

Les FHE, c'est facheux.

Lucas Roux & Eric Sageloli

15 février 2019

Un bref historique

- 2009 : Un premier plan par Craig 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 ;

GSW, premier essai :

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

Clé publique : pk

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

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

Déchiffrement : Evident

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** : $C_1 \times C_2$

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

- **NAND** : $C_1 * C_2 - \text{Id}$
- **Produit par scalaire** : λC_1

GSW, second essai :

Clé secrète : un vecteur $sk \in \mathbb{Z}_q^N$

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

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

Retour sur les opérations homomorphes :

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

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

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

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

Problème :

Les coefficients de $C_1 \vec{e}_2$ peuvent être gros

GSW, version finale :

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

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

$$\langle \text{Flatten}(C), \vec{s}k \rangle = \langle C, \vec{s}k \rangle \quad \text{pour un secret } \vec{s}k \text{ bien choisi}$$

Clé secrète : un vecteur $s \in \mathbb{Z}_q^N$ bien choisi

Clé publique : pk

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

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

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

$$\text{Decrypt}(\vec{s}k, C) = \left\lfloor \frac{(C \vec{s})_i}{v_i} \right\rfloor$$

Opérations homomorphes : on applique Flatten aux précédentes opérations homomorphes.

- **Somme** : $\text{Flatten}(C_1 + C_2)$

Le problème DLWE

Paramètres : $n, q \in \mathbb{N}$, une distribution χ sur \mathbb{Z}_q

Le problème DLWE(n, q, χ) consiste à distinguer deux distributions :

- La distribution qui crée uniformément $(\vec{a}, b) \in \mathbb{Z}_q^{n+1}$;
- La distribution qui utilise un secret $\vec{s} \in \mathbb{Z}_q^n$ tiré uniformément, et crée des vecteurs (\vec{a}, b) où

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

e_i étant échantillonné par χ .

à partir d'un ensemble d'échantillons.

FHE avec bootstrapping

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

$$C = D + \text{error} \xrightarrow{\text{Decrypt}(\text{sk}, C)} \mu \xrightarrow{\text{Encrypt}(\text{sk}, \mu)} C^{\text{new}} = D + \text{error}$$

FHE avec bootstrapping

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

$$C = D + \text{error} \xrightarrow{\text{Decrypt}(\text{sk}, C)} \mu \xrightarrow{\text{Encrypt}(\text{sk}, \mu)} C^{\text{new}} = D + \text{error}$$

Soit Π le circuit booléen tel que

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

FHE avec bootstrapping

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

$$C = D + \text{erreur} \xrightarrow{\text{Decrypt}(\text{sk}, C)} \mu \xrightarrow{\text{Encrypt}(\text{sk}, \mu)} C^{\text{new}} = D + \text{error}$$

Soit Π le circuit booléen tel que

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

Alors :

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

- Si Π contient assez peu de NAND, on peut avoir un FHE.

Découpage de Decrypt

L'algorithme de déchiffrement est le suivant :

1. trouver $1 \leq i \leq l$ tel que $q/4 \leq 2^i < q/2$
 2. calculer $a = C_i \cdot \vec{v}$
 3. retourner $\lfloor \frac{a}{v_i} \rfloor$
- On peut ramener le calcul du produit scalaire à une somme de nombres binaire ;
 - Diviser par une puissance de 2 est un shift à gauche sur l'écriture binaire ;
 - Calculer la valeur absolue implique essentiellement de faire un complément à 2.

Voyons comment sommer deux nombres binaires.

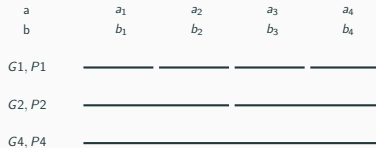
sommer deux listes :

Somme classique entre deux nombres binaires :

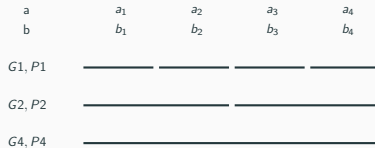
$$\begin{array}{ccccccc} & & c_1 & c_2 & & c_{s-1} & \\ & & \curvearrowright & \curvearrowright & & \curvearrowright & \\ & a_1 & a_2 & \cdots & a_s & & \\ + & b_1 & b_2 & \cdots & b_s & & \\ \hline & r_1 & r_2 & \cdots & r_s & & \end{array}$$

- a_1 et b_1 présents dans la formule booléenne de r_s .
- profondeur de NAND en $O(s)$

sommer deux listes : carry lookahead adder



sommer deux listes : carry lookahead adder



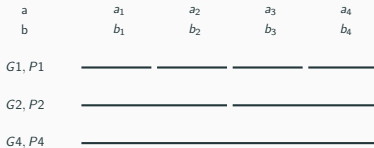
- G pour génération

$$\begin{array}{cccc} & a_5 & a_6 & a_7 & a_8 \\ + & b_5 & b_6 & b_7 & b_8 \\ \hline & X & X & X & X & 1 \end{array}$$

- P pour propagation

$$\begin{array}{cccc} & a_5 & a_6 & a_7 & a_8 \\ + & b_5 & b_6 & b_7 & b_8 \\ + & 1 & & & \\ \hline & X & X & X & X & 1 \end{array}$$

sommer deux listes : carry lookahead adder



- G pour génération

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

$$\begin{array}{cccc} & a_5 & a_6 & a_7 & a_8 \\ + & b_5 & b_6 & b_7 & b_8 \\ \hline & X & X & X & X & 1 \end{array}$$

- P pour propagation

$$\begin{array}{cccc} & a_5 & a_6 & a_7 & a_8 \\ + & b_5 & b_6 & b_7 & b_8 \\ + & 1 & & & \\ \hline & X & X & X & X & 1 \end{array}$$

sommer deux listes : carry lookahead adder

a	a_1	a_2	a_3	a_4
b	b_1	b_2	b_3	b_4
$G1, P1$	_____	_____	_____	_____
$G2, P2$	_____	_____	_____	_____
$G4, P4$	_____	_____	_____	_____

- G pour génération

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

$$\begin{array}{cccc}
 & a_5 & a_6 & a_7 & a_8 \\
 + & b_5 & b_6 & b_7 & b_8 \\
 \hline
 & X & X & X & X & 1
 \end{array}$$

$$G2^i, P2^i \quad \text{_____} \quad 1 \quad \text{_____}$$

$$G2^{i+1}, P2^{i+1} \quad \text{_____} \quad 1 \quad \text{_____} \quad 2 \quad \text{_____}$$

- P pour propagation

$$\begin{array}{cccc}
 & a_5 & a_6 & a_7 & a_8 \\
 + & b_5 & b_6 & b_7 & b_8 \\
 + & 1 & & & \\
 \hline
 & X & X & X & X & 1
 \end{array}$$

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

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

sommer deux listes : carry lookahead added

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	?	?	?	?	?	?	?	?

Les variables de générations de blocs commençants par 0 calculent des retenues.

sommer deux listes : carry lookahead added

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

Les variables de générations de blocs commençants par 0 calculent des retenues.

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

sommer deux listes : carry lookahead added

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

Les variables de générations de blocs commençants par 0 calculent des retenues.

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

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

Effectuer un bootstrapping

Profondeur de NAND totale en :

$$\mathcal{O}(\log(\log(q)) + \log(n))$$

avec n un paramètre du système.

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

Effectuer un bootstrapping

Profondeur de NAND totale en :

$$\mathcal{O}(\log(\log(q)) + \log(n))$$

avec n un paramètre du système.

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

sécurité	taille du secret	taille de la clé	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

The Gate Bootstrapping API

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

Performances : pour un ordinateur 64-bit simple coeur (i7-4930MX) cadencé à 3.00GHz, le bootstrapping se fait en un temps moyen de 52ms et 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, on obtient un temps de 2.10s.

On a alors des données de taille :

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

- Flatten, dont nous avons caché la définition sous le tapis ;
- Deux autres algorithmes de déchiffrement ;
- Les gaussiennes discrètes ;
- Choix des paramètres pour un leveled FHE ;
- Contraintes de sécurité supplémentaires liées au bootstrapping ;