Implémentation d'un FHE

Lucas Roux et Eric Sageloli

4 février 2019

Table des matières

1 Notations

2 Introduction

```
for moi in range(haha):
        je suis un oiseau
```

Listing 1 – ceci est un test

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 de $m = m(\lambda)$ 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 : où e_i est crée obtenue par χ.*

Notons qu'alors, $n = O(P(\lambda), \log(q) = O(P(\lambda))$ 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 ». Cela semble facile 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, $q = O(\text{poly}(n))$, plus polynomial en n , et le théorème 3.1 de [?] lorsque q est un produit de premiers $p_i \in O(\text{poly}(n))$, 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 $n \geq 1$ un entier, $2 \leq q \leq poly(n)$ un nombre premier et χ une distribution sur $\mathbb{Z}_q$. Supposons avoir accés 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 $\mathcal{W}$ 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 démonstration faite dans [?] L'automate W' va trouver s coordoonée par coordonnée. Montrons comment W' obtient la première coordonnée s_1.

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

$$f_k : (a, b) \mapsto (a + (l, 0, \cdots, 0), b + l \cdot k)$$

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

f_k appliqué à un échantillon uniforme donne un échantillon uniforme tandis qu'appliqué à un échantillon de $A_{s,\chi}$, il 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. Ce qui se fait en temps polynomial car $p < \text{poly}(n)$ □

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 $DLWE_{n,q,\chi}$ concisite à décider si une matrice $A \in \mathbb{Z}_q^{m\times(n+1)}$ est uniforme ou bien si il existe un vecteur $\vec{v} = (1 \quad -\vec{s})$ tel que $A \cdot \vec{v} \in \mathbb{Z}_q^m$ est crée 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 à alors trouvé un « petit »vecteur du réseau engendré par les colonnes de

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

3.2 La gaussienne discrète

Nous reprenons ici [?] 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 :

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

Pour un entier $q > 0$, Nous définissions enfin la gaussienne discrète D_α^q modulo un entier $q > 0$ comme la composition la fonction qui a $x \in \mathbb{Z}_q$ attribue

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

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

Le lemme 4.2 de [?] :

Proposition 2. *Pour tout $\epsilon > 0$, $s \geq \eta_\epsilon(\mathbb{Z})$ et tout $t > 0$:*

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

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

4 Présentation du cryptosystème

4.1 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 :

4.2 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 $(\sum\limits_{i=0}^{l-1} 2^i a_{1,i}, ..., \sum\limits_{i=0}^{l-1} 2^i a_{k,i})$.
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 $(a_1, 2a_1, 2^2 a_1, ..., 2^{l-1} a_1, a_2, ..., 2^{l-1} a_k)$.

Proposition 3. *Soient $\vec{a}$ et $\vec{b}$ dans $\mathbb{Z}_q^k$.*
On a $\langle \boldsymbol{BitDecomp}(\vec{a}), \boldsymbol{PowersOf2}(\vec{b})\rangle = \langle \vec{a}, \vec{b}\rangle$.

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

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

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

□

4.3 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 $\vec{s} = (1, -t_1, ..., -t_n)$.
On note $\vec{v} =$ **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.

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 : $C =$ **Flatten**$(\mu \times I_N +$**BitDecomp**$(R \times A))$.

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 $\frac{q}{4}$ et $\frac{q}{2}$, $\frac{q}{2}$ compris. On note C_i la ième ligne de C. On calcule ensuite $x_i = \langle C_i, \vec{v} \rangle$ et on renvoie $\lfloor \frac{x_i}{v_i} \rceil$.

4.4 Opérations homomorphes

On rappelle que $\vec{v}$ est de la forme **PowersOf2**$(\vec{s})$ et que donc **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 $M_\alpha =$ **Flatten**$(\alpha \times I_N)$ puis l'on renvoie **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 de $M_\alpha\times\vec{e}$. L'impact de celle-ci sur le déchiffrement sera donc au maximum de N fois l'impact de $\vec{e}$ quel que soi α.

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 **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 de $\vec{e_1}+\vec{e_2}$. L'impact de celle-ci sur le déchiffrement sera donc au maximum de la somme de l'impact de $\vec{e_1}$ et de l'impact de $\vec{e_2}$.

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 **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 de $\mu_2*\vec{e_1}+C_1\times\vec{e_2}$. La matrice C_1 étant de la forme **Flatten**(c_1), elle ne contient que des 0 et des 1. $C_1\times\vec{e_2}$ a donc un impact sur le déchiffrement d'au maximum N fois l'impact de $\vec{e_1}$. $\mu_2*\vec{e_1}$ n'a pas ce genre de contrainte et son impact maximum ne peut guère être limité qu'en limitant les valeurs des messages.

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 de $-(\mu_2 * \vec{e_1} + C_1 \times \vec{e_2})$. Son impact sur le déchiffrement est donc équivalent à celui d'une opération $\mathbf{Mult}(C_1, C_2)$. Cependant, μ_2 étant ici égal à 0 ou 1, on obtient un impact inférieur ou égal à l'impact de $\vec{e_1}$ plus N fois celui de $\vec{e_2}$.

5 Analyse du cryptosystème : sécurité, profondeur des circuits

5.1 Sécurité du cryptosystème

Définition 4. *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 5. *Soit $(X_i)_{1\le i\le n}$ et resp .$(Y_i)_{1\le i\le n}$ deux n-uplets de distributions indépendantes.*

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

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

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

□

Proposition 6. *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 :

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

□

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 5 □

Proposition 7. *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

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

On a :

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

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. □

5.2 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. Pour savoir quels paramètres choisir, il existe deux méthodes :

Tout d'abord, l'on peut et l'on doit étudier le sujet sur le plan théorique afin de déterminer l'ordre de grandeur des paramètres.

Cependant, cette analyse asymptotique est insuffisante pour les cas concrets. Pour palier à ce problème, l'on peut étudier un certain choix précis de paramètres en essayant de l'attaquer afin d'estimer sa résistance concrète aux attaques.

5.2.1 Analyse asymptotique de la profondeur des circuits

5.2.2 Résumé des contraintes sur les paramètres

5.3 Choix concrets de paramètres

5.3.1 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.

```
sage: n, alpha, q = Param.Regev(128)
sage: costs = estimate_lwe(n, alpha, q)
usvp: rop: 2^57.7, red: 2^57.7, δ_0: 1.009214, β: 102, d: 357, m: 228
 dec: rop: 2^61.5, m: 229, red: 2^61.5, δ_0: 1.009595, β: 93, d: 357,
     babai: 2^46.8, babai_op: 2^61.9, repeat: 293, ε: 0.015625
dual: rop: 2^81.4, m: 376, red: 2^81.4, δ_0: 1.008810, β: 111, d: 376,
     |v|: 736.521, repeat: 2^19.0, ε: 0.003906
```

Les variables rendues pour chaque attaques ne sont pas toutes utiles, certaines étant strictement internes à ces attaques. Les trois variables qui nous interresse 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 [1] et que nous allons présenter ici. Notons que nous avons pour cela regardé les codes sources des fonctions sage, disponibles dans leur github ([?]).

Les paramètres de Regev

La fonction Regev permet, à partir d'un paramètre n, de créer des paramètres n, q, χ suivant les recommandations faites dans [?] où le théorème 1.1 explique qu'alors, $\mathrm{LWE}_{n,q,\chi}$ se réduit à un problème de réseaux réputé difficile.
Plus précisément, pour n donné, il retourne

$$q = \mathrm{NextPrime}(n^2), \chi = D_\sigma^q \quad \text{où} \quad \sigma = \frac{q}{\sqrt{2\pi n}\log(n)^2}$$

```
q = ZZ(next_prime(n**2))
s = RR(1/(RR(n).sqrt() * log(n, 2)**2) * q)
D = DiscreteGaussianDistributionIntegerSampler(s/sqrt(2*pi.n()), q)
```

1. un autre utilise la distribution uniforme

Les parametres de Lindner et Peiker

Il sont définis dans [?] et on peut voir dans le listing suivant la façon dont sage crée ces paramètres.

```
- ``n`` - security parameter (integer > 0)
- ``delta`` - error probability per symbol (default: 0.01)
- ``m`` - number of allowed samples or ``None`` in which case ``m=2*n +
  128`` as in [LP2011]_ (default: ``None``)
"""
if m is None:
    m = 2*n + 128
# Find c>=1 such that c*exp((1-c**2)/2))**(2*n) == 2**-40
#          (c*exp((1-c**2)/2))**(2*n) == 2**-40
#    log((c*exp((1-c**2)/2))**(2*n)) == -40*log(2)
#        (2*n)*log(c*exp((1-c**2)/2)) == -40*log(2)
#  2*n*(log(c)+log(exp((1-c**2)/2))) == -40*log(2)
#             2*n*(log(c)+(1-c**2)/2) == -40*log(2)
#               2*n*log(c)+n*(1-c**2) == -40*log(2)
#  2*n*log(c)+n*(1-c**2) + 40*log(2) == 0
c = SR.var('c')
c = find_root(2*n*log(c)+n*(1-c**2) + 40*log(2) == 0, 1, 10)
# Upper bound on s**2/t
s_t_bound = (sqrt(2) * pi / c / sqrt(2*n*log(2/delta))).n()
# Interpretation of "choose q just large enough to allow
# for a Gaussian parameter s>=8" in [LP2011]_
q = next_prime(floor(2**round(log(256 / s_t_bound, 2))))
# Gaussian parameter as defined in [LP2011]_
s = sqrt(s_t_bound*floor(q/4))
# Transform s into stddev
stddev = s/sqrt(2*pi.n())
D   = DiscreteGaussianDistributionIntegerSampler(stddev)
LWE.__init__(self, n=n, q=q, D=D, secret_dist='noise', m=m)
```

5.3.2 Proposition de choix sécurité pour très faible profondeur

6 Mise en place d'un bootstrapping

6.1 Un point sur la sécurité

JE N'AI PAS RETROUVÉ LE TERME UTILISÉ PAR MR. CASTAGNOS SUR LA SÉCURITÉ ICI. CIRCULAIRE ? CYCLIQUE ? UN TRUC DE CE GENRE.
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.
Il faut pouvoir s'assurer qu'un attaquant ne puisse pas en tirer d'information sur la première clé. Cependant, le système étant IND-CPA, cette propriété est toujours vérifié 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.

6.2 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.
N'étant pas sous la contrainte d'être particulièrement performant, nous avons écris des fonctions booléennes homomorphiques `NO`, `AND`, `OR` et `XOR` avec des portes **NAND**.
DETAILS NECESSAIRES (nb NAND, profondeur)
Cela nous permettra de l'exécuter homomorphiquement.
Pour cela, nous considèrerons la décomposition binaire du secret et considèreront avoir un chiffré par la seconde clé de chaque bit du premier secret.

6.3 Sommer des vecteurs avec une profondeur en NAND minimale

Nous voulons pouvoir sommer homomorphiquement des nombres à partir des chiffrés de leurs décompositions binaires traitées comme des listes de tailles lim. Pour cela, nous allons étudier deux algorithmes.

Le premier, que nous appelerons **basic_sum** est l'algorithme naïf de somme binaire, commençant par les bits de poids faible puis remontant vers les bits de poids plus élevés en conservant des retenues.
Soient $A = \sum\limits_{i=0}^{lim} a_i 2^i$ et $B = \sum\limits_{i=0}^{lim} b_i 2^i$ que l'ont veux sommer.
On nomme $s_0, s_1, ..., s_{lim}$ les bits de la somme et r_i la retenue après être passé sur les bits a_i et b_i.
On définit $r_{-1} = 0$.
On calcule alors juste :
$s_i = a_i \oplus b_i \oplus r_{i-1}$ et
$r_i = (a_i \wedge b_i) \vee (r_{i-1} \wedge (a_i \vee b_i))$.
Le problème de cette méthode est que l'on réutilise la r_{i-1} pour calculer r_i. Cela signifie entre autre que a_0 et b_0 sont utilisés tous les calculs.
Calculer s_i peut se faire en utilisant r_{i-1} que pour un $\oplus$, ce qui n'ajoute que 4 à la profondeur en **NAND** du calcul.
r_i peut être 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**.
$A + B$ se calcule donc avec une profondeur de $4lim + 2$ **NAND** en a_0 et b_0.

Le second, que nous appelerons **reduced_sum** est un algorithme assez utilisé qui à trois nombres écris en binaire associe deux listes de tailles lim dont la somme est égale à la somme des trois nombres donnés en entrée.
Soient $A = \sum\limits_{i=0}^{lim} a_i 2^i$, $B = \sum\limits_{i=0}^{lim} b_i 2^i$ et $C = \sum\limits_{i=0}^{lim} c_i 2^i$ que l'ont veux sommer.
On nomme $X = \sum\limits_{i=0}^{lim} x_i 2^i$ et $Y = \sum\limits_{i=0}^{lim} y_i 2^i$ les nombres obtenus.
On fixe $y_0 = 0$.
On calcule ensuite :
$x_i = a_i \oplus b_i \oplus c_i$ et
$y_i = \overline{(\overline{a_{i-1}} \wedge \overline{b_{i-1}}) \oplus (\overline{b_{i-1}} \wedge \overline{c_{i-1}}) \oplus (\overline{a_{i-1}} \wedge \overline{c_{i-1}})}$ pour $i \neq 0$.
On remarque que cette méthode, contrairement à **basic_sum**, n'effectue pas de récurence. La profondeur totale 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 $\oplus$ succesifs 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**.

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

7 Implémentation d'un FHE avec bootstrapping « jouet »

7.1 Présentation de notre arborescence

7.2 Une présentation générale : chiffrer et déchiffrer

7.3 Tester des choix de paramètres sur des fonctions sans bootstrapping

7.4 Tester des choix de paramètres sur des fonctions avec bootstrapping

8 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 » [2] 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 ;

8.1 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 [3] 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.

8.2 The Gate Bootstrapping API

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

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.

8.2.1 Le problème TFHE

8.2.2 Un exemple simple fourni par leur tutorial

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 :

2. Pas forcément pour nos machines et avec notre implémentation

3. Sur Linux debian 4.9.0-8-amd64, nous avons dû utiliser les backports debians pour avoir une version de cmake suffisament récente

— 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] 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.

[3] 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`.

[4] 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`.

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

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

[7] 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.

[8] 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.

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

[10] 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.

[11] Richard Lindner and Chris Peikert. Better key sizes (and attacks) for LWE-Based encryption. Cryptology ePrint Archive, Report 2010/592, 2010. `http://eprint.iacr.org/2010/592`.

[12] 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`.

[13] 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.

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

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

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