HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

TIC
Télécommunication
Sécurité de l'information

# Travail de Bachelor

Chiffrement / Signature d'emails

**Non Confidentiel**

| | |
|---|---|
| **Étudiant :** | **Bonjour Mickael** |
| **Travail proposé par :** | Duc Alexandre |
| | Nom de l'entreprise/institution |
| | Adresse |
| | NPA Ville |
| **Enseignant responsable :** | Duc Alexandre |
| **Année académique** | 2019-2020 |

Yverdon-les-Bains, le 19 juin 2020

HAUTE ÉCOLE
D'INGÉNIERIE ET DE GESTION
DU CANTON DE VAUD
www.heig-vd.ch

Mickael Bonjour

Département TIC
Filière Télécommunications
Orientation Sécurité de l'information
Étudiant Bonjour Mickael
Enseignant responsable Duc Alexandre

# Travail de Bachelor 2019-2020

Chiffrement / Signature d'emails

---

Nom de l'entreprise/institution

**Résumé publiable**

Dans ce travail de bachelor je vais analyser les solutions actuelles de messagerie sécurisée afin d'identifier les primitives cryptographiques dont j'ai besoin. Ensuite j'établirais un *Proof Of Concept* amenant la technologie choisie dans un cadre de messagerie électronique sécurisée.

| | | |
|---|---|---|
| Étudiant : | Date et lieu : | Signature : |
| Bonjour Mickael | ..... | ..... |
| Enseignant responsable : | Date et lieu : | Signature : |
| Duc Alexandre | ..... | ..... |
| Nom de l'entreprise/institution : | Date et lieu : | Signature : |
| Nom Prénom de la personne confiant l'étude | ..... | ..... |

# Préambule

Ce travail de Bachelor (ci-après TB) est réalisé en fin de cursus d'études, en vue de l'obtention du titre de Bachelor of Science HES-SO en Ingénierie / Economie d'entreprise.

En tant que travail académique, son contenu, sans préjuger de sa valeur, n'engage ni la responsabilité de l'auteur, ni celles du jury du travail de Bachelor et de l'Ecole.

Toute utilisation, même partielle, de ce TB doit être faite dans le respect du droit d'auteur.

HEIG-VD

Vincent Peiris
Chef de département TIC

Yverdon-les-Bains, le 19 juin 2020

# Authentification

Le soussigné, Mickael Bonjour, atteste par la présente avoir réalisé ce travail et n'avoir utilisé aucune autre source que celles expressément mentionnées.

Yverdon-les-bains, le 19 juin 2020

Mickael Bonjour

# Table des matières

# Chapitre 1

# Cahier des charges

## Résumé du problème

Les outils de chiffrement et de signature d'email actuels se résument principalement à S/MIME et à PGP.

Ces deux solutions sont anciennes, souffrent assez régulièrement de nouvelles vulnérabilités et ne proposent pas certaines propriétés cryptographiques qui pourraient être utiles (par exemple, la "forward secrecy". Le but de ce travail de bachelor est d'étudier quelles propriétés seraient utiles pour la sécurisation des emails, de proposer un nouveau protocole les implémentant et de développer un proof of concept.

### Problématique

La problématique principale est résumée ci-dessus mais le principal problème c'est surtout que les technologies utilisées sont vieilles et elles souffrent de vulnérabilités par conception qui ont été mitigées en enlevant des options à l'utilisateur.

### Solutions existantes

PGP, S/MIME, PEP, messagerie instantanée (Signal)...

### Solutions possibles

Une solution possible est d'utiliser le système mis en place dans ce travail, cependant il faudrait une relecture et des analyses plus approfondies pour s'en assurer. Un des points possibles c'est de passer plus à de la messagerie instantanée dans la mesure du possible. Ou encore de s'orienter sur des nouvelles technologies comme PEP ou PGP mais en appliquant strictement les *Best Practices* et en se formant un peu sur leur utilisation qui n'est pas donnée à tout le monde.

# Cahier des charges

Voici un résumé du cahier des charges sous formes d'objectifs à atteindre :

— Analyser les besoins d'un système d'E-mails actuel.
— Analyser et étudier les solutions de sécurité existantes.
— Comprendre et évaluer les propriétés cryptographiques défendues.
— Établir une liste des propriétés cryptographiques voulues pour un système de mails sécurisés.
— Trouver une primitive cryptographique satisfaisant les besoins énoncés et l'étudier pour en comprendre les bases et les besoins nécessaires en termes de sécurité.
— Établir la spécification pour un nouveau protocole en utilisant la primitive choisie.
— Faire un Proof Of Concept du protocole proposé.

Si le temps le permet :

— Comprendre plus en détails les mathématiques derrière la primitive utilisée.
— Faire un prototype de client mail utilisant une architecture mise en place pour le POC.

## Déroulement

Tout d'abord je vais m'intéresser à faire une évaluation des concepts existants en messagerie sécurisée, tel que PGP et S/MIME pour les emails ou encore Signal pour la messagerie instantanée. Ayant vu ce qu'il se fait j'essaie de trouver une solution alternative pour le chiffrement et la signature d'emails. De là je vais conceptualiser un protocole et l'implémenter au sein d'un *Proof Of Concept*.

## Livrables

Les délivrables seront les suivants :

1. Une documentation contenant :
   — Une analyse de l'état de l'art
   — La décision qui découle de l'analyse
   — Spécifications
   — L'implémentation faites et les choix faits
   — Proof Of Concept
   — Les problèmes connus
2. Le code du *Proof Of Concept* fait, expliqué à l'aide de commentaires.

# Chapitre 2

# Introduction

Ce travail de Bachelor a pour but de sensibiliser à la vulnérabilité dans les systèmes actuels de messagerie électronique. Il propose aussi un nouveau protocole permettant de sécuriser ce type de messagerie à l'aide d'une primitive cryptographique peu implémentées, le *Certificateless Public Key Cryptography*. Ma démarche ans ce travail de bachelor est de voir si des solutions s'offrent à nous en considérons ce qu'il se fait sur le marché actuellement. Et en essayant d'améliorer les solutions actuelles proposées qui peuvent souffrir d'un manque de sécurité assez souvent ou (et plus souvent) un manque de simplicité d'utilisation.
Ce travail est découpé en plusieurs parties. En effet, on commence par une analyse de l'état de l'art, donc à regarder ce qui existe et voir pourquoi il faudrait de nouvelles solutions. Puis une présentation de la primitive cryptographique utilisée pour ma proposition dans ce travail. Enfin la présentation de l'architecture de mon protocole et une implémentation proposée en *Proof Of Concept* ainsi que les choix importants qui ont été faits en rapport à cette implémentation.

# Chapitre 3

# Analyse - État de l'art

La problématique principale est la difficulté d'utilisation des sécurités mises en places au-dessus des protocoles de base pour les emails. De plus des vulnérabilités (EFAIL) qui réussissent à récupérer le texte clair a démontré que la sécurité n'était pas bien implémentée. Les vulnérabilités proviennent plus d'un défaut de conception inhérent aux mails. Je vais ici décrire les principaux problèmes trouvés sur PGP et S/MIME lors de mes tests d'utilisation et ce que j'ai trouvé durant mes recherches. De plus je vais analyser des systèmes de mails sécurisés tel que Protonmail et Tutanota.

## 3.1 Protocoles existants

### 3.1.1 PGP

**Fonctionnement.** PGP (Pretty Good Privacy ou Assez bonne confidentialité) est un moyen de chiffrer des données (mails, fichiers, ...) qui est beaucoup représenté lorsque l'on parle de sécurité email car c'est le plus utilisé avec S/MIME (c.f. chapitre suivant). C'est une méthode de chiffrement hybride (utilise le chiffrement symétrique et assymétrique) qui fonctionne comme montré sur la figure 3.1.
Ce fonctionnement hybride est défendu à cause de la lenteur et la non-praticité d'un chiffrement asymétrique sur un certain nombre de données. Ainsi en chiffrant uniquement la clé symétrique qui a servi à chiffrer le tout l'on peut déchiffrer bien plus rapidement et simplement le message (typiquement avec un chiffrement symétrique tel qu'AES qui a le droit à des instructions dédiées dans certains processeurs). Contrairement à des chiffrements asymétriques qui sont plus contraignants. Et l'on n'utilise pas directement le chiffrement symétrique car il a besoin d'un secret partagé dès le début de la communication. PGP utilise un système de clés... PGP est aussi critiqué pour son manque de "Forward Secrecy"...

**Propriétés cryptographiques** Le problème qui est souvent reproché à PGP c'est qu'il n'implémentes pas de *Forward Secrecy*. La *Forward Secrecy* permet d'affirmer que si l'on a une brèche à un instant $T$, et qu'un attaquant récupère cette clé, il ne pourra pas déchiffrer

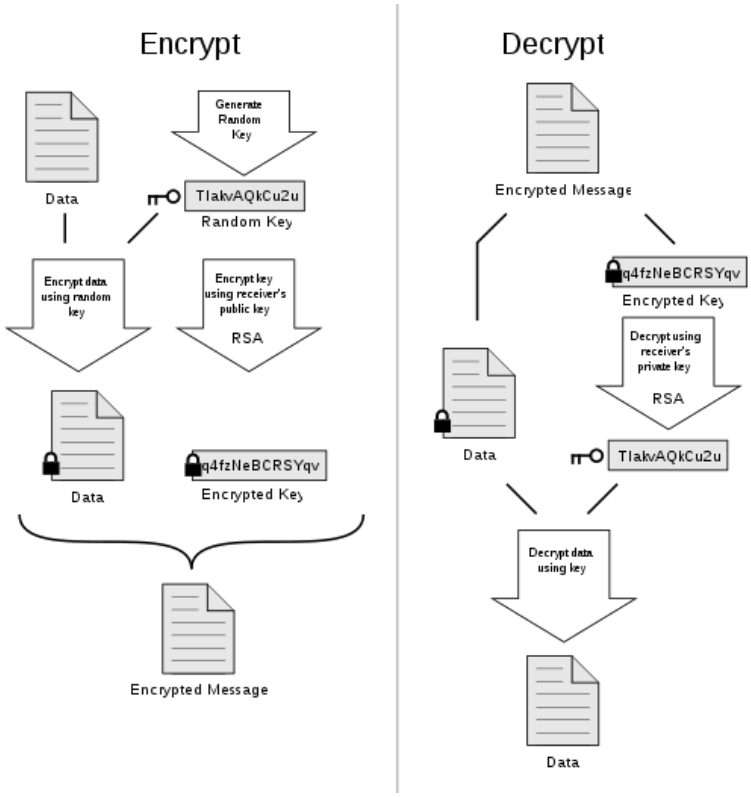


FIGURE 3.1 – Le fonctionnement global de PGP

les anciens messages. De plus, la gestion des clés PGP est très problématique, en effet lors de mes tests il était difficile de connecter un serveur de clés p. ex. Ou de recevoir une clé d'un correspondant pour la sauvegarder. Et même en la recevant, comment savoir si cette clé n'a pas été modifiée via un *MITM* p.ex. ? -> utiliser un autre canal pour vérifier l'empreinte.

**Web of Trust** Comment faire confiance a une clé -> surtout pour email -> Comment initialiser une confiance ?

**Autocrypt** Autocrypt est une manière d'échanger des clés entres emails, ces échanges ne sont pas considérés sécurisé par la communauté (Wikipedia -> à creuser). C'est une façon de s'échanger des clés de manière automatisée mais pas forcément sécurisée (utilise les mails).

**Utilisation** Pour mes tests j'ai fait en sorte de trouver l'utilisation la plus simple possible pour voir si un utilisateur lambda pouvait arriver à mettre en place ce genre de sécurité. Il s'est avéré que cela était assez simple au départ, mais dès lors que l'on veut envoyer un mail chiffré à un correspondant cela se complique un peu. J'ai juste eu à installer un

Add-On sur mon logiciel de messagerie (Thunderbird dans mon cas) qui s'appelle Enigmail. Ensuite Enigmail a générer mes clés PGP (de manière totallement opaque -> à creuser). Puis j'ai écrit un mail, en appuyant sur un petit cadenas mon mail partait chiffré et signé (uniquement si on a la clé du correspondant). Bien, cependant c'est très opaque et on ne sait pas ce qu'Enigmail et Autocrypt font réellement derrière les décors. L'utilisateur doit encore choisir s'il veut chiffrer ses mails ou non par contre il faut que le destinataire utilise PGP et que l'on ait sa clé publique. J'ai donc expérimenté à plus bas niveau ce qu'il se passait.

### 3.1.2 PEP

**Citation** *By default, communications between pep peers always work end-to-end encrypted – no eavesdrop-ping in between shall be possible by design.*

**Utilisation** pep assure un chiffrement de bout-en-bout par design, ils n'ont en effet pas de serveurs en soit et chiffre à l'aide d'un *handshake* fait entres les deux personnes via des *trustwords*. Ce sont des mots qu'il faut vérifier entres les deux partis afin d'être sûr que la connexion est bien authentifiée. -> à creuser mais à priori PEP utilise PGP pour le chiffrement des messages.

### 3.1.3 S/MIME

**Fonctionnement** Basé sur le même principe que PGP principalement, mais avec des certificats pour prouver la légitimité des clés publiques. Pour l'utilisation il faut se créer un certificat, plusieurs classes de confiance existe.

**Propriétés cryptographiques**

**Utilisation**

## 3.2 Implémentations existantes

### 3.2.1 Protonmail

**Revendications** Protonmail revendique beaucoup de propriétés cryptographiques, tel que le zero-access encryption. Et l'end-to-end chiffrement + zero-knowledge pour les messages sécurisés, même avec leur fonctionnalité de (Chiffrement vers l'extérieur) utilisant AES256-GCM. Pour l'authentification Protonmail utilise une manière fortement sécurisée (SRP) pour ne pas avoir d'informations direct sur le mot de passe de l'utilisateur.

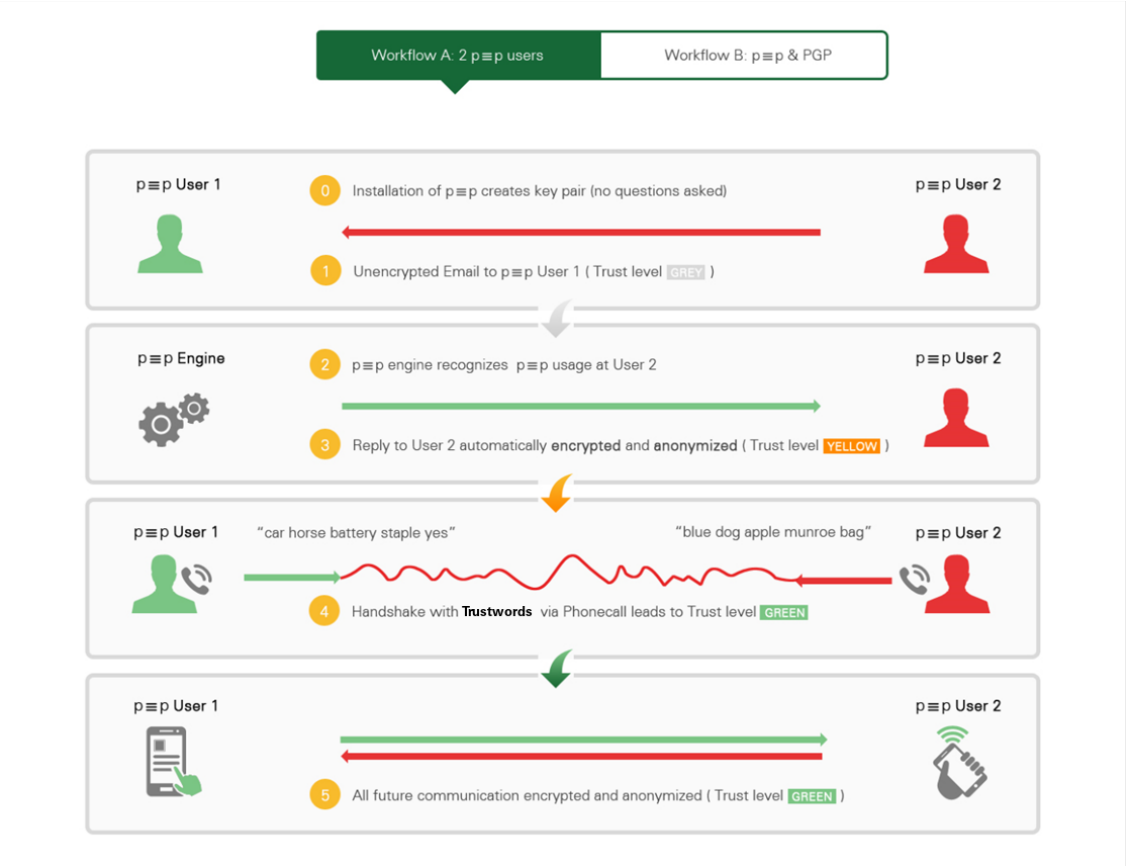


FIGURE 3.2 – Le fonctionnement global de pEp

**Fonctionnement** Protonmail a plusieurs modes de fonctionnement dépendant du destinataire final. En effet de Protonmail à Protonmail les mails sont chiffrés à l'aide de PGP automatiquement. L'on peut utiliser Protonmail pour utiliser PGP si l'on a la clé de notre destinataire par exemple. Et l'on peut écrire un mail chiffré à quelqu'un qui n'utilise pas PGP grâce à une fonctionnalité de chiffrement vers l'extérieur. Cette fonctionnalité enverra une URL au destinataire qui, en la consultant, pourra déchiffrer le mail en utilisant un mot de passe communiqué de manière sécurisées entres les deux partis auparavant.

**Open Source** Tout leur code est open-source afin d'avoir une validation externe, de plus ils ont un programme de Bug Bounty pour les chercheurs.

### 3.2.2 Tutanota

**Fonctionnement** Tout ce que j'ai vu pour le moment c'est que Tutanota utilise AES128-CBC ? Mais dans PGP ou ailleurs ?

## 3.3 Attaque existantes

### 3.3.1 Défauts webmail

Selon un chercheur[4] l'infrastructure de Protonmail aurait des failles vis son webmail. Mais son papier est en fait plus général et parle des webmails en règle général. Il part du principe que les serveurs de Protonmail ne sont pas des serveurs à faire confiance, pour ainsi prouver le zero-knowledge de Protonmail. Par contre, le fait qu'il ne peuvent pas être mis en confiance est un problème selon lui, car c'est ces serveurs qui vont délivrer le code d'OpenPGP afin de faire le chiffrement. Cela indique que si Protonmail était corrompu le fait d'avoir le code délivré par Protonmail pourrait avoir des effets néfastes. Comme p.ex l'extraction de la clé privée PGP. La conclusion est dès le moment où vous avez utilisé une fois le webmail de protonmail la clé PGP est corrompue.

### 3.3.2 EFAIL

Malgré ces sécurités qui pourraient être mises en place à l'heure actuelle, une attaque nommée EFAIL[5] a été faite en 2018 et est toujours possible aujourd'hui(à vérifier / tester). En effet cette attaque a seulement été mitigée en évitant d'afficher les contenus HTML et les images dans boites mails de base. Car le problème vient de là principalement, des problèmes sont liés aussi aux modes de chiffrement utilisé (typiquement CBC et CFB) grâce à des "gadgets". Cette attaque permet en fait d'injecter une image dans l'HTML du message (typiquement dans les headers du mail), puis faire en sorte de récupérer le contenu du message déchiffrer dans un paramètre de l'URL.

## 3.4 Signal

L'analyse s'est faite aussi pour la messagerie instantanée à cause de sa ressemblance avec la messagerie électronique. Cela m'a finalement permis de me rendre compte que ce n'était pas des protocoles idéaux.

### 3.4.1 Fonctionnement

### 3.4.2 Problèmes d'intégrations

Le problème avec le protocole Signal quant à mes besoins niveaux mails est le *forward secrecy* qui est très fort. En effet comme vu dans le chapitre précédent il utilise une clé par message grâce au *Double Ratchet*. Cependant ce fonctionnement comporte un gros problème en rapport aux mails, en effet si l'on veut pouvoir récupérer les anciens mails reçus/envoyés cela devient vraiment compliqué. En effet, j'ai envie d'une *forward secrecy* dans mon système de mail, mais j'ai aussi besoin de récupérer les messages assez facilement si l'on connait la clé privée.

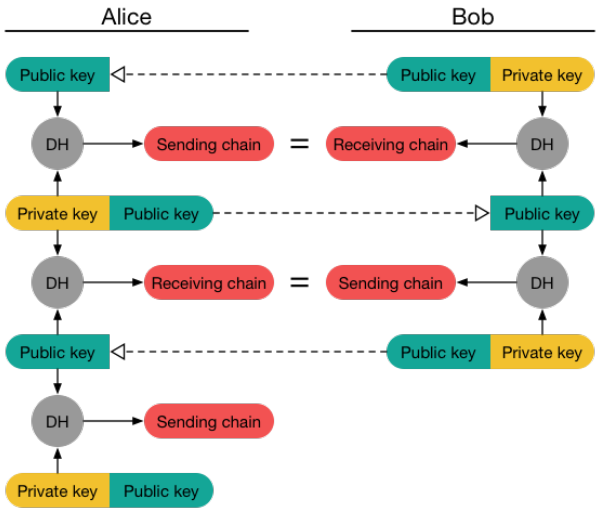


FIGURE 3.3 – Schéma fonctionnement de Signal

## 3.5 Compromis

Du coup pour passer à l'implémentation concrète d'un nouveau protocole il faut faire des compromis et aller chercher dans des primitives moins connues.
Je suis tout de même rester sur un système de clés publiques comme PGP le fait. Cependant cette primitive a une identité propre à chaque clé publique.
De plus pour avoir une *forward secrecy* l'on peut ajouter une notion de temps ou de token à l'ID pour chaque batch de messages.

### 3.5.1 Résultats des recherches

Comme mentionné avant les recherches ont beaucoup été orientées sur le protocole Signal qui a une très bonne forward secrecy, résilience et break-in recovery. Cependant le problème avec l'utilisation des mails c'est d'avoir envie de consulter tout ces mails depuis n'importe quel appareil. Ce n'est malheureusement pas le cas avec Signal à moins de conserver une *root key* quelque part qui ferait s'effondrer les caractéristiques principales du protocole.
PGP est la solution prédominante pour s'envoyer des mails chiffrés cependant il est compliquer de l'utiliser. En faisant quelques essais de mon côté je me suis heurté à beaucoup de difficultés et de problèmes avec les clés PGP.

## 3.6 Primitives

### 3.6.1 Primitives analysées

- Certificateless PKC[1]
- HIBE - Hierarchised Based encryption
- Identity based encryption

### 3.6.2 Primitive choisie

- Certificateless PKC[1]

Tout d'abord le HIBE aurait été compliqué à mettre en place aussi à cause de sa forte forward secrecy, la hiérarchie doit de plus être commune sur tout les appareils, ce qui prendrait probablement du temps à un nouvel arrivage d'appareil pour un compte donné. J'ai choisi cette primitive au final car elle proposait des propriétés très intéressantes pour ma manière d'implémenter dans les mails cela. En effet, similaire à de l'identity based avec un ID pour désigner une clé publique. Le problème avec l'identity based encryption c'est le fait que le serveur central génère la clé publique et la clé secrète de l'utilisateur, cela amène ce qu'on appelle le *key escrow* problème. C'est le fait qu'une entité connaisse à elle seule toutes les clés. Ce problème est résolu dans le certificateless en introduisant des *Partial Private Keys* permettant d'avoir une clé secrète partiellement générée par le serveur et par l'utilisateur.

## 3.7 Recherches sur la primitive

### 3.7.1 Schémas Certificateless de Chiffrement

Pour choisir parmi les nombreux schémas existants en certificateless pour le chiffrement j'ai établi un tableau comparatif des différentes manières de faire inspiré d'un livre[6]. En suivant ce tableau je me suis rendu compte que la construction de Dent-Libert-Paterson[3] était probablement la plus adaptée en vue des propriétés qu'elle présentait. Le tableau se trouve en annexe B.

### 3.7.2 Schémas Certificateless de Signature

Pour choisir parmi les nombreux schémas certificateless pour la signature j'ai établi un tableau comparatif des différentes manières de faire inspiré du livre[6]. En analysant les différentes possibilités dans ce tableau il n'y en as pas une quantité énorme qui se dégage, en effet l'on peut voir que beaucoup de schémas de signature sont cassés, mon choix s'est porté au final sur la construction de Zhang et Zhang[7] pour des signatures robustes en Certificateless. J'ai pris cette construction car elle est résistante au Malicious KGC (si le KGC a été setup avec des paramètres vulnérables) et en plus elle date de 2008 et n'a apparemment pas été cassée. Le tableau se trouve en annexe B.

# Chapitre 4

# Architecture / Design du protocole

Dans ce chapitre je vais m'intéresser à expliquer le fonctionnement de la *certificateless cryptography* et démontrer comment je l'ai utilisée afin de l'intégrer à un protocole de chiffrement de mail.

## 4.1 Fonctionnement Certificateless PKC

Je vais ici découper les différents algorithmes présent dans le certificateless public key cryptography. En passant par le chiffrement et la signature. Ces algorithmes seront accompagnés d'explications sur leur utilité. Les noms donnés aux algorithmes seront réutilisés ensuite pour les schémas afin de démontrer l'architecture du protocole mis en place. L'on peut voir des définitions spécifiques dans l'article sur lequel je me suis appuyé pour ce travail[3].

### 4.1.1 Chiffrement

- Setup (seulement une fois par le KGC).
- *Partial-Private-Key-Extract* Calcul d'une clé privée partielles lorsque qu'un client le demande pour identité donnée.
- *Set-Secret-Value* Le client ne le fait qu'une fois pour tirer sa valeur secrète.
- *Set-Private-Key* Le client combine ses clés partielles et sa clé secrète pour obtenir une clé privée afin de déchiffrer les message reçus, chiffrés avec une certaine identité.
- *Set-Public-Key* Le client ne le fait qu'une fois, il calcule sa clé publique en fonction de sa valeur secrète.
- *Encrypt* Chiffre un message avec la clé publique du destinataire et son identité.
- *Decrypt* Déchiffres un message utilisant sa clé privée et l'identité utilisée pendant le chiffrement.

### 4.1.2 Signature

Pour la signature les algorithmes sont les mêmes avec une différence dans leur conception et évidemment le *Encrypt* et *Decrypt* sont remplacé par *Sign* et *Verify*. Dans la littérature certificateless les schémas de signatures sont beaucoup plus cassés que ceux de chiffrement apparemment (voir tableau ). Il faut donc faire attention à suivre les schémas afin de vérifier que le schéma choisi ne soit pas mis à mal.

## 4.2 Design du protocole

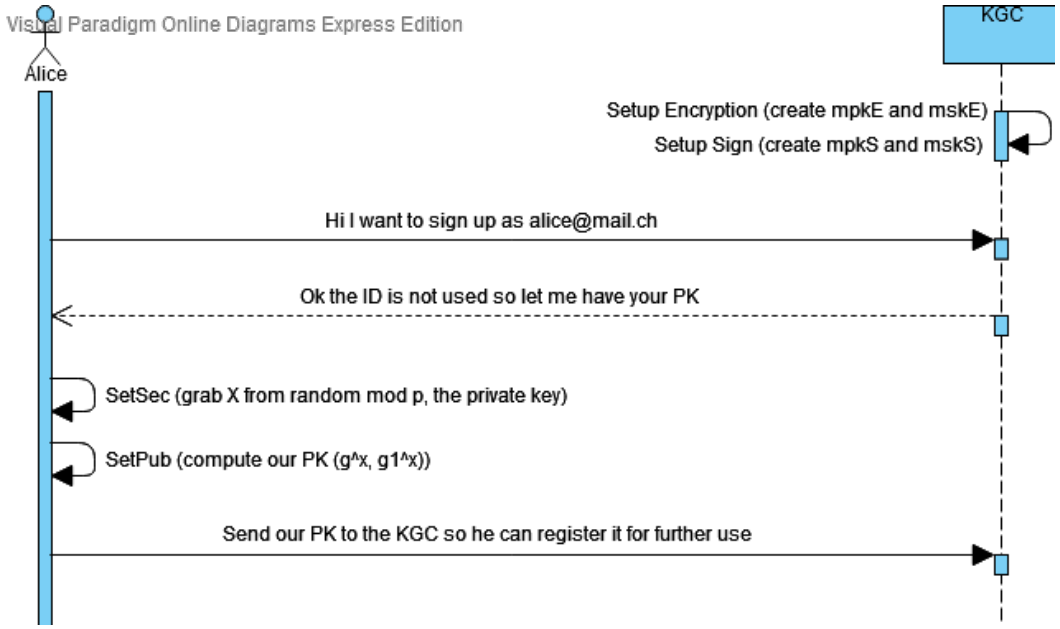


FIGURE 4.1 – Schéma de la première connexion

Dans la figure 4.1 l'on voit la première connexion d'un utilisateur mais aussi le Setup du serveur, cette étape ne se fera qu'une fois dans la vie du KGC. Mais le Setup doit tout de même être fait pour la partie de signature et la partie de chiffrement. Ensuite Alice veut s'enregistrer auprès du KGC, ainsi le KGC lui renvoi les paramètres publiques (mpkS et mpkE) si aucun utilisateur n'a déjà cet email. L'utilisateur va alors crée sa valeur secrète tirée aléatoire modulo p puis générer sa clé publique. Pour finir Alice envoi sa clé publique au KGC afin qu'il l'associe à son ID et puisses le donner aux personnes qui veulent envoyer un mail à Alice.

Dans la figure 4.2 l'on voit comment se déroulerait l'envoi d'un message à Bob :

— Tout d'abord Alice va récupérer le clé publique de Bob via son ID (aka email).

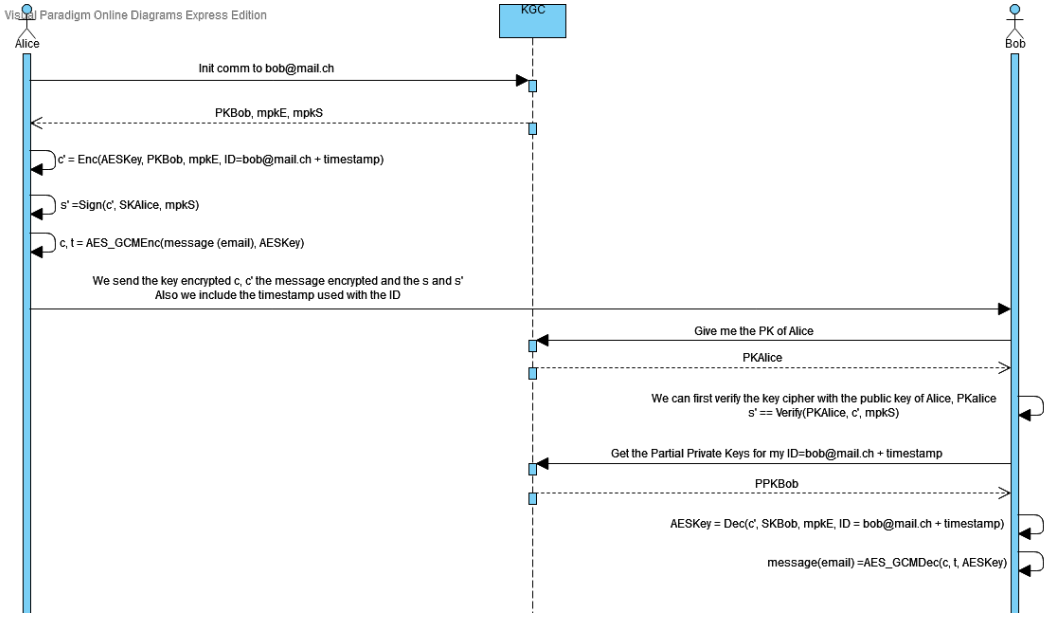


FIGURE 4.2 – Alice envoi un message à Bob

— Elle devra aussi récupérer sa clé privée partielle de signature pour créer ses clés privées afin de signer le message. Elle va le faire à l'aide de son ID et du même timestamp qu'utilisé pour la suite.

— Elle va ensuite tirer une valeur aléatoire dans Gt qui représentera sa clé AES pour la suite, elle va chiffrer cet élément à l'aide de la clé publique de Bob et de son ID complété par un timestamp. Ce timestamp sert à garder une certaine Forward Secrecy. Le cipher sera c'.

— Elle va calculer la signature du cipher donné (s' sur la figure)

— Alice utilisera un chiffrement authentifié comme AES_GCM pour chiffrer et authentifié son mail à Bob, t pour le tag et c pour le cipher.

— Finalement elle va envoyer tout ces éléments à bob (à savoir, l'ID utilisé, c, c', t, s' et l'IV utilisé pour AES_GCM).

Mais dans la figure 4.2 l'on voit aussi comment la réception du côté d'Alice se déroulerait :

— A la réception la première chose à faire et de vérifier le cipher de la clé AES. Pour cela l'on va demander la clé publique d'Alice au KGC. Puis on va vérifier ce cipher c' à l'aide de sa signature s'.

— Ensuite Bob va récupérer sa clé privée partielle via le KGC en fournissant son ID avec le timestamp envoyé par Alice. Il va ainsi pouvoir former sa clé privée.

— Avec s clé privée il va pouvoir déchiffrer c' et obtenir la clé AES pour la suite.

— Une fois que l'on a la clé AES l'on peut simplement déchiffrer à l'aide de'AES_GCM c pour obtenir le message initial.

## 4.3 Architecture globale

Dans cette figure 4.3 je présente uniquement l'architecture globale pour bien représenter les différents acteurs présents dans le protocole et ainsi avoir une vue d'ensemble pour faciliter la compréhension.

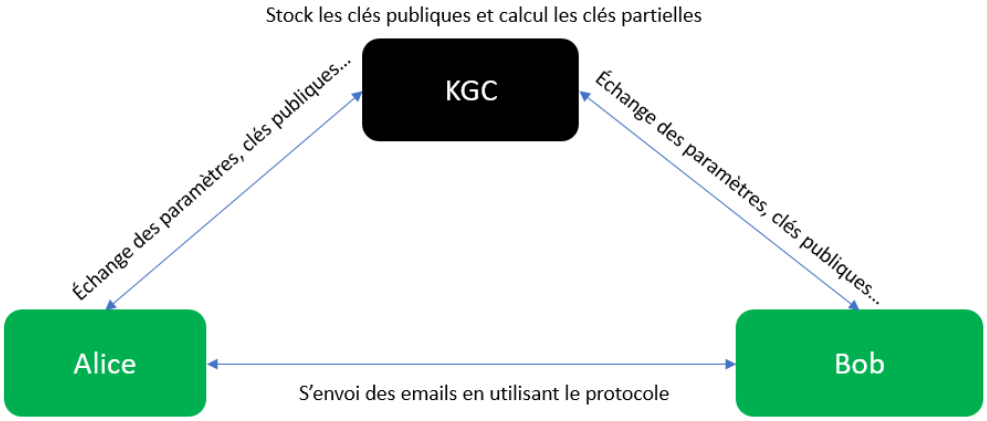


FIGURE 4.3 – Schéma très global du protocole

# Chapitre 5

# Implémentation

## 5.1 Choix d'implémentations

### 5.1.1 Langage

Au départ le choix du langage s'est porté sur sagemath (framework python) afin de mieux comprendre les différents calculs et faire un premier POC du chiffrement. Cependant l'implémentation du POC était lente et le changement d'algorithme pour les pairings était difficile. Je me suis donc orienté sur le C pour avoir de meilleures performances et pouvoir mieux gérer ma mémoire car c'est un point important dans des logiciels implémentant de la cryptographie. Pour pouvoir faire facilement des calculs sur les courbes elliptiques et les pairings en C il me fallait une librairie.

### 5.1.2 Librairie

La librairie utilisée est RELIC Toolkit[2], c'est une librairie en cours de développement qui se veut efficiente. Sa concurrence avec MIRACL m'a fait hésiter dans mon choix, mais MIRACL est plus codée en C++ avec des équivalences en C j'ai donc choisi RELIC.

### 5.1.3 Courbe utilisée

La courbe utilisée pour le POC est la BLS12-P381, en effet cette courbe est assez efficiente et compatible avec les pairings. De plus RELIC l'a dans ses options et fonctionne bien, elle a un niveau de sécurité de 128bits. Je voulais prendre une courbe avec une plus grande sécurité cependant RELIC ne l'a pas encore totalement implémenté (certains tests ne passes pas), mais la librairie étant toujours en cours de développement il faudrait suivre ça de près, le code ne changerait en effet pas.

### 5.1.4 Dérivation de la clé AES

Le but de mon schéma certificateless est de chiffrer et signer une clé AES qui permettra à mon message d'avoir un chiffrement authentifié. Pour cela il me faut dériver un élément de Gt en clé AES.
Pour cela j'ai utilisé une fonction permettant d'écrire sous forme compressée mon élément en bytes. Puis j'ai effectué un hachage avec SHA256 dessus, ainsi le résultat du hachage est une clé AES-256. La fonction de hachage doit être par conséquent cryptographiquement sûre.

### 5.1.5 Fonctions de hachage - signature

Pour le schéma de signature il nous faut plusieurs fonctions de hachage différentes (3 en fait). Pour appliquer cela j'ai utilisé la même méthode de mapping disponible ans RELIC pour mapper une char array (tableau de byte) à un point sur G2 à savoir g2_map. Pour H1, la première fonction de hachage j'ai simplement utilisé cette fonction directement, mais pour H2 et H3 j'ai ajouté un byte devant les données à mapper respectivement les bytes '01' et '02'. Ceci afin de différencier les hash générés.

## 5.2 Implémentation certificateless

Pour pouvoir implémenter ce schéma de chiffrement et signature certificateless dans un système hybride il a fallu penser à une manière d'encapsuler la clé et les données envoyées. Pour cela j'ai essayé de faire un système comparable à la figure 5.1. L'on voit les données qui seront envoyées au destinataire et en fonction de quoi elles sont crées.

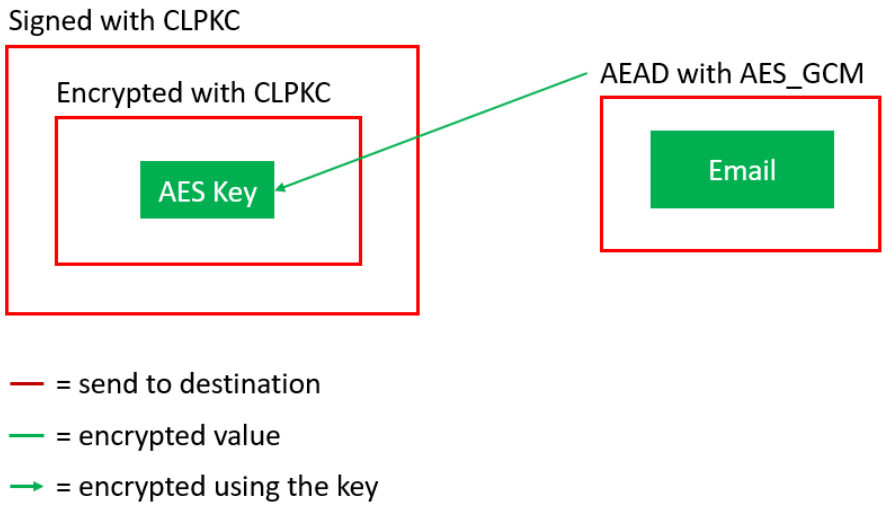


FIGURE 5.1 – Schéma encapsulation des données

### 5.2.1 Déroulement POC

Je mets ici temporairement le code du main afin de voir le déroulement global d'un é change et voir ce qui a été implémenté jusque là.

```
#include "cipherPOC.h"
#include "signaturePOC.h"
#include "sodium.h"

// Utils function for encrypting / decrypting AES_GCM
char* encrypt_message(const char* m, unsigned char* key, unsigned char* nonce, unsigned char* cipher, unsigned long long* cipher_len, const size_t* m_len){
    randombytes_buf(nonce, sizeof nonce);
    crypto_aead_aes256gcm_encrypt(cipher, cipher_len, m, *m_len,NULL,0,NULL, nonce, key);
}

void decrypt_message(unsigned char* decrypted, unsigned char* cipher, unsigned char* nonce, unsigned char* key, unsigned long long cipher_len){
    unsigned long long decrypted_len;
    if (cipher_len < crypto_aead_aes256gcm_ABYTES ||
        crypto_aead_aes256gcm_decrypt(decrypted, &decrypted_len,
                                      NULL,
                                      cipher, cipher_len,
                                      NULL,
                                      0,
                                      nonce, key) != 0) {
        /* message forged! */
        printf("Message not correctly authenticated ! Aborting decryption...\n");
    }
}

void get_key(char *aesk, gt_t originalM) {
    int sizeAESK = gt_size_bin(originalM,1);
    char aeskBin [sizeAESK];
    gt_write_bin(aeskBin, sizeAESK, originalM, 1);
    md_map_sh256(aesk, aeskBin, sizeAESK);
    printf("AES Key : ");
    for(int i=0;i < 32;i++)
        printf("%02X",(unsigned char)aesk[i]);
    printf("\n");
}

int main() {
    if(core_init() == RLC_ERR){
        printf("RELIC INIT ERROR !\n");
    }
    if(sodium_init() < 0) {
        printf("LIBSODIUM INIT ERROR !\n");
    }
    if(pc_param_set_any() == RLC_OK){
        // Server doing this once
```

```
45          pc_param_print();
46          // Setup the encrypting and signing parameters for KGC
47
48          printf("Security : %d\n", pc_param_level());
49
50          // MPK struct, Master Public Key structure to store
51          mpkStruct mpkSession;
52          mpkStructSig mpkSignature;
53
54          // Master secret key of KGC for encrypting
55          g2_t msk;
56          g2_null(msk)
57          g2_new(msk)
58
59          setup(256, &mpkSession, &msk);
60
61          // Master key of KGC for signing
62          bn_t masterSecret;
63          bn_null(masterSecret)
64          bn_new(masterSecret)
65
66          setupSig(256, &mpkSignature, &masterSecret);
67          printf("Setup successful !\n");
68          // ------------------------------------------------------------
69          // At this point, setup of KGC for encrypting and signing is successful
70
71          // Now we can go for user's private keys (encrypting and signing)
72          bn_t x;
73          setSec(&x);
74
75          bn_t xSig;
76          setSecSig(&xSig);
77          // ------------------------------------------------------------
78          // Private keys set
79
80          // Now we can go to set Public keys for both signing and encrypting
81          PK myPK;
82          setPub(x, mpkSession, &myPK);
83
84          PKSig myPKSig;
85          setPubSig(xSig, mpkSignature, &myPKSig);
86          // ------------------------------------------------------------
87          // Public keys set
88
89
90          // The other user takes ID of the destination and PK to encrypt his message
91          // With the final version we will need to append a timestamp on the ID
92          char ID[] = "mickael.bonjour@hotmail.fr";
93
94          gt_t AESK;
95          gt_null(AESK);
```

```
96          gt_new(AESK);
97          // For now we take m (AES Key) randomly from Gt
98          gt_rand(AESK);
99
100         unsigned char aesk [crypto_secretbox_KEYBYTES];
101         get_key(aesk, AESK);
102
103         char* m = "This message will be encrypted";
104         printf("Message : %s\n", m);
105         unsigned char nonceAES[crypto_aead_aes256gcm_NPUBBYTES];
106         size_t m_len = strlen(m);
107         unsigned long long cipher_len;
108         unsigned char ciphertextAES[m_len + crypto_aead_aes256gcm_ABYTES];
109         encrypt_message(m, aesk, nonceAES, ciphertextAES, &cipher_len, &m_len);
110         printf("Encrypted message : %s\n", ciphertextAES);
111
112         cipher c;
113         encrypt(AESK, myPK, ID, mpkSession, &c);
114
115         // For the signature we need our PPK
116         PPKSig myPartialKeysSig;
117
118         //The sender needs to extract (via KGC) and setPriv to get his private key and
            ↪  sign the message
119         extractSig(mpkSignature, masterSecret, ID , &myPartialKeysSig);
120
121         // Computes Secret User Keys for Signature
122         SKSig mySecretKeysSig;
123         setPrivSig(xSig, myPartialKeysSig, mpkSignature, ID, &mySecretKeysSig);
124
125         // Computes the message to sign, so the cipher struct
126         int c0size = gt_size_bin(c.c0,1);
127         int c1Size = g1_size_bin(c.c1, 1);
128         int c2Size = g2_size_bin(c.c2, 1);
129         int c3Size = g2_size_bin(c.c3, 1);
130         uint8_t mSig[c0size+c1Size+c2Size+c3Size];
131         gt_write_bin(mSig, c0size, c.c0, 1);
132         g1_write_bin(&mSig[c0size], c1Size, c.c1, 1);
133         g2_write_bin(&mSig[c0size + c1Size], c2Size, c.c2, 1);
134         g2_write_bin(&mSig[c0size + c1Size + c2Size], c3Size, c.c3, 1);
135
136         // Structure of an signature
137         signature s;
138         // We can sign using our private keys and public ones
139         sign(mSig, mySecretKeysSig, myPKSig, ID, mpkSignature, &s);
140         // ------------------------------------------------------------------
141         // Now the message is encrypted and authentified with an AES Key and the key is
            ↪  encrypted and signed using CLPKC
142         // ------------------------------------------------------------------
143
144
```

```
145
146         // We can go for decrypting and verification
147         // For this we need our Partial Private Keys with the ID used to encrypt the
            ↪  message
148
149         // We can verify directly with the public keys of the sender
150         int test = verify(s, myPKSig, mpkSignature, ID, mSig);
151         printf("\nVerification of the key (0 if correct 1 if not) : %d\n", test);
152         // if the verif is ok we can continue, otherwise we can stop here
153         if(test == 0) {
154             PPK myPartialKeys;
155             g2_null(myPartialKeys->d1)
156             g2_new(myPartialKeys->d1)
157
158             g1_null(myPartialKeys->d2)
159             g1_new(myPartialKeys->d2)
160
161             //The receiver needs to extract (via KGC) and setPriv to get his private
                ↪  key and decrypt the cipher
162             extract(mpkSession, msk, ID, &myPartialKeys);
163
164             // Computes Secret User Keys
165             SK mySecretKeys;
166             g2_null(mySecretKeys->s1)
167             g2_new(mySecretKeys->s1)
168
169             g1_null(mySecretKeys->s2)
170             g1_new(mySecretKeys->s2)
171             setPriv(x, myPartialKeys, mpkSession, ID, &mySecretKeys);
172
173             // We can decrypt now
174             gt_t decryptedMessage;
175             gt_null(decryptedMessage)
176             gt_new(decryptedMessage)
177             decrypt(c, mySecretKeys, myPK, mpkSession, ID, &decryptedMessage);
178
179             char aeskDecrypted[crypto_secretbox_KEYBYTES];
180             get_key(aeskDecrypted, decryptedMessage);
181
182             unsigned char decrypted[m_len];
183             decrypt_message(decrypted, ciphertextAES, nonceAES, aeskDecrypted,
                ↪  cipher_len);
184             printf("Decrypted message : %s\n", decrypted);
185         }
186
187         // For test purposes
188         // We change the message to see the signature not being correct again
189         unsigned char* mSigCorrupt = "The message to be signed !!";
190         printf("Message changed to simulate corruption\n");
191
192         // We can verify now with the public keys of the sender
```

```
193         test = verify(s, myPKSig, mpkSignature, ID, mSigCorrupt);
194         printf("Verification (0 if correct 1 if not) : %d\n", test);
195     }
196     core_clean();
197 }
```

# Chapitre 6

# Conclusion

# Bibliographie


[1] Sattam S. Al-Riyami and Kenneth G. Paterson. Certificateless public key cryptography. In Chi-Sung Laih, editor, *Advances in Cryptology - ASIACRYPT 2003, 9th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, November 30 - December 4, 2003, Proceedings*, volume 2894 of *Lecture Notes in Computer Science*, pages 452–473. Springer, 2003.

[2] D. F. Aranha, C. P. L. Gouvêa, T. Markmann, R. S. Wahby, and K. Liao. RELIC is an Efficient LIbrary for Cryptography. https ://github.com/relic-toolkit/relic.

[3] Alexander W. Dent, Benoît Libert, and Kenneth G. Paterson. Certificateless encryption schemes strongly secure in the standard model. In Ronald Cramer, editor, *Public Key Cryptography - PKC 2008, 11th International Workshop on Practice and Theory in Public-Key Cryptography, Barcelona, Spain, March 9-12, 2008. Proceedings*, volume 4939 of *Lecture Notes in Computer Science*, pages 344–359. Springer, 2008.

[4] Nadim Kobeissi. An analysis of the protonmail cryptographic architecture. *IACR Cryptol. ePrint Arch.*, 2018 :1121, 2018.

[5] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail : Breaking S/MIME and openpgp email encryption using exfiltration channels. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 549–566. USENIX Association, 2018.

[6] Hu Xiong, Zhen Qin, and Athanasios V. Vasilakos. *Introduction to Certificateless Cryptography*. CRC Press, Inc., USA, 2016.

[7] Lei Zhang and Futai Zhang. A new provably secure certificateless signature scheme. In *Proceedings of IEEE International Conference on Communications, ICC 2008, Beijing, China, 19-23 May 2008*, pages 1685–1689. IEEE, 2008.

# Table des figures

# Annexe A

# Outils utilisés pour la compilation

## A.1 RELIC Toolkit

Pour pouvoir faire des calculs de *Pairings* et sur des courbes elliptiques je me suis fier à RELIC Toolkit[2] qui est une librairie C permettant ce genre de calculs assez simplement. Cette librairie demande à être compilée avec une certaine courbe et certaines options (typiquement fonction de hachage et autres...). Des presets existent et c'est donc ce que j'ai utilisé pour ce POC. Cela demande donc de fournir la librairie précompilée avec les bonnes options pour l'utilisateur. L'inconvénient c'est donc que pour mettre à jour une courbe il va falloir recompiler toute la librairie et la fournir à l'utilisateur, néanmoins on n'aura pas à changer de code.

## A.2 Libsodium

Pour faire du chiffrement authentifié j'ai utilisé libsodium [1], en effet, m'étant un peu familiarisé avec la librairie il m'a semblé être le choix le plus évident en plus de fournir des méthodes de chiffrement simples à mettre en place. Nécessite d'avoir libsodium en librairie linkée.

1. https ://libsodium.gitbook.io/doc/

# Annexe B

# Fichiers

Je liste ici les fichiers annexes à mon rapport, ce qu'ils contiennent et comment les utiliser si besoin.

## B.1 Code du POC

Le code est en annexe du rapport avec le nom *POCCertificatelessCryptography*. Le *README.md* présent à la source devrait être suffisant pour compiler soit même le code. Le code est très commenté au niveau du *main.c* pour bien montrer les différentes étapes telles qu'elle pourraient arriver dans une implémentation finale.

## B.2 Tableaux comparatifs

Les tableaux comparatifs cité dans le chapitre 3 apparaissent sous forme de feuille dans un fichier excel se nommant *ComparatifsCLPKCSchemes.xlsx*. La feuille nommée CLEs contient un comparatif des schémas de chiffrement tandis que la feuille CLSs contient un comparatif des schémas de signatures.