Implémentation d'un FHE

Lucas Roux et Eric Sageloli

6 février 2019

Table des matières

1 Notations **3**

2 Introduction **4**

3 Notions préliminaires **5**
3.1 LWE et DLWE 5
3.2 Réseaux euclidiens 6
3.3 La gaussienne discrète 7

4 FHE, SFHE et bootstrapping **7**

5 Présentation du cryptosystème **7**
5.1 L'idée générale 7
5.2 Fonctions utilisées 8
5.3 Définition du cryptosystème 9
5.4 Autres algorithmes de déchiffrement 11
5.4.1 mp_decrypt : q est une puissance de 2 11
5.4.2 mp_all_q_decrypt : q est quelconque 11
5.5 Opérations homomorphes 11

6 Analyse du cryptosystème : sécurité, profondeur des circuits **13**
6.1 Sécurité du cryptosystème 13
6.2 Contraintes asymptotiques pour les choix de paramètres 15
6.2.1 Analyse de la profondeur des circuits 16
6.2.2 Résumé des contraintes sur les paramètres 16
6.3 Choix concrets de paramètres 16
6.3.1 Présentation de lwe_estimator 16
6.3.2 Proposition de choix sécurité pour très faible profondeur 17

7 Mise en place d'un bootstrapping **17**
7.1 Un point sur la sécurité 17
7.2 Un premier découpage 17
7.3 Sommer des vecteurs avec une profondeur en NAND minimale 18
7.4 Prendre la valeur absolue dans $\mathbb{Z}_q$ 20

8 Implémentation d'un FHE avec bootstrapping « jouet » **21**
8.1 Présentation de notre arborescence 21
8.1.1 GWS_scheme 21
8.1.2 analysis 21
8.1.3 unitary_test 22

9 Des librairies pour du FHE **22**
9.1 La librarie SEAL 22
9.2 The Gate Bootstrapping API 22
9.2.1 Un exemple simple 23

1 Notations

Soit $q > 0$. La valeur absolue d'un élément $x \in \mathbb{Z}_q$ sera par définition la valeur absolue dans $\mathbb{Z}$ de son représentatant dans $[\![-q/2, q/2]\!]$. La norme infinie $\|\vec{x}\|_\infty$ d'un vecteur $\vec{x} \in \mathbb{Z}_q^n$ sera alors le maximum des valeurs absolues de ses coordonnées.

2 Introduction

Certains systèmes cryptographiques, comme RSA, ElGamal ou encore le crytosystème de Pallier possèdent une propriété intéressante : faire le produit de deux chiffrés revient à chiffrer le produit de leurs clairs. Cette proriété offre à un attaquant des informations qui affaiblissent le chiffré - on parle de malléabilité -, mais offre aussi des perspectives intéressante : la possibilité d'appliquer des opérations sur les chiffrés de données sans avoir à les déchiffrer permet de travailler avec des entitiées sans avoir leur dévoiler des données que l'on estime sensibles.

Toutefois, pouvoir uniquement multiplier des chiffrés [1] est trop limité, ne permettant pas de faire beaucoup de manipulations intéressantes. Il a fallu attendre jusqu'en 2009, avec la thèse de Craig Gentry [**?**] pour que devienne plausible la possibilité d'un cryptosystème « commutant »à la fois avec la somme, le produit et la multiplication par des scalaires ; on parle alors de « Fully homomorphic cryptosystem »(FHE).

Celui-ci utilisait des réseaux euclidiens dits « idéaux »et utilisait une tecnique dite de « bootstrapping »pour passer d'un « Somewhat fully homomorphic cryptosystem », limitant le nombre d'opérations possibles afin que le déchiffrement fonctionne toujours, vers un vrai FHE.

A partir de là, de nombreuses tentatives de cryptosystèmes dits « fully homomorphics »ont emergés. Notamment, à partir de 2011, des cryptosystèmes dits de seconde génération, basés sur le problème Learning with Error ont emergés. Ils avaient la particularité d'avoir une somme entre chiffrés facile à mettre en place, tandis que la multiplication demandait elle plus de travail, demandant notamment une opération dite de « relinéarisation »assez complexe.

Le but de notre rapport est d'étudier le cryptosystème GWS, publié par Gentry, Sahai et Waters en 2013 (voir [**?**]), dit de troisième génération, dont le but fut justement de trouver un cryptotsystème dont la somme tout comme le produit de chiffrés soient « naturels ».

Pour cela, nous le présenterons, étudierons sa sécurité, puis en présenterons une implémentation jouet [2] faite en `sagemath` que nous avons réalisée pour ce projet (voir CITER LE GIT). Enfin, nous jetterons un œil a des API open-source actuellement disponibles pour utiliser des FHE.

1. ou bien seulement les sommer, comme cela peut être le cas, par exemple avec des variables de ElGamal
2. sans regards sur les très nombreuses optimisations aujourd'hui faites dans les vraies implémentations de GSW et ses dérivés

3 Notions préliminaires

3.1 LWE et DLWE

Nous présentons ici les définitions du Learning with Error (LWE) dans leur version décisionnelle et calculatoire.

Définition 1. *Decisional Learning with Errors (DLWE), version décisionnelle*

Pour un paramètre de sécurité λ, soit $n = n(\lambda)$, $q = q(\lambda)$ des entiers et $\chi = \chi(\lambda)$ une distribution sur $\mathbb{Z}$.

Le problème $DLWE_{n,q,\chi}$ consiste à devoir distinguer deux distributions sur $\mathbb{Z}_q^{n+1}$ à partir d'un nombre polynomial $m = m(\lambda)$ d'échantillons qu'une des deux à produite. La première distribution crée des vecteurs $(\vec{a}_i, b_i) \in \mathbb{Z}_q^{n+1}$ uniforme. La deuxième utilise un $\vec{s} \in \mathbb{Z}_q^n$ tiré uniformément et prend pour valeurs des vecteurs $(\vec{a}_i, b_i)$ pour lesquels :

$$b_i = \langle \vec{a}_i, \vec{s} \rangle + e_i$$

où e_i est crée obtenue par χ.

Notons qu'alors, $n = O(P(\lambda))$ et $\log(q) = O(P(\lambda))$ pour un polynome P.

Définition 2. *Learning with Errors (LWE)*

Pour un paramètre de sécurité λ, soit $n = n(\lambda), q = q(\lambda)$ des entiers et $\chi = \chi(\lambda)$ une distribution sur $\mathbb{Z}$. On tire $\vec{s} \in \mathbb{Z}_q^n$ uniformément et on considère la distribution qui prend pour valeurs des vecteurs $(\vec{a}_i, b_i)$ pour lesquels :

$$b_i = \langle \vec{a}_i, \vec{s} \rangle + e_i$$

où e_i est crée obtenue par χ.

Le problème $LWE_{n,q,\chi}$ consiste à trouver $\vec{s}$ à partir d'un nombre polynomial $m = m(\lambda)$ d'échantillons.

Ces deux problèmes sont en fait « équivalents ». C'est assez évident de LWE vers DLWE. De plus, le Lemme 4.2 de [**?**] montre comment réduire à DLWE à LWE sous certaines hypothèses lorsque q est premier et $q = \mathcal{O}(\text{poly}(n))$. Le théorème 3.1 de [**?**] montre la même chose mais lorsque q est un produit de premiers $p_i \in \mathcal{O}(\text{poly}(n))$, comme ce sera le cas lorsque nous considèrerons $q = 2^k$.

Voyons par exemple le cas (plus facile) où q est premier :

Proposition 1. *DWLE vers LWE*

Soit $n \geqslant 1$ un entier, $2 \leqslant q \leqslant poly(n)$ un nombre premier et χ une distribution sur $\mathbb{Z}_q$. Supposons avoir accès à un automate $\mathcal{W}$ qui accepte avec une probabilité exponentiellement proche de 1 les distributions $A_{s,\xi}$ et rejete avec une probabilité exponentiellement proche de 1 la distributions uniforme U.

Il existe alors un automate $\mathcal{V}$ qui, étant donné des échantillons de $\mathcal{A}_{s,\chi}$ pour un certain s, retrouve s avec une probabilité exponentiellement proche de 1.

Démonstration. Nous indiquons ici la démontration faite dans [**?**].

L'automate W' va trouver s coordonée par coordonnée. Montrons comment W' obtient la première coordonnée s_1.

Pour $k \in \mathbb{Z}_q$, on considère la fonction :

$$f_{k,1} : (a, b) \mapsto (a + (l, 0, ..., 0), b + l \cdot k)$$

avec $l \in \mathbb{Z}_q$ échantilloné uniformément sur $\mathbb{Z}_q$.

$f_{k,1}$ appliquée à un échantillon uniforme donne un échantillon uniforme tandis qu'appliquée à un échantillon de $A_{s,\chi}$, elle donne un échantillon de $A_{s,\chi}$ si $k = s_1$, et uniforme sinon.

On peut faire une recherche exhaustive sur les $k \in \mathbb{Z}_q$ jusqu'à en trouver un accepté par W, qui sera le bon avec une probabilité exponentiellement proche de 1.

Cela se fait en temps polynomial car $q < \text{poly}(n)$ et $f_{k,1}$ s'execute en temps polynomial.

On peut effectuer la même chose avec la fonction

$$f_{k,i} : (a, b) \mapsto (a + (0, 0, ..., l, 0, ..., 0), b + l \cdot k)$$

avec le l ajouté à a en $i$ème position $\forall i$.

On retrouve ainsi s avec n calculs polynomiaux en n, ce qui reste évidemment polynomial en n.

La probabilité de se tromper est n fois quelque chose d'exponentiellement proche de 0 et reste donc exponentiellement proche de 0. □

Pour analyser la sécurité du cryptosystème, nous utiliserons le problème DLWE. Comme l'indique le théorème 1 de [**?**], il est possible de réduire le problème LWE à des problèmes sur des réseaux.

Indiquons ici de façon informelle comment passer du problème LWE à un problème de type SVP (short vector problem). Tout d'abord, nous aurons besoin d'exprimer LWE sous une forme matricielle :

Définition 3. *versions matricielles de DLWE et LWE*

En prenant les paramètres de la précédente définition, le problème $DLWE_{n,q,\chi}$ consiste à décider si une matrice $A \in \mathbb{Z}_q^{m\times(n+1)}$ est uniforme ou bien s'il existe un vecteur $\vec{v} = (1 \quad -\vec{s})$ tel que $A \cdot \vec{v} \in \mathbb{Z}_q^m$ est créé à partir de χ^m. Autrement dit, avec les notations de la formulation classique de LWE, si les lignes de A sont de la forme $(b_i, \vec{a}_i)$.

Le problème $LWE_{n,q,\chi}$ consiste lui à trouver $\vec{v}$ à partir de A.

Nous allons ici considérer le problème LWE calculatoire, dans lequel il faut trouver le vecteur $\vec{v}$ tel que :

$$A \cdot \vec{v} = \vec{e} \mod q$$

où les coordonnées de $\vec{e}$ sont créées par χ.

De façon équivalente, il faut trouver un vecteur $(* \quad \vec{v})$ tel que :

$$\begin{bmatrix} q & A \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} * \\ \vec{v} \end{bmatrix} = \begin{bmatrix} \vec{e} \\ \vec{v} \end{bmatrix}$$

Si la distribution χ créé de petites valeurs, on voit qu'on a alors trouvé un « petit »vecteur du réseau engendré par les colonnes de

$$\begin{bmatrix} q & A \\ 0 & 1 \end{bmatrix}$$

3.2 Réseaux euclidiens

Nous rappelons ici quelques résultats sur les réseaux euclidiens, tels qu'énoncés dans [**?**]. Ils nous serons utiles pour définir les gaussiennes discrètes ainsi que pour comprendre un des algorithme de déchiffrement du cryptosystème GSW.

Tous les réseaux considérés ici sont de rang plein, autrement dit, si $\subset \mathbb{R}^n$, alors L est de dimension n.

Définition 4. *Soit $L \subset \mathbb{R}^n$ un réseau. Le dual de L est défini comme étant :*

$$L^* = \{v \in \mathbb{R}^n \ : \ \langle x, v\rangle \in \mathbb{Z} \ \text{ pour tout } x \in L\}$$

Proposition 2. *Soit $L \subset \mathbb{R}^n$ un réseau euclidien. Soit B est une base de L.*
B^{-t} est une base de L^*.

Pour q un entier et $A \in \mathbb{Z}^{n\times m}$, on pose :

$$\Lambda^{\perp}(A) = \{z \in \mathbb{Z}^m \ : \ Az = 0 \mod q\}$$

$$\Lambda(A^t) = \{z \in \mathbb{Z}^m \ : \ \exists s \in \mathbb{Z}_q^n, z = A^t s \mod q\}$$

Proposition 3. *Conservant les notations précédentes,*

$$q \cdot \Lambda^{\perp}(A)^* = \Lambda(A^t)$$

3.3 La gaussienne discrète

Très souvent, la distribution χ choisi pour avoir des paramètres sécurités pour le problème LWE est une gaussienne discrète. Nous nous proposons ici d'en indiquer la définition, ainsi que d'en indiquer certaines propriétés.

Rappellons aussi qu'une famille $\{\chi_n\}_n$ de distributions est dite B-bornée pour une borne $B = B(n)$ si la fonction suivante est négligeable :

$$n \mapsto \mathbb{P}\left(\chi_n > B(n)\right)$$

Nous reprenons ici les notations de [**?**].

Soit un entier $n > 0$ et $\sigma > 0$. On définit la densité gaussienne sur $\mathbb{R}^n$ comme la fonction qui à $x \in \mathbb{R}^n$ attribue :

$$\rho_{s,c}(x) = e^{\phi * \frac{||x-c||}{s}^2}$$

Puis, pour un réseau $\Lambda \in \mathbb{R}^n$, nous définissons la gaussienne discrète D_α comme la distribution de support Λ de loi de probabilité :

$$D_{\Lambda,s,c}(x) = \frac{\rho_{s,c}(x)}{\sum_{l\in\Lambda}\rho_{s,c}(l)}$$

Enfin, pour un entier $q > 0$, nous définissons la gaussienne discrète D^q_α modulo un entier $q > 0$ comme la composition la fonction qui a $x \in \mathbb{Z}_q$ attribue

LA COMPOSITION LA FONCTION ? WTF ?

$$D_{\mathbb{Z},s,c}(\pi^{-1}(x))$$

où π est la projection $\mathbb{Z} \to \mathbb{Z}_q$.

La définition et la position suivante (Le lemme 4.2 de [**?**]) permettent de trouver une borne à une famille de gaussienne.

Définition 5. *Pour un réseau L de dimension n et un réel $\epsilon > 0$, le paramètre $\eta_\epsilon(L)$ est le plus petit réel $s > 0$ tel que*

$$\rho_{1/s}(L^* \setminus \{0\}) \leqslant \epsilon$$

Proposition 4. *Pour tout $\epsilon > 0$, $s \geqslant \eta_\epsilon(\mathbb{Z})$ et pour tout $t > 0$:*

$$\mathbb{P}\left(|x - c| \geq t \cdot s\right) \leqslant 2e^{-\pi t^2} \cdot \frac{1+\epsilon}{1-\epsilon}$$

Notamment, pour $0 < \epsilon < 1/2$ et $t \geqslant \omega(\sqrt{\log(n)})$, cette probabilité est négligeable.

4 FHE, SFHE et bootstrapping

5 Présentation du cryptosystème

5.1 L'idée générale

L'idée de ce cryptosystme consiste à prendre pour secret un certain vecteur $\vec{v} \in \mathbb{Z}_q^N$ pour certains paramètres $q, N \in \mathbb{N}$, puis à chiffrer un message $m \in \mathbb{Z}_q$ à l'aide d'une matrice $C \in \mathbb{Z}_q^{N\times N}$ ayant m pour valeur propre associée au vecteur propre $\vec{v}$. Autrement dit, avec :

$$C \cdot \vec{v} = m\vec{v} \mod q$$

De là, il est facile de voir que pour $\lambda \in \mathbb{Z}$ et C_1 et C_2 chiffrés de m_1 et m_2, on a :

$$\begin{aligned}(C_1 + C_2) \cdot \vec{v} &= (m_1 + m_2)\vec{v}\\ (C_1 \times C_2) \cdot \vec{v} &= (m_1 + m_2)\vec{v}\\ (\lambda C_2) \cdot \vec{v} &= (\lambda m_1)\vec{v}\end{aligned}$$

Toutefois, un tel système n'est pas sécurisé car C n'a qu'un nombre fini de valeurs propres, et il semble donc facile de retrouver le secret $\vec{v}$.

La solution consiste alors à ajouter du bruit au chiffré, c'est à dire à chiffrer $m \in \mathbb{Z}_q$ par une matrice $C \in \mathbb{Z}^{N \times N}$ telle que :

$$C\vec{v} = m\vec{v} + \vec{e}$$

pour une « petite »erreur $\vec{e}$. Si le vecteur $\vec{v}$ contient un grand coefficient v_i, on voit alors qu'il reste possible de retrouver m avec

$$\frac{(C\vec{v})_i}{v_i} = \frac{m + e_i}{v_i}$$

Nous verrons que pour de bons choix de paramètres, déchiffrer un tel message permet de résoudre une instance de LWE.

Toutefois, l'ajout d'une erreur comporte ses inconvénients. Si nous revenons aux équations précédentes, en introduisant les erreurs $\vec{e}_i$ pour chiffrer m_i ($i \in \{1, 2\}$), on obtient :

$$\begin{aligned}
(C_1 + C_2) \cdot \vec{v} &= (m_1 + m_2)\vec{v} + (\vec{e}_1 + \vec{e}_2) \\
(C_1 \times C_2) \cdot \vec{v} &= (m_1 * m_2)\vec{v} + C_1\vec{e}_2 + m_2\vec{e}_1 \\
(\lambda C_2) \cdot \vec{v} &= (\lambda m_1) + \lambda e_i \vec{v}
\end{aligned}$$

Notamment, on voit que le terme $C_1 * \vec{e}_2$ peut être très grand même pour un petit $\vec{e}_2$. Nous verrons par la suite comment choisir nos paramètres, et notamment $\vec{v}$, afin de toujours pouvoir se ramener à des chiffrés $C \in \{0, 1\}^{N \times N}$. De cette façon, on aura :

5.2 Fonctions utilisées

Plusieurs algorithmes seront utiles pour pouvoir bien définir le système FHE :

BitDecomp

Entrée : Cet algorithme prends en entrée un vecteur $\vec{a} = (a_1, ..., a_k) \in \mathbb{Z}_q^k$.
Sortie : Cet algorithme retourne la décomposition binaire des éléments de $\vec{a}$ sous la forme d'un vecteur.
Algorithme : Pour chaque a_i, on détermine sa représentation binaire avec les bits de faibles puissance à gauche et non à droite. On retourne la concaténation de ces représentations binaires sous la forme d'un vecteur.

BitDecomp^{-1}

Entrée : Cet algorithme prends en entrée un vecteur $\vec{a} = (a_{1,0}, ..., a_{1,l-1}, a_{2,0}, ..., a_{k,l-1})$.
Sortie : Cet algorithme renvoie $(\sum_{i=0}^{l-1} 2^i a_{1,i}, ..., \sum_{i=0}^{l-1} 2^i a_{k,i})$.
Remarque : Si tous les $a_{i,j}$ sont dans $\{0, 1\}$, cet algorithme inverse bien **BitDecomp**, cependant, sa définition ne le limite pas aux vecteurs $\in \{0, 1\}^{k \times l}$.

Flatten

Entrée : Cet algorithme prends en entrée un vecteur $\vec{a} = (a_{1,0}, ..., a_{1,l-1}, a_{2,0}, ..., a_{k,l-1})$.
Sortie : Cet algorithme retourne un vecteur $\vec{b} = (b_{1,0}, ..., b_{1,l-1}, b_{2,0}, ..., b_{k,l-1})$ dont les éléments sont tous dans $\{0, 1\}$.
Algorithme : On calcule **BitDecomp**$^{-1}(\vec{a})$ et on obtient un vecteur $\vec{z} \in \mathbb{Z}_q^k$. On applique ensuite **BitDecomp** à $\vec{z}$ et l'on renvoie le résultat obtenu.

PowersOf2

Entrée : Cet algorithme prends en entrée un vecteur $\vec{a} = (a_1, ..., a_k) \in \mathbb{Z}_q^k$.
Sortie : Cet algorithme renvoie $(a_1, 2a_1, 2^2a_1, ..., 2^{l-1}a_1, a_2, ..., 2^{l-1}a_k)$.

Proposition 5. *Soient $\vec{a}$ et $\vec{b}$ dans $\mathbb{Z}_q^k$.*
On a $\langle \textbf{\textit{BitDecomp}}(\vec{a}), \textbf{\textit{PowersOf2}}(\vec{b})\rangle = \langle \vec{a}, \vec{b}\rangle$.

Démonstration.

$$\begin{aligned}\langle \mathbf{BitDecomp}(\vec{a}), \mathbf{PowersOf2}(\vec{b})\rangle &= \sum_{i=1}^{k}\sum_{j=0}^{l-1} a_{i,j} * (2^j * b_i) \\ &= \sum_{i=1}^{k} b_i * \sum_{j=0}^{l-1}(a_{i,j} * 2^j) \\ &= \sum_{i=1}^{k} b_i * a_i \\ &= \langle \vec{a}, \vec{b}\rangle.\end{aligned}$$

□

Proposition 6. *Soient $\vec{a}$ dans $\mathbb{Z}_q^{k\times l}$ et $\vec{b}$ dans $\mathbb{Z}_q^k$.*
On a $\langle \vec{a}, \textbf{\textit{PowersOf2}}(\vec{b})\rangle = \langle \textbf{\textit{BitDecomp}}^{-1}(\vec{a}), \vec{b}\rangle = \langle \textbf{\textit{Flatten}}(\vec{a}), \textbf{\textit{PowersOf2}}(\vec{b})\rangle$.

Démonstration.

$$\begin{aligned}\langle \vec{a}, \mathbf{PowersOf2}(\vec{b})\rangle &= \sum_{i=1}^{k}\sum_{j=0}^{l-1} a_{j+li} * (2^j * b_i) \\ &= \sum_{i=1}^{k} b_i * \sum_{j=0}^{l-1}(a_{j+li} * 2^j) \\ &= \langle \mathbf{BitDecomp}^{-1}(\vec{a}), \vec{b}\rangle\end{aligned}$$

Soit $c = \mathbf{BitDecomp}^{-1}(\vec{a})$.

$$\begin{aligned}\langle \mathbf{Flatten}(\vec{a}), \mathbf{PowersOf2}(\vec{b})\rangle &= \langle \mathbf{BitDecomp}(\vec{c}), \mathbf{PowersOf2}(\vec{b})\rangle \\ &= \sum_{i=1}^{k}\sum_{j=0}^{l-1} c_{i,j} * (2^j * b_i) \\ &= \sum_{i=1}^{k} b_i * \sum_{j=0}^{l-1}(c_{i,j} * 2^j) \\ &= \sum_{i=1}^{k} b_i * c_i \\ &= \langle \mathbf{BitDecomp}^{-1}(\vec{a}), \vec{b}\rangle \\ &= \langle \vec{a}, \mathbf{PowersOf2}(\vec{b})\rangle\end{aligned}$$

□

5.3 Définition du cryptosystème

On rappelle que les paramètres du système défini ici sont : le paramètre de dimension n, le modulus q, un modèle de distribution de l'erreur χ ainsi que m, qui, tout comme n influera la taille des matrices manipulées.
On note $l = \lfloor \log q \rfloor + 1$ et $N = (n+1)\ l$.

Setup

Entrée : Cet algorithme prends en entrée 1^λ et 1^L avec λ paramètre de sécurité et L paramètre de profondeur.
Sortie : Cet algorithme retourne les paramètres n, q, χ, m du système.
Algorithme : On définit des paramètres permettant de pouvoir effectuer au moins L opérations sur un chiffré et de toujours pouvoir le déchiffrer correctemment tout en assurant qu'un adversaire attaquant le système doive effectuer au moins 2^λ opérations, quelle que soit l'attaque qu'il choisisse. La façon de déterminer ces paramètres n'est pas définie afin de pouvoir l'adapter suivant l'evolution des attaques.

SecretKeyGen

Entrée : Cet algorithme n'a besoin en entrée que des paramètres donnés par **Setup**.
Sortie : Cet algorithme retourne la clé secrète $\vec{s} \in \mathbb{Z}_q^{n+1}$.
Algorithme : On génère aléatoirement un vecteur $\vec{t} \in \mathbb{Z}_q^n$. On définit la clé secrète comme $\vec{s} = (1, -t_1, ..., -t_n)$.
Taille : Comme on l'a dit, $\vec{s} \in \mathbb{Z}_q^{n+1}$. Par définition, q, et donc tous élément de $\mathbb{Z}_q$, s'écrit en l bits. $\vec{s}$ fait donc une taille de $l * (n+1) = N$ bits.
On note $\vec{v} = \textbf{PowersOf2}(\vec{s})$.

PublicKeyGen

Entrée : Cet algorithme n'a besoin en entrée que des paramètres donnés par **Setup** et d'une clé secrète construite avec ces mêmes paramètres.
Sortie : Cet algorithme retourne la clé publique $A \in \mathbb{Z}_q^{m \times n}$.
Algorithme : On génère une matrice uniforme $B \in \mathbb{Z}_q^{n \times m}$ et un vecteur $\vec{e}$ de m éléments choisis suivant la distribution χ. On définit $\vec{b} = B \times \vec{t} + \vec{e}$. La clé publique est la matrice constituée de l'indentation de $\vec{b}$ considéré comme un vecteur colonne et de B.
Taille : $A \in \mathbb{Z}_q^{m \times n}$ donc A s'écrit en $l * n * m = N * (m-1)$ bits.

Encrypt

Entrée : Cet algorithme prend en entrée les paramètres du système, la clé publique et un message $\mu \in \mathbb{Z}_q$.
Sortie : Cet algorithme retourne le chiffré $C \in \mathbb{Z}_q^{N \times N}$ de μ.
Algorithme : On génère uniformément une matrice $R \in \{0,1\}^{N \times m}$. Le chiffré est : $C = \textbf{Flatten}(\mu \times I_N + \textbf{BitDecomp}(R \times A))$.
Taille : $C \in \mathbb{Z}_q^{N \times N}$ s'écrit en $l * N^2$ bits.

Dec

Entrée : Cet algorithme prend en entrée les paramètres du système, la clé secrète et un chiffré d'un message $\mu \in \{0,1\}$.
Sortie : Cet algorithme retourne le clair du chiffré si l'erreur de ce dernier n'est pas trop élevée.
Algorithme : On rappelle que les l premiers coefficients de $\vec{v}$ sont les puissances de 0 à $l-1$ de 2. Soit $i \leqslant l$ tel que le i+1ème coefficient de $\vec{v}$, égal à 2^i, soit compris entre $q/4$ et $q/2$, $q/2$ compris. On note C_i la ième ligne de C. On calcule ensuite $x_i = \langle C_i, \vec{v} \rangle$ et on renvoie $\lfloor x_i / v_i \rceil$.

Définition 6. *On appellera erreur d'un chiffré C d'un message μ le vecteur $\vec{e}$ tel que*

$$C \cdot \vec{v} = \mu \, \vec{v} + \vec{e}$$

Proposition 7. ***Dec*** *décrypte avec succès les chiffrés dont l'erreur $\vec{e}$ satisfait $\|\vec{e}\|_1 < q/8$.*

Démonstration. On a $x_i = \mu * v_i + e$ avec $|e| \leqslant \|\vec{e}\|_1$ et $\frac{x_i}{v_i} = \mu + \frac{e}{v_i}$.
$|v_i| > \frac{q}{4}$, d'où $|\frac{e}{v_i}| < 1/2$, donc $\lfloor \frac{x_i}{v_i} \rceil = \mu$. □

5.4 Autres algorithmes de déchiffrement

L'algorithme de déchiffrement que nous avons présenté fonctionne sans contraintes sur q mais ne déchiffre que des chiffrés de 0 et de 1.

Nous proposons ici une analyse un peu plus fine du déchiffrement pour montrer comment faire pour des chiffrés de n'importe quel élément de $\mathbb{Z}_q$.

Pour cela, remarquons qu'en partant d'un chiffré C de $m \in \mathbb{Z}_q$ On a :

$$C \cdot \vec{v} = m\vec{v} + \vec{e} \mod q$$

pour une erreur $\vec{e}$.
En considérant l'équation sur les l premières coordoonées, on obtient :

$$\vec{a} = m\vec{p} + \vec{e} \mod q \quad \text{où} \quad \vec{p} = (1\ 2\ \cdots\ 2^{l-1})$$

Notant $L = \Lambda(\vec{p}^t)$, on constate qu'on peut retrouver $m\vec{p}$ en trouvant le vecteur de L le plus proche de $\vec{a}$

De cette idée, on déduit 2 algorithmes de déchiffrements supplémentaires, dépendant de la façon dont on résout le problème du vecteur le plus proche.

— **mp_decrypt** qui suppose que q est une puissance de 2.
— **mp_all_q_decrypt** sans hypothèses sur q.

5.4.1 mp_decrypt : q est une puissance de 2

L'algorithme, présenté dans [**?**], utilise le fait que $q = 2^l$.

En regardant la dernière coordonnée de :

$$\vec{a} = m\vec{p} + \vec{e} \mod q \quad \text{où} \quad \vec{p} = (1\ 2\ \cdots\ 2^{l-1})$$

On obtient :

$$m2^{l-1} + e_l \mod 2^l$$

qui est proche de 0 si m est pair et de $q/2$ sinon. On déduit de cette façon le premier bit de l'écriture en binaire de m et la méthode est similaire pour complètement déduire m.

5.4.2 mp_all_q_decrypt : q est quelconque

Le travail effectué ici est notamment tiré de la section 4 de [**?**].

En utilisant la proposition 3, on constate que

$$L = q \cdot \Lambda^{\perp}(\vec{p})$$

Il nous suffit donc de trouver une base B de $\Lambda^{\perp}(\vec{p})$ pour en déduire une base $q\,B^{-t}$ de L.

Or, il est facile de voir que

$$B = \begin{bmatrix} 2 & & & & q_0 \\ -1 & 2 & & & q_1 \\ & -1 & & & \\ & & \ddots & & \vdots \\ & & & 2 & q_{k-2} \\ & & & -1 & q_{k-1} \end{bmatrix}$$

convient.

On peut alors par exemple utiliser l'algorithme nearest plane de Baibai à partir de cette base pour déchiffrer. Notons que des bornes sur les vecteurs de la décomposition de Gram-Schmidt de cette matrice sont données dans [**?**], ce qui peut-être intéressant car cela est lié au domaine fondamental utilisé par l'algorithme.

5.5 Opérations homomorphes

On rappelle que $\vec{v}$ est de la forme $\mathbf{PowersOf2}(\vec{s})$ et que donc $\mathbf{Flatten}(A) \cdot \vec{v} = A \times \vec{v}$ pour tout A.

MultConst

Entrée : Cet algorithme prend en entrée les paramètres du système, un chiffré $C \in \mathbb{Z}_q^{N\times N}$ d'un message μ et une constante $\alpha \in \mathbb{Z}_q$.
Sortie : Cet algorithme retourne un chiffré de $\alpha \cdot \mu$.
Algorithme : On calcule $M_\alpha = \textbf{Flatten}(\alpha \times I_N)$ puis l'on renvoie $\textbf{Flatten}(M_\alpha \times C)$.

Démonstration.

$$\begin{aligned}\textbf{MultConst}(C,\alpha)\times\vec{v} &= M_\alpha \times C \times \vec{v}\\ &= M_\alpha \cdot (\mu * \vec{v} + \vec{e})\\ &= M_\alpha \times \mu * \vec{v} + M_\alpha \times \vec{e}\\ &= \alpha * \mu * \vec{v} + M_\alpha \times \vec{e}\end{aligned}$$

□

Erreur : Le chiffré à une erreur $e_2 = M_\alpha \times \vec{e}$.

$$\|e_2\|_\infty \leq N\|e_1\|_\infty$$

Add

Entrée : Cet algorithme prend en entrée les paramètres du système et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ des messages $\mu_1, \mu_2 \in \mathbb{Z}_q$.
Sortie : Cet algorithme retourne un chiffré de $\mu_1 + \mu_2$.
Algorithme : On calcule et on retourne $\textbf{Flatten}(C_1 + C_2)$.

Démonstration.

$$\begin{aligned}\textbf{Add}(C_1,C_2)\times\vec{v} &= (C_1 + C_2)\times\vec{v}\\ &= (\mu_1 * \vec{v} + \vec{e_1}) + (\mu_2 * \vec{v} + \vec{e_2})\\ &= (\mu_1 + \mu_2) * \vec{v} + \vec{e_1} + \vec{e_2}\end{aligned}$$

□

Erreur : Le chiffré à une erreur $e_3 = \vec{e_1} + \vec{e_2}$.

$$\|e_3\|_\infty \leqslant \|e_1\|_\infty + \|e_2\|_\infty$$

Mult

Entrée : Cet algorithme prend en entrée les paramètres du système et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ des messages $\mu_1, \mu_2 \in \mathbb{Z}_q$.
Sortie : Cet algorithme retourne un chiffré de $\mu_1 * \mu_2$.
Algorithme : On calcule et on retourne $\textbf{Flatten}(C_1 \times C_2)$.

Démonstration.

$$\begin{aligned}\textbf{Mult}(C_1,C_2)\times\vec{v} &= (C_1 \times C_2)\times\vec{v}\\ &= C_1 \times (\mu_2 * \vec{v} + \vec{e_2})\\ &= \mu_2 * C_1 \times \vec{v} + C_1 \times \vec{e_2}\\ &= \mu_2 * (\mu_1 * \vec{v} + \vec{e_1}) + C_1 \times \vec{e_2}\\ &= (\mu_1 * \mu_2) * \vec{v} + \mu_2 * \vec{e_1} + C_1 \times \vec{e_2}\end{aligned}$$

□

Erreur : Le chiffré à une erreur $e_3 = \mu_2 * \vec{e_1} + C_1 \times \vec{e_2}$. La matrice C_1 étant de la forme $\textbf{Flatten}(c_1)$, elle ne contient que des 0 et des 1.

$$\|e_3\|_\infty \leqslant \mu_2\|e_1\|_\infty + N\|e_2\|_\infty$$

NAND

Entrée : Cet algorithme prend en entrée les paramètres du système et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ des messages $\mu_1, \mu_2 \in \{0,1\}$.
Sortie : Cet algorithme retourne un chiffré de $\overline{(\mu_1 \wedge \mu_2)} = 1 - \mu_1 * \mu_2$.
Algorithme : On calcule et on retourne $\mathbf{Flatten}(I_N - C_1 \cdot C_2)$.

Démonstration.

$$\begin{aligned}\mathbf{NAND}(C_1, C_2) \times \vec{v} &= (I_N - C_1 \times C_2) \times \vec{v} \\ &= \vec{v} - \mathbf{Mult}(C_1, C_2) \\ &= \vec{v} - (\mu_1 * \mu_2) * \vec{v} - \mu_2 * \vec{e_1} + C_1 \times \vec{e_2} \\ &= (1 - \mu_1 * \mu_2) * \vec{v} - \mu_2 * \vec{e_1} - C_1 \times \vec{e_2}\end{aligned}$$

□

Erreur : Le chiffré à une erreur $e_3 = -(\mu_2 * \vec{e_1} + C_1 \times \vec{e_2})$. On est dans un contexte similaire à $\mathbf{Mult}(C_1, C_2)$, mais μ_2 est ici égal à 0 ou 1.

$$\|e_3\|_\infty \leqslant \|e_1\|_\infty + N\|e_2\|_\infty \leqslant (N+1)\max(\|e_1\|_\infty, \|e_2\|_\infty)$$

6 Analyse du cryptosystème : sécurité, profondeur des circuits

6.1 Sécurité du cryptosystème

Définition 7. *Distance statistique Soit X et Y deux variables aléatoires supportée par un ensemble $\mathcal{V}$ et à valeur dans un groupe abélien G. On définit la distance statistique entre X et Y, notée $SD(X,Y)$, comme étant la somme :*

$$\frac{1}{2}\sum_{v\in\mathcal{V}} |\mathbb{P}(X = v) - \mathbb{P}(Y = v)|$$

De plus, on dira que deux familles $\{X_i\}_{i\in\mathbb{N}}$, $\{Y_i\}_{i\in\mathbb{N}}$ de distributions sont statistiquement indistinguables si la fonction

$$i \longrightarrow SD(X_i, Y_i)$$

est négligeable.

Proposition 8. *Soit $(X_i)_{1\leq i\leq n}$ et resp $.(Y_i)_{1\leq i\leq n}$ deux n-uplets de distributions indépendantes.*

$$SD\left((X_1, \cdots, X_n), (Y_1, \cdots, Y_n)\right) \leq \sum_{i=1}^{n} SD(X_i, Y_i)$$

Démonstration. Montrons le pour $n = 2$, la suite se déduisant par récurrence. On a :

$$\begin{aligned}&\mathrm{SD}\left((X_1, X_2), (Y_1, Y_2)\right) \\ &= \frac{1}{2}\sum_{(u,v)} |\mathbb{P}(X_1 = u)\mathbb{P}(X_2 = v) - \mathbb{P}(Y_1 = u)\mathbb{P}(Y_2 = v)| \\ &\leq \frac{1}{2}\sum_{(u,v)} |\mathbb{P}(X_1 = u)\left(\mathbb{P}(X_2 = v) - \mathbb{P}(Y_2 = v)\right) - \left(\mathbb{P}(X_1 = u) - \mathbb{P}(Y_1 = u)\right)\mathbb{P}(Y_2 = v)| \\ &\leq \frac{1}{2}\sum_{(u,v)} \mathbb{P}(X_1 = u)\,|(\mathbb{P}(X_2 = v) - \mathbb{P}(Y_2 = v))| + \frac{1}{2}\sum_{(u,v)} \mathbb{P}(Y_2 = v)\,|(\mathbb{P}(X_1 = u) - \mathbb{P}(Y_1 = u))| \\ &= \mathrm{SD}(X_1, Y_1) + \mathrm{SD}(X_2, Y_2)\end{aligned}$$

□

Proposition 9. *Si deux familles de distributions sont statistiquement indistinguables, elles sont calculatoirement indistinguables.*

Le lemme suivant correspond au Claim 5.2 présent dans [**?**] et nous sera utile pour la suite.

Lemme 1. *Soit G un groupe abélien fini. Pour $r > 1$ et $\mathcal{F} \subset (g_1, \ldots, g_r) \in G^r$, on note $s_\mathcal{F}$ la distribution aléatoire qui à un aléa fait correspondre la somme $\sum_{i \in X} g_i$ pour un sous-ensemble choisi de façon uniforme $X \subset [\![1, r]\!]$. D'autre part, on considère la distribution uniforme U sur G Alors, on a :*

$$\mathbb{E}_{\mathcal{F} \subset G^r}(SD(s_\mathcal{F}, U)) \leq \sqrt{\frac{|G|}{2^r}}$$

Notamment,

$$\mathbb{P}\left(SD(s_\mathcal{F}, U) \geq \sqrt[4]{\frac{|G|}{2^r}}\right) \leq \sqrt[4]{\frac{|G|}{2^r}}$$

Démonstration. Remarquons que :

$$\begin{aligned}\sum_{h \in G} \mathbb{P}(s_F = h)^2 &= \mathbb{P}\left(\sum_i b_i g_i = \sum_i b'_i g_i\right) \\ &\leq \frac{1}{2^l} + \mathbb{P}\left(\sum_i b_i g_i = \sum_i b'_i g_i |\ (b_i)_i \neq (b'_i)_i\right)\end{aligned}$$

Or, pour $(b_i)_i \neq (b'_i)_i$,

$$\mathbb{P}\left((g_i)_i\ :\ \sum_i b_i g_i = \sum_i b'_i g_i\right) = \frac{1}{|G|}$$

D'où on déduit que :

$$\mathbb{E}_\mathcal{F}\left(\sum_h \mathbb{P}(s_\mathcal{F} = h)^2\right) \leq \frac{1}{2^l} + \frac{1}{|G|}$$

Ce qui implique que :

$$\begin{aligned}\mathbb{E}_\mathcal{F}\left[\sum_h \left|\mathbb{P}(s_F = h) - 1/|G|\right|\right] &\leq \mathbb{E}_\mathcal{F}\left[|G|^{1/2}\left(\sum_h \left(\mathbb{P}(s_F = h) - 1/|G|\right)^2\right)^{1/2}\right] \\ &= \sqrt{|G|}\ \mathbb{E}_\mathcal{F}\left[\left(\sum_h \mathbb{P}(s_F = h)^2 - 1/|G|\right)^{1/2}\right] \\ &\leq \sqrt{|G|}\ \left(\mathbb{E}_\mathcal{F}\left[\sum_h \mathbb{P}(s_F = h)^2\right] - \frac{1}{|G|}\right)^{1/2} \\ &\leq \sqrt{\frac{|G|}{2^l}}\end{aligned}$$

□

Corolaire 1. *Soit G un groupe abélien fini. Pour $r > 1$ et $\mathcal{F} \subset (g_1, \ldots, g_r) \in G^r$, on note $s_\mathcal{F}$ la distribution aléatoire qui à un aléa fait correspondre la somme $\sum_{i \in X} g_i$ pour un sous-ensemble choisi de façon uniforme $X \subset [\![1, r]\!]$. Considérons alors le n-uplet $S_\mathcal{F} = (X_1, \ldots, X_r)$ où les X_i sont indépendants de même loi $s_\mathcal{F}$. D'autre part, on considère la distribution uniforme U sur G^r Alors, on a :*

$$\mathbb{E}_{\mathcal{F} \subset G^r}(SD(s_\mathcal{F}, U)) \leq \sqrt{r^2 \frac{|G|}{2^r}}$$

Notamment,

$$\mathbb{P}\left(SD(s_\mathcal{F}, U) \geq \sqrt[4]{r^2 \frac{|G|}{2^r}}\right) \leq \sqrt[4]{r^2 \frac{|G|}{2^r}}$$

Démonstration. Découle directement de la proposition précédente ainsi que de la proposition 8 □

Proposition 10. *Supposons avoir pris des paramètres (n, q, χ, m) tels que l'hypothèse $LWE_{n,q,\chi}$ soit vraie. Alors pour $\epsilon > 0$ et $m > (1+\epsilon)(n+1)\log(q)$ et $m = \mathcal{O}(n\log(q))$, la distribution jointe (A, RA) est calculatoirement indistinguable de la distribution uniforme sur $\mathbb{Z}_q^{m\times(n+1)} \times \mathbb{Z}_q^{N\times(n+1)}$*

Démonstration. On peut deja voir que comme A est calculatoirement indistinguable de U, (A, RA) l'est de (U, RU) car on peut facilement créer (A, RA) (resp. (U, RU)) à partir de A (resp. U).

Il nous faut donc monter que $\mathcal{D}_1 = (U, RU)$ est calculatoirement indistinguable de $\mathcal{D}_2 = (U, V)$ où V est uniforme.

On peut alors utiliser le lemme précédent avec $G = \mathbb{Z}_q^{n+1}$ et $r = m$ afin de voir qu'il echiste une constante $\lambda > 0$ telle que :

$$\mathbb{E}_{\mathcal{U}\subset\mathbb{Z}_q^{m\times n+1}}(SD(RU, V)) \leq \sqrt{m^2\frac{q^{n+1}}{2^m}} \leq \lambda n\log(q)\sqrt{\frac{1}{q^{\epsilon(n+1)}}} =: f(n)$$

Et, notant $Y = \{\ U : \mathrm{SD}(RU, V) \geq \sqrt{f(n)}\}$, on obtient :

$$\mathbb{P}(U \in Y) \leq \sqrt{f(n)}$$

où f est négligeable en n.

Soit $(x, y) \in \mathbb{Z}_q^{m\times(n+1)} \times \mathbb{Z}_q^{m\times(n+1)}$

$$\begin{aligned}
&|\mathbb{P}(D_1 = (x, y)) - \mathbb{P}(D_2 = (x, y))| \\
&\leq \mathbb{P}(x \in Y)\left|\mathbb{P}(D_1 = (x, y)|x \in Y) - \mathbb{P}(D_2 = (x, y)|x \in Y)\right| + \mathbb{P}(x \notin Y) \\
&\leq |\mathbb{P}(D_1 = (x, y)|x \in Y) - \mathbb{P}(D_2 = (x, y)|x \in Y)| + \sqrt{f(n)} \\
&\leq 2\sqrt{f(n)}
\end{aligned}$$

Ainsi, il n'est pas possible qu'un automate $\mathcal{A}$ polynomial probabiliste puisse distinguer $\mathcal{D}_1$ de $\mathcal{D}_2$ car elle sont statistiquement indistinguables. □

Théorème 1. *Sous les hypothèses de la proposition précédente, le cryptosysteme est IND-CPA.*

Démonstration. Comme un automate polynomial probabiliste ne peut pas distinguer $A_{s,\chi}$ de la distribution uniforme, on peut supposer que la clef publique A est uniforme.

Considérons alors un chiffré

$$C = \mathrm{Flatten}\left(\mu \cdot \mathrm{I}_N + \mathrm{BitDecomp}(R \cdot A)\right) \in \mathbb{Z}_q^{N\times N}$$

On a :

$$\mathrm{BitDecomp}^{-1}(C) = \mu * \mathrm{BitDecomp}(\mathrm{I}_N) + R \cdot A$$

Par la proposition précédente, un automate polynomial probabiliste $\mathcal{A}$ ne peut pas distinguer $R \cdot A$ d'une matrice uniforme. On peut donc supposer que $R \cdot A$ est uniforme, et que le chiffrement est donc un one-time pad.

On en déduit qu'il n'existe pas d'automate polynomial probabiliste $\mathcal{A}$ permettant de déchiffrer efficacement les chiffrés de ce cryptosystème. □

6.2 Contraintes asymptotiques pour les choix de paramètres

Le choix des paramètres du système est une étape cruciale de la mise en place de celui-ci, car ce sont eux qui détermineront les degrés de sécurités et la profondeur des circuits calculables.

Sur ce point, deux approches sont possibles : une étude asymptotique ou bien une étude « concrète »des paramètres.

Nous nous interesserons ici uniquement à l'algorithme de déchiffrement **Dec**, nous ne considèrerons donc que des chiffrés de 0 ou 1 et nous interesserons principalement à la profondeur maximale possible de NAND permettant encore de déchiffrer.

6.2.1 Analyse de la profondeur des circuits

Si χ est B-bornée, on voit en appliquant itérativement la minoration de bruit pour l'opération NAND que pour une profondeur de L NAND, le bruit est majoré en norme infinie par :

$$(N+1)^L B$$

Or, afin de fonctinner, l'algorithme de dechiffrement **Dec** necessite que la norme infinie du bruit soit majorée par q/8. On obtient donc la condition suivante :

$$L \leq \frac{\log(q) - 3 - \log(B)}{\log(N+1)} \tag{1}$$

6.2.2 Résumé des contraintes sur les paramètres

Nous indiquons ici quelques contraintes asymptotiques précisées dans la section 3.4 de [**?**]. Nous n'avons pas essayé de comprendre les raisons qui poussent à celles-ci car cela aurait necessité une étude bien plus poussée des différentes façon d'attaquer LWE.

L'article conseille de faire croitere n linéairement avec $\log(q/B)$, et, notant $\kappa = \log(q)$, d'avoir :

$$\kappa = O(L\log(N)) = O(L(\log(n) + \log(\kappa)) = O(L\log(n))$$

6.3 Choix concrets de paramètres

6.3.1 Présentation de lwe_estimator

Initialement utilisé dans l'article [**?**],`lwe_estimator` (disponible à l'adresse [**?**]) est un module de sagemath actuellement maintenu par Martin Albrecht et destiné à estimer la résistance face à diverses attaques de paramètres précis pour le problème de learning with error.

estimate_lwe :

Cependant, l'intérêt premier de ce module est qu'il estime la résistance des paramètres choisis à plusieurs attaques. Pour cela, on utilise la fonction `estimate_lwe` :

```
estimate_lwe(n, alpha=None, q=None, secret_distribution=True, m=oo,
             reduction_cost_model=reduction_default_cost,
             skip=("mitm", "arora-gb", "bkw"))
```

Cette dernière prends en arguments les paramètres usuels de LWE, n, α et q, ainsi que d'autres arguments optionnels et rend plusieurs résultats dont le sens n'est pas forcément évident. En fait, elle retourne tout un ensemble de variables pour chaque attaque vérifiée. Le module contient 6 attaques différentes, mais n'en testera que trois par defaut. Cela peut être modifié lorsque l'on appelle la fonction `estimate_lwe` via l'argument skip.

FIGURE 1 DEVRAIT ÊTRE ICI

Les variables rendues pour chaque attaques ne sont pas toutes utiles, certaines étant strictement internes à ces attaques. Les trois variables qui nous intéresse sont : "rop", "m" et "mem" :

- "rop" (ring operations) est une estimation du nombre d'opérations à effectuer afin de résoudre ce cas de LWE avec cette attaque.
- "mem" (memory) est une estimation de la mémoire qui sera exploitée.
- "m" indique le nombre d'échantillons nécessaires pour résoudre le problème avec ces valeurs de "rop" et "mem". A noter que l'on peut limiter le nombre d'échantillons disponibles pour l'attaquant lorsqu'on appelle `estimate_lwe`.

le module `sage.crypto.lwe` contient des choix de paramètres pour le problème LWE, dont deux qui utilisent la gaussienne discrète [3] et que nous allons présenter ici. Notons que nous avons pour cela regardé les codes sources des fonctions sage, disponibles dans leur github ([**?**]).

3. un autre utilise la distribution uniforme

```
sage: load("estimator.py")
sage: n = 2048; q = 2^60 - 2^14 + 1; α = 8/q; m = 2*n
sage: _ = estimate_lwe(n, α, q, secret_distribution=(-1,1),
          reduction_cost_model=BKZ.sieve, m=m)
usvp: rop: =2^115.5,  red: =2^115.5,  δ_0: 1.004975,
      β: 288,  d: 4013,  m: 1964
 dec: rop: =2^127.1,  m:  =2^11.1,  red: =2^127.1,  δ_0: 1.004663,
      β: 318, d: 4237,
      babai: =2^114.8,  babai_op: =2^129.9,  repeat: 7,  ϵ: 0.500000
dual: rop: =2^118.4,  m:  =2^11.0,  red: =2^118.4,  δ_0: 1.004864,
      β: 298,
      repeat:  =2^58.8,  d: 4090,  c:    3.909,  k: 30, postprocess: 13
```

FIGURE 1 – Analyse de sécurité des paramètres tirés de la librairie SEAL

6.3.2 Proposition de choix sécurité pour très faible profondeur

En utilisant les paramètres suivants, tirés de l'API SEAL :

$$n = 2048 \quad q = 2^{60} - 2^{14} + 1 \quad \alpha = \frac{8}{q} \quad m = 2n$$

On voit que l'estimation proposée par lwe indique l'attaque la plus rapide demande 2^{115} opérations de base dans l'anneau $\mathbb{Z}_q$, soit un facteur de sécurité de 115. De plus, l'équation (1) indique qu'une profondeur de NAND $L = 3$ est possible.
On a ici un secret de 15 Ko, une clé publique de 7.6 Mo et des chiffrés de 13 Go.

En utilisant les paramètres suivants, tirés de [**?**] :

$$n = 804 \quad q = 2^{31} - 19 \quad \alpha = \frac{\sqrt{2\pi} * 57}{q} \quad m = 4972$$

On voit que l'estimation proposée par lwe indique l'attaque la plus rapide demande 2^{129} opérations de base dans l'anneau $\mathbb{Z}_q$, soit un facteur de sécurité de 129. De plus, l'équation (1) indique qu'une profondeur de NAND $L = 1$ est possible.
On a ici un secret de 3 Ko, une clé publique de 5 Mo et des chiffrés de 2 Go.

7 Mise en place d'un bootstrapping

7.1 Un point sur la sécurité

Nous avons vu que le système cryptographique que nous étudions est IND-CPA, ce qui est le niveau de sécurité théorique que l'on veut généralement. La première question qui se pose pour le bootstrapping est de savoir si l'on garde ce niveau de sécurité.

Le problème est que pour effectuer un déchiffrement homomorphe, il faut au moins un chiffré de la première clé secrète par la deuxième clé publique, ou, dans notre cas, un chiffré de chaque bit du secret. Il faut pouvoir s'assurer qu'un attaquant soit incapable d'en tirer des informations sur la première clé. On appelle cela la sécurité circulaire.

Comme nous l'avons rappelé, le système est IND-CPA, ce qui implique que cette propriété est toujours vérifiée lorsque les deux clés sont indépendantes l'une de l'autre. Il suffit donc de générer les clés indépendamment les unes des autres pour s'assurer d'avoir le niveau de sécurité désiré.

7.2 Un premier découpage

Afin de pouvoir effectuer un bootstrapping à partir de l'algorithme de déchiffrement **Dec**, nous allons avoir besoin de l'exprimer uniquement à partir d'opérations **NAND** sur des 0 et des 1.

La première clé secrète est un vecteur d'éléments de $\mathbb{Z}_q$, pas de $\{0,1\}$, ce qui signifie que pour pouvoir l'utiliser dans le bootstrapping, on est contraint de se servir des chiffrés de chaque bit de chaque élément de la clé.

Pour cela, on considère qu'à tous les éléments de $\mathbb{Z}_q$ est associé une liste de $l-1$ éléments contenant son écriture en binaire.

```
decrypt(C) :
        trouver 1 ≤ i ≤ l tel que q/4 ≤ 2^i < q/2
        calculer a = C_i · v
        retourner |a/v_i|
```

Notons que :

$$\begin{aligned} C_i \cdot \vec{v} &= \sum_{j=0}^{N} C_{i,j} v_j \\ &= \sum_{j=0}^{N} \sum_{k=0}^{l} \left(C_{i,j,k} 2^k \right) v_j \\ &= \sum_{j=0}^{N} \sum_{k=0}^{l} C_{i,j,k} (2^k v_j) \end{aligned}$$

Or, même lorsque l'algorithme se fait homomorphiquement, les valeurs $C_{i,j,k} \in \{ 0, 1 \}$ sont connues. On réduit donc le problème de calculer un produit scalaire à celui de faire la somme d'au plus $l * N$ vecteurs constitués de 0 et de 1.

Nous allons maintenant voir comment décrire l'algorithme uniquement avec des portes **NAND**, et majorer la profondeurs en **NAND** de l'algorithme final.

Comme nous utiliserons aussi les portes logiques `NO`, `AND`, `OR` et `XOR`, notons que :

— $\texttt{NO}(a) = \overline{a}$ se fait en un **NAND** ;
— $\texttt{AND}(a, b) = a \wedge b$ se calcule en deux **NAND** et est de profondeur 2 ;
— $\texttt{OR}(a, b) = a \vee b$ se fait en trois **NAND** et est de profondeur 2 ;
— $\texttt{XOR}(a, b) = a \oplus b$ se calcule en six **NAND** et est de profondeur 4.

7.3 Sommer des vecteurs avec une profondeur en NAND minimale

Nous voulons pouvoir sommer homomorphiquement nb nombres vus comme des listes de taille s dont les éléments sont des chiffrés de 0 ou de 1 en minimisant la profondeur de NAND requise.

Pour cela, nous allons tout d'abord étudier deux opérations simples sur des listes.

basic_sum : addition classique de deux listes :

Il s'agit de l'algorithme naïf de somme de deux nombres binaire, commençant par les bits de poids faible puis remontant vers les bits de poids plus élevés en conservant des retenues, sauf celle qui « sort »des listes. On l'appellera ici **basic_sum**. Soient

$$A = \sum_{i=0}^{s-1} a_i 2^i \quad B = \sum_{i=0}^{s-1} b_i 2^i$$

que l'ont veux sommer. La somme

$$C = \sum_{i=0}^{s-1} c_i 2^i$$

est alors définie par :

```
r_{-1} = 0
for i in range(s):
        c_i = a_i ⊕ b_i ⊕ r_{i-1}
        r_i = (a_i ∧ b_i) ∨ (r_{i-1} ∧ (a_i ∨ b_i))
```

Le problème de cette méthode est que la profondeur de NAND nécessaire explose du fait que la formule exprimant c_{s-1} dépend de a_0 et b_0, et ce à cause de la récursivité du calcul des $r - i$.

Calculer c_i peut se faire en n'utilisant la retenue que pour un $\oplus$, ce qui n'ajoute que 4 à la profondeur en **NAND** du calcul. r_i est calculé en appliquant un `AND` et un `OR` à r_{i-1}, ce qui ajoute aussi 4 à la profondeur. $r_0 = a_0 \wedge b_0$ et peut donc être trouvé avec une profondeur de 2 **NAND**.

On obtient alors la proposition suivante :

Proposition 11. *L'algorithme* ***basic_sum*** *nécessite une profondeur de* $4 * s - 2$ ***NAND****.*

reduced_sum :

Soient

$$A = \sum_{i=0}^{s-1} a_i 2^i, \quad B = \sum_{i=0}^{s-1} b_i 2^i, \quad C = \sum_{i=0}^{s-1} c_i 2^i.$$

reduced_sum va créer deux nombres qu'on nomme $X = \sum_{i=0}^{s-1} x_i 2^i$ et $Y = \sum_{i=0}^{s-1} y_i 2^i$ tels que $A + B + C = X + Y$.

Ils se construisent ainsi, avec la convention que pour tout vecteur v, $v_i = 0 \ \forall i \notin [\![0, s-1]\!]$:

```
for i in range(s):
```

$$x_i = a_i \oplus b_i \oplus c_i$$

$$y_i = \overline{(\overline{a_{i-1}} \wedge \overline{b_{i-1}}) \oplus (\overline{b_{i-1}} \wedge \overline{c_{i-1}}) \oplus (\overline{a_{i-1}} \wedge \overline{c_{i-1}})}$$

On remarque qu'ici, les coordonnées des résultats ne dépendent que des coordonnées voisines. La profondeur totale en NAND sera donc le max de la profondeur du calcul de x_i et de celle du calcul de y_i.

Calculer x_i consiste en deux $\oplus$ successifs dont le deuxième utilise le résultat du premier. La profondeur en **NAND** est donc de 8. La profondeur maximale du calcul de y_i est celle des éléments impliqués dans deux $\oplus$. Ces éléments subissent donc deux `NO`, un `AND` et deux `XOR`, atteignant ainsi une profondeur de 12 **NAND**.

Proposition 12. *L'algorithme* ***reduction_sum*** *nécessite une profondeur de* 12 ***NAND****.*

On comprend donc que pour sommer plusieurs listes, nous avons tout intérêt à utiliser **reduced_sum** jusqu'à qu'il ne reste plus que deux listes, puis à utiliser **basic_sum**, mais comment organiser ces "additions" de façon à minimiser la profondeur totale ?

organiser la somme des nb **listes :**

L'algorithme naïf consisterai dans le premier cas à appliquer **basic_sum** à deux listes puis à l'appliquer à chaque fois au résultat de la dernière somme avec une autre liste. Les deux variables additionnées lors de la première addition serait impliquées dans chacune des $nb - 1$ opérations et la profondeur en **NAND** serait donc de $(nb - 1)(4s - 2)$.

Dans le cas de **reduced_sum**, l'algorithme naïf consisterai en : appliquer **reduced_sum** à trois des listes à additionner, puis en choisir une autre et appliquer **reduced_sum** à cette dernière accompagnée des deux listes précédemment retournée par **reduced_sum** et ainsi de suite, jusqu'à ce qu'il ne reste plus de liste à ajouter au dernier résultat. On y applique alors **basic_sum**. On atteint ainsi une profondeur de $12(nb - 2) + 4s - 2 = 4(3nb + s) - 26$ **NAND**.

Les additions peuvent cependant être bien mieux ordonnées :

Par exemple, en visualisant l'organisation comme un arbre, un arbre équilibré réduit au maximum la profondeur et permet à de sommer les nb listes en utilisant uniquement **basic_sum** avec une profondeur en **basic_sum** de $\log_2(nb)$. On obtient donc une profondeur en **NAND** en $\mathcal{O}(\log(nb) * (4s - 2)) = \mathcal{O}(s * \log(nb))$ et non plus en $\mathcal{O}(s * nb)$.

Si l'on construit un équivalent pour **reduced_sum**, toujours en terminant par un **basic_sum**, on obtient une profondeur en **NAND** en $\mathcal{O}(\log(nb) + s)$.

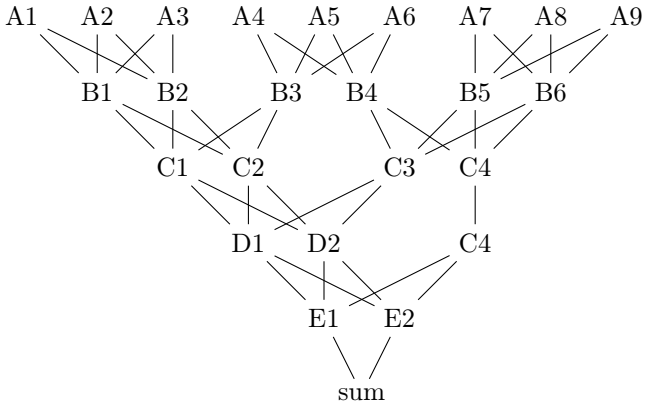
Proposition 13. *Il est possible de sommer nb vecteurs de s éléments avec une profondeur de* $4s + 36 \log_3(nb) - 22$ ***NAND****.*

Démonstration. Soit $p \geqslant 3$ tel que $3^{p-1} < nb \leqslant 3^p$.

En réunissant les listes à notre disposition par groupes de 3 et en leur appliquant **reduced_sum**, puis en recommançant deux fois on se retrouve avec au plus $8 * 3^{p-3} < 3^{p-1}$ listes à sommer.

On peut donc se ramener à $nb' \leqslant 9$ avec une profondeur en **reduced_sum** de $3(p-2)$.

De là, on peut calculer la somme totale avec une profondeur de quatre **reduced_sum** et un **basic_sum** :



On obtient ainsi une profondeur totale en **NAND** de $4s - 2 + 12(3p-2) = 4s + 36p - 22$. Or, par définition, $p = \lceil \log_3(nb) \rceil$ donc on a bien une profondeur en **NAND** de $4s + 36\log_3(nb) - 22$. □

7.4 Prendre la valeur absolue dans $\mathbb{Z}_q$

Rappelons que la valeur absolue d'un élément $x \in \mathbb{Z}_q$ est par définition la valeur absolue de son représentant dans $[\![-q/2, q/2]\!]$.

Dans notre situation, $q = 2^l$ et nous représentons $a \in \mathbb{Z}_q$ par une liste de taille[4] $l-1$, le bit de poids faible étant à gauche. Autrement dit :

$$a = [a_0, ..., a_{l-1}] \quad \text{pour représenter } a = \sum_{i=0}^{l-1} a_i 2^i$$

On peut alors calculer en binaire la valeur absolue de a ainsi :

```
if a_{l-1} = 0:
        # on a a < q/2
        return a
else:
        # on a a < q/2, alors |a - q| = ((2^l - 1) - a + 1)
        a = [NOT(a_i) for i in range(l)]
        return basic_sum(a, 1)
```

Toutefois, il n'est pas possible de faire de conditions homomorphiquement avec ce système, aussi, afin de pouvoir l'écrire homomorphiquement, on va en fait considérer le code suivant :

```
b = basic_sum([NOT(a_i) for i in range(l)], 1)
return [(a_{l-1} ∧ a_i) ∨ (¬a_{l_1} ∧ b_i) for i in range(l)]
```

On peut alors utiliser les comptes déjà fait, notamment concernant la profondeur en **NAND** de **basic_sum**, pour conclure :

Proposition 14. *En conservant nos conventions pour la représentation des données, prendre la valeur absolue d'un élément $a \in \mathbb{Z}_q$ demande une profondeur de $4s+2$ **NAND**.*

En additionnant nos différents comptes, on peut alors conclure :

Proposition 15. *On peut effectuer l'algorithme **Dec** avec une profondeur de RESULTAT NAND.*

4. Pour des raisons techniques, il est en fait représenté par une liste de taille l, mais nous ne faisons alors pas attention au dernier bit

8 Implémentation d'un FHE avec bootstrapping « jouet »

8.1 Présentation de notre arborescence

Nous avons concu une implémentation simple du cryptosystème GSW en sagemath, située dans le dossier `GSW_implementation`. Celui-ci contient quatres dossiers :

- `GSW_scheme` contenant l'implémentation de GWS ;
- `analysis` contenant des fonctions permettant de tester les fonctionnalités de notre implémentation ou encore de voir les performances en terme de sécurités de certains choix de paramètres ;
- `unitary_test` contenant des tests assurant le bon fonctionnement des fonctions codées dans les autres dossiers ;
- `lwe_estimator` contient les fichiers sources de l'API lwe_estimator que nous avons présenté dans ce rapport ;

Nous proposons ici de faire une revue rapide des trois premiers dossiers. Notez que pour « attacher »avec sage un des sources, il faut rester à la racine pour éviter des problèmes liés à l'utilisation de chemin relatifs pour les imports.

8.1.1 GWS_scheme

Ce dossier contient les fichiers suivants :

- `GSW_scheme.sage` contient les fonctions principales de GWS dont `setup`, `encrypt` et les 3 algorithmes de déchiffrements que sont `basic_decrypt`, `mp_decrypt` et `mp_all_q_decrypt`. Il contient aussi différentes variables globales, dont `decrypt` permettant d'indiquer quel est l'algorithme de dechiffrement par défault et les variables `bs_foo` indiquant quels paramètres sont utilisés lorsque on utilise des fonctins avec bootstrapping ;
- `auxilliary_functions.sage` contient l'implémentation des diverses fonctions auxilliaires utilisées pour chiffrer et déchiffrer les messages, comme par exemple `flatten` ; ou encore une implémentation du nearest plane de Babai ;
- `params_maker.sage` contient diverses fonctions permettant, à partir d'un n, de retourner des paramètres n, q, χ, m utilisés par le cryptosystème. Le fichier `GWS_scheme.sage` contient une variable globale `params_maker` permettant de fichier celui qu'utilise la fonction `sectup` ;
- `homomorphic_functions.sage` contient la version homomorphe d'opérations de base comme la somme, ou encore le NAND ;
- `bootstrapping.sage` contient les fonctions necessaires pour effectuer l'algorithme `basic_decrypt` homomorphiquement (il s'agit de la fonction `h_basic_decrypt`). On y trouve donc notamment diverses façon de sommer homomorphiquement des listes de chiffrés de 0 et de 1. Notez que la fonction `bootstrapping_arguments` permet de faire un bootstrapping en retournant une valeur « mise à jour »des chiffrés passés en argument, cette fonction est notamment utilisée dans `analysis/h_circuits_with_bootstrapping.sage`.

8.1.2 analysis

Ce dossier contient les fichiers suivants :

- `depth_security.sage` contenant l'implémentation de GWS ;
- `circuits.sage` contenant des fonctions permettant de tester les fonctionnalités de notre implémentation ou encore de voir les performances en terme de sécurités de certains choix de paramètres ;
- `clear_functions.sage` contient des versions « en clair »de fonctions homomorphes, utilisées dans les circuits ;
- `h_circuits_with_bootstrapping.sage` contient des exemples de fonctions utilisant des bootstrappings. Elles permettent de voir si, pour certaines fonctions f, appliquer f homomorphiquement sur des chiffrés revient au même que d'abord l'appliquer sur les clairs puis chiffrer. On peut toutes les lancer en utilisant la fonction `all_circuit_without_bs` ;
- `h_circuits_without_bootstrapping.sage` contient des examples de fonctions n'utilisant pas de bootstrappings. fonctions utilisant des bootstrappings. Elles permettent de voir si, pour certaines fonctions f, appliquer f homomorphiquement sur des chiffrés revient au même que d'abord l'appliquer sur les clairs puis chiffrer. On peut toutes les lancer en utilisant la fonction `all_circuit_with_bs` ;

— `circuits.sage` Contient des fonctions permettant d'écrire sous forme de string des fonctions algébriques simples, ce qui est utilisé dans `h_circuits_without_bootstrapping.sage`. Par exemple, on peut écrire `abc|*c+a~bc` pour signifier la fonction

$$(a, b, c) \mapsto c * (a + (b\text{NAND}c))$$

;
— `all_circuit_analysis.sage` contient la fonction `analysis_main` qui lance les différents circuits avec et sans bootstrappings des fichiers précédents.

8.1.3 unitary_test

Ce dossier contient un fichier `framework_test.sage` permettant de mettre en forme les sorties des différentes fonctions de test, puis un fichier de test correspondant à chaque fichier du dossier `GSW_scheme`. Chacun de ses fichiers contient une fonction `test_main_FOO` ne demandant aucun argument et permettant de lancer les différents tests qu'il contient. De plus, le fichier `all_main_test.sage` contient une fonction `test_main` permettant de lancer toutes les fonctions de forme `test_main_FOO`. On peut donc se faire une idée du travail réalisé sur les tests en la lancant.

— `depth_security.sage` contenant l'implémentation de GWS ;
— `circuits.sage` contenant des fonctions permettant de tester les fonctionnalités de notre implémentation ou encore de voir les performances en terme de sécurités de certains choix de paramètres ;
— `clear_functions.sage` contenant des tests assurant le bon fonctionnement des fonctions codées dans les autres dossiers ;
— `h_circuits_with_bootstrapping.sage` contient les fichiers sources de l'API lwe_estimator que nous avons présenté dans ce rapport ;
— `h_circuits_without_bootstrapping.sage` contient les fichiers sources de l'API lwe_estimator que nous avons présenté dans ce rapport ;

9 Des librairies pour du FHE

Plusieurs librairies open-sources implémentant divers FHE sont disponibles. On peut notamment en trouver une liste sur HomomorphicEncryption.org [**?**], qui se décrit comme « an open consortium of industry, government and academia to standardize homomorphic encryption ».

Nous proposons ici d'en évoquer deux :

— The Simple Encrypted Arithmetic Library (SEAL) [**?**], dont nous avons tiré des paramètres « réalistes » [5] sécurisés et autorisant une profondeur de NAND non null (même si irréaliste : seulement 3) ;
— The Gate Bootstrapping API [**?**] qui implémente une variation du cryptosystème GSW ;

9.1 La librarie SEAL

Acronyme de Simple Encrypted Arithmetic Library, SEAL (voir [**?**]) est une librairie écrite par le « cryptography research group »de Microsoft, en C++ sous licence MIT. Elle se propose d'implémenter deux FHE de seconde génération : BVS [**?**] et CKKS [**?**].

Son installation est facile [6] et il est directement possible de compiler un executable permettant de tester diverses fonctionnalitées de la librarie. De plus, la documentation [**?**], malheureusement non à jour, indique quelques points théoriques autant du point de vue mathématique que des choix de représentation des données.

9.2 The Gate Bootstrapping API

Notre présentation s'appuie sur celle donnée dans la page officielle (voir [**?**]) qui est claire et bien documentée.

5. Pas forcément pour nos machines et avec notre implémentation
6. Sur Linux debian 4.9.0-8-amd64, nous avons dû utilister les backports debians pour avoir une version de cmake suffisament récente

l'API Gate Bootstrapping est une librairie open source utilisable en C, C++ et s'appuyant notamment sur des travaux de I. Chillotti, N. Gama, M. Georgieve et M. Izabachène (voir [**?**] et [**?**]).

Elle utilise une version modifiée du cryptosysteme GSW ([**?**]) étudié dans notre rapport, et permettant aussi bien du LHE que du FHE. C'est pourquoi nous allons parler plus en détail de celle-ci.

Ses performances sont interessantes ; il est notamment indiqué dans la sous-section 4.2 de [**?**] que pour un ordinateur 64-bit simple coeur (i7-4930MX) cadencé à 3.00GHz, le bootstrapping se fait en un temps moyen de 52ms. de clée de bootstrapping d'environ 24MO.

Pour arriver à de tels résultats, de nombreuses modifications et optimisations dans le codes ont été faites. Notamment, le problème sur lequel s'appuie le cryptosystème n'est plus LWE mais TFHE, présenté dans les librairies suscitées.

9.2.1 Un exemple simple

En plus d'une présentation de leur API, leur site de présentation contient un tutorial sous forme de 3 fichiers de codes simples permettant de simuler une »communication chiffrée »entre Alice et le cloud :

- Alice génère des clés, chiffre des données et les envoies ainsi que la clé de bootstrapping au cloud ;
- Le cloud applique homomorphiquement une fonction, le minimum entre deux nombres, aux données et les renvoie à Alice ;
- Alice déchiffre le résultat ;

Afin de manipuler la librairie, nous avons « mis en forme »ces fichiers en y ajoutant quelques modifications. Le tout est situé dans `using_tfhe_library` et il suffit de faire `make` pour compiler l'executable, sous couvert d'avoir la librairie tfhe installée.

Les fichiers sources ainsi que les hearders contiennent normalement assez de commentaire pour être lisibles. Nous proposons donc ici de résumer brièvement le rôle de chaque fichier source :

- `alice.c` contient des fonctions permettant de générer clés, chiffrer et déchiffrer ;
- `homomorphic_functions.c` contient deux exemples de fonctions appliquables homomorphiquement : le minimum de deux nombres (déjà présent dans le tutorial) et leur somme ;
- `cloud.c` contient une fonction permetant d'appliquer homomorphiquement une des fonctions de `homomorphic_functions.c` sur des chiffrés puis d'enregistrer le résultat ;
- enfin, `example_communication.c` utilise les fichiers précédents pour simuler une communication entre Alice et le cloud.

Références

[1] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of learning with errors. Cryptology ePrint Archive, Report 2015/046, 2015. http://eprint.iacr.org/2015/046.

[2] Erdem Alkim, Nina Bindel, Johannes Buchmann, Özgür Dagdelen, Edward Eaton, Gus Gutoski, Juliane Krämer, and Filip Pawlega. Revisiting tesla in the quantum random oracle model. Cryptology ePrint Archive, Report 2015/755, 2015. https://eprint.iacr.org/2015/755.

[3] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 409–437, Hong Kong, China, December 3–7, 2017. Springer, Heidelberg, Germany.

[4] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster fully homomorphic encryption : Bootstrapping in less than 0.1 seconds. Cryptology ePrint Archive, Report 2016/870, 2016. https://eprint.iacr.org/2016/870.

[5] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Improving tfhe : faster packed homomorphic operations and efficient circuit bootstrapping. Cryptology ePrint Archive, Report 2017/430, 2017. https://eprint.iacr.org/2017/430.

[6] Secrity estimate for the learning with error problem. https://bitbucket.org/malb/lwe-estimator. Accessed : 2019-02.

[7] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. Cryptology ePrint Archive, Report 2012/144, 2012. http://eprint.iacr.org/2012/144.

[8] Craig Gentry. A fully homomorphic encryption scheme, 2009. crypto.stanford.edu/craig.

[9] Craig Gentry, Chris Peikert, and Vinod Vaikuntanathan. Trapdoors for hard lattices and new cryptographic constructions. In Richard E. Ladner and Cynthia Dwork, editors, *40th Annual ACM Symposium on Theory of Computing*, pages 197–206, Victoria, British Columbia, Canada, May 17–20, 2008. ACM Press.

[10] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors : Conceptually-simpler, asymptotically-faster, attribute-based. Cryptology ePrint Archive, Report 2013/340, 2013. http://eprint.iacr.org/2013/340.

[11] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors : Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013, Part I*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92, Santa Barbara, CA, USA, August 18–22, 2013. Springer, Heidelberg, Germany.

[12] Homomorphic Encryption Standardization. http://homomorphicencryption.org/. Accessed : 2019-02.

[13] Kim Laine. Simple Encrypted Arithmetic library 2.3.1. Microsoft Reseach, WA, USA. https ://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf.

[14] Daniele Micciancio and Chris Peikert. Trapdoors for lattices : Simpler, tighter, faster, smaller. Cryptology ePrint Archive, Report 2011/501, 2011. http://eprint.iacr.org/2011/501.

[15] Daniele Micciancio and Chris Peikert. Trapdoors for lattices : Simpler, tighter, faster, smaller. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, volume 7237 of *Lecture Notes in Computer Science*, pages 700–718, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

[16] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In Harold N. Gabow and Ronald Fagin, editors, *37th Annual ACM Symposium on Theory of Computing*, pages 84–93, Baltimore, MA, USA, May 22–24, 2005. ACM Press.

[17] Github of sage, open source mathematical software. https://github.com/sagemath/sage. Accessed : 2019-01-23.

[18] Simple Encrypted Arithmetic Library (release 3.1.0). https://github.com/Microsoft/SEAL, December 2018. Microsoft Research, Redmond, WA.

[19] A fast open-source library for fully homomorphic encryption. https://tfhe.github.io/tfhe/. Accessed : 2019-02.