Cryptographie

Authentification

Gabriel Chênevert

16 décembre 2025

Aujourd'hui

Courbes elliptiques (en 5 minutes)

Authentification de message

Signatures

Certificats

Rappel : DLP

Dans n'importe quel groupe fini $(G, \boldsymbol{+})$, l'exponentiation en base $g \in G$ fixé :

$$\ell \mapsto \ell \boldsymbol{\times} g$$

est une fonction $\mathrm{ord}_G(g)$-périodique efficacement calculable (en $\mathcal{O}(\log \ell)$)

dont l'inverse :

$$\mathrm{dlog}_G(x, g) = \ell \quad \Longleftrightarrow \quad x = \ell \boldsymbol{\times} g$$

l'est en général beaucoup moins (meilleur algorithme générique en $\mathcal{O}(\sqrt{\mathrm{ord}_G(g)})$).

Qu'est-ce qu'une courbe elliptique ?

Rien d'autre qu'une façon de fabriquer des groupes finis pour lesquels le DLP est génériquement difficile.

Paramètres :

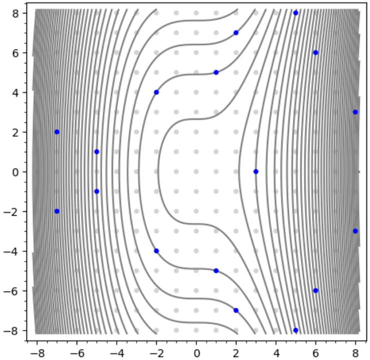
- un nombre premier $p > 2$
- deux coefficients a et b tels que $4a^3 + 27b^2 \underset{p}{\not\equiv} 0$

Éléments :

- les couples $(x, y) \in (\mathbb{Z}/p\mathbb{Z})^2$ tels que $y^2 \underset{p}{\equiv} x^3 + ax + b$
- ainsi qu'un point spécial à l'infini $O = (?, \infty)$

Exemple

$$y^2 \underset{17}{\equiv} x^3 + 7$$




```python
from ECC import *

E = EllipticCurve(17,0,7)

P = E.point(8,3)

P + P
```

```
(5,8)
```

D'un point de vue purement opérationnel, c'est tout ce qu'il y à savoir !

Aujourd'hui

Courbes elliptiques (en 5 minutes)

Authentification de message

Signatures

Certificats

Notion d'authentification

Indépendemment de toute notion de confidentialité, une propriété désirable d'un encodage est de pouvoir garantir l'*authenticité* d'un message même en présence d'adversaires.

Lorsque Bob reçoit un message d'Alice, comment garantir que celui-ci n'a pas été modifié en transit par Oscar ?

Idée : ajouter une empreinte (somme de contrôle)

On peut ajouter une valeur déduite du message (donc redondante) afin d'en vérifier l'intégrité.

Alice : ajoute à son message m une empreinte $h = H(m)$.

Bob : vérifie à la réception de (m, h) que si $h = H(m)$.

(Sinon : problème de transmission détecté, soit dans m soit dans h)

Exemple

$m =$ Tu me dois 10 €

$h =$ `c7b12b33fdd17399`

$m_{\text{reçu}} =$ Tu me dois 10 €

$h_{\text{reçu}} =$ `c7b12b33fdd17399`

$h_{\text{calculé}} =$ `c7b12b33fdd17399`

Ok !

Exemple (suite)

$m =$ Tu me dois 10 €

$h =$ `c7b12b33fdd17399`

$m_{\text{reçu}} =$ Tu me dois 100 €

$h_{\text{reçu}} =$ `c7b12b33fdd17399`

$h_{\text{calculé}} =$ `08821af9be531f29`

Erreur !

Propriétés désirables de H

- taille fixe : $H : \{0,1\}^* \to \{0,1\}^n$ avec n fixé
- **déterminisme** : $m = m' \implies H(m) = H(m')$
- **avalanche** : $m \approx m', m \neq m' \implies H(m) \not\approx H(m')$

  (exactement l'opposé de la **continuité**)
- **résistance aux collisions** : en pratique, difficile de trouver $m \neq m'$ tels quel

$$H(m) = H(m')$$

Fonctions d'empreinte cryptographiques

Une *fonction d'empreinte cryptographique* sur n bits est une fonction possédant les propriétés précédentes.

Une attaque générique sur une fonction d'empreinte sur n bits génère des collisions en $\mathcal{O}(2^{n/2})$ étapes $\implies$ au maximum $n/2$ bits de sécurité.

Exemples :

- MD5 (1991) $n = 128$ dépréciée
- SHA-1 (1995) $n = 160$ dépréciée
- SHA-2 (2001) $n = 256$ ou 512
- BLAKE2 (2008) $n = 256$ ou 512
- SHA-3 (2012) $n = 256$ ou 512

Exemples

```python
from hashlib import md5, sha1, sha256
```

```python
message = b"A hash function turns arbitrary inputs into digests of fixed size"

print("msg :", message)
print()
print("MD5 :", md5(message).hexdigest())    # 128 bits
print("SHA1:", sha1(message).hexdigest())   # 160 bits
print("SHA2:", sha256(message).hexdigest()) # 256 bits
```

```
msg : b'A hash function turns arbitrary inputs into digests of fixed size'

MD5 : faf72bc60a7d6011de46eb3d275c7335
SHA1: 1571fbba39bc28b1cf77ad88f55db6000ce6d980
SHA2: 42830b8a9452e04b8909d902ebd8b7fe9d6de88931ffd77a66c6b68da64c3d7b
```

Mais ça ne suffit pas...

$m =$ Tu me dois 100 €

$h =$ `08821af9be531f29`

$m_{\text{received}} =$ Tu me dois 100 €

$h_{\text{received}} =$ `08821af9be531f29`

$h_{\text{computed}} =$ `08821af9be531f29`

Ok ! ...

Problème

Même si la valeur de H ne peut pas être manipulée . . .

n'importe qui peut calculer une empreinte valide !

Si on introduit une clé symétrique, on obtient la notion de *code de vérification de message* (MAC) :

par exemple HMAC (construit à partir d'une fonction d'empreinte) ou CBC-MAC (construit à partir d'un chiffrement par blocs)

Primitives composites

Si on souhaite à la fois garantir la confidentialité et l'intégrité des données : *chiffrement authentifié*

Obtenu en combinant un chiffrement symétrique + un MAC ou en utilisant un mode opératoire dédié

(ex. : OCB, EAX, EtM, GCM, CCM, . . .).

Attention : le chiffrement authentifié n'empêche pas les *attaques par rejeu* en lui-même

$\Longrightarrow$ *Authenticated Encryption with Associated Data* (AEAD)

le message est chiffré, le message + les métadonnées sont authentifiées

Aujourd'hui

Courbes elliptiques (en 5 minutes)

Authentification de message

Signatures

Certificats

Le problème avec les MACs

Alice et Bob ont exactement les mêmes capacités, donc ce système ne peut pas les protéger *contre l'autre*

Falsification :

Bob : "Je m'appelle Alice et dois 100 € à Bob." ✗

Répudiation :

Alice : "Je m'appelle Alice et dois 100 € à Bob." ✓

Alice : "Hé ce n'est pas moi qui ai dit ça, c'était Bob !" ✗

La signature numérique fournit

- authentification de l’émetteur
- intégrité du message
- lien entre le message et l’émetteur
- non-falsification
- non-répudiation

Définition

Un *schéma de signature numérique* est une paire d'algorithmes :

- **signature** $S(k_{\text{priv}}, m)$
- **vérification** $V(k_{\text{pub}}, m, s) \in \{0, 1\}$

(ainsi qu'une **génération de clés** qui produit des paires valides $(k_{\text{priv}}, k_{\text{pub}})$)

Construction standard

À partir d'une fonction d'empreinte cryptographique et d'un cryptosystème asymétrique.

Pour signer un message m avec clé privée k_e :

- Alice calcule $h = H(m)$;
- ajoute $s = E(k_e, h)$ à m.

À la réception d'une paire (m, s), Bob :

- vérifie avec la clé public associée si

$$D(k_d, s) \stackrel{?}{=} H(m).$$

Quelques algorithmes de signature

- RSA-PSS
- DSA (basé sur le DLP modulaire)
- ECDSA (basé sur le DLP sur les courbes elliptiques)

mais ces deux derniers (standards du NIST) sont des constructions *ad hoc* sans preuve de sécurité formelle

Mieux : les *signatures de Schnorr* pour lesquelles on sait qu'elles sont aussi dures à falsifier que le DLP sous-jacent (Seurin 2012)

Algorithme de signature de Schnorr (1/2)

Paramètres:

- un groupe $(G, \boldsymbol{+})$ et $g \in G$ d'ordre premier q pour lequel le DLP est difficile
- une fonction d'empreinte cryptographique $H : \{0,1\}^* \to [\![0, q[\![$

Clés:

- privée $k \in]\!]0, q[\![$
- publique $p = k \boldsymbol{\times} g$.

Algorithme de signature de Schnorr (2/2)

Signature d'un message m :

- on choisit $r \in]\!]0, q[\![$ aléatoire, à usage unique
- on calcule $e = H(r \,✖\, g \parallel m)$, $s \underset{q}{\equiv} r - ke$
- la signature est (s, e)

Vérification :

- on teste si $H((s \,✖\, g) \,✚\, (e \,✖\, p) \parallel m) \stackrel{?}{=} e$

Aujourd'hui

Courbes elliptiques (en 5 minutes)

Authentification de message

Signatures

Certificats

Le problème avec les clés publiques

Bob peut vérifier les signatures d'Alice à condition de disposer de sa clé publique.

Alice peut la diffuser publiquement...

...mais comment s'assurer qu'il s'agit bien de la sienne (i.e. l'authentifier) ?

De retour à la case départ (encore)

Certificat

Un tiers de confiance *certifie* la paire (Alice, k_{pub}) en diffusant :

"Je certifie que la clé publique d'Alice est k_{pub}."

Protocole revu

Pour signer un message m, Alice :

- calcule $s = S(k_{\text{priv}}, m)$
- envoie (m, s) ainsi que son certificat pour k_{pub}

Bob :

- vérifie que le certificat et valide
- vérifie la signature en utilisant k_{pub}

Gestion de la confiance

Deux approches principales :

- décentralisée (ex. PGP)
- hiérarchique (ex. X.509):

  chaînes d'autorités de certification (CAs), listes de révocation . . .

Racines de confiance : `/etc/ssl/certs` (Linux), `certmgr.msc` (Windows)