Université Claude Bernard Lyon 1

LIFPROJET — Printemps 2018 fabrice.mouhartem@ens-lyon.fr

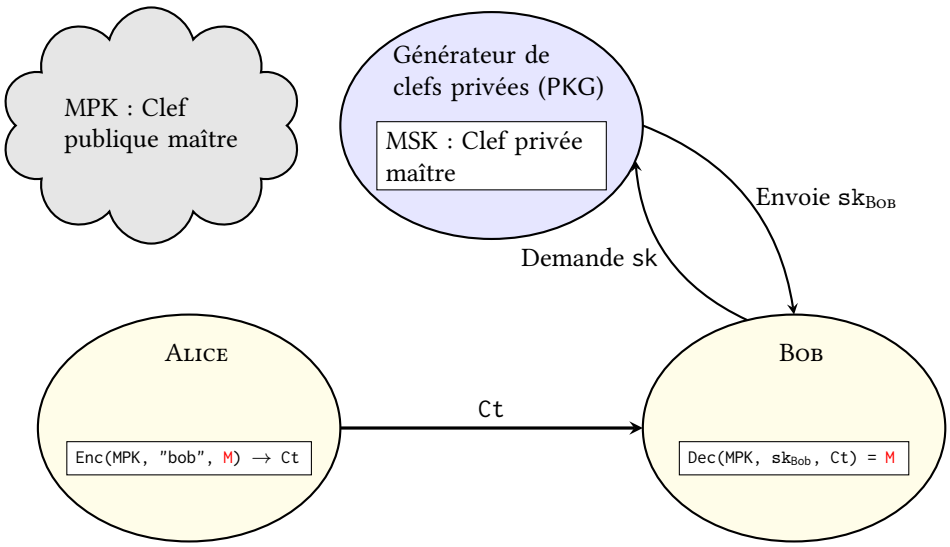
FM1. Chiffrement fondé sur l'identité

MISE EN GARDE. Ceci est un brouillon incomplet et non relu qui est là pour donner des indications en début de projet. Il sera proprifié par la suite.

1 Introduction

Le but de ce projet est de mettre en place un système de chiffrement « *avancé* », c'est-à-dire qui va au delà des méthodes classiques de chiffrement/déchiffrement où chaque utilisateur envoie sa clef publique à un autre utilisateur. En effet, cette méthode pose le problème de l'échange de clefs sécurisé. Par exemple, sur ma page professionnelle, je donne un lien vers ma clef publique. En revanche rien ne garanti que mon site n'ait pas été mis à défaut par une personne tierce, qui aurait échangé ma clef.

Pour répondre à ce problème, nous nous intéressons à une solution proposée pour la première fois par Adi Shamir en 1984 [Sha84] : le **chiffrement fondé sur l'identité** (ou IBE). Le principe est le suivant, on déplace le problème de l'échange de clefs pour que la génération de la clef se fasse localement à partir d'une *identité*. C'est-à-dire une chaîne qui identifie uniquement son propriétaire, comme une adresse e-mail par exemple. Les interactions au sein du système sont décrites en figure 1.



SOURCE : https://en.wikipedia.org/wiki/ID-based_encryption

FIGURE 1 – Chiffrement fondé sur l'identité.

De cette manière, au lieu de devoir sécuriser chaque échange de clefs, l'établissement d'un canal sécurisé se fait uniquement entre la première interaction d'un nouvel utilisateur (par exemple BOB), et de l'autorité de gestion des clefs (le PKG).

Ici, nous avons pour le moment regardé le point de vue théorique de la construction. En pratique, il s'avère que ces constructions sont coûteuses. On va donc utiliser la technique du *chiffrement hybride*.

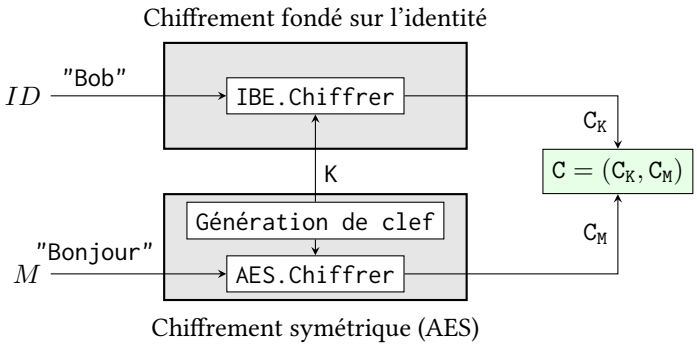


FIGURE 2 – Chiffrement hybride

C'est-à-dire qu'on va encapsuler notre message dans un `chiffrement symétrique`. Cette technique est illustrée en figure 2. Comme on utilisera une bibliothèque (comme `OpenSSL`) pour le chiffrement symétrique, un autre avantage de cette méthode sera que l'on n'aura pas besoin de se demander comment encoder les messages longs (par exemple si on a un e-mail qui contient un `PDF`).

2 Descriptions des différents IBE

Un IBE est donné par les quatre algorithmes suivants dont le comportement est décrit en figure 1.

Initialisation : Cet algorithme prend en entrée un paramètre de sécurité et renvoie les paramètres publiques MPK, et la clef secrète maître MSK qui sera gardée par le PKG.

Extraction : Cet algorithme lancé par le PKG prend en entrée une identité $ID \in \{0,1\}^\star$ et la clef secrète maître MSK et renvoie une clef secrète pour l'identité ID : $\mathtt{sk}_{ID}$

Chiffrer : À partir d'une identité ID et d'un message M, cet algorithme renvoie un chiffré `C` pour l'identité ID.

Déchiffrer : À partir d'un chiffré `C` et d'une clef secrète $\mathtt{sk}_{ID}$, cet algorithme renvoie un message M ou un symbole d'erreur $\perp$ (par exemple en levant une exception).

2.1 L'IBE de Cocks

Cet IBE [Coc01] repose sur l'arithmétique modulaire (de manière similaire à RSA). On va travailler dans l'anneau $\mathbb{Z}_{/n\mathbb{Z}}$, où n est un semi-premier assez grand.

Ici, le message à chiffrer correspond à un bit, encodé comme 1 ou -1. Ainsi pour chiffrer plusieurs bits, il faut lancer le chiffrement sur plusieurs bits en parallèle.

NOTATION. Dans ce qui suit, $\left(\frac{a}{n}\right)$ est de *symbole de Jacobi* de a et n. C'est la généralisation du symbole de Legendre (qui permet de savoir si un nombre est un carré ou non modulo p premier). Celui-ci se calcule dans *gmp* en utilisant la fonction `int mpz_jacobi (const mpz_t a, const mpz_t p)`.

Les quatre algorithmes décrivant ce schéma sont comme suit.

Initialisation : Cet algorithme commence par choisir un module RSA $n = p \cdot q$ où p et q sont des entiers de Blum, c'est-à-dire des nombres premiers tels que $p \equiv 3 \bmod 4$ et $q \equiv 3 \bmod 4$. Les entiers p et q sont de taille 2^λ où $\lambda \in \{1024, 2048, 4096\}$. De plus cet algorithme spécifie une fonction de hachage cryptographique $\mathcal{H}$ (par exemple `SHA-2`).

Les paramètres publiques sont $\mathsf{MPK} \coloneqq (n, \mathcal{H})$ et la clef secrète est $\mathsf{MSK} \coloneqq (p, q)$.

Extraction : À partir de $\mathsf{MSK} = (p, q)$, $\mathsf{MPK} = (n, \mathcal{H})$ et l'identité ID, cet algorithme va calculer les itérées de $\mathcal{H}$ sur ID jusqu'à ce que $\left(\frac{\mathcal{H}(\mathcal{H}(\cdots(\mathcal{H}(ID))\cdots))}{n}\right) = 1$. Cet entier trouvé (c'est-à-dire $\mathcal{H}(\mathcal{H}(\cdots(\mathcal{H}(ID)\cdots)))$ sera noté a dans la suite.

On calcule ensuite $\mathtt{sk}_{ID} = a^{\frac{n+5-p-q}{8}} \bmod n$ et on envoie ensuite à l'utilisateur.

Chiffrer : Pour chiffrer un bit vu comme 1 ou -1, pour l'identité ID, on commence par tirer deux entiers t_1 et t_2 différents tels que $\left(\frac{t_1}{n}\right) = \left(\frac{t_2}{n}\right) = m$ (ça devrait être le cas au bout de 5–6 essais). On recalcule ensuite a comme dans l'extraction.

Le chiffré est composé de deux parties : $c_1 = t_1 + a \cdot t_1^{-1} \bmod n$ et $c_2 = t_2 - a \cdot t_2^{-1} \bmod n$. Ici, t_1^{-1} réfère à l'inverse modulaire de t_1 modulo n. C'est-à-dire un élément tel que $t_1^{-1} \cdot t_1 = 1 \bmod n$. Celui-ci existera dans la majorité des cas (si on trouve un élément qui n'est pas inversible, alors on a de bonnes chances d'avoir trouvé un facteur de n, et ça sent pas bon pour la sécurité du protocole). Pour le calculer, *gmp* propose la fonction `int mpz_invert (mpz_t rop, const mpz_t op1, const mpz_t op2)`. Finalement, on renvoie $c = (c_1, c_2)$.

Déchiffrer : Pour déchiffrer le chiffré $c = (c_1, c_2)$ à l'aide de la clef secrète $\mathtt{sk}_{ID} = r$, on commence par calculer la valeur

$$\alpha \coloneqq \begin{cases} c_1 + 2r & \text{si } r^2 = a \bmod n \\ c_2 + 2r & \text{si } r^2 = -a \bmod n \end{cases}$$

Ici encore, $r^2 = -a$ peut sembler contre-intuitif. Mais on travaille modulo n. Le calcul de $\mathtt{sk}_{ID}$ nous donne en effet une racine carrée de a ou $-a$. Pour finalement calculer et renvoyer m comme $m \coloneqq \left(\frac{\alpha}{n}\right)$.

Comment utiliser GMP ? La bibliothèque *gmp* propose une interface en C++ qui propose toutes les surcharges d'opérateurs usuels (`+`, `*`, `+`, `-`, `%`, …) pour la classe `class mpz_class`.
Pour cela il vous suffira d'inclure le fichier `gpmxx.h` : `#include <gmpxx.h>`, et de compiler avec les options `-lgmp -lgmpxx`.

2.2 L'IBE de Boneh-Franklin

Cet IBE repose sur les groupes munis d'un couplage cryptographique. Un couplage est une application bilinéaire décrite en définition 1. Cette structure a été utilisée pour la première fois en cryptographie par Antoine Joux en 2000 [Jou00].

Définition 1. Soient $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ deux groupes cycliques d'ordre p. On note g un générateur de $\mathbb{G}$. Un couplage est une application $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ qui vérifie les propriétés suivantes :

1. Bilinéarité : pour tout $a, b \in \mathbb{Z}_p$, $e(g^a, g^b) = e(g^b, g^a) = e(g, g)^{ab}$.
2. Non-dégénérescence : $e(g, h) = 1_{\mathbb{G}_T} \iff g = 1_{\mathbb{G}}$ ou $h = 1_{\mathbb{G}}$.
3. Le couplage est calculable efficacement.

Cette définition est suffisante pour manipuler les couplages en utilisant la propriété (1). Nous utiliserons les couplages de Barreto-Naehrig implantés dans la bibliothèque Relic : https://github.com/relic-toolkit.

Cet IBE [BF03] fonctionne comme suit.

Initialisation : Cet algorithme choisit un couplage sûr pour le paramètre λ, c'est-à-dire $\mathbb{G}, \mathbb{G}_T, g$ et e comme définis en définition 1.

On tire ensuite un entier $\alpha \hookleftarrow \mathbb{Z}_p = \{0, 1, 2, \ldots, p-1\}$ uniformément au hasard, qui formera la clef secrète maître MSK. À partir de celle-ci, on calcule un élément de groupe $h = g^\alpha$ qui fera parti de la clef publique.

Comme dans l'IBE de Cocks, on définit une fonction de hachage $\mathcal{H} : \{0,1\} \to \mathbb{G}$; par exemple `SHA-2` où la sortie est interprété comme élément de groupe, par exemple via la fonction de relic : `g2_map(g2_t rop, uint8_t* vect, size_t len)`.
Les paramètres publiques sont donc $\mathsf{MPK} \coloneqq (\mathbb{G}, \mathbb{G}_T, g, h = g^\alpha, \mathcal{H})$ et la clef secrète maître est $\mathsf{MSK} \coloneqq \alpha$.

Extraction : Pour extraire la clef secrète d'une identité ID, le PKG va calculer $\mathtt{sk}_{ID} \coloneqq \mathcal{H}(ID)^\alpha$.

Chiffrer : Pour chiffrer un message $m \in \mathbb{G}_T$ (vous choisirez un encodage cryptographiquement sûr. Posez moi des questions pour plus de détails) pour l'identité ID, l'utilisateur va tiré un aléa (qu'on peut voir comme un *nonce*) $r \hookleftarrow \mathbb{Z}_p$ et va l'utiliser pour calculer le chiffré en deux parties (comme dans l'IBE de Cocks) : $c \coloneqq (c_1, c_2) = (g^r, M \cdot e(h, \mathcal{H}(ID))^r)$.

Déchiffrer : Pour déchiffrer un message à l'aide de $\mathtt{sk}_{ID} = \mathcal{H}(ID)^\alpha$, cet algorithme calcule et renvoie le message $m = c_2 / e(c_1, \mathtt{sk}_{ID})$.

Précaution. Nous avons utilisé des notations de couplage « symétriques » pour simplifier les notations. Dans Relic, les couplages sont vu comme une fonction $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. Cela ne change fondamentalement rien, mais vous devrez faire attention aux groupes d'arrivées.

Explications. Ce cryptosystème peut être vu comme une généralisation du cryptosystème d'ElGamal (utilisé dans gnupg par exemple) où la génération de clefs est calculée à l'aide du couplage. C'est ainsi que l'on arrive à déléguer le calcul de la clef publique de Bob sans révéler d'information sur la clef secrète maître.

Comment utiliser Relic ? Pour l'installation de Relic, soit vous êtes sur votre machine où vous avez les droits root, auquel cas il n'est pas besoin de s'embêter : suivez le `readme`.
Autrement, il vous faudra ruser un peu. Une recommandation est d'installer relic dans votre répertoire local. Pour cela :

```
cd chemin/vers/relic
mkdir build
cd build
cmake -DCMAKE_INSTALL_PREFIX=~/.local ..
make -j
make install
```

Il vous faudra ensuite spécifier à votre compilateur favoris le lien vers relic :
`-I /home/username/.local/include/relic -L /home/username/.local/lib -lrelic`
Et finalement au lancement indiquer la position de `librelic.so` :
`LD_LIBRARY_PATH=/home/username/.local/lib ./executable`
Ce qui peut être rajouté dans votre `.bashrc` (ou `.zshrc` suivant votre *shell*) par exemple :
`export LD_LIBRARY_PATH=/home/username/.local/lib`.
Attention, il s'agit d'une bibliothèque C, pour l'utiliser en C++, il vous faudra spécifier à votre compilateur qu'il s'agit de C :

```
extern "C" {
#include <relic.h>
}
```

Autrement le compilateur pensera qu'il s'agit de C++, où le nom des fonctions appelées par le linker sont modifiées pour permettre l'utilisation d'espace de noms, ce qui n'est pas fait en C.

3 Consignes

La manière dont le PKG est instancié, ainsi que la vérification de l'identité correspondante à l'e-mail attribué est laissé au choix des groupes.

La première étape consistera à choisir la manière dont votre infrastructure sera construite, et m'envoyer des explications courtes mais complètes pour validation. Un cahier des charges prévisionnel sera établi pour la suite du projet.

Ensuite, il s'agira de construire les différentes parties de l'application : d'abord le cœur du sujet, c'est-à-dire le chiffrement fondé sur l'identité, et ensuite rajouter la structure du chiffrement hybride autour, et son déploiement sur une architecture contrôlable *via* la ligne de commande.

La dernière étape consistera en l'écriture d'une extension pour `thunderbird` qui servira à faciliter l'utilisation du programme.

Tout au long du développement, les fonctions devront être implantés de manière modulaire, par exemple en utilisant le système de classes et de *templates* du `C++` ; mais pas obligatoirement, la seule contrainte est que le code soit lisible et facilement réinstanciable. Pour cela une documentation claire sera demandée (cf. `doxygen` par exemple… même si avouons le-nous, les documentations `doxygen` ne sont pas très claires, mais ont le mérite d'être complètes).

Il est recommandé de faire des tests via `Sagemath` pour se familiariser avec les protocoles et faire des tests rapides. Cette partie sera développée dans le rapport.

Références

[Sha84] Adi Shamir, Identity-Based Cryptosystem and Signature Schemes. In *Crypto*, 1984, Springer.

[Coc01] Clifford Cocks, An Identity Based Encryption Scheme Based on Quadratic Residues. In *ICCC*, 2001.

[Jou00] Antoine Joux, A One Round Protocol for Tripartite Diffie–Hellman. In *ANTS*, 2000, Springer.

[BF03] Dan Boneh et Matt Franklin, Identity based encryption from the Weil pairing. In *SIAM J. of Computing*, 2003, SIAM.