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

Département TIC
Filière Télécommunications
Orientation Sécurité de l'information

Travail de Bachelor

Chiffrement / signature d'emails

Étudiant **Mickael Bonjour**
Enseignant responsable Prof. Alexandre Duc
Année académique 2019-2020

Yverdon-les-Bains, le 30 juillet 2020

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

Travail de Bachelor 2019-2020

Chiffrement / signature d'emails

Résumé publiable

Encore aujourd'hui, les systèmes de messagerie électronique sont largement utilisés pour s'échanger des messages ou des fichiers, parfois même confidentiels. Cependant, ces données ne sont pas toujours sécurisées correctement. Les systèmes en vigueur pour sécuriser les e-mails ont souvent subit des failles critiques permettant de récupérer les données en clair, et leur complexité d'utilisation ne fait pas l'unanimité parmi les utilisateurs.

Ce projet propose une analyse des différents systèmes de messagerie sécurisés en vigueur ainsi que des propriétés cryptographiques atteintes par ces différents protocoles. De là, une primitive cryptographique sera choisie afin d'atteindre des meilleures propriétés et une simplicité d'utilisation plus accessibles aux utilisateurs. Un *Proof Of Concept* sera fait pour évaluer si la primitive pourrait être utilisée en production. Ensuite, une comparaison sera faite pour mettre en relation les systèmes analysés et l'état de l'art déjà présent dans cette primitive.

Le *Proof Of Concept* a permis de démontrer la facilité d'utilisation induite par un tel système. En effet, cette primitive permet de lier une adresse mail à une certaine clé publique sans passer par des certificats et son infrastructure complexe qui en découle. Ceci est très avantageux et plus simple d'utilisation.

Étudiant :	Date et lieu :	Signature :
Mickael Bonjour	.....	.....
Enseignant responsable :	Date et lieu :	Signature :
Prof. Alexandre Duc	.....	.....

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.

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 30 juillet 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 30 juillet 2020

Mickael Bonjour

Cahier des charges

Résumé du problème

Les outils de chiffrement et de signature de mails 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 sécurité des emails, de proposer un nouveau protocole les implémentant et de développer un proof of concept.

Problématique

Les systèmes de mails sécurisés souffrent de manque de praticité quant à leur implémentations, de plus elles ont été prouvées vulnérables à plusieurs reprises.

Solutions existantes

Les solutions existantes sont représentées majoritairement par S/MIME et PGP. Cependant, des nouveaux protocoles émergent tel que PEP et des fournisseurs proposent des implémentations transparentes de PGP, p. ex. comme le fait Protonmail. De plus, l'on pourrait s'orienter aussi sur la messagerie instantanée qui bénéficie de protocoles sécurisés comme Signal.

Solutions possibles

Un début de solution est proposé dans ce papier à l'aide d'un nouveau système qui pourrait être mis facilement en place et qui bénéficierait de meilleures propriétés que les protocoles actuellement utilisés. L'autre solution serait de rester avec PGP et S/MIME malgré le manque d'intégration dont ils font preuves.

Cahier des charges

Voici un résumé du cahier des charges sous forme d'une liste d'objectifs à atteindre :

- Analyser les besoins d'un système de messagerie 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 faite et les choix faits
    - Proof Of Concept
    - Les problèmes connus
2. Le code du *Proof Of Concept* fait, expliqué à l'aide de commentaires.

Table des matières

Chapitre 1

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 répandue, le *Certificateless Public Key Cryptography*. Ma démarche dans ce travail de bachelor est de voir si des solutions s'offrent à nous, en considérant ce qui se fait sur le marché actuellement. En essayant d'améliorer les solutions actuelles proposées qui peuvent assez souvent souffrir d'un manque de sécurité ou 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, ce qui existe et analyser pourquoi il faudrait trouver de nouvelles solutions. Puis une présentation de la primitive cryptographique utilisée pour ma proposition dans ce travail ainsi que la raison qui amène à ce choix. Enfin, une présentation de l'architecture du protocole imaginé, une implémentation proposée en *Proof Of Concept* ainsi que les choix importants qui ont été faits en rapport à cette implémentation. En commentant évidemment les problèmes connus et les améliorations qui seraient possibles et envisageables si l'on voulait faire de ce *Proof Of Concept* une réalité.

Chapitre 2

Analyse - État de l'art

Ce chapitre va s'intéresser aux différentes propositions d'implémentations de système de messagerie sécurisée afin de voir où en est l'état de l'art. Pour les systèmes les plus connus tels que PGP et S/MIME sont analysés mais aussi les implémentations de ces protocoles dans des clients mails tel que Protonmail ou Tutanota. L'analyse s'élargit aussi à des protocoles plus orientés vers la messagerie instantanée comme Signal.

2.1 Système de messagerie

Dans un système de messagerie les besoins principaux sont surtout de pouvoir consulter sa boîte mail à tout moment avec les anciens et nouveaux mails reçus. De plus, il est préférable de pouvoir envoyer des mails aussi. Ces envois peuvent avoir plusieurs propriétés et fonctionnalités. L'on pourrait envoyer un mail à de multiples destinataires et même sans que les uns et les autres sachent exactement à qui est envoyé le mail exactement (copies cachées). Un client mail permet aussi d'envoyer des pièces jointes, celles-ci seront encodées au sein du message et envoyées avec. Un mail a aussi un état pour savoir s'il a déjà été lu ou non. Parmi l'utilisation simple d'un système de messagerie il y aussi le fait que l'on veut pouvoir voir ses mails de n'importe quel appareil à n'importe quel moment. Des possibilités de suivis sont généralement possible, afin de savoir si le destinataire a reçu / lu le mail. Il est aussi généralement possible de donner un importance à certains mails.

2.1.1 Détails techniques

Afin d'établir les futures notations utilisées ci-après et montrer le fonctionnement global d'un système de messagerie électronique la Figure 2.1 sera la base de l'explication. Dans cette Figure l'on peut voir que 3 protocoles différents sont utilisés pour la gestion des emails ;

SMTP, IMAP et POP3. Ces 3 protocoles sont utilisés par différents acteurs, le MUA (Mail User Agent), le MTA (Mail Transfer Agent) et le MDA (Mail Delivery Agent).

Le MUA est en fait un client mail qui va s'occuper d'envoyer des mails ou de les recevoir (rechercher sur le MDA).

Les MTAs sont les serveurs mails responsables du bon acheminement des mails. Ainsi les 3 protocoles énoncés plus hauts sont soit pour l'envoi et la transmission (SMTP) soit pour la récupération des messages (IMAP et POP3).

Lors de l'envoi, un MUA va simplement renseigner les destinataires du message ainsi que sa source, son sujet et son message puis le serveur va transmettre ces informations au MTA du domaine de destination qui s'occupera de le transmettre au MDA (souvent les 2 à la fois), celui-ci stocke les mails en attendant qu'un MUA fasse une demande via POP3 ou IMAP. IMAP est souvent préféré car les mails restent ainsi sur le serveur mail et est donc consultable depuis un autre appareil utilisant aussi IMAP. POP3 va plutôt télécharger les mails et les enlever du serveur, ils ne seront donc plus disponibles par le biais d'un autre appareil.

Les MTAs sont des serveurs de transmission de données, transmises en clair jusqu'à l'introduction d'ESMTP et de la directive STARTTLS, permettant un niveau basique de sécurité entre 2 MTAS pour le transfert de mails. Cependant, si un MTAs est mal configuré et ne permet pas cette directive, le mail transitera en clair. C'est dans ce contexte là ainsi que celui du stockage des mails en clair par le MDA que des solutions de chiffrement de mail en E2E (End to End - Chiffrement de bout-en-bout) ont vu le jour.

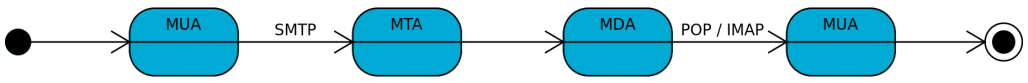

FIGURE 2.1 – Le fonctionnement d'un système de mail [7]

2.2 Protocoles existants

Dans cette section une analyse des différents protocoles existants pour sécuriser la messagerie électronique est proposée. Puis une recherche sur leur implémentation au sein de certains clients mails est faite. De plus, une présentation d'un protocole de messagerie instantanée fortement sécurisé est faite afin de voir s'il est possible d'implémenter cela dans un système de mails.

2.2.1 PGP

Fonctionnement. PGP (Pretty Good Privacy ou Assez bon niveau de confidentialité) est un moyen de chiffrer des données (mails, fichiers, ...). C'est une méthode de chiffrement hybride (utilise le chiffrement symétrique et asymétrique) qui fonctionne comme montré sur la Figure 2.2. Comme on peut le voir, on tire une clé symétrique aléatoirement qui permettra de chiffrer notre mail avec un chiffrement symétrique comme AES. Ensuite, l'on va chiffrer cette clé symétrique à l'aide d'un chiffrement asymétrique, en utilisant la clé publique du destinataire.

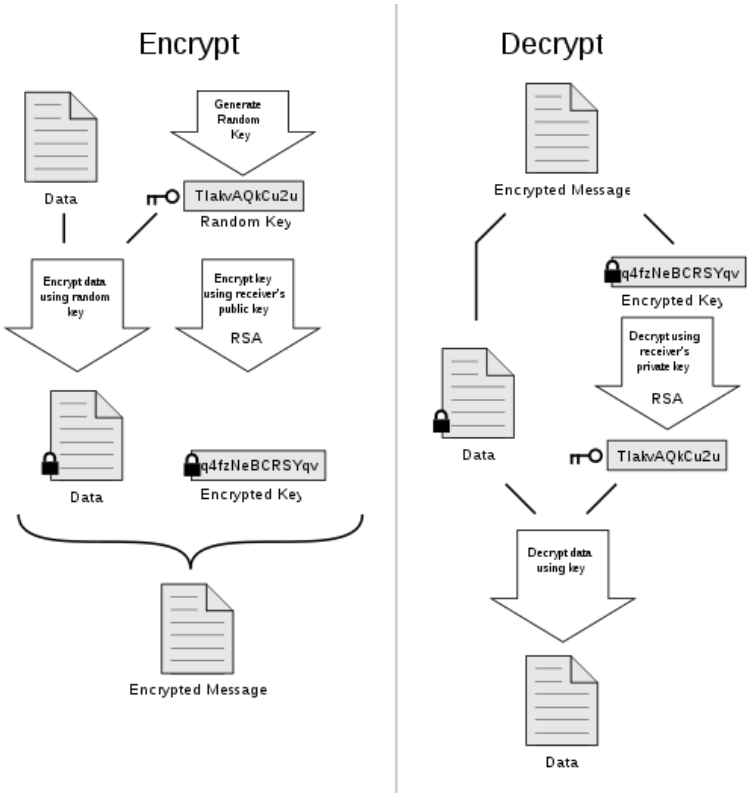

FIGURE 2.2 – Le fonctionnement global de PGP [8]

Ce fonctionnement hybride est expliqué par la lenteur d'un chiffrement asymétrique sur un certain nombre de données. Ainsi en chiffrant uniquement la clé symétrique qui a servi à chiffrer le message, le déchiffrement est bien plus rapidement effectué. Typiquement, avec un chiffrement symétrique tel qu'AES ayant des instructions dédiées dans certains processeurs. Contrairement à des chiffrements asymétriques qui sont plus contraignants. Mais il est nécessaire de passer par cette phase asymétrique, on a en effet besoin d'un secret partagé dès le début de la communication si cette méthode n'est pas utilisée.

Pour ce qui est des primitives cryptographiques proposées dans la RFC4880 [6], elle sont listées ci-après dans la table 2.1.

Symetric	Asymetric	Hash	Compression
IDEA	RSA	MD5	ZIP
TripleDES	ElGamal	SHA-1	ZLIB
CAST5	DSA	RIPE-MD160	BZip2
Blowfish	ECDSA	SHA256	
AES-128	Diffie-Hellman	SHA384	
AES-192		SHA512	
AES-256		SHA224	
Twofish-256			
IDEA			

TABLE 2.1 – Table des algorithmes utilisés par PGP

Attention MD5 a été annoncé déprécié. Il faut savoir que pour chacune de ces catégories il y a 10 éléments réservés pour des primitives privées/expérimentales.

Lorsqu'un mail est chiffré et signé avec PGP, il est d'abord hasher puis ce hash est signé avec la clé privée de l'utilisateur afin de faire une signature digitale. Le message et la signature sera alors chiffrée à l'aide la clé symétrique.

L'organisation d'un message PGP se fait via des "paquets" d'informations encodés en base64. La RFC définit bien ces types de paquets, leur fonctionnement et les différents codes associés. Sur le blog de Conrad Irwin [1] l'on peut entrer un message PGP et ainsi voir l'organisation d'un message, ou entrer des clés publiques ou une clé privée (déconseillé tout de même). Ainsi, un exemple de mail envoyé à plusieurs destinataires est représenté dans la Figure 2.3. Cela démontre comment fonctionnes PGP, en effet le message étant chiffré avec une clé symétrique, le chiffré sera le même pour tout le monde. Mais afin que tous les destinataires puisses avoir la clé symétrique, elle est chiffrée à l'aide des clés publiques des différents destinataires (et de la source, pour pouvoir la déchiffrer à l'avenir et ne pas conserver le mail en clair dans la boite d'envoi). À noter qu'avec l'option de *blind copy* (option permettant d'envoyer à un utilisateur un mail sans qu'il sache qu'il a aussi été envoyé à un autre utilisateur), PGP *leak* les receveurs des mails avec leur KeyID qui sera présent dans le message PGP [5].

PGP utilise donc un système de clés publiques afin d'envoyer des clés symétriques. Comment obtient-on une clé publique pour envoyer notre message ?

Selon certaines utilisations les clés peuvent être obtenues via un serveur de clés, mais cela implique un parti tier auquel il faut faire confiance, ainsi ce qui peut être fait c'est aussi d'avoir son propre serveur de clés. Mais pour être sûr que tel clé appartienne bien à tel utilisateur, PGP a introduit dès ses débuts un système décentralisé de confiance appelé le

1. `https://cirw.in/gpg-decoder/`

FIGURE 2.3 – Exemple décodage d'un message PGP

"Web of Trust". Ainsi, les utilisateurs signent entre eux leurs certificats d'identité, chacun des utilisateurs aura des clés de confiance et ainsi de suite. Cela permet d'avoir une toile de confiance entres les utilisateurs.

Cependant, ce système est difficile à utiliser, il est nécessaire de faire attention à quelles clés les utilisateurs signent et approuvent. Surtout pour les nouveaux utilisateurs qui ne peuvent faire vérifier leurs clés publiques facilement et ne seront donc pas vérifié, c'est pour cela que des "fêtes" de signature de clés sont organisées afin de se rencontrer personnellement et vérifier que tel utilisateur est bien telle personne. Ces évènements sont d'ailleurs possible grâce aux Fingerprint des clés publiques. Ce sont des chaînes hexadécimales de 20 bytes permettant d'identifier une clé publique plus facilement.

Pour expliquer plus précisément le *Web Of Trust* prenant un exemple : Alice a une clé publique 0xAAAAAAAAAAAA et elle signe la clé de Bob 0xBBBBBBBBBBB car elle le connaît. De la même manière Bob signe la clé publique de Carol 0xCCCCCCCCCCCC. Grâce à cela, Alice peut être sûre que cette clé appartient bien à Carol car il y a un chemin de signature valide entre sa clé publique et celle de Carol.

Ces certificats d'identité sont signés à l'aide de SHA-1, cependant il a été prouvé que des attaques sur cet algorithme au niveau des collisions peuvent être faites [15]. Heureusement, le standard a été modifié dans les nouvelles version de GnuPG pour signer les identités des utilisateurs.

Dans la dernière spécification de PGP des moyens de créer des autorités de certification ont été ajouté afin d'avoir un système décentralisé de *trust signatures*. Ceux-ci ont plusieurs niveaux de confiance définissant s'ils pourront ou non délivrer des signatures. Ceci permet de se baser sur un système qui ressemble à un PKI mais avec une flexibilité sur les CAs (utilisateurs) auxquels on fait confiance ou non.

Propriétés cryptographiques. PGP a surtout été crée pour fournir du **chiffrement de bout-en-bout** afin de résoudre les problèmes de transmissions en clair entre les MTAs et le stockage des mails en clair dans les MDAs. Ceci même lors d'une récupération de mails sur un serveur, les mails seraient chiffrés. Ensuite, PGP propose de signer ou non ses mails ce qui amène donc de la **répudiation** (si non-signé, le mail ne pourra pas être utilisé pour prouver qu'il a été envoyé par telle personne) et **non-répudiation** (mail signé, ainsi l'on peut prouver que l'envoyeur a bien envoyé le mail).

Le problème qui est souvent reproché à PGP c'est qu'il n'implémente 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 notre clé privée, il ne pourra pas déchiffrer les anciens messages chiffrés avant l'instant t.

Utilisation. Lors des tests, l'utilisation la plus simple possible a été utilisée 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.

L'installation d'un Add-On sur le logiciel de messagerie (Thunderbird dans ce cas) s'appelant Enigmail, a été nécessaire pour évaluer les cas d'utilisation. Ensuite, Enigmail a généré les clés PGP (de manière totalement automatisée à l'aide GnuPG). Puis l'envoi d'un mail se fait simplement via des icônes et des options dans le client mail. Cependant, le fonctionnement est très opaque et on ne sait pas ce qu'Enigmail fait réellement derrière les décors. L'utilisateur doit encore choisir s'il veut chiffrer ses mails ou non. De plus, Enigmail utilise Autocrypt, un système permettant d'envoyer la clé publique directement dans le mail. Cette technologie ne fait pas encore l'unanimité et est donc à utiliser avec précaution.

Pour les clés, Enigmail les envoie sur des serveurs de clés par défaut. Il va aussi interroger ces serveurs si une clé publique pour un destinataire est disponible dessus lors du chiffrement d'un message. Ces serveurs sont les suivants : keys.opengpg.org (vks), hkps.pool.sks-keyservers.net (hkps), pgp.mit.edu (hkps). Les protocoles vks (Verifying Keyservers) et hkps (HTTP keyserver protocol over TLS) sont des interfaces avec des serveurs de clés afin d'enregistrer des nouvelles clés ou trouver une clé sur le serveur par différents moyens (email, key-id, fingerprint). Les clés générées ont été générées avec les algorithmes EdDSA 4096bits par défaut, et dans les paramètres avancés il est possible de choisir entre cet algorithme et RSA (algorithme non spécifié).

2.2.2 S/MIME

Fonctionnement. S/MIME (Secure/Multipurpose Internet Mail Extensions) se base sur un système de PKI (Public Key Infrastructure) pour chiffrer et signer les mails. Dans une telle infrastructure les CAs (Certificate Authorities) garantissent que les certificats décrivent

bien l'entité.

MIME est un standard qui étend le format de mail standard afin de pouvoir transmettre des données plus complexes que le format ASCII. En effet, cela permet de transmettre d'autres sets de caractères et la possibilité de transmettre des fichiers joints audio, vidéo, images, programmes aux emails. MIME permet de séparer le message en partie, c'est d'ailleurs là dessus que l'attaque EFAIL (c.f. Section 2.4.2) s'appuie. Ces parties ont différent format de données et le message MIME est défini selon un type parmi : mixed, digest, alternative, related, report, signed, encrypted, form-data, mixed-replace, byteranges. Ces types de contenu sont là pour définir de quoi est composé le message et comment le décoder, fonctionnant via une notion de frontières entres les différentes parties.

Dans ce contexte S/MIME vient ajouter un type MIME *application/pkcs7-mime*, un format de données, qui fabrique une enveloppe chiffrée d'une entité MIME. Pour les signatures S/MIME utilisera plus les signatures détachées et le type *multipart/signed* pour cela ou encore *application/x-pkcs7-signature*. Anciennement, c'était effectivement le format de message PKCS#7 utilisé pour le format des messages chiffrés. Cependant, à l'heure actuelle c'est la spécification de CMS (Cryptographic Message Syntax) qui est utilisé dans ces types S/MIME.

Afin de commencer à signer des messages avec S/MIME et pouvoir recevoir des messages chiffrés il faut un certificat. Ce certificat peut être obtenu soit par une autorité de certification interne ou une autorité externe. Ces certificats peuvent être de classe 1 (vérification que le propriétaire du certificat peut recevoir des messages envoyés au "From :" de ses messages) ou de classe 2 avec plus de précisions sur le propriétaire du certificat.

Pour signer un message, S/MIME va utiliser la clé privée lié au certificat de la source, ainsi le message va être signé avec la primitive adéquate par rapport aux informations du certificat. Puis la signature sera envoyée avec le certificat, afin que le destinataire puisses vérifier la signature à l'aide du certificat, et vérifier le certificat avec un CA.

Comme le présente la RFC8551 [21], le chiffrement effectué par S/MIME s'approche de celui fait par PGP, en effet S/MIME va chiffrer la clé de chiffrement symétrique (Content Encryption Key dans CMS) une fois par destinataire en utilisant la clé publique authentifiée par leur certificat respectif et aussi une fois pour la source du message (afin de pouvoir relire le message envoyé à l'avenir). Les algorithmes qui doivent être pris en charge dans CMS sont les suivants selon la RFC8551 dans la table 2.2.

Table 2.2 – Table des algorithmes utilisés par S/MIME

Symetric	Asymetric	Hash	Signature
AES-128 CBC	ECDH sur P-256	SHA-256	ECDSA (courbe P-256 et SHA-256)
ChaCha20-Poly1305	ECDH avec HKDF-256	SHA-512	EdDSA (courbe 25519 avec PureEdDSA mode)
AES128-GCM	RSA		RSA PKCS #1 v1.5 avec SHA-256
AES256-GCM	RSAES-OAEP		RSASSA-PSS avec SHA-256

Propriétés cryptographiques. S/MIME est aussi créé pour établir un **chiffrement bout-en-bout** et ainsi éviter de révéler trop d'informations lors d'une brèche dans un MDA par exemple. **Authentification** et **intégrité** du message en utilisant la signature digitale, en plus de **non-répudiation** grâce à celle-ci. Le fait d'avoir le message chiffré de bout en bout permet un certain respect de la vie privée, on ne peut voir les données envoyées.

Utilisation. Pour utiliser S/MIME et avoir des exemples et références de certificats, deux fournisseurs **gratuits** ont été utilisés pour évaluer les certificats S/MIME.

Tout d'abord, un plugin firefox qui permet d'avoir des certificats pour GMail et envoyer des mails signés et chiffrés à l'aide de S/MIME a été testé. La clé privée est générée par l'extension localement et n'est pas sauvegardée dans un cloud, il est possible de la sécurisé à l'aide d'une *passphrase*. Cependant, les certificats ne sont pas vérifié correctement comme on peut le voir à la réception dans MeSince (le prochain programme testé) à la Figure 2.5. Le certificat utilise RSA (algorithme exact non spécifié) avec SHA256 pour la signature des messages.

FIGURE 2.4 – Erreur de vérification pour Fossa

Pour tester S/MIME, un compte MeSince [2] a été utilisé. En effet, ce service permet d'utiliser S/MIME afin de chiffrer et signer ses mails, et ils fournissent les certificats, seulement il ne fonctionnait pas avec Gmail. De plus, le service fourni n'a pas fonctionné pour se connecter et

2. `https://www.mesince.com`

récupérer son certificat, ainsi le certificat a été généré et peut être utilisé par leur application mobile pour envoyer des mails signés et chiffrés, cependant il n'est pas accessible et ne peux donc pas être directement analysé. Sur l'application mobile, il n'y a pas moyen de vérifier la clé générée ni même de l'exporter.

Attention cependant, le même problème avec Fossa arrive comme le montre la Figure 2.5. Par contre l'avertissement n'est pas très voyant au sein de Gmail (qui intègre pourtant le support S/MIME dans G Suite). Les certificats utilisent RSA (algorithme exact non spécifié) avec SHA256 pour la signature des messages.

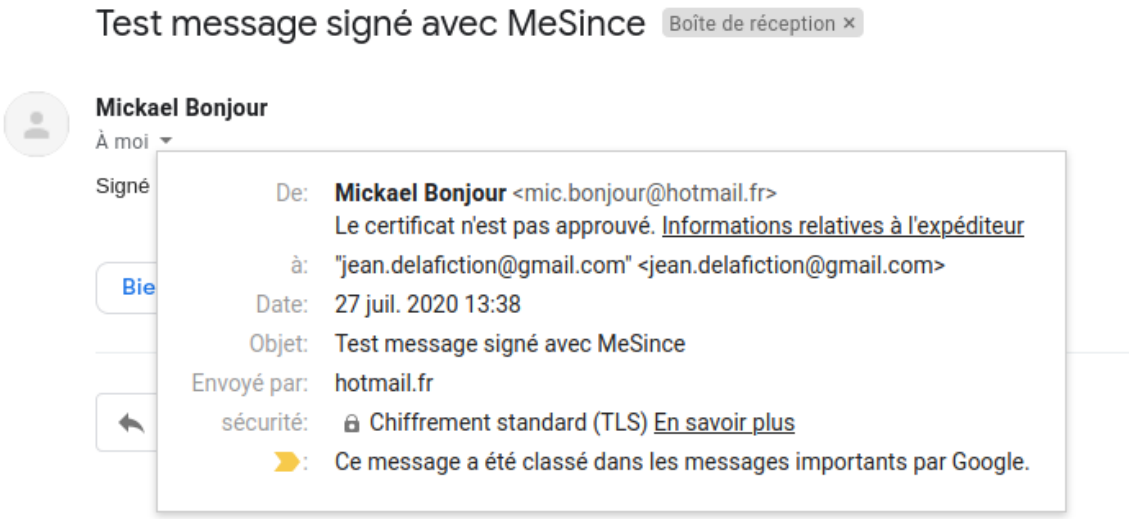

FIGURE 2.5 – Erreur de vérification pour MeSince

MeSince informe que la clé privée est automatiquement sauvegardée dans leur cloud sécurisé, ce qui n'est pas forcément une bonne nouvelle (fuite de leur cloud, cloud dont ils ont accès). De plus, la clé est générée automatiquement, donc aucune vérification de la part de l'utilisateur ne peut être faite.

Cependant, ces deux tests effectués ne représentent pas vraiment une utilisation réelle de S/MIME, en effet, le meilleur moyen de tester S/MIME aurait été d'avoir un nom de domaine à soi et de générer des certificats S/MIME pour une adresse privée. Ensuite, il faudra importer manuellement ses certificats et ses clés dans le client mail utilisé.

2.3 Implémentations existantes

Dans cette section, certaines implémentations des protocoles discutés dans la section précédente sont analysés, particulièrement ceux implémentant PGP.

2.3.1 Protonmail

Revendications. Protonmail revendique beaucoup de propriétés cryptographiques, telles que le zero-access encryption (lors de la réception d'un message externe chiffré ou non Protonmail le chiffrera avec la clé publique de l'utilisateur pour ne plus y avoir accès dans le futur). Et le chiffrement de bout-en-bout pour les messages sécurisés, cela même avec leur fonctionnalité de chiffrement vers l'extérieur utilisant AES256-GCM. Mettre une date d'expiration sur un mail est aussi possible, afin que le destinataire ne puisse le lire que dans un temps imparti.

Pour l'authentification Protonmail utilise une manière fortement sécurisée (SRP) pour ne pas avoir d'informations directes sur le mot de passe de l'utilisateur.

Protonmail chiffre automatiquement les mails d'un utilisateur Protonmail à un autre via PGP.

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. On peut utiliser Protonmail pour utiliser PGP si on a la clé de notre destinataire par exemple. Et 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 auparavant de manière sécurisée entre les deux parties. Un exemple de mail utilisant cette fonctionnalité est présenté dans la Figure 2.6.

Open Source. Leur code est open-source afin d'avoir une validation externe, de plus ils ont un programme de Bug Bounty pour les chercheurs. Ils ont largement contribué au projet d'intégration openpgp, en javascript et en Go.

Utilisation. Lors de l'utilisation de protonmail, l'analyse des clés PGP générées a montré qu'elles utilisent RSA. Pour envoyer ou recevoir des messages via PGP, Protonmail le fait de manière très transparente. Apparemment en utilisant les mêmes serveurs de clés cités plus haut pour Enigmail. En effet, les deux comptes mails de tests ont pu s'envoyer des messages sans s'envoyer leurs clés publiques au préalable.

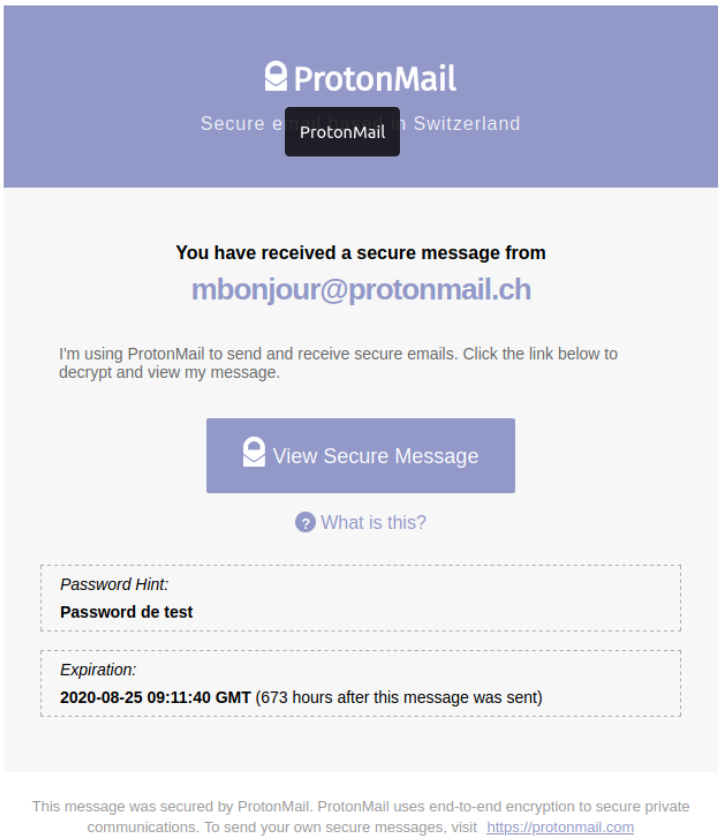

FIGURE 2.6 – Présentation d'un email chiffré Protonmail

Un mot de passe supplémentaire est utilisé pour chiffrer notre boite mail, afin de protéger les mails si une brèche survenait au niveau du stockage Protonmail. De plus, la technologie utilisée pour l'authentification et les mots de passe est SRP (Secure Remote Password). Permettant de ne pas avoir de *hashs* de mots de passe stockés chez Protonmail.

Par contre, la génération de la clé privée à la création du compte est assez obscure mais de l'information est disponible sur le site de Protonmail [3].

Critiques. Protonmail est très critiqué de manière générale sur les réseaux sociaux. De plus une réponse [4] au chercheur Nadim Kobeissi [14] avait été formulée par Protonmail suite à son papier sur l'insécurité de leur webmail. Cependant, ils sont surtout critiqués sur les réseaux sociaux par des chercheurs en sécurité [5] pour leur publicité mensongère, en effet, leur

3. `https://protonmail.com/support/knowledge-base/how-is-the-private-key-stored/`
4. `https://protonmail.com/blog/cryptographic-architecture-response/`
5. `https://twitter.com/FiloSottile/status/1277068367728435202`

page d'accueil indiquait une sécurité pour **tous** les mails sortants. Ce qui n'est pas le cas, uniquement les mails intentionnellement chiffrés à l'aide de PGP ou chiffrés vers l'extérieur sont effectivement chiffrés. L'annonce ne représentait donc pas la réalité, ce qui peut être mal interprété par les utilisateurs. Après vérification, la page d'accueil a bien été modifiée pour ne plus inclure la mention de chiffrement automatique pour tous les emails. Le commentaire fait référence aussi à d'autres points :

- Ingère les emails en plaintext - par rapport à leur politique de zero-access, ils ont au moins une fois accès au plaintext d'un email non chiffré.
- Pas de réelles solutions pour du chiffrement vers l'extérieur - La solution amenant l'utilisateur distant à se connecter à leur service pour déchiffrer un message n'est pas une solution viable.
- Utilise de la cryptographie dans le navigateur non validée - Modules cryptographiques servis par les serveurs de Protonmail, on revient ici aux revendications de ce papier [14]. Malgré cela une solution existe désormais, utiliser un bridge disponible sur github [6].
- Agit comme un serveur de clés, sans transparence de clés - Suite à la mise en place d'un serveur de clés pour chercher les clés PGP d'utilisateur Protonmail pas de transparence sur ce serveur n'a été communiqué.

2.3.2 Tutanota

Fonctionnement. Tutanota est un client mail sécurisé, il permet de chiffrer les mails vers l'extérieur. Cela en utilisant AES256-CBC (selon analyse du code source [7]) mais aussi RSA apparemment, sans savoir le contexte exact, peut-être pour vérifier le client desktop.

Utilisation. L'utilisation de Tutanota est facile, en effet Tutanota utilise le même principe que Protonmail pour le chiffrement vers l'extérieur. En chiffrant le mail et en le gardant sur ses serveurs puis en envoyant un mail au destinataire comme celui présenté dans la Figure 2.7.

2.4 Attaques existantes

Présentation des attaques faites sur les différents systèmes présentés jusqu'ici ainsi que leur fonctionnement global.

6. `https://github.com/ProtonMail/proton-bridge`
7. `https://github.com/tutao/tutanota/blob/master/src/api/worker/crypto/Aes.js`

FIGURE 2.7 – Présentation d'un email chiffré Tutanota

2.4.1 Défauts webmails

Selon un chercheur, l'infrastructure de Protonmail aurait des failles via son webmail [14]. Mais son papier est en fait plus général et parle des webmails en règle générale.

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'ils 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 que dès le moment où vous avez utilisé une fois le webmail de Protonmail la clé PGP pourrait être corrompue ou connue de Protonmail.

2.4.2 EFAIL

Malgré ces sécurités qui pourraient être mises en place à l'heure actuelle, une attaque nommée EFAIL [19] a été faite en 2018. En effet, cette attaque a seulement été mitigée en évitant d'afficher les contenus HTML et les images dans les boîtes mails de base. Car le problème vient principalement de là, des problèmes sont liés aussi aux modes de chiffrement utilisé (typiquement CBC et CFB pour S/MIME et PGP respectivement) grâce à des "gadgets".

Cette attaque permet en fait d'injecter une image dans l'HTML du message, puis faire en

sorte de récupérer le contenu du message déchiffré dans l'URL. Ceci est possible grâce au multi parties de MIME et en les abusant afin d'entourer le message chiffré d'une balise *img* et de mettre en source le message qui sera déchiffré selon les règles de MIME. Ainsi l'attaquant aura le message déchiffré dans l'URL qu'il peut contrôler.

Cette attaque exploite en fait une erreur par rapport à la gestion des messages utilisants HTML et multipart/mixed. Il faudrait en effet vérifier dans ces cas que le document HTML est un document entier. Ainsi que de ne pas traiter le contenu chiffré de la même origine que du contenu non protégé. La spécification [21] S/MIME a aussi mise à jour pour implémenter des chiffrements authentifiés et remplacer CBC qui permettait des attaques par gadgets.

2.4.3 SHA-1 Shambles

Récemment [15] une attaque sur SHA-1 avec un préfixe choisi a été démontrée en pratique [8]. Cette attaque permet de se faire passer pour un autre utilisateur grâce à une attaque par préfixe choisi sur SHA-1. SHA-1 était en effet l'algorithme de hachage par défaut pour signer les clés d'autres utilisateurs dans le *Web Of Trust* de PGP.

L'attaque fonctionne ainsi pour un l'attaquant, Bob, voulant se faire passer pour Alice :

- Il va générer deux clés PGP à l'aide de l'attaque par préfixe choisi, une clé A contenant une grande clé publique mais qui sera facilement factorisable avec l'identité d'Alice puis une clé B contenant une petite clé publique ainsi que l'identité de Bob au sein d'un objet JPEG, et après la fin de ce format l'identité d'Alice "cachée". Le fait de générer ces deux clés avec l'attaque permet d'avoir un hash commun pour les deux clés publiques $H(A) = H(B)$.
- Bob va donner sa clé publique à d'autre utilisateurs et demander de la signer (clé tout à fait correcte à première vue).
- Il va pouvoir utiliser les signatures effectuées sur sa clé sur la clé A générée auparavant.
- Cela a pour effet d'avoir une clé publique contenant l'identité d'Alice qui est signée pas des utilisateurs du *Web Of Trust*. Les autres utilisateurs faisant confiance à ces utilisateurs feront donc confiance à la clé d'Alice alors que c'est une clé forgée.

2.5 Signal

L'analyse s'est faite aussi pour la messagerie instantanée à cause de sa ressemblance avec la messagerie électronique.

8. `https://sha-mbles.github.io/`

2.5.1 Fonctionnement

Le fonctionnement de Signal est complexe à expliquer, ainsi la Figure 2.8 sera plus parlante pour expliquer le Diffie-Hellman Ratchet. Ce premier Ratchet permet d'utiliser Diffie-Hellman afin d'avoir une première sortie synchronisée entre l'envoi et la réception d'un utilisateur. Au début de la discussion, Bob va envoyer sa clé publique et Alice va pouvoir commencer son premier Ratchet en générant elle aussi une paire de clés Diffie-Hellman. On verra dans la Figure 2.9 ce qu'il se passe ensuite pour le chiffrement des messages mais une fois cette opération finie, Alice envoie sa clé publique précédemment générée à Bob pour qu'il puisse générer la même clé partagée à l'aide de Diffie-Hellman et ainsi déchiffrer le contenu en s'aidant du Deuxième Ratchet.

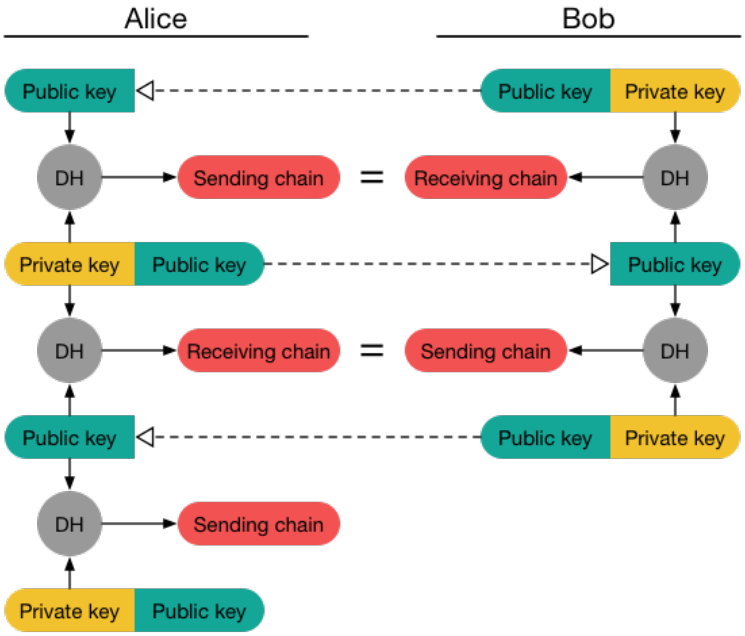

FIGURE 2.8 – Fonctionnement du DH Ratchet [18]

Par contre, l'image est une simplification, la clé DH ne va pas être utilisée telle quelle, elle sera insérée dans une KDF avec une Root Key (partagée à l'aide de X3DH) qui sortira la prochaine Root Key et la Receiving/Sending Chain Key qui sera donnée au deuxième Ratchet.

Le deuxième Ratchet (voir Figure 2.9) va générer une clé pour chaque message/batch de messages afin d'envoyer un message chiffré avec le moins d'utilisation de la même clé possible.

Ainsi la forward secrecy est plus forte.Dans cet exemple on voit qu'Alice a envoyé un message chiffré à l'aide d'A1 puis qu'elle a reçu un nouveau DH Ratchet de Bob qui lui a permis de générer sa prochaine Receiving Chain Key et de déchiffrer les messages que Bob a envoyés (B1).

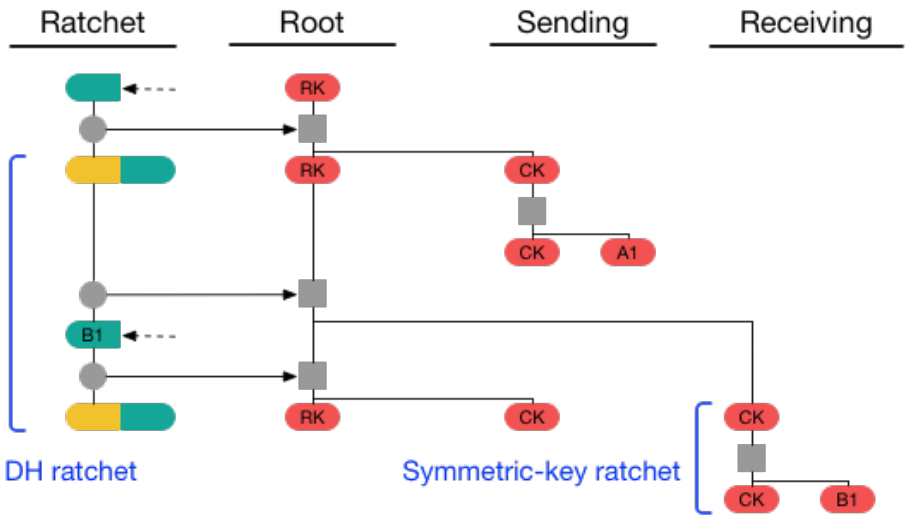

FIGURE 2.9 – Fonctionnement du deuxième Ratchet [18]

2.5.2 Propriétés cryptographiques

Signal a de nombreux avantages concernant les propriétés cryptographiques qu'il promet. En effet le *Double Ratchet* permet de la **Break-in recovery**, a une **Perfect Forward Secrecy**, **End-to-End encryption**, **Non-Répudiation**.

2.5.3 Problèmes d'intégrations

Le problème avec le protocole Signal quant aux besoins niveaux mails est la *perfect forward secrecy* qui est très forte. En effet, comme vu dans la Section précédente, 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. La *forward secrecy* est une propriété utile dans un système de mails, mais il faut pouvoir aussi récupérer les messages facilement si l'on connaît la clé

privée. Alors qu'avec ce système de Double Ratchet il nous faudrait enregistrer chaque Diffie-Hellman Ratchet pour reconstruire toute les clés utilisée.

2.6 Compromis

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 resté sur un système de clés publiques comme PGP le fait. Cependant, cette primitive a une identité propre à chaque clé publique ce qui évite un système de certificat trop complexe comme S/MIME.

De plus, pour avoir une *forward secrecy*, on peut ajouter une notion de temps ou de token à l'ID pour chaque batch de messages comme expliqué plus précisément dans la Section 2.8.7.

2.6.1 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 tous ses 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.

S/MIME est la solution prédominante pour s'envoyer des mails chiffrés, cependant il est compliqué de l'utiliser. Il est en effet difficile d'obtenir un certificat pour envoyer des mails et échanger avec une autre personne ayant S/MIME. De plus, la complexité d'un système de PKI et l'overhead induit est assez conséquent.

En faisant quelques essais PGP de mon côté, je me suis heurté à beaucoup de difficultés et de problèmes avec les clés PGP. Notamment pour se les échanger, mais pour envoyer un mail ensuite ce n'est pas si complexe. Le problème étant de bien voir les primitives utilisées pour chiffrer/signer notre email, en effet, les solutions *plug-and-play like* ne permettent pas une gestion précise des primitives cryptographiques utilisées. Cela pourrait induire à des primitives par défaut non sécurisées, comme l'a démontré EFAIL(c.f. 2.4.2).

2.7 Primitives

Cette section présente les primitives considérées pour implémenter un système de messagerie sécurisée. Elle contient un aperçu des primitives analysées, pourquoi elle a été ou non retenue et leur fonctionnement en quelques mots.

2.7.1 Primitives analysées

Les primitives analysées sont présentées ci-après. Elles ont en commun de s'appuyer sur un principe basé sur l'identité. C'est très pratique dans un système de mail car une identité peut-être très vite définie par le biais d'une adresse email. Voici donc les technologies auxquelles les recherches se sont portées :

- Identity based encryption [22], primitive instaurant en premier le binding entre un ID et le chiffrement, ceci afin d'éviter les infrastructures complexes de PKI. Cependant, cette primitive introduit le *key escrow problem*, un problème inhérent de la construction qui nécessite un KGC générant les clés privées pour les utilisateurs et qui en a par la conséquente connaissance.
- HIBE - Hierarchical Identity Based encryption [12]
- Certificateless PKC [1] (Voir ci-après)

2.7.2 Primitive choisie

- Certificateless PKC [1]
Cette primitive a été choisie car elle est similaire à de l'identity based encryption 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 de tous les utilisateurs. Ce problème est résolu dans le certificateless en introduisant des *Partial Private Keys* permettant d'avoir une clé secrète partiellement générée par le serveur (KGC - Key Generation Center) et par l'utilisateur puis assemblée pour former la clé privée seulement connue de l'utilisateur. De plus, cette primitive a l'avantage de ne pas introduire de certificats et ainsi évite la complexité d'une infrastructure de PKI (Public Key Infrastructure).

2.8 Recherches sur la primitive

Dans cette section je vais introduire les détails techniques et les principes mathématiques utilisés. De plus, le choix du schéma parmi tous ceux analysés est détaillé ici. Ainsi que des précisions sur certains principes introduits par ce schéma.

2.8.1 Principes mathématiques

Les variantes de *Certificateless Cryptography* choisies utilisent un concept appelé les *pairings* ou *bilinear map groups* ou *couplages*. Les informations suivantes ainsi que le choix de la librairie découle du livre *Guide to Pairing-Based Cryptography* [17].

Des groupes tels que $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ d'un ordre premier p pour lesquels il existe un mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ avec les propriétés suivantes :

1. Bilinéarité : $e(g^a, h^b) = e(g, h)^{ab}$ pour tout $(g, h) \in \mathbb{G}_1 \times \mathbb{G}_2$ et $a, b \in \mathbb{Z}$;

2. Non-dégénérescence : $e(g, h) \neq 1_{\mathbb{G}_T}$ tant que $g, h \neq 1_{\mathbb{G}_{1,2}}$;
Plusieurs types de Pairings existent :

- **Type 1** : Lorsque $\mathbb{G}_1 = \mathbb{G}_2$;
- **Type 2** : Lorsque $\mathbb{G}_1 \neq \mathbb{G}_2$ mais qu'un isomorphisme $\phi : \mathbb{G}_1 \to \mathbb{G}_2$ est connu, mais pas dans l'autre direction.
- **Type 3** : Lorsque $\mathbb{G}_1 \neq \mathbb{G}_2$ et qu'aucun isomorphise est connu entre $\mathbb{G}_1$ et $\mathbb{G}_2$, dans n'importe quelle direction.

Les différents schémas analysés utilisent souvent les pairings de Type 1, cependant la courbe utilisée et RELIC me permet de faire des pairings de Type 3. Une conversion est faite dans les schémas choisis comme vu ci-après. En effet, les pairings de Type 3 sont des pairings plus efficients avec des courbes plus petites tandis que les pairings de type 1 étaient plus utilisés dans les débuts de la *Pairing Based Cryptography*.

2.8.2 Notations

Avant d'analyser les différents schémas, il faut connaître certaines notions présentes dans les tableaux comparatifs afin de mieux les comprendre. Liste non exhaustive de ces notions et de leurs significations est faite ici.

Types. Les types présentés peuvent être soit **concret** soit **générique**. Les types concrets sont des schémas qui présentent leurs algorithmes en utilisant des calculs bien établis et présentent l'entierté du fonctionnement de leurs schémas, tandis que les schémas présentés génériques peuvent s'appuyer sur d'autres problèmes et se baser sur des algorithmes déjà existants.

Modèle de sécurité. Ces modèles définissent sur quoi le schéma va se reposer pour établir sa sécurité et comment il va l'évaluer face à un adversaire. À nouveau, il existe deux modèles présents dans les schémas analysés, le *Random Oracle Model* et le *Standard Model*. Le *Random Oracle Model* se base sur des oracles aléatoires mais est un peu controversé. En effet, l'aléatoire cryptographiquement sûr est difficile à atteindre, ainsi habituellement le *Random Oracle Model* implémente ces oracles via des fonctions de hachage. Le modèle standard se base lui sur des problèmes mathématiquement difficiles tel que DDH (Decisional Diffie Hellman).

Modèle d'adversaires. Pour évaluer les schémas de certificateless public key cryptography il y a différents niveaux de sécurité établis pour deux types d'adversaires différents. Ces adversaires ont été décrits dans le papier d'Al-Riyami-Paterson [1] pour la première fois afin de prouver que leurs schémas était IND-CCA sûr dans le modèle Standard. Ces deux adversaires ont été défini comme suit :

- Type I (*outsider adversaries*) est permis de remplacer des clés publiques, obtenir des clés partielles privées, et des clés privées puis faire des requêtes de déchiffrements.
- Type II (*honest but curious KGC*). L'adversaire de Type II est en fait un KGC connaissant la Master Secret Key et qui peut donc générer des PPK, obtenir des clés privées et faire des requêtes de déchiffrement tout en faisant confiance à ce KGC pour ne pas qu'il remplace des clés publiques.

Pour chacun des types d'adversaires, il existe différents niveaux de sécurité comme défini dans le livre *Introduction to Ceritifcateless Cryptography* [23].

2.8.3 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 de [23]. En suivant ce tableau, je me suis rendu compte que la construction de Dent-Libert-Paterson [9] était probablement la plus adaptée en vue des propriétés qu'elle présentait. Le tableau se trouve en annexe B.

2.8.4 Détails techniques

Les détails techniques sur le chiffrement avec la *Certificateless Cryptography* sont présentés ci-après. Afin de faire la liaison avec l'implémentation, le code C montrant l'implémentation de chaque algorithme est listé. Ce code est simplifié pour ne pas inclure la création des différents éléments, il est là uniquement pour montrer comment les calculs ont été effectués et non comment la mémoire a été allouée.

Le chiffrement se base sur le problème difficile *The Decision 3-Party Diffie-Hellman Problem* (3-DDH), décider si $T = g^{abc}$ ayant $(g^a, g^b, g^c, T) \in \mathbb{G}_4$.

Pour expliquer les détails techniques, je vais ici montrer les calculs faits dans le schéma choisi [9] et les expliquer. Cependant, dans [9] les pairings symétriques sont utilisés (c.f. Section 2.8.1) mais il est mentionné que l'adaptation à des pairings asymétriques est triviale, c'est donc les pairings asymétriques qui seront utilisés. De plus, [9] utilise un groupe multiplicatif, ainsi la conversion vers un groupe additif (car l'implémentation utilise des courbes elliptiques) est faite. Cela facilite la lecture avec l'implémentation faite :

Setup$(1^k, n)$ **:** Avec $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ avec un ordre $p > 2^k$. g est un générateur de $\mathbb{G}_1$. Ensuite $g_1 = g \cdot \gamma$ pour un $\gamma \leftarrow \mathbb{Z}_p^*$ aléatoire. Puis $g_2 \leftarrow \mathbb{G}_2$. Deux vecteurs (U,V) seront tirés aléatoirement dans $\mathbb{G}_2^{n+1}$ en tant que fonctions de hash notés :

$$F_u(ID) = u' \sum_{i=1}^{n} u_j^{i_j} \quad \text{and} \quad F_v(w) = v' \sum_{i=1}^{n} v_j^{w_j}$$

On va aussi prendre une fonction de hash résistante aux collisions : $H : \{0,1\}^* \rightarrow \{0,1\}^n$. Au final notre mpk (master public key) est :

$$mpk \leftarrow (g, g_1, g_2, U, V)$$

Et le msk (master seret key) est $msk \leftarrow g_2 \cdot \gamma$.

Code Source : Fonction de setup

```
1    void setup(int k, encryption_mpk* mpkSetup, g2_t* msk){
2        // g = generator of G1
3        g1_get_gen(mpkSetup->g);
4        g1_get_ord(p);
5
6        // gamma = random from Zp
7        bn_rand_mod(gamma, p);
8        // g1 = gamma*g
9        g1_mul(mpkSetup->g1, mpkSetup->g, gamma);
10       // g2 = generator of G2
11       g2_get_gen(mpkSetup->g2);
12
13       g2_get_ord(p);
14       // Generate 2 arrays with G2 elements for the U and  vectors
15       for(int i =0; i < MESSAGE_SPACE; ++i){
16           bn_rand_mod(uvGen,p);
17           g2_mul(mpkSetup->u[i], mpkSetup->g2, uvGen);
18           bn_rand_mod(uvGen,p);
19           g2_mul(mpkSetup->v[i], mpkSetup->g2, uvGen);
20       }
21
22       // The master secret key is msk = gamma*g2
23       g2_mul(*msk, mpkSetup->g2, gamma);
24   }
```

Extract(mpk, γ, ID) **:** On prend $r \leftarrow \mathbb{Z}_p^*$ puis on retourne $d_{ID} \leftarrow (d_1, d_2) = (g_2 \cdot \gamma + F_u(ID) \cdot r, g \cdot r)$

Code Source : Code pour l'extraction

```
1    void extract(encryption_mpk mpk, g2_t msk, char* ID, encryption_ppk*
     ↪  partialKeys){
2        // r random from Zp
3        bn_rand_mod(r,p);
4
5        // Computes d1 = msk + r*Fu(ID)
6        g2_t temp;
7        g2_null(temp)
8        g2_new(temp)
9
10       F(ID, mpk.u, &temp, strlen(ID));
11       g2_mul(temp, temp, r);
12       g2_add(partialKeys->d1, msk, temp);
13
14       // Computes d2 = r*g
15       g1_mul(partialKeys->d2, mpk.g, r);
16   }
```

SetSec(mpk) **:** Retourne un secret aléatoirement choisi $x_{ID} \leftarrow \mathbb{Z}_p^*$.

Code Source : Construction de la valeur secrète

```
1    void setSec(bn_t* x){
2        g1_get_ord(p);
3        bn_rand_mod(*x, p);
4    }
```

SetPub(x_{ID}, mpk) **:** Retourne $pk_{ID} \leftarrow (X, Y) = (g \cdot x_{ID}, g_1 \cdot x_{ID})$.

Code Source : Fonction de construction pour la clé publique

```
1    void setPub(bn_t x, encryption_mpk mpkSession, encryption_pk* PKtoGen){
2        g2_mul(PKtoGen->X, mpkSession.g2, x);
3        g1_mul(PKtoGen->Y, mpkSession.g1, x);
4    }
```

SetPriv(x_{ID}, d_{ID}, mpk) **:** On choisit $r' \leftarrow \mathbb{Z}_p^*$ puis on reprends $(d_1, d_2) \leftarrow d_{ID}$ et on va prendre en secret key :

$$sk_{ID} \leftarrow (s_1, s_2) = (d_1 \cdot x_{ID} + F_u(ID) \cdot r', d_2 \cdot x_{ID} + g \cdot r')$$

Avec sk_{ID} étant la clé secrète de l'utilisateur, donnée par l'Extract (notre Partial Private Key) est la valeur secrète de SetSec.

Code Source : Création de la clé privée

```
1    void setPriv(bn_t x, encryption_ppk d, encryption_mpk mpk, char* ID,
     ↪  encryption_sk* secretKeys){
2        bn_rand_mod(r, p);
3
4        // Computes s1 = x*d1 + r*Fu(ID)
5        g2_t pointTemp;
6        g2_null(pointTemp)
7        g2_new(pointTemp)
8
9        g2_mul(secretKeys->s1,d.d1, x);
10       F(ID, mpk.u, &pointTemp, strlen(ID));
11       g2_mul(pointTemp, pointTemp, r);
12       g2_add(secretKeys->s1, secretKeys->s1, pointTemp);
13
14       // Computes s2 = x*d2 + r*g
15       g1_t temp;
16       g1_null(temp)
17       g1_new(temp)
18
19       g1_mul(secretKeys->s2, d.d2, x);
20       g1_mul(temp, mpk.g, r);
21       g1_add(secretKeys->s2, secretKeys->s2, temp);
22   }
```

Encrypt(m, pk_{ID}, ID, mpk) **:** Pour chiffrer $m \in \mathbb{G}_T$, on va reprendre $(X, Y) \leftarrow pk_{ID}$. Pour chiffrer ce message on va tirer aléatoirement $s \leftarrow \mathbb{Z}_p^*$ puis calculer :

$$C = (C_0, C_1, C_2, C_3) \leftarrow (m + e(Y, g_2) \cdot s, g \cdot s, F_u(ID) \cdot s, F_v(w) \cdot s)$$

Où $w \leftarrow H(C_0, C_1, C_2, ID, pk_{ID})$.

Code Source : Code pour le chiffrement

```
1    void encrypt(gt_t m, encryption_pk pk, unsigned char* ID, encryption_mpk mpk,
     ↪  cipher* c){
2        bn_rand_mod(s, p);
3        // Computes C0 = e(Y,g2)^s*m
4        gt_t temp;
5        gt_null(temp)
6        gt_new(temp)
7        pc_map(temp, pk.Y, mpk.g2);
8        gt_exp(temp, temp, s);
9        gt_mul(c->c0, m, temp)
10       gt_free(temp)
11
12       // Computes C1 = s*g
13       g1_mul(c->c1, mpk.g, s);
```

```
14
15          // Computes C2 = s*Fu(ID)
16          g2_t pointTemp;
17          g2_null(pointTemp)
18          g2_new(pointTemp)
19          F(ID, mpk.u, &pointTemp, strlen(ID));
20          g2_mul(c->c2, pointTemp, s);
21          g2_free(pointTemp)
22
23          // Computes C3 = s*Fu(w) où w = C0, C1, C2, ID, PK.x, PK.y
24          g2_t pointTemp2;
25          g2_null(pointTemp2)
26          g2_new(pointTemp2)
27
28          // Construction of the w bytes object to hash
29          // ...
30
31          F(w, mpk.v, &pointTemp2, c0size + c1Size + c2Size + strlen(ID) + pkXSize +
            ↪  pkYSize);
32          g2_mul(c->c3, pointTemp2, s);
33      }
```

Decrypt(C, sk_{ID}, mpk) **:** On peut reprendre $(C_0, C_1, C_2, C_3) \leftarrow C$ est la clé privée $(s_1, s_2) \leftarrow sk_{ID}$. Afin d'accélérer le déchiffrement le calcul suivant peut être fait en tirant une valeur aléatoire $\alpha \leftarrow \mathbb{Z}_p^*$:

$$m = C_0 + \frac{e(s_2 + \alpha \cdot g, C_2) \cdot e(\alpha \cdot g, C_3)}{e(C_1, s_1 + F_u(ID) \cdot \alpha + F_v(w) \cdot \alpha)}$$

Qui donnera m le texte en clair si le chiffré était bien formaté ou un élément aléatoire dans G_T.

Code Source : Code pour le déchiffrement

```
1       void decrypt(cipher c, encryption_sk sk, encryption_pk pk, encryption_mpk  mpk,
        ↪  char* ID, gt_t* m){
2           bn_rand_mod(alpha, p);
3           // Construction of the w bytes object to hash (e(Ppub, Qa)*e(U,
            ↪  H2(m,ID,PK,U))*e(Ppub, H3(m,ID,PK)))
4           // ...
5           // Constructs our point
6           F(w, mpk.v, &pointFv, c0size + c1Size + c2Size + strlen(ID) + pkXSize +
            ↪  pkYSize);
7           F(ID, mpk.u, &pointFu, strlen(ID));
8
9           // m = numerateur * numerateur2 / denominateur
10          // alphaG = alpha * g
11          g1_mul(alphaG, mpk.g, alpha);
12          g1_add(tempNumerateur, sk.s2, alphaG);
```

```
13          // numerateur = e(s2 + alphaG, C2)
14          pc_map(numerateur, tempNumerateur, c.c2);
15          // numerateur2 = e(alphaG, C3)
16          pc_map(numerateur2, alphaG, c.c3);
17          gt_mul(numerateur, numerateur, numerateur2);
18
19          g2_mul(pointFu, pointFu, alpha);
20          g2_mul(pointFv, pointFv, alpha);
21
22          g2_add(Fpoints, sk.s1, pointFu)
23          g2_add(Fpoints,Fpoints, pointFv)
24          pc_map(denominateur, c.c1, Fpoints);
25
26          gt_inv(denominateur, denominateur);
27          gt_mul(*m, numerateur, denominateur);
28          gt_mul(*m, c.c0, *m);
29      }
```

2.8.5 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 de [23]. En analysant les différentes possibilités dans ce tableau, il y a peu de solutions qui se dégagent, en effet, 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 [26] 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) datant de 2006 et n'a pas été cassée depuis. Le tableau se trouve en Annexe B.

2.8.6 Détails techniques

Là encore, le code C montrant l'implémentation de chaque algorithme est listé. Ce code est simplifié pour ne pas inclure la création des différents éléments, il est là uniquement pour montrer comment les calculs ont été effectués et non comment la mémoire a été allouée.

La signature se base sur le problème difficile *The Computational Diffie-Hellman Problem* (CDH).

Ayant P, aP, bP où a, b aléatoires $\in \mathbb{Z}_q^*$ il n'est pas possible de trouver abP.

Pour expliquer les détails techniques, je vais ici montrer les calculs faits dans le schéma choisi [26] et les expliquer. Cependant dans [26] les pairings sont de type symétriques (c.f. Section 2.8.1), les pairings utilisés dans l'implémentation sont de type asymétriques.

Setup(1^k) **:** Tout d'abord on va prendre les groupes d'ordre q énoncés auparavant. Puis on choisit un générateur $P \in \mathbb{G}_1$. La *master secret key* va être choisie aléatoirement $s \in \mathbb{Z}_q^*$. Puis la clé publique calculée : $P_{pub} = sP$. Finalement, trois fonctions de hash distinctes H_1, H_2, H_3 vont être choisies, chacune d'elle *mappant* de $\{0,1\}^*$ à $\mathbb{G}_2$. Pour cela j'ai choisi de faire du *Hash Domain Separation* comme expliqué dans le Chapitre 4. On défini les **params** $= (\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, q, P, P_{pub}, H_1, H_2, H_3)$

Code Source : Code C pour le Setup

```
1   void setupSig(int i, signature_mpk *mpk, bn_t *s){
2       // q = Order of G1
3       g1_get_ord(q);
4
5       // s = random Zq
6       bn_rand_mod(*s, q);
7
8       // Choose a generator P
9       g1_get_gen(mpk->P);
10      // Setup Ppub = s*P
11      g1_mul(mpk->Ppub, mpk->P, *s);
12  }
```

Partial-Private-Key-Extract$(params, s, ID_A)$ **:** Pour avoir la *Partial Private Key* (D_A) de l'user A avec l'identité ID_A. Calculer $Q_A = H_1(ID_A)$. Alors $D_A = sQ_A$.

Code Source : Code C pour l'extraction de la clé partielle privée

```
1   void extractSig(signature_mpk mpk, bn_t msk, char* ID, signature_ppk *
    ↪  partialKeys) {
2       // Qa = H1(ID)
3       g2_map(qa, ID, strlen(ID));
4       // D = msk*Qa
5       g2_mul(partialKeys->D, qa, msk);
6       pc_map(test1, mpk.P, partialKeys->D);
7       pc_map(test2, mpk.Ppub, qa);
8
9       if (gt_cmp(test2, test1) == RLC_EQ) {
10          printf("The partial private key extraction is correct !\n");
11      }
12  }
```

Set-Secret-Value : La valeur secrète $x \in \mathbb{Z}_q^*$ est tirée aléatoirement.

Code Source : Code C pour créer la valeur secrète

```
1    void setSecSig(bn_t* x){
2        // q = Zq
3        g1_get_ord(q);
4        // x = random from Zq
5        bn_rand_mod(*x, q);
6    }
```

Set-Public-Key$(params, x)$ **:** La clé publique PK_A de l'utilisateur A est $PK_A = xP$.

Code Source : Code C pour créer la clé publique

```
1    void setPubSig(bn_t x, signature_mpk mpkSession, signature_pk* PKtoGen){
2        // Public key : Ppub = x*P
3        g1_mul(PKtoGen->Ppub, mpkSession.P, x);
4    }
```

Set-Private-Key$(params, D_A, x)$ **:** La clé privée SK_A de l'utilisateur A est calculée comme ceci $SK_A = (D_A, x)$.

Code Source : Code C pour la création de la clé privée

```
1    void setPrivSig(bn_t x, signature_ppk d, signature_mpk mpk, char* ID,
     ↪  signature_sk * secretKeys){
2        // The private key is composed with D, the partial private key
3        g2_copy(secretKeys->D, d.D);
4        // And x the secret value
5        bn_copy(secretKeys->x, x);
6    }
```

CL-Sign$(params, m, ID_A, SK_A)$ **:** Tout d'abord $r \in \mathbb{Z}_q^*$ est tiré aléatoirement puis on calcule les 2 composantes de la signature :

$$U = rP$$

$$V = D_A + rH_2(m, ID_A, PK_A, U) + xH_3(m, ID_A, PK_A)$$

Ainsi ces composantes forment la signature $\sigma = (U, V)$.

Code Source : Code C pour la signature

```
1    void sign(unsigned char* m, signature_sk sk, signature_pk pk, unsigned char* ID,
     ↪  signature_mpk mpk, signature* s){
2        // r = random from Zq
3        bn_rand_mod(r, q);
4
5        //Computes U = r*P
```

```
6         g1_mul(s->U, mpk.P, r);
7         // Computes V = D + r*H2(m,ID,PK,U) + x*H3(m,ID,PK)
8         g2_copy(s->V, sk.D);
9         // Construct H2(m,ID,PK,U) and H3(m,ID,PK)
10        // ...
11        functionH2(&h2, concat1, lenConcat1);
12        functionH3(&h3, concat2, lenConcat2);
13
14        g2_mul(h2, h2, r);
15        g2_mul(h3, h3, sk.x);
16
17        g2_add(s->V, s->V, h2);
18        g2_add(s->V, s->V, h3);
19    }
```

CL-Verify$(params, PK_A, m, ID_A, \sigma)$ **:** Tout d'abord on va calculer $Q_A = H_1(ID_A)$ puis vérifier cette équation afin de prouver que la signature est correcte :

$$e(V, P) = e(P_{pub}, Q_A) \cdot e(U, H_2(m, ID_A, PK_A, U)) \cdot e(PK_A, H_3(m, ID_A, PK_A))$$

Code Source : Code C pour la vérification

```
1     int verify(signature s, signature_pk pk, signature_mpk mpk, char* ID, unsigned
      ↪  char* m){
2         // Qa = H1(ID)
3         g2_map(qa, ID, strlen(ID));
4         // leftOPerand = e(P,V)
5         pc_map(leftOperand, mpk.P, s.V);
6         // rightOperand = e(Ppub, Qa)
7         pc_map(rightOperand, mpk.Ppub, qa);
8         // Construct H2(m,ID,PK,U) and H3(m,ID,PK)
9         // ...
10        functionH2(&h2, concat1, lenConcat1);
11        functionH3(&h3, concat2, lenConcat2);
12
13        // temp = e(U, H2(m,ID,PK,U))
14        pc_map(temp, s.U, h2);
15        // rightOPerand = e(Ppub, Qa)*e(U, H2(m,ID,PK,U))
16        gt_mul(rightOperand, rightOperand, temp);
17        gt_null(temp)
18        gt_new(temp)
19        // temp = e(Ppub, H3(m,ID,PK))
20        pc_map(temp, pk.Ppub, h3);
21        // rightOperand = e(Ppub, Qa)*e(U, H2(m,ID,PK,U))*e(Ppub, H3(m,ID,PK))
22        gt_mul(rightOperand, rightOperand, temp);
23
24        // The signature is correct if e(P,V) = e(Ppub, Qa)*e(U,
          ↪  H2(m,ID,PK,U))*e(Ppub, H3(m,ID,PK))
```

```
25          if (gt_cmp(leftOperand, rightOperand) == RLC_EQ) {
26              result = 0;
27          }
28      }
```

2.8.7 Ajout d'une pseudo Forward Secrecy

La primitive n'offre pas de forward secrecy par défaut, cependant il existe un moyen d'avoir une *pseudo forward secrecy*, en effet si l'on ajoute un élément à l'ID en fonction des chiffrements on peut établir un simili de cette propriété.

Cela s'explique dans la façon de fonctionner de *Certificateless*, lors du chiffrement l'o peut utiliser la clé publique de l'utilisateur ID et chiffrer à l'aide de ID+timestamp et ainsi avoir un chiffrement pour un destinataire ayant la valeur secrète correspondante et pouvant extraire une clé partielle pour ID+timestamp. Les chiffrements effectués avec un ID+timestamp ont besoin d'une clé privée construite pour ID+timestamp.

Cela permet donc d'avoir une Forward Secrecy par rapport à la clé privée, en effet pour chaque timestamp une clé privée différente est nécessaire. Cependant la valeur secrète utilisée pour cette clé privée sera toujours utilisée. Cette solution permet de limiter les dégats si une clé privée fuit mais pas si une valeur secrète est découverte.

2.9 État de l'art

Dans cette section, une analyse et une comparaison entre les différentes solutions trouvées utilisant la *Certificateless Cryptography* pour une implémentation dans des systèmes de mails. Les articles sont présentés par date de publication.

2.9.1 Email Encryption System Using Certificateless Public Key Encryption Scheme

L'article [10] présente une façon de faire pour chiffrer les mails à l'aide de *Certificateless Cryptography*. Il va d'abord comparer 6 schémas pour choisir celui à utiliser par rapport à ses propriétés. Ensuite il va comparer les différents algorithmes au niveau du temps avec une implémentation simple en J2SE.

Détails techniques. Les détails techniques ne sont pas très fourni dans cet article, en effet, il est mentionné uniquement le choix du schéma (Whang-Huang-Yang). Puis une comparaison des temps entre les différents algorithmes de la primitive. Finalement ils présentent

la différence de temps entre le chiffrement du message via le certificateless et via une clé AES qui est chiffrée avec le certificateless.

Conclusion. Ce papier nous conforte dans l'idée de l'utilisation d'AES pour la rapidité du chiffrement qui va avec cette primitive. Cependant, ils n'expliquent pas comment la clé AES est prise et chiffrée réellement. Une implémentation existe en J2SE mais je ne l'ai pas trouvée. Le schéma choisi l'a été pour son avantage de ne pas utiliser les *pairings* et est donc plus rapide. Puis parmi les autres schémas qui n'utilisent pas les *pairings* à ce moment là, un est de type générique (c.f. sous-section 2.8.2) et l'autre est vulnérable aux *outsider attacks* (c.f. sous-section 2.8.2).

2.9.2 An End-To-End Secure Mail System Based on Certificateless Cryptography in the Standard Model

L'article [16] présente une façon de chiffrer et signer dans un système de mails avec le schéma original de *Certificateless Cryptography* à savoir le schéma d'Al-Riyami et Paterson [1]. Un article complet définissant bien le contexte de mails et formalisant pour la première fois un moyen de chiffrer et signer des mails avec de la *Certificateless Cryptography*. Cela en expliquant dans les détails comment ils feraient, sans implémentations citées de ce schéma.

Détails techniques. Les détails techniques intéressants dans ce papier sont la manière d'encapsuler la clé de chiffrement du message. Sinon le reste s'appuie sur le schéma d'Al-Riyami et Paterson.

Pour établir une clé de chiffrement symétrique afin de chiffrer le mail on va tout d'abord tirer une valeur aléatoire $t \in \mathbb{Z}_p$ puis la chiffrer avec CL-PKC en utilisant la clé publique du destinataire $t* = Enc_{P_B}$. Ce t^* sera envoyé avec l' email. Pour en tirer une clé symétrique on va établir : $K_{AB} = tx_A P_B$ à l'aide de la clé privée de la source et la clé publique du destinataire et enfin la valeur aléatoire tirée auparavant. Puis on va calculer la clé symétrique $K = H_2(Q_A||Q_B||K_{AB})$.

Conclusion. Ce papier est assez complet concernant la partie fonctionnement des mails en globalité et offre une bonne idée pour la construction d'une clé symétrique par mail envoyé. Cependant la mise en place de la clé symétrique et la preuve de son fonctionnement n'est pas très explicitée. D'ailleurs il y a selon moi une erreur dans le papier original pour la logique de déchiffrement de t^* et de la récupération de la clé symétrique. De plus, le système de signature d'Al-Riyami et Paterson a été cassé par [13].

2.9.3 Practical Implementation of a Secure Email System Using Certificateless Cryptography and Domain Name System

L'article [4] traite le problème de la même façon que le précédent mais essaie d'aller plus loin dans les détails d'une implémentation à plus grande échelle (utilisation DNS). Il reprend le même schéma et les mêmes principes pour la création de la clé symétrique de chiffrement. Le même schéma de signature est présent aussi, qui est cassé rappelons-le. Le but serait d'avoir une entrée DNS similaire au DKIM déjà utilisé pour les mails afin d'informer les utilisateurs quelle adresse distribue les clés publiques du domaine en question.

Détails techniques. Beaucoup de détails concernant les Domain Policies qui pourraient être appliqués aux domaines pour la distribution des clés publiques. Proposition d'utiliser les headers des mails pour transmettre la signature du message et informer le destinataire si le mail est chiffré ou non et de transmettre les IDS utilisés et le Timestamp utilisé. En effet, l'introduction d'un timestamp est proposé ici pour avoir un temps d'expiration au mail. Les Domain Policies sont là pour informer les utilisateurs si les mails de ce domaine doivent être signés/chiffrés ou non.

Conclusion. Une implémentation est citée utilisant la librairie MIRACL et en utilisant le C++ comme langage de programmation. L'implémentation est citée comme extension *Thunderbird* en C++ / Javascript. Mais il n'y a pas de réel guide pour implémenter cela au monde réel avec des exemples de configuration DNS et autres. Pas vraiment d'explications sur l'utilisation d'une multitude de KGC ou un seul central, comment les synchroniser et autre... Par contre, beaucoup d'explications sur comment pourrait fonctionner une entrée DNS afin d'informer aux utilisateurs où aller pour récupérer les clés publiques des utilisateurs du domaine en question et des *policies* qui pourraient s'appliquer à ce domaine.

2.9.4 PriviPK : Certificate-less and secure email communication

L'article [2] propose une implémentation très concrète utilisant CL-PKC pour communiquer de manière sécurisée dans la messagerie électronique. Il décrit beaucoup d'aspects que [10] [16] [4] n'ont pas mentionnés comme la *key transparency*. L'article insiste sur la transparence du protocole pour l'utilisateur afin qu'il n'ait pas d'opérations fastidieuses à faire (comme c'est le cas dans PGP et S/MIME par exemple). Cet article s'appuie sur un système de *key agreement* proposé dans la littérature de la cryptographie basée sur l'identité.

Détails techniques. CONIKS serveur, authentification via les clients mails déjà existants (gmail et yahoo), mise en place d'un système de key agreement id-based repensée pour le certificateless. Analysant le code github on peut remarquer que l'implémentation faite est

en fait une modification du Nylas Mail Engine. Cet *engine* a été modifié quelque peu afin d'intégrer du chiffrement lors de l'envoi (sauf si la clé publique du destinataire n'est pas trouvée, auquel cas une erreur intervient). Dans le code on voit beaucoup de TODO laissés aux endroits ajoutés pour le chiffrement de PriviPK.

Conclusion. Cet article est assez intéressant et c'est la seule véritable implémentation trouvée, il y a un repository sur github [9]. Cependant il s'appuie sur du *key agreement*, la création de la clé symétrique est effectuée via les informations de la clé publique du destinataire ainsi que de la clé privée de la source.

Par ailleurs, il insiste sur la transparence et sur l'utilisation des authentifications déjà présentes sur les clients mails comme Gmail et Yahoo. Ceci afin de vérifier que l'utilisateur peut effectivement se connecter à ce compte et obtenir les mails envoyés.

2.9.5 A certificateless one-way group key agreement protocol for end-to-end email encryption

L'article [24] les auteurs présentent un moyen d'avoir une clé partagée entre n-parties et avec un seul message, ce qui permet dans un système de messagerie de n'avoir qu'à envoyer un mail avec les informations nécessaires pour recomposer la clé partagée. Cette clé partagée est utilisée afin de chiffrer le mail et de l'envoyer ensuite avec les informations nécessaires à la création de la clé partagée. De plus, le système est n-parties, cela veut dire que l'on peut envoyer le mail à n personnes et le chiffrer avec la même clé comme le fait PGP.

Détails techniques. Pour ce qui est des détails techniques on peut voir que le principe est de créer une clé partagée à l'aide des différents ID et clés publiques des destinataires. On aura une sous-clé x_i pour chaque utilisateur i. On va construire un $y_i = x_0 \dots x_{i-1} + x_{i+1} \dots x_n$ pour un utilisateur où l'on additionnera tous les x des utilisateurs sauf de l'utilisateur i. Ainsi à la réception du message, l'utilisateur pourra recréer la clé partagée en faisant $y_i + x_i = x_0 + \dots + x_n = K$. Ce K sera ensuite utilisé pour chiffrer le mail.

Conclusion. Ce système est simple et efficace mais ne permets pas la signature des éléments nécessaires à la création de la clé partagée, on peut donc envisager des DOS afin qu'un utilisateur ne puisses plus lire ces messages. Cependant, c'est une construction intéressante se basant sur un *key-agreement* via le *Certificateless Cryptography* et non pas sur ses possibilités de chiffrement/signature.

9. `https://github.com/PriviPK`

Chapitre 3

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.

3.1 Architecture globale

Dans la Figure 3.1, 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.

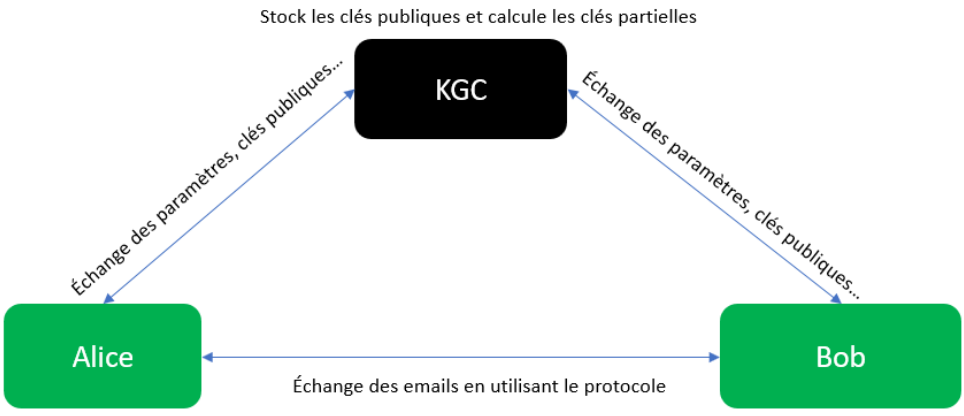

FIGURE 3.1 – Schéma global du protocole

3.2 Acteurs

Les parties impliquées sont les suivantes comme vu à la Figure 3.1.

- Alice : L'envoyeur du mail en direction de Bob. Alice doit discuter avec le KGC pour construire sa clé privée (afin de signer) et récupérer la clé publique de Bob.
- Bob : Le destinataire du message, communique uniquement avec le KGC en ayant reçu le message d'Alice afin de récupérer sa clé publique pour vérifier la signature et construire sa clé privée pour déchiffrer le message.
- KGC : Permet aux différents acteurs de pouvoir récupérer les clés publiques des clients, mais aussi de recevoir les *Partial Private Keys* qui permettent aux acteurs de construire leur clé privée.

Ces différents acteurs sont les principaux présents dans un exemple de *Certificateless Cryptography* dans le système de messagerie implémenté par la suite, mais on pourrait imaginer un serveur gérant uniquement les clés publiques.

3.3 Fonctionnement Certificateless PKC

Je vais ici découper les différents algorithmes présents 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. On peut voir des définitions spécifiques dans l'article sur lequel je me suis appuyé pour ce travail [9].

3.3.1 Chiffrement

La liste des différents algorithmes de *Certificateless Cryptography* et leur description, les détails techniques de leurs implémentations sont disponibles à la Section 2.8.

- *Setup.* (seulement une fois par le KGC).
- *Partial-Private-Key-Extract.* Calcul d'une clé privée partielle lorsqu'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 messages 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.* Déchiffre un message utilisant sa clé privée et l'identité utilisée pendant le chiffrement.

3.3.2 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 apparemment beaucoup plus cassés que ceux de chiffrement (voir tableau en Annexe B). Il faut donc faire attention à vérifier régulièrement les différents schémas afin de vérifier que le schéma choisi ne soit pas mis à mal.

3.4 Design du protocole

Dans cette section le fonctionnement du protocole est décrit à l'aide de diagramme de séquences. Des notations sont utilisées afin de séparer correctement les paramètres utiles au chiffrement sont dénotés d'un E alors que les paramètres pour la signature sont dénotés d'un S.

3.4.1 Premier contact

Description du premier contact effectué avec le KGC. Le KGC nécessite d'être initialisé avant ce premier contact.

La Figure 3.2 permet d'expliquer la première connexion d'un utilisateur. Alice veut s'enregistrer auprès du KGC, ainsi le KGC lui renvoie les paramètres publiques (mpk_S et mpk_E) si aucun utilisateur n'a déjà cette adresse email (ID). Ces paramètres publiques sont assez lourds, en effet, ils font environ 52 kB au total.

L'utilisateur va alors créer sa valeur secrète puis générer sa clé publique. Pour finir, Alice envoie sa clé publique au KGC afin qu'il l'associe à son ID et puisse le donner aux personnes qui veulent envoyer un mail à Alice.

3.4.2 Envoi d'un message

Présentation des différentes actions faites lors de l'envoi d'un message d'Alice à Bob. La Figure 3.3 permet d'avoir un aperçu du fonctionnement du protocole.

Ainsi, l'envoi d'un message se déroule comme suit :

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

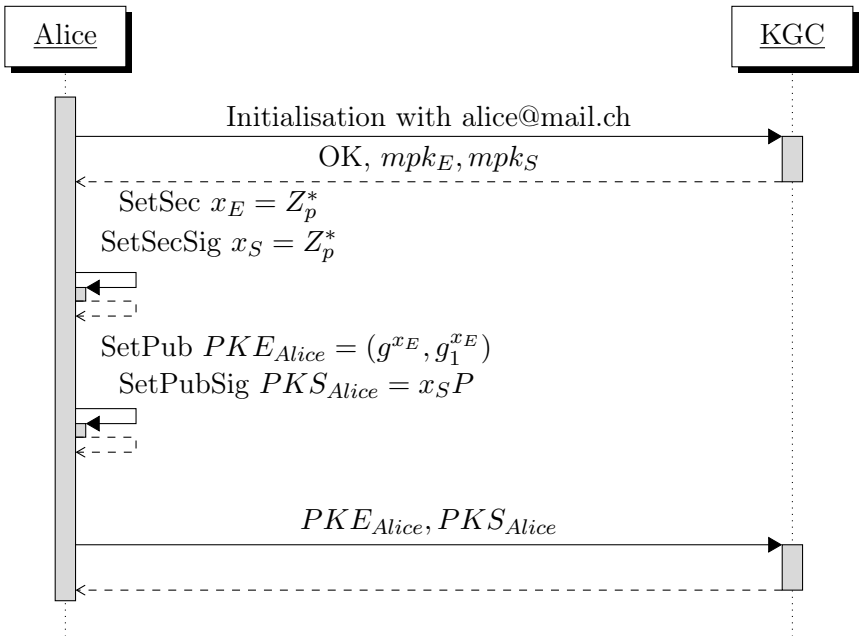

FIGURE 3.2 – Schéma de la première connexion

- 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 comme expliqué dans la Section 2.8.7. Le texte chiffré sera c'.
- Elle va calculer la signature du texte chiffré donné (s' sur la Figure 3.3)
- Alice utilisera un chiffrement authentifié comme AES_GCM pour chiffrer et authentifier son mail à Bob, t pour le tag et c pour le texte chiffré.
- Finalement elle va envoyer tous ces éléments à Bob (à savoir, l'ID utilisé, c, c', t, s' et l'IV utilisé pour AES_GCM).

3.4.3 Réception d'un message

Présentation des différentes actions faites lors de la réception d'un message pour Bob. La Figure 3.4 permet d'avoir un aperçu du fonctionnement de la réception.

Ainsi la réception déclenche les étapes suivantes :

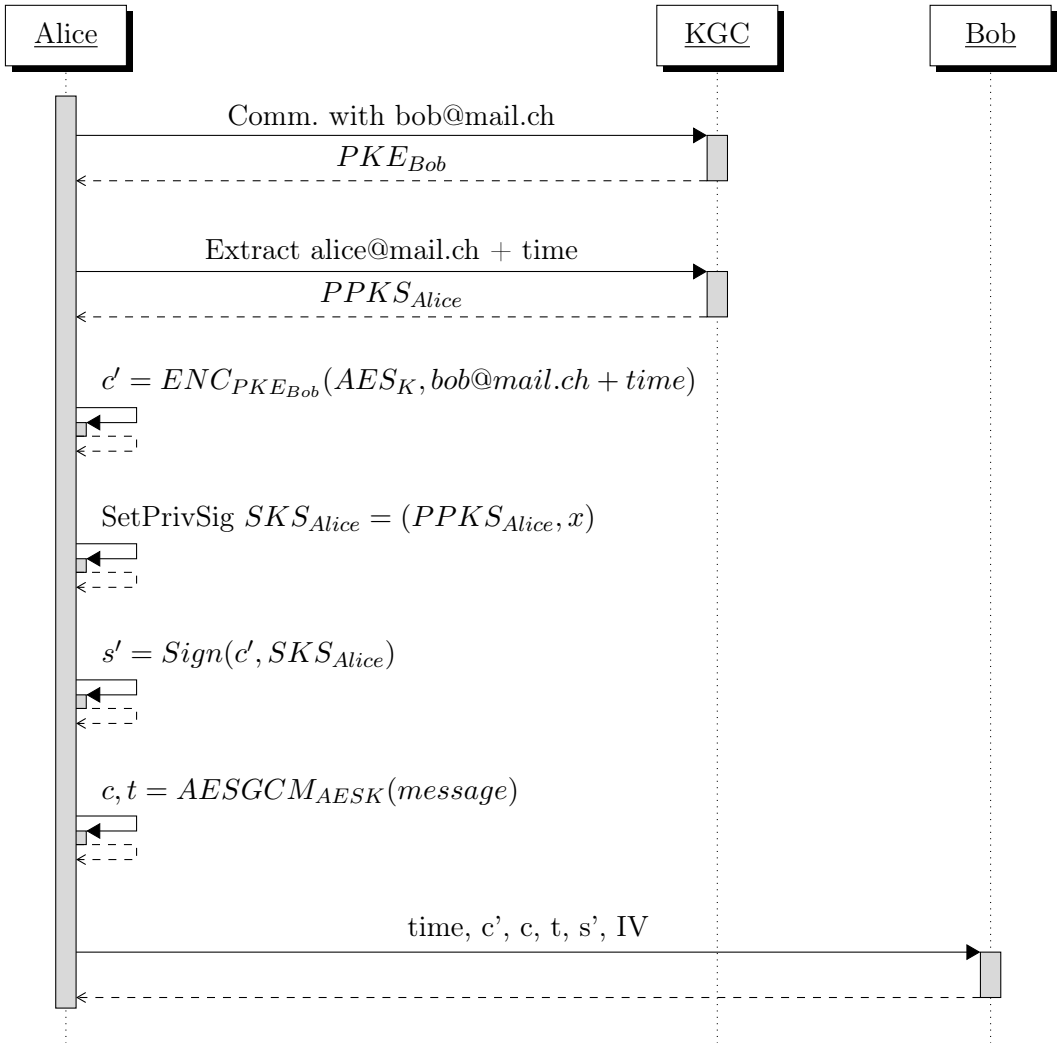

FIGURE 3.3 – Alice envoie un message à Bob

- A la réception, la première chose à faire est de vérifier le texte chiffré de la clé AES. Pour cela, on va demander la clé publique d'Alice au KGC. Puis on va vérifier ce texte chiffré 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 on peut simplement déchiffrer à l'aide d'AES_GCM le chiffré c pour obtenir le message initial.

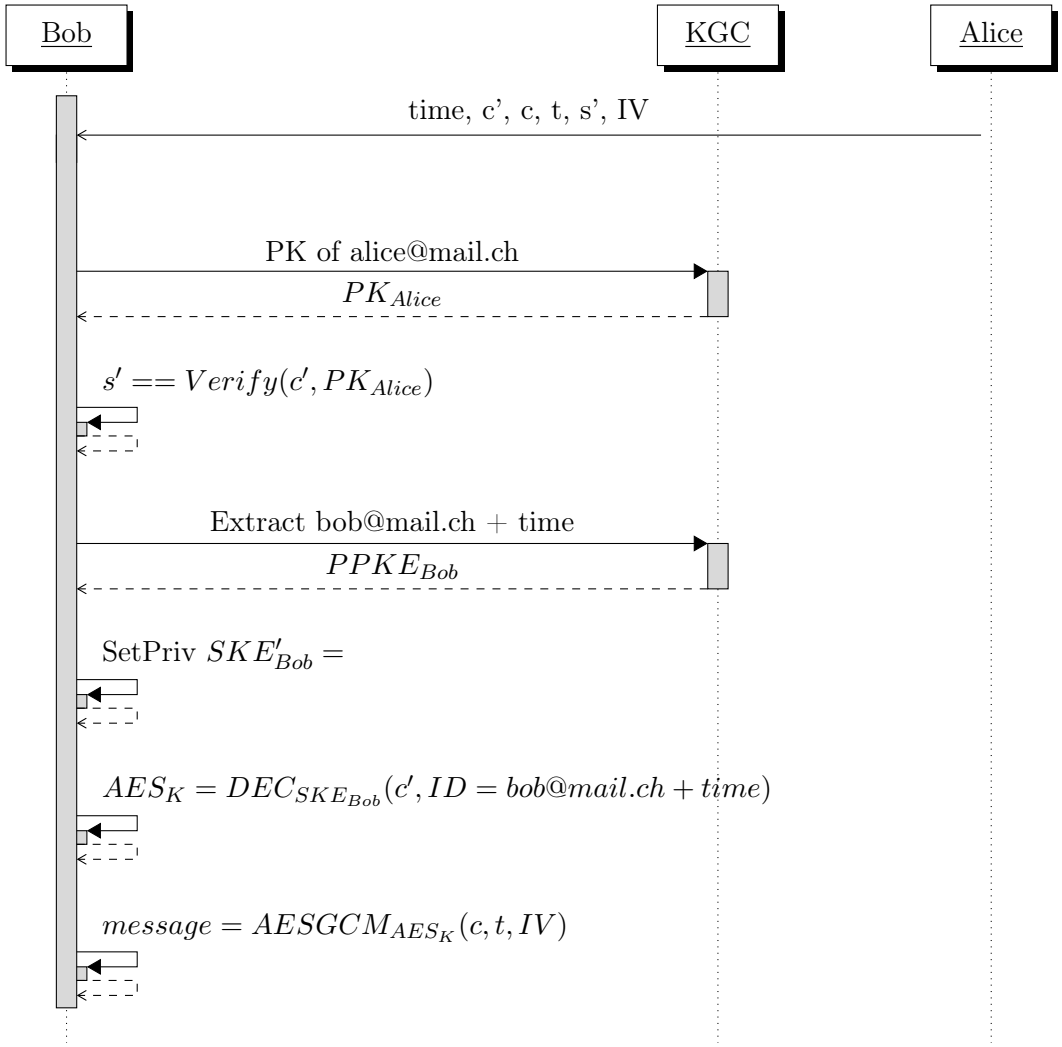

FIGURE 3.4 – Bob reçoit le message

Chapitre 4

Implémentation

4.1 Choix d'implémentations

4.1.1 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/déchiffrement. 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 la mémoire de mon implémentation. Pour pouvoir faire facilement des calculs sur les courbes elliptiques et les pairings en C il me fallait une librairie, ce que je décris dans la section suivante. Comme on peut le voir sur la table 4.1, une différence des temps d'exécution entre les deux langages. Il faut cependant mettre en lumière que les temps sont calculés avec des courbes différentes et des couplages différents (Sage avec des Weil pairings, C avec ate pairings).

Temps des algorithmes entre les différents langages [s]		
Algorithmes	C	Sage
Setup	0.2856898	6.5858234
Encrypt	0.0061584	7.6450206
Decrypt	0.00951	3.3274426

TABLE 4.1 – Table de comparaison des temps d'exécution pour les différents algorithmes de Certificateless Cryptography

4.1.2 Librairie cryptographique

En lisant le livre *Guide to Paring-Based Cryptography* [17], des librairies et implémentations en Sage sont conseillées et des exemples de codes sont montrés. C'est de ce livre que les recherches pour les différentes librairie ont commencées.

La librairie finalement utilisée est RELIC Toolkit [3], 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. De plus j'ai trouvé que RELIC était généralement plus adapté dans le domaine universitaire pour des POC, puis il est plus efficient que d'autres librairies [20].

4.1.3 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 implémentée (certains tests concernant $\mathbb{G}_2$ 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. Pour que RELIC utilise cette courbe il faut le mentionner à la compilation. Heureusement RELIC établi des *presets* pour la compilation de la librairie, la version GMP a été utilisée pour ce POC.

4.1.4 Dérivation de la clé AES

Le but de mon schéma certificateless est de chiffrer puis signer une clé AES qui permettra à mon message d'avoir un chiffrement authentifié. Pour cela il me faut dériver un élément de $\mathbb{G}_t$ en clé AES. En effet, le chiffrement dans le schéma certificateless se fait sur un élément de $\mathbb{G}_t$.
Pour cela j'ai utilisé une fonction permettant d'écrire sous forme compressée cet élément en bytes (fourni par la librairie RELIC utilisée et la fonction gt_write_bin()). Puis j'ai effectué un hachage avec SHA256 dessus, ainsi le résultat du hachage est une clé de 256 bits utilisable par AES-256-GCM. La fonction de hachage doit être par conséquent cryptographiquement sûre.

Code Source : Création de la clé AES depuis un élément $\mathbb{G}_T$

```
1    void get_key(char *aesk, gt_t originalM) {
2        // Get the binary data of the Gt element
3        int sizeAESK = gt_size_bin(originalM,1);
4        uint8_t aeskBin[sizeAESK];
5        gt_write_bin(aeskBin, sizeAESK, originalM, 1);
```

```
6         uint8_t master_key[32];
7         // Hash with SHA-256 to have an master key for KDF from the Gt binary data
8         md_map_sh256(master_key, aeskBin, sizeAESK);
9         // KDF the "master key" to have a usable key to encrypt the data
10        crypto_kdf_derive_from_key(aesk, 32, 1, "AES-KEY", master_key);
11    }
```

4.1.5 Pseudo Forward Secrecy- Timestamp

Afin d'obtenir une *Pseudo Forward Secrecy* l'utilisation d'un timestamp dans l'ID est utilisée comme démontré dans la Section 2.8.7.

L'implémentation prend un timestamp par semaine, ainsi si une clé privé fuite, on ne pourra déchiffrer que les messages de cette semaine. Pour le lier à l'ID un "+" a été ajouté entre l'ID et le timestamp.

Code Source : Gestion du timestamp

```
1     strcat(destinationTimestamp, "+");
2     time_t timestampNow = time(NULL);
3     timestampNow -= timestampNow % 604800;
4     sprintf(timestampStr, "%d", timestampNow);
```

4.1.6 Fonctions de hachage - signature

Pour le schéma de signature, il nous faut plusieurs fonctions de hachage différentes. En effet, ce schéma est basé sur le *Random Oracle Model* comme défini dans le Chapitre 2. Pour appliquer cela, j'ai utilisé la même méthode de mapping disponible dans 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 séparer les domaines des résultats des hashs, cela s'appelle du *Hash Domain Separation*. En effet, on peut voir dans ce draft [11] qui défini comme une simulation pour prendre en compte plusieurs *Random Oracle*.

Code Source : Fonction H2 pour le hachage vers un point de la courbe $\mathbb{G}_2$

```
1     void functionH2(g2_t* to_point, char* bytes_from, int len_bytes){
2         uint8_t to_hash[len_bytes + 1];
3         // Hash domain separation adding 1 byte \x01 before the actual data to hash
4         to_hash[0] = '\x01';
5         memcpy(to_hash + 1, bytes_from, len_bytes);
```

```
6         g2_map(*to_point, to_hash, len_bytes + 1);
7     }
```

4.1.7 Sérialisation des données

Pour la sérialisation des données, typiquement les clés publiques et les clés privées partielles envoyées en réseau ou les clés publiques enregistrées dans les fichiers par exemple, j'ai utilisé la librairie binn [1]. Cela permet de packer facilement des données binaires, pour cela RELIC met à disposition des méthodes g1_write_bin g1_read_bin qui a permis de faire ces enregistrements binaires. Ainsi les transferts de données sont simplifiés.

Cependant, il faut faire attention à certaines choses, on ne peut lire et écrire simultanément à l'aide de binn, si l'on crée un objet via un buffer on ne pourra pas modifier cet objet. Cela m'a posé des problèmes pour l'enregistrement des données secrètes, j'ai donc dû copier l'objet lu pour pouvoir le modifier et sauver les nouveaux paramètres.

4.1.8 Enregistrement des clés publiques (serveur)

Pour l'enregistrement j'ai utilisé une petite base de données NoSQL stockant les clés publiques des utilisateurs sur le KGC. Cela permet de facilement récupérer une clé publique pour un utilisateur si besoin. Pour implémenter cela j'ai utilisé la librairie UnQlite [2]. J'ai stocké les clés publiques pour le schéma de signature et de chiffrement séparément, en effet, l'entrée pour la signature porte le nom "signature/ID" et le chiffrement "encryption/ID".

4.1.9 Récupération via IMAP

Utilisation de la librairie libetPan [3] et de l'exemple imap-sample.c pour la récupération email par IMAP. IMAP a été choisi plutôt que POP3 afin de laisser les mesages sur le serveur et ainsi avoir plusieurs appareils pouvant y accéder. Pour parser les emails cette librairie est aussi choisie ainsi que son exemple mime-parse.c.

4.2 Implémentation clés de chiffrement

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. Pour cela j'ai

1. https://github.com/liteserver/binn
2. https://unqlite.org/
3. https://github.com/dinhvh/libetpan

essayé de faire un système comparable à la Figure 4.1.

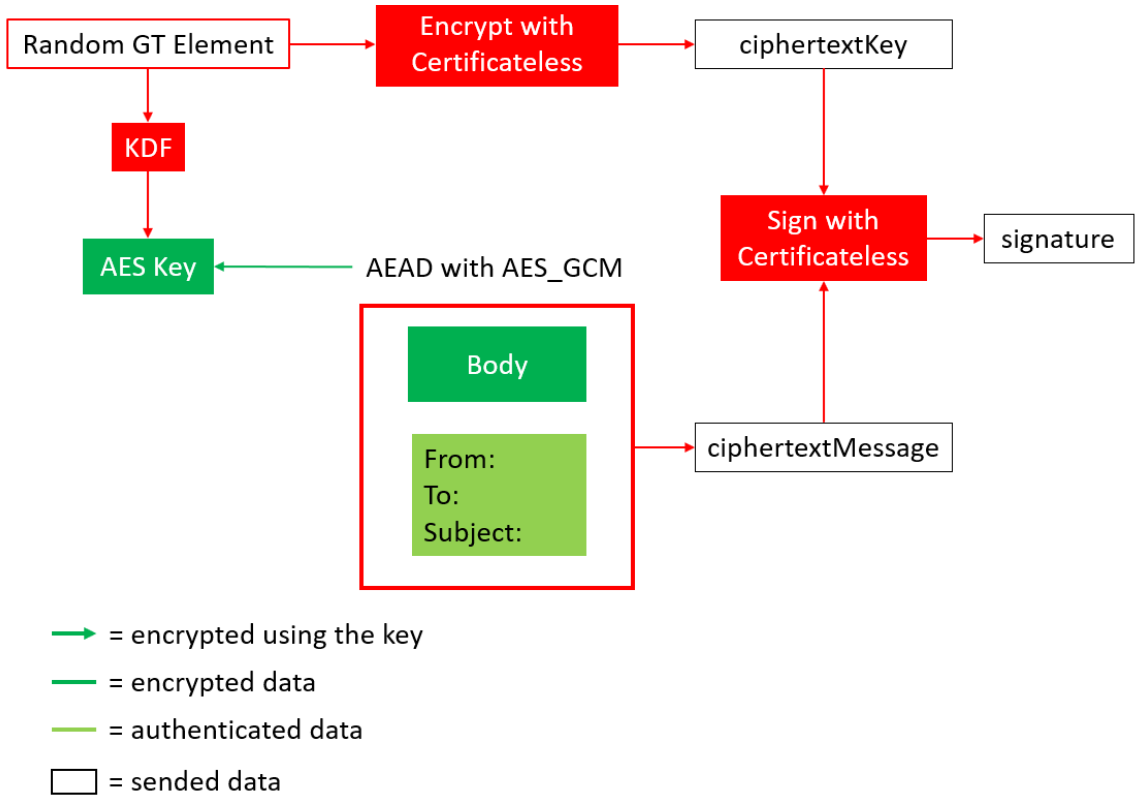

FIGURE 4.1 – Schéma encapsulation des données

4.3 Fonctionnement global POC (KGC)

Présentation du fonctionnement global de l'implémentation du KGC pour le POC. De plus, les problèmes connus et des propositions d'améliorations sont présentés.

4.3.1 Fonctionnement

Le KGC est un élément important du protocole, en effet c'est lui qui va fournir une partie de la clé privée de l'utilisateur. Dans mon cas il permet de distribuer les clés publiques de ses utilisateurs, mais habituellement on fera plutôt appel à un serveur dédié pour la gestion des clés.

Par mesure de généricité j'ai établi des codes d'opérations (arbitraires) afin de définir les opé-

rations demandées au serveur par le client. Cela à l'aide de la librairie binn et les constructions d'objets proposés.

Structure des paquets reçus. Pour la structure des paquets que le KGC va traiter, ils se présentent sous la forme d'un objet binn qui a comme propriétés au moins un code d'opération **opCode** et un **ID** associé. Cela permet de trier le paquet et de l'associer à une opération afin de traiter la donnée amenée avec le paquet. L'ID sert aussi différemment en fonction des codes employés.

Ainsi un paquet typique sera :
{opCode : PK, ID : alice@wonderland.com, payload : xxx}

Codes d'opérations. Les différents codes d'opérations sont :

- HELO : Permet de s'annoncer au KGC pour la première fois, le KGC répondra systématiquement avec les paramètres globaux du système. Cela implique la Master Public Key de chiffrement du KGC ainsi que celle du schéma de signature.
- PK : Permet d'annoncer les clés publiques de l'utilisateur avec un certain ID. Le paquet est composé de {ID : alice@wonderland.com, opCode : PK, PKE : base64 de la PKE, PKS : base64 de la PKS}. La PKE est encodée en base64 par le client est envoyée au serveur. Elle est aussi représentée à l'aide d'un objet binn mais le serveur n'a pas besoin d'en prendre connaissance, il stocke donc cette donnée telle quelle dans la base de donnée NoSQL. La même chose est faite pour la PKS, la clé publique pour la signature.
- GPE : Permet de récupérer la clé publique de chiffrement (utile pour chiffrer un message) d'un utilisateur ayant l'ID mentionné. Ainsi le serveur va simplement regarder dans la base de donnée pour "encryption/ID" et récupérer la clé encodée en base64 et la renvoyer à l'utilisateur. Si le serveur ne trouve pas cette clé, il va renvoyer une erreur dans l'objet et ainsi à la réception on va d'abord regarder cette erreur.
- GPS : Fonctionne de la même manière que "GPE" mais pour les clés publiques de Signature (utile pour vérifier une signature).
- SE : Permet de faire la "Signature Extraction" et donc de demander la Partial Private Key pour l'utilisateur ID. Utilisé lors de la signature d'un message afin de construire sa clé privée et de signer le message avec.
- EE : Permet de faire la "Encryption Extraction" et donc de demander la Partial Private Key pour l'utilisateur ID. Utilisé lors du déchiffrement d'un message pour reconstruire sa clé privée.

4.3.2 Problèmes connus

TODO

4.3.3 Améliorations

Quelques améliorations seraient possibles mais restent à mettre en place, cela permet de donner quelques pistes pour la continuation du travail :

Vérification email. Implémentation d'une vérification par email afin d'être sûr que l'adresse email annoncée appartient bien à l'utilisateur qui s'authentifie pour la première fois. Typiquement lors du "HELO", on pourrait envoyer un mail de vérification avec un code sur l'email annoncé (s'il n'est pas dans la base de données) et demander le code envoyé avant de pouvoir uploader sa clé publique. Ainsi le client devrait pouvoir implémenter cette fonctionnalité aussi. Cela permettrait d'être sûr que tel utilisateur a effectivement tel email.

Mise en place à large échelle Si l'on veut pouvoir mettre en place ce genr de système à une large échelle, il faut prévoir une manière d'avoir plusieurs KGC qui communiquent entres eux dans un système global de réplication par exemple. Ceci afin de pouvoir avoir les mêmes paramètres sur l'ensemble des KGC et que le système fonctionne.

Une autre solution serait par exemple d'ajouter un "tag" d'appartenance à un ID et celui-ci sera donc lié à un certain KGC. Ceci afin de partager la charge qui serait normalement sur un seul KGC. Ce genre de solutions pourrait permettre d'avoir par exemple un KGC par domaine ou alors de les partager via les TLDs. De plus, les KGC pourraient avoir des paramètres générés différemment enrtes eux, ce qui ne seraient pas le cas dans un système de réplication. Cette solution ajoute cependant de la complexité au système, et l'envoi de paquets plus volumineux sur le réseau pour les paramètres globaux d'un KGC qui sont de 52kB, nécessaire dans le cas du chiffrement ou de la vérification de la signature.

Pour le système de clés on pourrait reprendre l'idée de [4] afin d'aller chercher le serveur de clés pour tel domaine.

4.4 Fonctionnement global POC (Client)

Présentation du fonctionnement global de l'implémentation du client mail sécurisé pour le POC, des améliorations possibles et des problèmes connus.

4.4.1 Fonctionnement

Présentation du fonctionnement du POC et les différentes fonctionnalités implémentées dans le client.

Sécurité connexion mail. Pour la connexion au serveur SMTP / IMAP j'ai fait attention à la connexion sécurisée pour éviter de *leak* des mots de passe des utilisateurs du POC. En effet, cela permet de chiffrer les communications avec les serveurs de Gmail comme démontré dans la Figure 4.2.

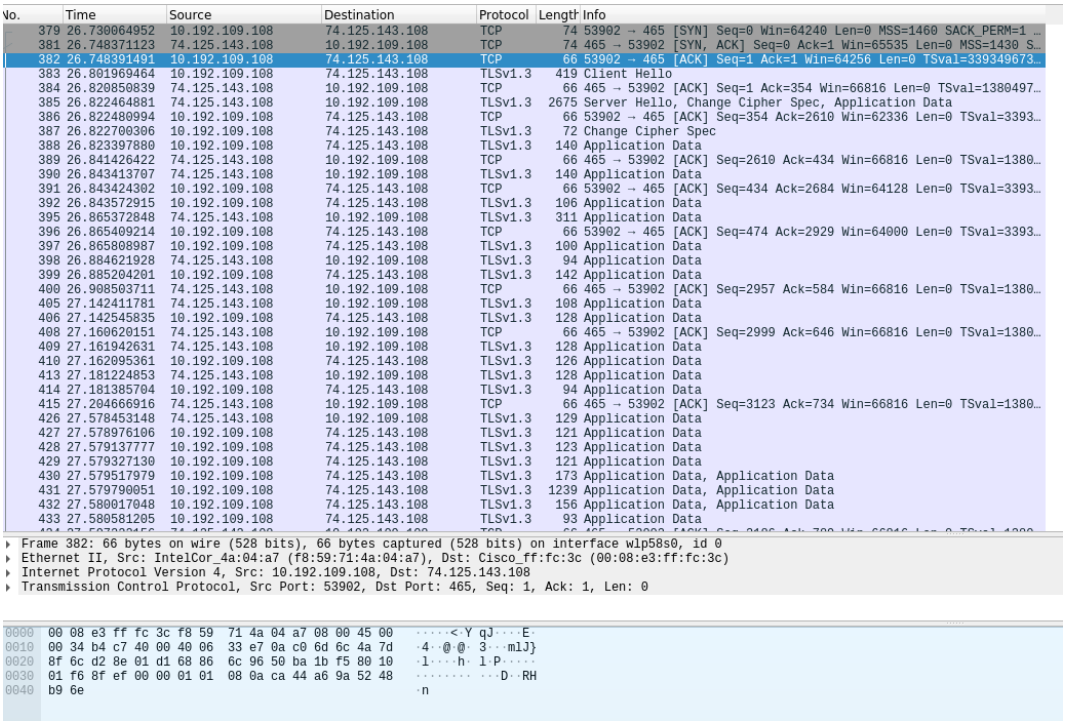

FIGURE 4.2 – Connexion SSL/TLS avec le serveur email

4.4.2 Problèmes connus

Les problèmes qu'il reste à résoudre à ce jour :

Développement sécurisé. Lors du développement j'ai fait attention au maximum d'avoir le moins possible de fuites mémoires à l'aide de sanitizers et de valgrind. Cependant cela ne suffirait pas pour une application correctement sécurisée, il faudra mettre à 0 les structures utilisées et qui stockent des informations confidentielles.

Enregistrement lors d'un crash Lors d'un crash du client en plein millieu du programme ou de l'écriture/chiffrement du fichier des secrets il se peut que celui-ci deviennent corrompu.

Il faudrait éviter de le corrompre car la seule manière d'y remédier serait de le supprimer et ainsi de recréer ses valeurs secrètes.

4.4.3 Améliorations

Les améliorations à amener dans le client :

Multiples destinataires. Pour le moment l'implémentation ne prends pas en compte une situation où un mail doit être envoyé à des destinataires multiples, c'est une fonctionnalité importante à mettre en œuvre dans une implémentation de client mail. Pour ce faire, il faudra spécifier dans les headers l'utilisateur ciblé par tel cipher et signature. Ainsi, lors de la réception, le client prendra les options X-ID-CIPHER-B64 p.ex. Ou alors trouver un moyen d'envoyer un mail différent à chaque utilisateur, sans perdre la possibilité qu'un destinataire puisse choisir de répondre à tous.

Possibilité d'ajouter des pièces jointes. Pour le moment la possibilité d'ajout de pièces jointes n'a pas été pris en considération. Cependant, une des librairies choisies pour la réception des emails pourrait composer des messages contenant des pièces jointes. Il faudrait ainsi les chiffrer avec la clé symétrique avant de l'ajouter dans le mail.

GUI. Mettre en place une interface utilisateur pour le client mail, cela aiderait à rendre le chiffrement plus transparent et plus simple pour l'utilisateur. En effet, demander à l'utilisateur d'écrire son mail au terminal n'est pas spécialement agréable.

4.5 Comparaisons avec l'état de l'art

Dans cette section je vais présenter les différents protocoles et implémentations existantes présentées au Chapitre 2 et les comparer à l'implémentation faite dans ce travail (ci-après CLPKC-POC). Tout d'abord en présentant les différentes propriétés cryptographiques, les tailles d'overhead et l'utilisabilité.

4.5.1 Propriétés cryptographiques

Ici je fais un comparatif sur les différentes propriétés cryptographiques que les systèmes de mails sécurisés proposent avec mon implémentation Certificateless. On peut le voir dans la table 4.2.

Comparaisons des propriétés cryptographiques proposées		
Implémentations	E2EE	Forward Secrecy
CLPKC-POC	Oui	Oui
PGP	Oui	Non
S/MIME	Oui	Non

TABLE 4.2 – Table de comparaison des différentes propriétés cryptographiques

4.5.2 Temps des différentes implémentations

Comparaison du temps mis pour signer / chiffrer et déchiffrer / vérifier un mail entre les différentes implémentations existantes. Dans la table 4.3 on voit les calculs faits.

Comparaisons des temps d'exécution entre différentes implémentations proposées		
Implémentations	Chiffrement	Déchiffrement
CLPKC-POC	0.0061584s	0.00951s
PGP	0	0
S/MIME	0	0

TABLE 4.3 – Table de comparaison des temps d'exécution entre les implémentations de mails chiffrés

4.5.3 Overhead induit

Ici je présente les différents *overhead* que j'ai remarqué en utilisant les différents systèmes de mails sécurisés analysés au Chapitre 2. Dans le tableau 4.4 on voit la taille d'overhead induit par les différents systèmes testés.

Comparaisons de l'overhead induit dans un mail		
Implémentations	Taille overhead	Contenu
CLPKC-POC	Environ 1200 bytes	Signature, timestamp, nonce, Encrypted Session Key
PGP	Environ 300 bytes	Encrypted Session key
S/MIME	0	TODO

TABLE 4.4 – Table de comparaison des différents overhead en rapport avec les solutions existantes

4.5.4 Différences d'utilisabilité

Comparaison de la facilité d'utilisation entre PGP, S/MIME et CLPKC-POC. Cela permet de comparer les différents cas d'utilisation et de voir la souplesse des solutions.

Global Globalement l'utilisation de CLPKC-POC est souvent plus simple que l'utilisation de PGP.

KGC non accessibles

Chapitre 5

Conclusion

Le travail de bachelor se décomposait en deux parties principales, l'analyse des protocoles existants et l'implémentation d'un *Proof Of Concept* utilisant la primitive trouvée lors de l'analyse.

5.1 Conclusions sur l'analyse

Lors de ce travail, la recherche a permis de relever les problèmes inhérents aux différentes solutions existantes et ainsi développer un sens critique. Cependant, il est vrai que chiffrer avec PGP ou S/MIME est de toutes manières mieux que de ne pas chiffrer du tout.

La partie d'analyse faite dans ce travail permet également de voir comment d'autres personnes ont implémentées ce genre de primitive dans un système de messagerie et ainsi les comparer à la solution imaginée dans ce travail. Un travail de comparaison a été effectué afin d'évaluer les performances, la taille des données induites dans le mail et les propriétés cryptographiques assurées.

5.2 Conclusions sur l'implémentation

L'implémentation faite dans ce travail est un POC qui a pour but de prouver la faisabilité de l'incorporation d'un tel chiffrement dans un système de messagerie. Dans ce cadre, le POC n'est pas aboutit avec toutes les options possibles pour un client mail, mais permet tout de même de s'échanger des emails chiffrés entres deux personnes qui seraient inclues dans le système.

En conclusion, ce travail présente avant tout une façon de chiffrer des mails à l'aide d'une primitive peu répandue mais efficace. Cette implémentation pourrait être améliorée afin

d'avoir un vrai programme de client mail et ainsi l'intégrer en production. Il faudra répondre à des questions concernant l'intégration du KGC dans le monde réel (que faire pour avoir plusieurs KGC, comment gérer l'accès restreint à un KGC, ...), ceci afin d'avoir un système plus complet et utilisable dans le monde réel.

De plus, l'intégration d'une *Forward Secrecy* plus propre devra être pensée. En effet, si la valeur secrète d'un client fuit, les messages pourront être déchiffrés.

5.3 Futures directions

Pour améliorer ce système de chiffrement et signature il va falloir penser à plusieurs questions qui se posent dans le cadre d'une intégration plus globale. Des pistes sont données dans le Chapitre 4 mais aucuns tests n'a été effectué concernant l'implémentation de ces idées.

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] Mashael AlSabah, Alin Tomescu, Ilia A. Lebedev, Dimitrios N. Serpanos, and Srinivas Devadas. Privipk : Certificate-less and secure email communication. *Comput. Secur.*, 70 :1–15, 2017.

[3] 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`.

[4] Suresh Kumar Balakrishnan and V. P. Jagathy Raj. Practical implementation of a secure email system using certificateless cryptography and domain name system. *I. J. Network Security*, 18(1) :99–107, 2016.

[5] Adman Barth and Dan Boneh. Correcting privacy violations in blind-carbon-copy (bcc) encrypted email. *Stanford Papers*, 2005 :17, 2005.

[6] J. Callas, L. Donnerhacke, H. Finney, D. Shaw, and R. Thayer. Openpgp message format. RFC 4880, RFC Editor, November 2007. `http://www.rfc-editor.org/rfc/rfc4880.txt`.

[7] Wikimedia Commons. File :e-mail.svg — wikimedia commons, the free media repository, 2014. [Online ; accessed 26-July-2020].

[8] Wikimedia Commons. File :pgp diagram.svg — wikimedia commons, the free media repository, 2020. [Online ; accessed 28-July-2020].

[9] 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.

[10] Yee-Lee Er, Wei-Chuen Yau, Syh-Yuan Tan, and Bok-Min Goi. Email encryption system using certificateless public key encryption scheme. In James Jong Hyuk Park, Jongsung

Kim, Deqing Zou, and Yang Sun Lee, editors, *Information Technology Convergence, Secure and Trust Computing, and Data Management - ITCS 2012 & STA 2012, Gwangju, Korea, September 6-8, 2012*, volume 180 of *Lecture Notes in Electrical Engineering*, pages 179–186. Springer, 2012.

[11] Armando Faz-Hernandez, Sam Scott, Nick Sullivan, Riad Wahby, and Christopher Wood. Hashing to elliptic curves. Internet-Draft draft-irtf-cfrg-hash-to-curve-09, IETF Secretariat, June 2020. `http://www.ietf.org/internet-drafts/draft-irtf-cfrg-hash-to-curve-09.txt`.

[12] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 466–481. Springer, 2002.

[13] Xinyi Huang, Willy Susilo, Yi Mu, and Futai Zhang. On the security of certificateless signature schemes from asiacrypt 2003. In Yvo Desmedt, Huaxiong Wang, Yi Mu, and Yongqing Li, editors, *Cryptology and Network Security, 4th International Conference, CANS 2005, Xiamen, China, December 14-16, 2005, Proceedings*, volume 3810 of *Lecture Notes in Computer Science*, pages 13–25. Springer, 2005.

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

[15] Gaëtan Leurent and Thomas Peyrin. SHA-1 is a shambles - first chosen-prefix collision on SHA-1 and application to the PGP web of trust. *IACR Cryptol. ePrint Arch.*, 2020 :14, 2020.

[16] Bazara Barry Mohammed Hassouna, Nashwa Mohamed and Eihab Bashier. An end-to-end secure mail system based on certificateless cryptography in the standard security model. *IJCSI International Journal of Computer Science Issues, Vol. 10, Issue 2, No 3, March 2013*, 2013 :8, 2013.

[17] Nadia El Mrabet and Marc Joye. *Guide to Pairing-Based Cryptography.* Chapman & Hall/CRC, 2016.

[18] Trevor Perrin and Moxie Marlinspike. The Double Ratchet Algorithm. Technical report, 2016.

[19] 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.

[20] Lucian Popa, Bogdan Groza, and Pal-Stefan Murvay. Performance evaluation of elliptic curve libraries on automotive-grade microcontrollers. In *Proceedings of the 14th Inter-*

national Conference on Availability, Reliability and Security, ARES '19, New York, NY, USA, 2019. Association for Computing Machinery.

[21] J. Schaad, B. Ramsdell, and S. Turner. Secure/multipurpose internet mail extensions (s/mime) version 4.0 message specification. RFC 8551, RFC Editor, April 2019.

[22] Adi Shamir. Identity-based cryptosystems and signature schemes. In G. R. Blakley and David Chaum, editors, *Advances in Cryptology, Proceedings of CRYPTO '84, Santa Barbara, California, USA, August 19-22, 1984, Proceedings*, volume 196 of *Lecture Notes in Computer Science*, pages 47–53. Springer, 1984.

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

[24] Jyh-haw Yeh, Srisarguru Sridhar, Gaby G. Dagher, Hung-Min Sun, Ning Shen, and Kathleen Dakota White. A certificateless one-way group key agreement protocol for end-to-end email encryption. In *23rd IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2018, Taipei, Taiwan, December 4-7, 2018*, pages 34–43. IEEE, 2018.

[25] 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.

[26] Zhenfeng Zhang, Duncan S. Wong, Jing Xu, and Dengguo Feng. Certificateless public-key signature : Security model and efficient construction. In Jianying Zhou, Moti Yung, and Feng Bao, editors, *Applied Cryptography and Network Security, 4th International Conference, ACNS 2006, Singapore, June 6-9, 2006, Proceedings*, volume 3989 of *Lecture Notes in Computer Science*, pages 293–308, 2006.

Table des figures

Liste des tableaux

Annexe A

Outils utilisés pour la compilation

A.1 RELIC Toolkit

Pour pouvoir faire des calculs de *Pairings* sur des courbes elliptiques je me suis fié à RELIC Toolkit [3] qui est une librairie C permettant ce genre de calculs assez simplement. Le choix de la librairie est expliquée plus en détails dans le chapitre 4.

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.

A.2 Libsodium

Pour faire du chiffrement authentifié l'utilisation de la librairie libsodium [1] était le premier choix. En effet, la librairie ne m'est pas totalement étrangère et est très complète au niveau du chiffrement authentifié, gestion de la mémoire sécurisée, algorithmes de dérivation de clés et d'autres. Ainsi le POC nécessite d'avoir libsodium installé sur la machine exécutant le client du POC.

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

A.3 Libbinn

Binn [2] est une librairie de sérialisation en C, permettant de créer des objets, des maps et des listes. Cela a été très utile pour la sérialisation des données entre le serveur et le client du POC et lors de l'enregistrement des différents paramètres du côté du client. Le POC utilise cette librairie et il faut donc avoir la librairie installée pour la compilation.

A.4 Libetpan

LibetPan [3] est une librairie visant à simplifier la gestion des emails dans le code C. Ainsi cette librairie est utilisée pour la récupération des emails depuis le serveur GMail mais aussi pour *parser* les mails reçus et analyser le contenu afin de pouvoir le déchiffrer. Cette librairie aurait pu être utilisée pour l'envoi des mails, mais la connaissance de cette librairie est parvenue après libcurl.

A.5 Libcurl

Libcurl [4] permet d'envoyer toutes sortes de requêtes simplement à l'aide d'une API de programmation assez simple et efficace. Dans le POC elle est utilisée afin d'envoyer des requêtes SMTPs pour envoyer des emails chiffrés générés par le POC.

A.6 UnQlite

UnQlite [5] permet d'avoir une base de données NoSQL gérée en C et ainsi ne pas avoir de base de données complexes pour une utilisation simple comme dans ce cas. En effet, lors de l'enregistrement des clés publiques des utilisateurs, il suffit juste de pouvoir retrouver telle clé par rapport à tel ID. Ainsi une base de données avec ce genre d'entrée suffit. Sur leur site UnQlite propose une version précompilée de la librairie payante, cependant le github [6] est publique et l'on peut donc le compiler de nous même. Ce qui doit d'ailleurs être fait dans le cadre de ce POC. En effet l'instance UnQlite fonctionnant sur le KGC et le KGC fonctionnant dans un environnement multi-threadé il est nécessaire de compiler la librairie avec un flag de compilation activant le support multiThread.

2. `https://github.com/liteserver/binn`
3. `https://www.etpan.org/libetpan.html`
4. `https://curl.haxx.se/libcurl/`
5. `https://unqlite.org/`
6. `https://github.com/symisc/unqlite`

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 soi-même le code. Le code est très commenté au niveau du *main.c* pour bien montrer les différentes étapes telles qu'elles pourraient arriver dans une implémentation finale.

B.2 Tableaux comparatifs

Les tableaux comparatifs cités dans le chapitre 2 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.