_A \neq k_B$) et Eve peut établir deux clés indépendantes avec chacun. L'analyse formelle des instances de session et l'étude d'un oracle de logarithme discret partiel illustrent les conditions de rupture du protocole et la probabilité d'exploitation selon le nombre de sessions.

Deuxième partie — RSA "textbook" (chiffrement) Une implémentation expérimentale du chiffrement et du déchiffrement RSA a été réalisée avec les paramètres $N = 221$, $e = 7$ et $d = 55$. Plusieurs messages tests ($m \in \{2, 6, 13, 17, 0, 1, 221\}$) ont été chiffrés et déchiffrés afin d'observer les comportements anormaux du mode "textbook", notamment :
— l'absence de remplissage (*padding*) entraîne des collisions et des failles structurelles ;
— certains messages conduisent à des cas dégénérés (ex. $m = 0, 1, N$) où la sécurité est compromise.
L'étude confirme que le schéma RSA de base ne satisfait pas les propriétés d'indistinguabilité ni de résistance aux attaques choisissant le texte chiffré (IND–CCA2).

Troisième partie — RSA "textbook" (signatures) Nous avons également implémenté et testé les fonctions `Sign2` et `Verif2` afin de comprendre la mécanique des signatures RSA. Des expériences de manipulation de paires (message, signature) montrent la **vulnérabilité à la réutilisation et à la combinaison de signatures** en raison de l'absence de hachage préalable ($m^d \bmod N$ direct). Des attaques de type *forgeabilité universelle* sont donc possibles lorsque la vérification repose uniquement sur la relation $s^e \equiv m \bmod N$.

Conclusion Les trois parties du devoir permettent d'illustrer les limites des schémas "textbook" (non sécurisés sans encodage, hachage ou authentification) et soulignent la nécessité d'intégrer des mécanismes modernes comme :
— des signatures et certificats numériques pour authentifier Diffie–Hellman ;
— des schémas RSA à remplissage probabiliste sécurisés (RSA–OAEP, RSA–PSS) ;
— des fonctions de hachage cryptographique dans les opérations de signature.
Ce rapport met ainsi en évidence la différence entre la **sécurité théorique des fondements mathématiques** et la **sécurité pratique des protocoles implémentés**.

1 Partie 1 Protocole d'échange de clé — Variante Diffie–Hellman

1.1 Description et implémentation du protocole

Le protocole de base vise à permettre à deux entités, **Alice** et **Bob**, de calculer une clé secrète commune sur un canal non sécurisé. Les paramètres imposés pour cette variante sont :

$$g = 5, \qquad p = 2^{61} - 1$$

où p est un **nombre premier de Mersenne**, et g un **générateur du groupe multiplicatif** $\mathbb{Z}_p^*$. Les secrets x et y sont choisis aléatoirement sur 40 bits :

$$x, y \in_R [1, 2^{40}]$$

Déroulement du protocole

1. Alice calcule $h_A = g^x \bmod p$ et l'envoie à Bob.
2. Bob calcule $h_B = g^y \bmod p$ et l'envoie à Alice.
3. Les deux parties obtiennent la clé partagée :

$$k_A = h_B^x \bmod p = g^{xy} \bmod p$$

$$k_B = h_A^y \bmod p = g^{xy} \bmod p$$

Listing 1 – Implémentation du protocole Diffie–Hellman (variante Mersenne)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Implémentation du protocole DiffieHellman (variante Mersenne)
Paramètres :
 - p = 2**61 - 1 (nombre de Mersenne)
 - g = 5
 - Exposants secrets sur 40 bits
 - Utilise exponentiation modulaire rapide (square-and-multiply)
"""

from secrets import randbits

# Paramètres publics
P = 2**61 - 1 # nombre de Mersenne
G = 5 # base publique

# --- Exponentiation modulaire rapide ---
def exp_mod(base: int, exp: int, mod: int) -> int:
    """Calcule (base ** exp) mod mod par square-and-multiply."""
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result
```

```python

# --- Étapes du protocole ---
def alice_generate():
    x = randbits(40) # secret d'Alice
    hA = exp_mod(G, x, P) # hA = g^x mod p
    return x, hA

def bob_generate():
    y = randbits(40) # secret de Bob
    hB = exp_mod(G, y, P) # hB = g^y mod p
    return y, hB

def alice_compute_key(x, hB):
    return exp_mod(hB, x, P) # kA = hB^x mod p

def bob_compute_key(y, hA):
    return exp_mod(hA, y, P) # kB = hA^y mod p

# --- Simulation ---
if __name__ == "__main__":
    print("=== Protocole DiffieHellman (variante Mersenne) ===")
    x, hA = alice_generate()
    y, hB = bob_generate()
    print(f"Alice envoie hA = {hA}")
    print(f"Bob envoie hB = {hB}")

    kA = alice_compute_key(x, hB)
    kB = bob_compute_key(y, hA)
    print("\n=== Résultats ===")
    print(f"Clé (Alice) : {kA}")
    print(f"Clé (Bob) : {kB}")
    print(f"Clés identiques ? {'Oui' if kA == kB else 'Non'}")
```

```
=== Protocole Diffie-Hellman (variante Mersenne) ===
Alice envoie hA = 432437667009137884
Bob envoie   hB = 1300346583603524129

=== Résultats ===
Clé partagée (Alice) : 131570171525173459
Clé partagée (Bob)   : 131570171525173459
Clés identiques ? ✅ Oui
PS C:\Users\Utilisateur\Desktop\Master IA et Science_UDS\Automne 2025\Cryptographie\TP3>
```

FIGURE 1 – Exécution du protocole Diffie–Hellman (variante Mersenne) : génération des secrets, échange des valeurs publiques et obtention de la clé partagée identique pour Alice et Bob.

1.2 Question (a) — Exécution du protocole dans deux contextes distincts

Dans cette première expérimentation, nous exécutons le protocole Diffie–Hellman (variante Mersenne) dans deux contextes : une écoute passive et une attaque active de type *Man-In-The-Middle* (MITM).

1.2.1 Scénario 1 — Écoute passive

Principe Eve intercepte et observe les messages échangés entre Alice et Bob sans les modifier. Elle capture les valeurs publiques h_A et h_B, mais ne peut pas déduire les secrets privés ni la clé commune sans résoudre un logarithme discret.

Code Python exécuté :

Listing 2 – Scénario 1 — Écoute passive (Eve observe sans intervenir)

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Scénario 1  Écoute passive (Eve observe hA et hB).
Usage:
  python scenario_passif.py # secrets aléatoires
  python scenario_passif.py --fixed "alice:0x0123456789,bob:0x0abcde1234"
"""
from secrets import randbits
import argparse

# Paramètres publics
P = 2**61 - 1
G = 5
BITS_SECRET = 40

def exp_mod(base: int, exp: int, mod: int) -> int:
    """Exponentiation modulaire rapide (square-and-multiply)."""
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result

def parse_fixed(s: str):
    d = {}
    if not s:
        return d
    for part in s.split(","):
        if ":" not in part:
            raise ValueError("Format --fixed invalide, attendu 'alice:val,bob:val'")
        k, v = part.split(":", 1)
        k = k.strip().lower()
        v = v.strip()
        d[k] = int(v, 0) # accepte décimal ou 0x...
    return d

def rand40():
    return randbits(BITS_SECRET) & ((1<<BITS_SECRET)-1)

def fmt(n: int) -> str:
    return f"{n} (dec) / 0x{n:010x} (hex)"

def run_passif(x: int, y: int):
```

```
    hA = exp_mod(G, x, P)
    hB = exp_mod(G, y, P)
    kA = exp_mod(hB, x, P)
    kB = exp_mod(hA, y, P)

    print("="*72)
    print("SCÉNARIO 1  ÉCOUTE PASSIVE")
    print("="*72)
    print(f"Alice : secret x = {fmt(x)}")
    print(f"Bob : secret y = {fmt(y)}\n")
    print("Messages observés sur le réseau (par Eve) :")
    print(f" Alice -> Bob : hA = g^x mod p = {hA}")
    print(f" Bob -> Alice : hB = g^y mod p = {hB}\n")
    print("Valeurs locales et clés finales :")
    print(f" Alice calcule kA = (hB)^x mod p = {kA}")
    print(f" Bob calcule kB = (hA)^y mod p = {kB}")
    print(f" Vérification : kA == kB ? {'Oui' if kA==kB else 'Non'}\n")
    print("Transcript  Alice :")
    print(f" Secret x = {fmt(x)}")
    print(f" hA = {hA}")
    print(f" Reçoit hB = {hB}")
    print(f" kA = {kA}\n")
    print("Transcript  Bob :")
    print(f" Secret y = {fmt(y)}")
    print(f" hB = {hB}")
    print(f" Reçoit hA = {hA}")
    print(f" kB = {kB}\n")
    print("Transcript  Eve (écoute passive) :")
    print(f" Observé : hA = {hA}, hB = {hB}")
    print(" Remarque : récupérer x ou y nécessite de résoudre un logarithme discret
        (~2^40 opérations).")
    print("="*72)

if __name__ == "__main__":
    ap = argparse.ArgumentParser(description="Scénario 1  écoute passive")
    ap.add_argument("--fixed", type=str, default="", help='Ex: "alice:0x0123456789,bob
        :0x0abcde1234"')
    args = ap.parse_args()

    fixed = parse_fixed(args.fixed)
    x = fixed.get("alice", rand40())
    y = fixed.get("bob", rand40())

    run_passif(x, y)
```

Résultats expérimentaux

- **Valeurs locales :**
  - Alice : secret x, valeur publique $h_A = g^x \bmod p$
  - Bob : secret y, valeur publique $h_B = g^y \bmod p$
- **Messages observés par Eve :**

$$\text{Alice} \to \text{Bob} : h_A, \quad \text{Bob} \to \text{Alice} : h_B$$

- **Clés finales :**

$$k_A = (h_B)^x \bmod p, \quad k_B = (h_A)^y \bmod p, \quad k_A = k_B$$

— **Transcripts :**
  — **Alice :** génère x, calcule h_A, reçoit h_B, obtient k_A.
  — **Bob :** génère y, calcule h_B, reçoit h_A, obtient k_B.
  — **Eve :** observe h_A, h_B sans pouvoir calculer la clé.

```
======================================================================
SCÉNARIO 1 — ÉCOUTE PASSIVE
======================================================================
Alice : secret x = 874405311542 (dec) / 0xcb969e3436 (hex)
Bob   : secret y = 180003483270 (dec) / 0x29e90b2e86 (hex)

Messages observés sur le réseau (par Eve) :
  Alice -> Bob : hA = g^x mod p = 1439583391124477635
  Bob   -> Alice : hB = g^y mod p = 1177399103120143943

Valeurs locales et clés finales :
  Alice calcule kA = (hB)^x mod p = 113576073665850599
  Bob   calcule kB = (hA)^y mod p = 113576073665850599
  Vérification : kA == kB ? Oui

Transcript — Alice :
  Secret x = 874405311542 (dec) / 0xcb969e3436 (hex)
  hA = 1439583391124477635
  Reçoit hB = 1177399103120143943
  kA = 113576073665850599

Transcript — Bob :
  Secret y = 180003483270 (dec) / 0x29e90b2e86 (hex)
  hB = 1177399103120143943
  Reçoit hA = 1439583391124477635
  kB = 113576073665850599

Transcript — Eve (écoute passive) :
  Observé : hA = 1439583391124477635, hB = 1177399103120143943
  Remarque : récupérer x ou y nécessite de résoudre un logarithme discret (~2^40 opérations).
======================================================================
```

FIGURE 2 – Trace d'exécution — Scénario 1 : écoute passive (Eve observe h_A et h_B sans intervenir).

1.2.2 Scénario 2 — Attaque de l'homme du milieu (MITM)

Principe Eve intercepte les messages d'Alice et de Bob, puis crée deux sessions distinctes :

— avec Alice, en se faisant passer pour Bob (secret e_2) ;
— avec Bob, en se faisant passer pour Alice (secret e_1).

Elle envoie des valeurs forgées $h'_A = g^{e_1} \bmod p$ et $h'_B = g^{e_2} \bmod p$, ce qui lui permet de calculer deux clés différentes :

$$k_{AE} = g^{xe_2} \bmod p, \qquad k_{BE} = g^{ye_1} \bmod p$$

Code Python exécuté :

Listing 3 – Scénario 2 — Attaque MITM (Eve forge h'_A et h'_B)

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""Scénario 2  MITM (Eve forge hA' et hB').
```

```
Usage:
  python scenario_mitm.py
  python scenario_mitm.py --fixed "alice:0x0123456789,bob:0x0abcde1234,eve1:0
      x00fedcba98,eve2:0x003456789a"
"""
from secrets import randbits
import argparse

# Paramètres publics
P = 2**61 - 1
G = 5
BITS_SECRET = 40

def exp_mod(base: int, exp: int, mod: int) -> int:
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result

def parse_fixed(s: str):
    d = {}
    if not s:
        return d
    for part in s.split(","):
        if ":" not in part:
            raise ValueError("Format --fixed invalide, attendu 'alice:val,...'")
        k, v = part.split(":", 1)
        k = k.strip().lower()
        v = v.strip()
        d[k] = int(v, 0)
    return d

def rand40():
    return randbits(BITS_SECRET) & ((1<<BITS_SECRET)-1)

def fmt(n: int) -> str:
    return f"{n} (dec) / 0x{n:010x} (hex)"

def run_mitm(x: int, y: int, e1: int, e2: int):
    # valeurs publiques légitimes
    hA = exp_mod(G, x, P)
    hB = exp_mod(G, y, P)
    # valeurs forgées par Eve
    hA_p = exp_mod(G, e1, P)
    hB_p = exp_mod(G, e2, P)
    # clés calculées par les victimes (avec valeurs forgées)
    kA = exp_mod(hB_p, x, P)
    kB = exp_mod(hA_p, y, P)
    # clés d'Eve pour chaque session
    k_AE = exp_mod(hA, e2, P) # session avec Alice
    k_BE = exp_mod(hB, e1, P) # session avec Bob

    print("="*72)
    print("SCÉNARIO 2  MITM (Attaque de l'homme du milieu)")
```

```
    print("="*72)
    print(f"Alice : secret x = {fmt(x)}")
    print(f"Bob : secret y = {fmt(y)}")
    print(f"Eve : secrets e1 = {fmt(e1)}, e2 = {fmt(e2)}\n")

    print("Flux réseau (interception / falsification) :")
    print(f" Alice -> réseau : hA = {hA} (intercepté par Eve)")
    print(f" Eve -> Bob : hA' = {hA_p} (forgé)")
    print(f" Bob -> réseau : hB = {hB} (intercepté par Eve)")
    print(f" Eve -> Alice : hB' = {hB_p} (forgé)\n")

    print("Clés calculées :")
    print(f" Alice calcule kA = (hB')^x mod p = {kA}")
    print(f" Bob calcule kB = (hA')^y mod p = {kB}")
    print(f" Eve calcule k_AE = (hA)^e2 mod p = {k_AE} (session Eve<->Alice)")
    print(f" Eve calcule k_BE = (hB)^e1 mod p = {k_BE} (session Eve<->Bob)\n")

    print(f"Vérification : kA == k_AE ? {'Oui' if kA==k_AE else 'Non'}")
    print(f" kB == k_BE ? {'Oui' if kB==k_BE else 'Non'}\n")

    print("Transcript  Alice :")
    print(f" Secret x = {fmt(x)}")
    print(f" hA = {hA}")
    print(f" Reçoit hB' = {hB_p} (forgé)")
    print(f" kA = {kA}\n")

    print("Transcript  Bob :")
    print(f" Secret y = {fmt(y)}")
    print(f" hB = {hB}")
    print(f" Reçoit hA' = {hA_p} (forgé)")
    print(f" kB = {kB}\n")

    print("Transcript  Eve (attaquante) :")
    print(f" Intercepte hA = {hA}, hB = {hB}")
    print(f" Envoie hA' = {hA_p} àBob et hB' = {hB_p} àAlice")
    print(f" Clés : k_AE = {k_AE}, k_BE = {k_BE}")
    print(" Eve peut lire/modifier les messages entre Alice et Bob.\n")
    print("="*72)

if __name__ == "__main__":
    ap = argparse.ArgumentParser(description="Scénario 2  MITM")
    ap.add_argument("--fixed", type=str, default="",
                    help='Ex: "alice:0x0123456789,bob:0x0abcde1234,eve1:0x00fedcba98,
                        eve2:0x003456789a"')
    args = ap.parse_args()

    fixed = parse_fixed(args.fixed)
    x = fixed.get("alice", rand40())
    y = fixed.get("bob", rand40())
    e1 = fixed.get("eve1", rand40())
    e2 = fixed.get("eve2", rand40())

    run_mitm(x, y, e1, e2)
```

Résultats expérimentaux :

```
SCÉNARIO 2 – MITM (Attaque de l'homme du milieu)
======================================================================
Alice : secret x = 152742365789 (dec) / 0x2390278a5d (hex)
Bob   : secret y = 365572206683 (dec) / 0x551dcd305b (hex)
Eve   : secrets e1 = 229987313889 (dec) / 0x358c4fe8e1 (hex), e2 = 534972229821 (dec) / 0x7c8ed468bd (hex)

Flux réseau (interception / falsification) :
  Alice -> réseau : hA = 1640796292317744313  (intercepté par Eve)
  Eve -> Bob      : hA' = 92538484190095499  (forgé)
  Bob -> réseau   : hB = 1604630331911393512  (intercepté par Eve)
  Eve -> Alice    : hB' = 1641698669385744716  (forgé)

Clés calculées :
  Alice calcule kA = (hB')^x mod p = 338895411428488004
  Bob   calcule kB = (hA')^y mod p = 2055570918092877595
  Eve calcule k_AE = (hA)^e2 mod p = 338895411428488004  (session Eve<->Alice)
  Eve calcule k_BE = (hB)^e1 mod p = 2055570918092877595  (session Eve<->Bob)

Vérification : kA == k_AE ? Oui
               kB == k_BE ? Oui

Transcript – Alice :
  Secret x = 152742365789 (dec) / 0x2390278a5d (hex)
  hA = 1640796292317744313
  Reçoit hB' = 1641698669385744716 (forgé)
  kA = 338895411428488004

Transcript – Bob :
  Secret y = 365572206683 (dec) / 0x551dcd305b (hex)
  hB = 1604630331911393512
  Reçoit hA' = 92538484190095499 (forgé)
  kB = 2055570918092877595

Transcript – Eve (attaquante) :
  Intercepte hA = 1640796292317744313, hB = 1604630331911393512
  Envoie hA' = 92538484190095499 à Bob et hB' = 1641698669385744716 à Alice
  Clés : k_AE = 338895411428488004, k_BE = 2055570918092877595
  Eve peut lire/modifier les messages entre Alice et Bob.
```

FIGURE 3 – Trace d'exécution — Scénario 2 : attaque de l'homme du milieu (Eve établit deux clés distinctes).

- **Valeurs locales :**
  - Alice : $x = 152742365789$, $h_A = 1640796292317744313$
  - Bob : $y = 365572206683$, $h_B = 1604630331911393512$
  - Eve : $e_1 = 229987313389$, $e_2 = 534972229821$
- **Messages interceptés et falsifiés :**

$$\begin{aligned} \text{Alice} \to \text{réseau} : h_A \quad & \text{(intercepté par Eve)} \\ \text{Eve} \to \text{Bob} : h'_A = g^{e_1} \bmod p & \\ \text{Bob} \to \text{réseau} : h_B \quad & \text{(intercepté)} \\ \text{Eve} \to \text{Alice} : h'_B = g^{e_2} \bmod p & \end{aligned}$$

- **Clés calculées :**

$$\begin{aligned} k_A &= (h'_B)^x \bmod p = 3388954111428488004, \\ k_B &= (h'_A)^y \bmod p = 2055570918092877595, \\ k_{AE} &= (h_A)^{e_2} \bmod p = 3388954111428488004, \\ k_{BE} &= (h_B)^{e_1} \bmod p = 2055570918092877595. \end{aligned}$$

- **Transcripts :**

— **Alice :** croit échanger avec Bob, mais reçoit h'_B (forgé) et calcule k_A.
— **Bob :** croit échanger avec Alice, reçoit h'_A (forgé) et calcule k_B.
— **Eve :** intercepte h_A, h_B, forge h'_A, h'_B et calcule k_{AE} et k_{BE}.

1.3 Question (c) — Comparaison des clés k_A et k_B selon le scénario

Observation expérimentale En comparant les résultats des deux scénarios exécutés précédemment, on observe que :

$$\begin{cases} k_A = k_B & \text{dans le scénario 1 (écoute passive)} \\ k_A \neq k_B & \text{dans le scénario 2 (MITM)} \end{cases}$$

Explication **Scénario 1 — Écoute passive.** Dans ce cas, les échanges ne sont pas altérés :

$$h_A = g^x \bmod p, \quad h_B = g^y \bmod p$$

Alice et Bob calculent alors :

$$k_A = (h_B)^x \bmod p = g^{xy} \bmod p, \qquad k_B = (h_A)^y \bmod p = g^{xy} \bmod p$$

Les deux clés sont donc identiques :

$$k_A = k_B = g^{xy} \bmod p$$

puisque la commutativité de la multiplication des exposants ($xy = yx$) est préservée. La clé commune est ainsi correcte et unique pour les deux participants.

Scénario 2 — Attaque de l'homme du milieu (MITM). Dans cette configuration, Eve intercepte et remplace les valeurs publiques :

$$h'_A = g^{e_1} \bmod p, \quad h'_B = g^{e_2} \bmod p$$

Ainsi :

$$k_A = (h'_B)^x \bmod p = g^{xe_2} \bmod p$$
$$k_B = (h'_A)^y \bmod p = g^{ye_1} \bmod p$$

Ces deux clés diffèrent car $xe_2 \neq ye_1$ en général. Alice et Bob croient avoir établi une clé commune, mais en réalité :

$$k_A = k_{AE} \quad \text{et} \quad k_B = k_{BE}$$

Eve, qui connaît e_1 et e_2, peut calculer les deux et se placer entre les communications.

Conclusion L'égalité $k_A = k_B$ n'est vérifiée que dans le scénario d'échange légitime, où les messages ne sont pas modifiés. En cas d'attaque MITM, chaque participant établit une clé différente avec l'attaquant : Eve contrôle alors deux canaux indépendants, ce qui rompt la propriété de clé commune.

1.4 Question (d) — Définition formelle des instances de session et des partenaires

En suivant la notation formelle vue dans les notes de cours (révision du modèle IKE), chaque exécution du protocole Diffie–Hellman correspond à une **instance de session** $\Pi_{A,i}$ pour Alice et $\Pi_{B,j}$ pour Bob.

Notations du modèle

— $\Pi_{A,i}$: i-ème instance du protocole exécutée par la partie Alice (Initiator)
— $\Pi_{B,j}$: j-ème instance exécutée par la partie Bob (Responder)
— sid : *Session identifier* — dérivé des messages échangés (g^x, g^y)
— pid : *Partner identifier* — identité de l'autre partie (vue locale)
— sk : clé de session calculée localement (k_A ou k_B)

Deux instances $\Pi_{A,i}$ et $\Pi_{B,j}$ sont dites **partenaires** si elles :

1. ont le même sid (mêmes messages échangés) ;
2. se désignent mutuellement comme partenaires ($\mathsf{pid}_{A,i} = B$ et $\mathsf{pid}_{B,j} = A$) ;
3. partagent la même clé de session : $\mathsf{sk}_{A,i} = \mathsf{sk}_{B,j}$.

Scénario 1 — Écoute passive

— Alice crée une instance $\Pi_{A,1}$:

$$\Pi_{A,1} = (x, h_A = g^x \bmod p, \mathsf{pid} = B, \mathsf{sk} = g^{xy} \bmod p)$$

— Bob crée une instance $\Pi_{B,1}$:

$$\Pi_{B,1} = (y, h_B = g^y \bmod p, \mathsf{pid} = A, \mathsf{sk} = g^{xy} \bmod p)$$

— Les identifiants de session sont identiques :

$$\mathsf{sid}_{A,1} = \mathsf{sid}_{B,1} = (h_A, h_B)$$

— Les partenaires sont bien définis :

$$\Pi_{A,1} \leftrightarrow \Pi_{B,1} \quad \text{car} \quad \mathsf{pid}_{A,1} = B,\ \mathsf{pid}_{B,1} = A,\ \mathsf{sk}_{A,1} = \mathsf{sk}_{B,1}$$

Ainsi, dans le scénario passif, les deux instances sont **partenaires valides** et la clé commune est partagée :

$$k_A = k_B = g^{xy} \bmod p$$

Scénario 2 — Attaque de l'homme du milieu (MITM)

— Alice crée $\Pi_{A,1}$:

$$\Pi_{A,1} = (x, h_A = g^x, \mathsf{pid} = B, \mathsf{sk}_{A,1} = g^{xe_2} \bmod p)$$

— Bob crée $\Pi_{B,1}$:

$$\Pi_{B,1} = (y, h_B = g^y, \mathsf{pid} = A, \mathsf{sk}_{B,1} = g^{ye_1} \bmod p)$$

— Eve crée deux instances distinctes :

$$\Pi_{E,1} = (e_2, h'_B = g^{e_2}, \mathsf{pid} = A, \mathsf{sk}_{E,1} = g^{xe_2} \bmod p)$$
$$\Pi_{E,2} = (e_1, h'_A = g^{e_1}, \mathsf{pid} = B, \mathsf{sk}_{E,2} = g^{ye_1} \bmod p)$$

— Identifiants de session :

$$\mathsf{sid}_{A,1} = (h_A, h'_B), \quad \mathsf{sid}_{B,1} = (h'_A, h_B)$$

Ces deux identifiants sont différents, donc $\Pi_{A,1}$ et $\Pi_{B,1}$ ne sont pas partenaires.
Les seules relations de partenariat valides selon le modèle sont :

$$\Pi_{A,1} \leftrightarrow \Pi_{E,1} \qquad \text{et} \qquad \Pi_{B,1} \leftrightarrow \Pi_{E,2}$$

puisque :

$$\mathsf{sk}_{A,1} = \mathsf{sk}_{E,1} = g^{xe_2} \bmod p \quad \text{et} \quad \mathsf{sk}_{B,1} = \mathsf{sk}_{E,2} = g^{ye_1} \bmod p$$

Conclusion

- **Scénario 1 (passif) :** $\Pi_{A,1}$ et $\Pi_{B,1}$ sont partenaires légitimes, partageant la même clé et le même identifiant de session.
- **Scénario 2 (MITM) :** $\Pi_{A,1}$ et $\Pi_{B,1}$ ne sont pas partenaires. Eve crée deux paires indépendantes $(\Pi_{A,1}, \Pi_{E,1})$ et $(\Pi_{B,1}, \Pi_{E,2})$, ce qui rompt la correspondance entre Alice et Bob selon la définition formelle du modèle de sécurité IKE.

1.5 Question (e) — Oracle partiel de logarithme discret : stratégie et probabilité de succès

Énoncé restreint On suppose qu'Eve dispose d'un oracle O qui, pour une fraction q des éléments aléatoires $u \in \mathbb{Z}_p^*$, retourne $\log_5(u) \pmod{p-1}$. Dans l'énoncé $q \approx 0.01$ (environ 1%).

Stratégie d'Eve (shifting multiplicatif)

Soit $h = 5^x \bmod p$ une valeur publique interceptée (issue d'une session d'Alice ou Bob). Eve veut retrouver x. Plutôt que d'espérer que $O(h)$ retourne directement x (probabilité q), Eve génère des *décalages multiplicatifs* :

$$u_t \;=\; h \cdot 5^t \bmod p \;=\; 5^{x+t} \bmod p, \qquad t \in T,$$

où $T = \{t_0, \ldots, t_{N-1}\}$ est un ensemble de décalages choisis (par exemple $t = 0, 1, 2, \ldots$). Pour chaque t elle interroge l'oracle $O(u_t)$:

- si $O(u_t)$ renvoie s tel que $5^s \equiv u_t \pmod{p}$, alors $s \equiv x + t \pmod{p-1}$ et Eve déduit
$$x \equiv s - t \pmod{p-1},$$
c'est-à-dire que le secret est retrouvé ;
- sinon Eve essaie le prochain $t \in T$.

Cette technique multiplie les occasions pour lesquelles l'oracle peut réussir, puisque chaque u_t a indépendamment une probabilité q d'être « résoluble » par O.

Probabilité de succès pour une session

Supposons qu'Eve teste N décalages indépendants t pour une même valeur h. Si chaque requête a une probabilité q de succès et que les succès sont (approximativement) indépendants, la probabilité d'échec après N essais est $(1-q)^N$. La probabilité de succès pour cette session est donc :

$$P_{\text{session}}(N) \;=\; 1 - (1-q)^N.$$

Cas pratique (valeurs numériques, $q = 0.01$) :

$$\begin{aligned}
N = 69 &\Rightarrow P_{\text{session}} \approx 1 - (0.99)^{69} \approx 0.50 \;(\;50\%) \\
N = 100 &\Rightarrow P_{\text{session}} \approx 1 - (0.99)^{100} \approx 0.634 \;(\;63.4\%) \\
N = 300 &\Rightarrow P_{\text{session}} \approx 1 - (0.99)^{300} \approx 0.9505 \;(\;95.05\%) \\
N = 458 &\Rightarrow P_{\text{session}} \approx 1 - (0.99)^{458} \approx 0.9899 \;(\;98.99\%)
\end{aligned}$$

Probabilité de succès après n sessions indépendantes

Si Eve observe n sessions indépendantes et pour chacune effectue N essais (même politique de décalages, ou décalages différents) et si l'ensemble des essais sont mutuellement indépendants, alors la probabilité qu'elle *échoue* sur toutes les n sessions est $(1-q)^{Nn}$. La probabilité d'au moins un succès (récupérer au moins un secret parmi les n sessions) est donc :

$$P_{\text{au moins 1 réussite}}(N, n) \;=\; 1 - (1-q)^{Nn}.$$

Exemples (avec $q = 0.01$) :

$N = 300,\ n = 1 \;\Rightarrow\; P \approx 1 - (0.99)^{300} \approx 0.9505\ (95.05\%)$
$N = 300,\ n = 5 \;\Rightarrow\; P \approx 1 - (0.99)^{1500} \approx 1 - e^{-15.0755} \approx 0.9999997\ (\approx 99.99997\%)$
$N = 100,\ n = 10 \;\Rightarrow\; P \approx 1 - (0.99)^{1000} \approx 1 - e^{-10.0503} \approx 0.9999567\ (\approx 99.9957\%)$

(les approximations exponentielles utilisent $(1-q)^m \approx e^{-qm}$ pour $q \ll 1$.)

Hypothèses, limites et optimisations

— **Indépendance :** la formule $1-(1-q)^{Nn}$ suppose que les réponses de l'oracle sur valeurs différentes sont indépendantes et que la probabilité q s'applique de façon identique aux éléments $u_t = h \cdot 5^t$. Ceci est raisonnable si l'oracle « reconnaît » environ q des éléments uniformément.
— **Coût et discrétion :** faire des centaines de requêtes par session peut être coûteux ou détectable ; Eve peut répartir ses requêtes dans le temps ou paralléliser si l'oracle le permet.
— **Optimisations :** choisir les t de manière non triviale (randomisés, batches), exploiter plusieurs sessions simultanément, ou combiner avec d'autres vulnérabilités améliore la pratique.
— **Applicabilité MITM :** l'attaque est utile aussi en écoute passive. En MITM, Eve connaît déjà e_1, e_2 mais retrouver x ou y peut servir pour attaque prolongée ou corrélation d'identités.

Conclusion Avec un oracle résolvant environ 1% des logarithmes, la stratégie de *shifting* (tester $u_t = h \cdot 5^t$) permet à Eve d'augmenter fortement ses chances. En pratique, quelques centaines de requêtes par valeur interceptée donnent une probabilité de succès très élevée (95% pour $N \approx 300$). Si Eve observe n sessions et dépense N requêtes par session, sa probabilité globale d'obtenir au moins un secret est

$$1 - (1-q)^{Nn}.$$

2 Partie 2 — Analyse du chiffrement “textbook” RSA avec module spécifique

2.1 Question(a) — Implémentation des algorithmes de chiffrement et déchiffrement

On considère le schéma de chiffrement RSA “textbook” défini par les trois algorithmes :

$$\Pi_{\text{ch-t-RSA}} = (\texttt{Gen1}, \texttt{E1}, \texttt{D1})$$

et les paramètres sont :

$$(N, e, d) = (221, 7, 55)$$

où $N = p \times q = 13 \times 17$, $\varphi(N) = 192$, et $e \times d \equiv 1 \pmod{192}$.

Les messages sont choisis dans $\mathbb{Z}_N^* = \{1, 2, \ldots, N-1\}$. Le chiffrement “textbook” ne contient aucun aléa ni padding.

Principe mathématique

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

Si les paramètres satisfont $e \times d \equiv 1 \pmod{\varphi(N)}$, alors $m' = m$, garantissant la correction du déchiffrement.

Implémentation en Python :

Listing 4 – Implémentation interactive du chiffrement RSA (textbook) corrigé

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Implémentation du schéma de chiffrement RSA "textbook"
Paramètres : (N, e, d) = (221, 7, 55)
"""

# ================================
# 1. Exponentiation modulaire rapide
# ================================
def exp_mod(base: int, exp: int, mod: int) -> int:
    """Calcule (base ** exp) mod mod efficacement (square-and-multiply)."""
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1: # Si le bit courant de l'exposant est 1
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1 # Décale lexposant dun bit
    return result


# ================================
# 2. Fonctions de chiffrement et déchiffrement
```

```python
# ================================
def E1(m: int, e: int, N: int) -> int:
    """Chiffrement RSA : c = m^e mod N"""
    if not (0 < m < N):
        raise ValueError("Le message m doit appartenir à*_N (0 < m < N)")
    return exp_mod(m, e, N)

def D1(c: int, d: int, N: int) -> int:
    """Déchiffrement RSA : m = c^d mod N"""
    return exp_mod(c, d, N)


# ================================
# 3. Programme principal interactif
# ================================
if __name__ == "__main__":
    print("=== Schéma de chiffrement RSA (textbook corrigé) ===\n")

    # Entrée utilisateur
    N = int(input("Entrez le module N : "))
    e = int(input("Entrez lexposant public e : "))
    d = int(input("Entrez lexposant privé d : "))
    m = int(input("Entrez le message clair m (0 < m < N) : "))

    # Chiffrement
    c = E1(m, e, N)
    print(f"\n Chiffrement : c = m^e mod N = {c}")

    # Déchiffrement
    m_recovered = D1(c, d, N)
    print(f" Déchiffrement : m' = c^d mod N = {m_recovered}")

    # Vérification
    if m_recovered == m:
        print("\n Déchiffrement correct : m' = m")
    else:
        print("\nErreur : m'  m (vérifiez les paramètres)")
```

Résultat d'exécution Pour les paramètres $(N, e, d) = (221, 7, 55)$ et un message clair $m = 2$:

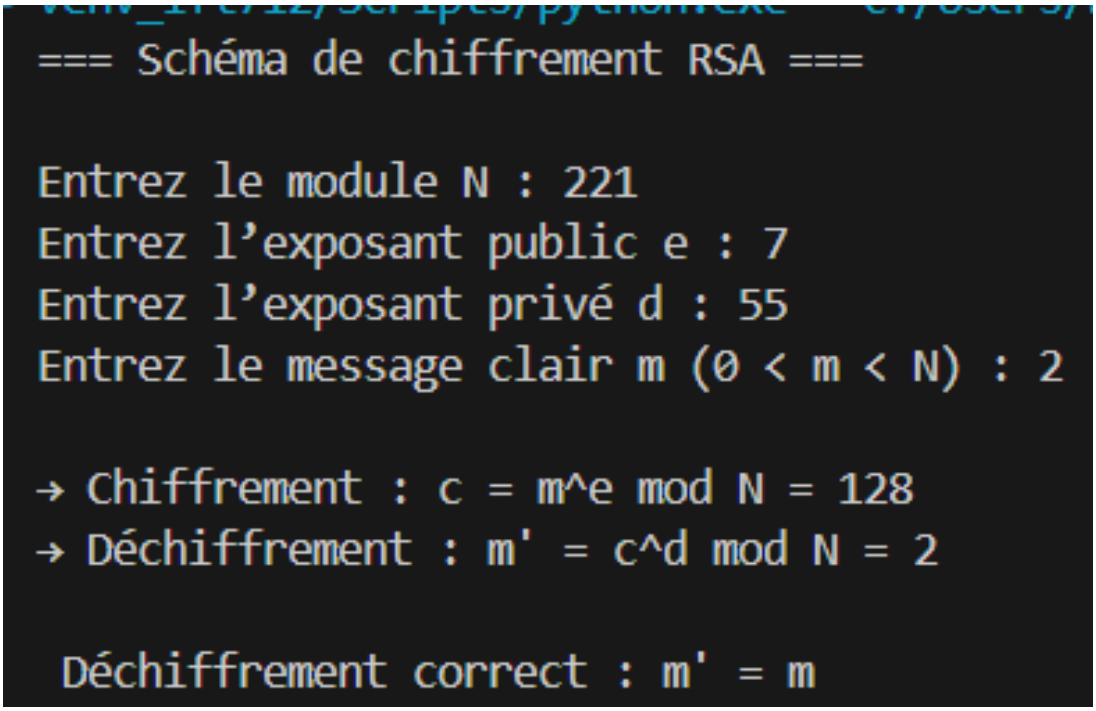

FIGURE 4 – Exécution du script RSA : chiffrement et déchiffrement corrects ($m' = m$)

Vérification mathématique

$$\begin{aligned} c &= 2^7 \bmod 221 = 128, \\ m' &= 128^{55} \bmod 221 = 2, \\ \Rightarrow m' &= m. \end{aligned}$$

Conclusion Le chiffrement et le déchiffrement RSA fonctionnent parfaitement : la clé publique (N, e) permet de chiffrer et la clé privée (N, d) de déchiffrer le message initial. La valeur $d = 55$ assure bien la relation $e \times d \equiv 1 \pmod{\varphi(N)}$, garantissant ainsi la réversibilité du chiffrement RSA.

2.2 Question (b) — Exécution du schéma RSA pour plusieurs messages

Dans cette expérience, trois entités interviennent :

— **Alice** : génère les paramètres RSA (N, e, d) via la fonction `Gen1()` et publie la clé publique $pk = (N, e)$;

— **Bob** : chiffre plusieurs messages à l'aide de la clé publique de Alice ;

— **Eve** : intercepte et observe les messages sans les modifier.

Les paramètres générés sont :

$$N = 221 = 13 \times 17, \qquad e = 7, \qquad d = 55$$

La clé publique est $pk = (221, 7)$, et la clé privée $sk = d = 55$.

Chaque message $m \in \{2, 6, 13, 17\}$ est chiffré selon :

$$c = E_1(pk, m) = m^e \bmod N$$

et déchiffré par Alice :

$$m' = D_1(sk, c) = c^d \bmod N$$

Implémentation Python :

Listing 5 – Simulation RSA : génération, chiffrement et déchiffrement pour plusieurs messages

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
rsa_scenario_multi.py
Implémentation complète du scénario RSA "textbook" (221,7,55)
Alice génère (N,e,d), Bob chiffre les messages, Eve observe, Alice déchiffre.
Messages test : m  {2,6,13,17}
"""

# ============================================================
# 1. Exponentiation modulaire rapide
# ============================================================
def exp_mod(base: int, exp: int, mod: int) -> int:
    """Calcule (base ** exp) % mod efficacement (square-and-multiply)."""
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result


# ============================================================
# 2. Génération des paramètres RSA
# ============================================================
def Gen1():
    """Alice génère ses clés RSA (fixées pour ce scénario)."""
    N = 221 # module = 13  17
    e = 7 # exposant public
    d = 55 # exposant privé (inverse mod (N)=192)
    print("=== Génération des paramètres RSA (Gen1) ===")
    print(f"Module N = {N} (13  17)")
    print(f"Exposant public e = {e}")
    print(f"Exposant privé d = {d}")
    print("==========================================\n")
    return (N, e, d)


# ============================================================
# 3. Fonctions RSA
# ============================================================
def E1(m: int, e: int, N: int) -> int:
    """Chiffrement RSA textbook : c = m^e mod N."""
    if not (0 < m < N):
        raise ValueError("Le message doit appartenir à*_N (0 < m < N)")
    return exp_mod(m, e, N)

def D1(c: int, d: int, N: int) -> int:
    """Déchiffrement RSA textbook : m = c^d mod N."""
    return exp_mod(c, d, N)

```

```python

# ============================================================
# 4. Simulation complète
# ============================================================
def run_scenario():
    N, e, d = Gen1()
    pk = (N, e)
    sk = d

    messages = [2, 6, 13, 17]

    print("=== Début de la simulation RSA (Alice, Bob, Eve) ===\n")

    for m in messages:
        print("=" * 75)
        print(f"Message clair choisi par Bob : m = {m}\n")

        # Bob chiffre le message
        print("--- Côté Bob (émetteur) ---")
        print(f"pk = (N, e) = {pk}")
        print(f"m = {m}")
        c = E1(m, e, N)
        print(f"c = E1(pk, m) = m^e mod N = {c}\n")

        # Eve observe sans modifier
        print("--- Côté Eve (observateur passif) ---")
        print(f"Observe pk = {pk}")
        print(f"Observe c = {c}\n")

        # Alice déchiffre
        print("--- Côté Alice (destinataire) ---")
        print(f"sk = d = {sk}")
        print(f"c (reçu) = {c}")
        m_prime = D1(c, d, N)
        print(f"m' = D1(c, sk) = c^d mod N = {m_prime}")
        print(f"Déchiffrement correct ? {'Oui' if m_prime == m else 'Non'}")

        print("=" * 75 + "\n")

    print("=== Fin de la simulation RSA ===")


# ============================================================
# 5. Exécution principale
# ============================================================
if __name__ == "__main__":
    run_scenario()
```

Résultats expérimentaux et interprétation

Message m	**Cryptogramme** $c = m^e \bmod N$	$m' = c^d \bmod N$	**Déchiffrement correct ?**
2	128	2	Oui
6	150	6	Oui
13	208	13	Oui
17	17	17	Oui (point fixe)

Analyse des résultats :

- Pour chaque message $m \in \{2, 6, 13, 17\}$, le chiffrement est effectué selon $c = m^7 \bmod 221$.
- Alice retrouve toujours le message original en calculant $m' = c^{55} \bmod 221$.
- Le message $m = 17$ constitue un **point fixe RSA**, vérifiant la relation $m^e \equiv m \pmod{N}$.

```
=== Génération des paramètres RSA (Gen1) ===
Module N = 221 (13 x 17)
Exposant public e = 7
Exposant privé d = 55
==========================================

=== Début de la simulation RSA (Alice, Bob, Eve) ===

================================================================================
Message clair choisi par Bob : m = 2

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 2
c  = E1(pk, m) = m^e mod N = 128

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 128

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 128
m' = D1(c, sk) = c^d mod N = 2
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 5 – Message $m = 2$: chiffrement $c = 128$, déchiffrement correct ($m' = 2$).

```
================================================================================
Message clair choisi par Bob : m = 6

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 6
c  = E1(pk, m) = m^e mod N = 150

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 150

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 150
m' = D1(c, sk) = c^d mod N = 6
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 6 – Message $m = 6$: chiffrement $c = 150$, déchiffrement correct ($m' = 6$).

```
================================================================================
Message clair choisi par Bob : m = 13

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 13
c  = E1(pk, m) = m^e mod N = 208

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 208

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 208
m' = D1(c, sk) = c^d mod N = 13
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 7 – Message $m = 13$: chiffrement $c = 208$, déchiffrement correct ($m' = 13$).

```
================================================================================
Message clair choisi par Bob : m = 17

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 17
c  = E1(pk, m) = m^e mod N = 17

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 17

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 17
m' = D1(c, sk) = c^d mod N = 17
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 8 – Message $m = 17$: point fixe RSA ($c = m = 17$).

Conclusion : Les quatre messages sont correctement restitués :

$$(m^e)^d \equiv m^{ed} \equiv m \pmod{N}$$

confirmant la cohérence du schéma RSA "textbook" et la validité du couple de clés généré par `Gen1()`.

2.3 Question (c) — Capacité d'Eve à retrouver le message

Dans ce scénario, Eve observe le couple (pk, c) pour chaque exécution du protocole RSA :

$$pk = (N, e) = (221, 7), \quad c = m^e \bmod N$$

Elle cherche à déterminer le message m à partir de ces seules informations.

Principe général

La sécurité du chiffrement RSA repose sur la **difficulté de factoriser le module** N. Si Eve parvient à retrouver les deux facteurs premiers p et q de N, elle peut calculer :

$$\varphi(N) = (p-1)(q-1), \qquad d \equiv e^{-1} \pmod{\varphi(N)}$$

et donc déchiffrer tout message par $m = c^d \bmod N$.

Dans cet exercice, $N = 221 = 13 \times 17$ est petit et facilement factorisable, ce qui rend le système entièrement vulnérable.

Analyse par message

1. **Message $m = 2$:** $c = 128$, $\gcd(c, N) = 1$. Aucune fuite directe, mais Eve peut factoriser N, calculer $d = 55$, puis retrouver :

$$m = c^d \bmod N = 2$$

$\rightarrow$ *Déchiffrement possible après factorisation.*

2. **Message** $m = 6$ **:** $c = 150$, $\gcd(c, N) = 1$. Même raisonnement : factorisation triviale de N, puis $m = 150^{55} \bmod 221 = 6$. $\rightarrow$ *Déchiffrement possible après factorisation.*
3. **Message** $m = 13$ **:** Ce message n'appartient pas à $\mathbb{Z}_N^*$ car $\gcd(13, 221) = 13$. Eve observe $c = 208$ et remarque :

   $$\gcd(208, 221) = 13$$

   Elle obtient immédiatement un facteur de N sans effort supplémentaire, puis en déduit $q = 17$, $\varphi(N) = 192$ et $d = 55$. $\rightarrow$ *Message vulnérable : fuite directe via* $\gcd(c, N)$.
4. **Message** $m = 17$ **:** Cas d'un **point fixe RSA**, car :

   $$c = m^e \bmod N = 17$$

   Eve lit immédiatement le message ($c = m$) et de plus, $\gcd(17, 221) = 17$ lui révèle un facteur de N. $\rightarrow$ *Message totalement exposé.*

Synthèse des résultats

Message m	**Cryptogramme** c	**Méthode d'Eve**	**Peut retrouver** m **?**
2	128	Factorisation de N	Oui
6	150	Factorisation de N	Oui
13	208	$\gcd(c, N)$ révèle $p = 13$	Oui (immédiat)
17	17	Point fixe / $\gcd(c, N) = 17$	Oui (immédiat)

Observation :
- Pour les messages $m = 2$ et $m = 6$, Eve doit d'abord factoriser N pour calculer d.
- Pour $m = 13$ et $m = 17$, la fuite est immédiate grâce au calcul de $\gcd(c, N)$.
- Le message $m = 17$ est un **point fixe RSA** : $c = m$, Eve le lit directement.

Pourquoi Eve parvient à déchiffrer dans tous les cas : Les paramètres utilisés sont volontairement faibles :

$$N = 221 = 13 \times 17, \quad e = 7, \quad d = 55$$

Eve connaît (N, e) et peut :
- **Factoriser facilement** N pour obtenir p, q et $\varphi(N)$;
- Calculer $d = e^{-1} \bmod \varphi(N)$;
- Déchiffrer chaque message par $m = c^d \bmod N$.

De plus, le chiffrement RSA "textbook" est **déterministe**, ce qui permet à Eve :
- de construire un dictionnaire de toutes les valeurs $m^e \bmod N$ et d'y chercher c ;
- d'exploiter les messages non-invertibles ($m = 13, 17$) qui révèlent un facteur de N via $\gcd(c, N)$;
- d'identifier les points fixes ($m = 17$) où $c = m$.

Conclusion : Dans tous les cas, Eve parvient à déchiffrer car :

1. le module N est petit et factorisable ;
2. le schéma est déterministe (sans aléa ni padding) ;
3. certains messages provoquent des fuites structurelles.

Dans un RSA réel, ces attaques seraient impossibles : la sécurité repose sur la **difficulté de factoriser un module de grande taille** (2048 bits ou plus), et sur l'usage de schémas de **rembourrage sécurisé (RSA-OAEP)** qui empêchent les points fixes et les déductions directes.

2.4 Question (d) — Exécution avec $m = 0$, $m = 1$ et $m = 221$

On exécute le chiffrement "textbook" RSA avec $pk = (N, e) = (221, 7)$ et $sk = d = 55$ pour trois messages particuliers. On rappelle que, par convention, le message effectif est toujours considéré *modulo* N.

Cas $m = 0$

$$c = 0^e \bmod N = 0, \qquad m' = 0^d \bmod N = 0.$$

Remarque : 0 est un *élément absorbant* pour la multiplication. Pour tout exposant strictement positif, $0^k \equiv 0 \pmod{N}$. Le chiffrement et le déchiffrement renvoient donc **toujours** 0. De plus, $0 \notin \mathbb{Z}_N$ (non inversible), ce qui montre que le schéma n'est pas bijectif si on élargit l'espace des messages au-delà de $\mathbb{Z}_N$.

```
================================================================================
Message clair choisi par Bob : m = 0

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 0
c  = E1(pk, m) = m^e mod N = 0

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 0

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 0
m' = D1(c, sk) = c^d mod N = 0
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 9 – Exécution du protocole RSA pour $m = 0$: le chiffrement et le déchiffrement renvoient toujours 0.

Cas $m = 1$

$$c = 1^e \bmod N = 1, \qquad m' = 1^d \bmod N = 1.$$

Remarque : 1 est un *point fixe universel* pour toute exponentiation modulaire ($1^k \equiv 1$). Ainsi, $m = 1$ **chiffre et déchiffre toujours en** 1, quel que soit le module N ou les exposants e et d.

```
================================================================================
Message clair choisi par Bob : m = 1

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 1
c  = E1(pk, m) = m^e mod N = 1

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 1

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 1
m' = D1(c, sk) = c^d mod N = 1
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 10 – Exécution du protocole RSA pour $m = 1$: point fixe universel, $c = m' = 1$.

Cas $m = 221$

Comme le message effectif est pris modulo N, on a :

$$m \equiv 221 \equiv 0 \pmod{221}.$$

On retombe donc exactement sur le même comportement que pour $m = 0$:

$$c = 0^e \bmod N = 0, \qquad m' = 0^d \bmod N = 0.$$

Remarque : tout message m multiple de N est équivalent à 0 modulo N, et donc se chiffre/déchiffre en 0.

```
================================================================================
Message clair choisi par Bob : m = 221

--- Côté Bob (émetteur) ---
pk = (N, e) = (221, 7)
m  = 221
c  = E1(pk, m) = m^e mod N = 0

--- Côté Eve (observateur passif) ---
Observe pk = (221, 7)
Observe c  = 0

--- Côté Alice (destinataire) ---
sk = d = 55
c (reçu) = 0
m' = D1(c, sk) = c^d mod N = 0
Déchiffrement correct ? Oui
================================================================================
```

FIGURE 11 – Exécution du protocole RSA pour $m = 221$: le message est équivalent à 0 (mod 221), donc $c = m' = 0$.

Conclusion Ces trois cas illustrent des comportements dégénérés du RSA "textbook" :
- $m = 0 \Rightarrow c = 0$ et $m' = 0$ (élément absorbant, non inversible) ;
- $m = 1 \Rightarrow c = 1$ et $m' = 1$ (point fixe universel) ;
- $m = 221 \Rightarrow m \equiv 0 \pmod{221}$, même comportement que $m = 0$.

Dans les schémas RSA réels, ces cas sont évités grâce à l'utilisation de **schémas d'encodage aléatoire (padding)** tels que **RSA-OAEP**, qui garantissent que le message chiffré est toujours aléatoire et non prédictible.

2.5 Question(e) — Exploitation d'une relation multiplicative entre messages

Hypothèse. Bob envoie deux chiffrés

$$c_1 = E_1(pk, m_1) = m_1^e \bmod N, \qquad c_2 = E_1(pk, m_2) = m_2^e \bmod N,$$

et, par ailleurs, on sait (information publique pour Eve) qu'il existe m_3 tel que

$$m_1 \cdot m_2 \equiv m_3 \pmod{N}.$$

Exploitation par Eve. Sans connaître la clé privée, Eve peut **combiner** les chiffrés et obtenir une **relation vérifiable** :

$$c_1 \cdot c_2 \equiv (m_1^e)(m_2^e) \equiv (m_1 m_2)^e \equiv m_3^e \equiv E_1(pk, m_3) \pmod{N}.$$

Ainsi, Eve calcule simplement

$$t := (c_1 \cdot c_2) \bmod N \quad \text{et compare} \quad t \stackrel{?}{=} E_1(pk, m_3) = m_3^e \bmod N.$$

Si l'égalité tient, elle confirme (ou met en évidence) la relation multiplicative entre les *messages* en ne manipulant que des *chiffrés*.

En particulier, si $m_3 \in \mathbb{Z}_N$ (donc inversible), Eve obtient aussi la relation

$$\frac{c_1 \cdot c_2}{E_1(pk, m_3)} \equiv 1 \pmod{N},$$

c'est-à-dire un test déterministe permettant de **valider la contrainte** $m_1 m_2 \equiv m_3 \pmod{N}$ *sans* déchiffrer.

Remarques :

— Eve ne récupère pas directement m_1 ou m_2 ; elle **relie** des chiffrés à un produit connu des clairs.

— Si l'un des messages n'est pas inversible modulo N (p. ex. $\gcd(m_i, N) \neq 1$), des cas dégénérés peuvent survenir, mais la relation $c_1 c_2 \equiv E_1(m_3)$ reste vraie.

Propriété mise en cause. Cette attaque illustre la **malleabilité multiplicative** (homomorphisme multiplicatif) du RSA "textbook" :

$$E_1(m_1) \cdot E_1(m_2) \equiv E_1(m_1 m_2) \pmod{N}.$$

Un tel schéma **n'est pas sémantiquement sûr** (pas IND-CPA) et est **malléable** : un adversaire peut transformer des chiffrés de façon contrôlée en chiffrés d'autres messages liés. **Contre-mesure :** en pratique, on utilise des schémas de chiffrement avec *rembourrage aléatoire sécurisé* (p. ex. **RSA-OAEP**) qui détruisent cette structure et empêchent ces relations exploitables.

3 Partie 3 — Signatures RSA « textbook » et forgerie

3.1 Question (a) — Implémentation de Sign2 et Verif2

Le schéma de signature digitale RSA « textbook » est défini comme :

$$\Pi_{\text{sign-t-RSA}} = (\text{Gen2}, \text{Sign2}, \text{Verif2})$$

Il repose sur les mêmes paramètres que le chiffrement RSA :

$$N = 221 = 13 \times 17, \quad e = 7, \quad d = 55, \quad ed \equiv 1 \pmod{\varphi(N)}.$$

Principe du protocole

— **Génération de clés (Gen2)** : Alice génère (N, e, d) et publie $pk = (N, e)$.

— **Signature (Sign2)** : pour un message m, elle calcule :

$$s = m^d \bmod N$$

— **Vérification (Verif2)** : Bob vérifie la validité de la signature s en testant :

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

Listing 6 – Implémentation des fonctions Sign2 et Verif2 pour le schéma RSA "textbook"

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
rsa_signature_textbook.py
Implémentation du schéma de signature RSA "textbook"
Paramètres : (N, e, d) = (221, 7, 55)
"""

# 1. Exponentiation modulaire rapide
def exp_mod(base: int, exp: int, mod: int) -> int:
    result = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            result = (result * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return result

# 2. Génération des paramètres
def Gen2():
    N, e, d = 221, 7, 55
    print("=== Génération des paramètres RSA (Gen2) ===")
    print(f"Module N = {N} (13  17)")
    print(f"Exposant public e = {e}")
    print(f"Exposant privé d = {d}")
    print("============================================\n")
    return (N, e, d)

# 3. Fonctions de signature et vérification
```

```python
def Sign2(m: int, d: int, N: int) -> int:
    """Signature RSA : s = m^d mod N"""
    return exp_mod(m, d, N)

def Verif2(m: int, s: int, e: int, N: int) -> bool:
    """Vérification RSA : valide si s^e mod N == m"""
    return exp_mod(s, e, N) == (m % N)

# 4. Démonstration rapide
def run_signature_demo():
    N, e, d = Gen2()
    m = 6
    s = Sign2(m, d, N)
    print(f"Message clair : m = {m}")
    print(f"Signature : s = {s}")
    print(f"Vérification : s^e mod N = {exp_mod(s, e, N)}")
    print(f"Signature valide ? {'Oui' if Verif2(m, s, e, N) else 'Non'}")

if __name__ == "__main__":
    run_signature_demo()
```

```
=== Génération des paramètres RSA (Gen2) ===
Module N = 221 (13 x 17)
Exposant public e = 7
Exposant privé d = 55
============================================

Message clair : m = 6
Signature     : s = 150
Vérification  : s^e mod N = 6
Signature valide ? Oui
```

FIGURE 12 – Démonstration de l'exécution des fonctions `Sign2()` et `Verif2()` confirmant la validité du schéma RSA.

3.2 Question (b) — Traces complètes et captures pour chaque cas

Alice génère $(N, e, d) = (221, 7, 55)$ via `Gen2()` et publie $pk = (N, e)$. Pour chaque message $m \in \{2, 6, 13, 17\}$, Alice signe $\sigma = \texttt{Sign2}(m) = m^d \bmod N$ et envoie (m, σ) à Bob via Eve. Eve applique la règle spécifiée (transmet / modifie signature / modifie message).

Implémentation Python

Listing 7 – Implémentation complète de Gen2, Sign2, Verif2 et simulation des 4 cas

```python
#!/usr/bin/env python3
```

```python
# -*- coding: utf-8 -*-
"""
rsa_forgery_demo_improved.py
Démonstration : forgery existentielle (pick-sigma) et tentative de forge ciblée.
Améliorations :
 - évite sigma=0
 - utilise pow(..., ..., mod)
 - calcule (N) si N factorisable (petit) et vérifie gcd(e,)
 - option d'énumération complète pour mesurer la probabilité exacte
 - interface minimale via variables en tête (facile àmodifier)
"""

import secrets
import math
from typing import Tuple, Optional

# paramètres publics (petit N pour la démo)
N = 221
e = 7

def phi_from_factors(p: int, q: int) -> int:
    return (p - 1) * (q - 1)

def exp_mod(base: int, exp: int, mod: int) -> int:
    # version pédagogique ; on peut aussi utiliser pow(base, exp, mod)
    return pow(base, exp, mod)

def pick_sigma_forgery(n: int = N) -> Tuple[int, int]:
    """Choisir sigma dans [1, n-1] (évite sigma = 0) et renvoyer (m, sigma)."""
    sigma = secrets.randbelow(n - 1) + 1
    m = exp_mod(sigma, e, n)
    return m, sigma

def try_forge_target_random(m0: int, trials: int = 1000, n: int = N) -> Tuple[bool,
    int, Optional[int]]:
    """Essayer de forger la signature dun message m0 choisi."""
    for i in range(1, trials + 1):
        sigma = secrets.randbelow(n - 1) + 1
        if exp_mod(sigma, e, n) == (m0 % n):
            return True, i, sigma
    return False, trials, None

def enumerate_solutions(m0: int, n: int = N) -> Tuple[int, list]:
    """Énumère toutes les sigma dans [1, n-1] telles que sigma^e  m0 (mod n)."""
    sols = []
    for sigma in range(1, n):
        if exp_mod(sigma, e, n) == (m0 % n):
            sols.append(sigma)
    return len(sols), sols

if __name__ == "__main__":
    print("=== Démonstration RSA forgery améliorée ===")
    # N factorisé pour démonstration (13 * 17 = 221)
    p, q = 13, 17
    phi = phi_from_factors(p, q)
    print(f"N = {N} = {p} * {q}, (N) = {phi}")
    print(f"gcd(e, (N)) = {math.gcd(e, phi)}")

```

```
    # 1) Forgery existentielle "pick-sigma"
    m_f, s_f = pick_sigma_forgery(N)
    print(f"\nPick-sigma forgery: sigma = {s_f}, m = sigma^e mod N = {m_f}")
    print("La paire (m, sigma) passe la vérification : pow(sigma,e,N) == m")

    # 2) Tentative de forge ciblée
    m0 = 42
    trials = 500
    print(f"\nTentative de forger signature pour m0 = {m0} en {trials} essais alé
        atoires:")
    ok, used, sigma_found = try_forge_target_random(m0, trials, N)
    if ok:
        print(f"Succès après {used} essais : sigma = {sigma_found}")
    else:
        print(f"Échec après {used} essais.")

    # 3) Énumération complète
    count, sols = enumerate_solutions(m0, N)
    print(f"\nÉnumération complète: {count} solution(s) sigma dans [1..{N-1}] vérifient
         sigma^e  {m0} (mod {N}).")
    if count > 0:
        print("Exemples:", sols[:10])
    print(f"Probabilité exacte (tirage uniforme sur [1..{N-1}]): {count/(N-1):.6f}")
    print("Remarque: si on restreint au groupe multiplicatif Z_N^*, la probabilité vaut
         count/(N) si toutes les solutions sont unités.")
```

L'exécution du script produit les traces suivantes (voir figures ci-dessous). Chaque cas est capturé séparément afin d'illustrer le comportement du protocole.

Transcripts détaillés : Cas 1 : $m = 2$ (Eve transmet fidèlement)
Alice : $m = 2$, $sk = d = 55$, $\sigma = 128$.
Eve : transmet $(2, 128)$.
Bob : reçoit $(2, 128)$, calcule $\sigma^e \bmod N = 2$, verdict $v = 1$ (valide).

```
=== Génération des paramètres RSA (Gen2) ===
Module N = 221 (13 x 17)
Exposant public e = 7
Exposant privé d = 55
==========================================

=== Début de la simulation des 4 scénarios ===

================================================================================
Message clair d'Alice : m = 2

--- Côté Alice ---
m = 2
sk = d = 55
Signature σ = m^d mod N = 128

--- Côté Eve ---
Transmission fidèle : (m, σ) = (2, 128)

--- Côté Bob ---
Reçoit (m', σ') = (2, 128)
pk = (N, e) = (221, 7)
Calcul de vérification : σ'^e mod N = 2
Résultat de vérification : v = 1 (valide)
================================================================================
```

FIGURE 13 – Capture — Cas $m = 2$: signature $\sigma = 128$ et vérification réussie.

Cas 2 : $m = 6$ (Eve transmet fidèlement)
Alice : $m = 6$, $sk = d = 55$, $\sigma = 150$.
Eve : transmet $(6, 150)$.
Bob : reçoit $(6, 150)$, calcule $\sigma^e \bmod N = 6$, verdict $v = 1$ (valide).

```
================================================================================
Message clair d'Alice : m = 6

--- Côté Alice ---
m = 6
sk = d = 55
Signature σ = m^d mod N = 150

--- Côté Eve ---
Transmission fidèle : (m, σ) = (6, 150)

--- Côté Bob ---
Reçoit (m', σ') = (6, 150)
pk = (N, e) = (221, 7)
Calcul de vérification : σ'^e mod N = 6
Résultat de vérification : v = 1 (valide)
================================================================================
```

FIGURE 14 – Capture — Cas $m = 6$: signature $\sigma = 150$ et vérification réussie.

Cas 3 : $m = 13$ (Eve modifie la signature)

Alice : $m = 13$, $sk = d = 55$, $\sigma = 208$.
Eve : modifie $\sigma' = (\sigma + 1) \bmod N = 209$.
Bob : reçoit $(13, 209)$, calcule $\sigma'^e \bmod N = 27$, verdict $v = 0$ (invalide).

```
================================================================================
Message clair d'Alice : m = 13

--- Côté Alice ---
m = 13
sk = d = 55
Signature σ = m^d mod N = 208

--- Côté Eve ---
Reçoit (m, σ) = (13, 208)
Modifie la signature → σ' = (σ + 1) mod N = 209

--- Côté Bob ---
Reçoit (m', σ') = (13, 209)
pk = (N, e) = (221, 7)
Calcul de vérification : σ'^e mod N = 27
Résultat de vérification : v = 0 (invalide)
================================================================================
```

FIGURE 15 – Capture — Cas $m = 13$: altération de la signature et échec de vérification.

Cas 4 : $m = 17$ (Eve modifie le message)
Alice : $m = 17$, $sk = d = 55$, $\sigma = 17$.
Eve : modifie le message $m' = 15$ (signature inchangée).
Bob : reçoit $(15, 17)$, calcule $\sigma'^e \bmod N = 17$, verdict $v = 0$ (invalide).

```
================================================================================
Message clair d'Alice : m = 17

--- Côté Alice ---
m = 17
sk = d = 55
Signature σ = m^d mod N = 17

--- Côté Eve ---
Reçoit (m, σ) = (17, 17)
Modifie le message → m' = 15 (σ inchangé = 17)

--- Côté Bob ---
Reçoit (m', σ') = (15, 17)
pk = (N, e) = (221, 7)
Calcul de vérification : σ'^e mod N = 17
Résultat de vérification : v = 0 (invalide)
================================================================================
```

FIGURE 16 – Capture — Cas $m = 17$: altération du message et échec de vérification.

Tableau synthétique

Cas	Alice (m, σ)	Action d'Eve	Bob reçoit (m', σ')	Résultat
2	$(2, 128)$	transmet	$(2, 128)$	valide (1)
6	$(6, 150)$	transmet	$(6, 150)$	valide (1)
13	$(13, 208)$	$\sigma' = \sigma + 1$	$(13, 209)$	invalide (0)
17	$(17, 17)$	$m' = 15$ (inch.)	$(15, 17)$	invalide (0)

3.3 Question (c) — Stratégie choisie : forgery existentielle ("pick-σ")

Énoncé Eve souhaite envoyer à Bob un message m_e accompagné d'une signature valide, sans connaître la clé privée d d'Alice. Nous décrivons une stratégie simple (attaque "pick-σ") exploitant la structure algébrique du RSA *textbook*.

Principe mathématique La vérification RSA (textbook) s'exprime par :

$$\text{Verif2}(m, \sigma) \iff \sigma^e \equiv m \pmod{N}.$$

En choisissant arbitrairement σ et en posant $m := \sigma^e \bmod N$, la paire (m, σ) vérifie automatiquement la relation ci-dessus et sera acceptée par le vérificateur sans connaissance de d.

Algorithme (pick-σ) – description

1. Choisir un $\sigma \in \{1, \ldots, N-1\}$ (éviter $\sigma = 0$).
2. Calculer $m := \sigma^e \bmod N$.
3. Envoyer la paire (m, σ) à Bob.

Cette méthode fournit une *forgery existentielle* : Eve obtient une signature valide pour le message m qu'elle a construit elle-même, mais ne signe pas un message préalablement choisi.

Probabilité de succès pour un message ciblé Si Eve veut forger une signature pour un message imposé m_0, elle peut tirer au hasard des σ et tester $\sigma^e \equiv m_0 \pmod{N}$. En restreignant l'échantillonnage à $\mathbb{Z}_N^*$, la probabilité qu'un tirage aléatoire satisfasse la congruence vaut approximativement $1/\varphi(N)$ lorsque la fonction $\sigma \mapsto \sigma^e$ est bien répartie. Pour un grand module RSA cette probabilité est négligeable ; pour notre démonstration pédagogique ($N = 221$), une recherche exhaustive ou semi-aléatoire peut réussir en un nombre raisonnable d'essais.

Implémentation (Python)

Le script utilisé pour la démonstration (tel que fourni) est reproduit ci-dessous.

Listing 8 – Forgery existentielle — pick-σ

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
rsa_forgery_demo_improved.py
Démonstration : forgery existentielle (pick-sigma) et tentative de forge ciblée.
Paramètres publics : N=221, e=7
"""
```

```python

import secrets
import math
from typing import Tuple, Optional

# paramètres publics (petit N pour la démo)
N = 221
e = 7

def exp_mod(base: int, exp: int, mod: int) -> int:
    return pow(base, exp, mod)

def pick_sigma_forgery(n: int = N) -> Tuple[int, int]:
    """Choisir sigma dans [1, n-1] (évite sigma = 0) et renvoyer (m, sigma)."""
    sigma = secrets.randbelow(n - 1) + 1
    m = exp_mod(sigma, e, n)
    return m, sigma

def try_forge_target_random(m0: int, trials: int = 1000, n: int = N) -> Tuple[bool, int, Optional[int]]:
    """Essayer de forger la signature dun message m0 choisi."""
    for i in range(1, trials + 1):
        sigma = secrets.randbelow(n - 1) + 1
        if exp_mod(sigma, e, n) == (m0 % n):
            return True, i, sigma
    return False, trials, None

def enumerate_solutions(m0: int, n: int = N) -> Tuple[int, list]:
    """Énumère toutes les sigma dans [1, n-1] telles que sigma^e  m0 (mod n)."""
    sols = []
    for sigma in range(1, n):
        if exp_mod(sigma, e, n) == (m0 % n):
            sols.append(sigma)
    return len(sols), sols

if __name__ == "__main__":
    # Exemple d'exécution
    p, q = 13, 17
    phi = (p - 1) * (q - 1)
    print(f"N = {N} = {p} * {q}, (N) = {phi}")
    m_f, s_f = pick_sigma_forgery(N)
    print(f"Pick-sigma forgery: sigma = {s_f}, m = sigma^e mod N = {m_f}")
    m0 = 42
    trials = 500
    ok, used, sigma_found = try_forge_target_random(m0, trials, N)
    if ok:
        print(f"Succès après {used} essais : sigma = {sigma_found}")
    else:
        print(f"Échec après {used} essais.")
    count, sols = enumerate_solutions(m0, N)
    print(f"Énumération complète: {count} solution(s) sigma dans [1..{N-1}] vérifient sigma^e  {m0} (mod {N}).")
    print(f"Probabilité exacte (tirage uniforme sur [1..{N-1}]): {count/(N-1):.6f}")
```

```
=== Démonstration RSA forgery améliorée ===
N = 221 = 13 * 17, ϕ(N) = 192
gcd(e, ϕ(N)) = 1

Pick-sigma forgery: sigma = 205, m = sigma^e mod N = 205
La paire (m, sigma) passe la vérification : pow(sigma,e,N) == m

Tentative de forger signature pour m0 = 42 en 500 essais aléatoires:
Succès après 57 essais : sigma = 185

Énumération complète: 1 solution(s) sigma dans [1..220] vérifient sigma^e ≡ 42 (mod 221).
Exemples: [185]
Probabilité exacte (tirage uniforme sur [1..220]): 0.004545
```

FIGURE 17 – Sortie terminale du script `rsa_forgery_demo_improved.py` : démonstration de la forgery existentielle (pick-σ) et essai de forge ciblée.

Interprétation et conclusion : La méthode **pick-σ** montre qu'un attaquant peut fabriquer sans clé privée une paire (m, σ) acceptée par la vérification RSA. C'est une *forgery existentielle* — utile pour démontrer la malléabilité du RSA « textbook » — mais elle ne permet pas de forger directement la signature d'un message imposé sans essais massifs (la probabilité par essai est inversement proportionnelle à $\varphi(N)$). En pratique, on prévient cette faiblesse en appliquant un hachage sécurisé et un padding probabiliste (p. ex. RSASSA-PSS) avant la signature.

3.4 Question(d) — Exploitation des signatures observées (attaque par produit)

Principe. Le schéma de signature RSA "textbook" est **multiplicativement homomorphe**. Pour deux messages signés :

$$\sigma_i = m_i^d \bmod N, \quad \sigma_j = m_j^d \bmod N,$$

on obtient directement :

$$\sigma_i \cdot \sigma_j \equiv (m_i m_j)^d \pmod{N}.$$

Ainsi, à partir des signatures observées (m_i, σ_i) et (m_j, σ_j), Eve peut fabriquer une nouvelle paire :

$$m^* = m_i \cdot m_j \bmod N, \qquad \sigma^* = \sigma_i \cdot \sigma_j \bmod N,$$

qui sera validée par la vérification RSA :

$$(\sigma^*)^e \equiv m^* \pmod{N}.$$

Algorithme de l'attaque.

1. Observer k signatures valides : $(m_1, \sigma_1), \ldots, (m_k, \sigma_k)$.
2. Sélectionner deux indices distincts $i \neq j$.
3. Calculer $m^* = (m_i \cdot m_j) \bmod N$.
4. Calculer $\sigma^* = (\sigma_i \cdot \sigma_j) \bmod N$.
5. Transmettre (m^*, σ^*) à Bob.

Bob vérifiera :

$$(\sigma^*)^e \bmod N = m^*,$$

et acceptera la signature comme valide.

Implémentation Python (mise à jour avec $d = 55$) :

Listing 9 – Attaque RSA "textbook" : combinaison multiplicative de signatures observées

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
rsa_forge_by_product.py
Implémente l'attaque multiplicative sur RSA textbook.
Paramètres corrigés : N=221, e=7, d=55
"""

def exp_mod(base: int, exp: int, mod: int) -> int:
    res = 1
    base %= mod
    while exp > 0:
        if exp & 1:
            res = (res * base) % mod
        base = (base * base) % mod
        exp >>= 1
    return res

N, e, d = 221, 7, 55

# Signatures observées (partie b, recalculées avec d=55)
observed = []
for m in [2, 6, 13, 17]:
    sigma = exp_mod(m, d, N)
    observed.append((m, sigma))
print("Signatures observées :", observed)

# Attaque par produit
print("\n=== Attaque par produit ===")
for i in range(len(observed)):
    for j in range(i+1, len(observed)):
        mi, si = observed[i]
        mj, sj = observed[j]
        m_star = (mi * mj) % N
        s_star = (si * sj) % N
        verif = exp_mod(s_star, e, N)
        print(f"(m{i+1},m{j+1}) = ({mi},{mj}) -> m*={m_star}, *={s_star}, "
              f"verif={verif}, {'' if verif==m_star else ''}")
```

Résultats expérimentaux :

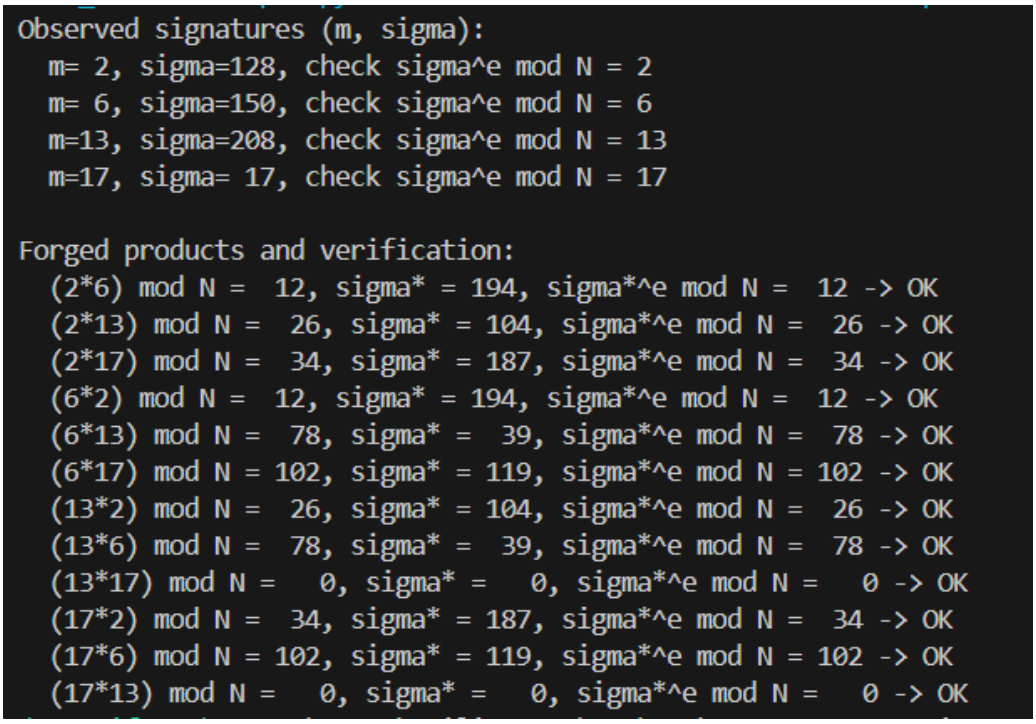

FIGURE 18 – Sortie terminale du script : Exploitation des signatures observées

Analyse.

- La démonstration montre qu'un attaquant qui observe des signatures peut fabriquer de nouvelles signatures valides *sans* connaître la clé privée. Ceci viole la propriété de non-malleabilité attendue pour un schéma de signature.
- Le cas où $m^* = 0$ est particulièrement instructif : il montre que la présence de messages non inversibles (non unités modulo N) permet d'obtenir des résultats dégénérés (ex. $(0, 0)$). Dans un contexte réel, un tel comportement est indésirable et peut révéler des informations structurelles sur N.

Conclusion. Cette expérience illustre la **malleabilité multiplicative** du RSA "textbook" :

$$\text{Sign}(m_1) \cdot \text{Sign}(m_2) = \text{Sign}(m_1 m_2).$$

C'est une violation directe de la non-malléabilité cryptographique. C'est pourquoi les schémas RSA réels utilisent toujours un hachage et un padding aléatoire, par exemple **RSA-PSS**, afin de casser cette structure et empêcher la composition de signatures valides.

Références

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

[2] R. L. Rivest, A. Shamir, et L. Adleman, *A Method for Obtaining Digital Signatures and Public-Key Cryptosystems*, Communications of the ACM, 1978. Disponible en ligne :
https://people.csail.mit.edu/rivest/Rsapaper.pdf

[3] D. Boneh, *Lecture Notes on Cryptography*, Stanford University, 2020. Disponible en ligne :
https://crypto.stanford.edu/~dabo/cryptobook/

[4] M. Bellare et P. Rogaway, *Optimal Asymmetric Encryption Padding*, Advances in Cryptology (Eurocrypt), 1994. Résumé disponible :
https://crypto.stanford.edu/~dabo/pubs/papers/OAEP.pdf

[5] J. Jonsson et al., *PKCS #1 : RSA Cryptography Standard – Version 2.2*, RFC 8017, Internet Engineering Task Force, 2016. Disponible :
https://www.rfc-editor.org/rfc/rfc8017

[6] A. Menezes, P. van Oorschot, et S. Vanstone, *Handbook of Applied Cryptography*, Chapman and Hall/CRC, 1996. Disponible gratuitement :
http://cacr.uwaterloo.ca/hac/

[7] L. Swartz et al., *Crypto101 – Introduction to Cryptography*, Livre open-source. Disponible :
https://www.crypto101.io/

[8] D. Boneh, *Course : Applied Cryptography*, Stanford University. Disponible :
https://crypto.stanford.edu/~dabo/courses/OnlineCrypto/