Travail d'étude NET2

Chiffrement homomorphe

EPITA ING1 Promo 2022

Mai 2020

Table des matières

Introduction

L'utilité la plus ancienne du chiffrement est de garantir qu'une information ne tombe pas entre de mauvaises mains lors de son trajet de l'emetteur jusqu'au destinataire souhaité. Ces techniques ont longtemps été reservées aux domaines où la sécurité est primordiale tel que le domaine militaire, car elles étaient difficiles à mettre en place.

Avec l'essort des ordinateurs personnels qui ont apporté la puissance de calcul et d'internet qui a apporté le besoin de sécuriser les données, le chiffrement est aujourd'hui utilisé par la plupart des internautes avec le protocol HTTPS par exemple. Ce protocole permet aux utilisateurs de transmettre des données à un serveur, considéré comme sûr, en passant à travers un réseau qui lui pourrait être *écouté*, sans que les données ne soient compromises.

Ce regain d'intérêt pour la cryptographie dans l'informatique a poussé certains chercheurs à imaginer d'autres usages du chiffrement. En effet, avec l'invention du chiffrement asymétrique, qui utilise une clé pour chiffrer et un autre pour déchiffrer, il a été possible d'imaginer un système de signature numérique. Un utilisateur peut alors prouver mathématiquement qu'un message vient bien de lui en le signant avec sa clé privée et en montrant sa clé publique. Ce type de mécanisme est un pillier du fonctionnement des cryptomonnaies où le registre des transactions est distribuée sur une multitude de machines. L'intégrité du registre est garantie non pas par les différents agents du réseau mais par le système de la blockchain lui-même.

Dans l'informatique, les algorithmes de chiffrement sont avant tout des outils qui permettent de garantir la sécurité des données sans nécessiter la confiance envers les autres agents.

Avec le besoin croissant de puissance de calcul, le *cloud computing* est de plus en plus utilisé pour effectuer des opérations couteuses sur des données, il est par exemple possible de louer des GPU pour entrainer des réseaux de neurones. Le problème est qu'il faut faire confiance aux fournisseurs de ces machines pour ne pas enregistrer les données que nous envoyons. Pour cette raison, il serait compliqué de traiter des données sensibles (par exemple médicales) avec du *cloud computing*.

Ces considérations ne sont pas nouvelles puisque déjà en 1978, le problème

de trouver un algorithme de chiffrement permettant d'effectuer des opérations sur les données sans les déchiffrer puis de renvoyer le résultat à l'utilisateur pour qu'il le déchiffre, avait été proposé. Ce type de chiffrement est appelé homomorphe et il aura fallu attendre 2009 pour que le chercheur Craig Gentry propose un algorithme de chiffrement totalement homomorphe [1].

Depuis, de nombreux algorithmes optimisant les temps de calcul se sont succédées et certains projets concrêts utilisant ce type de chiffrement ont vu le jour. J'aborderais ces projets dans la 2ème partie. D'abord, je vais vous expliquer les mécanismes dérrière le chiffrement homomorphe.

1 Le chiffrement homomorphe

1.1 Le concept

Le chiffrement homomorphe est un algorithme de chiffrement avec des propriétés particulières. Lorsque l'on chiffre des données avec cette algorithme, disons des nombres, il doit être possible d'effectuer des opérations mathématiques sur les données chiffrées puis de retrouver, lors du déchiffrement, le même résultat que si on avait appliqué les opérations directement sur les données originales non-chiffrées. Dans la pratique, utiliser un service proposant le chiffrement homomorphe pourrait ressembler à cela :

- Le service propose une fonction de génération de clé qui permet à l'utilisateur de chiffrer les données qu'il veut envoyer sur le serveur.
- Les données chiffrées sont envoyées sur le serveur.
- L'utilisateur peut envoyer une requète chiffrée au serveur.
- Le serveur applique alors des opérations (de manière homomorphique) aux données chiffrées.
- L'utilisateur reçoit le résultat de sa requête et le déchiffre.

La confidentialité des données a été assurée.

Certains algorithmes de chiffrement sont, parfois sans que cela soit voulu, partiellement homomorphe. Par exemple, le chiffrement RSA (qui est utilisé dans le protocole HTTPS) est, dans sa version sans *padding*, homomorphe à la multiplication. Cela veut dire qu'il est possible de multiplier deux nombres chiffrés et d'obtenir en dechiffrant le résultat, le produit des nombes initiaux. J'ai fait un exemple en Python ici.

Pour être plus formel, la fonction de chiffrement d'un algorithme partiellement homomorphe est un morphisme de groupes [2].

1.2 Morphisme d'anneaux

Cette partie s'appuie sur votre cours préféré de Formalisme Ensembles et Structures [3].

On pose,

- l'ensemble des messages non-chiffrées M,

- l'ensemble des messages chiffrées C,

- l'application f, qui est la fonction de chiffrement, prend en paramètre la clé de chiffrement k et le message à chiffrer. Pour simplifier la lecture, on écrira f_k la fonction de chiffrement avec la clé k.

On choisit une opération mathématique $\oplus$ que l'on aimerait appliquer aux données chiffrées. Cette opération doit exister dans M : $\oplus_M$ et dans C : $\oplus_C$. Si on prend l'exemple du RSA, $\oplus$ est la multiplication et $\oplus_M = \oplus_C$ mais ce n'est pas toujours le cas.

On peut alors définir deux groupes $(M, \oplus_M)$ et $(C, \oplus_C)$. f_k est un morphisme de $(M, \oplus_M)$ et $(C, \oplus_C)$ si et seulement si

$$\forall (m_1, m_2) \in M^2,\ f_k(m_1 \oplus_M m_2) = f_k(m_1) \oplus_C f_k(m_2)$$

On voit bien grâce a cette définition que si on applique une fonction de déchiffrement avec la clé k, nommée d_k, à l'équation on obtient

$$d_k(f_k(m_1) \oplus_C f_k(m_2)) = m_1 \oplus_M m_2$$

Autrement dit, déchiffrer l'opération $\oplus_C$ entre les deux messages chiffrées revient à avoir appliquer $\oplus_M$ aux messages non-chiffrées.

Les méthodes de chiffrement utilisant un morphisme de groupes sont des systèmes partiellement homomorphe car ils permettent uniquement d'effectuer une opération $\oplus$ (en général soit l'addition soit la multiplication).

Ce qui nous interesse, c'est un chiffrement totalement homomorphe qui permettrait d'additionner et de multiplier les données chiffrées. Il faut alors parler de morphisme d'anneaux. C'est le même principe que pour les groupes mais avec deux opérations $+$ et $\times$. On a donc les anneaux $(M, +_M, \times_M)$, $(C, +_C, \times_C)$ et notre application f_k qui vérifie les équations

$$\forall (m_1, m_2) \in M^2,\ f_k(m_1 +_M m_2) = f_k(m_1) +_C f_k(m_2)$$
$$f_k(m_1 \times_M m_2) = f_k(m_1) \times_C f_k(m_2)$$

Avoir un algorithme de chiffrement homomorphe à la multiplication et à l'addition est très intéressant puisqu'il est alors possible de reconstruire la fonction logique NON-ET (NAND) tel que NAND(x, y) = 1 - xy. Or la fonction NON-ET permet de reconstruire toutes les autres fonctions logiques, elle peut d'ailleurs être utilisée pour faire un processeur.

Pendant plusieures années, seuls des algorithmes dits *presque* homormorphe (somewhat homomorphic en anglais) ont été conçus. Ces algorithmes permettent bien d'appliquer plusieurs types d'opérations sur les données chiffrée mais seulement un nombre limité de fois avant que celles-ci ne deviennent indéchiffrables. En effet, à chaque fois qu'un nombre est chiffré avec ce type d'algorithme, un bruit est ajouté à ce nombre. Evidemment, si l'on multiplie le nombre chiffré, le bruit est également multiplié. Après plusieurs opérations, le bruit est du même ordre de grandeur que le nombre original, le résultat est alors inutilisable.

1.3 Réamorçage

C'est dans ce contexte que Craig Gentry a proposé le premier chiffrement totalement homomorphe. Son idée ingénieuse a été de partir d'un algorithme *presque* homomorphe et d'appliquer un réamorçage. En effet, dans un système *presque* homomorphe, le chiffrement et les opérations augmentent le bruit du messages chiffrés mais à la fin lorsque l'utilisateur applique la fonction de dechiffrement, il supprime ce bruit (à condition que le bruit soit inférieur à un certain seuil). S'il est possible d'appliquer tout type d'opération sur un message chiffré, nous pouvons appliquer la fonction de déchiffrement elle-même de manière homomorphique qui « remettrait le bruit à zéro ».

L'idée n'est bien sûr pas de complétement déchiffrer le message mais, comme le nomme Craig Gentry, de *rechiffrer* le message de manière homomorphique. Pour se faire, il faut générer une deuxième clé pour chiffrer à nouveau le message chiffré, il est alors doublement chiffré. Ensuite, on peut appliquer à ce message la fonction de déchiffrement, de manière homomorphique, avec une troisième clé, dérivée de la clé initiale, pour réinitialiser le bruit.

Cette étape peut sembler magique et est assez compliquée à comprendre, j'ai relu plusieurs fois le chapitre 4 de la publication [1] pour appréhender la fonction de rechiffrement. Je trouve que l'utilisation de la fonction *Evaluate* dans

la publication originale rend la manipulation difficile à suivre, je vais donc essayer de faire sans.

Ce système fonctionne avec un couple clé privée sk, clé publique pk tel que $d_{sk}(f_{pk}(x)) = x$

- L'utilisateur génère deux couples de clés : $(sk1, pk1)$ et $(sk2, pk2)$
- L'utilisateur chiffre un message m avec $pk1$ tel que $c_m = f_{pk1}(m)$
- L'utilisateur chiffre $sk1$ avec $pk2$ tel que $\overline{sk1} = f_{pk2}(sk1)$
- L'utilisateur envoie $pk2, \overline{sk1}$ et c_m au serveur
- Le serveur effectue des opérations sur c_m et avant que le bruit ne devienne trop grand, il décide d'appliquer la fonction *rechiffre* sur c_m :

- On chiffre le message chiffré une nouvelle fois avec $pk2$.

$$\overline{c_m} = f_{pk2}(c_m) = f_{pk2}(f_{pk1}(m))$$

- On va déchiffrer la chiffrement intérieur (f_{pk1}) de $\overline{c_m}$ en appliquant la fonction de déchiffrement d. C'est à ce moment que j'ai du relire plusieurs fois l'explication pour comprendre.

$$d(\overline{sk1}, \overline{c_m}) = d(f_{pk2}(sk1), f_{pk2}(c_m))$$

Or f est un morphisme donc $g(f(x), f(y)) = f(g(x, y))$

Ainsi,

$$d(f_{pk2}(sk1), f_{pk2}(c_m)) = f_{pk2}(d(sk1, c_m))$$

On peut écrire $d(sk1, c_m)$ comme $d_{sk1}(c_m)$ c'est simplement un raccourci d'écriture, or $d_{sk1}(c_m) = m$

Enfin, on a,

$$f_{pk2}(d_{sk1}(c_m)) = f_{pk2}(m)$$

Après la fonction de *rechiffrement*, on se retrouve avec un message chiffré par $pk2$ au lieu de $pk1$. Le bruit associé au message chiffré a été remis à zéro vis à vis des opérations effectuées avant le *rechiffrement*. On peut évidemment répéter ce processus autant de fois que nécéssaire.

L'utilisateur peut finalement appliquer d_{sk2} sur le résultat pour le déchiffrer.

2 Exemples d'utilisations

2.1 Porticor

Porticor Ltd. est, à ma connaissance, la première entreprise à proposer une utilisation commerciale du chiffrement homomorphe. L'entreprise a été fondée en 2012 soutenue par Red Hat et Amazon AWS, elle a été achetée par Intuit pour plusieurs dizaines de millions de dollars en 2015.

La principale difficulté lorsque l'on veut utiliser le chiffrement homomorphe est liée à la lenteur de calcul des algorithmes actuels. Il est donc, pour l'instant, impossible d'éxecuter toutes les opérations dans le *cloud* de manière homomorphique. Porticor Ltd. a donc rusé en utilisant le chiffrement homomorphe uniquement pour le traitement des clés de l'utilisateur [4].

Ils proposent bien à l'utilisateur du stockage de données chiffrées. Mais ce dernier est un chiffrement symétrique classique (AES). L'utilisateur peut intéragir avec ses données à travers une interface propriètaire qui va déchiffrer les données. La vraie différence de Porticor est dans la manière dont les clés de déchiffrement sont réparties.

Le principe est le même que pour un coffre sécurisé dans un banque, il faut deux clés pour l'ouvrir. Le propriétaire du coffre en a une et la banque possède l'autre. Avec Porticor, la clé AES est coupée en deux parties, l'une est stockée chez l'utilisateur et l'autre chez Porticor. L'interface de déchiffrement a donc besoin de recombiner les deux clés pour traiter les données.

Image tirée du white paper de Porticor Ltd.

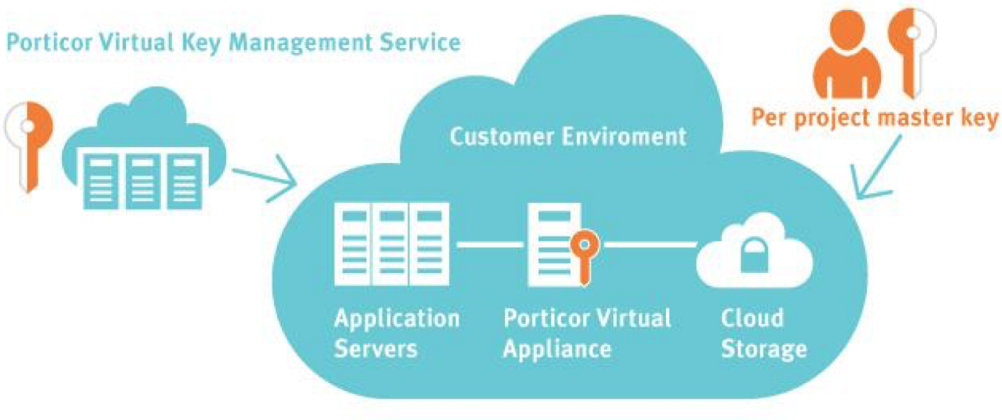

Figure 1 - Patented Split-Key Management Technology

Cependant, il serait trop risqué d'utiliser directement la clé de l'utilisateur puisqu'elle pourrait être volée en cas d'attaque sur le serveur. Avant d'envoyer sa clé sur l'interface de déchiffrement, l'utilisateur peut chiffrer sa clé avec un algorithme homomorphe. Sur le serveur de Porticor, les deux clés sont alors recombinées grâce aux opérations rendues possible par le chiffrement homomorphe. Même si un attaquant arrivait à voler la clé recombinée, il pourrait uniquement déchiffrer la requête en cours car le chiffrement de la clé de l'utilisateur change à chaque utilisation.

2.2 Microsoft

2.2.1 SEAL

En 2018, Microsoft a rendu publique la librairie C++ SEAL (Simple Encrypted Arithmetic Library) [5]. Cette librairie implémente le chiffrement homomorphe BFV [6] qui permet de chiffrer des entiers signés et non-signés. Elle implémente aussi le chiffrement CKKS qui permet de travailler avec des nombres à virgule mais avec moins de précision.

La sécurité de ces chiffrements repose sur le problème Ring Learning with Error (R-LWE) dans un espace discret [7]. Dans SEAL, la clé privé s est un polynome dont les coefficients sont choisis uniformément dans $\{-1, 0, 1\}$.

On peut alors générer une clé publique $(a, b) = (a, as + e)$ où a et e sont également des polynomes aléatoires. D'après le R-LWE, il est supposé très difficile

pour un attaquant de retrouver s à partir de (a, b). J'ai essayé d'implémenter le chiffrement BFV ici.

Pour utiliser la librairie, il faut d'abord compiler le dépôt GitHub SEAL. Ensuite, on peut écrire un programme en C++ ou en CSharp. J'ai suivi l'exemple dans la documentation de SEAL. Tout d'abord, il y a quelques paramètres à initialiser.

```
/* On précise quel chiffrement utiliser (ici BFV) */
EncryptionParameters parms(scheme_type::BFV);
/* La taille de la clé */
size_t poly_modulus_degree = 4096;
parms.set_poly_modulus_degree(poly_modulus_degree);
parms.set_coeff_modulus(CoeffModulus::BFVDefault(poly_modulus_degree));
/* Taille de l'espace des messages non chiffrées M
  doit être 2 fois plus grand que les nombres à chiffrer
*/
parms.set_plain_modulus(128);
```

Ensuite, on peut générer nos clés et les interfaces avec lesquelles nous allons effectuer le chiffrement et les opérations.

```
/* Génération des clés */
KeyGenerator keygen(context);
PublicKey public_key = keygen.public_key();
SecretKey secret_key = keygen.secret_key();
/* Initialisation des interfaces */
Encryptor encryptor(context, public_key);
Evaluator evaluator(context);
Decryptor decryptor(context, secret_key);
```

Il faut encoder les données avant de les chiffrer.

```
int x = 6;
/* On encode x */
Plaintext x_plain(to_string(x));

Ciphertext x_encrypted;
/* On chiffre x */
encryptor.encrypt(x_plain, x_encrypted);
```

SEAL nous permet d'appliquer des fonctions polynomiales sur les nombres chiffrés. Ici on va appliquer $f(x) = x^2 + 6$.

```
Ciphertext tmp;
/* On applique la fonction carré x */
evaluator.square(x_encrypted, tmp);
Plaintext plain_six("6");
/* Et on ajoute 6 */
evaluator.add_plain_inplace(tmp, plain_six);
```

Voici le résultat de ce programme trouvable ici.

```
SEAL/SEALDemo % make
[ 50%] Building CXX object CMakeFiles/sealdemo.dir/sealdemo.cpp.o
[100%] Linking CXX executable sealdemo
[100%] Built target sealdemo
SEAL/SEALDemo % ./sealdemo
On calcule (x^2 + 6) avec x=6.
Le résultat déchiffré est '42'.
SEAL/SEALDemo %
```

2.2.2 CryptoNets

En 2016, Microsoft a démontré qu'il était possible d'utiliser le chiffrement homomorphe pour des projets d'envergure. Ils ont créé un réseau de neurones convolutifs (CNN), nommé CryptoNets, capable de prendre en entrée des images chiffrées.

Pour se faire, ils ont dû adapter certaines fonctions de manière homomorphique [8].

— Le produit matriciel entre l'image et les poids du CNN.

Pour diminuer le temps de calcul, ils ont pu utiliser le fait qu'une seule opérande est chiffrée. En effet, les poids du réseau de neurones n'étant pas chiffrés, le produit avec l'image chiffrée est donc plus simple et rapide.

— Le *max pooling* pour sous-échantillonner l'image lors de la convolution.

La fonction max n'est pas polynomiale, or le chiffrement homomorphe ne permet que d'appliquer des fonctions polynomiales. Ils utilisent la norme infinie de vecteur comme approximation :

$$\max(x_1, \ldots, x_n) = ||\vec{x}||_\infty = \lim_{d \to \infty} \left(\sum_i x_i^d \right)^{1/d}$$

— Les fonctions d'activation (ex : sigmoid, ReLU).

Ces fonctions ne sont pas non plus polynomiales. CryptoNets les a remplacées par la fonction polynomiale non-linéaire $f(x) = x^2$.

Ils ont ensuite testé CryptoNets sur la fameuse base de données MNIST.

Base de données MNIST

Le réseau de neurones doit être capable de reconnaître des chiffres écrits à la main. La difficulté pour l'équipe de Microsoft est bien entendu de travailler sur des images ayant un certain bruit dû au chiffrement homomorphe. Ils ont obtenu une précision de 99% ce qui est équivalent à un CNN sans chiffrement homomorphe.

Pour un lot d'images, CryptoNets prend 122 secondes à les chiffrer, 570 secondes à appliquer le CNN et 5 secondes pour déchiffrer le résultat. Pour rappel, dans l'implémentation du chiffrement de Craig Gentry en 2011, il fallait plusieurs dizaines de minutes pour faire des multiplications sur les nombres chiffrés.

Si on remplace les images MNIST par des images médicales, ce genre de réseau de neurones chiffré prend tout son sens puisqu'il permet de faire des diagnostiques en assurant la confidentialité des données.

3 Conclusion

Le chiffrement homomorphe est un sujet passionnant et prometteur. Il pourrait changer drastiquement les infrastructures du *cloud computing* et permettre une utilisation des données plus éthique dans le cadre du *machine learning*.

Malgré d'importantes avancées ces dix dernières années, de nombreux défis techniques restent à surmonter pour voir des applications de ce chiffrement dans le monde réel. La difficulté pour tout type de chiffrement est de réduire les temps de calculs sans compromettre la sécurité. En 2019, des failles de sécurité dans l'implémentation de SEAL ont été trouvées dans certains cas particuliers [9]. En effet, si un attaquant peut construire un message très spécifique et le faire déchiffrer par le serveur, il serait capable de reconstruire la clé privé de chiffrement.

Compte tenu de l'augmentation de la puissance de calcul et de l'intérêt pour les possibilités offertes par le chiffrement homomorphe, je ne doute pas que ce dernier aura des utilisations innovantes dans les prochaines années.

4 Références

[1] Craig Gentry "A fully homomorphic encryption scheme", 2009
https://crypto.stanford.edu/craig/craig-thesis.pdf

[2] Caroline Fontaine, Fabien Galand "A Survey of Homomorphic Encryption for Nonspecialists", 2007
https://link.springer.com/content/pdf/10.1155/2007/13801.pdf

[3] Edouard Marchais "Formalisme, Ensembles et Structure", 2019
https://moodle.cri.epita.fr/

[4] Porticor Ltd. "Securing Data in the Cloud", 2013
https://files.meetup.com/1304559/Porticor White Paper.pdf

[5] Hao Chen, Kim Laine, Rachel Player "Simple Encrypted Arithmetic Library - SEAL v2.1", 2017
https://eprint.iacr.org/2017/224.pdf

[6] Junfeng Fan, Frederik Vercauteren "Somewhat Practical Fully Homomorphic Encryption", 2012
https://eprint.iacr.org/2012/144.pdf

[7] Hao Chen "A Survey on Ring-LWE Cryptography", 2016
https://youtu.be/pENf_Tuvn3s

[8] Nathan Dowlin, Ran Gilad-Bachrach, Kim Laine, Kristin Lauter, Michael Naehrig, John Wernsing "CryptoNets : Applying Neural Networks to Encrypted Data with High Throughput and Accuracy", 2016

[9] Zhiniang Peng "Danger of using fully homomorphic encryption : A look at Microsoft SEAL", 2019
https://arxiv.org/pdf/1906.07127.pdf