Implémentation d'un FHE

Lucas Roux et Eric Sageloli

17 février 2019

Table des matières

1 Introduction

Certains systèmes cryptographiques, comme RSA, ElGamal ou encore le cryptosystème de Pallier possèdent une propriété intéressante : faire le produit de deux chiffrés revient à chiffrer le produit de leurs clairs. Cette propriété 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 entités 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 [9] 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 » ainsi qu'une technique 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 FHE ont émergé, basant essentiellement leur sécurité sur le problème dit du Learning With Error (LWE) mis en avant en 2005 par Oded Regev dans l'article [18]. En font partie les cryptosystèmes dits de seconde génération apparus vers 2011. Il furent les premiers à permettre une implémentation « réaliste », et 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. Une troisième génération est ensuite apparue avec le cryptosystème GWS, publié par Gentry, Sahai et Waters en 2013 (voir [11]), se distinguant par une nouvelle approche permettant une définition du produit aussi simple que la somme.

Le but de notre rapport et des codes associés, présents dans le github [6], est d'étudier le cryptosystème GSW à la fois sous des aspects théoriques et pratiques. Pour cela, nous le présenterons, étudierons sa sécurité et les paramètres permettant de l'assurer, puis en présenterons une implémentation jouet [2] faite en `sagemath` que nous avons réalisée pour ce projet. Enfin, nous jetterons aussi un œil à des API open-source permettant d'utiliser des FHE. Afin d'être exhaustif, nous avons essayé de donner des définitions pour chacune des notions évoquées dans notre travail, notamment concernant les gaussienne discrètes et les diverses définitions associées aux FHE. Notez toutefois que celles-ci sont généralement tirés, ou inspirées d'autres expositions plus détaillées, qui seront évidemment précisés et que nous conseillons au lecteur de se référer. En ce sens, ce rapport permet aussi de faire une petite synthèse bibliographique.

1. ou bien seulement les sommer, comme cela peut être le cas, par exemple avec des variantes de ElGamal

2. sans regards sur les très nombreuses optimisations aujourd'hui faites dans les vraies implémentations de GSW et ses dérivés

2 Notions préliminaires

2.1 Différentes normes

Pour $x \in \mathbb{N}$, on note $|x|_{\text{bin}} := \lfloor \log(x) \rfloor + 1$. Pour $q > 0$, la valeur absolue d'un élément $x \in \mathbb{Z}_q$ sera par définition la valeur absolue dans $\mathbb{Z}$ de son représentant 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 et la norme $\|x\|_1$ la somme des valeurs absolues de ses coordonnées.

2.2 LWE et DLWE

Nous présentons ici les définitions des problèmes Learning With Error (LWE) et Decisional Learning With Error (DWLE).

Définition 1. *Decisional Learning with Errors (DLWE)*

Pour un paramètre de sécurité λ, soit $n = n(\lambda)$, $q = q(\lambda)$ des entiers et $\chi = \chi(\lambda)$ une distribution sur $\mathbb{Z}_q$, tous générés en temps 1^λ.

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 $(\vec{a}_i, b_i) \in \mathbb{Z}_q^{n+1}$ de façon uniforme. La deuxième utilise un secret $\vec{s} \in \mathbb{Z}_q^n$ tiré uniformément et prend pour valeurs des vecteurs $(\vec{a}_i, b_i)$ où :

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

e_i étant échantillonné par χ.

Notons que nécessairement, $n = \mathcal{O}(P(\lambda))$ et $\log(q) = \mathcal{O}(P(\lambda))$ pour un polynome P pour que les données soient générés en temps 1^λ.

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 $\mathbb{Z}$, tous générés en temps 1^λ. On tire $\vec{s} \in \mathbb{Z}_q^n$ uniformément et on considère la distribution sur $\mathbb{Z}_q^{n+1}$ qui prend pour valeurs des vecteurs $(\vec{a}_i, b_i)$ où :

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

e_i étant échantillonné 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 ». Il est assez évident que savoir résoudre LWE permet de résoudre DLWE. Pour l'autre sens, le Lemme 4.2 de [18] montre comment réduire à DLWE à LWE sous notamment les hypothèses que q soit premier et $q = \mathcal{O}(\text{poly}(n))$. Le théorème 3.1 de [16] 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 $q = 2^k$.

Regardons plus précisément 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 (resp. qui refuse avec une probabilité négligeable) les distributions $A_{s,\xi}$ et rejete avec une probabilité exponentiellement proche de 1 (resp. accepte avec une probabilité négligeable) 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 (resp. ne trouve pas s avec une probabilité négligeable).

Démonstration. Nous indiquons ici la démonstration faite dans [18], avec les bornes exponentielles. La démonstration est la même dans l'autre cadre (celui précisé en « resp ») en utilisant que le produit d'une fonction négligeable par un polynôme reste négligeable.

L'automate W' va trouver s coordonnée 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 pour tout $1 \leqslant i \leqslant n + 1$.

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

De plus, la probabilité de se tromper vaut n fois un terme 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 [12], 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 (shortest 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 $\vec{v} = (1 \ -\vec{s})$ 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.

Considérons donc le problème LWE : 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 crée de petites valeurs, on voit qu'on a alors trouvé un « petit » vecteur du réseau engendré par les colonnes de

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

Choix de paramètres pour rendre DLWE difficile :

Intéressons nous maintenant à la façon de choisir des paramètres pour que le problème $DLWE$ soit difficile. Il s'agit d'une question épineuse, liée à l'efficacité des différentes façons d'attaquer $DLWE$, qui est encore aujourd'hui un sujet de recherche important.

Nous utiliserons pour notre part les remarques faites dans [13], qui datent de 2017 et doivent donc être prises avec précautions.

Il impose déjà que la distribution χ soit « concentrée sur de petites valeurs » . Plus précisément, il pose l'hypothèse suivante :

Hypothèse 1. *Hypothèse sur la probabilité χ*
il doit exister $\alpha = \alpha(n) \ll 1$ tel que la fonction :

$$n \mapsto \mathbb{P}[x \leftarrow \chi : |x| > \alpha q]$$

soit négligeable.

Notons qu'il faut donc $\alpha q > 1$ pour que χ puisse générer autre chose que des zéros.

Rappelons 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 :

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

Ainsi, cette hypothèse peut s'exprimer ainsi : la famille de distribution $\chi_n = \chi$ est B-bornée pour $B = \alpha q$.

On considère alors deux hypothèses de difficulté concernant DLWE.

Hypothèse 2. *hypothèse pour le leveled GSW*
Il existe un $0 < \epsilon < 1$ tel que le problème DLWE soit difficile pour

$$q \approx 2^{n^\epsilon} \quad \alpha q = n \quad m \text{ polynomial en } n$$

Hypothèse 3. *hypothèse pour le GSW avec bootstrapping*
Le problème DLWE est difficile pour

$$q \approx 2^{polylog(n)} \quad \alpha q = n \quad m \text{ polynomial en } n$$

2.3 Réseaux euclidiens

Nous rappelons ici quelques résultats sur les réseaux euclidiens, tels qu'énoncés dans [17]. 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 :*

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

Proposition 2. *Soit $L \subset \mathbb{R}^n$ un réseau euclidien. Si B est base de L, alors 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,*

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

2.4 La gaussienne discrète

Très souvent, la distribution χ choisie pour avoir des paramètres sécurités pour les problèmes LWE/DLWE est une gaussienne discrète. Nous nous proposons ici d'en indiquer la définition ainsi que certaines propriétés. Notamment, la propriété 6 montre qu'il suffit de prendre $\chi = D_s^q$ avec un $s > 1.5$ constant pour qu'avec les valeurs de α spécifiées dans les hypothèses 2 et 3, l'hypothèse 1 est satisfaite.

Nous reprenons ici les notations de [10].

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

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

Puis, pour un réseau $\Lambda \in \mathbb{R}^n$, nous définissons la gaussienne discrète $D_{\Lambda,s,c}$ comme la distribution de support Λ et 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_s^q modulo un entier $q > 0$ comme loi de probabilité qui a $x \in \mathbb{Z}_q$ attribue

$$D_{\mathbb{Z},s,0}(\pi^{-1}(x))$$

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

Nous introduisons maintenant les notions et propositions permettant de montrer pourquoi on peut choisir des gaussiennes discrètes afin de satisfait l'hypothèse 1.

Définition 5. *Smoothing parameter*

Pour un réseau L de dimension n et un réel $\epsilon > 0$, le paramètre $\eta_\epsilon(L)$, dit smoothing parameter, est le plus petit réel $s > 0$ tel que

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

L'article [21] indique une borne intéressante sur le smoothing parameter :

Proposition 4. *Pour $\epsilon < 0.086435$, on a la borne :*

$$\eta_\epsilon(\mathbb{Z}) \leqslant \sqrt{\frac{\ln(\epsilon/44 + 2/\epsilon)}{\pi}}$$

Notamment, pour $\epsilon = 0.08$, $\eta_\epsilon(\mathbb{Z}) \leqslant 1.5$.

Proposition 5. *Pour tout $\epsilon > 0$, $s \geqslant \eta_\epsilon(\mathbb{Z})$ et pour tout $t > 0$:*

$$\mathbb{P}(x \leftarrow D_{\mathbb{Z},s,c} \ : \ |x - c| \geqslant t \cdot s) \leqslant 2e^{-\pi t^2} \cdot \frac{1+\epsilon}{1-\epsilon}$$

Proposition 6. *Pour $s \geqslant 1.5$ et $\alpha = \alpha(\lambda)$, si il existe une constante τ telle que $\alpha \geqslant \tau n/q$, alors :*

$$\lambda \to \ \mathbb{P}(x \leftarrow D_s^q \ : \ |x| \geqslant q\alpha)$$

est négligeable et D_s^q satisfait donc l'hypothèse 1.

Démonstration. On veut appliquer la proposition 5. Notons d'abord que la borne précédemment donnée permet de dire que $s > \eta_\epsilon(\mathbb{Z})$.

De plus, en appliquant la proposition précédente avec $t = \frac{q\alpha}{s}$, on voit que

$$\mathbb{P}(x \leftarrow D_s^q \ : \ |x| \geqslant q\alpha) = \mathbb{P}\left(x \leftarrow D_s^q \ : \ |x| \geqslant s\left(\frac{q\alpha}{s}\right)\right)$$

est négligeable si il existe un $u > 0$ tel que

$$q\alpha/s \geq un.$$

Or, $u = s\tau$ convient. □

3 FHE, SFHE et bootstrapping

Nous indiquons ici les définitions de base d'un FHE tirées de l'exposition faite dans [13] que nous conseillons de lire pour avoir plus de détails.

Notons que le cryptosystème que nous allons étudier a pour ensemble de clairs $\mathbb{Z}_q$, mais que ces définitions seront faites pour un cryptosystème ne permettant de ne chiffrer que 0 ou 1. Cela peut sembler réducteur, mais nous ne prouverons que le cryptosysteme GSW peut-être adapté en FHE et LHE [3] qu'en restreignant les clairs à l'ensemble $\{0,1\}$.

Définition 6. *Cryptosystème homomorphe*

Un cryptosystème homomorphe est constitué de 4 fonctions :

$$\boldsymbol{HE} = (\boldsymbol{Keygen}, \boldsymbol{Encrypt}, \boldsymbol{Decrypt}, \boldsymbol{Eval})$$

où l'espace des clairs est $\{0,1\}$ et l'espace des chiffrés est un ensemble C. Plus précisément :

- $(pk, sk) \leftarrow \boldsymbol{Keygen}(1^\lambda)$, *$\lambda$ étant le paramètre de sécurité, retourne une clé publique et une clé secrète.*
- $c \leftarrow \boldsymbol{Encrypt}_{pk}(\mu)$ *prend une clé publique pk, un clair μ et retourne un chiffré $c \in C$*
- $\mu \leftarrow \boldsymbol{Decrypt}_{sk}(c)$ *prend une clé secrète sk, un chiffré $c \in C$ et retourne un clair μ.*
- $\vec{c}_f \leftarrow \boldsymbol{Eval}_{pk}(\Pi, \vec{c})$ *prend une clé publique pk, un circuit Π, un vecteur de chiffrés $\vec{c}$ et retourne un vecteur de chiffrés $\vec{c}_f$ dont les composantes correspondent aux sorties du circuit Π.*

Définition 7. *Correction Soit $\boldsymbol{E} = (\boldsymbol{Keygen}, \boldsymbol{Encrypt}, \boldsymbol{Decrypt}, \boldsymbol{Eval})$ un cryptosystème homomorphe et $\mathcal{C} = \{\mathcal{C}_\tau\}_{\tau \in \mathbb{N}}$ une famille de circuits. On dira que $\boldsymbol{E}$ est correct pour $\mathcal{C}$ si il déchiffre avec succès les messages qui viennent d'être chiffrés ainsi que ceux qui ont été évalués sur un circuit de $\mathcal{C}$. Voir [13] pour les formules détaillées.*

Définition 8. *Compacité*

Un cryptosystème homomorphe $\boldsymbol{E} = (\boldsymbol{Keygen}, \boldsymbol{Encrypt}, \boldsymbol{Decrypt}, \boldsymbol{Eval})$ est compact si il existe un polynome B tel que pour tout $\lambda, \tau \in \mathbb{N}$, tout circuit Π à t entrées et une seule sortie, et un chiffré $b = (b_1, \cdots, b_t) \in \{0,1\}^t$, on a :

$$\mathbb{P}\left[|c'| \leqslant B(\lambda) : (sk, pk) \leftarrow \boldsymbol{Keygen}(1^\lambda, 1^\tau), \vec{c} \leftarrow \boldsymbol{Encrypt}_{pk}(\vec{b}), \vec{c} \leftarrow \boldsymbol{Eval}_{pk}(\Pi, \vec{c})\right] = 1.$$

Définition 9. *FHE, LHE et SWHE*

Soit $\boldsymbol{E}$ un cryptosystème homomorphe.

- *On dit que c'est un fully homomorphic encryption (abrégé en FHE) si il est correct pour une famille $\mathcal{C}$ telle que $\mathcal{C}_1$ contient déjà tous les circuits booléens.*
- *On dit que c'est un leveled homomorphic encryption (abrégé en LHE) si il est correct pour une famille $\mathcal{C}$ telle que pour tout τ, $\mathcal{C}_\tau$ contient les circuits booléens de profondeur plus petite ou égale à τ. Plus généralement, l'idée est de pouvoir prévoir, pour tout L, les paramètres à utiliser afin de pouvoir évaluer avec succès les circuits d'une profondeur L*
- *On dit, informellement, qu'il s'agit d'un somewhat homomorphic encryption (abrégé en SWHE) si il est correct pour une famille $\mathcal{C}$ de circuits tels que la complexité des circuits de $\mathcal{C}_\tau$ grandis avec τ.*

Nous allons maintenant définir le bootstrapping en introduisant pour cela quelques notations.

- Pour $\lambda, \tau \in \mathbb{N}$, on note $\mathcal{CT}_{\mathbf{E}}(\lambda, \tau)$ l'union des chiffrés de messages ainsi que des évaluations de chiffrés de messages par des circuits de $\mathcal{C}_\tau$, les formules détaillées étant donnés dans [13] ;

3. voir les défintions suivantes

— On considère pour tout chiffré $c \in \mathcal{CT}_{\mathbf{E}}(\lambda, \tau)$ le circuit $D_c(\text{bsk}) := \mathbf{Decrypt}_{\text{sk}}(c)$ ayant pour entrée la décomposition binaire de sk et appliquant l'algorithme de déchiffrement de c avec la clé sk (notons bien que c n'est pas une entrée du circuit, voir la remarque suivante). On crée aussi le circuit suivant, avec pour unique entrée sk, appelé circuit de déchiffrement augmenté de c_1 et c_2

$$D^*_{c_1,c_2}(sk) := \text{NAND}(D_{c_1}(\text{sk}), D_{c_2}(\text{sk})).$$

Remarque 1. *Soyons un peu plus précis sur ce que signifie de dire que $D_c(bsk)$ n'a pour entrée que sk. Cela veut dire que quand on applique **Eval** au circuit D_c avec un vecteur de chiffrés $\vec{c_{bsk}}$, seules changent dans D_c les portes NAND qui deviennent des NAND homomorphes.*

*Il n'est donc à priori pas possible avec cette définition d'utiliser des coefficients de c dans le circuits D_c et d'espérer qu'ils soient alors chiffrés quand on leur applique **Eval**.*

Définition 10. *Bootstrappable encryption*

Un cryptosystème homomorphe $\boldsymbol{E} = (\boldsymbol{Keygen}, \boldsymbol{Encrypt}, \boldsymbol{Decrypt}, \boldsymbol{Eval})$ est bootstrappable si il peut évaluer homomorphiquement tous les circuits de chiffrement augmentés. Autrement dit, si il existe une fonction τ bornée par un polynôme tel que pour tous $\lambda \in \mathbb{N}$ et tous chiffrés $c_1, c_2 \in \mathcal{CT}_{\boldsymbol{E}}(\lambda, \tau(\lambda))$, on a $D^*_{c_1,c_2} \in \mathcal{C}_{\tau(\lambda)}$.

Définition 11. *weak circular security*

Soit $\boldsymbol{E} = (\boldsymbol{Keygen}, \boldsymbol{Encrypt}, \boldsymbol{Decrypt})$ un cryptosystème et $\mathcal{A}$ un algorithme polynomial probabiliste. On considère l'expérience suivante,

— $1^\tau \leftarrow \mathcal{A}(1^\lambda)$.

— $(sk,\ pk) \leftarrow \boldsymbol{Keygen}(1^\lambda, 1^\tau)$.

— *On chiffre chaque bit de la clé secrète et on obtient un vecteur de chiffrés $\vec{c^*} \leftarrow \boldsymbol{Encrypt}_{pk}s_k$.*

— *Un bit $b \in \{0,1\}$ est tiré uniformément.*

— *On crée $c \leftarrow \boldsymbol{Encrypt}_{pk}(b)$.*

— $b^* \leftarrow \mathcal{A}(pk, \vec{c^*}, c)$.

— *L'expérience est réussie si $b = b^*$ et ratée sinon.*

Notant $p(\lambda, \mathcal{A})$ la probabilité de succès de cette expérience, on dit que $\mathcal{A}$ a un avantage de sécurité circulaire sur $\boldsymbol{E}$ si :

$$\lambda \mapsto |p(\lambda, \mathcal{A}) - \frac{1}{2}|$$

est négligeable.

Théorème 1. *Tout cryptosysteme homomorphe bootstrappable avec sécurité circulaire peut être transformé en un cryptosystème fully homomorphe compact.*

4 Présentation du cryptosystème

4.1 L'idée générale

L'idée de ce cryptosystème 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, C_2 chiffrés respectifs de m_1 et m_2, on a :

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

Toutefois, un tel système n'est pas sécurisé car C n'a qu'un nombre fini de valeurs propres, et il est 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 DLWE.

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 $C_1 \cdot \vec{e}_2$ peut être très grand même pour un petit $\vec{e}_2$. Nous allons voir à la section suivante comment définir une fonction **Flatten** transformant une matrice à coefficients dans $\mathbb{Z}_q$ en une matrice à coefficients dans $\{0, 1\}$ telle que, pour un bon choix de $\vec{v}$, on ait :

$$\textbf{Flatten}(C) \cdot \vec{v} = C \cdot \vec{v}.$$

Cela permettra alors d'éviter une explosion de l'erreur lors de l'application d'opérations homomorphes.

4.2 Fonctions auxiliaires utiles au cryptosystème

BitDecomp :

Entrée : Un vecteur $\vec{a} = (a_1, ..., a_k) \in \mathbb{Z}_q^k$.
Sortie : 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. On retourne la concaténation de ces représentations binaires sous la forme d'un vecteur.

BitDecomp^{-1} :

Entrée : Un vecteur $\vec{a} = (a_{1,0}, ..., a_{1,l-1}, a_{2,0}, ..., a_{k,l-1})$.
Sortie : $(\sum_{i=0}^{l-1} 2^i a_{1,i}, ..., \sum_{i=0}^{l-1} 2^i a_{k,i})$.
Remarque : C'est un inverse à gauche de **BitDecomp** sur toutes les entrées et un inverse à droite de **BitDecomp** sur les vecteurs uniquement constitués de 0 et de 1. Il est toutefois aussi défini sur les vecteurs à valeur dans $\mathbb{Z}_q$.

PowersOf2 :

Entrée : Un vecteur $\vec{a} = (a_1, ..., a_k) \in \mathbb{Z}_q^k$.
Sortie : $(a_1, 2a_1, 2^2a_1, ..., 2^{l-1}a_1, a_2, ..., 2^{l-1}a_k)$.

Flatten :

Entrée : Un vecteur $\vec{a} = (a_{1,0}, ..., a_{1,l-1}, a_{2,0}, ..., a_{k,l-1})$.
Sortie : Un vecteur $\vec{b} = (b_{1,0}, ..., b_{1,l-1}, b_{2,0}, ..., b_{k,l-1})$ à valeurs dans $\{0,1\}$ ayant même produit scalaire que $\vec{a}$ avec les vecteurs de forme $\mathbf{PowersOf2}(\vec{u})$ pour tout vecteur $\vec{u}$ (voir la proposition 8).
Algorithme : $\mathbf{Flatten} = \mathbf{BitDecomp}^{-1} \circ \mathbf{BitDecomp}$

Voyons maintenant quelques propriétés de ces applications, notamment concernant **Flatten**, qui joue un rôle clé dans notre cryptosystème en réduisant la valeur absolue des coefficients des chiffrés ainsi que celle du bruit produit après des opérations homomorphes.

Proposition 7. *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.

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

□

Proposition 8. *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.

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

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

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

□

4.3 Définition du cryptosystème

Nous allons ici définir le cryptosystème GSW en utilisant conjointement l'article de base [11] ainsi que d'intéressantes remarques faites par Shai Halevi dans [13].

On rappelle que GSW utilise les paramètres suivants :

— le paramètre de dimension n ;
— le modulus q ;
— le paramètre de taille d'échantillon m ;
— la distribution χ.

De plus, on note $l = |q|_{\text{bin}} = \lfloor \log q \rfloor + 1$ et $N = (n+1)\, l$.

Setup :

Entrée : 1^λ et 1^L où λ est le paramètre de sécurité et L le paramètre de profondeur.
Sortie : Les paramètres n, q, χ, m soumis à la contrainte de profondeur imposée par L, ainsi que des contraintes de sécurités imposées par λ.
Le paramètre L indique que les paramètres créés doivent permettre d'évaluer homomorphiquement un circuit de NAND de profondeur L tout en pouvant déchiffrer correctement. Nous montrerons qu'il est alors nécessaire de satisfaire la condition dite « de longueur »

$$q > 8nm(1+N)^L. \tag{1}$$

Algorithme : Deux sous-sections sont consacrés au choix des paramètres :

— la sous-section 5.2 page 19 dans le cadre d'un LHE ;
— la sous-section 6.5 page 30 dans le cadre d'un FHE avec bootstrapping.

KeyGen :

Entrée : Les paramètres donnés par **Setup**.
Sortie : La clé secrète $\vec{s} \in \mathbb{Z}_q^{n+1}$ ainsi que la clé publique $A \in \mathbb{Z}_q^{m \times n+1}$ vérifiant la contrainte $\|A \cdot \vec{s}\|_\infty \leqslant n$.
Algorithme :
clé secrète : On génère un vecteur $t \in \mathbb{Z}_q^n$ avec χ et on définit la clé secrète comme **PowersOf2**$(\vec{s})$ pour $\vec{s} = (1, -t_1, ..., -t_n)$.
clé publique : On génère uniformément $B \in \mathbb{Z}_q^{n \times m}$ 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 $A = \vec{b} \,||\, B$, concaténation de $\vec{b}$ considéré comme un vecteur colonne et de B.
Si la contrainte $\|A \cdot \vec{s}\|_\infty \leqslant n$ n'est pas vérifiée, on recrée un jeu de clés.
Taille : Comme on l'a dit, $\vec{s} \in \mathbb{Z}_q^{n+1}$. On peut de plus représenter tout élément de $\mathbb{Z}_q$ en $l = |q|_{\text{bin}}$ bits. $\vec{s}$ peut donc se représenter en $l * (n+1) = N$ bits.
D'autre part, $A \in \mathbb{Z}_q^{m \times n+1}$ donc A se représente en $l * (n+1) * m = N * m$ bits.
Remarque : Générer A et $\vec{s}$ jusqu'à avoir $\|A \cdot \vec{s}\|_\infty \leqslant n$ peut poser des problèmes de sécurité, car ce n'est alors plus tout à fait χ qui est utilisée. Notons toutefois que $A \cdot \vec{v} = \vec{e}$, et que si $\alpha = n/q$ comme le préconisent les hypothèses sur les paramètres que nous utiliserons,

$$\mathbb{P}\left(\vec{e} \leftarrow \chi : \|\vec{e}\|_\infty > n\right)$$

est négligeable.

Encrypt :

Entrée : Les paramètres du système, une clé publique et un message $\mu \in \mathbb{Z}_q$.
Sortie : Un chiffré $C \in \mathbb{Z}_q^{N \times N}$ de μ.
Algorithme : On génère uniformément une matrice $R \in \{0,1\}^{N \times m}$ puis on pose :

$$C = \textbf{Flatten}(\mu \times I_N + \textbf{BitDecomp}(R \times A)).$$

Taille : $C \in \mathbb{Z}_q^{N\times N}$ se représente en $l * N^2$ bits.

Decrypt :

Entrée : Les paramètres du système, une clé secrète et un chiffré d'un message $\mu \in \{0,1\}$.
Sortie : Le clair du chiffré si l'erreur de ce dernier n'est pas trop élevée (voir la proposition 9)
Algorithme : On rappelle que les l premiers coefficients de $\vec{v}$ sont

$$1, 2, \cdots, 2^{l-1}.$$

On trouve alors $i \in [\![0, l-1]\!]$ tel que [4] $|\vec{v}_i| > q/4$. Notant C_i la ième ligne de C, on calcule ensuite

$$x_i = \langle C_i, \vec{v} \rangle$$

et on retourne $\lfloor x_i/v_i \rceil$.

Définition 12. *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 9. ***Decrypt*** *décrypte avec succès les chiffrés dont l'erreur $\vec{e}$ satisfait $\|\vec{e}\|_\infty < q/8$.*

Démonstration. Dans ce cas, on a $x_i = \mu * v_i + e$ avec $|e| \leqslant \|\vec{e}\|_\infty$. Comme $|v_i| > \frac{q}{4}$, on a $|\frac{e}{v_i}| < 1/2$. On a donc $\lfloor \frac{x_i}{v_i} \rceil = \mu$. □

4.4 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 afin de pouvoir déchiffrer des chiffrés de n'importe quel élément de $\mathbb{Z}_q$. Notons cependant que ces algorithmes demandent certainement une profondeur plus grande de NAND pour être implémentés. Il serait notamment très improbable que le dernier algorithme que nous allons présenter puisse s'exécuter avec une profondeur assez petite pour permettre un bootstrapping.

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 coordonnées, 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 que l'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.

4. n'oublions pas qu'ici nous considérons la valeur absolue sur $\mathbb{Z}_q$

4.4.1 mp_decrypt : q est une puissance de 2

L'algorithme, présenté dans [11], 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 on trouve de proche en proche les autres bits de l'écriture binaire de m.

4.4.2 mp_all_q_decrypt : q est quelconque

Le travail effectué ici est notamment tiré de la section 4 de [17].
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 de vecteur le plus proche **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 [17], ce qui peut être intéressant, car cela est lié au domaine fondamental utilisé par l'algorithme et donc à sa réussite.

4.5 Opérations homomorphes

Nous allons ici présenter diverses opérations homomorphes pour l'algorithme GSW. Toutefois, seule l'opération NAND sera permise pour le LHE et le FHE avec bootstrapping que nous considèrerons car d'une part, seuls les chiffrés de 0 et de 1 seront autorisé et d'autre part, nos choix de paramètres ne sont fait qu'en supposant des circuits contenant uniquement des NAND [5].

On rappelle que $\vec{v}$ est de la forme $\mathbf{PowersOf2}(\vec{s})$ et donc que

$$\mathbf{Flatten}(C) \cdot \vec{v} = C \times \vec{v} \quad \text{pour tout } C \in \mathbb{Z}^{N \times N}.$$

h_MultConst :

5. Ce choix est cohérent, tout circuit booléen pouvant être construit uniquement à partir de portes NAND.

Entrée : Les paramètres du cryptosystème, un chiffré $C \in \mathbb{Z}_q^{N\times N}$ d'un message μ et une constante $\alpha \in \mathbb{Z}_q$.
Sortie : Un chiffré de $\alpha \cdot \mu$.
Algorithme : On calcule $M_\alpha = \mathbf{Flatten}(\alpha \times I_N)$ puis on retourne

$$\mathbf{Flatten}(M_\alpha \times C).$$

Cela fonctionne car :

$$\begin{aligned}\mathbf{h\_MultConst}(C,\alpha)\times\vec{v} &= M_\alpha \times C \times \vec{v}\\ &= M_\alpha \cdot (\mu * \vec{v} + \vec{e})\\ &= M_\alpha \times \mu * \vec{v} + M_\alpha \times \vec{e}\\ &= \alpha * \mu * \vec{v} + M_\alpha \times \vec{e}.\end{aligned}$$

Erreur : Le chiffré a une erreur $e_2 = M_\alpha \times \vec{e}$ majorée par

$$\|e_2\|_\infty \leqslant N\|e_1\|_\infty.$$

h_Add :

Entrée : Les paramètres du cryptosystème et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ des messages respectifs $\mu_1, \mu_2 \in \mathbb{Z}_q$.
Sortie : Un chiffré de $\mu_1 + \mu_2$.
Algorithme : On retourne

$$\mathbf{Flatten}(C_1 + C_2).$$

Cela fonctionne car :

$$\begin{aligned}\mathbf{h\_Add}(C_1,C_2)\times\vec{v} &= (C_1 + C_2)\times\vec{v}\\ &= (\mu_1 * \vec{v} + \vec{e_1}) + (\mu_2 * \vec{v} + \vec{e_2})\\ &= (\mu_1 + \mu_2) * \vec{v} + \vec{e_1} + \vec{e_2}.\end{aligned}$$

Erreur : Le chiffré a une erreur $e_3 = \vec{e_1} + \vec{e_2}$ majorée par

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

h_Mult :

Entrée : Les paramètres du système et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ respectifs des messages $\mu_1, \mu_2 \in \mathbb{Z}_q$.
Sortie : Un chiffré de $\mu_1 * \mu_2$.
Algorithme : On retourne

$$\mathbf{Flatten}(C_1 \times C_2).$$

Cela fonctionne car

$$\begin{aligned}\mathbf{h\_Mult}(C_1,C_2)\times\vec{v} &= (C_1 \times C_2)\times\vec{v}\\ &= C_1 \times (\mu_2 * \vec{v} + \vec{e_2})\\ &= \mu_2 * C_1 \times \vec{v} + C_1 \times \vec{e_2}\\ &= \mu_2 * (\mu_1 * \vec{v} + \vec{e_1}) + C_1 \times \vec{e_2}\\ &= (\mu_1 * \mu_2) * \vec{v} + \mu_2 * \vec{e_1} + C_1 \times \vec{e_2}.\end{aligned}$$

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

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

h_NAND :

Entrée : Les paramètres du système et deux chiffrés $C_1, C_2 \in \mathbb{Z}_q^{N\times N}$ respectifs des messages $\mu_1, \mu_2 \in \{0,1\}$.
Sortie : Un chiffré de $\overline{(\mu_1 \wedge \mu_2)}$.
Algorithme : On retourne

$$\textbf{Flatten}(I_N - C_1 \cdot C_2).$$

. Cela fonctionne car on a utilisé le fait que $\overline{(\mu_1 \wedge \mu_2)} = 1 - \mu_1 * \mu_2$, ainsi :

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

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

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

4.6 Correction du cryptosystème

Ici, nous considérons comme seules opérations homomorphes les portes **h_NAND** et comme algorithme de déchiffrement **Decrypt**.

Proposition 10. *Si la condition de longueur*

$$q > 8nm(1+N)^L$$

*est respectée, on peut appliquer L portes **h_NAND** à un chiffré de 0 ou de 1 et le déchiffrer correctement.*

Démonstration. Pour $\mu \in \{0,1\}$ et $C = \textbf{Encrypt}(\mu)$, on a :

$$C \cdot \vec{v} \ = \mu\vec{v} + \langle \text{BitDecomp}(R \cdot A), \vec{v}\rangle.$$

On peut donc minorer ainsi son erreur :

$$\begin{aligned}\|\langle \text{BitDecomp}(R \cdot A), \vec{v}\rangle\|_\infty &= \|\mu\vec{v} + \langle R \cdot A, \vec{s}\rangle\|_\infty \\ &= \|\mu\vec{v} + \langle R\cdot, A \cdot \vec{s}\rangle\|_\infty \\ &\leqslant mn \quad \text{car } R \text{ est à valeurs dans } \{0,1\} \text{ et } \|A \cdot \vec{s}\|_\infty \leqslant n\end{aligned}$$

Utilisant la majoration de l'erreur indiquée dans la définition de **h_NAND**, on voit qu'après l'application de i portes, le bruit e_i du chiffré c_i satisfait :

$$\|e_i\|_\infty \leqslant (N+1)^i mn.$$

Il suffit donc de montrer qu'on peut correctement déchiffrer c_L, qui correspond au cas avec la plus grande erreur. Par la proposition 9, on voit qu'il faut alors avoir :

$$\|e_L\|_\infty < q/8,$$

ce qui est vrai si

$$(N+1)^L < q/8.$$

On retrouve donc la condition de longueur. □

5 Sécurité IND-CPA et paramètres pour un leveled GSW

5.1 Sécurité du cryptosystème

Définition 13. *Distance statistique*

Soit X et Y deux variables aléatoires supportées par un ensemble $\mathcal{V}$ et à valeurs dans un groupe abélien G. On définit la distance statistique entre X et Y, notée $SD(X,Y)$, comme étant

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

Définition 14. *Familles statistiquement indistinguables, calculatoirement indistinguables*

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.*
- *calculatoiremt indistinguables il n'existe pas d'automate polynomial probabiliste $\mathcal{A}$ pouvant distinguer X_i et Y_i avec un avantage non négligeable.*

Proposition 11. *Si deux familles de distributions sont statistiquement indistinguables, elles sont calculatoirement indistinguables.*

Proposition 12. *Soit $(X_i)_{1\leqslant i\leqslant n}$ et $(Y_i)_{1\leqslant i\leqslant n}$ deux n-uplets de distributions indépendantes.*

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

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

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

□

Le lemme suivant correspond au Claim 5.2 présent dans [18].

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. On a alors :*

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

Notamment,

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

Démonstration. Remarquons que :

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

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 :

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

Ce qui implique que :

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

□

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 :*

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

Notamment,

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

Démonstration. Découle directement de la proposition précédente ainsi que de la proposition 12. □

Proposition 13. *Supposons avoir pris des paramètres (n, q, χ, m) tels que l'hypothèse $LWE_{n,q,\chi}$ soit vraie et soit $\tau > 0$.*

Si $m > (1+\tau)(n+1)\log(q)$, la distribution jointe (A, RA), où $A \in \mathbb{Z}_q^{m \times (n+1)}$ est une clé publique et $R \in \mathbb{Z}_q^{N \times m}$ est créée en tirant les coefficients uniformément dans $\{0, 1\}$, 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 déjà 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 montrer 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 existe une constante $\lambda > 0$ telle que :

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

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

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

où f est négligeable en n.

Pour $(x, y) \in \mathbb{Z}_q^{m\times(n+1)} \times \mathbb{Z}_q^{m\times(n+1)}$, on a alors :

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

Ainsi, les deux familles de distributions sont statistiquement indistinguables, et donc calculatoirement indistinguables. □

Théorème 2. *Sécurité IND-CPA*
Sous les hypothèses de la proposition précédente, le cryptosystème est IND-CPA.

Démonstration. Comme un automate polynomial probabiliste ne peut pas distinguer $A_{s,\chi}$ de la distribution uniforme, on peut supposer que les coefficients de la clé publique A ont été tirés uniformément dans $\{0, 1\}$.

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 RA d'une matrice uniforme. On peut donc supposer que RA est uniforme, et donc que le chiffrement est 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. □

Nous allons maintenant nous intéresser au choix de paramètres de notre cryptosystème. Ils détermineront les degrés de sécurités et la profondeur des circuits calculables.

Sur ce point, deux approches sont possibles : une étude sur la sécurité asymptotique d'une famille de paramètres et une étude plus concrète, se demandant quelle est la sécurité d'un seul choix de paramètres. Nous pratiquerons ici les deux approches.

Posons ici notre contexte de travail.

Nous nous intéresserons ici uniquement à l'algorithme de déchiffrement **Decrypt** et ne considèrerons donc que des chiffrés de 0 ou 1. Comme tout circuit booléen peut être construit uniquement avec des NAND, ce sera la seule opération que nous considèrerons. Notre façon de voir si un circuit peut-être évalué homomorphiquement puis déchiffré avec succès sera alors de considérer sa profondeur en NAND.

5.2 Choix asymptotique de paramètres pour un leveled GSW

Nous allons ici supposer que l'hypothèse 2 est vraie[6] ; autrement dit, que le problème DLWE est difficile avec les paramètres suivants :

$$q \approx 2^{n^\epsilon}, \quad \alpha q = n, \quad m \text{ polynomial en } n.$$

6. Sachant qu'elle est faite dans un article datant de 2017, il est possible qu'on sache aujourd'hui qu'elle est fausse. Il faudrait faire des recherches supplémentaires pour savoir ce qu'il en est

De plus, le théorème 2 nécessite d'avoir

$$m > (1+\tau)(n+1)\log(q)$$

pour un $\tau > 0$ pour que le cryptosystème soit IND-CPA. Nous prenons alors :

$$m = 2(n+1)(\lfloor \log(q) \rfloor + 1) = 2(n+1)|q|_{\text{bin}} = 2N.$$

Enfin, nous devons aussi respecter la condition de longueur pour pouvoir appliquer une profondeur de L **h_NAND** à notre chiffré :

$$q > 8nm(1+N)^L.$$

En utilisant les inégalités $m < 2(N+1)$ et $n < (N+1)$ dans cette dernière équation, on voit alors qu'il nous suffit d'avoir :

$$q > 16(1+N)^{L+2}.$$

Ceci nous amène à un premier jeu de contraintes :

$$\begin{cases} \alpha = n \cdot 2^{-n^\epsilon} \\ q = \lceil 2^{n^\epsilon} \rceil \\ m = 2(n+1)|q|_{\text{bin}} \\ n^\epsilon > 4 + (L+2)\log(1+N) \end{cases}$$

Nous allons encore simplifier la dernière contrainte en utilisant

$$\begin{aligned} (L+2)\log(1+N) &\leqslant (L+2)(2+\log(N)) \\ &\leqslant (L+2)(2+\log(n+1)+\log(|q|_{\text{bin}})) \\ &\leqslant (L+2)(4+\log(n)+\log(\log(q)) \\ &\leqslant (L+2)(5+\log(n)+\log(n^\epsilon) \\ &\leqslant (L+2)(5+(1+\epsilon)\log(n)) \\ &\leqslant 5(L+2)+(L+2)(1+\epsilon)\log(n)) \\ &\leqslant 3L\log(n) \quad \text{pour } n \text{ assez grand} \end{aligned}$$

ce qui au final, nous donne :

$$\begin{cases} \alpha = n \cdot 2^{-n^\epsilon} = \\ q = \lceil 2^{n^\epsilon} \rceil \\ m = 2(n+1)|q|_{\text{bin}} \\ n^\epsilon > 3L\log(n) \end{cases}$$

Il nous reste donc à trouver une valeur dépendante du paramètre de sécurité λ pour n. En posant $n = \rho^{1/\epsilon}$, on voit que la dernière contrainte devient :

$$\frac{\rho}{\log(\rho)} > \frac{3L}{\epsilon}$$

et en prenant $\rho = \text{cst}\, a \log(a)$, cela devient :

$$\log(a) > \frac{3L}{\epsilon\ \text{cst}\ a}\left(\log(a) + \log(\log(a)) + \log(cst)\right)$$

qui est vérifié pour $a = L$ et $cst = 6/\epsilon$, car alors on obtient :

$$\log(L) > \frac{1}{2}\left(\log(L) + \log(\log(L)) + 1 + \log(3) - \log(\epsilon)\right)$$

$$\Leftrightarrow \log(L) - \log(\log(L)) - 1 - \log(3) > \log(\frac{1}{\epsilon})$$

ce qui est vrai pour L assez grand. On obtient donc

$$n = \rho^{1/\epsilon} = \left(\frac{6 * L * \log(L)}{\epsilon}\right)^{1/\epsilon}.$$

On en déduit le théorème suivant :

Théorème 3. *leveled GSW*

Pour L et λ assez grands et sous l'hypothèse sur DWLE (page 6), les paramètres suivants permettent de faire une profondeur de ***h_NAND*** *de L et rendent le cryptosystème GSW IND-CPA :*

$$\begin{cases} n = \max(\lambda, \lceil 6/\epsilon \log(L) \log(\log(L)) \rceil) \\ \alpha = n \cdot 2^{-n^\epsilon} \\ q = \lceil 2^{n^\epsilon} \rceil \\ m = 2(n+1)|q|_{bin} \end{cases}$$

	taille du secret	taille de la clé	taille d'un chiffré
$L = 10,\ \epsilon = 0.1$	131 octets	264 Ko	397 Ko
$L = 10,\ \epsilon = 0.5$	194 octets	581 Ko	3 Mo
$L = 100,\ \epsilon = 0.1$	410 octets	3 Mo	4 Mo
$L = 100,\ \epsilon = 0.5$	411 octets	3 Mo	4 Mo
$L = 1000,\ \epsilon = 0.1$	745 octets	8 Mo	13 Mo
$L = 1000,\ \epsilon = 0.5$	1 Ko	15 Mo	151 Mo

FIGURE 1 – Taille des données suivant les paramètres présentés précédemment avec $\lambda = 128$

Remarquons toutefois qu'il s'agit uniquement de paramètres « théoriques » étant donné que la sécurité n'est assurée qu'asymptotiquement. Voyons maintenant si nous pouvons trouver des paramètres concrets permettant une utilisation raisonnable sur un ordinateur.

5.3 Choix de paramètres concrets pour un leveled GSW

5.3.1 Présentation de lwe_estimator

Initialement utilisé dans l'article [1], `lwe_estimator` (disponible à l'adresse [7]) est un module de sagemath principalement maintenu par Martin Albrecht et destiné à estimer la résistance de paramètres précis face à diverses attaques connues pour le problème de learning with error.

Nous avons pensé qu'il pouvait être intéressant de l'utiliser afin de voir si nous pouvions trouver des paramètres offrants une sécurité concrète, et non uniquement une famille de paramètres offrant une sécurité asymptotique. Notons toutefois que la sécurité de notre cryptosystème n'est pas basée sur LWE mais DLWE, ce qui biaise dès le départ notre approche. De plus, la démonstration IND-CPA est elle aussi asymptotique et nous n'avons pas vu s'il est possible de réduire une attaque du cryptosystème sur un seul jeu de paramètres à une attaque de DLWE.

Ces avertissements faits, regardons plus en détail comment fonctionne `lwe_estimator`.

estimate_lwe :

Pour estimer la résistance de paramètres choisis sur un panel d'attaques, on utilise la fonction `estimate_lwe` dont le prototype est :

```
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 prend en arguments les paramètres suivants qui lui permettent de créer une instance LWE :

```
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,  δ0: 1.004975,
      β: 288,  d: 4013,  m: 1964
 dec: rop: =2^127.1,  m:  =2^11.1,  red: =2^127.1,  δ0
: 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,  δ0
: 1.004864,
      β: 298,
      repeat:  =2^58.8,  d: 4090,  c:    3.909,
k: 30, postprocess: 13
```

FIGURE 2 – Analyse de sécurité des paramètres tirés de la librairie SEAL

— les paramètres n, q habituels ;
— un paramètre α égal à s/q où s est le paramètre de la gaussienne discrète utilisée comme paramètre χ (à ne pas confondre avec le paramètre α de l'hypothèse 1) ;
— d'autres arguments optionnels.

Le paramètre `m` est optionnel car chaque attaque contre LWE évaluée utilise le nombre d'échantillons `m` qui lui est nécessaire pour fonctionner et l'indique en sortie de la fonction. En réalité, même en indiquant un `m` en option, la sortie peut préciser des valeurs de `m` supérieure à notre indication.

Elle retourne ensuite une résumé de la mémoire, du temps et d'autres paramètres spécifiques que nécessitent diverses attaques contre LWE avec ce choix de paramètres . Le module contient 6 attaques différentes, mais n'en testera que trois par défaut. Cela peut être modifié lorsque l'on appelle la fonction `estimate_lwe` via l'argument skip.

Une sortie de la fonction `estimate_lwe` est montrée dans la figure 2. Les principaux paramètres concernant le coût de chaque attaque sont `rop`, `mem` et `m`, où :
— `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 les paramètres choisis.

5.3.2 Proposition de choix sécurisés pour très faible profondeur de NAND

Commençons par noter que `lwe_estimate` n'arrive pas à estimer la sécurité des paramètres du bootstrapping (étudiés dans une section ultérieure), même avec un paramètre de sécurité $\lambda = 50$. Autre mauvaise nouvelle, la sécurité estimée des paramètres leveled que l'on a trouvé n'est pas bonne, ni pour $\epsilon = 1/3$, ni pour un epsilon bien plus petit, et ce même avec de grands λ, le paramètre `rop` reste proche de 2^{30} pour une des attaques. Cela peut signifier plusieurs choses :
— l'hypothèse de sécurité faite dans [13] n'est plus réaliste ;
— le choix de ϵ doit être fait finement ;
— l'estimation est trop grossière.

Nous n'avons pas été dans le détail pour voir ce qu'il en est.

Nous nous sommes donc tournés vers d'autres choix de paramètres concrets, parmi ceux proposés dans [7].

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_estimator` indique que 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, la condition de longueur est respectée pour $L = 3$ donc une profondeur de 3 NAND est possible.
Taille : secret : 15 Ko, une clé publique : 7.6 Mo, chiffré d'un message : 13 Go.

En utilisant les paramètres suivants, tirés de [2] :

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

on voit que l'estimation proposée par lwe indique que 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, la condition de longueur est respectée pour $L = 1$ donc une profondeur de 3 NAND est possible.
Taille : secret : 3 Ko, clé publique : 5 Mo, chiffré d'un message : 2 Go.

6 Mise en place d'un bootstrapping

Dans cette section, nous supposerons que q est une puissance de 2, ne regarderons des chiffrés que de 0 ou 1.

Afin de pouvoir effectuer un bootstrapping à partir de l'algorithme de déchiffrement **Decrypt**, nous allons avoir besoin de d'exprimer, pour tout chiffré C, l'algorithme de déchiffrement de C comme un circuit booléen ayant pour entrée le chiffré de la clé secrète, pour cela préalablement représenté uniquement à l'aide de 0 et de 1.

Nous allons donc créer ce circuit et nous trouverons une majoration de la profondeur de NAND qu'il nécessite afin de pouvoir, dans une dernière partie, trouver des paramètres sécurités permettant le bootstrapping.

Rappelons enfin que $l = \log_2(q) + 1$ et que $N = (n + 1) * l$.

6.1 Un point sur la sécurité

Nous avons vu que le système cryptographique que nous étudions est IND-CPA. La première question qui se pose pour le bootstrapping est de savoir si il a la sécurité circulaire, expliquée dans la définition 11. Nous n'avons pas trouvé de démonstration de cette sécurité dans l'article original [11] ou encore dans d'autres expositions (comme [13]). Nous devons donc la supposer pour cette section.

6.2 Un premier découpage

Comme la clé secrète est un vecteur de $\mathbb{Z}_q^N$ et qu'un élément de $\mathbb{Z}_q$ a une décomposition binaire d'au plus $\log_2(q)$ éléments, nous pouvons représenter la clé secrète par une liste de $N \log_2(q)$ 0 et 1.

L'algorithme que nous allons étudier est le suivant :

`decrypt(C) :`
`trouver` $1 \leqslant i \leqslant l$ `tel que` $q/4 \leqslant 2^i < q/2$
`calculer` $a = C_i \cdot \vec{v}$
`retourner` $|\frac{a}{\vec{v}_i}|$

Insistons sur le fait que le chiffré C n'est pas une entrée du circuit. Notamment, même lors de l'application homomorphe de ce circuit, nous pouvons supposer que C est connu, alors que seul les chiffrés des $N \log_2(q)$ éléments de la représentation en 0 et 1 de la clé secrète sera connu.

Nous allons tout de suite profiter de cela en remarquant que

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

Les valeurs $C_{i,j,k} \in \{0,1\}$ étant connues, on réduit le problème de calculer un produit scalaire à celui de faire la somme d'au plus $l * N = |q|^2_{\text{bin}}(n+1)$ listes constitués de 0 et de 1.

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.

De plus, pour f et g des formules de profondeur de NAND respectives u et v, notons que :
- $\overline{f}$ a une profondeur $u+1$;
- $f \wedge g$ et $f \vee g$ ont une profondeur de $\max(u,v)+2$;
- $f \oplus g$ a une profondeur de $\max(u,v)+4$;

6.3 Sommer des listes de 0 et de 1 en minimisant la profondeur de NAND

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

Commençons tout d'abord par étudier la somme de 2 listes.

6.3.1 basic_sum : addition classique de deux listes

Il s'agit de l'algorithme naïf de somme de deux nombres en base 2, 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 veut sommer. La somme

$$D = \sum_{i=0}^{s-1} d_i 2^i$$

est alors définie par :

```
c_{-1} = 0
for i in range(s):
        d_i = a_i ⊕ b_i ⊕ c_{i-1}
        c_i = (a_i ∧ b_i) ∨ (c_{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 d_{s-1} dépend de a_0 et b_0, et ce à cause de la récursivité du calcul des c_i.

Calculer d_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. c_i est calculé en appliquant un AND et un OR à c_{i-1}, ce qui ajoute aussi 4 à la profondeur. $c_0 = a_0 \wedge b_0$ et peut donc être trouvé avec une profondeur de 2 NAND.

On obtient alors la proposition suivante :

Proposition 14. *L'algorithme* ***basic_sum*** *nécessite une profondeur de* $4*s-2$ *NAND.*

Il s'avère qu'il existe une méthode pour additionner deux listes en $\mathcal{O}(\log_2(s))$. Son idée consiste à ne pas calculer les retenues une à une « linéairement », mais à introduire un arbre binaire permettant de faire en partie en parallèle le calcul des retenues. Il s'agit du carry lookahead adder que nous présentons maintenant.

6.3.2 carry lookahead adder : addition de deux listes avec une profondeur plus faible de NAND

Nous appellerons **cla_sum** cet algorithme. Sa différence avec la somme classique est que les retenues sont en partie calculées en parallèle via une structure en arbre binaire, permettant de diminuer la profondeur de NAND utilisée.

Bien que nous en avons vu plusieurs expositions, nous n'en avons pas trouvé une qui définissait complètement le cas général. Nous exposons donc ici le détail des formules et des démonstrations sur la profondeur en NAND.

Notons que pour simplifier les formules, et uniquement dans cette sous-section, nous ferons ici commencer les indices de listes par 1.

On suppose que les listes ont pour taille une puissance de deux : $s = 2^u$. On pose alors, notant le OU logique par l'addition et le ET logique par un produit :

$G1_i = a_i b_i \quad P1_i = a_i + b_i \qquad$ pour $1 \leqslant i \leqslant 2^u$

$G2_i = G1_{2i} + G1_{2i-1}P1_{2i} \qquad P2_i = P1_{2i-1}P1_{2i} \quad$ pour $1 \leqslant i \leqslant 2^{u-1}$

$\cdots$

$(G2^k)_i = (G2^{k-1})_{2i} + (G2^{k-1})_{2i-1}(P2^{k-1})_{2i} \qquad (P2^k)_i = (P2^{k-1})_{2i-1}(P2^{k-1})_{2i} \quad$ pour $1 \leqslant i \leqslant 2^{u-k}$

$\cdots$

$G2^u = (G2^{u-1})_2 + (G2^{u-1})_1(P2^{u-1})_2 \qquad P2^u = (P2^{u-1})_1(P2^{u-1})_2.$

Les variables G sont dites variables de générations et celles avec un P sont dites variables de propagations. Ces noms sont justifiés par la propriété suivante :

Proposition 15. *Considérons* $0 \leqslant k \leqslant u$ *et* $1 \leqslant i \leqslant 2^{u-k}$ *et notons B le bloc :*

$$(i-1)2^k + 1, \cdots, i\,2^k$$

On considère alors $a_{|B}$ *et* $b_{|B}$ *les nombres binaires obtenus en restreignant les écritures binaires de a et b aux positions B. Alors,*

- $(G2^k)_i$ *est vraie si et seulement si l'algorithme de somme binaire classique entre* $a_{|B}$ *et* $b_{|B}$ *crée une retenue à la dernière position, c'est à dire à la position*[7] 2^k*. On dit alors que le bloc B génère une retenue.*
- $(P2^k)_i$ *est vraie si et seulement si, considérant l'algorithme de somme binaire entre* $a_{|B}$ *et* $b_{|B}$ *où on ajoute une retenue à la position 1 (ce qui revient à considérer la somme* $a_{|B} + b_{|B} + 1$*), il y a alors une retenue en dernière position. On dit alors que le bloc B propage les retenues.*

Démonstration. Procédons par récurrence. Pour $k = 0$, cela est facile : la somme de deux bits u et v ne produit une retenue que si uv est vraie, et ne propage une retenue que si $u + v$ est vraie.

7. en commençant à compter à partir de 1

Maintenant, observons les formules génériques :

$$(G2^k)_i = (G2^{k-1})_{2i} + (G2^{k-1})_{2i-1}(P2^{k-1})_{2i}$$

$$(P2^k)_i = (P2^{k-1})_{2i-1}(P2^{k-1})_{2i}.$$

En notant L et R les deux moitiés de B ($B = L||R$) et en admettant par récurrence que la proposition est vraie pour les variables $(G2^{k-1})_j$ et $(P2^{k-1})_j$ pour tout j, on peut dire que :

— $(G2^k)_i$ n'est vraie que si R génère une retenue ou bien si L en génère une qui est propagée par R. Ce qui est équivalent à dire que B génère une retenue.

— $(P2^k)_i$ n'est vraie que si une retenue arrivant au début du bloc L est propagée et qu'une retenue arrivant au début du bloc R est propagée. Ce qui, en composant les deux propagations, est équivalent à dire qu'une retenue arrivant au début du bloc B est propagée.

□

Proposition 16. *Le calcul de $(G2^k)_i$ a une profondeur en NAND de $2 + 4k$ et celui de $(P2^k)_i$ de $2 + 2k$.*

Démonstration. La démonstration est directe par récurrence. □

Nous allons maintenant utiliser ses variables pour calculer les retenues c_j. Pour cela, on définit pour un entier $a \in Z$:

$$\nu(a) = 2^k \text{ où } k = \min\{j : 2^j | a\},$$
$$\theta(a) = m2^{\nu(a)} \text{ où } m2^{\nu(a)} \leqslant a < (m+1)2^{\nu(a)}.$$

Notons qu'alors

$$a = \nu(a) + \theta(a).$$

Par exemple :

$$\nu(6) = 2,\ \theta(6) = 4;\ \nu(13) = 1,\ \theta(13) = 12;\ \nu(12) = 4,\ \theta(12) = 8;\ \nu(1) = 1,\ \theta(1) = 0.$$

On définit alors les retenues c_j par la formule :

$$c_j = G\nu(j)_{j/\nu(j)} + c_{\theta(j)}P\nu(j)_{j/\nu(j)} \tag{2}$$

où $c_0 = 0$ par définition.

Voyons cela sur un exemple si on considère des listes de taille $8 = 2^3$, les formules des retenues sont :

$$c_1 = G1_1 + c_0P1_1 = G1_1$$
$$c_2 = G2_1 + c_0P2_1 = G2_1$$
$$c_3 = G1_3 + c_2P1_3$$
$$c_4 = G4_1 + c_0P4_1 = G4_1$$
$$c_5 = G1_5 + c_4P1_5$$
$$c_6 = G2_3 + c_4P2_3$$
$$c_7 = G1_7 + c_6P1_7$$
$$c_8 = G8_1 + c_0P8_1 = G8_1$$

Ainsi, si on veut additionner $a = 00111100$ et $b = 01010101$, on obtient :

a	0	0	1	1	1	1	0	0
b	0	1	0	1	0	1	0	1
G1	0	0	0	1	0	1	0	0
P1	0	1	1	1	1	1	0	1
G2		0		1		1		0
P2		0		1		1		0
G4				1				0
P4				0				0
G8								0
P8								0

et on trouve avec les formules précédentes :

$$(c_1, ..., c_8) = (0, 0, 0, 1, 1, 1, 0, 0)$$

ce qui correspond bien aux retenues.

On va maintenant voir que cette méthode est bien plus intéressante que la méthode classique concernant la profondeur de NAND.

Proposition 17. *Pour deux listes de taille u, calculer toutes les retenues nécessite une profondeur de moins de $8\log_2(u) + 2$ NAND.*

Démonstration. Notant $D(f)$ la profondeur de NAND d'une formule f, la formule (2) nous permet de voir que pour tout $1 \leqslant j \leqslant 2^u$:

$$\begin{aligned} D(c_j) &\leqslant 2 + \max\left[2 + 4\log_2(\nu(j)),\ \ 2 + \max\left(D(c_{\theta(j)}), 2 + 2\log_2(\nu(j))\right)\right] \\ &\leqslant 2 + \max\left(2 + 4\log_2(\nu(j)),\ \ 2 + D(c_{\theta(j)})\right) \\ &\leqslant 4 + \max\left(2 + 4\log_2(\nu(j)), D(c_{\theta(j)})\right) \\ &\leqslant 4\log_2(u) + (2 + 4\log_2(u)) = 8\log_2(u) + 2. \end{aligned}$$

La dernière inégalité utilisant $D(c_0) = D(0) = 0$ ainsi que le fait que la séquence $(j, \theta(j), \theta(\theta(j)), ...)$ devient constante à 0 en moins de $\log_2(u)$ étapes.

Cela se montre en constatant que $\theta^n(j)$ est un multiple positif de 2^τ inférieur à u, pour un $\tau > n$ et ne peut donc valoir que 0 si $n > \log_2(u)$. □

Une fois calculée les retenues $(c_i)_i$, la somme entre deux listes a et b se fait comme la somme classique, avec : $r_i = a_i \oplus b_i \oplus c_i$.

On en déduit donc :

Théorème 4. *L'algorithme **cla_sum** fait la somme de 2 listes de taille s avec une profondeur de moins de $8\lceil\log_2(s)\rceil + 6$ NAND, donc en $\mathcal{O}(log_2(s))$.*

Démonstration. Il suffit de compléter les listes par des 0 pour se réduire au cas où la taille vaut $2^{\lceil\log_2(s)\rceil}$ et d'appliquer le travail fait précédemment. □

Même si **cla_sum** est très efficace, l'utiliser sur plusieurs listes peut s'avérer couteux. Nous allons maintenant voir une opération dont la profondeur de NAND est constante et qui nous permettra de n'utiliser cla_sum qu'une seule fois pour faire la somme de plusieurs listes.

6.3.3 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 retourne deux nombres

$$X = \sum_{i=0}^{s-1} x_i 2^i \quad Y = \sum_{i=0}^{s-1} y_i 2^i$$

tels que

$$A + B + C = X + Y.$$

Ils se construisent ainsi, avec la convention $v_{-1} = 0$ pour toute liste v :

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

$$\begin{aligned} x_i &= a_i \oplus b_i \oplus c_i \\ y_i &= \overline{(\overline{a_{i-1}} \wedge \overline{b_{i-1}}) \oplus (\overline{b_{i-1}} \wedge \overline{c_{i-1}}) \oplus (\overline{a_{i-1}} \wedge \overline{c_{i-1}})} \end{aligned}$$

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

Or :

— 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.

On a donc la proposition suivante :

Proposition 18. *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 **cla_sum**, mais comment organiser ces "additions" de façon à minimiser la profondeur totale ?

6.3.4 Sommer nb listes

L'algorithme naïf n'utilisant que **cla_sum** consisterait à appliquer **cla_sum** à deux listes, puis à sommer le résultat de ce calcul avec une nouvelle liste et ainsi de suite jusqu'à avoir sommé toutes les nb listes. On voit qu'ainsi, l'arbre dont les nœuds correspondent à une application de **cla_sum** aurait alors une profondeur de $nb - 1$, ce qui fait que la profondeur totale nécessaire serait bornée par $(nb - 1)(8\lceil \log_2(s) \rceil + 6)$.

Nous avons toutefois vu qu'il y a tout intérêt à utiliser **reduced_sum** plutôt que **cla_sum** tant que cela est possible. Il semble donc préférable d'appliquer **reduced_sum** à trois des listes à additionner, puis d'appliquer à nouveau l'algorithme aux deux résultats auxquels on à ajouté une nouvelle liste et procéder ainsi de suite jusqu'à ne plus avoir que deux listes qu'on somme alors avec **cla_sum**.

L'arbre dont les nœuds correspondent à une application de **reduced_sum** ou bien de **cla_sum** a alors une profondeur de $nb - 1$, avec une seule utilisation de **cla_sum**. On a donc une profondeur en NAND majorée par :

$$12(nb - 2) + 8\lceil \log_2(s) \rceil + 6 = 12nb + 8\lceil \log_2(s) \rceil - 18$$

Les additions peuvent cependant être bien mieux ordonnées, en utilisant un arbre équilibré, c'est à dire ayant une profondeur équilibrée entre ses branches.

Dans l'exemple n'utilisant que **basic_sum**, on peut par exemple prendre comme arbre équilibré un arbre binaire (quitte à compléter les listes par des 0 pour [8] que leur taille soit une puissance de 2), ce qui donne une profondeur en NAND de $(\lfloor \log_2(nb) \rfloor)(8\lceil \log_2(s) \rceil + 6)$

8. ou mieux, par une valeur particulière comme -1 permettant de savoir qu'il n'y avait rien ici. Le gain peut sembler ridicule vu que l'on parle de somme de bits, mais il ne faut pas oublier que ce circuit s'applique ensuite homomorphiquement

Dans l'exemple utilisant d'abord **reduced_sum** puis une fois **cla_sum**, cela se fait de la même façon, même si la démonstration est un peu moins évidente du fait que **reducted_sum** prend 3 listes et en retourne 2.

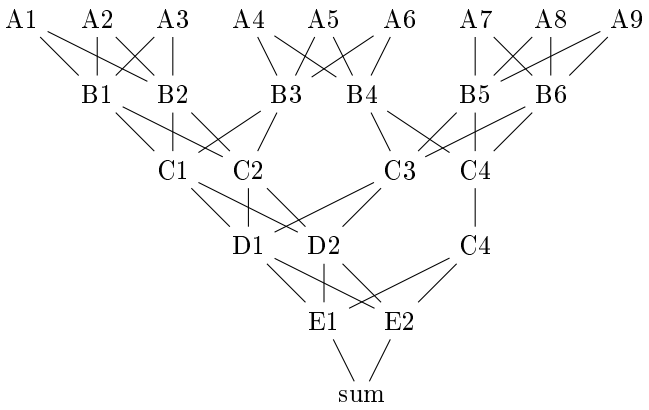
Proposition 19. *Il est possible de sommer nb listes de s éléments avec une profondeur de NAND majorée par* $8\lceil\log_2(s)\rceil + 36\lceil\log_3(nb)\rceil - 18$. *Soit, une profondeur en* $\mathcal{O}(\log_3(nb) + \log_2(s))$.

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 (quitte à rajouter au plus 2 listes de zéros) et en leur appliquant **reduced_sum**, puis en recommençant 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 $3(p-2)$.

Supposant maintenant, quite à rajouter des listes de zéros, que $nb' = 9$. Le dessin suivant montre comment calculer la somme totale avec une profondeur de quatre **reduced_sum** et un **basic_sum** :



En utilisant la proposition 18 ainsi que le théorème 4, on obtient que la profondeur totale en NAND est majorée par :

$$8\lceil\log_2(s)\rceil + 6 + 12(3p-2) = 8\lceil\log_2(s)\rceil + 36p - 18.$$

Or, par définition, $p = \lceil\log_3(nb)\rceil$, ce qui permet de conclure. □

6.4 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[9] $l-1$, le bit de poids faible étant à gauche. Autrement dit :

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

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

```
if a_{l-2} = 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 cla_sum(a, [1,0,...,0])
```

9. 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

Toutefois, nous devons représenter cette algorithme par un circuit booléen et il n'est donc pas possible de faire de conditions. On va donc considérer la modification du code suivante :

```
b = cla_sum([¬a_i for i in range(l)], [1,0,...,0] )
return [(a_{l-1} ∧ a_i) ∨ (¬a_{l_1} ∧ b_i) for i in range(l)]
```

Notre algorithme a encore un dernier problème : il n'est pas possible d'utiliser des constantes dans notre circuit booléen, on ne peut donc à priori pas faire l'instruction

```
b = cla_sum([¬a_i for i in range(l)], [1,0,...,0] )
```

Notons toutefois que pour un booléen x

$$\overline{x \wedge 0} = x \qquad \overline{x \wedge 1} = \overline{x}$$

Il est donc possible de remplacer tout NAND avec une constante par une opération sans constante ayant une profondeur de NAND inférieur ou égale.
On peut donc se passer de cette constante, et le calcul de profondeur que l'on effectue avec cette constante majore celui qu'on aurait fait en s'en passant.

Utilisons le théorème 4 pour conclure :

Proposition 20. *Il est possible de calculer la valeur absolue en binaire d'un élément $a \in \mathbb{Z}_q$ avec une profondeur de moins de $8\lceil \log_2(s) \rceil + 11$ NAND.*

Maintenant que nous avons étudié comment sommer des listes et comment calculer la valeur absolue dans $\mathbb{Z}_q$, nous pouvons déterminer une majoration du nombre de NAND nécessaire pour appliquer l'algorithme de déchiffrement tel que décrit dans le listing 6.2 page 23.

Théorème 5. *Si q est une puissance de 2, on peut effectuer l'algorithme **Decrypt** avec une profondeur de NAND plus petite que*

$$88\lceil \log_2(\log_2(q)) \rceil + 36\lceil \log_2(n) \rceil$$

et donc, en $\mathcal{O}(\log_2(n) + \log_2(\log_2(q)))$

Démonstration. En utilisant les propositions précédentes ainsi que le fait qu'il y a au plus $N * l = \log_2(q)^2 * (n+1)$ listes à sommer, on majore la profondeur de NAND par

$$16\lceil \log_2(\log_2(q)) \rceil + 36 \log_3\left(\log_2(q)^2 * (n+1)\right) = 16\lceil \log_2(\log_2(q)) \rceil + 72\lceil \log_3(\log_2(q)) \rceil + 36\lceil \log_3(n+1) \rceil$$

et on majore le logarithme en base 3 par celui en base 2 pour conclure, en utilisant aussi que $\log_3(n+1) \leqslant \log_2(n)$ (dès $n \geqslant 2$). □

Nous avons maintenant toutes les informations nécessaires pour choisir des paramètres permettant le bootstrapping.

6.5 Choix asymptotique de paramètres pour GSW avec bootstrapping

On va utiliser l'inégalité grossière :

$$88\lceil \log_2(\log_2(q)) \rceil + 36 \log_2(\log_2(n)) \leqslant 90 \left(\log_2(\log_2(q)) + \log_2(n)\right)$$

pour voir que, pour obtenir un GWS avec bootstrapping, il suffit de pouvoir évaluer les circuits dont la profondeur en NAND est majorée par

$$L = 90 \left(\log_2(\log_2(q)) + \log_2(n)\right).$$

L'inégalité est assez grossière pour qu'on n'ait pas besoin de rajouter de 1 à la profondeur maximale pour s'assurer de pouvoir effectuer des calculs hors de ce circuit.

On a vu dans la section 5.2 que pour avoir une profondeur de L NAND, il suffit d'avoir

$$q > 16(1+N)^{L+2}.$$

En injectant la valeur précédente dans L et en utilisant une majoration grossière, on voit qu'il suffit d'avoir

$$q > ((n+1)(\lceil \log_2(q) \rceil + 1))^{90(\log_2(n)+\log_2(\log_2(q)))}. \tag{3}$$

Suivant le raisonnement de Shai Halevi dans [13], on remarque qu'il existe un $\rho > 90$ tel que

$$((n+1)(\lceil \log_2(q) \rceil + 1))^{90(\log_2(n)+\log_2(\log_2(q)))} < (n\log_2(q)))^{\rho\,(\log_2(n)+\log_2(\log_2(q)))}.$$

En utilisant l'hypothèse 3 (page 6), notre jeu de contraintes est alors :

$$\begin{cases} \alpha = n/q \\ q = 2^{\mathrm{polylog}(n)} \\ m = 2(n+1)|q|_{\mathrm{bin}} \\ q > (n\log_2(q)))^{\rho\,[\log_2(n)+\log_2(\log_2(q))]} \end{cases}$$

Cela nous amène à un choix de paramètres :

Théorème 6. *Paramètres sécurisés pour GSW avec bootstrapping*

Nous gardons les notations précédentes. Sous l'hypothèse de sécurité circulaire ainsi que l'hypothèse 3 de difficulté de DWLE, les paramètres suivants rendent le cryptosystème GSW bootstrappable et IND-CPA :

$$\begin{cases} n = \lambda \\ q = \lceil n^{\rho\log^2(n)} \rceil \\ m = 2(n+1)|q|_{bin} \\ \alpha = n/q = 2^{\Theta(\log^2(n))} \end{cases}$$

Démonstration. Il nous faut montrer que l'inégalité

$$q > (n\log_2(q))^{\,\rho\,(\log_2(n)+\log_2(\log_2(q)))}$$

est vérifiée pour n assez grand.

En substituant q par sa valeur à gauche, l'inégalité devient :

$$n^{\rho\log^2(n)} > (n\log_2(q)))^{\,\rho\,(\log_2(n)+\log_2(\log_2(q))}.$$

En passant au logarithme, l'inégalité devient :

$$\log^3(n) > [\log_2(n) + \log_2(\log_2(q))]^2.$$

Or,

$$\log_2(\log_2(q)) \leqslant \log_2(\rho\log^3(n)+1) \leqslant 1 + \log_2(\rho) + 3\log_2(\log_2(n)).$$

Il suffit donc de montrer que :

$$\log^3(n) > (1 + \log_2(\rho) + 3\log_2(\log_2(n)) + \log_2(n))^2,$$

ce qui est vrai pour n assez grand. □

Remarque 2. *Notons que dans notre première tentative, avec* ***basic_sum*** *à la place de* ***cla_sum****, les contraintes obtenues étaient impossibles à respecter.*

	taille du secret	taille de la clé	taille d'un chiffré
$\lambda = 8,\ \rho = 90$	3 Ko	113 Mo	135 Go
$\lambda = 8,\ \rho = 100$	3 Ko	139 Mo	186 Go
$\lambda = 16,\ \rho = 90$	12 Ko	2 Go	6 To
$\lambda = 16,\ \rho = 100$	13 Ko	3 Go	9 To
$\lambda = 32,\ \rho = 90$	45 Ko	32 Go	176 To
$\lambda = 32,\ \rho = 100$	50 Ko	40 Go	242 To

FIGURE 3 – Taille des données suivant les paramètres étudiés précédemment

La figure 3 nous montre la taille nécessaire pour stocker les différents composants de GSW en utilisant ces paramètres.

7 Implémentation d'un FHE avec bootstrapping « jouet »

7.1 Présentation de notre arborescence

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

- `GSW_scheme` contenant l'implémentation du cryptosystème GSW ;
- `analysis` contenant des fonctions permettant de tester notre implémentation sur divers circuits homomorphes ainsi que de voir les performances en terme de sécurités et de profondeur de NAND possible de certains choix de paramètres ;
- `unitary_test` contenant des tests assurant le bon fonctionnement des fonctions codées dans les autres dossiers ;
- `lwe_estimator` contenant les fichiers sources de l'API lwe_estimator présentée dans ce rapport dans la section 5.3.1 page 21 ;

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

7.1.1 GWS_scheme

Ce dossier contient les fichiers suivants :

- `GSW_scheme.sage` contient les fonctions principales de GWS dont `setup`, `encrypt`, `keys_gen` et les 3 algorithmes de déchiffrements que sont `basic_decrypt` (l'algorithme **Decrypt** du rapport), `mp_decrypt` et `mp_all_q_decrypt`. Il contient aussi différentes variables globales, dont `decrypt` permettant d'indiquer quel est l'algorithme de déchiffrement par défaut ;
- `auxilliary_functions.sage` contient l'implémentation des diverses fonctions auxiliaires 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 alors une variable globale `params_maker` permettant de choisir lequel utilise la fonction `setup` du cryptosystème GSW ;
- `homomorphic_functions.sage` contient la version homomorphe d'opérations de base comme la somme ou encore le NAND ;
- `bootstrapping.sage` contient les fonctions nécessaires pour effectuer l'algorithme `basic_decrypt` homomorphiquement (il s'agit de la fonction `h_basic_decrypt`). On y trouve donc notamment les diverses façon de sommer homomorphiquement des listes de chiffrés de 0 et de 1 présentées dans la sous-section 6.3.4 page 6.3.4. Notez que la fonction `bootstrapping_arguments` permet d'effectuer un déchiffrement homomorphe sur chacun des chiffrés passés en argument et que `setup_bs_params` permet d'initialiser des variables globales `bs_pk,bs_sk,bs_`

`lk,bs_params,bs_encrypted_sk` situées dans `bootstrapping.sage` et notamment utilisés dans `analysis/h_circuits_with_bootstrapping.sage`.

7.1.2 analysis

Ce dossier contient les fichiers suivants :

— `depth_security.sage` contient une fonction permettant de lancer `lwe_estimator` sur des paramètres définis dans `params_maker.sage` et une autre permettant de voir quel est le plus grand L satisfaisant la condition de longueur (équation (1) page 12) sur des paramètres définis dans `params_maker.sage`

— `h_circuits_with_bootstrapping.sage` contient des exemples de fonctions utilisant le bootstrapping. Elles permettent de voir si, pour certaines fonctions f, appliquer f homomorphiquement sur des chiffrés permet bien d'obtenir un chiffré de l'application de f aux clairs. On peut toutes les lancer en utilisant la fonction `all_circuit_with_bs` ;

— `h_circuits_without_bootstrapping.sage` contient des examples de fonctions n'utilisant pas de bootstrappings. Tout comme les fonctions de `h_circuits_with_bootstrapping.sage`, elles permettent de voir si, pour certaines fonctions f, appliquer f homomorphiquement sur des chiffrés permet bien d'obtenir un chiffré de l'application de f aux clairs. On peut toutes les lancer en utilisant la fonction `all_circuit_without_bs` ;

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

— `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

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

7.1.3 unitary_test

Ce dossier contient un fichier `framework_test.sage` permettant de mettre en forme les sorties des différentes fonctions de test, ainsi qu'un fichier de test correspondant à chaque fichier du dossier `GSW_scheme` permettant de s'assurer du bon fonctionnement des fonctions du fichier associé. Chacun de ces 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` des autres fichiers de test. On peut donc se faire une idée du travail réalisé sur les tests en la lançant.

8 Des librairies pour du FHE

Plusieurs librairies open source implémentant divers FHE sont disponibles. On peut notamment en trouver une liste sur HomomorphicEncryption.org [14] 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) [19], dont nous avons tiré des paramètres « réalistes » [10] sécurisés et autorisant une profondeur de NAND non nulle (même si irréaliste : seulement 3, voir la sous-section 5.3.2 page 22) ;

— The Gate Bootstrapping API [20] qui implémente une variation du cryptosystème GSW ;

10. Pas forcément pour nos machines et avec notre implémentation

8.1 La librarie SEAL

Acronyme de « Simple Encrypted Arithmetic Library », SEAL [19] 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 [8] et CKKS [3].

Son installation est facile [11] et un executable d'exemple est déjà présent pour tester diverses fonctionnalités de la librarie. De plus, la documentation [15], 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 [20]) qui est claire et bien documentée.

L'API Gate Bootstrapping est une librairie open source utilisable en C et C++ s'appuyant notamment sur des travaux de I. Chillotti, N. Gama, M. Georgieve et M. Izabachène (voir [4] et [5]).

Elle utilise une version modifiée du cryptosysteme GSW [12] étudié dans notre rapport, et permettant aussi bien du LHE que du FHE. C'est pourquoi nous allons en parler plus en détails.

Ses performances sont interessantes ; il est notamment indiqué dans la sous-section 4.2 de [4] que pour un ordinateur 64-bit simple coeur (i7-4930MX) cadencé à 3.00GHz, le bootstrapping se fait en un temps moyen de 52ms et que la clé de bootstrapping fait environ 24 Mo.

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 une variante nommée TFHE, présentée dans [4].

8.2.1 Un exemple simple

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

- Alice génère des clés, chiffre des données et les envoie accompagnées de la clé de bootstrapping au cloud ;
- Le cloud applique homomorphiquement une fonction, le minimum entre deux nombres, aux données et en renvoie le résultat 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'exécutable, à condition d'avoir la librairie tfhe installée.

Les fichiers sources ainsi que les headers contiennent normalement assez de commentaire pour être lisibles. Voici un résumé du rôle de chaque fichier source :

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

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

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] Sageloli Eric and Roux Lucas. Github du projet. Accessed : 2019-02-15.

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

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

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

[10] 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.

[11] 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.

[12] 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.

[13] Shai Halevi. A fully homomorphic encryption scheme, 2017. https://shaih.github.io/pubs/he-chapter.pdf.

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Zhongxiang Zheng, Guangwu Xu, and Chunhuan Zhao. Discrete gaussian measures and new bounds of the smoothing parameter for lattices. Cryptology ePrint Archive, Report 2018/786, 2018. `https://eprint.iacr.org/2018/786`.