🔑 CM3

Cryptographie asymétrique · RSA, hachage, signatures, DH

Module Cryptographie · Première année ingénieur

Kahoot

https://rahoot.isen-cyber.ovh

🔙 Rappel CM2

Le cliffhanger

🤔 Le paradoxe du CM2

Le chiffrement symétrique (AES) est **rapide** et **sûr**. ✅

Mais Alice et Bob doivent partager une **clé secrète**. 🔑

- 📡 Envoyer la clé sur le réseau ? → Eve l'intercepte 😈
- 👫 Se rencontrer physiquement ? → Pas scalable 🌍
- 💀 Un seul échange compromis → tout est compromis

Comment partager un secret… sans canal sécurisé ?

Pendant **3000 ans**, la cryptographie n'a pas su répondre à cette question.

"We stand today on the brink of a revolution in cryptography."

- Whitfield Diffie & Martin Hellman, 1976

💡 1976

L’année qui change tout

Whitfield Diffie
(1944–)

Martin Hellman
(1945–)

👥 Diffie & Hellman

- 🎓 Deux chercheurs de **Stanford**
- 📄 **Novembre 1976** : publient *New Directions in Cryptography*
- 💥 L'article **invente** la cryptographie à clé publique
- 🏆 **Turing Award 2015** (équivalent du Nobel en informatique) pour ces travaux

> *"The development of computer-controlled communication networks promises effortless and inexpensive contact between people or computers..."*

L'article ouvre par cette intuition : **les réseaux numériques vont créer un besoin massif de cryptographie**. Il faut un nouveau paradigme.

💡 L’idée révolutionnaire : deux clés

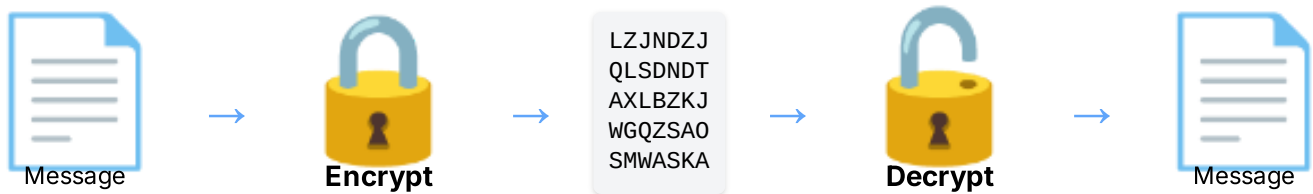
Diffie et Hellman proposent un **changement radical** :

- ❌ Au lieu d’**une seule clé** secrète partagée…
- ✅ Chaque personne possède **deux clés** mathématiquement liées :
  - 🔓 Une **clé publique** : connue de tout le monde
  - 🔐 Une **clé privée** : connue uniquement du propriétaire

Propriété magique : ce qui est chiffré avec l’une **ne peut être déchiffré que par l’autre**.

🔒 Analogie : un **cadenas ouvert** que je distribue à tout le monde. Seule **ma clé** peut le rouvrir.

🔒 Rappel : chiffrement symétrique (CM2)

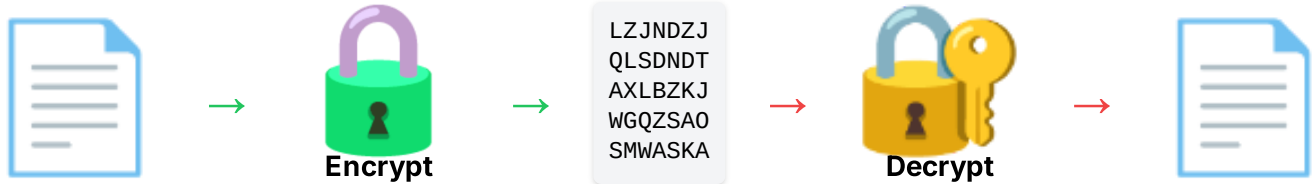


🔑

Une seule clé secrète partagée — connue d'Alice ET de Bob

⚠️ Comment Alice transmet-elle cette clé à Bob sans qu'Eve l'intercepte ?

🔓 Chiffrement asymétrique : la théorie



Clé publique
distribuée à tous

Clé privée
gardée secrète

➡️ **Plus besoin de partager un secret au préalable. Eve peut tout voir mais ne peut rien déchiffrer.**

🤐 Anecdote : le secret du GCHQ

En réalité… la crypto à clé publique a été inventée **avant 1976**. 🤔

- 🇬🇧 **1969** : James Ellis, mathématicien au **GCHQ** (services secrets britanniques), conçoit le concept
- 🧮 **1973** : Clifford Cocks invente un algorithme RSA-like
- 🤝 **1974** : Malcolm Williamson invente un protocole DH-like

🤐 Tout est **classifié top secret**.

📜 Ce n'est qu'en **1997** que le GCHQ déclassifie ces travaux — Ellis vient de mourir, à un mois près.

Morale : la science ouverte (Diffie, Hellman, RSA) a fait gagner **20 ans** à l'humanité. La cryptographie publique a transformé l'économie mondiale ; la cryptographie classifiée est restée enfermée dans un coffre.

🎯 Les deux usages de la crypto asymétrique

Avec une paire de clés (publique, privée), on peut faire **deux choses fondamentales** :

- 🔒 **Chiffrer** : tout le monde peut chiffrer un message pour Alice avec sa **clé publique**, mais seule Alice peut le déchiffrer avec sa **clé privée** → **confidentialité**
- ✍️ **Signer** : Alice chiffre une empreinte avec sa **clé privée**, n'importe qui peut vérifier avec sa **clé publique** → **authenticité + intégrité**

C'est la base de **HTTPS**, **PGP**, **SSH**, **Bitcoin**, **passeports biométriques**, **mises à jour logicielles**...

➡️ Tout repose sur ces deux opérations.

🔢 RSA

Rivest – Shamir – Adleman (1977)

Rivest

Shamir

Adleman

MIT, 1977

👥 Les inventeurs de RSA

- 🎓 Trois chercheurs du **MIT**
- 📅 **Avril 1977** : publication de l'algorithme RSA
- 🧮 Première **réalisation concrète** de l'idée de Diffie-Hellman
- 🏆 **Turing Award 2002**

📖 Anecdote : Rivest a eu l'idée pendant la nuit de **Pâques 1977**, après avoir lu l'article de Diffie-Hellman. Il rentre chez lui à 3h du matin, écrit le brouillon. Le matin, il teste ses idées avec Shamir et Adleman.

🇮🇱 Adi Shamir est aussi connu pour le **Shamir Secret Sharing** (partage de secret) et le système d'identification **Fiat-Shamir**.

📰 RSA-129 : le défi de Scientific American

Août 1977 : Martin Gardner publie dans *Scientific American* un défi de Rivest et al. :

- 🔢 Un nombre de **129 chiffres** (~426 bits) à factoriser
- 💰 Récompense : **100 dollars**
- 📝 Rivest estime qu'il faudra **40 quadrillions d'années** pour le casser

⏱️ **17 ans plus tard** (1994), une équipe internationale de 600 volontaires factorise RSA-129 en utilisant le réseau Internet naissant et 8 mois de calcul.

📝 Le message déchiffré : **"THE MAGIC WORDS ARE SQUEAMISH OSSIFRAGE"**

➡️ Leçon : **les tailles de clés vieillissent**. RSA-129 (426 bits) est trivial aujourd'hui ; on utilise **2048 ou 4096 bits**.

RSA

Caractéristique & principe

🧮 RSA : caractéristique

- 🔓 **Fonction à sens unique** : `y = f(x)` est facile, `x = f⁻¹(y)` ne l'est pas
- 🔢 Basée sur le **problème de la factorisation** des grands nombres premiers

📐 La fonction RSA :

$$f(x) = x^e \bmod n \qquad \text{avec } n = p \times q, \quad e < n$$

🔑 On calcule **d** tel que :

$$e \cdot d \equiv 1 \pmod{(p-1)(q-1)} \implies f(x)^d \equiv x \pmod{n}$$

➡️ **e** = clé publique 🟢 · **d** = clé privée 🔴

🧮 **Comment trouver d ?** C'est l'**inverse modulaire** de e mod φ — on cherche un entier qui multiplié par e donne 1 modulo φ.

Ex : e=3, φ=20 → on essaie d=1, 2, 3, …, **d=7** car 3·7 = 21 = 20 + 1 ≡ 1 [20] ✅

🔢 Factorisation : facile ou difficile ?

Petits nombres ✅ trivial : 187 = 11 × 17, 1001 = 7 × 11 × 13

Grands nombres ❌ infaisable : un nombre de **2048 bits** ≈ **617 chiffres décimaux**, pas d'algorithme efficace connu.

Records publics de factorisation :

Année	Bits factorisés
1994	426 (RSA-129)
2020	829
Aujourd'hui	toujours **< 1024 bits**

➡️ RSA-2048 reste hors de portée. C'est ce qui fait la sécurité de RSA. 💪

🔧 Principe : Bob génère ses clés

1️⃣ **n = p · q** (deux grands premiers)

2️⃣ **φ = (p – 1)(q – 1)**

3️⃣ Choisir **e** tel que **pgcd(e, φ) = 1**

4️⃣ Calculer **d** tel que **d · e ≡ 1 [φ]**

🔓 Clé publique : **(n, e)** → partagée avec tout le monde

🔐 Clé privée : **(n, d)** → gardée secrète 🤫

📝 La fonction `inverse_modulaire(e, phi)` qui calcule **d** vous est **fournie** dans le scaffolding du TD3 (Euclide étendu).

🔒 Principe : chiffrer & déchiffrer

👩 Alice chiffre

Avec la **clé publique** de Bob 🟢

$$C = M^e \bmod n$$

Alice envoie **C** sur le réseau.

👨 Bob déchiffre

Avec sa **clé privée** 🔴

$$M = C^d \bmod n$$

Bob retrouve le message original.

🧮 Exemple complet : p = 3, q = 11

- p = 3, q = 11
- **n = 3 × 11 = 33**
- **φ = 2 × 10 = 20**
- **e = 3** (pgcd(3, 20) = 1 ✅)
- **d = 7** (car 3 × 7 = 21 = 1 × 20 + 1 ≡ 1 [20])

🔓 Clé publique : **(n, e) = (33, 3)**

🔐 Clé privée : **(n, d) = (33, 7)**

⚠️ Avec p=11, q=13, casser RSA est **trivial** : on factorise 143 en quelques secondes. Pour la sécurité réelle, il faut **2048 bits** au minimum.

🔒 RSA en action

Chiffrer · Déchiffrer

🔒 Alice chiffre M = 2

Clé publique de Bob : **(n, e) = (33, 3)**

Chiffrement : C = M^e mod n = **2^3 mod 33**

- $2^3 = 8$
- 8 mod 33 = **8**

📦 **C = 8**

Alice envoie C = 8 à Bob sur n'importe quel canal — Eve voit 8 mais ne peut rien en faire.

🔓 Bob déchiffre C = 8

Clé privée de Bob : **(n, d) = (33, 7)**

Déchiffrement : M = C^d mod n = **8^7 mod 33**

8^7 = 2 097 152 — gros nombre, mais Python sait le faire **directement modulo 33** :

```
n, d, C = 33, 7, 8
M = pow(C, d, n)
print(f"M = {M}")
```

✅ M = **2** — Bob retrouve le message original ! 🎉

📝 En pratique : `pow(C, d, n)` (en Python) ou `BN_mod_exp` (en OpenSSL) — c'est l'**exponentiation modulaire rapide**, jamais à faire à la main.

📊 RSA : flux de chiffrement

😈 Eve intercepte **C** mais ne peut pas calculer M sans **d**.

Pour trouver d, elle devrait factoriser n → **infaisable** pour 2048 bits. 🌌

🛡️ Sécurité de RSA

Casser RSA, c'est :

- 🧩 Soit retrouver **d** depuis (n, e) → équivalent à factoriser **n**
- 🧩 Soit factoriser **n** → on retrouve p, q, φ(n), donc d

➡️ La sécurité de RSA repose **entièrement** sur la difficulté de **factoriser**.

Algorithme	Complexité de factorisation
Force brute	2^(n/2)
GNFS (meilleur classique)	sous-exponentielle
Algorithme de Shor (quantique)	**polynomiale** ⚛️

⚛️ **Menace quantique** : un ordinateur quantique de taille suffisante casserait RSA en quelques heures. C'est pourquoi le NIST standardise des **algorithmes post-quantiques** depuis 2024 (CRYSTALS-Kyber, Dilithium...).

🐌 RSA est lent

L'exponentiation modulaire sur 2048 bits est **coûteuse**.

- ⚡ AES : ~**1 GB/s** sur un CPU moderne
- 🐌 RSA : ~**100 KB/s — 10 000× plus lent**

➡️ En pratique, on **ne chiffre jamais** un gros message avec RSA.

Schéma hybride (utilisé partout, dont TLS) :

1. 🎲 Alice tire une **clé AES aléatoire** K
2. 🔒 Alice chiffre le **message** avec AES-K (rapide)
3. 🔑 Alice chiffre **K** avec la clé publique RSA de Bob (lent mais court)
4. 📦 Alice envoie : `RSA(K) + AES_K(message)`

C'est exactement ce que fait HTTPS — on verra ça au CM4. 🌐

🤝 Chiffrement hybride : le combo gagnant

👩 Alice

① AES chiffre le message avec une clé secrète aléatoire 🔑

② RSA chiffre la clé secrète avec la **clé publique de Bob** 🟢

👨 Bob

③ Bob déchiffre la clé avec sa **clé privée** 🔴

④ AES déchiffre le message avec la clé récupérée

💡 **Vitesse de l'AES sur les données + simplicité du RSA pour échanger la clé. C'est ce que fait HTTPS.**

🔧 RSA en pratique : OpenSSL

```
# Générer une clé privée RSA 2048 bits
openssl genpkey -algorithm RSA -out priv.pem -pkeyopt rsa_keygen_bits:2048

# Extraire la clé publique
openssl rsa -in priv.pem -pubout -out pub.pem

# Chiffrer un fichier avec la clé publique
openssl pkeyutl -encrypt -pubin -inkey pub.pem -in message.txt -out chiffre.bin

# Déchiffrer avec la clé privée
openssl pkeyutl -decrypt -inkey priv.pem -in chiffre.bin -out clair.txt
```

📝 Vous manipulerez ces commandes en **TD3**.

⚠️ **Padding obligatoire** : OpenSSL ajoute par défaut un **padding OAEP** avant chiffrement. Sans padding, RSA est déterministe et vulnérable (deux mêmes messages → même chiffré).

#️⃣ Fonctions de hachage

L’empreinte digitale numérique

🧬 Qu'est-ce qu'une fonction de hachage ?

Entrée : données de taille **arbitraire** (fichier, message, image...)

Sortie : empreinte de taille **fixe** — le **digest** ou **hash**

```
"Hello"        → 185f8db32271fe25f561a6fc938b2e264306ec304eda518007d1764826381969
"Hello World"  → a591a6d40bf420404a011733cfb7b190d62c65bf0bcda32b57b277d9ad9f146e
"Hello world"  → 64ec88ca00b268e5ba1a35678a1b5316d212f4f366b2477232534a8aeca37f3c
```

Le hash est l'**empreinte digitale** 🧬 d'un document.

➡️ Si le document change d'**un seul bit**, le hash est **complètement différent**.

📏 Les algorithmes de hachage usuels

Algorithme	Année	Digest	Statut
MD5	1992	128 bits (32 hex)	💀 Cassé
SHA-1	1995	160 bits (40 hex)	⚠️ Deprecated
SHA-256	2001	256 bits (64 hex)	✅ Standard
SHA-512	2001	512 bits (128 hex)	✅ Standard
SHA-3	2015	224–512 bits	✅ Alternative
BLAKE3	2020	variable	✅ Très rapide

🎯 À quoi servent les fonctions de hachage ?

- ✅ **Vérifier l'intégrité** d'un fichier téléchargé (`sha256sum ubuntu.iso`)
- 🔑 **Stocker les mots de passe** (avec `bcrypt`, `argon2id`)
- ✍️ **Signer numériquement** : on signe le hash, pas le message (à venir)
- 🐙 **Identifiants Git** : chaque commit est un SHA-1 de son contenu
- ₿ **Bitcoin** : la preuve de travail = SHA-256(SHA-256(bloc))
- 🔗 **Chaînage de blocs** : chaque bloc référence le hash du précédent

➡️ Les fonctions de hachage sont **partout** dans la sécurité moderne.

📋 5 propriétés fondamentales

1. 🔁 **Déterministe** : même entrée → même hash, **toujours**
2. 🚫 **Résistance à la préimage** : impossible de retrouver M depuis H(M)
3. 🚫 **Résistance à la 2ème préimage** : impossible de trouver M′ ≠ M tel que H(M) = H(M′)
4. 💥 **Résistance aux collisions** : impossible de trouver deux messages M, M′ tels que H(M) = H(M′)
5. 🌊 **Effet avalanche** : un seul bit changé → hash **complètement** différent

🔍 Différence (2) et (3) ? Dans (2) M est **fixé** par l'attaquant ; dans (3) il choisit **les deux** messages → plus facile.

🎂 Le paradoxe des anniversaires

Combien de personnes faut-il dans une salle pour avoir 50 % de chances que **deux d'entre elles partagent un anniversaire** ?

➡️ **Seulement 23 personnes.** 🤯

Application aux hash : trouver une **collision** est **beaucoup plus facile** que trouver une préimage.

Hash de **n bits**	Préimage	Collision
Effort attendu	2^n	2^(n/2)
MD5 (128 bits)	2^{128}	**2^{64}**
SHA-1 (160 bits)	2^{160}	**2^{80}**
SHA-256 (256 bits)	2^{256}	**2^{128}**

➡️ C'est pourquoi on **double** la taille du digest par rapport à la sécurité visée. SHA-256 vise 128 bits de sécurité contre les collisions.

🌊 Démo : effet avalanche

```python
import hashlib

msg1 = b"Hello"
msg2 = b"hello"

h1 = hashlib.sha256(msg1).hexdigest()
h2 = hashlib.sha256(msg2).hexdigest()

print(f"SHA-256('Hello') = {h1}")
print(f"SHA-256('hello') = {h2}")
print(f"\nIdentiques ? {h1 == h2}")

# Compter les bits différents
b1 = int(h1, 16)
b2 = int(h2, 16)
diff_bits = bin(b1 ^ b2).count("1")
print(f"Bits différents : {diff_bits}/256 (~{100*diff_bits/256:.0f}%)")
```

➡️ Un seul bit changé en entrée → **~50 %** des bits du hash changent. C'est exactement ce qu'on attend d'une fonction cryptographique.

💀 MD5 (1992) : cassé

Conçu par **Ron Rivest** (le R de RSA) en 1992. Très utilisé dans les années 90-2000.

- 🔢 Digest : **128 bits**
- ⚡ Très rapide
- 💥 **2004** : Wang & Yu (Chine) trouvent des collisions en quelques secondes sur PC
- 💥 **2008** : Sotirov, Stevens et al. exploitent MD5 pour créer un **faux certificat CA** valide → ils auraient pu usurper n'importe quel site HTTPS

❌ **MD5 ne doit JAMAIS être utilisé pour la sécurité.**

⚠️ Cas tolérés : checksum d'intégrité **non-adversarial** (téléchargement, déduplication).

⚠️ SHA-1 (1995) : SHAttered

- 🔢 Digest : **160 bits** · 📐 Conçu par la **NSA** · 📜 Utilisé 20 ans dans TLS, certificats, **Git**…

💥 **Février 2017 — CWI + Google** publient **SHAttered** : la première collision SHA-1 réelle.

```
$ openssl sha1 shattered-1.pdf
```

SHAttered
First SHA-1 collision
CWI · Google · 2017

```
38762cf7f55934b34d
179ae6a4c80cadccbb7f0a
```

≡

```
$ openssl sha1 shattered-2.pdf
```

SHAttered
First SHA-1 collision
CWI · Google · 2017

```
38762cf7f55934b34d
179ae6a4c80cadccbb7f0a
```

➡️ Deux PDFs visuellement différents, **exact même SHA-1**.

💰 Coût : **6500 ans-CPU + 110 ans-GPU** ≈ 110 000 $ · 💥 **2020** : chosen-prefix à 45 000 $ (Leurent & Peyrin) → faux certificats possibles.

✅ SHA-256 (2001) : le standard actuel

- 🔢 Digest : **256 bits** (64 caractères hex)
- 🛡️ Famille **SHA-2**, conçue par la NSA en 2001
- 🚫 **Aucune collision connue** à ce jour

Utilisé partout :

- ₿ **Bitcoin** : preuve de travail (SHA-256 doublé)
- 🔒 **TLS / HTTPS** : signature des certificats
- 📦 **Signatures de logiciels** : APK Android, paquets `.deb`, `.rpm`
- 🔐 **PGP / GPG** : empreintes des clés
- 🐙 **Git** : transition en cours depuis SHA-1

✍️ Signature numérique

🤔 Le problème de l'authenticité

👩 Alice envoie un message à Bob. 👨

Comment Bob peut-il prouver que c'est bien Alice qui l'a envoyé ?

😈 Eve pourrait avoir :

- ✏️ **Modifié** le message en transit (intégrité)
- 🎭 **Fabriqué** un faux message au nom d'Alice (authenticité)
- 🔁 **Rejoué** un ancien message d'Alice

Le chiffrement seul **ne suffit pas** : un message chiffré peut être **modifié** sans qu'on s'en rende compte (les bits du chiffré peuvent flipper).

➡️ Il faut un mécanisme dédié : la **signature numérique**.

💡 RSA inversé : la signature

Idée : utiliser RSA dans l'**autre sens**.

- 🔒 **Chiffrement classique** : Bob chiffre avec la clé **publique** d'Alice. Seule Alice peut déchiffrer.
- ✍️ **Signature** : Alice **chiffre avec sa clé privée**. **N'importe qui** peut "déchiffrer" avec sa clé publique.

Pourquoi c'est pertinent ?

➡️ Si seule Alice possède sa clé privée, alors **seule Alice** a pu produire ce qui se déchiffre avec sa clé publique.

C'est la **propriété miroir** de RSA : ce qui marche dans un sens marche dans l'autre.

🐌 Pourquoi signer le hash et pas le message ?

RSA est **lent** : signer un fichier de 100 Mo prendrait des heures.

💡 **Solution** : on ne signe pas le message entier, on signe son **hash** (256 bits).

- 📏 SHA-256 produit **toujours 256 bits**, peu importe la taille du fichier
- ⚡ Signer 256 bits avec RSA est **instantané**
- 🛡️ Les propriétés du hash (résistance aux collisions) garantissent qu'on ne peut pas substituer un autre message avec le même hash

⚠️ **C'est pourquoi MD5 et SHA-1 sont dangereux pour les signatures** : si on peut trouver une collision, on peut signer un message légitime et faire valoir la signature pour un autre message malveillant.

✍️ Processus de signature

👩 Alice signe

1. Calcule **H(M)** = SHA-256 #️⃣
2. Chiffre H(M) avec sa **clé privée** 🔐 → **signature S**
3. Envoie **(M, S)** 📨

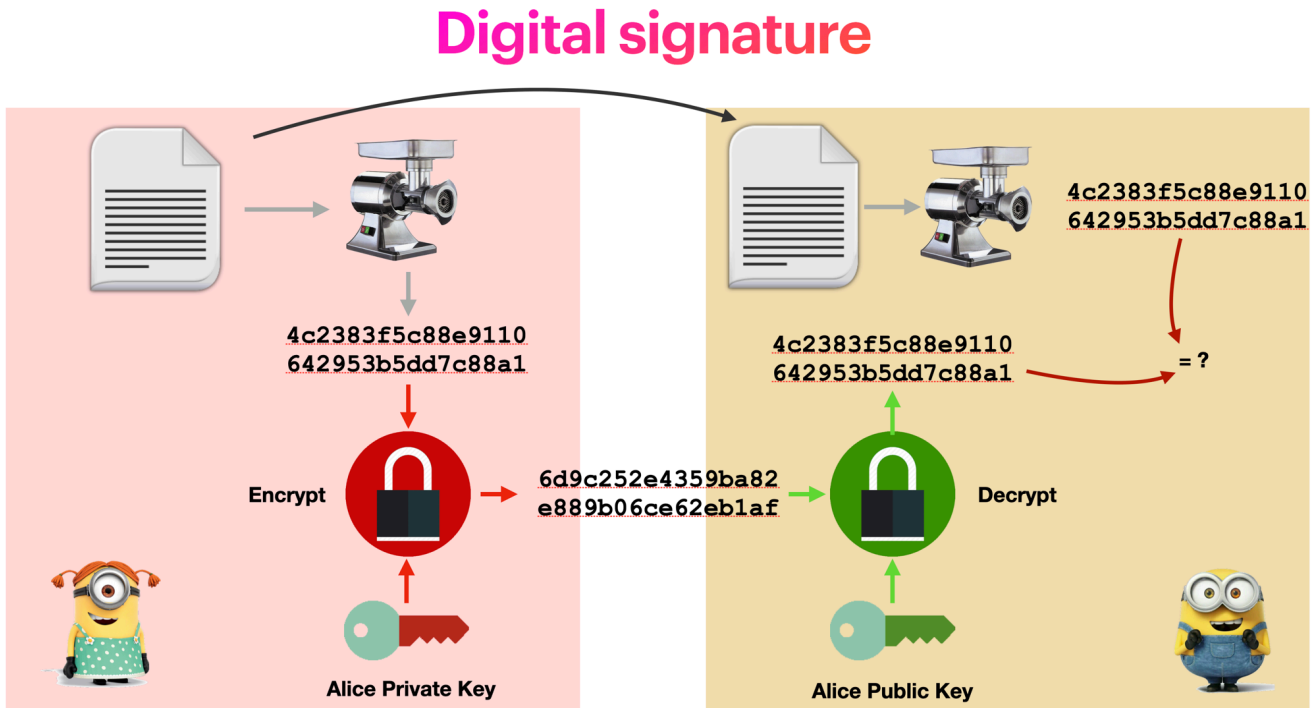
```
S = H(M)^d mod n
```

👨 Bob vérifie

1. Calcule **H(M)** de son côté #️⃣
2. Déchiffre S avec la **clé publique d'Alice** 🔓 → **H'**
3. Compare : **H(M) == H'** ?
   - ✅ Oui → authentique + intègre
   - ❌ Non → altéré ou faux

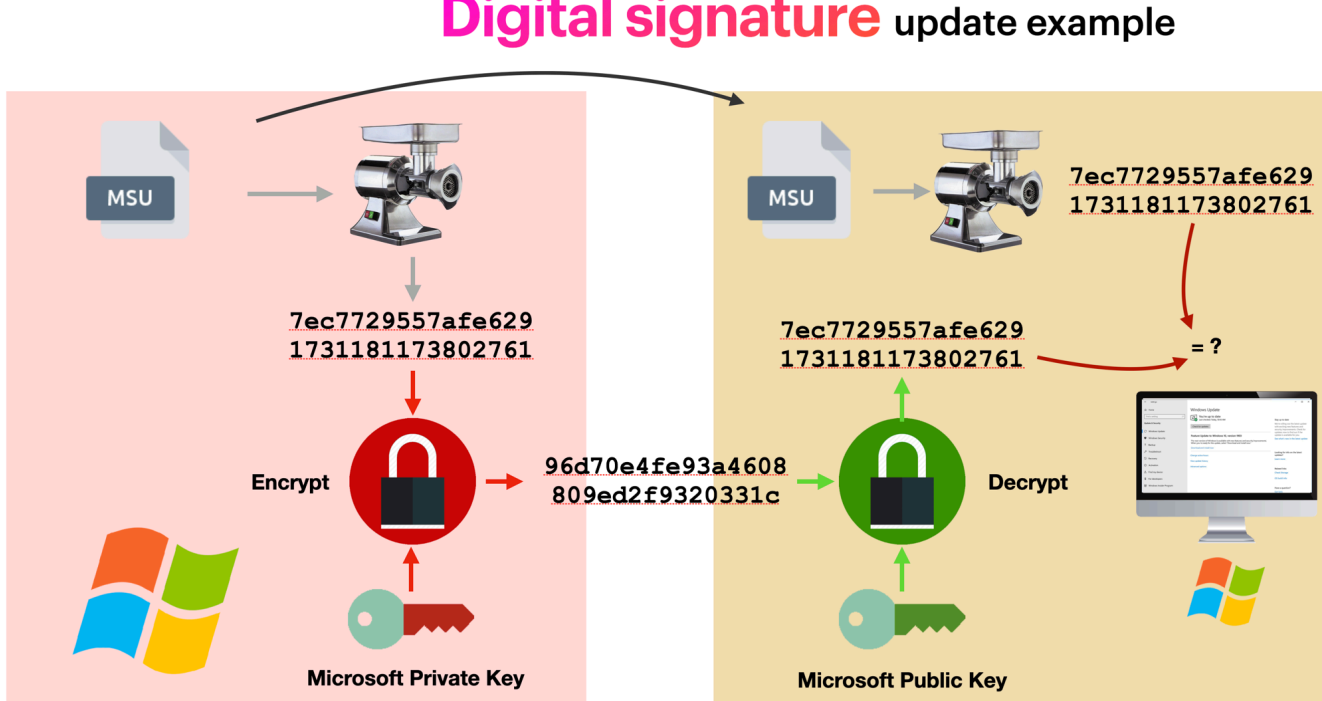
```
H' = S^e mod n
```

✍️ Signature : vue d'ensemble



🔴 Alice signe avec sa **clé privée** · 🟢 Bob vérifie avec la **clé publique d'Alice** · le hash recalculé doit correspondre.

🪟 Exemple concret : une mise à jour Windows



🏭 Microsoft signe le fichier `.MSU` avec sa **clé privée** · 💻 Votre Windows vérifie la signature avec la **clé publique Microsoft** intégrée à l'OS — sinon il refuse l'installation.

⚖️ Chiffrement vs Signature

	🔒 Chiffrement	✍️ Signature
Quel objectif ?	**Confidentialité**	**Authenticité + intégrité**
Qui chiffre/signe ?	L'**émetteur**	L'**émetteur**
Avec quelle clé ?	Clé **publique** du destinataire	Clé **privée** de l'émetteur
Qui déchiffre/vérifie ?	Le **destinataire**	**N'importe qui**
Avec quelle clé ?	Sa propre clé **privée**	Clé **publique** de l'émetteur

➡️ **Symétrie inverse** : pour chiffrer pour quelqu'un on utilise **sa** clé publique ; pour signer on utilise **sa propre** clé privée.

🔧 Signature en pratique : OpenSSL

```
# Alice signe un fichier (hash SHA-256 + signature RSA)
openssl dgst -sha256 -sign priv.pem -out message.sig message.txt

# Bob vérifie avec la clé publique d'Alice
openssl dgst -sha256 -verify pub.pem -signature message.sig message.txt
# → "Verified OK" ou "Verification Failure"
```

📝 Vous ferez ça en **TD3**.

Ce qui se passe en réalité :

1. OpenSSL calcule `H = SHA-256(message.txt)`
2. OpenSSL chiffre H avec la clé privée RSA → `signature`
3. À la vérification : recalcul du hash + déchiffrement de la signature + comparaison

🌍 Signatures partout

- 📦 **Logiciels** : Apple, Microsoft, Google signent leurs binaires
- 📱 **Apps mobiles** : Android demande chaque APK signé
- 🔄 **Mises à jour OS** : signature obligatoire (sinon l'OS refuse)
- 🪪 **Certificats TLS** : signés par une CA (chaîne de confiance — CM4)
- 💸 **Bitcoin** : chaque transaction est une signature ECDSA (cousine de RSA)
- 🛂 **Passeports biométriques** : la puce contient une signature de l'État

➡️ Sans signature numérique, **rien** ne marcherait dans la sécurité moderne.

🤝 Diffie-Hellman

L’échange de clé sans canal sécurisé

🤔 Rappel : à quoi sert DH ?

On a vu **RSA** : il permet de chiffrer et signer.

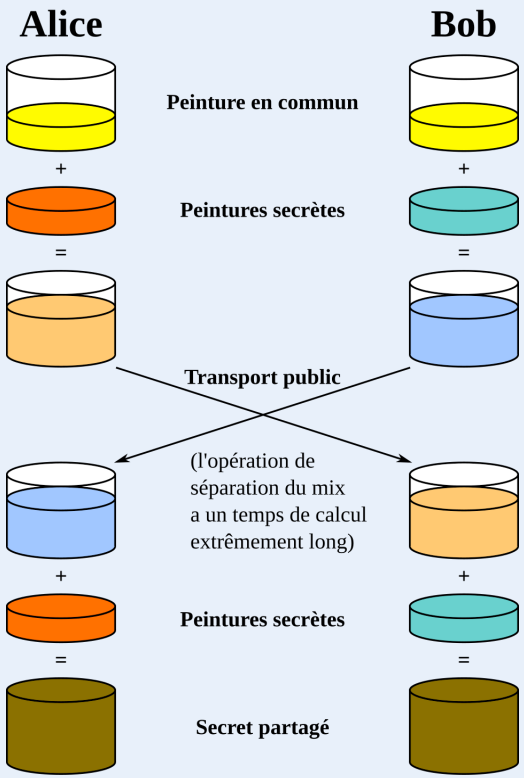
Mais RSA est **lent**. On veut continuer à utiliser AES (rapide).

➡️ Il faut un moyen pour Alice et Bob de **se mettre d'accord** sur une **clé AES** sans qu'Eve puisse l'apprendre.

C'est exactement ce que fait **Diffie-Hellman** :

- 🤝 Pas un chiffrement, juste un **protocole d'échange de clé**
- 🔢 À la fin, Alice et Bob partagent un **secret commun** que personne d'autre ne connaît
- 🔑 Ce secret sert ensuite de clé AES

🎨 L'analogie des couleurs



1️⃣ Alice et Bob conviennent d'une **peinture commune** 🟡 (publique)

2️⃣ Chacun a une **peinture secrète** : Alice 🔴, Bob 🔵

3️⃣ Chacun mélange sa peinture secrète avec la jaune → Alice 🟠, Bob 🟦 (échangés sur le réseau)

4️⃣ Chacun **rajoute sa propre peinture secrète** au mélange reçu

✅ Même couleur finale 🟫 — c'est le **secret partagé**

😈 Eve voit 🟡 🟠 🟦 mais ne peut pas **séparer** les pigments pour retrouver 🔴 ou 🔵.

🤔 Première idée : et si on multipliait ?

👩 Alice	😈 Eve	👨 Bob
	g = 17	
a = 4		**b = 6**
g × a = 68	Eve voit aussi : **68, 102**	**g × b = 102**
	↘️ échange ↙️	
g × b × a = 408		**g × a × b = 408**

✅ **Même secret partagé : 408** 🎉

😈 ...mais Eve casse le schéma en une division

Eve a écouté tout l'échange et connaît : $g = 17$ · $g \times a = 68$ · $g \times b = 102$

$$a = g \times a \div g$$

$$= 68 \div 17$$

$$= 4$$ 💀

La multiplication est trop facile à inverser.

➡️ **Il nous faut une opération à sens unique.**

💡 La parade : remplacer × par l'exponentiation

👩 Alice	😈 Eve	👨 Bob
	g	
a		b
g^a	↘️ ↙️	g^b
$(g^b)^a = g^{ab}$		$(g^a)^b = g^{ab}$

✅ Alice et Bob arrivent au même secret g^{ab}... mais Eve va-t-elle s'arrêter là ?

😈 Eve essaie aussi : le logarithme

Eve a écouté l'échange et connaît : $g \cdot g^a \cdot g^b$

📐 *Rappel : $\log(g^a) = a \cdot \log(g)$*

$$a = \log(g^a) \div \log(g)$$

$$= (a \cdot \log(g)) \div \log(g)$$

$$= a$$ 💀

Le log classique inverse l'exponentiation aussi facilement que la division inverse la multiplication.

➡️ **Il manque encore quelque chose pour rendre cette opération vraiment à sens unique.**

🪄 La vraie parade : tout faire modulo p

- 🔢 On choisit un **grand nombre premier p** (public)
- 🧮 Tous les calculs se font **modulo p** : Alice envoie **g^a mod p**, Bob envoie **g^b mod p**

😈 Eve voudrait toujours retrouver `a` à partir de `gᵃ mod p`…

➡️ C'est le **problème du logarithme discret** : la fonction `puissance mod p` mélange tellement les valeurs qu'**aucune formule** ne permet de l'inverser.

🌌 Sur **p de 2048 bits** : aucun algorithme efficace connu, **infaisable** avec les ordinateurs actuels.

🌐 C'est exactement ce qu'utilise **TLS 1.3** (DHE / ECDHE) — CM4.

🔄 DH éphémère et Forward Secrecy

DH éphémère (DHE) : Alice et Bob génèrent **a, b nouveaux** à chaque session.

- 🎲 La clé partagée K change à **chaque conversation**
- 🗑️ a, b sont **détruits** après usage
- 🛡️ Si Eve enregistre tout le trafic et compromet plus tard la clé long-terme → elle ne peut **toujours pas** déchiffrer les sessions passées

C'est la **Forward Secrecy** (PFS) — **obligatoire** dans TLS 1.3.

⚛️ **Très important face au quantique** : *harvest now, decrypt later*. Les agences enregistrent aujourd'hui le trafic chiffré, en attendant un ordinateur quantique pour le déchiffrer dans 10-20 ans. La forward secrecy est la première ligne de défense.

Le talon d’Achille de DH

🎭 Eve passive vs Eve active

Jusqu'ici, **Eve écoute** mais ne touche pas au trafic (passive).

Et si **Eve modifie** le trafic en temps réel ? ⚡

C'est l'attaque **Man-in-the-Middle (MITM)**.

➡️ DH est **vulnérable au MITM** car il n'authentifie **personne**.

Alice ne peut pas distinguer Bob d'un attaquant qui prétend être Bob.

😈 Attaque MITM sur Diffie-Hellman

👩 Alice	😈 Eve (au milieu)	👨 Bob
$a = 4$	$e_A = 2$ · $e_B = 3$	$b = 6$
	$g = 17$	
$g \times a = 68$ →	Eve intercepte à Alice (faux Bob)	
← Eve envoie $g \times e_A = 34$		
	Eve intercepte à Bob (fausse Alice)	← $g \times b = 102$
		Eve envoie $g \times e_B = 51$ →
Secret = $34 \times 4 = 136$	Avec Alice : 136 Avec Bob : 306	Secret = $51 \times 6 = 306$

⚠️ **Alice et Bob ont des secrets différents — Eve est au milieu et déchiffre tout.**

🛡️ Comment se protéger du MITM ?

Le problème : Alice **ne sait pas** que A vient bien de Bob (et inversement).

➡️ Il faut **authentifier** les messages échangés :

- ✍️ Alice **signe** A avec sa clé privée RSA
- 👨 Bob vérifie la signature avec la clé publique d'Alice
- ⛔ Si Eve injecte un faux A, sa signature ne correspondra pas → détection

Mais comment Bob est-il sûr de la clé publique d'Alice ?

➡️ Il faut un **tiers de confiance** : une **autorité de certification (CA)** qui signe les clés publiques.

C'est le rôle des **certificats X.509** — au CM4. 🪪

🧩 DH + RSA = la solution complète

Problème	Solution	Ce qu'on garantit
🤝 Échange de clé sans canal sûr	**Diffie-Hellman**	Confidentialité
🪪 Authentifier les parties	**RSA + certificats**	Authenticité
🔒 Chiffrer rapidement les données	**AES + clé DH**	Confidentialité (rapide)
✅ Vérifier l'intégrité	**HMAC** ou **AEAD**	Intégrité

➡️ C'est **exactement** ce que fait HTTPS / TLS. On verra le handshake complet au **CM4**. 🌐

📦 PGP

La crypto libre pour le peuple

🕊️ Phil Zimmermann et PGP (1991)

1991 : Phil Zimmermann publie **PGP** (*Pretty Good Privacy*), un logiciel libre combinant **RSA + AES + SHA + signatures**.

- 📨 Permet à n'importe qui de chiffrer et signer ses emails
- 🌍 Diffusé sur Internet → utilisé partout dans le monde
- ⚖️ **Le gouvernement américain l'inculpe** pour exportation d'armes (la crypto forte était classée munition militaire jusqu'en 1996)
- 📚 Contournement légendaire : Zimmermann publie le code source dans un **livre papier** (protégé par le 1er amendement) puis le scanne à l'étranger
- ✅ Procédure abandonnée en 1996. La crypto forte est désormais légale civilement.

➡️ Sans Zimmermann, on ne pourrait probablement pas utiliser HTTPS aujourd'hui sans demander une licence. 🇺🇸

📊 PGP en pratique aujourd'hui

GnuPG (GPG) : implémentation libre du standard OpenPGP.

- 🔐 Chiffrement et signature d'emails (Thunderbird + Enigmail)
- 📦 Signature des **paquets logiciels** : Debian, Ubuntu, Arch…
- 🔑 Authentification SSH par clé GPG
- 🗝️ **Web of Trust** : pas d'autorité centrale, les utilisateurs signent les clés des autres → confiance distribuée

⚠️ PGP a une réputation d'**ergonomie difficile**. Les messageries modernes (Signal, WhatsApp) utilisent les mêmes briques (DH, AES, signatures) mais cachent toute la complexité — c'est ce qui a permis l'adoption massive du chiffrement de bout en bout.

Synthèse

📝 À retenir

- 💡 **1976** : Diffie & Hellman inventent la crypto à clé publique
- 🧮 **RSA** : chiffrement asymétrique basé sur la **factorisation**
- 🔑 Paire de clés : **(n, e)** publique, **(n, d)** privée
- ⚡ RSA est lent → schéma **hybride** (RSA chiffre une clé AES)
- #️⃣ **Hachage** : empreinte irréversible (SHA-256). MD5/SHA-1 cassés.
- ✍️ **Signature** : H(M) chiffré avec clé privée → authenticité + intégrité
- 🤝 **Diffie-Hellman** : échange de clé via le **logarithme discret**
- 😈 DH **seul** est vulnérable au **MITM** → besoin d'authentification (certificats)
- 🌐 **DH + RSA + AES + SHA-256 = HTTPS** → CM4

🧪 TD3 : RSA & Diffie-Hellman

Vous allez :

- 🧮 Calculer **RSA à la main** (petits nombres : chiffrement, déchiffrement, signature)
- 👥 Faire un **Diffie-Hellman en trinôme** avec vos voisins (Alice, Bob, Eve)
- 😈 Réaliser une **attaque MITM** sur DH et constater l'absence d'authentification
- 🖥️ Manipuler **OpenSSL** : générer des clés, chiffrer, signer, vérifier
- 🌊 Observer concrètement l'**effet avalanche** sur SHA-256

❓ Questions ?

CM3 · Cryptographie asymétrique