Rapport de stage
Étude et implantation de chiffrement homomorphe

Mathieu Valois

30 août 2016

Résumé

FRANÇAIS - Le chiffrement homomorphe est aujourd'hui une part de la cryptographie si importante qu'elle semble être un phénomène de mode. Si bien que depuis les premiers travaux de Craig Gentry en 2009 des dizaines de schémas ont vu le jour, ce qui rend la tâche difficile à un néophyte pour s'y retrouver dans cette mer de schémas disponibles. Notre travail a donc consisté à rassembler et comparer les plus intéressants d'entre eux de manière détaillée afin qu'il soit plus aisée de bien comprendre toutes les notions impliquées dans ces schémas.

ENGLISH - The homomorphic encryption is nowadays such an important part of the science of cryptography that it seems to be a trend. So much important that since the first Craig Gentry's blueprint in 2009, dozens of schemes were born, hardening the work of a beginner not to be confused in such an amount of available schemes. Our work consisted to gather and compare the most intersting of them in a detailed maner in such a way that it will be easier to understand every concept involved in these schemes.

Table des matières

0 Remerciements

Je remercie premièrement mon maître de stage **Patrick Lacharme** pour son aide tout au long de ce stage, mais aussi pour toutes les remarques constructives qu'il a pu faire concernant le présent rapport. Il a notamment contribué à rendre ce rapport plus cohérent et plus en accord avec les papiers de recherche.

Je remercie également l'équipe **Monétique et Biométrie** pour son excellente ambiance de travail et pour m'avoir accueillit dans ses nouveaux locaux.

Enfin, je remercie **Mehdi Tibouchi, Shai Halevi et Leo Ducas** pour leurs réponses rapides et pertinentes à mes questions concernant leurs cryptosystèmes.

1 Introduction

1.1 Contexte

La cryptographie est l'art d'utiliser des techniques pour garantir certaines propriétés aux messages lors d'une communication entre plusieurs entités. 3 propriétés sont généralement demandées. La confidentialité vise à masquer le contenu d'un message à l'aide d'une clé (tel un mot de passe) afin qu'une personne tierce à la communication ne puisse pas en comprendre le contenu. L'authenticité tend à s'assurer que la personne avec laquelle on communique est bien celle qu'elle prétend être. Finalement, l'intégrité veut garantir que le message expédié et celui reçu sont exactement les mêmes et qu'il n'a pas été altéré.

Pour préserver la confidentialité d'un message, on utilise classiquement une méthode qui s'appelle le chiffrement : on modifie le message à l'aide d'un algorithme et d'une clé afin que le nouveau contenu n'ai plus aucun lien avec le message d'origine. Il existe aujourd'hui 2 types d'algorithme de chiffrements : celui à clé secrète dit symétrique (la clé pour chiffrer un message est aussi celle pour le déchiffrer) et celui à clé publique dit asymétrique (une paire de clés publique/privée mathématiquement dépendantes). En pratique, les algorithmes symétriques sont beaucoup plus rapides que les algorithmes asymétriques, ils requièrent néanmoins de se mettre d'accord sur une clé commune ce qui n'est pas possible dans le cas où quelqu'un écoute la conversation (à moins d'organiser une rencontre physique, très contraignante). C'est

pourquoi il est d'usage d'utiliser le chiffrement à clé publique ou d'effectuer un échange de clé Diffie-Hellman pour initialiser une clé secrète et ainsi être capable d'utiliser des algorithmes symétriques.

La plupart des algorithmes à clé publique reposent sur des problèmes mathématiques dits difficiles, le plus connu RSA par exemple utilise le problème RSA : étant donnés un entier n produit de deux nombres premiers distincts p et q, un entier naturel e premier avec $(p-1)(q-1)$, et un entier naturel c, trouver un entier naturel m tel que $m^e \equiv c \pmod{n}$. Ce problème est facile si l'on connaît la factorisation du nombre n, ce qui est le cas de l'individu qui engendre les clés privée/publique car connaître p et q est nécessaire pour déchiffrer un message. Jusqu'à aujourd'hui, ce problème tient puisqu'il n'existe pas d'algorithme qui permet de factoriser un nombre ou de résoudre le problème RSA efficacement avec un ordinateur classique. Cependant, avec l'arrivée de l'informatique quantique, toute une partie de la cryptographie classique à clé publique tombe : factoriser un nombre s'effectuerait en temps $O((logN)^3)$ avec N la taille en bit du module RSA n. Cet algorithme existe, même si aucun ordinateur quantique n'est encore capable de réellement s'en servir, et il s'appelle l'**algorithme de Shor**[22]. La cryptographie basée sur les courbes elliptiques ne résistera pas non plus à l'arrivée des ordinateurs quantiques (le problème du logarithme discret pourra être résolu en un temps polynomial).

Il est donc important de se concentrer sur des manières de construire de nouveaux schémas de chiffrement qui résistent à l'informatique quantique. Il en existe déjà, citons **Mc Eliece**[18], **Quantum Key Distribution**[6], **Stern**[24], **Unbalanced Oil and Vinegar**[17]. Craig Gentry propose dans sa thèse en 2009 [12] un nouveau schéma de chiffrement à clé publique basé sur les réseaux euclidiens résistant à l'informatique quantique, et ce schéma à la particularité d'être **homomorphe**. Même si ces schémas résistent à l'informatique quantique, on ne se contentera ici que d'étudier leur homomorphisme.

1.2 Une mise en œuvre pratique sur l'embarqué : le stage d'Enguerran Bernard[7]

Enguerran Bernard a effectué son stage de fin d'études 2013/2014 au sein du Commissariat à l'énergie atomique et aux énergies alternatives (CEA), s'intitulant "Démonstrateur du calcul homomorphe sur système Androïd". Le principal objectif de ce stage était de mettre en œuvre, en tenant compte des contrainte techniques du téléphone, un système de client/serveur (FI-

GURE 1) permettant de vérifier si le taux de certaines substances (cholestérol, triglycérides, ...) dépassaient un seuil donné ou non, tout cela de façon homomorphe de telle sorte que le serveur n'ai à aucun moment connaissance des données envoyées par le client. Pour cela il s'est servit de la librairie C++ HElib et l'a donc compilé de façon astucieuse afin de la faire fonctionner sur le smartphone avec son environnement Java.

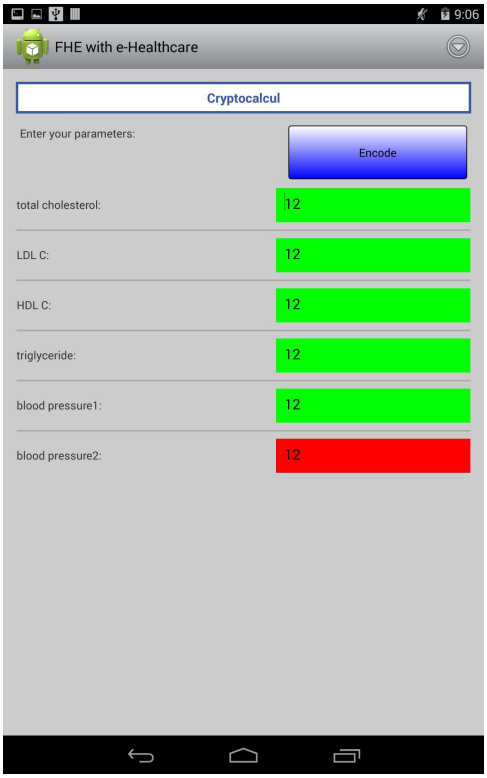

FIGURE 1 – L'application en question. L'utilisateur entre ses données, les envoie au serveur avec le bouton **encode**, le serveur retourne le résultat du calcul et l'application déchiffre puis affiche le résultat sous forme de code couleur.

Évidemment, les limitations matérielles du téléphone conduisent à des problèmes de temps et d'espace quand il s'agit d'invoquer la génération de clés, le chiffrement et le déchiffrement. Enguerran Bernard soulève que compte tenu de la taille d'un bit chiffré et que l'application devait en chiffrer 48 (un octet pour chaque champs), l'espace mémoire n'était pas suffisant. Il a donc fallu chiffrer un premier bit pour l'envoyer directement au serveur avant de chiffrer le suivant (en appelant le GarbageCollector pour libérer la

mémoire), jusqu'au dernier.

Son rapport est particulièrement intéressant car il regroupe les difficultés que l'on peut avoir lorsqu'on tente d'implanter un schéma homomorphe et plus précisément sur un système embarqué. On peut donc constater que même si les cryptosystèmes homomorphes s'exécutent efficacement sur des ordinateurs classiques, il faut requérir à des astuces pour les adapter aux smartphones.

1.3 Objectifs du stage

Dans ce rapport, nous allons premièrement décrire le tout premier schéma totalement homomorphe introduit par la thèse de Craig Gentry en 2009 [12]. Ayant proposé un cryptosystème homomorphe basé sur des problèmes difficiles concernant les réseaux euclidiens, il est le précurseur dans ce domaine et ses travaux ont motivé certains chercheurs en cryptographie à s'y intéresser. Nous allons ensuite faire un état de l'art en exposant 4 cryptosystèmes homomorphes : Brakerski-Gentry-Vaikuntanathan (**BGV**), une revisite du schéma de Gentry basé sur le problème **LWE**, van Dijk-Gentry-Halevi-Vaikuntanathan (**DGHV**) un schéma basé sur les entiers et les problèmes **SSSP** et **AGCD**, ensuite Coron-Naccache-Tibouchi (**CNT**) une amélioration de **DGHV** réduisant d'un facteur 80 la taille de la clé publique en utilisant des générateurs pseudo-aléatoires et enfin Ducas-Micciancio (**DM**), un cryptosystème également basé sur **LWE** ayant été construit pour effectuer un rechiffrement en 1 seconde.

Il conviendra ensuite d'étudier leurs implantations : **DGHV** codé par mes soins en **Python** puis en **Sage** pour des questions de performances, mais on verra que son amélioration **CNT** codé par ses créateurs en **Sage** est bien plus efficace sur tous les aspects, et enfin **HElib**, la librairie C++ écrite par Shai Halevi et Victor Shoup implantant le schéma **BGV**.

Enfin une comparaison des performances de chacune des implantations sera détaillée, mettant en avant que malgré son âge, **HElib** reste globalement l'implantation la plus efficace grâce à ses améliorations au fil des années (batching ou regroupement de données, bootstrapping, multi-threading) et à son usage très bas niveau de librairies mathématiques en C++. Même si évidemment, selon le contexte et l'application que l'on souhaite faire du schéma, d'autres peuvent être plus intéressants.

2 Craig Gentry ou le précurseur du chiffrement totalement homomorphe

En 1978 Rivest, Shamir et Dertouzos[21] remarquent une propriété à la fois intéressante et inquiétante de leur cryptosystème RSA : le produit de deux nombres chiffrés est égal au chiffré du produit des deux nombres en clair. Mathématiquement : $E(x) \times E(y) \pmod{n} = E(x \times y) \pmod{n}$ avec E la fonction de chiffrement, et x, y deux entiers. RSA est donc dit **partiellement homomorphe** pour la loi multiplication.

Définition 1 *Un cryptosystème est dit **partiellement** homomorphe s'il commute avec certaines lois mathématiques sur l'espace des messages chiffrés.*

Définition 2 *Un cryptosystème est dit **totalement** homomorphe s'il commute avec toutes les lois sur l'espace des messages chiffrés.*

Ils se demandent donc s'il ne pouvait pas exister des schémas qui seraient **totalement homomorphe**.

En 2009, soit 30 ans après le questionnement de Rivest et Shamir, Craig Gentry propose dans le cadre de sa thèse un nouveau schéma **totalement homomorphe** basé sur des problèmes concernant les réseaux euclidiens.

Définition 3 *Un réseau euclidien est un sous-groupe discret additif de R^n.*

Définition 4 *Une base $b = \{b_1, b_2, ..., b_n\}$ d'un réseau euclidien L dans R^n est un ensemble de n vecteurs linéairement indépendants. Les combinaisons linéaires entières des vecteurs de cette base forment L.*

$$L = \{x \in \mathbb{R}^n \ tq \ x = \sum_{i=1}^{n} x_i b_i, x_i \in \mathbb{Z}\}$$

La Figure 2 montre un exemple visuel de réseau euclidien.

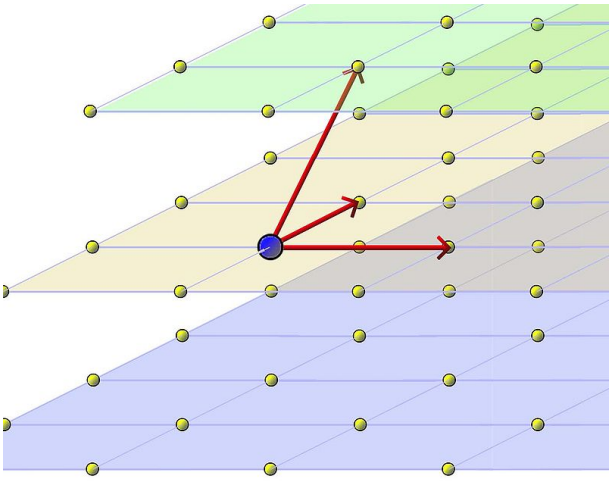

FIGURE 2 – Réseau euclidien de dimension 3 avec 3 vecteurs formant une base (*Wikipédia*)

Les deux problèmes difficiles utilisés par Gentry pour son cryptosystème sont les suivants :

— **Définition 5** ***Shortest Vector Problem (SVP)*** *: étant donnés un réseau L et une base B de L, trouver le vecteur non nul le plus court du réseau (au sens de sa norme)*

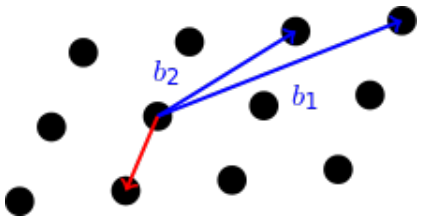

FIGURE 3 – En bleu la base donnée du réseau, en rouge le vecteur le plus court

— **Définition 6** ***Closest Vector Problem (CVP)*** *: étant donnés un réseau L, une base B de L et un vecteur v, trouver le vecteur le plus proche de v appartenant à L*

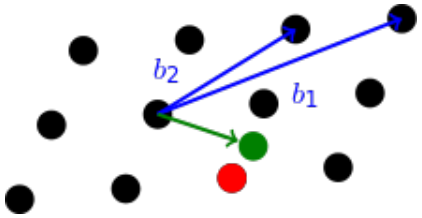

FIGURE 4 – En bleu la base donnée du réseau, en vert le vecteur donné, en rouge le vecteur le plus proche du vecteur vert appartenant au réseau

Pour ces deux problèmes, trouver la solution est en général difficile sauf si l'on dispose d'une "bonne" base, c'est à dire une base "assez" courte et "assez" orthogonale. La difficulté du problème SVP permet de s'assurer que la clé privée ne peut pas être trouvée facilement. Quant au problème CVP, sa difficulté permet de s'assurer que le bruit d'un chiffré n'est pas calculable donc qu'il est difficile de trouver le texte clair correspondant.

Ainsi Gentry propose le schéma suivant :

Gentry.KeyGen(λ). Soit R un anneau : $R = \mathbb{Z}[X]/f(x)$ avec $f(x)$ un polynôme unitaire de degré n irréductible. Soit I un idéal dans R, c'est à dire un réseau $I \subset \mathbb{Z}^n$ avec la donnée d'une base $B_I : \mathbb{Z}^n/B_I = P(B_I) = \{P_i\}$ est l'espace des textes clairs, avec $P(B_I)$ le parallélépipède associé à cet espace.

Chaque entité choisit un idéal J avec la contrainte $I + J = R$ ainsi que 2 bases $B_J^{S_k}$ (bonne base), $B_J^{P_k}$ (mauvaise base).

Soit D une distribution donnée ajoutant du bruit au texte clair donné. Soit $P(B_J^{S_k}) = \{\sum_{i=1}^{n} x_i b_i \ : \ x_i \in [-1/2, 1/2]\}$ l'ensemble des points du réseau contenus dans le parallélépipède défini par la base $B_J^{S_k}$ et centré en l'origine.

La clé publique est $R, B_I, B_J^{P_k}$ et la clé privée est $B_J^{S_k}$.

Gentry.Encrypt($R, B_I, B_J^{P_k}, \Pi$). Soit Π le message clair à chiffrer. On lui ajoute du bruit i selon la distribution $D : \Pi + i \xleftarrow{D} \Pi$.

Le chiffré est : $\Psi \leftarrow (\Pi + i) \mod B_J^{P_k}$ (on soustrait itérativement à $\Pi + i$ des vecteurs de la base $B_J^{P_k}$ jusqu'à ce que le résultat soit dans le parallélépipède $P(B_J^{P_k})$).

Gentry.Decrypt$(B_J^{S_k}, \Psi)$. Soit Ψ le message chiffré à déchiffrer. $\Pi^* \leftarrow (\Psi \mod B_J^{S_k}) \mod B_I$. $\Pi^* = \Pi$ à la condition que le Ψ soit correct, c'est à dire ne contienne pas trop de bruit, formellement $\Pi + i \in P(B_J^{S_k})$.

Gentry.Recrypt$(B_J'^{P_k}, \bar{B}_J^{S_k}, Gentry.Encrypt(), Gentry.Decrypt(), \Psi)$. Soit $\bar{B}_J^{S_k}$ le chiffré de la clé privée $B_J^{S_k}$ (on considère la clé privée comme un message à chiffrer) et $B_J'^{P_k}$ une nouvelle clé publique.

Calculer $\bar{\Psi} \leftarrow$ **Gentry.Encrypt**$(R, B_I, B_J'^{P_k}, \Psi)$ (on chiffre le chiffré avec la nouvelle clé publique).

Le rechiffré est **Gentry.Decrypt**$(\bar{B}_J^{S_k}, \bar{\Psi})$.

Il faut savoir que ce schéma (contrairement à ceux sur les entiers par la suite) permet aussi de chiffrer autre chose que des bits (c'est le paramètre "rho" dans l'implantation).

Afin de chiffrer un message, premièrement on le convertit en vecteur Π du réseau, puis on lui ajoute du "bruit" (on choisit un vecteur $\Pi + i$ suffisamment loin de manière à ce que Π reste quand même son vecteur le plus proche) et on calcule son modulo $B_J^{P_k}$ (on soustrait des vecteurs de la base $B_J^{P_k}$ jusqu'à ce que le résultat Ψ soit dans $P(B_J^{P_k})$). Pour déchiffrer, on résout le problème CVP avec la base privée (problème facile quand on a une bonne base du réseau). Faire la somme de 2 chiffrés consiste à faire la somme des 2 vecteurs respectifs modulo la clé publique, et le produit en le produit vectoriel de leurs vecteurs modulo la clé publique. On remarque, que le "bruit" s'accumule lors des opérations homomorphes : l'addition le double, la multiplication le met au carré. Si l'on fait trop d'opérations, le vecteur du texte chiffré sera plus proche d'un autre vecteur du réseau que de celui auquel correspond le clair.

Afin de rechiffrer, il nous faut posséder la clé de rechiffrement : c'est le chiffré de la clé privée, c'est à dire de la chiffrer en la considérant comme un message (une suite de bits si $\mathcal{P} = 2$, d'octets si $\mathcal{P} = 8$, ...). Ensuite on chiffre le message Ψ avec une nouvelle clé publique, puis on calcule le circuit de déchiffrement avec le chiffré de la clé privée.

Définition 7 *Un cryptosystème est dit "Somewhat" homomorphe s'il permet d'effectuer un nombre limité d'opérations homomorphes sur les chiffrés, au delà duquel le message n'est plus déchiffrable (trop de bruit introduit).*

Pour pallier cette limite, Gentry introduit la notion de "bootstrap". L'idée est de pouvoir évaluer le circuit de déchiffrement dans le domaine du chiffré, retournant ainsi un nouveau texte chiffré ayant le même texte clair qu'avant l'opération.

Définition 8 *Un circuit [booléen] est un ensemble de portes "et" (produit) et "xor" (somme)*[1] *ayant t entrées et une unique sortie, et dont les portes qui le composent relient leurs entrées avec leurs sorties. Il peut être représenté comme un polynôme t-varié. Toutes les fonctions (opérations) [booléennes] et algorithmes peuvent être écrits sous forme d'un circuit [booléen] composé de portes "et" et "xor".*

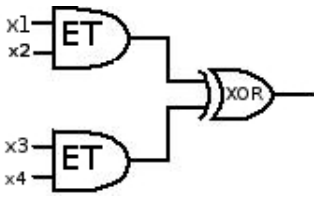

FIGURE 5 – un circuit [booléen]. Ici, le circuit calcule $x1x2 + x3x4$

Définition 9 *Un cryptosystème est dit "Fully" homomorphe (ou bootstrappable) s'il est capable d'évaluer son propre circuit de déchiffrement plus (au moins) une opération supplémentaire de façon homomorphe*[2]*.*

Il montre dans sa thèse comment il est possible de "réduire" le circuit de déchiffrement de telle sorte que sa profondeur multiplicative (le degré du polynôme associé) soit suffisamment petite pour qu'il puisse être évalué par le système (plus une opération supplémentaire) et que le schéma reste correct (le clair associé reste le même après déchiffrement) : on parle de squash.

Une implantation de ce schéma existe en C++, écrite par Craig Gentry et Shai Halevi [13] qui a servi notamment pour générer des challenges publics. C'est cette implantation qui est mentionnée par "Gentry" dans le tableau FIGURE 7. Nous avons effectué une batterie de tests concernant ce schéma pour en évaluer les performances : génération des clés, chiffrement,

1. uniquement ceux deux là dans notre cas, en général un circuit peut contenir n'importe quelle porte logique

2. afin de pouvoir effectuer une opération sur le résultat, sinon le bootstrap n'a pas grand intérêt

déchiffrement, rechiffrement, somme et produit pour 3 différents paramètres de sécurité $\lambda \in \{10, 80, 128\}$ (10 étant un example jouet, 80 le nombre de bits de sécurité requis pour l'embarqué à bas coût et 128 pour les applications modernes sur ordinateur classique).

Pour 10 bits de sécurité, il en faut en moyenne quasiment une heure pour générer les clés, 2 minutes 30 pour chiffrer un bit, 1 milliseconde pour faire la somme de deux chiffrés, 537 millisecondes pour faire le produit de deux chiffrés, 594 millisecondes pour déchiffrer et 5 minutes pour rechiffrer. Si l'on souhaite 80 bits de sécurité, il faudra 2 heures 20 pour générer les clés, 2 minutes 40 pour chiffrer un bit, 1 milliseconde pour additionner deux bits entre eux, 541 millisecondes pour en faire le produit, 596 millisecondes pour le déchiffrer, et 40 minutes pour le rechiffrer. Alors que pour 128 bits, il faudra 3 heures 20 pour générer les clés, presque 3 minutes pour chiffrer un bit, 1 millisecondes pour faire la somme de deux bits, 583 millisecondes pour faire le produit de deux bits, quasiment 600 millisecondes pour le déchiffrer et 1 heure 40 pour le rechiffrer. On constate que les deux opérations les plus coûteuses sont le rechiffrement et la génération des clés. Ce sont aussi les deux opérations sur lesquelles l'augmentation du paramètre de sécurité influe le plus (pour les autres opérations le temps est pratiquement constant). En effet, lorsque λ grandit, la difficulté des problèmes **SVP** et **CVP** augmente aussi et donc fait grandir la taille des clés afin que l'algorithme le plus efficace (Hermite-Korkine-Zolotarev [19] pour **SVP** et **CVP** puisque leur difficulté est la même d'après [14]) coûte au moins 2^λ opérations.

3 État de l'art

3.1 Le cryptosystème Brakerski-Gentry-Vaikunthanathan (BGV)

Après les travaux de Gentry en 2009, quelques cryptosystèmes ont vu le jour s'inspirant de celui d'origine basé sur les réseaux euclidiens.

Le cryptosystème BGV pour Zvika Brakerski, Craig Gentry et Vinod Vaikuntanathan [8] consiste en une redéfinition du système de Gentry en utilisant d'autres problèmes : Decisional Learning With Error (dLWE), Decisional Ring Learning With Error (dRLWE) et Decisional General Learning With Error (dGLWE). C'est ce schéma qui est implanté dans **HElib** (avec pas mal d'améliorations aujourd'hui).

Définition 10 ***(GLWE décisionnel)*** *Soient λ un paramètre de sécurité, $n = n(\lambda)$ une dimension entière, $f(x) = x^d + 1$ avec $d = d(\lambda)$ une puissance de 2, $q = q(\lambda) \geq 2$ un entier premier, $R = \mathbb{Z}[x]/(f(x))$, $R_q = R/qR$, $\chi = \chi(\lambda)$ une distribution sur R. Le problème $GLWE_{n,f,q,\chi}$ est de distinguer les deux distributions suivantes :*

- *un couple (a_i, b_i) pioché uniformément dans R_q^{n+1}*
- *un couple (a_i, b_i) tel que $a_i \leftarrow R_q^n$ uniformément, $b_i = \langle a_i, s_i \rangle + e_i$ avec $e_i \leftarrow \chi$, $s_i \leftarrow R_q^n$ uniformément.*

La supposition $GLWE_{n,f,q,\chi}$ est de dire que le problème $GLWE_{n,f,q,\chi}$ est difficile.

À partir du problème dGLWE (dLWE et dRLWE sont des cas particuliers de dGLWE), **Bakerski**, **Gentry**, et **Vaikuntanathan** construisent un "leveled" schéma totalement homomorphe mais cependant sans bootstrap (il ne permet d'évaluer que des circuit de hauteur inférieur à une valeur pré-définie à l'initialisation du schéma, mais pas de rechiffrer une donnée) :

BGV.KeyGen(λ, μ). Choisir un module q de taille λ bits et choisir les autres paramètres $(d = d(\lambda, \mu, b), n = n(\lambda, \mu, b), N = \lceil (2n + 1) \log q \rceil, \chi = \chi(\lambda, \mu, b))$ de manière à ce que le schéma soit basé sur une instance de GLWE fournissant λ bits de sécurité contre les attaques connues. Soit $R = \mathbb{Z}[x]/(x^d + 1)$ et noter $params = (q, d, n, N, \chi)$.

Tirer $s' \leftarrow \chi^n$ et fixer $sk = s \leftarrow (1, s'_1, \ldots, s'_n) \in R_q^{n+1}$. La clé privée est sk.

Générer une matrice $A' \leftarrow R_q^{N \times n}$ uniformément et un vecteur $e \leftarrow \chi^N$ et fixer $j \leftarrow A's' + 2e$. Soit A la matrice de $(n + 1)$ colonnes telle que la première soit j et le reste soit les n colonnes de $-A'$. La clé publique est $pk = A$.

BGV.Encrypt($params, pk, m$). Pour chiffrer un message $m \in R_2$, fixer $m' \leftarrow (m, 0, \ldots, 0) \in R_q^{n+1}$, tirer $r \leftarrow R_2^N$ et retourner le chiffré $c \leftarrow m' + A^T r \in R_q^{n+1}$.

BGV.Evaluate($pk, C, c_1, ..., c_t$). Étant donné le circuit [booléen] C avec t entrées et t chiffrés $c_1, ..., c_t$, appliquer la somme (lors d'un XOR) et le produit (lors d'un ET) dans R_q^{n+1} et retourner le résultat.

BGV.Decrypt$(params, sk, c)$**.** Retourner $m \leftarrow (\langle c, s \rangle \mod q) \mod 2$

Ce schéma autorise le chiffrement d'entiers de taille arbitraire, puisqu'il possède un paramètre p, entier positif premier, le modulo utilisé par le schéma. Ainsi, si $p = 2$ toutes les opérations seront effectuées modulo 2 (on chiffre donc des bits), si $p = 7$ alors modulo 7, ... Néanmoins, plus on s'autorise un grand p plus les performances du schéma faiblissent.

Une autre notion intéressante proposée par ce schéma est le système de batching, c'est à dire qu'il permet d'effectuer une même opération sur un ensemble de chiffrés (un vecteur dont les composantes sont des messages de l'espace des textes chiffrés) pour un coût équivalent. Il est donc possible par exemple de calculer le produit de plusieurs couples de chiffrés en un temps similaire à si on le calculait pour un seul couple. Les composantes dudit vecteur sont appelés les "slots".

Avec **BGV**, on ne choisit pas réellement le nombre de slots que l'on souhaite utiliser, c'est un paramètre qui dépend du contexte (les paramètres m,p,r,L dans l'implantation HElib), et donc il n'est pas possible de le changer pour un contexte donné. Ce qui explique pourquoi dans le tableau récapitulatif de fin la taille du chiffré pour 80 bits est plus faible que pour 10 bits de sécurité.

C'est par la suite, en Janvier 2015, que Shai Halevi et Victor Shoup introduisent le bootstrapping dans **HElib** [16], permettant d'obtenir un schéma totalement homomorphe. L'idée est de changer le modulo utilisé dans le chiffré pour en choisir un autre, de plus grande taille mais qui conserve la correction du schéma (le déchiffrement est toujours fonctionnel) et qui réduit le bruit du chiffré. C'est ce schéma que l'on décrit et nomme **BGV** dans la suite du rapport.

3.2 Le schéma van Dijk-Gentry-Halevi-Vaikuntanathan (DGHV)

Ce schéma est apparu dans une publication en 2010 [25] par **Marten Van Dijk**, **Craig Gentry**, **Shai Halevi** et **Vinod Vaikuntanathan**. Le but est de construire un cryptosystème simple basé sur les entiers, et plus précisément l'arithmétique modulaire (sommes et produits modulo un entier).

Les problèmes sur lequels est construit ce schéma s'appellent Sparse Sub-

set Sum Problem (SSSP) et Approximate Greatest Common Divisor (AGCD).

Définition 11 ***(SSSP).*** *Soit $A = \{a_1, a_2, ..., a_n\}$ un grand ensemble d'entiers, $t, M \in \mathbb{Z}$. Le problème* ***SSSP*** *est de trouver un sous-ensemble S de A tel que la somme des entiers dans S vaille $t \pmod M$.*

SSSP est une version particulière du problème du sac à dos.

Définition 12 ***(AGCD).*** *Soient p un entier impair de taille η, γ un entier et $D_{\gamma,\rho}(p)$ la distribution telle que :*

$$D_{\gamma,\rho}(p) = \{\textit{choisir } q \leftarrow \mathbb{Z} \cap [0, 2^\gamma/p[, \; r \leftarrow \mathbb{Z} \cap]-2^\rho, 2^\rho[, \textit{ retourner } x = pq + r\}$$

$D_{\gamma,\rho}(p)$ est un distribution qui génère des "presque" multiples de p.

Le problème (ρ, η, γ)-agcd est de retrouver p ayant un nombre polynomial d'éléments de $D_{\gamma,\rho}(p)$ pour p un nombre aléatoire donné, entier impair de taille η.

SSSP est utilisé dans la génération de la clé privée (voir ci-dessous). La difficulté du problème **SSSP** nous garantit qu'un attaquant ne sera pas capable de retrouver l'ensemble des u_i dont la somme vaut $x_p \pmod{2^{\kappa+1}}$. **AGCD** est également utilisé dans la génération des clés pour tirer les x_i. Sa difficulté nous garantit qu'un attaquant ne sera pas capable de retrouver le secret p.

Maintenant, expliquons le choix des paramètres de ce schéma. Notons γ la taille des x_i, η la taille du secret initial, ρ la taille du bruit r_i, τ le nombre de x_i dans la clé publique, ρ' la taille du deuxième bruit r utilisé dans le chiffrement, κ la taille des u_i, θ le poids de Hamming de la clé secrète, Θ la taille de la clé secrète.

Avant toute chose, il faut savoir que la sécurité asymptotique ($\omega()$) est vue de deux manières : celle des théoriciens et celle des praticiens. Les théoriciens estiment que le schéma est sûr si l'on ne peut pas l'attaquer en temps polynomial en λ, donc que λ correspond à une sécurité superpolynomiale en λ (c'est à dire $2^{\omega(log\lambda)}$). Les praticiens quant à eux demandent que λ représente les bits de sécurité donc que λ corresponde à une sécurité en 2^λ. C'est pourquoi ci-dessous le paramètre ρ n'a pas la même valeur asymptotique ($\omega(\lambda)$) que dans la publication originale ($\omega(log\lambda)$).

Afin de se protéger de l'attaque par force brute sur le bruit ρ, il faut choisir $\rho = \omega(\lambda)$ (cf. Section 3 - [25]). Rendre le schéma totalement homomorphe nécessite d'être capable d'évaluer le circuit de déchiffrement, il faut donc pouvoir évaluer des circuit d'au moins sa profondeur, d'où $\eta \geq \rho.\Theta(\lambda log^2\lambda)$ (cf. Section 6.2 - [25]). Pour résister aux attaques connues sur les réseaux adaptées au problème AGCD, il faut que $\gamma = \omega(\eta^2\lambda)$ (cf. Section 5 - [25]). Aussi, pour pouvoir utiliser le **Leftover Hash Lemma** dans la réduction vers AGCD, τ doit satisfaire $\tau \geq \gamma + \omega(\lambda)$ (cf. Lemme 4.3 - [25]). Ensuite, il nous faut résister aux attaques force brute sur la clé privée, donc $\theta = \omega(\lambda)$ et choisir la taille Θ de la clé secrète suffisamment grande, c'est à dire $\Theta = \omega(\kappa.\lambda)$ (cf. 6.3 - [25]).

Voici les exemples donnés dans [2] pour $\lambda \in \{52, 62, 72\}$ bits de sécurité : $\gamma \in \{843033, 4251866, 19575950\}$, $\eta \in \{1558, 2128, 2698\}$, $\rho \in \{41, 56, 71\}$, $\tau \in \{572, 2110, 7659\}$, $\rho' \in \{189, 259, 328\}$, $\kappa \in \{843071, 4251903, 19575999\}$, $\theta \in \{15, 15, 15\}$, $\Theta \in \{555, 2070, 7965\}$. Pour des raisons de performances, certains de ces paramètres ne correspondent pas à leur description en théorie.

Le schéma devient donc le suivant (après l'avoir rendu "fully" homomorphe) :

KeyGen(λ). Soit p est un entier impair de taille η : $p \leftarrow]2\mathbb{Z} + 1[\cap [2^{\eta-1}, 2^{\eta}[$.

Pour la clé publique, tirer $x_i \leftarrow D_{\gamma,\rho}(p)$ pour $i = 0, ..., \tau$. Renommer afin que x_0 soit le plus grand. Recommencer jusqu'à ce que x_0 soit impair et $x_0 \pmod p$ soit pair. Notons $pk^* = \langle x_0, x1, ..., x_\tau \rangle$. Soit $x_p \leftarrow \lfloor 2^\kappa / p \rceil$.

Choisir aléatoirement un vecteur $\vec{s} = \langle s1, ..., s_\Theta \rangle$ de Θ bits et de poids de Hamming θ. Notons $S = \{i : s_i = 1\}$. Choisir aléatoirement des entiers $u_i \in [0, 2^{\kappa+1}[$ sous la condition que $\sum\limits_{i \in S} u_i = x_p \pmod{2^{\kappa+1}}$. Notons $y_i = u_i / 2^\kappa$ et $\vec{y} = \{y_1, ..., y_\Theta\}$.

La clé privée est $sk = \vec{s}$ et la clé publique est $pk = (pk^*, \vec{y})$.

Encrypt(pk,$m \in 0, 1$). $pk = \langle x_0, x_1, ..., x_\tau \rangle$. Choisir un sous-ensemble $S \subseteq 1, 2, ..., \tau$ et un entier aléatoire $r \in [-2^{\rho'}, 2^{\rho'}]$ et notons $c^* \leftarrow (m + 2r + 2\sum\limits_{i \in S} x_i) \bmod x_0$.

Pour $i \in \{1, ..., \Theta\}$ calculer $z_i \leftarrow (c^* \times y_i) \pmod 2$ en ne conservant que $n = \lceil log\theta \rceil + 3$ bits de précision après la virgule flottante pour

chaque z_i.

Retourner c^* et $\vec{z} = \langle z_1, ..., z_\Theta \rangle$.

Evaluate(pk,C**,**$c_1, ..., c_t$**)**. Étant donné le circuit [booléen] C avec t entrées et t chiffrés $c_1, ..., c_t$, appliquer la somme (lors d'un XOR) et la multiplication (lors d'un ET) dans $\mathbb{Z}$ sur les chiffrés, et retourner le résultat.

Ensuite, effectuer la dernière étape de **Encrypt** pour mettre à jour les z_i.

Decrypt($\vec{s}$**,**c^***,**$\vec{z}$**)**. Retourner $m' \leftarrow (c^* - \lfloor \sum_i s_i z_i \rceil) \pmod 2$.

L'inconvénient de ce schéma est la taille de la clé publique : environ 800 MB pour la clé publique avec un paramètre de sécurité de 72 bits d'après [10]. En effet, pour calculer la taille de la clé publique il suffit de compter le nombre de x_i qui la compose (τ) et de le multiplier par la taille de chaque x_i (γ) : dans ce schéma la taille de la clé publique est donc $\tau \times \gamma$. Ce qui donne pour 72 bits de sécurité 7659×19575950 bits = 18741 MegaBytes. Mais avec une amélioration exposée dans [9] les auteurs arrivent à réduire la taille de la clé publique à 800 MB environ[3]. Une augmentation de la taille des clés entraîne une augmentation du nombre de calculs effectués lors d'une opération (on voit bien que si le nombre de x_i augmente, la fonction **Encrypt** nécessite davantage de calculs), de facto des opérations plus lentes et un schéma moins performant (pour le chiffrement, la somme, le produit, le déchiffrement et le rechiffrement puisque les clés sont déjà générées).

3.3 Le schéma Coron-Naccache-Tibouchi (CNT)

Afin d'améliorer le schéma **DGHV**, Jean-Sébastien Coron, David Naccache et Mehdi Tibouchi (**CNT**) [10] publient en 2012 un nouveau schéma réduisant la taille de la clé publique à 10 MB, soit un facteur 80 comparé à DGHV. Leur variante reste tout de même sémantiquement sûre, mais dans le

3. la taille de la clé est $2\gamma(\beta + \sqrt{\Theta} + 1)$ bits avec $\beta = 88$, $\Theta = 7897$ et $\gamma = 19 \times 10^6$ pour 72 bits de sécurité

modèle de l'oracle aléatoire (alors que **DGHV** est sémantiquement sûr dans le modèle standard). Ils améliorent également les précédentes attaques sur **AGCD**.

La description complète du schéma **CNT** est la suivante :

CNT.KeyGen(λ). Soit p un entier premier de taille η . Tirer aléatoirement un nombre impair $q_0 \in [0, 2^\gamma/p)$. Noter $x_0 = q_0 \times p$. Initialiser un générateur pseudo-aléatoire f_1 avec un graine aléatoire se_1. Avec $f_1(se_1)$ générer un ensemble d'entiers $\chi_i \in [0, 2^\gamma)$ pour $1 \leq i \leq \tau$. Pour $1 \leq i \leq \tau$ calculer :

$$\delta_i = (\chi_i \mod p) + \xi_i \times p - r_i$$

où $r_i \leftarrow \mathbb{Z} \cap (-2^\rho, 2^\rho)$ et $\xi_i \leftarrow \mathbb{Z} \cap [0, 2^{\gamma+\eta}/p)$.

Noter $pk^* = (se_1, x_0, \delta_1, ..., \delta_\tau)$ et $x_i = \chi_i - \delta_i$.

Générer aléatoirement un vecteur binaire s de taille Θ et de poids de Hamming θ sous les conditions que $s_1 = 1$, que pour tout $k \in [0, \theta)$ il y a au plus 1 bit non nul parmi les s_i pour $k \times B + 1 \leq i < (k + 1) \times B + 1$ avec $B = \lfloor \Theta/\theta \rfloor$.

Initialiser un deuxième générateur pseudo-aléatoire f_2 avec une graine aléatoire se_2 puis avec $f_2(se_2)$ générer des entiers $u_i \in [0, 2^{\kappa+1})$ pour $2 \leq i \leq \Theta$ avec $\kappa = \gamma + n + 2$. Ensuite fixer u_1 tel que :

$$\sum_{i=1}^{\Theta} s_i \times u_i = x_p \mod 2^{\kappa+1}$$

avec $x_p = \lfloor 2^\kappa/p \rceil$.

Initialiser un troisième générateur pseudo-aléatoire f_3 avec une graine aléatoire se_3. Générer avec $f_3(se_3)$ des entiers $\chi'_i \in [0, 2^\gamma)$ pour $1 \leq i \leq \Theta$. Générer aléatoirement des entiers $r'_i \in (-2^\rho, 2^\rho)$ et $\xi'_i \in [0, 2^{\gamma+\eta}/p)$ et soit :

$$\delta'_i = (\chi'_i \mod p) + \xi'_i \times p - 2r'_i - s_i$$

Le chiffré σ (utilisé lors du rechiffrement) de la clé publique s est donc définie par :

$$\sigma_i = \chi'_i - \delta'_i$$

Finalement, retourner la clé secrète $sk = s$ et la clé publique $pk = (pk^*, se_2, u_1, se_3, \delta')$ avec $\delta' = (\delta'_1, ..., \delta'_\Theta)$ et σ.

CNT.Encrypt$(pk, m \in \{0,1\})$**.** Retrouver les entiers x_i avec pk^* comme suivant : initialiser f_1 avec se_1, générer les χ_i et calculer les $x_i = \chi_i - \delta_i$.

Choisir un vecteur aléatoire $b = \underset{1 \leq i \leq \tau}{b_i} \in [0, 2^\alpha)^\tau$ et un entier aléatoire $r \in (-2^{\rho'}, 2^{\rho'})$.

Retourner le chiffré $c^* = m + 2r + 2\sum_{i=1}^{\tau} b_i \times x_i \mod x_0$.

CNT.Add(pk, c_1^*, c_2^*)**.** Retourner $(c_1^* + c_2^*) \mod x_0$

CNT.Mult(pk, c_1^*, c_2^*)**.** Retourner $(c_1^* \times c_2^*) \mod x_0$

CNT.Expand(pk, c^*)**.** Pour $1 \leq i \leq \Theta$ regénérer les u_i avec $f_2(se_2)$ puis calculer $y_i = u_i / 2^\kappa$. Calculer les z_i tels que :

$$z_i = (c^* \times y_i) \mod 2$$

en ne conservant que $n = \lceil log_2(\theta + 1) \rceil$ bits de précision après la virgule flottante.

Retourner $c = (c^*, z)$ avec $z = (z_1, \ldots, z_\Theta)$.

CNT.Decrypt(sk, c)**.** Retourner $m = (c^* - \lfloor \sum_{i=1}^{\Theta} s_i \times z_i \rceil) \mod 2$

CNT.Recrypt(pk, c)**.** Recalculer les σ_i depuis pk puis appliquer le circuit de déchiffrement au chiffré "étendu" z avec la clé secrète chiffrée σ. Retourner ce résultat comme chiffré rafraîchit.

Comme énoncé précédemment, ce schéma offre un facteur 80 comparé à **DGHV** puisqu'il utilise 3 générateurs pseudo-aléatoires dont il ne conserve que la graine dans la clé publique. Ainsi avec f_1 il conserve se_1 pour générer

les δ_i de taille γ au nombre de τ, soit un facteur $\tau \times \gamma$. Avec f_2 il ne conserve que se_2 pour générer les u_i de taille $\kappa + 1$ en quantité Θ soit un facteur $\Theta \times (\kappa + 1)$. Enfin, avec f_3 il ne conserve que se_3 pour générer les δ_i' de taille γ en quantité Θ offrant un facteur $\Theta \times \gamma$.

3.4 Ducas-Micciancio (DM)

Les deux principaux goulots d'étranglement des schémas homomorphes sont la génération des clés et le rechiffrement (Recrypt). On remarque par exemple que BGV peut prendre une dizaine de minutes pour rechiffrer un chiffré entre 80 et 128 bits de sécurité. Pour autant la génération des clés reste quand même largement abordable comparée au rechiffrement, mais consacrer 1 minute à une opération que l'on exécutera qu'une fois (génération des clés) reste raisonnable.

En 2015 paraît une publication [11] proposant un schéma de chiffrement homomorphe dont le rechiffrement prend moins d'une seconde, et dont le code source est disponible publiquement [3]. Il faut noter que cette opération prend moins d'une seconde à être exécutée mais sur un unique chiffré, alors que BGV, lui, prend 7 minutes sur des chiffrés "batchés" (pour 80 bits par exemple, le batching peut contenir jusqu'à 756 chiffrés, ce qui fait donc une demi-seconde par chiffré). C'est un schéma différent pour des besoins différents, mais ce n'est pas une amélioration de BGV. Néanmoins au lieu de commuter avec l'opération "+" ("AND") et "×" ("XOR"), il commute avec l'opération "NAND" (a NAND $b = 1 - a \times b$).

La porte NAND est universelle, c'est à dire que toutes les fonctions logiques peuvent être écrites à l'aide uniquement de portes NAND (tout comme la porte NOR). Ainsi, la multiplication binaire étant une porte "AND", elle s'écrit $NAND(NAND(A, B), NAND(A, B))$, et l'addition binaire étant le "XOR", il s'écrit $NAND(NAND(A, NAND(A, B)), NAND(B, NAND(A, B)))$. Un "AND" coûte 2 appels à "NAND" ($NAND(A, B)$ est appelé 2 fois) tandis qu'un "XOR" coûte 4 appels à "NAND" ($NAND(A, B)$ est appelé également 2 fois). On constate alors que contrairement aux autres schémas de chiffrements homomorphes, la somme est plus coûteuse que le produit de deux chiffrés.

Tout comme **BGV**, ce schéma repose lui aussi sur le problème **LWE**.

Il semble possible d'après les auteurs d'étendre ce schéma à d'autres opérations que le NAND, et ainsi limiter le nombre de rechiffrements pour des opérations de base. Ici on fait appel à 2 et 4 rechiffrements, alors qu'il est

possible de ne faire appel qu'à 1 rechiffrement si l'on ajoute le support de l'addition et de la multiplication binaire au schéma.

4 Implantation

Cette partie servira à exposer les particularités et difficultés à implanter **DGHV** et à adapter **CNT** pour les expériences. Les tests et les calculs ont été effectués sur les serveurs du GREYC (https://portail.greyc.fr/fr/ressources/calcul), essentiellement sur Jet ayant 2 processeurs Intel Xeon E5-2680 v3 48 cœurs cadencés à 2,5GHz et 512Go et fonctionnant sous Ubuntu.

Afin de calculer le temps d'exécution de certaines instructions en **Sage**, il est pratique d'utiliser la fonction *cputime()* disponible directement dans **Sage**. Ainsi pour stocker le temps dans une variable t et ensuite connaître le temps depuis t, il suffit de faire :

```
sage: t = cputime()
sage: <Instruction longue>
sage: temps_exec = cputime(t)
```

Cette fonction étant très pratique mais non disponible en C++, il convient de définir une fonction ayant un comportement équivalent de cette manière :

```
#include <time.h>
#include <sys/time.h>

double cputime(const double before=0){
        return ((double)clock() / CLOCKS_PER_SEC) - before;
}
```

4.1 DGHV

Le langage utilisé est **Python**. Nous avons premièrement déterminé les paramètres du schéma en fonction du paramètre de sécurité d'après 3.2 et [25] que[4] $\rho = \theta = \lambda$, $\rho' = \Theta = 2\lambda$, $\eta = \lambda^2\lceil log_2(\lambda^2)\rceil$, $\gamma = \lambda^5$, $\tau = \gamma + \lambda$ et $\kappa = \gamma\eta/\rho'$.

Nous exposons maintenant des détails d'implantation qui peuvent être utiles pour quiconque souhaiterait implanter des schémas.

4. Même si [2] a en pratique des paramètres différents

DGHV.KeyGen demande de générer aléatoirement $\vec{s} = \langle s_1, ..., s_\Theta \rangle$ avec un poids de Hamming de θ. La manière la plus directe de construire un tel vecteur est celle du rejet : on génère aléatoirement un vecteur de taille Θ et s'il n'est pas de poids de Hamming θ on recommence. La probabilité de réussite pour un essai est $2^{-\Theta}\binom{\Theta}{\theta}$ (probabilité d'avoir uniquement θ 1 indépendamment de l'ordre). Une manière plus efficace est de construire $\vec{s}$ tel que pour $i \in \{1, ..., \theta\}$, $s_i = 1$ et $s_i = 0$ pour le reste. Ensuite, mélanger aléatoirement $\vec{s}$.

Ensuite, comment s'assurer que $\sum_{i \in S} u_i = x_p \pmod{2^{\kappa+1}}$? Similairement, la méthode du rejet n'est pas efficace. Il suffit toutefois d'en générer $i - 1$ et de choisir le dernier de telle sorte que la propriété soit vérifiée, c'est à dire choisir les u_i pour $i \neq 1$ aléatoirement [5], ainsi $u_1 = x_p - \sum_{i \in S - \{1\}} u_i \pmod{2^{\kappa+1}}$ (notion de degré de liberté).

Un point non négligeable quand on travaille sur les schémas basés sur les entiers : la précision. Dans **DGHV.Encrypt** et **DGHV.Evaluate**, les z_i sont calculés en ne conservant que $n = \lceil log\theta \rceil + 3$ bits de précisions. Or **Python** lui ne gère que jusqu'à 53 bits de précision, alors le schéma n'est plus correct pour des grands paramètres (lorsque θ a une taille plus grande que 50). C'est pourquoi il convient premièrement d'intégrer une librairie de gestion de la virgule flottante plus précise : **BigFloat**. Elle permet une gestion de la précision de manière infinie, c'est à dire qu'il n'y a pas de limite concernant le nombre de bits de précision que l'on veut conserver après un calcul.

Par exemple, **BigFloat** peut calculer une racine carrée avec un certain nombre de bits de précision (en comparant avec le *sqrt* de python) :

```
>>> from bigfloat import sqrt as bfsqrt, precision as bfprecision
>>> from math import sqrt as pysqrt
>>> pysqrt(26) # racine carrée classique en Python
5.0990195135927845
>>> bfsqrt(26) # par défaut bigfloat utilise les 53 bits de Python
BigFloat.exact('5.0990195135927845', precision=53)
>>> with bfprecision(100): # on fixe un contexte dans lequel 100
    bits de précision sont utilisés
...     bfsqrt(26)
...
BigFloat.exact('5.0990195135927848300282241090254', precision=100)
```

Tout de même, plus une grande précision est demandée à **BigFloat** et plus les calculs prennent de temps. Il est par exemple (pour ce schéma)

5. en supposant que $1 \in S$, sinon choisir i aléatoirement dans S

impossible d'effectuer des opérations sur les chiffrés lorsque $\lambda \geq 10$ car les opérations peuvent prendre jusqu'à 30 minutes pour une simple génération de clés (**BGV** en prend autant pour $\lambda \gg 128$).

Heureusement il existe une alternative à **Python** pour faire des calculs de mathématiques poussées : **Sage** [5]. Sage permet d'écrire du code en Python, de l'interpréter avec **iPython** [20] et d'exécuter des primitives calculatoires écrites puis compilées en C++, et utilise notamment la librairie GNU Multi-Precision (GMP) ce qui rend les calculs très rapide.

L'avantage de **Sage** et **GMP** comparés à **Python** et **BigFloat** est que Sage repose en plus grande partie sur des librairies mathématiques compilées machine, contrairement au combo **Python**/**BigFloat** où uniquement **BigFloat** fait appel à du C++ pour les calculs. De plus, **Sage** propose des fonctions pour générer des éléments depuis les ensembles tels que $\mathbb{Z}$ ou $\mathbb{R}$ en utilisant la fonction :

```
sage: borne_inf, borne_sup = (0, 2^16)
sage: ZZ.random_element(borne_inf, borne_sup) # un élément dans Z
49665
sage: RR.random_element(borne_inf, borne_sup) # un élément dans R
26267.5144081688
sage: QQ.random_element(borne_inf, borne_sup) # un élément dans Q
2/593
```

Aussi, **Sage** permet de sauvegarder et charger des objets sur et depuis le disque dur de façon très simple :

```
sage: nom_fichier = 'private.key'
sage: save(sk, nom_fichier) # écrit dans le fichier private.key.sobj
    l'objet sk
...
sage: sk = load(nom_fichier) # charge depuis private.key.sobj l'
   objet sk
```

C'est une fonction très pratique quand on sait que la génération des clés pour un tel schéma prend plusieurs minutes : on ne les génère qu'une fois pour les sauvegarder sur le disque puis on les charge depuis ce dernier les fois suivantes.

Grâce à Sage, il a été possible d'augmenter le paramètre de sécurité jusqu'à 40, au delà duquel la génération de clés pouvait prendre plus de 30 minutes[6]. Comme indiqué dans le TABLEAU 7 pour 10 bits de sécurité la

6. Certes 30 minutes restent raisonnables pour la génération de clés, mais pas pour 40

génération des clés prend 1 minute 40, alors que pour 80 bits de sécurité la fonction de génération des clés même après une semaine de calcul sur les serveurs du GREYC n'a pas terminé (et une erreur de taille mémoire qui a obligé le système d'exploitation à tuer le processus **Sage**. Sa faiblesse calculatoire a poussé d'autres chercheurs à construire une amélioration permettant d'augmenter la sécurité raisonnablement de ce schéma.

4.2 CNT

En même temps que d'avoir publié [10], les auteurs ont écrit du code en **Sage** mettant en œuvre le schéma **CNT**, disponible à cette adresse [2].

Notre implantation de ce schéma en **Sage** n'était pas à la hauteur de celle de ses auteurs. Le rechiffrement (boostrapping) étant une opération très lente, le schéma en Sage n'était pas utilisable. En réalité, l'implantation du schéma par les auteurs comporte une partie du code en C++ : Sage permet d'écrire du code en **Cython** (un langage combinant la simplicité de Python et la puissance de C/C++). Une des parties les plus lentes lors du rechiffrement est le produit scalaire (utilisé dans le déchiffrement, évalué de façon homomorphe), c'est donc le produit scalaire qui a été écrit et optimisé pour rendre le rechiffrement acceptable en temps.

Une erreur d'implantation qui rendait le schéma incorrect se trouvait dans le code source (le bruit ajouté au moment du chiffrement n'était pas doublé). Elle se trouve ligne 180 du fichier *dghv.sage*. Dans la publication du schéma **CNT** dans l'annexe A, le calcul du chiffré se fait ainsi "$c = m+2r+2\sum_{i=1}^{\tau} b_i \times x_i \mod x_0$" alors que dans l'implantation en **Sage** "$c = m + 2r + \sum_{i=1}^{\tau} b_i \times x_i \mod x_0$" était écrit (il manque le 2 devant la somme).

Puisque le schéma ne permet que de chiffrer que des bits (0 ou 1), il est intéressant de pouvoir représenter des entiers et d'effectuer des opérations entre eux. Il suffit de se rappeler des cours de machine numérique de la première année de licence pour se souvenir de comment représenter les entiers en binaire. Pour les entiers positifs, la représentation binaire classique suffit. Malgré tout, pour les entiers négatifs, la représentation en complément à 2 doit être utilisée.

Pour représenter des nombres entier positifs en binaire on utilise la notation binaire classique (décomposition binaire). Pour autant, elle ne permet pas directement de représenter les nombres entiers négatifs, c'est pourquoi

bits de sécurité

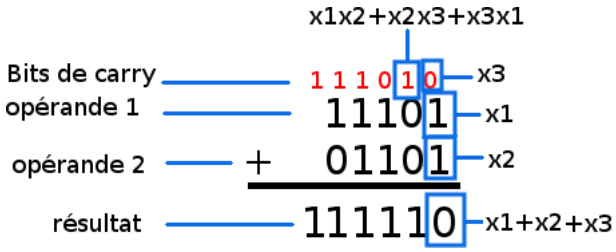

FIGURE 6 – Une addition binaire similaire aux additions entières que l'on apprend à l'école primaire

en premier lieu on introduit la notion de complément à 2 : le premier bit du nombre représente son signe (0 pour positif, 1 pour négatif). Ainsi, pour représenter un nombre négatif, il suffit de prendre la représentation binaire de son opposé (un nombre positif donc), d'inverser chacun de ses bits (y compris le bit de signe) et d'y ajouter la représentation binaire du nombre 1 (un 1 tout à droite complété avec que des 0 sur sa gauche). Cette représentation à l'avantage de ne pas requérir de changement dans l'architecture du processeur qui effectuera les additions (hormis de savoir calculer la représentation binaire d'un entier négatif).

Pour additionner 2 nombres chiffrés de 5 bits chacun, il faut environ 8 secondes tandis que pour les multiplier entre eux, il en faut 64 (pour 42 bits de sécurité). En effet, lors de l'addition bit à bit de deux nombres en représentation binaire, pour chaque paire de bits il faut sommer celui de l'opérande de gauche (x_1), celui de l'opérande de droite (x_2), et l'ancien bit de carry (x_3). Ensuite pour que le bit de carry vaille 1, il faut avoir au moins 2 bits de la colonne à 1, ce qui se résume par le polynôme $x_1x_2 + x_2x_3 + x_3x_1$ avec le produit comme un **XOR** et la somme comme un **OU** binaire.

4.3 HElib (BGV)

La plus fameuse des librairies de chiffrement homomorphe aujourd'hui met en œuvre le schéma **BGV** avec des améliorations apportées depuis la publication du schéma telles que le rechiffrement et le batching. Elle est écrite en C++ et utilise les librairies GMP et NTL.

La documentation de HElib (disponible sur github et [15]) donne les premières bases pour l'utiliser. Cependant, elle n'a pas été mise à jour depuis un certain temps tandis que des modifications ont été apportées à la librairie.

Il faut premièrement installer GMP et NTL, le mieux étant d'installer GMP avec le gestionnaire de paquet du système et NTL depuis les sources. Sur ce point la documentation de HElib est correcte.

Détaillons l'utilisation du schéma avec le boostrapping activé. Afin d'éviter de polluer le rapport, on omet volontairement certaines parties du code inhérente au langage qui ne portent aucune information concernant le schéma (par exemple, une boucle for pour copier des valeurs dans un tableau). Le code complet est disponible en annexe pour les personnes qui souhaiteraient se servir de la librairie.

Il faut ensuite créer un fichier du nom que l'on veut dans le dossier *src/* portant l'extension *.cpp* (par exemple *monprogramme.cpp*), puis inclure les entêtes dont nous allons nous servir :

```
#include "FHE.h"
#include "EncryptedArray.h"
#include "EvalMap.h"
```

Il va nous falloir instancier le schéma puis générer les clés. Il faut savoir que HElib ne fournit pas de calculs de paramètres en fonction du paramètre de sécurité souhaité. Tout de même elle met à disposition une liste de paramètres fonctionnels et il est possible d'estimer le paramètre de sécurité en fonction de ceux-ci, j'ai alors pour chaque paquet de paramètres calculé λ et conservé les plus intéressants ($\lambda = 80$ et $\lambda = 127$). Ces paramètres peuvent être retrouvés dans le fichier *Test_ bootstrapping.cpp* de la librairie. Aussi, le fichier *parameters.cpp* est aussi intéressant et permet de trouver des paramètres fonctionnels par la force brute.

```
int main(int argc, char *argv[])
{ /* FIXING VARIABLES */
long values[2][19] = {
               // 80 bits
               {2,21168,27305,28,43,635,0,
               10796,26059,0,42,18,0,100,1,
               25,23,3,64},
               { // 128 bits
                /*p=*/2,/*phim=*/26400,/*m=*/27311,
                /*d=*/55,/*m1=*/31,/*m2*/881,/*m3=*/0,
                /*g1=*/21145,/*g2=*/1830,/*g3=*/0,
                /*ord1=*/30,/*ord2=*/16,/*ord3=*/0,
                /*c_m=*/100,/*r=*/1,/*L=*/25,/*B=*/23,
                /*c=*/3,/*skHwt=*/64
                }
```

```
};
```

Pour discuter un peu des paramètres les plus intéressants quand il s'agit d'utiliser cette librairie, il faut savoir que p correspond au modulo pour les entiers que l'on veut représenter (toutes les opérations effectuées au sein du système seront modulo p). Cela correspond au nombre maximum (non inclut) que l'on peut chiffrer, c'est à dire la valeur maximale de chaque composante du vecteur (batching) que l'on chiffre. Ainsi, si $p = 2$ on ne peut représenter que 0 ou 1, les entiers de 1 à 12 si $p = 13$, etc. p doit être premier. Le paramètre L correspond à la hauteur maximale des circuits que l'on veut pouvoir évaluer, mais il ne prend pas en compte le degré multiplicatif du circuit, c'est à dire qu'il n'est pour le moment pas possible de savoir en fonction de L le nombre de multiplications successives qu'il est possible de faire avant d'essayer. Par exemple, pour $L = 25$ il est possible d'effectuer 2 multiplications au delà desquelles le bruit est trop important.

Initialisons le contexte :

```
/* COMPUTING CONTEXT AND PARAMETERS */
FHEcontext context(m, p, r, gens, ords);
context.bitsPerLevel = B;
buildModChain(context, L, c);
context.makeBootstrappable(mvec, 0, false);
context.rcData.skHwt = skHwt;

long nPrimes = context.numPrimes();
IndexSet allPrimes(0,nPrimes-1);
double bitsize = context.logOfProduct(allPrimes)/log(2.0);

long p2r = context.alMod.getPPowR();
context.zMStar.set_cM(mValues[13]/100.0);
```

On génère ensuite les clés :

```
FHESecKey secretKey(context);
FHEPubKey& publicKey = secretKey;
secretKey.GenSecKey(skHwt);   // Clé secrète de poids de Hamming
    skHwt
addSome1DMatrices(secretKey); // On calcule les matrice de
    changement de clé
addFrbMatrices(secretKey); // On calcule les matrice de changement
    de clé
secretKey.genRecryptData(); // Clé de rechiffrement pour le
    boostrapping
```

Chiffrons les données que l'on souhaite :

```
/* INIT SUPPORT */
EncryptedArray ea(context);
NewPlaintextArray plaintextarray(ea), plaintextarray2(ea);
Ctxt ciphertext(publicKey), ciphertext2(publicKey);
encode(ea,plaintextarray,0); // Encodage vectoriel du bit 0
encode(ea,plaintextarray2,1); // Encodage vectoriel du bit 1

/* BEGIN ENCRYPTION */
ea.encrypt(c1,publicKey,plaintextarray);
ea.encrypt(c2,publicKey,plaintextarray2);
ciphertext *= ciphertext2; // On stocke dans ciphertext le résultat
    du produit avec ciphertext2
publicKey.reCrypt(ciphertext); // On rechiffre ciphertext si on veut
```

Pour déchiffrer :

```
/* BEGIN DECRYPTION */
ea.decrypt(ciphertext,secretKey,plaintextarray2); // On stocke dans
    plaintextarray2 le déchiffré de ciphertext

/* CHECK RESULT */
vector<long> decrypted(ea.size());
decode(ea,decrypted,plaintextarray2); // On décode le vecteur
    plaintextarray2 pour le stocker dans decrypted
```

Enfin, pour compiler et exécuter le tout, il suffit d'incanter la commande magique suivante :

```
$ make monprogramme_x && ./monprogramme_x
```

On peut remarquer par la même occasion que HElib permet non pas de chiffrer des données une à une, mais se sert d'un vecteur complet pour porter les données. Ainsi, il est possible d'appliquer toute une série d'opérations similaires sur plusieurs chiffrés en n'utilisant qu'un unique vecteur (ici *NewPlaintextArray*) : cette technique s'appelle le Batching [23]. Cela n'exclut évidement pas l'utilisation classique pour ne chiffrer qu'une valeur : il suffit de se servir de la première composante et d'ignorer les autres. Le batching permet un gain en performances non-négligeable.

HElib propose d'être exécuté en mode multi-threads. Il faut compiler NTL [7] et HElib [8] avec les flags multithread. Les opérations qui profitent du

7. -NTL_THREADS=on NTL_THREAD_BOOST=on
8. -pthread -DFHE_THREADS -DFHE_DCRT_THREADS

multi-threading sont les opérations de la classe **DoubleCRT** (telle que l'opération Fast Fourier Transform (FFT)) et le bootstrapping. Après avoir effectué des essais avec la version 1 thread, 4 threads et 16 threads sur les serveurs de calculs du GREYC, les résultats n'étaient pas concluants : pas d'amélioration notable en essayant divers paramètres de sécurité. Le multi-thread est encore au stade de développement, ce qui peut expliquer qu'aucun gain en temps notable n'est visible.

4.4 Duccas-Micciancio (DM)

Ce schéma qui propose un rechiffrement en moins d'une seconde, n'accepte comme opération homomorphe que le *NAND*. Afin de compiler le code source **FHEW** [3] (c'est le nom de l'implantation du schéma), il faut disposer de la librairie **FFTW3** permettant de calculer des transformées de Fourier rapides. Par exemple avec Debian Jessie, cette librairie s'installe avec les paquets *libfftw3-bin, libfftw3-dev, libfftw3-double3, libfftw3-long3, libfftw3-quad3, libfftw3-single3*. Il faut ensuite compiler le programme **FHEW** avec *make*.

Le fichier source *fhewTest.cpp* contient les instructions de base pour générer les clés, chiffrer, effectuer le NAND et déchiffrer. Le rechiffrement étant rapide, il est effectué systématiquement à la fin de chaque NAND par la librairie.

Afin de comparer les performances avec les autres schémas de la somme et du produit (AND et XOR binaires) de deux chiffrés, nous avons besoin de deux fonctions dans le fichier *FHEW.cpp* qui implantent le AND binaire et le XOR binaire tout deux avec des portes NAND :

```
// Homomorphic binary AND
void HAND(LWE::CipherText* res, const EvalKey& EK, const LWE::
    CipherText& ct1, const LWE::CipherText& ct2){
          LWE::CipherText gauche;
          HomNAND(&gauche,EK,ct1,ct2);
          HomNAND(res,EK,gauche,gauche);
}

//Homomorphic binary XOR
void HXOR(LWE::CipherText* res, const EvalKey& EK, const LWE::
    CipherText& a, const LWE::CipherText& b){
        LWE::CipherText c1,c2,c3;
```

-DFHE_BOOT_THREADS

```
        HomNAND(&c1,EK,a,b);
        HomNAND(&c2,EK,a,c1);
        HomNAND(&c3,EK,b,c1);
        HomNAND(res,EK,c2,c3);
}
```

Le produit (AND) est composé de 2 portes NAND tandis que la somme (XOR) en contient 4, ce qui explique que la somme est plus lente que le produit à l'exécution. La taille de la clé publique est en théorie de 1Go (cf. Section 6.2 - [3]), mais en pratique elle est 2 fois plus grosse (2,4Go) car elle est représentée sous sa forme de transformée de Fourier rapide en double précision.

4.5 Performances

La TABLE 7 regroupe 5 schémas homomorphes testés par mes soins afin d'évaluer les temps en pratiques d'exécution des différents algorithmes qui les composent, avec différents paramètres de sécurités (10,80 et 128). Les cases vides signifient que les tests n'ont pas été possibles soit parce que l'implantation ne permettait pas de choisir le paramètre de sécurité, soit parce que l'algorithme prenait un temps extrêmement long. Pour pouvoir comparer à peu près équitablement les schémas, certaines approximations ont été introduite : par exemple, les temps d'exécution du schéma DM qui en réalité propose 100 bits de sécurité au lieu de 80. Aussi, tous les schémas ne sont pas codés dans le même langage, ce qui introduit encore un facteur supplémentaire qui biaise la comparaison.

On constate que le tout premier schéma totalement homomorphe, celui de **Gentry**, met un temps assez long à générer les paires de clés : plusieurs heures pour des paramètres raisonnables (80 bits au moins). Aussi, il est relativement long pour chiffrer un unique bit : quasiment 3 minutes pour 128 bits de sécurité. Le rechiffrement, opération connue pour être très coûteuse, s'exécute elle aussi en un temps de l'ordre de grandeur d'une heure voire 1 heure 30 pour 128 bits. BGV est quant à lui bien plus intéressants : l'augmentation du niveau de sécurité n'a de grand impact que sur le rechiffrement (la génération des clés n'est pas sensiblement plus longue, ainsi que le chiffrement, déchiffrement, la somme et le produit) même si pour 128 bits de sécurité il faut 29 minutes en moyenne pour rechiffrer 480 slots de chiffrés. DM lui est construit de telle sorte que le rechiffrement s'exécute en moins d'une seconde, et fait de lui le schéma le plus intéressant s'il on doit souvent rechiffrer. Toutefois la porte logique avec laquelle il commute (NAND) en-

traîne que la somme de 2 bits se fait en 2 secondes compte tenu du nombre de portes NAND (4) que possède la somme binaire, ce qui rend le produit de 2 chiffrés plus rapide car composé de moins de portes NAND (2). Son temps de génération des clés reste tout de même raisonnable comparé à Gentry et BGV. Concernant l'implantation de DGHV, la génération des clés est très lente : rien que pour 10 bits de sécurité cela prend plus d'une minute 30, et il n'a pas été possible d'aller plus loin que ce niveau de sécurité. Les autres opérations, hormis le produit, restent raisonnables en terme de temps d'exécution. Enfin CNT reste assez rapide concernant la génération des clés, jusqu'à 6 minutes pour 80 bits de sécurité, mais pas au delà (plus d'une semaine de calcul et le processus s'est fait tuer pour avoir utilisé trop de mémoire). 1 minute pour chiffrer un bit, 4 pour le rechiffrer c'est encore abordable compte tenu du temps que prennent la somme et le produit de deux chiffrés.

Même si son temps pour générer les clés n'est pas le plus intéressant, HElib (BGV) reste selon moi l'implantation la plus efficace dans sa généralité. Elle permet à la fois de gérer de grands paramètres de sécurité (128 bits) et de rechiffrer en temps suffisamment raisonnable. Générer les clés est une opération que l'on effectue en général qu'une seule fois sur le long terme, on peut donc accepter qu'une telle opération soit relativement lente (il ne faudrait pas non plus qu'elle dure une journée à mon sens).

Schéma \ algo	Génération des clés			Chiffrement			Somme		
λ	10	80	128	10	80	128	10	80	128
[4] Gentry (C++, 2009)	53m	140m (132m)	201m	153s	164s	177s	1033μs	1051μs	1074μs
[1] BGV (C++, 2014)	47s	57s	65s	742ms	836ms	890ms	920μs	2072μs	2593μs
[3] DM (C++, 2014)	λ fixe	13s	λ fixe		1s			2s	
DGHV (Sage, 2010)	102s	Beaucoup	Très beaucoup	1s			985μs		
[2] CNT (Sage, 2012)	40ms	6m	Très beaucoup	2ms	61s		20μs	3ms	

Schéma \ algo	Produit			Déchiffrement			Rechiffrement		
λ	10	80	128	10	80	128	10	80	128
[4] Gentry (C++, 2009)	537ms	541ms	583ms	594ms	596ms	598ms	5m	40m (31m)	102m
[1] BGV (C++, 2014)	790ms	1037ms	1096ms	204ms	328ms	359ms	15,9m	16,2m	29m
[3] DM (C++, 2014)		1s			1s			(0,5s)	
DGHV (Sage, 2010)	8s			1s			150ms		
[2] CNT (Sage, 2012)	30μs	830ms		193μs	20ms		150ms	4m	

Schéma	Taille des clés			Taille du chiffré (normalisé)		
λ	10	80	128	10	80	128
[4] Gentry (C++, 2009)	1,4Go	7Go	15Go	3,7Mo	3,7Mo	3,7Mo
[1] BGV (C++, 2014)	920Mo	1,4Go	2,0Go	14,3Ko [9]	10Ko [10]	19,8Ko [11]
[3] DM (C++, 2014)		2,4Go			2Ko	
DGHV (Sage, 2010)	1,3Go	(802Mo)		14Ko		
[2] CNT (Sage, 2012)	22Ko	17Mo		23Ko	21Mo	

FIGURE 7 – Comparaison des différents schémas homomorphes en fonction de l'algorithme et du niveau de sécurité. $\lambda = 10$ étant un exemple jouet, $\lambda = 80$ le niveau de sécurité demandé aux objets embarqués et $\lambda = 128$ le niveau de sécurité conseillé de nos jours. Entre parenthèses les résultats tirés de la thèse de Gentry (pour la génération des clés et le rechiffrement) et la taille de la clé publique d'après la publication de DGHV (le reste étant des résultats pratiques), puis le résultat théorique du rechiffrement d'après la publication de DM.

9. 4,5Mo/320 slots
10. 7,4Mo/756 slots
11. 9,3Mo/480 slots

5 Conclusion

Depuis les travaux de Craig Gentry en 2009, le domaine du chiffrement homomorphe est en plein essor, c'est même un phénomène de mode. Beaucoup de publications ont vu le jour mettant à disposition différentes variantes du schéma d'origine basé sur les réseaux. Malgré cela les schémas proposés sont encore trop peu efficaces en pratique pour être utilisables à grande échelle. Des applications telles que le vote électronique sont fonctionnelles mais ne permettent pas de voter un grand nombre de fois sans rechiffrer.

Glossaire

AGCD Approximate Greatest Common Divisor. 16

Batching Aussi appelée SIMD (*Simple Instruction on Multiple Data*). Technique qui permet d'agréger plusieurs chiffrés pour leur appliquer la même opération. En général elle permet un gain en efficacité et est donc plus rapide que d'appliquer l'opération individuellement sur les chiffrés. 29

CEA Commissariat à l'énergie atomique et aux énergies alternatives. 5

dGLWE Decisional General Learning With Error. 13, 14

dLWE Decisional Learning With Error. 13, 14

dRLWE Decisional Ring Learning With Error. 13, 14

FFT Fast Fourier Transform. 30

GMP GNU Multi-Precision. 24, 26, 27

NTL Une librairie pour effectuer des calculs sur la théorie des nombres. http://shoup.net/ntl/ . 26, 27

SSSP Sparse Subset Sum Problem. 15

Références

[1] Code source de BGV. https://github.com/shaih/helib.

[2] Code source de CNT. https://github.com/coron/fhe.

[3] Code source de DM. https://github.com/lducas/FHEW.

[4] Public Challenges for Fully-Homomorphic Encryption. http://researcher.watson.ibm.com/researcher/view_group.php?id=1548.

[5] System for Algebra and Geometry Experimentation.

[6] CH Bennett et G Brassard : BB84. page 175, 1984.

[7] Enguerran Bernard, Sergui Carpov et Patrick Lacharme : Démonstrateur du calcul homomorphe sur système Androïd. Mémoire de D.E.A., ENSICAEN, 2013/2014.

[8] Zvika Brakerski, Craig Gentry et Vinod Vaikuntanathan : (Leveled) fully homomorphic encryption without bootstrapping. *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 309–325, 2012.

[9] Jean-Sébastien Coron, Avradip Mandal, David Naccache et Mehdi Tibouchi : Fully homomorphic encryption over the integers with shorter public keys. pages 487–504, 2011.

[10] Jean-Sébastien Coron, David Naccache et Mehdi Tibouchi : Public key compression and modulus switching for fully homomorphic encryption over the integers, 2012.

[11] Léo Ducas et Daniele Micciancio : FHEW : Bootstrapping homomorphic encryption in less than a second. pages 617–640, 2015.

[12] Craig Gentry : Fully homomorphic encryption using ideal lattices. *STOC*, 2009.

[13] Craig Gentry et Shai Halevi : Implementing Gentry's fully-homomorphic encryption scheme. pages 129–148, 2011.

[14] Oded Goldreich, Daniele Micciancio, Shmuel Safra et J-P Seifert : Approximating shortest lattice vectors is not harder than approximating closest lattice vectors. *Information Processing Letters*, 71(2):55–61, 1999.

[15] Shai Halevi et Victor Shoup : Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 2013.

[16] Shai Halevi et Victor Shoup : Bootstrapping for HElib. *Advances in Cryptology–EUROCRYPT 2015*, pages 641–670, 2015.

[17] Aviad Kipnis, Jacques Patarin et Louis Goubin : Unbalanced oil and vinegar signature schemes. pages 206–222, 1999.

[18] RJ McEliece : A public-key cryptosystem based on algebraic. *Coding Thv*, 4244:114–116, 1978.

[19] Phong Q Nguyen et Damien Stehlé : Low-dimensional lattice basis reduction revisited. *ACM Transactions on Algorithms (TALG)*, 5(4):46, 2009.

[20] Fernando Pérez et Brian E. Granger : IPython : a System for Interactive Scientific Computing. *Computing in Science and Engineering*, 9(3):21–29, mai 2007.

[21] Ronald L Rivest, Len Adleman et Michael L Dertouzos : On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[22] Peter W Shor : Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[23] Nigel Smart et Frederik Vercauteren : Fully Homomorphic SIMD Operations. 2011.

[24] Jacques Stern : A new identification scheme based on syndrome decoding. pages 13–21, 1993.

[25] Marten Van Dijk, Craig Gentry, Shai Halevi et Vinod Vaikuntanathan : Fully homomorphic encryption over the integers. *Advances in cryptology–EUROCRYPT 2010*, pages 24–43, 2010.

Annexes

A Code source d'utilisation de HElib

```
#include "FHE.h"
#include "EncryptedArray.h"
#include "EvalMap.h"
/* Pour la fonction cputime */
#include <time.h>
#include <sys/time.h>

/*
 * renvoie le temps en secondes
 * renvoie le temps en secondes depuis t si t est donné en argument
 */
double cputime(const double before=0){
       return ((double)clock() / CLOCKS_PER_SEC)-before;
```

```
}
void rec(){
/* FIXING VARIABLES */
        long values[2][19] = {
              // 80 bits
              {2,21168,27305,28,43,635,0,
              10796,26059,0,42,18,0,100,1,
              25,23,3,64},
              { // 128 bits
               /*p=*/2,/*phim=*/26400,/*m=*/27311,
               /*d=*/55,/*m1=*/31,/*m2*/881,/*m3=*/0,
               /*g1=*/21145,/*g2=*/1830,/*g3=*/0,
               /*ord1=*/30,/*ord2=*/16,/*ord3=*/0,
               /*c_m=*/100,/*r=*/1,/*L=*/25,/*B=*/23,
               /*c=*/3,/*skHwt=*/64
               }
        };
        // on copie les valeurs dans le tableau mValues
        long mValues[19];
        for (int v=0; v<19; v++){
              mValues[v] = values[1][v]; // on utilise les paramè
                  tres pour 127 bits
        }

    // on affiche la valeur de m
        cout << "m=" << mValues[2] << endl;

        Vec<long> mvec;
        vector<long> gens;
        vector<long> ords;

        int p = mValues[0];
        long phim = mValues[1];
        long m = mValues[2];
        // on s'assure que p est premier avec m
        assert(GCD(p, m) == 1);

        append(mvec, mValues[4]);
        if (mValues[5]>1) append(mvec, mValues[5]);
        if (mValues[6]>1) append(mvec, mValues[6]);
        gens.push_back(mValues[7]);
        if (mValues[8]>1) gens.push_back(mValues[8]);
        if (mValues[9]>1) gens.push_back(mValues[9]);
```

```
        ords.push_back(mValues[10]);
        if (abs(mValues[11])>1) ords.push_back(mValues[11]);
        if (abs(mValues[12])>1) ords.push_back(mValues[12]);
        int r = mValues[14],
               L = mValues[15],
               B = mValues[16],
               c = mValues[17],
               skHwt = mValues[18]
               ;


/* COMPUTING CONTEXT AND PARAMETERS */
        cout << "Computing parameters ... " << flush;
        double t = cputime();
        FHEcontext context(m, p, r, gens, ords);
        context.bitsPerLevel = B;
        buildModChain(context, L, c);
        context.makeBootstrappable(mvec, 0, false);
        context.rcData.skHwt = skHwt;

        long nPrimes = context.numPrimes();
        IndexSet allPrimes(0,nPrimes-1);
        double bitsize = context.logOfProduct(allPrimes)/log(2.0);

        long p2r = context.alMod.getPPowR();
        context.zMStar.set_cM(mValues[13]/100.0);
        cout << cputime(t) << "s" << endl; // affiche le temps d'exé
            cution pour calculer les paramètres

/* KEY GENERATION */
        cout << "Generating keys ... " << flush;
        t = cputime();
        FHESecKey secretKey(context);
        FHEPubKey& publicKey = secretKey;
        secretKey.GenSecKey(skHwt);   // A Hamming-weight-64 secret
            key
        addSome1DMatrices(secretKey); // compute key-switching
            matrices that we need
        addFrbMatrices(secretKey);
        secretKey.genRecryptData();
        cout << cputime(t) << "s" << endl; // affiche le temps d'exé
            cution pour générer les clés
```

```
        cerr << "Security parameter: " << context.securityLevel() <<
            endl;

        // permet de sauvegarder les clés dans des fichiers, pour les
             recharger plus tard
        ofstream seckey("secret.key", ofstream::out),
                        pubkey("public.key", ofstream::out);
        seckey << secretKey << endl;
        pubkey << publicKey << endl;
        seckey.close();
        pubkey.close();
        return;

/* INIT SUPPORT */
        EncryptedArray ea(context);
        NewPlaintextArray ptarray(ea),
                                        ptarray1(ea),
                                        ptarray2(ea);
        Ctxt c1(publicKey), c2(publicKey);
        encode(ea,ptarray,2);
        encode(ea,ptarray1,1);

/* BEGIN ENCRYPTION */
        cout << "Encryption ... " << flush;
        t = cputime();
        ea.encrypt(c1,publicKey,ptarray);
        ea.encrypt(c2,publicKey,ptarray1);
        cout << cputime(t) << "s" << endl; // affiche le temps pour
            chiffrer le vecteur

        cout << "Multiply ... " << flush;
        t = cputime();
        c1 *= c2;
        c1 *= c2;
        cout << cputime(t) << "s" << endl; // affiche le temps pour
            mutliplier les vecteurs 2 fois entre eux

        cout << "Recryption ... " << flush;
        t = cputime();
        publicKey.reCrypt(c1);
        cout << cputime(t) << "s" << endl; // affiche le temps pour
            rechiffrer un chiffré
```

```
/* BEGIN DECRYPTION */
        ea.decrypt(c1,secretKey,ptarray2);

/* CHECK RESULT */
        vector<long> before(ea.size()),
                                after(ea.size());
        decode(ea,after,ptarray2);
        cout << after << endl;
}

int main(int argc, char *argv[])
{
/* Dans le cas où le multi-threading est activé, on initialise la
    pool */
#ifdef FHE_BOOT_THREADS
  const int nthreads = 4;
  UniquePtr<MultiTask> localBootTask;
  localBootTask.make(nthreads);
  bootTask = localBootTask.get();
  cout << "*** nthreads = " << bootTask->getNumThreads() << endl;
#else
  cout << "*** no threads" << endl;
#endif
        rec();
        return 0;
}
```

B Définitions

Définition 13 *Le modèle standard est le modèle dans lequel un adversaire (au cryptosystème) est uniquement limité par le temps et la mémoire dont il dispose. On estime qu'un cryptosystème est sûr dans ce modèle pour une attaque si celle-ci requiert un temps exponentiel pour être réalisée.*

Définition 14 *Le modèle de l'oracle aléatoire est le modèle dans lequel intervient une boite noire qui répond de façon totalement aléatoirement et uniformément à chaque unique requête reçue. La sortie de cette boite noire est tirée dans un domaine (fini) prédéfini. L'oracle répond de la même manière lors qu'une requête déjà reçue est à nouveau faite. On considère un cryptosystème sûr dans ce modèle si étant donné un nombre polynomial de sorties*

de l'oracle, un attaquant n'est pas capable de deviner une seule entrée.

Définition 15 ***(LWE décisionnel)*** *: Soient λ un paramètre de sécurité, $n = n(\lambda)$ une dimension entière, $q = q(\lambda) \geq 2$ un entier et $\chi = \chi(\lambda)$ une distribution sur $\mathbb{Z}$. Le problème $LWE_{n,q,\chi}$ est de distinguer les deux distributions suivantes :*

- *des couples (a_i, b_i) sont tirés uniformément dans $\mathbb{Z}_q^{n+1}$*
- *$s_i, a_i \leftarrow \mathbb{Z}_q^n$ uniformément, $e_i \leftarrow \chi$, $b_i = \langle a_i, s_i \rangle + e_i$. Retourner (a_i, b_i).*

La supposition $LWE_{n,q,\chi}$ est de dire que le problème $LWE_{n,q,\chi}$ est difficile.

Définition 16 ***(RLWE décisionnel)*** *: Soient λ un paramètre de sécurité, $f(x) = x^d + 1$ avec $d = d(\lambda)$ une puissance de 2, $q = q(\lambda) \geq 2$ un entier, $R = \mathbb{Z}[x]/(f(x))$, $R_q = R/qR$, $\chi = \chi(\lambda)$ une distribution sur R. Le problème $RLWE_{d,q,\chi}$ est de distinguer les deux distributions suivantes :*

- *des couples (a_i, b_i) sont tirés uniformément dans R_q^2*
- *$s_i, a_i \leftarrow R_q$ uniformément, $e_i \leftarrow \chi$, $b_i = a_i.s_i + e_i$. Retourner (a_i, b_i).*

La supposition $RLWE_{d,q,\chi}$ est de dire que le problème $RLWE_{d,q,\chi}$ est difficile.