

# Étude d'un FHE de troisième génération

---

Lucas Roux & Eric Sageloli

17 février 2019

# Introduction

---

# Un bref historique

**Définition informelle** : un FHE (Fully homomorphic encryption) est un cryptosystème dont les chiffrés sont sur définis sur un anneau  $R$  et ayant des opérations sur les chiffrés qui « commutent » avec les opérations d'addition, de multiplication et de multiplication par un scalaire.

**Définition alternative** : l'ensemble des messages est  $\{0, 1\}$  et le cryptosystème commute avec l'opération NAND.

# Un bref historique

**Définition informelle** : un FHE (Fully homomorphic encryption) est un cryptosystème dont les chiffrés sont sur définis sur un anneau  $R$  et ayant des opérations sur les chiffrés qui « commutent » avec les opérations d'addition, de multiplication et de multiplication par un scalaire.

**Définition alternative** : l'ensemble des messages est  $\{0,1\}$  et le cryptosystème commute avec l'opération NAND.

- 2009 : un premier plan par C. Gentry dans sa thèse :
  - idée du bootstrapping ;
- 2011 : premiers FHE de seconde génération : Z. Brakerski, V. Vaikuntanathan, J. Fan, F. Vercauteren :
  - basés sur LWE et ses variantes (comme RLWE) ;
  - une somme simple à définir ;
  - un produit en 2 étapes ;
- 2013 : premiers FHE de troisième génération : GWS par C. Gentry, B. Waters and A. Sahai, en 2013 :
  - basés sur LWE et ses variantes ;
  - produit et somme de même nature ;

## Définition de GSW

---

# GSW, premier essai :

**Clé secrète** : un vecteur  $\vec{sk} \in \mathbb{Z}_q^n$

**Clé publique** : pk

**Chiffrement** :  $\text{Encrypt}(\text{pk}, \mu) = C \in \mathbb{Z}_q^{n \times n}$  telle que

$$C \vec{sk} = \mu \vec{sk}$$

**Déchiffrement** : évident : recherche de valeur propre

**Opérations homomorphes** :

Pour  $C_i = \text{Encrypt}(\mu_i)$  ( $1 \leq i \leq 2$ ) et  $\lambda \in \mathbb{Z}_q$ ,

- **Somme** :  $C_1 + C_2$

$$(C_1 + C_2) \vec{sk} = (\mu_1 + \mu_2) \vec{sk}$$

- **Produit** :  $C_1 \times C_2$

$$(C_1 \times C_2) \vec{sk} = C_1 (\mu_2 \vec{sk}) = (\mu_1 \mu_2) \vec{sk}$$

- **Produit par scalaire** :  $\lambda C_1$
- **NAND** :  $C_1 \times C_2 - \text{Id}$

**Clé secrète** : un vecteur  $\vec{sk} \in \mathbb{Z}_q^n$

**Clé publique** : pk

**Chiffrement** :  $\text{Encrypt}(\mu) = C \in \mathbb{Z}_q^{n \times n}$  telle que

$$C\vec{sk} = \vec{sk} + \vec{e} \quad \text{avec } \vec{e} \text{ petit}$$

**Déchiffrement** : on prend un  $i$  tel que  $\vec{sk}_i$  est grand

$$\begin{aligned} \text{Decrypt}(\vec{sk}, C) &= \left\lfloor \frac{\langle C_i, \vec{sk} \rangle}{\vec{sk}_i} \right\rfloor = \left\lfloor \frac{\mu \vec{sk}_i + \vec{e}_i}{\vec{sk}_i} \right\rfloor \\ &= \left\lfloor \mu + \frac{\vec{e}_i}{\vec{sk}_i} \right\rfloor \\ &= \mu \end{aligned}$$

Retour sur les opérations homomorphes :

- **Somme** :  $C_1 + C_2$

$$(C_1 + C_2) \vec{sk} = (\mu_1 + \mu_2) \vec{sk} + \vec{e}_1 + \vec{e}_2$$

- **NAND** :  $C_1 \times C_2 - \text{Id}$

$$\begin{aligned}(C_1 \times C_2 - \text{Id}) \vec{sk} &= C_1 \left( \mu_2 \vec{sk} + \vec{e}_2 - \vec{sk} \right) \\ &= (\mu_1 \mu_2 - 1) \vec{sk} + \mu_2 \vec{e}_1 + C_1 \vec{e}_2\end{aligned}$$

analysons  $\mu_2 \vec{e}_1 + C_1 \vec{e}_2$  :

- $\mu_2 \vec{e}_1$  ne rajoute pas beaucoup d'erreur ;
- $C_1 \vec{e}_2$  est plus problématique.



# GSW, troisième tentative

On utilise une fonction Flatten qui a notamment les propriétés suivantes :

$$C \in \mathbb{Z}_q^{n \times n} \implies \text{Flatten}(C) \in \{0, 1\}^{n \times n}$$

$$\text{Flatten}(C) \cdot \vec{sk} = C \cdot \vec{sk} \quad \text{pour un secret } \vec{sk} \text{ bien choisi}$$

**Clé secrète** : un vecteur  $\vec{sk} \in \mathbb{Z}_q^n$  bien choisi

**Clé publique** : pk

**Chiffrement** :  $\text{Encrypt}(\text{pk}, \mu) = \text{Flatten}(C) \in \mathbb{Z}_q^{n \times n}$  pour  $C$  telle que

$$C \vec{sk} = \vec{sk} + \vec{e} \quad \text{avec } \vec{e} \text{ petit}$$

**Déchiffrement** : on prend un  $i$  tel que  $\vec{sk}_i$  est grand et :

$$\text{Decrypt}(\vec{sk}, C) = \left\lfloor \frac{\langle C_i, \vec{sk} \rangle}{\vec{sk}_i} \right\rfloor$$

**Opérations homomorphes** : on applique Flatten aux précédentes opérations homomorphes.

# Le problème Decisional Learning With Error (DLWE)

**Paramètres** : le paramètre de sécurité  $\lambda$ , la dimension  $n = n(\lambda) \in \mathbb{N}$ , le module  $q = q(\lambda) \in \mathbb{N}$ , une distribution  $\chi = \chi(\lambda)$  à valeur dans  $\mathbb{Z}_q$ , un paramètre de nombre d'échantillons  $m = m(\lambda) \in \mathbb{N}$ .

**Problème DLWE**( $n, q, \chi, m$ ) : distinguer si  $A \in \mathbb{Z}_q^{m \times n+1}$

- a été échantillonnée uniformément ;
- $A = (\vec{b}^\top || B)$  où  $B \in \mathbb{Z}_q^{m \times n}$  est échantillonnée uniformément et

$$\vec{b} = \vec{e} + B\vec{t}$$

pour  $\vec{e}$  échantillonné par  $\chi$  et  $\vec{t}$  uniformément.

En notant  $\vec{sk} = (1 \ - \vec{t})$ , on a

$$A \vec{sk} = \vec{e}$$

**Hypothèse DLWE** : il existe une famille de paramètres telle qu'aucun algorithme polynomial  $\mathcal{A}$  n'ait un avantage non négligeable pour distinguer les deux cas.

# Clé publique et sécurité IND-CPA

**Idée** : prendre  $pk = A$ , et  $\chi$  qui échantillonne de petites valeurs.

Pour chiffrer  $\mu \in \mathbb{Z}_q$  :

$$C := \mu \text{Id} + RA \quad R \in \{0, 1\}^{m \times n} \text{ tirée uniformément.}$$

Ainsi :  $C \vec{s}k = C\mu + R\vec{e} = C\mu + \vec{e'}$  avec  $\vec{e'}$  petit

**Sécurité** : pour des paramètres t.q l'hypothèse DLWE est vérifié :

- $A$  est indistinguishable d'une matrice choisie uniformément.
- $\nexists$   $\mathcal{A}$  algorithme polynomial probabiliste distinguant :
  - $(A, RA)$
  - un couple de matrices choisies uniformément.

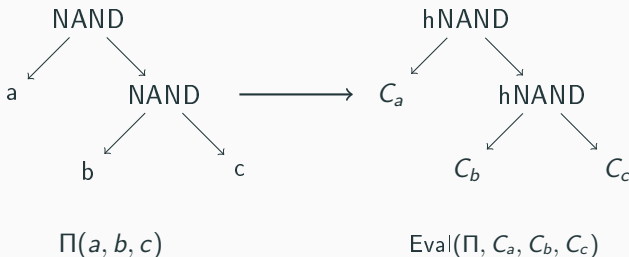
$C$  est indistinguishable d'un one-time pad.

# Mise en place d'un bootstrapping

---

# Evaluation d'un circuit booléen, profondeur de NAND

Soit  $a, b, c \in \{0, 1\}$  et  $C_a, C_b, C_c$  leurs chiffrés pour des clés  $(\vec{sk}, pk)$



Si le circuit  $\Pi$  a une profondeur de NAND assez faible :

$$\Pi(a, c, b) = \text{Decrypt}(\vec{sk}, \text{Eval}(\Pi, C_a, C_b, C_b))$$

On note :

$$\text{Encrypt}(pk, \Pi(a, c, b)) \approx \text{Eval}(\Pi, C_a, C_b, C_b)$$

# FHE avec bootstrapping

$$C = D + \text{erreur}$$

$$C = D + \text{erreur} \xrightarrow{\text{Decrypt}(\vec{sk}, C)} \mu \xrightarrow{\text{Encrypt}(pk, \mu)} C_{\text{new}} = D + \text{erreur}$$

# FHE avec bootstrapping

$$C = D + \text{erreur}$$

$$C = D + \text{erreur} \xrightarrow{\text{Decrypt}(\vec{sk}, C)} \mu \xrightarrow{\text{Encrypt}(pk, \mu)} C_{\text{new}} = D + \text{erreur}$$

Soit  $\Pi$  le circuit booléen tel que

$$\Pi(\overrightarrow{binsk}) = \text{Decrypt}(\vec{sk}, C)$$

# FHE avec bootstrapping

$$C = D + \text{erreur}$$

$$C = D + \text{erreur} \xrightarrow{\text{Decrypt}(\vec{sk}, C)} \mu \xrightarrow{\text{Encrypt}(pk, \mu)} C_{\text{new}} = D + \text{erreur}$$

Soit  $\Pi$  le circuit booléen tel que

$$\Pi(\overrightarrow{binsk}) = \text{Decrypt}(\vec{sk}, C)$$

Alors :

$$\begin{aligned} \text{Encrypt}(pk, \text{Decrypt}(\vec{sk}, C)) &= \text{Encrypt}(pk, \Pi(\overrightarrow{binsk})) \\ &\approx \text{Eval}(\Pi, \text{Encrypt}(pk, \overrightarrow{binsk})) \end{aligned}$$

- Si  $\Pi$  contient assez peu de NAND, on peut créer un FHE.



# Déchiffrement homomorphe : description de $\Pi$

L'algorithme de déchiffrement est le suivant :

1. trouver  $i$  tel que  $\vec{sk}_i$  est grand et une puissance de 2 ;
  2. calculer  $a = \langle C_i, \vec{sk} \rangle$
  3. retourner  $|\frac{a}{\vec{sk}_i}|$
- On peut ramener le calcul du produit scalaire à une somme de nombres binaire ;
  - La division est simplement un shift sur l'écriture binaire ;
  - Calculer la valeur absolue implique essentiellement de faire un complément à 2.

Voyons comment sommer deux nombres binaires.

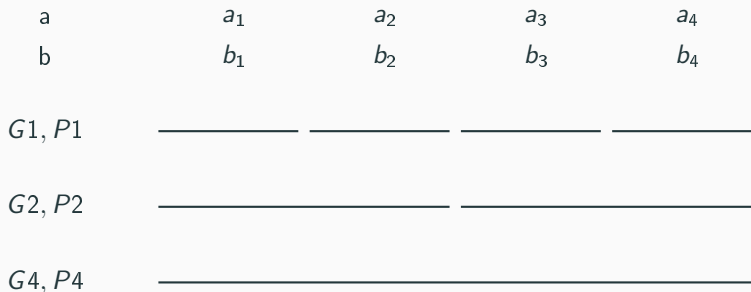
## Sommer deux listes :

Somme classique de deux nombres binaires :

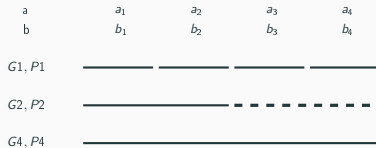
$$\begin{array}{ccccccc} & & c_1 & c_2 & & c_{s-1} & \\ & & \curvearrowright & \curvearrowright & & \curvearrowright & \\ & a_1 & a_2 & \cdots & a_s & & \\ + & b_1 & b_2 & \cdots & b_s & & \\ \hline & r_1 & r_2 & \cdots & r_s & & \end{array}$$

- $a_1$  et  $b_1$  présents dans la formule booléenne de  $r_s$
- profondeur de NAND en  $\mathcal{O}(s)$

## Sommer deux listes : carry lookahead adder



# Sommer deux listes : carry lookahead adder



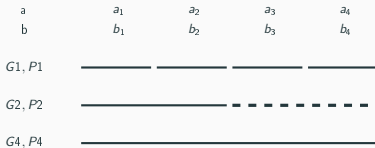
- $G$  pour génération

$$\begin{array}{r}
 a_3 \quad a_4 \\
 + \quad b_3 \quad b_4 \\
 \hline
 x \quad x \quad 1
 \end{array}$$

- $P$  pour propagation

$$\begin{array}{r}
 a_3 \quad a_4 \\
 + \quad b_3 \quad b_4 \\
 + \quad 1 \\
 \hline
 x \quad x \quad 1
 \end{array}$$

# Sommer deux listes : carry lookahead adder



- $G$  pour génération

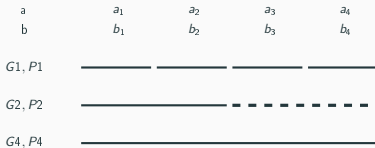
$$(G1)_i = a_i \wedge b_i \quad (P1)_i = a_i \vee b_i$$

$$\begin{array}{r}
 a_3 \quad a_4 \\
 + \quad b_3 \quad b_4 \\
 \hline
 x \quad x \quad 1
 \end{array}$$

- $P$  pour propagation

$$\begin{array}{r}
 a_3 \quad a_4 \\
 + \quad b_3 \quad b_4 \\
 + \quad 1 \\
 \hline
 x \quad x \quad 1
 \end{array}$$

# Sommer deux listes : carry lookahead adder



- $G$  pour génération

$$(G1)_i = a_i \wedge b_i \quad (P1)_i = a_i \vee b_i$$

$$\begin{array}{r} a_3 \quad a_4 \\ + \quad b_3 \quad b_4 \\ \hline x \quad x \quad 1 \end{array}$$

$$G2^i, P2^i \quad \begin{array}{cc} \underline{\quad 1 \quad} & \underline{\quad 2 \quad} \end{array}$$

$$G2^{i+1}, P2^{i+1} \quad \underline{\quad 1 \quad}$$

- $P$  pour propagation

$$\begin{array}{r} a_3 \quad a_4 \\ + \quad b_3 \quad b_4 \\ + \quad 1 \\ \hline x \quad x \quad 1 \end{array}$$

$$(G2^{i+1})_1 = (G2^i)_2 \vee ((G2^i)_1 \wedge (P2^i)_2)$$

$$(P2^{i+1})_1 = (G2^i)_1 \wedge (P2^i)_2$$

# Sommer deux listes : carry lookahead adder

a	0	0	1	1	1	1	0	0
b	0	1	0	1	0	1	0	1
$G1, P1$	<u>0 0</u>	<u>0 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 0</u>	<u>0 1</u>
$G2, P2$	<u>0 0</u>		<u>1 1</u>		<u>1 1</u>		<u>0 0</u>	
$G4, P4$		<u>1 0</u>				<u>0 0</u>		
$G8, P8$			<u>0 0</u>					
carry	?	?	?	?	?	?	?	?

# Sommer deux listes : carry lookahead adder

a	0	0	1	1	1	1	0	0
b	0	1	0	1	0	1	0	1
$G1, P1$	<u>0 0</u>	<u>0 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 0</u>	<u>0 1</u>
$G2, P2$	<u>0 0</u>		<u>1 1</u>		<u>1 1</u>		<u>0 0</u>	
$G4, P4$		<u>1 0</u>				<u>0 0</u>		
$G8, P8$				<u>0 0</u>				
carry	0	0	?	1	?	?	?	0



# Sommer deux listes : carry lookahead adder

a	0	0	1	1	1	1	0	0
b	0	1	0	1	0	1	0	1
$G1, P1$	<u>0 0</u>	<u>0 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 0</u>	<u>0 1</u>
$G2, P2$	<u>0 0</u>		<u>1 1</u>		<u>1 1</u>		<u>0 0</u>	
$G4, P4$		<u>1 0</u>				<u>0 0</u>		
$G8, P8$				<u>0 0</u>				
carry	0	0	?	1	?	1	?	0

$$c_6 = G2_3 \vee (c_4 \wedge P2_3) = 1 \vee (1 \wedge 1) = 1$$

## sommer deux listes : carry lookahead adder

a	0	0	1	1	1	1	0	0
b	0	1	0	1	0	1	0	1
$G1, P1$	<u>0 0</u>	<u>0 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 1</u>	<u>1 1</u>	<u>0 0</u>	<u>0 1</u>
$G2, P2$	<u>0 0</u>		<u>1 1</u>		<u>1 1</u>		<u>0 0</u>	
$G4, P4$			<u>1 0</u>			<u>0 0</u>		
$G8, P8$				<u>0 0</u>				
carry	0	0	?	1	?	1	?	0

$$c_6 = G2_3 \vee (c_4 \wedge P2_3) = 1 \vee (1 \wedge 1) = 1$$

$$c_7 = G1_7 \vee (c_6 \wedge P1_7) = 0 \vee (1 \wedge 0) = 0$$

# Effectuer un bootstrapping

**Carry lookahead adder :**

- profondeur de NAND en  $\mathcal{O}(\log(s))$

**Circuit booléen de déchiffrement  $\Pi$  :**

- Profondeur de NAND en  $\mathcal{O}(\log(\log(q)) + \log(n))$

**Conclusion :** Certains choix de paramètres permettent d'effectuer un bootstrapping en garantissant que le cryptosystème est IND-CPA.

# Effectuer un bootstrapping

**Carry lookahead adder :**

- profondeur de NAND en  $\mathcal{O}(\log(s))$

**Circuit booléen de déchiffrement  $\Pi$  :**

- Profondeur de NAND en  $\mathcal{O}(\log(\log(q)) + \log(n))$

**Conclusion :** Certains choix de paramètres permettent d'effectuer un bootstrapping en garantissant que le cryptosystème est IND-CPA.

**Taille des données permettant du bootstrapping :**

paramètre de sécurité	taille de pk	taille de sk	taille d'un chiffré
$\lambda = 8$	3 Ko	113 Mo	135 Go
$\lambda = 16$	12 Ko	2 Go	6 To
$\lambda = 32$	45 Ko	32 Go	176 To

# Un oeil sur le monde réel

---

# The Gate Bootstrapping API

- librairie open source pour du C/C++
- s'appuie sur des travaux de I. Chillotti, N. Gama, M. Georgieva et M. Izabachène
- utilise une version modifiée du cryptosystème GSW, avec une variante de LWE nommée TFHE.

**Performances** : pour un ordinateur 64-bit simple coeur (i7-4930MX) cadencé à 3.00GHz

- le bootstrapping se fait en un temps moyen de 52ms
- la clé de bootstrapping fait environ 24 Mo.

# The Gate Bootstrapping API

- Alice génère des clés, chiffre deux nombres de 16 bits et les inscrit dans un fichier ;
- le cloud récupère les données, applique homomorphiquement la fonction minimum aux deux nombres et inscrit le résultat dans un fichier ;
- Alice récupère et déchiffre le résultat.

**Performances** : pour un paramètre de sécurité  $\lambda = 110$  et sur un ordinateur 64-bit quadri-coeur (i5-7200U CPU) cadencé à 2.50GHz

- temps de 2.10s.

Données	Taille
Clé secrète	79 Mo
Clé publique et clé de bootstrapping	79 Mo
Chiffrement d'un bit	2 Ko
Chiffrement des deux nombres	64 Ko

# Ce dont nous n'avons pas parlé

Concernant la définition de GSW :

- Flatten, dont nous avons caché la définition sous le tapis ;
- deux autres algorithmes de déchiffrement.

Concernant les choix de paramètres et la sécurité :

- hypothèses de sécurité sur DLWE, définitions équivalentes, lien LWE/DLWE ;
- les gaussiennes discrètes ;
- la librairie sagemath `lwe_estimator` pour estimer la sécurité de paramètres LWE ;
- notion de leveled FHE et choix de paramètres pour un leveled FHE.

Concernant le bootstrapping :

- optimisations supplémentaires pour sommer plusieurs nombres binaires ;
- contraintes de sécurité supplémentaires (sécurité circulaire).