

# Cryptographie

Cryptographie asymétrique

Alexander Schaub<sup>1</sup>

[schaub.alexander@free.fr](mailto:schaub.alexander@free.fr)

17/12/2025

---

<sup>1</sup>DGA-MI, Bruz

## Dans l'épisode précédent...

- On a vu comment protéger la **confidentialité** d'un message...
- ... si l'émetteur et le destinataire partagent la **même** clé

Mais alors :

## Dans l'épisode précédent...

- On a vu comment protéger la **confidentialité** d'un message...
- ... si l'émetteur et le destinataire partagent la **même** clé

Mais alors :

- comment partager la clé si on ne s'est **jamais vu** ?
- comment garantir que le message n'a pas été **modifié** ?
- comment garantir **l'origine** du message ?

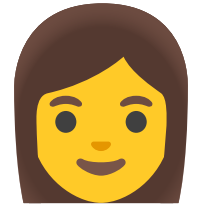
# Parlons un peu d'Alice et de Bob...



clé



Chiffré(clé, message)



clé



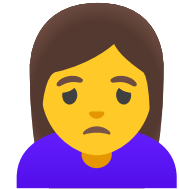
# Et s'ils ne se sont jamais vus ?



clé



Chiffré(clé, message)



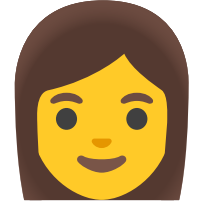
# Et s'ils ne se sont jamais vus ?



clé



clé, Chiffré(clé, message)



message

Trois services de la cryptographie  
asymétrique :

l'échange de clés,

le chiffrement asymétrique,

l'authenticité

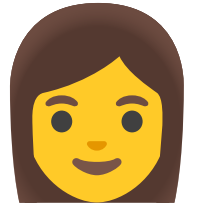
# Service n°1 : échange de clé



$K_{\text{priv}}^B$



$K_{\text{pub}}^B$



$K_{\text{priv}}^A$



$K_{\text{pub}}^B$



# Service n°1 : échange de clé

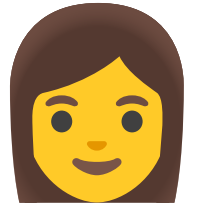


$K_{\text{priv}}^B$



$K_{\text{pub}}^A$

$K_{\text{priv}}^A, K_{\text{pub}}^B$

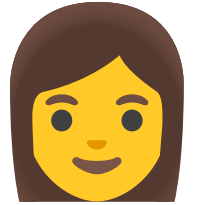


$K_{\text{pub}}^B, K_{\text{pub}}^A$

# Service n°1 : échange de clé



$K_{\text{priv}}^B, K_{\text{pub}}^A$



$K_{\text{priv}}^A, K_{\text{pub}}^B$

$$f(K_{\text{priv}}^B, K_{\text{pub}}^A) = K$$



$K_{\text{pub}}^B, K_{\text{pub}}^A$

$$K = f(K_{\text{priv}}^A, K_{\text{pub}}^B)$$

# Echange de clés de Diffie-Hellman

Comment trouver  $f$ ,  $K_{\text{priv}}$  et  $K_{\text{pub}}$  pour que cela fonctionne ?

1976: Schéma de Diffie-Hellman.

- Données partagées :  $p$  premier,  $g \in [1, p - 1]$ .
- Clés privées :  $a, b \in [1, p - 1]$
- Clés publiques :  $g^a[p], g^b[p]$
- Secret partagé :  $K = g^{ab}[p]$

Pourquoi ça marche ?  $(g^a)^b[p] = g^{ab}[p] = (g^b)^a[p]$

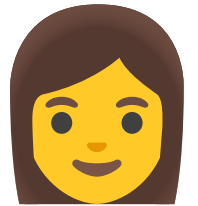
# Echangeons les clés avec Diffie-Hellman



$b$



$$B = g^B[p]$$



$a$



$B$

# Echangeons les clés avec Diffie-Hellman

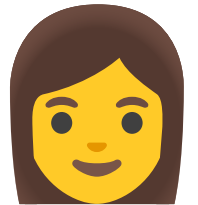


$b$



$$g^a[p] = A$$

$a, B$



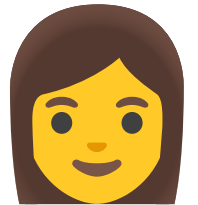
$B, A$

# Echangeons les clés avec Diffie-Hellman



$b, A$

$$A^b = K$$



$a, B$

$$K = B^a$$



$B, A$

## Et Eve dans tout ça ?

Eve connaît :

- $p, g$
- $A = g^a[p], B = g^b[p]$
- Doit calculer  $g^{ab}[p]$

Facile si elle pouvait calculer  $a$  à partir de  $g^a[p] \rightarrow$  problème du **logarithme discret** supposé difficile.

# Diffie-Hellman en pratique

- Dans le corps des entiers *modulo*  $p$  :  $p \approx 2048$  bits  $\rightarrow$  c'est beaucoup !
- On peut faire du Diffie-Hellman dans d'autres corps : les **courbes elliptiques** (on parle de ECDH - *Elliptic Curve Diffie-Hellman*)
  - il faut bien choisir ses courbes, son implémentation, etc...
  - mais la taille du corps est plus petite : entre 256 et  $\approx 500$  bits !
    - courbes elliptiques : solutions d'une équation de type  $y^2 = x^3 + ax + b$  dans un corps fini...



# Service n°2 : le chiffrement asymétrique

## Ou encapsulation de clé

Problème de Diffie-Hellman : protocole **interactif**

- Alice et Bob doivent **tous deux** échanger des messages avant d'établir la clé
- Parfois, ce n'est pas possible, on aimerait pouvoir utiliser un protocole **plus simple**

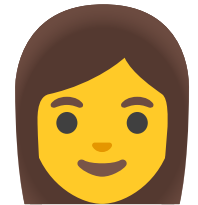
# Le chiffrement asymétrique



$K_{\text{priv}}^B$



$K_{\text{pub}}^B, K$




$$C = \text{Chif}(K_{\text{pub}}^B, K)$$

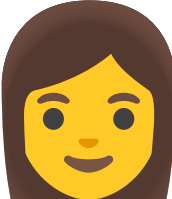


$K_{\text{pub}}^B$

# Le chiffrement asymétrique


$$K = \text{Déchif}(K_{\text{priv}}^B, C)$$

---

$$K_{\text{pub}}^B, K$$



$$K_{\text{pub}}^B$$

# Chiffrement asymétrique : pour résumer

Dans un système de chiffrement asymétrique :

- la clé privée est **secrète**, la clé publique est **connue de tous**
- on **chiffre** avec la **clé publique**, on **déchiffre** avec la clé **privée**
- on peut facilement retrouver la clé **publique** à partir de la clé **privée**  
(mais l'inverse n'est pas possible !)

# Chiffrement asymétrique : notation

Un système de chiffrement **asymétrique** se compose de:

- Quatre ensembles  $E$ ,  $F$ ,  $K$  (clés privées),  $K'$  (clés publiques)
- D'une fonction à **sens unique**  $h : K \mapsto K'$
- De deux fonctions,  $f : E \times K \mapsto F$ ,  $g : F \times K' \mapsto E$  telles que
  - $\forall x \in E, k \in K, g(f(x, h(k)), k) = x$
  - On ne doit pas pouvoir retrouver  $x$  à partir de  $f(x, h(k))$  sans connaître  $k$

# Le 1er algorithme de chiffrement asymétrique : RSA

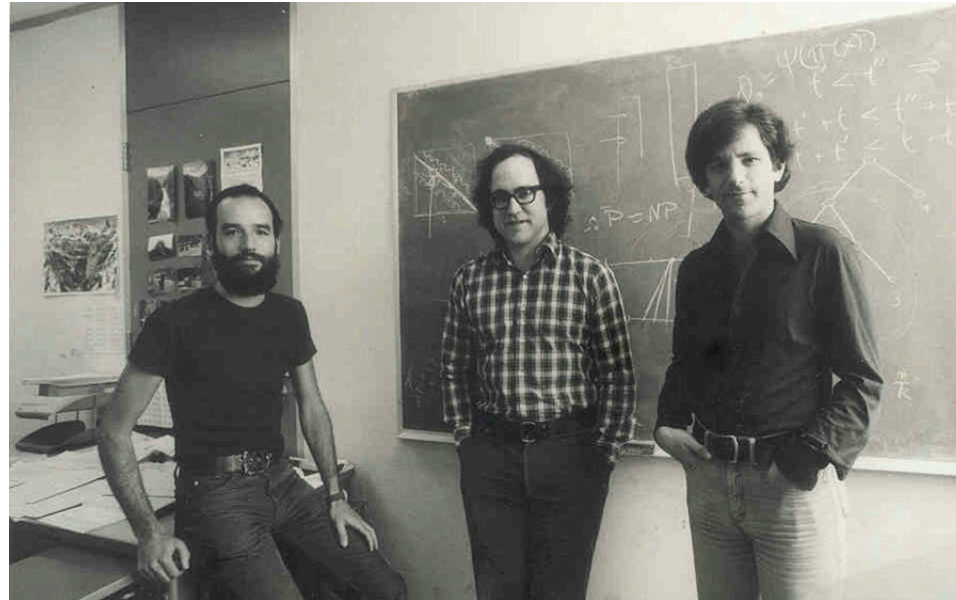


Image 1: Rivest, Shamir, Adleman

# RSA : définitions

Publié en 1977.

- Clé publique :
  - $n$  produit de deux grands nombre premiers  $p$  et  $q$
  - $e$ , un nombre quelconque premier avec  $(p - 1)(q - 1)$
- Clé privée :
  - $n$  comme précédemment
  - $d$  inverse de  $e$  modulo  $(p - 1)(q - 1)$

# RSA : chiffrons et déchiffrons !

Rappels:

$$n = p \cdot q$$

$$\varphi(n) = (p-1)(q-1)$$

$$\text{pgcd}(e, \varphi(n)) = 1$$

$$e \cdot d \equiv 1[\varphi(n)]$$

Pour chiffrer  $m < n$  :

- $C = m^e[n]$

Pour déchiffrer :

- $m = C^d[n]$

Pourquoi ça marche ?

$$m^{e \cdot d} = m[n] ???$$



# RSA : chiffrons et déchiffrons !

Rappels:

$$n = p \cdot q$$

$$\varphi(n) = (p-1)(q-1)$$

$$\text{pgcd}(e, \varphi(n)) = 1$$

$$e \cdot d \equiv 1[\varphi(n)]$$

Pour chiffrer  $m < n$  :

- $C = m^e[n]$

Pour déchiffrer :

- $m = C^d[n]$

Pourquoi ça marche ?

$$m^{e \cdot d} = m[n] ???$$

$$\text{En fait, } \forall m, m^{\varphi(n)} = 1[n]$$

Du coup, comme  $e \cdot d = 1 + k \cdot \varphi(n)$ ,

$$\begin{aligned} m^{e \cdot d}[n] &= m^{1+k \cdot \varphi(n)}[n] \\ &= m \cdot m^{k \cdot \varphi(n)}[n] \\ &= m \cdot (m^{\varphi(n)})^k[n] = m[n] \end{aligned}$$

## Pourquoi c'est sûr ?

Un attaquant doit pouvoir retrouver  $(p - 1)(q - 1)$  à partir de  $n \rightarrow$  cela revient à trouver  $p$  et  $q$ , donc de **factoriser**  $n$ .

Il existe peut-être des méthodes plus efficaces mais elle ne sont pas connues.

On considère que  $n \approx 2048$  bits confère une bonne sécurité aujourd'hui, et  $n \approx 4096$  bits confère suffisamment de sécurité pour toutes les applications usuelles.

## Envoyons des clés !

$m$  : clé AES = entre 128 bits et 256 bits

$e$  : historiquement la valeur 3 était beaucoup utilisée et elle est valide

Il suffit de générer  $p, q$ , calculer  $n$  et envoyer  $m^3[n]$  !

## Envoyons des clés !

$m$  : clé AES = entre 128 bits et 256 bits

$e$  : historiquement la valeur 3 était beaucoup utilisée et elle est valide

Il suffit de générer  $p, q$ , calculer  $n$  et envoyer  $m^3[n]$  !

## NE FAITES JAMAIS ÇA

$m^3$  est plus petit que  $n$ , il suffit de prendre la racine troisième pour décrypter

## Quand est-ce que RSA est sûr ?

La sécurité de RSA n'est effective **uniquement pour  $m$  généré uniformément dans  $[1; n - 1]$** . Pour simuler cela :

### Le retour des paddings !

- PKCS#1 v1.5 : PS est une chaîne aléatoire (d'octets non nuls) de longueur suffisante, et on chiffre :

0x00 || 0x02 || PS || 0x00 || m

- OAEP : mieux, plus moderne, plus sûr et plus compliqué

## Choix de $e$

Choix historique  $e = 3$  déconseillé (certaines attaques sont plus faciles avec  $e$  petit)

Tout le monde ou presque utilise  $e = 65537 = 2^{16} + 1$

# Chiffrer avec Diffie-Hellman ?

Cryptosystème de ElGamal (1985) :

On choisit  $p$  premier,  $g \in [1, p - 1]$  comme paramètres publics.

La clé privée est  $x \in [1, p - 1]$  choisie uniformément.

La clé publique est  $h = g^x$ .

Le chiffré est  $(c_1, c_2) = (g^y[p], m \cdot h^y[p])$  ( $y$  choisi uniformément)

Pour déchiffrer :  $s = c_1^x = h^y[p]$  puis  $m = c_2 \cdot s^{-1}[p]$

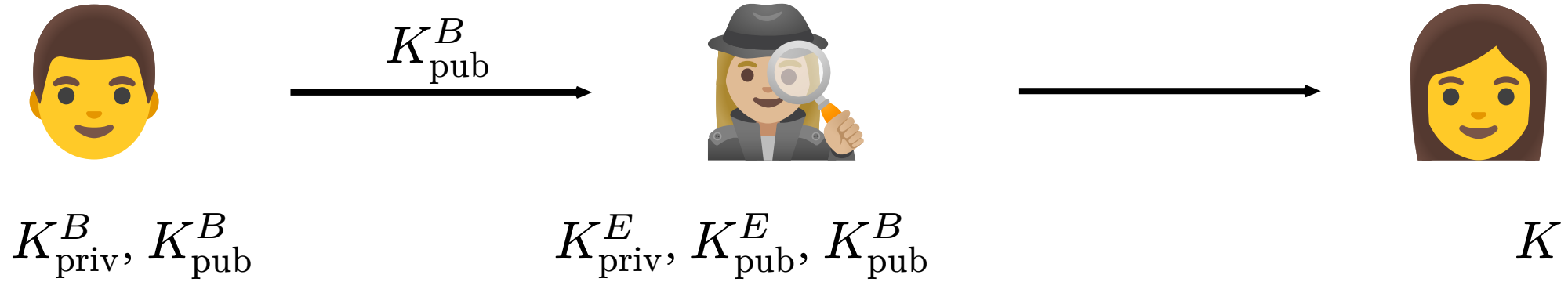
Algorithme général marche aussi sur des courbes elliptiques

## Interlude : le projet !

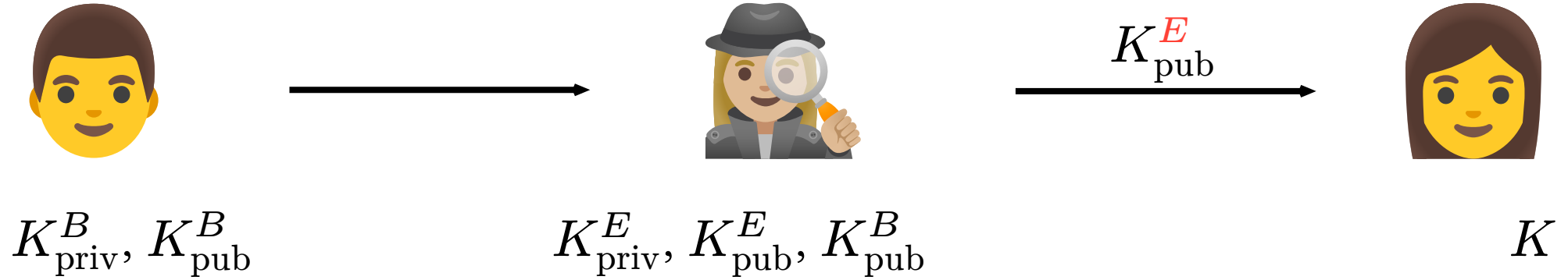
- Groupes de 2 ou 3
- Proposition de sujet d'ici la prochaine séance
- Noté sur des présentations de 30 (?) min



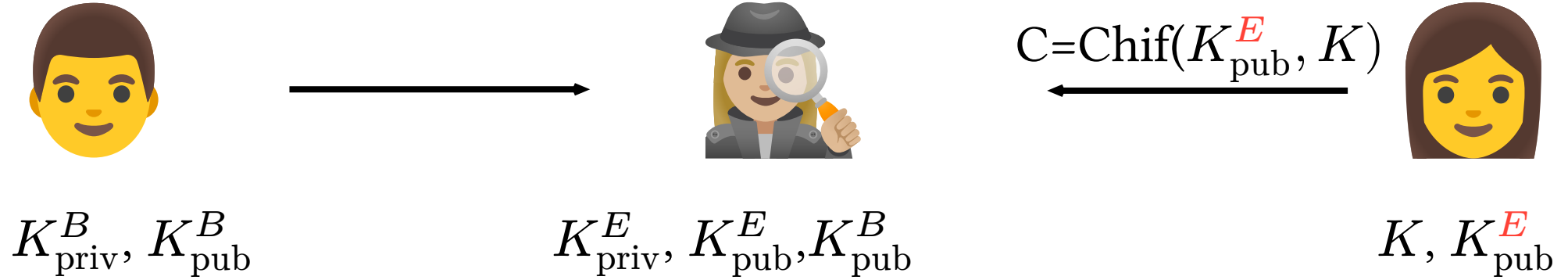
## Service n°3 : l'authenticité



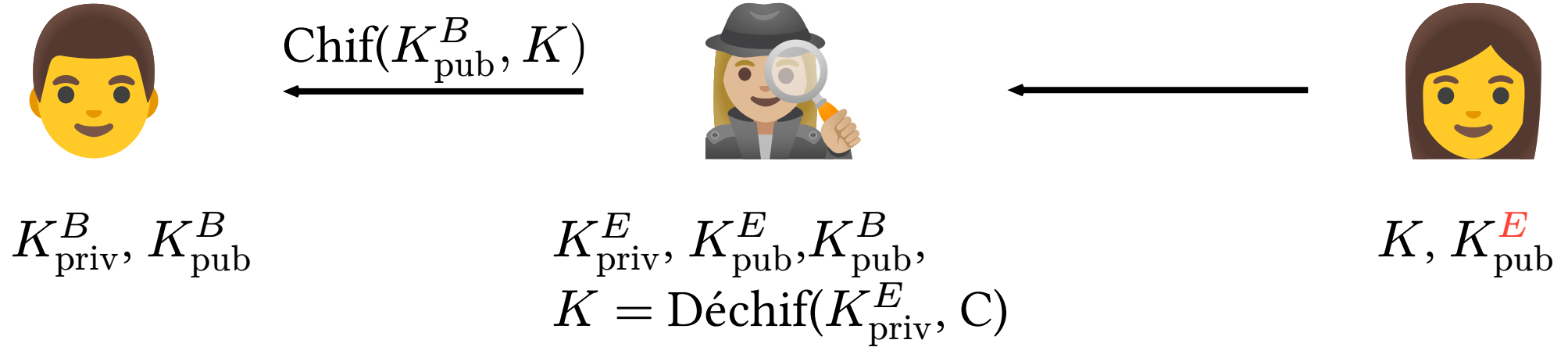
## Service n°3 : l'authenticité



## Service n°3 : l'authenticité



## Service n°3 : l'authenticité



## L'attaque de l'homme-du-milieu (*man-in-the-middle*)

Comme on a pu le voir :

- un attaquant **actif** peut intercepter **et modifier** des messages entre correspondants
- en cas de réussite, les correspondants **ne se doutent de rien** mais la **confidentialité** de leurs échanges va être **compromise**
- possible car rien ne garantit **l'origine** des messages reçus (ici, principalement  $K_{\text{pub}}^B$ )

# Les signatures électroniques

Permet de garantir **l'origine** du message.

Seul le détenteur de la **clé de génération de signature** peut authentifier des messages, tous les détenteurs de la **clé de vérification de signature** peuvent vérifier leur authenticité.

## Signature électronique : notation

- Quatre ensembles  $E$  (messages),  $F$  (signatures),  $K$  (clés de génération),  $K'$  (clés de vérification)
- D'une fonction à **sens unique**  $h : K \mapsto K'$
- De deux fonctions,  $f : E \times K \mapsto F$ ,  $g : F \times E \times K' \mapsto \{\text{true}, \text{false}\}$  telles que
  - $\forall x \in E, k \in K, g(f(x, k), x, h(k)) = \text{true}$
  - Trouver  $y \in F$  tel que  $g(y, x, h(k)) = \text{true}$  revient à connaître  $k$

# Un exemple de signature électronique : RSA !

- Clé de vérification :
  - $n$  produit de deux grands nombre premiers  $p$  et  $q$
  - $e$ , un nombre quelconque premier avec  $(p - 1)(q - 1)$
- Clé de génération :
  - $n$  comme précédemment
  - $d$  inverse de  $e$  modulo  $(p - 1)(q - 1)$



# RSA : signons et vérifions !

Rappels:

$$n = p \cdot q$$

$$\varphi(n) = (p-1)(q-1)$$

$$\text{pgcd}(e, \varphi(n)) = 1$$

$$e \cdot d \equiv 1[\varphi(n)]$$

Pour signer  $m < n$  :

$$S = m^d[n]$$

Pour vérifier :

$$S^e[n] = m \rightarrow \text{true sinon false}$$

Pourquoi ça marche ?

$$m^{e \cdot d} = m[n] \rightarrow \text{ah oui on vient de le voir}$$

## Signons des clés !

$m$  : clé AES = entre 128 bits et 256 bits

On envoie  $m^d[n]$ , et c'est bon cette fois non ?

## Signons des clés !

$m$  : clé AES = entre 128 bits et 256 bits

On envoie  $m^d[n]$ , et c'est bon cette fois non ?

**NE FAITES JAMAIS ÇA**

On peut trivialement générer des signatures pour tous les messages de type  $r^e[n]$ ,  $r$  quelconque par exemple.

En plus : on aimerait pouvoir signer des messages plus long que 4096 bits. Comment peut-on faire ?

## Aparté : les fonctions de hachage

On ne peut signer que des éléments relativement *courts*

## Aparté : les fonctions de hachage

On ne peut signer que des éléments relativement *courts*

Il faut donc **réduire** les messages avant de les signer

## Aparté : les fonctions de hachage

On ne peut signer que des éléments relativement *courts*

Il faut donc **réduire** les messages avant de les signer

Mais attention ! pas n'importe comment !

Il ne faut pas réduire **deux messages** de la **même façon**

## Fonctions de hachage : les attendus

Une **fonction de hachage** est une fonction  $f : \{0, 1\}^* \mapsto \{0, 1\}^n$  :

- résistante aux collisions : on ne doit pas pouvoir trouver  $m_1 \neq m_2$  tels que

$$f(m_1) = f(m_2)$$

- résistante à l'inversion : étant donné  $y \in \{0, 1\}^n$ , on ne doit pas pouvoir trouver  $m$  tel que

$$f(m) = y$$

## Fonctions de hachage : les bonus

En général, on peut aussi s'attendre aux propriétés suivantes :

- Indistinguishable de l'aléa : une fonction de hachage  $f$  est idéalement **indistinguishable** d'une fonction choisie au hasard
- Propriété *d'avalanche* : en changeant **1** bit de l'entrée, chaque bit de sortie a **une chance sur deux** d'être inversé
  - (c'est impliqué par la propriété précédente, voyez-vous pourquoi ?)



# Les fonctions de hachage modernes

- ~~MD4, MD5, SHA1~~ : on oublie, c'est cassé
- SHA2 : existe en plusieurs versions, en fonction de la taille de sortie :
  - SHA256, SHA384, SHA512, SHA512-256, SHA512-384
- SHA3 (*alias* Keccak ) : une famille de fonctions
  - Existe en version à tailles standardisées (256, 384, 512 bits)
  - Existe aussi en version à taille arbitraire

## Signatures : le retour

- On ne signe pas les clés, on signe **le haché** des clés
- On ne les signe pas non plus **directement**, c'est le retour du **padding** :

- ▶ PKCS#1 v1.5 (Signature) : déterministe

0x00 || 0x01 || FFFF...FFFF || 0x00 || Type de hash || m

- ▶ PSS : probabiliste et plus moderne
  - et également plus compliqué à décrire

# C'est tout pour aujourd'hui !

La séance prochaine, nous verrons :

- quelques services de plus : intégrité, dérivation de clés
- comment assembler les services en des **protocoles** de la vraie vie : TLS, SSH
- en bonus : un peu de temps pour le projet