International Journal of Innovation and Scientific Research
ISSN 2351-8014 Vol. 54 No. 2 May. 2021, pp. 125-137

http://www.ijisr.issr-journals.org/

Implémentation d'opérations arithmétiques homomorphiques sur les entiers chiffrés

[Implementation of homomorphic arithmetic operations on encrypted integers]

Paulin Boale Bomolo[1], Eugene Mbuyi Mukendi[2], and Simon Ntumba Badibanga[3]

[1]Chef des Travaux, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

[2]Professeur Ordinaire, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

[3]Professeur, Département des Mathématiques et Informatique, Faculté des Sciences, Université de Kinshasa, RD Congo

ABSTRACT: The full homomorphic encryption allows to the cloud for providing computations on encrypted data without decrypt them. Since the breakthrough of Gentry, this approach of cryptographic continue to tempt the scientific research and industries of New technologies of Information and Telecommunications. More recently, Léo Ducas and Daniele Micciancio were published a library called FHEW which allow to perform bootstrapping in less than a second.
In this paper, we propose an implementation of basics arithmetic operations on unsigned integer. The improvement on circuit allows us to implement homomorphic computations on unsigned integers of 8, 16 and 32 bytes. The correspondents circuits of addition, subtraction, multiplication and division were rewritten in equivalents circuits using the universal gate NAND. The obtained result is hopeful for using homomorphic encryption in wide application which need privacy application in real life.

KEYWORDS: Homomorphic encryption, bootstrapping, Learning With Errors, logic gate, circuit.

RESUME: Le chiffrement homomorphe complet permet au cloud de fournir des services de traitement sur les données chiffrées aux clients sans au préalable les déchiffrer. Depuis la percée de Gentry en 2009, cet aspect de la cryptographie continue à attirer l'attention de la communauté scientifique et de l'industrie de nouvelles technologies de l'information et de la télécommunication. Plus récemment, Léo Ducas et Daniele Micciancio ont publié une librairie FHEW qui permet d'effectuer un bootstrapping sur une opération logique NONET sur les bits en moins d'une seconde.
Dans cet article, une implémentation d'opérations arithmétique homomorphique de base sur les entiers chiffrés est proposée. Les optimisations sur les circuits arithmétiques ont permis d'implémenter les opérations homomorphiques sur 8, 16 et 32 bits d'addition, de soustraction, de multiplication et de division. Les circuits binaires représentant ces opérations ont réécrites dans des circuits équivalents en se basant sur l'opérateur universel NONET. Les résultats obtenus suscitent l'espoir pour le futur dans un environnement performant.

MOTS-CLEFS: Chiffrement homomorphe, bootstrapping, apprentissage avec erreurs, porte logique, circuit arithmétique.

1 INTRODUCTION

Un schéma de chiffrement homomorphique complet est un schéma de chiffrement qui permet d'effectuer des opérations arbitraires sur les données chiffrées. Cette définition a été énoncée pour la première fois par Rivest et les autres dans [19] et est connu pour avoir des multiples applications en sécurité surtout dans les nuages. Elle est restée un concept jusqu'à la percée

de Gentry [10, 11]. Et depuis, plusieurs schémas ont proposé vers la praticabilité du chiffrement homomorphique [8, 3, 7, 2, 12, 1, 13, 6,21].

Parmi eux, le schéma BGV [BGV122] est l'un de schéma le plus efficace, de plus il possède un potentiel énorme. Il est basé sur les hypothèses de sécurité du Learning With Errors [18] (LWE) et Ring Learning With Errors (RLWE) [17] et supporte les opérations SIMD sous certains paramètres [21]. Aussi, il a été implémenté par Halevi et Shoup [12] dans une libraire dénommée Helib [20]. Elle exécute pendant six minutes les données chiffrées de manière homomorphique en batch. En 2015, une nouvelle librairie qui permet d'effectuer des opérations sur un bit, et rafraichit le résultat en moins d'une seconde sur un ordinateur personnel a été proposée par [9]. Dénommée FHEW, elle se base sur le circuit NONET.

Dans cet article, la librairie FHEW est utilisée pour implémenter des opérations d'addition, de soustraction, de multiplication et de division avec reste. Notre implémentation est capable d'évaluer homomorphiquement des entiers de 8, 16 et 32 bits par ces opérations arithmétiques de base.

Deux travaux similaires dans ce domaine ont été publié par [4] et [5] en se basant sur la librairie Helib en traitant respectivement les entiers de 64 bits, 16 bits et 4 bits.

2 NOTIONS PRÉLIMINAIRES

Cette section présente les notions préliminaires utiles sur le chiffrement homomorphique, le schéma de Ducas-Micciano et la librairie FHEW.

2.1 SCHÉMA DE CHIFFREMENT HOMOMORPHIQUE COMPLET

2.1.1 SCHÉMA DE CHIFFREMENT À CLÉ PUBLIQUE

Un schéma de chiffrement à clé publique consiste en trois algorithmes qui sont $Keygen, Enc\ et\ Dec. Keygen$ est un algorithme qui prend en entrée un paramètre de sécurité λ, et donne en sortie une clé privée sk et une clé publique pk; pk décrit l'espace de messages en clair P et l'espace des cryptogrammes C. Enc est un algorithme qui prend en entrée la clé publique pk et un message en clair $\in P$, et donne en sortie un cryptogramme $c \in C$. Dec est un algorithme qui prend en entrée la clé privée sk et le cryptogramme c, et donne en sortie le message en clair b.

La complexité calculatoire de ces algorithmes doit être polynomiale dans le paramètre de sécurité. L'exactitude d'un chiffrement est définie comme suit: $si\ (sk, pk) \leftarrow Keygen, b\ \in P\ et\ c\ \leftarrow Enc(pk, b), alors\ Dec\ (sk, c) \rightarrow b$.

2.1.2 SCHÉMA DE CHIFFREMENT HOMOMORPHIQUE

Un schéma de chiffrement homomorphique possède en plus un algorithme efficace Eval. $Eval$ prend en entrée la clé publique pk, une fonction f et un vecteur des cryptogrammes $c = (c_1, \dots\dots, c_n)$ ou $c_i \leftarrow Enc(pk, b_i)$ pour $b_i \in P$, donne en sortie un cryptogramme $c \in C$.

L'exactitude pour le chiffrement homomorphique est définie comme-suit $si\ c \leftarrow Eval(pk, f, c)$, alors $Dec(sk, c) \leftarrow f(b_1, \dots\dots\dots\dots, b_n)$.

Dans la majorité de schéma de chiffrement homomorphique, la fonction f à évaluer est représentée sous la forme d'un circuit logique comportant des portes $OU\ Exclusif$ et ET qui correspondent respectivement à l'addition et la multiplication binaire. De plus, ce chiffrement évalue les circuits possédant une profondeur limitée. L'augmentation de ce paramètre entraine une croissance du bruit. Atteignant un seuil, ledit bruit rend le déchiffrement invalide.

2.1.3 SCHÉMA DE CHIFFREMENT HOMOMORPHIQUE COMPLET

Un schéma de chiffrement homomorphique complet est un schéma de chiffrement homomorphe qui est capable d'évaluer des circuits avec des profondeurs larges que celles de son circuit de déchiffrement. Cette condition permet d'effectuer le bootstrapping sur les données chiffrées dans le but de délimiter la profondeur de circuit.

2.2 SCHÉMA DE DUCAS-MICCIANO

Le schéma [9] est un schéma de chiffrement symétrique basé sur le problème du LWE. Les opérations homomorphiques sont des additions sur les vecteurs chiffrés, qui sont plus simple et plus efficaces que le produit de tenseurs et des opérations sur les matrices utilisées dans d'autres schémas sur AAE.

2.2.1 LE PROBLÈME D'APPRENTISSAGE AVEC ERREURS (LEARNING WITH ERRORS)

L'Apprentissage Avec Erreurs (AAE) est un problème qui a été prouvé difficile sur les réseaux, qui a été proposé par Regev [18].

Etant donné une dimension n, un modulus q, et une distribution aléatoire χ sur les entiers, le problème du AAE est initialisé par (n, q, χ):

- Tirer $a \leftarrow \mathbb{Z}_q^n$, $s \leftarrow \mathbb{Z}_q^n$, et $e \leftarrow \chi$ une erreur aléatoire;
- Ensuite, produire en sortie une instance du $AAE(a, b) = (a, (as + e)\ mod\ q) \in \mathbb{Z}_q^{n+1}$.

La distribution aléatoire formée par les différentes instances du AAE est calculatoirement difficile à distinguer d'une distribution uniforme sous $\mathbb{Z}_q^{n+1}$. Il est aussi difficile de trouver s de toutes ses instances.

2.2.2 SCHÉMA DE CHIFFREMENT HOMOMORPHIQUE DE DUCAS MICCIANCIO

2.2.2.1 CHOIX DES PARAMÈTRES

- Etant donné un paramètre de sécurité λ;
- Soit $n = n(\lambda)$ la dimension d'un vecteur chiffré,
- et soient $t(t \geq 2)$ et $q = q(\lambda)$ respectivement le modulus des messages en clair et des messages chiffrés;

La fonction aléatoire $\chi_r(.)$: $\mathbb{R} \rightarrow \mathbb{Z}$, ajoutant un bruit aléatoire e à la variable en paramètre ie $\chi_r(.) \equiv \chi_r(as) = as + e$ où $e \leftarrow \chi(\sigma, B)$: une distribution sous-gaussien de paramètre $\sigma\ et\ de\ limite\ B$, tel que $|\chi_r(x) - x| < {}^{q}/_{2t}$ pour un déchiffrement valide.

2.2.2.2 GÉNÉRATION DE LA CLÉ DE CHIFFREMENT

Etant donné un paramètre de sécurité λ, tirer de façon aléatoire une clé secrète s dans $\mathbb{Z}_q^n$, une distribution uniforme.

2.2.2.3 CHIFFREMENT

Etant donné un message $m \in \mathbb{Z}_t$ et une clé s, évaluer l'expression ci-dessous pour trouver le message chiffré de m:

$$c = AAE_s^{{}^{t}/_{q}}(m) = (a, b) = \left(a, \chi_r\left(as + {}^{mq}/_{t}\right)\ mod\ q\right) = \left(a, as + {}^{mq}/_{t} + e\right)$$

où $a \leftarrow \mathbb{Z}_q^n$: a est tiré aléatoirement d'une distribution uniforme sous $\mathbb{Z}_q^n$ et $|\chi_r(x) - x| < {}^{q}/_{4}$

2.2.2.4 DÉCHIFFREMENT

Etant donné un message chiffré $c = (a, b)$ et une clé secrète s, le déchiffrement est évalué comme-suit:

$$m' = \left\lfloor 2\ {}^{(b - as)}/_{q} \right\rfloor \quad (3)$$

Avec l'expression $({}^{2}/_{q})|e| < \frac{1}{2}$ doit être satisfait pour un déchiffrement valide, on a $|e| < {}^{q}/_{4}$.

2.2.2.5 LA PORTE HOMOMORPHIQUE NONET

Etant donné des cryptogrammes $c_i \in AAE_s^{{}^{4}/_{q}}\left(m_i, {}^{q}/_{16}\right)$ où c_0, c_1 chiffrant respectivement $m_0, m_1 \in \{0, 1\} \subset \{1, 2, 3, 4\}$. L'opération logique $NONET$ est évaluée homomorphiquement par l'expression ci-dessous:

$$HomNAND\big((a_0,b_0),(a_1,b_1)\big) = \left(-a_0 - a_1, \frac{5}{8}q - b_0 - b_1\right) \text{(4)}$$

Où le message en clair correspondant est $m = NAND(m_0, m_1) = 1 - m_0 m_1$. Le bruit contenu dans ce message est exprimé de la manière suivante:

$$err(a,b) = b - as - (1 - m_0 m_1)\frac{q}{2} = \frac{q}{4}\left(\frac{1}{2} - (m_0 - m_1)^2\right) - (e_0 + e_1) = \pm\frac{q}{8} - (e_0 + e_1)$$

La limite supérieure du bruit e dans le message chiffré est donnée par l'expression:

$$B' = \left|\frac{q}{8}\right| + |e_0| + |e_1| = \frac{q}{8} + \frac{q}{16} + \frac{q}{16} = \frac{q}{4}$$

Il ressort de $|e| < {}^{q}/_{2t}$ que le bruit e dans (4) doit satisfaire $|e| < \frac{q}{4}$ pour un déchiffrement valide. Ainsi le message chiffré (a, b) de (3) est le chiffré correspondant au message en clair $1 - m_0 m_1$ sous la clé secrète s.

2.2.2.6 LE BOOTSTRAPPING DANS [9]

De la théorie de Gentry, un message chiffré bruité peut-être rafraichi par l'application du bootstrapping pour réduire le bruit. Dans le schéma d'avant, l'algorithme pour le déchiffrement de message chiffré et l'algorithme utilisé pour le bootstrapping était la même. Ceci est une procédure qui est un déchiffrement homomorphique avec comme paramètre d'entrée un message chiffré et une clé de déchiffrement chiffré. Mais depuis [AP14], plusieurs sont maintenant divisées en deux schémas que sont interne et externe. $Enc\ ()$ $(E\ ())$ est appelé schéma interne (externe). Le message chiffré dans le schéma interne (externe) est appelé message chiffré interne (externe).

L'exécution homomorphiquement du déchiffrement interne sur le message interne permet réduire le bruit dans le chiffré et obtenir plus d'efficace dans la procédure de bootstrapping.

Dans le [9], une évaluation homomorphique est effectuée sur le message chiffré à travers l'expression suivante $E(b - as)$. Le calcul du produit du scalaire sur les vecteurs de $\mathbb{Z}_q^n$ étant très coûteux, [9] a implémenté un schéma d'accumulateur homomorphique basé sur la version anneau du [13]. Ledit schéma permet de calculer l'expression $E(b - as) = E\big(b - \sum_{i=1}^{i=n} a_i . s_i\big)$ en utilisant la propriété additive homomorphique dudit schéma. Ce schéma est décrit de la manière suivante:

Pour chiffrer un message m, supposons une clé secrète $z \in \mathcal{R}$ avec $\mathcal{R} = \frac{\mathbb{Z}[X]}{X^N+1}$, tirer un vecteur aléatoire $a \in \mathcal{R}^{2d_g}$ et une erreur de faible magnitude $e \in \mathcal{R}^{2d_g}$, et évaluer l'expression ci-dessous:

$$E(m) = [a, a.z + e] + uY^m G \in\in \mathcal{R}^{2d_g \times 2}$$

Où $G = \left(I, \mathcal{B}_g I, \dots\dots\dots, \mathcal{B}_g^{d_g - 1}\right)$ $\mathcal{R}^{2d_g \times 2}$

- Pour initialiser l'accumulateur avec une entrée $v \in \mathbb{Z}_q$ avec l'algorithme $Init(ACC \longleftarrow v)$, évaluer seulement l'expression ci-dessous:

$$ACC \coloneqq uY^v . G \in \mathcal{R}_Q^{2d_g \times 2}$$

- Pour incrémenter le contenu actuel de l'accumulateur $ACC \in \mathcal{R}^{2d_g \times 2}$ avec $C \in \mathcal{R}^{2d_g \times 2}$ avec l'algorithme $Incr\left(ACC \overset{+}{\leftarrow} C\right)$, premièrement traiter la décomposition de l'expression $u^{-1}ACC = \sum_{i=1}^{d_g} \mathcal{B}_g^{i-1} D_i$ dans la base $\mathcal{B}_g$ (où $D_i \in \mathcal{R}^{2d_g \times 2}$ avec des coefficients dans $\left\{\frac{1-\mathcal{B}_g}{2}, \dots\dots\dots, \frac{1-\mathcal{B}_g}{2}\right\}$, et ensuite mettre à jour l'accumulateur avec l'expression $ACC: = \left[D_1, \dots\dots\dots, D_{d_g}\right] . C$.

- Pour extraire le bit plus significatif, l'algorithme $msbExtract$ utilise la clé de commutation $\Re$ et un vecteur de test $t = -\sum_{i=0}^{q/2-1} \overrightarrow{Y}^i$. L'extraction du bit plus significatif est $t.\overrightarrow{Y^v} = -1\ si\ 0 \le v \le N, et + 1\ si\ N \le v \le 2N$. Où $\Re = \left\{K_{i,j,w}\right\}_{i,j,w} de\ z\ vers\ s: K_{i,j,w} \longleftarrow AAE_s^{q/q}\left(wz_i d_{ks}^j\right)\ et\ ACC\ est\ un\ l - message\ chiffree\ de\ v.$

Le schéma d'accumulateur homomorphique est paramétré par un modulus Q et un élément inversible u de $\mathbb{Z}_Q$ proche de ${}^{Q}/_{8}$. Et il est supposé que $Q = \mathcal{B}_g^{d_g}$ où $d_g \in \mathbb{N}$. Pour plus d'efficacité dans l'implémentation de l'accumulateur

homomorphique, [9] utilise la librairie FFT implémentant la transformée de fourrier discrète rapide surtout pour l'algorithme $Incr$.

Les entrées de l'étape d'extraction sont $ACC \in \mathcal{R}_Q^{2d_g \times 2}$ et un vecteur de test $t = (-1,1, \ldots\ldots\ldots,1)$. Quoique, dans cette procédure, seule la ligne de l'accumulateur est utile. De plus, le traitement de chaque de l'accumulateur est indépendant.

2.3 LA LIBRAIRIE FHEW

FHEW est une librairie mathématique libre qui est distribuée sous licence GNU. Il est basé sur l'article [1]. Son acronyme signifie Fastest Homomorphic Encryption in the West fait plus référence aux séries de fourrier FFTW (Fastest Fourrier transform in the West) qu'une requête de performance.

Elle fournit un schéma de chiffrement symétrique qui permet de chiffrer et de déchiffrer des messages de taille d'un bit, supportant l'évaluation homomorphique des circuits arbitraires sur les messages chiffrés en utilisant une clé d'évaluation publique.

Cette librairie s'exécute dans un environnement linux en se basant sur la librairie FFTW3 disponible à l'adresse suivante http://www.fftw.org/download.html et u compilateur C. Elle est écrite en C, mais le compilateur C++ est utilisé pour supporter certaines syntaxes qui rendent le code lisible. Des tests ont été effectués avec le compilateur g++ du compilateur GCC (GNU Compiler Collection). La version la plus récente sur le lien https://github.com/lducas/FHEW est 2.0-alpha du 30 mai 2017.

2.4 CIRCUIT DES PORTES BOOLÉENNES HOMOMORPHIQUE

La version 2.0-alpha du FHEW implémente cinq portes homomorphiques à savoir, NONET, ET, NONOU, OU et NON. Ces portes effectuent des opérations homomorphiques sur les bits, telle que l'opération homomorphique NONET.

3 LES OPÉRATIONS ARITHMÉTIQUES HOMOMORPHES AVEC LE [9]

3.1 INTRODUCTION

L'espace de messages en clair dans le [9] est $\mathbb{Z}_2$. Les opérations d'addition et de multiplication sont définies dans l'arithmétique binaire respectivement les portes logiques OUExclusif et ET. Ces portes sont la fondation de l'implémentation de circuits de plus en plus complexes tels les additionneurs, les soustracteurs, ….

Cette section présente une implémentation des opérations arithmétiques de base usant de ET et OUExclusif. Le [9] implémente l'opération homomorphique sur la porte logique NONET dont le bootstrapping est évalué en moins d'une seconde. La porte logique NONET est une porte universelle qui permet de représenter toutes les autres portes. Ainsi, les portes ET et OUExclusif s'exprimeront en fonction de la porte NONET respectivement de la manière suivante NONET (NONET (), NONET ()) et NONET (NONET (, NONET ()),NONET (, NONET ())).

Ces opérations arithmétiques seront effectuées sur des entiers respectivement d'une taille de 8, 16 et 32 bits.

3.2 ADDITION

Dans l'arithmétique des entiers, l'addition est une opération de base qui permet d'implémenter les opérations arithmétiques telles que la multiplication, la soustraction et la division.

Dans cet article, l'additionneur complet sera implémenté en utilisant deux algorithmes que sont l'additionneur à propagation de retenue et l'additionneur à retenue anticipée. Mais avant, l'additionneur sera représenté en fonction de la porte logique NONET.

3.2.1 ADDITIONNEUR COMPLET

Un additionneur complet est un additionneur qui comprend des demi-additionneurs et des additionneurs complets. La différence est qu'un additionneur n'accepte pas une retenue tandis que l'additionneur l'accepte. L'implémentation peut varier aussi longtemps que les expressions logiques de différentes implémentations sont équivalentes. Dans [5], pour exemple, les expressions de la somme et de la retenue peuvent écrites de la manière suivante :

$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i)$ (1) où $\oplus, .$ Sont respectivement le symbole de ET et OUExclusif.

$$s_i = a_i \oplus b_i \oplus c_i \quad (2)$$

Où a_i et b_i sont le ième bit de deux sommations, c_i est la ième retenue, et s_i est la ième somme de bits. L'expression (1) de la retenue peut être réduite à une expression équivalente comme-suit:

$$c_{i+1} = a_i . b_i \oplus c_i . (a_i \oplus b_i) = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i \quad (3)$$

L'expression (3) optimisée est trouvée dans [20]. Elle utilise seulement pour chaque bit une porte NONET, et par conséquent un additionneur complet est considéré comme une cellule d'additionneur d'un bit et d'une profondeur multiplicative équivalent à1.

3.2.1.1 L'ALGORITHME DE L'ADDITIONNEUR À PROPAGATION DE RETENUE

L'additionneur à propagation de retenue permet de réaliser l'addition de deux nombres binaires de n bits et d'une retenue optionnelle en entrée (c_{en}), en assurant la propagation de la retenue d'une cellule additionneur complet à l'autre. Le résultat est un nombre de n+1 bits, consistant en un nombre S et d'une retenue de débordement c_{sor}.

Il est construit par la mise en série de $n - 1$ additionneurs complets. Elle propage la retenue d'une de la cellule inférieure vers la cellule supérieure. De ce fait, elle ajoute un bit à la fois de bits moins significatifs vers des bits plus significatifs. La profondeur multiplicative est $L = n - 1$.

La somme et la retenue d'une cellule ne peuvent se produire que lorsqu'une retenue d'entrée est appliquée, cette technique a pour effet de retarder le processus d'addition. Le délai de propagation de chaque retenue de chaque cellule est égal à l'intervalle entre l'application de la retenue d'entrée et la production de la retenue de sortie. La somme de deux nombres dans un additionneur à propagation de retenue est obtenue après un temps cumulatif de délai de propagation.

L'algorithme de l'addition de deux nombres avec l'additionneur a propagation de retenue est décrit de la manière ci-dessous:

ALGORITHME 1:

Entrée: deux entrées entiers chiffrées de n bits a, b

Sortie: la somme s

$c_0 = 0$

$Pour\ i = 0$ à $n - 2$

Faire

$c_{i+1} = (a_i \oplus c_i) . (b_i \oplus c_i) \oplus c_i$

$s_i = a_i \oplus b_i \oplus c_i$

End for

$s_{n-1} = a_{n-1} \oplus b_{n-1} \oplus c_{n-1}$

Retourner s

3.2.1.2 L'ALGORITHME DE L'ADDITIONNEUR À RETENUE ANTICIPÉE

Dans une architecture à propagation de retenue, l'addition dépend de la propagation des retenues à travers des étages de l'additionneur parallèle. Pour réduire le délai de propagation et accélérer le processus d'addition, il est possible d'anticiper la retenue de sortie de chaque étage et de produire, à partir des entrées, la retenue par génération ou par propagation. Cette technique est appelée « anticipation de retenue ».

Une génération de retenue se produit lorsqu'une retenue est générée par l'additionneur complet. Une retenue ne peut avoir lieu que lorsque les deux bits d'entrée sont de 1. La retenue générée est noté G et équivaut à $G = AB$.

Une propagation de retenue est créée lorsqu'une retenue d'entrée est répercutée vers la retenue de sortie. Dans un additionneur complet, la propagation d'une retenue d'entrée peut avoir lieu lorsqu'au moins un des bits vaut 1. La retenue propagée notée P et équivaut à $P = A + B$.

La retenue de sortie d'un additionneur complet peut s'exprimer en tant que retenue propagée P ou en tant que retenue générée G. La retenue de sortie notée C_{sor} vaut 1 si la sortie générée vaut 1 ou si la sortie propagée vaut 1 et que la retenue d'entrée (C_{en}) vaut 1.

En d'autres termes, une retenue de sortie de 1 est générée par l'additionneur complet si $A = 1\ et\ B = 1$ ou par propagation de l'additionneur de la retenue d'entrée $(A = 1\ ou\ B = 1)\ et\ (C_{en} = 1)$. L'expression ci-dessous résume tous les cas: $C_{sor} = G + PC_{en}$.

Illustrons ce concept par une application à un additionneur parallèle de quatre bits. L'étage i produit une retenue de sortie soit par la génération de celui-ci soit par la propagation de la retenue interne vers le report de sortie. Pour chaque étage i, Il génère G_i et propage P_i de la manière suivante:

- La colonne i produit une retenue de sortie si les entrées A_i et B_i sont égales à un binaire: $G_i = A_i\, B_i$;
- La colonne i propage la retenue interne vers le report de sortie si une des entrées est égale à 1: $Pi_i = A_i + B_i$;
- La retenue de sortie de la colonne i est donnée par l'expression suivante:

$$C_i = A_i B_i + (A_i + B_i)\, C_{i-1} = G_i + P_i\, C_{i-1}.$$

L'algorithme de l'additionneur à anticipation de retenue peut-être décrite dans les étapes ci-dessous:

- Etape 1: calculer G_i et P_i pour toutes les colonnes;
- Etape 2: calculer les G et P pour chaque bloc de k −bits;
- Etape 3: le report d'entrée Cen se propage à travers le bloc de k −bits par les fonctions de génération et de propagation des retenues.

Exemple pour un bloc de 4 bits ($P_{3:0}$ et $G_{3:0}$):

$$G_{3:0} = G_3 + P_3\,(G_2 + P_2\,(G_1 + P_1 G_0)$$

$$P_{3:0} = P_3 P_2 P_1 P_0$$

De façon générale,

$$G_{i:j} = G_i + P_i\,(G_{i-1} + P_{i-1}\,(G_{i-2} + P_{i-2} G_j))$$

$$P_{i:j} = P_i P_{i-1} P_{i-2} P_j$$

$$C_i = G_{i:j} + P_{i:j}\, C_{j-1}$$

La complexité de l'algorithme de l'additionneur à retenue anticipée respectivement en temps est $O\,(n\ log\ n)$. L'additionneur à retenue anticipée de n est plus rapide que l'additionneur à propagation de retenue qui a respectivement une complexité en temps et espace de $O\,(n)$.

L'algorithme de l'addition de deux nombres avec l'additionneur à retenue anticipée est décrit de la manière ci-dessous:

ALGORITHME 2:

Entrée: deux entiers chiffrées a et b de n bits

Sortie: un entier chiffré s de n bits

Poser $c_{en} = c_0 = 0$

Pour i de 0 à $n - 2$

Faire

$c_i^g = a_i b_i$

$c_i^p = a_i + b_i$

$c_i^{sor} = c_i^g + c_p^i c_{i-1}$

$s_i = a_i \oplus b_i \oplus c_{i-1}$

Fin faire

Fin pour

$s_{n-1} = c_{n-1}$

Retourner s

3.3 MULTIPLICATION

L'algorithme classique de la multiplication de a et b, deux nombres de k bits, consiste à calculer des produits partiels en multipliant les b_i du multiplieur b par le nombre a tout entier puis additionner ces produits partiels pour obtenir le produit final p. p est un nombre de $2k$ bits. La figure 1 illustre les produits p_{ij} obtenus de la multiplication deux entiers a et b de trois bits:

$$\begin{array}{rcccccc} & & & a_2 & a_1 & a_0 \\ & & & \times\ b_2 & b_1 & b_0 \\ & & & p_{02} & p_{01} & p_{00} \\ & & p_{12} & p_{11} & p_{10} & & \\ + & p_{22} & p_{21} & p_{20} & & & \\ p_5 & p_4 & p_3 & p_2 & p_1 & p_0 \end{array}$$

Fig. 1. Multiplication classique

L'algorithme de l'addition de deux nombres avec l'additionneur à retenue anticipée est décrit de la manière ci-dessous:

ALGORITHME 3: Algorithme Classique de Multiplication.

Entrée: deux entiers chiffrés a et b de n bits

Sortie: un entier chiffré p de $2n$ bits

Pour i de 0 à $2n - 1$

  Faire

    Initialiser $p_i := 0$

  Fin faire

Fin pour

Pour i de 0 à $n - 1$

  Faire

    $R = 0$

    Pour j de 0 à $n - 1$

      Faire

        $(R, S) = p_{i+j} + b_i a_j + R$

        $P_{i+j} = S$

      Fin faire

    Fin pour

    $P_{i+n} = R$

  Fin faire

Fin pour

Retourner p

Cet algorithme multiplie deux nombres a et b de n bits, ajouter ce produit a la retenue précédente et puis ajouter ce dernier résultat au produit partiel p_{ij}. La complexité dudit algorithme est de $O(k^2)$ opérations. Il est asymptotiquement lent mais peut être utilisé dans l'implémentation pour des entiers dont la taille est inférieure à 128 bits.

3.4 SOUSTRACTEUR A PROPAGATION DE RETENUE

Un soustracteur est dérivé de l'expression logique d'un soustracteur à un bit appelé cellule et peut mettre ensemble les cellules pour composer le soustracteur à propagation de retenue dans la même logique que l'additionneur a propagation de retenue. L'expression logique d'un soustracteur à un bit est similaire à celui de l'additionneur complet, elle donne une expression optimisée du bit de différence d_i et le bit d'emprunt c_i:

$$c_{i+1} = (a_i \oplus c_i).(b_i \oplus c_i) \oplus b_i$$

$$d_i = a_i \oplus b_i \oplus c_i$$

En mettant ces cellules ensembles, un soustracteur à propagation de retenue $n - bit$ est obtenu. La profondeur multiplicative dudit soustracteur est égale à celle d'un additionneur à propagation de retenue, il est remarqué que chaque soustracteur possède une unique porte ET pour chaque bit emprunt, et est utilisé pour trouver le bit emprunt suivant.

3.5 DIVISION BINAIRE

La division binaire est l'opération la plus complexe de quatre. Notre implémentation de ladite opération est basée sur le principe étudié plus-haut sur l'additionneur binaire à propagation de retenue couplé d'un circuit SI-ALORS et des décalages à gauche entre deux étages $j\ et\ j\ -\ 1$. Une division combinatoire est implémentée en tenant du compte du fait que pour entiers chiffrés non signés a et b (a est supérieur à b).

Les équations logiques ci-dessous décrivent mieux cette opération:

$$c_{i+1} = (a_i \oplus c_i).(b_i \oplus c_i) \oplus c_i$$

$$d_i = a_i \oplus b_i \oplus c_i$$

L'expression SI-ALORS est équivalente à l'expression reprise ci-dessous:

$$r_i = a_i \oplus d_i$$

Ou r_i représente le quotient et $q_j\ =\ c_{i+1}$ est le quotient de l'étage j. L'algorithme de la division implémentée dans cet article n'étant pas optimal et de ce fait, il ne sera pas affiché dans la présente section.

4 IMPLÉMENTATION

4.1 ADDITIONNEUR À PROPAGATION DE RETENUE

La fonction HomHalfAdder implémente la somme de deux bits chiffrées. Elle retourne la somme chiffrée res de deux bits chiffrés a et b en utilisant les paramètres en entrée que sont deux bits chiffrés, une retenue initiale chiffré c et une clé d'évaluation homomorphique EK.

```
void HomHalfAdder (LWE:: CipherText* res, const EvalKey& EK, const LWE:: CipherText& a, const LWE:: CipherText& b, const LWE:: CipherText& c) {
    LWE:: CipherText s1;
    HomXOR (&s1, EK, a, b);
    HomXOR (res, EK, s1, c);
}
```

La fonction HomCarry implémente la retenue chiffrée de deux bits chiffrés a et b. elle retourne la retenue chiffrée res en utilisant les paramètres en entrée que sont deux bits chiffrés a et b, une retenue chiffrée initiale c et une clé d'évaluation EK.

```
void HomCarry (LWE:: CipherText* res, const EvalKey& EK, const LWE:: CipherText& a, const LWE:: CipherText& b, const LWE:: CipherText& c) {
    LWE:: CipherText c1, c2, c3;
    HomXOR (&c1, EK, a, c);
    HomXOR (&c2, EK, b, c);
    HomAND (&c3, EK, c1, c2);
    HomXOR (res, EK, c3, c);
}
```

Cette séquence d'instruction montre comment s'effectue l'addition de deux nombres chiffrés a et b de $n-2$ bits dont la somme chiffré s est représentée sur $n-1$ bits.

```
for (int i = 0; i <= N - 2; i++) {

FHEW:: HomHalfAdder (&cx3 [i], EK, cx1 [i],cx2 [i], ccarry [i]);
FHEW:: HomCarry (&ccarry [i+1], EK, cx1 [i],cx2 [i], ccarry [i]);

}

FHEW:: HomHalfAdder (&cx3 [N - 1], EK, cx1 [N - 2],cx2 [N - 2], ccarry [N - 2]);
```

4.2 SOUSTRACTEUR À PROPAGATION DE RETENUE

La fonction HomHalfSubstractor implémente la soustraction de deux bits chiffrées. Elle retourne la différence chiffré res de deux bits chiffrés a et b en utilisant les paramètres en entrée que sont deux bits chiffrés, un emprunt initial chiffré c et une clé d'évaluation homomorphique EK.

```
void HomHalfSubstractor (LWE:: CipherText* res, const EvalKey& EK, const LWE:: CipherText& a, const LWE:: CipherText& b, const LWE:: CipherText& c) {

LWE:: CipherText s1;
HomXOR (&s1, EK, a, b);
HomXOR (res, EK, s1, c);

}
```

La fonction HomBorrow implémente l'emprunt chiffré de deux bits chiffrés a et b. elle retourne l'emprunt chiffré res en utilisant les paramètres en entrée que sont deux bits chiffrés a et b, un emprunt chiffrée initiale c et une clé d'évaluation EK.

```
void HomBorrow (LWE:: CipherText* res, const EvalKey& EK, const LWE:: CipherText& a, const LWE:: CipherText& b, const LWE:: CipherText& c) {

LWE:: CipherText c1, c2, c3;
HomXOR (&c1, EK, a, c);
HomXOR (&c2, EK, b, c);
HomAND (&c3, EK, c1, c2);
HomXOR (res, EK, c3, b);

}
```

Cette séquence d'instruction montre comment s'effectue la soustraction de deux nombres chiffrés a et b de $n-2$ bits dont la différence chiffrée s est représentée sur $n-1$ bits.

```
for (int i = 0; i <= N - 2; i++) {

FHEW:: HomHalfSubstractor (&cx3 [i], EK, cx1 [i],cx2 [i], ccarry [i]);
FHEW:: HomBorrow (&ccarry [i+1], EK, cx1 [i],cx2 [i], ccarry [i]);

}

FHEW:: HomHalfAdder (&cx3 [N - 1], EK, cx1 [N - 2],cx2 [N - 2], ccarry [N - 2]);
```

5 RÉSULTAT EXPÉRIMENTAL

Ce point concerne les résultats expérimentaux de notre implémentation décrit au-dessus.

5.1 INITIALISATION DES PARAMÈTRES

Pour réaliser cette expérience, les paramètres par défaut fournis avec librairie ont été utilisés sans aucune modification. La librairie fournit deux fichiers FHEW.h et FHEW.cpp qui contiennent les définitions et les implémentations nécessaires à l'homomorphisme sur un bit en utilisant l'opération logique NAND.

Des fonctions utiles à notre expérience ont été ajoutées dans la librairie dans lesdits fichiers. Les entiers traités dans cette expérience ont une taille en bits respectivement de 8, 16 et 32 bits. Des exemples d'implémentation ont été repris à la section 4.

5.2 PERFORMANCE ET INTERPRÉTATION

Les implémentations ont été testées sur deux ordinateur portable possédant respectivement un processeur **AMD E1-2100 APU with Radeon ™ HD Graphics 1000 Mhz (1)** et un processeur **Intel® Core™ i5-3210 CPU @ 2.50 Ghz (2)** et les deux avec une mémoire volatile de 4 Giga-octets.

Dans le tableau 1, chaque colonne représente les opérations d'initialisation d'un schéma de chiffrement. La durée que prennent ces opérations sur notre ordinateur de test est reprise sur chaque ligne. Les opérations de génération de la clé secrète, de chiffrement et de chiffrement coûtent moins en termes de temps d'exécution et de stockage par rapport à l'opération de génération de la clé d'évaluation. D'une architecture à l'autre, avec le même espace de stockage, son temps de génération coûte moins de 382%.

Tableau 1. Performance des opérations de paramétrage du schéma

	Keygen	Evaluation Key	Chiffrement	Déchiffrement
Durée (ms)	**0.428**	**73 119.8 (1)**	**2.752**	**0.123**
		15 158.9 (2)		

Dans le tableau 2, chaque colonne représente les opérations logiques homomorphiques. Chaque ligne représente la durée d'exécution de ces opérations deux entiers respectivement sur 8, 16 et 32 bits. L'opération logique du OU exclusif est plus coûteuse que tous les autres. L'addition sur un bit sans report est plus coûteux (81 secondes) sur le processeur 1 que la multiplication (20 secondes) et sur le processeur 2 (4 secondes) et (15 secondes).

Tableau 2. Performance des opérations logique unaires et binaires représentées de manière équivalente avec l'opération NONET

	NAND	AND	XOR
	AMD E1-2100 APU with Radeon ™ HD Graphics 1000 Mhz		
8	**5 102**	**5 084**	**20 615**
16	**10 344**	**10 135**	**40 489**
32	**20 777**	**20 591**	**81 953**
	Intel® Core™ i5-3210 CPU @ 2.50 Ghz		
8	**971**	**955**	**3 795**
16	**1 886**	**1 888**	**7 543**
32	**3 765**	**3 803**	**15 149**

Du tableau 3, les opérations d'addition et de soustraction avec la propagation de retenue sont coûtent en temps par rapport aux mêmes opérations avec la retenue anticipée. Plus la taille double en bits alors le temps d'exécution sur chaque additionneur augmente en moyenne de 100%. La multiplication est gourmande en temps. Pour la multiplication, le temps d'exécution augmente de presque 400% si la taille de deux entiers double en bits. La division est pire en temps d'exécution car son facteur d'exécution est d'au moins 4 en moyenne par rapport à la multiplication.

Tableau 3. Opération arithmétique sur 8, 16 et 32 bits

AMD E1-2100 APU with Radeon ™ HD Graphics 1000 Mhz	8 bits	16 bits	32 bits
Additionneur à propagation de retenue	97 916	206 781	421 650
Additionneur à retenue anticipée	88 927	185 950	367 906
Soustracteur à propagation de retenue	99 036	205 925	422 038
Multiplication	903 413	3 609 330	14 489 521
Division	3 600 000	14 422 4556	60 000 000

Du tableau 4, les mêmes conclusions sont valables que pour le tableau 3. Du processeur 1 au processeur 2, le temps d'exécution pour tous les circuits est réduit d'environ 400% ou plus.

Tableau 4. Opération arithmétique sur 8, 16 et 32 bits

Intel® Core™ i5-3210 CPU @ 2.50 Ghz	**8 bits**	**16 bits**	**32 bits**
Additionneur à retenue propagée	19 208	39 278.9	79 565.3
Additionneur à retenue anticipée	17 998.0	36 776.0	69 400.2
Soustracteur à propagation de retenue	20 102	40 305	80 121
Multiplication	169 324	676 527	2 731 140
Division	645 000	2 522 678	11 345 677

6 CONCLUSION

Cet article présente une implémentation de l'arithmétique homomorphique basée sur la librairie FHEW. Cette évaluation homomorphique consiste à l'implémentation de différents circuits représentant ces différentes opérations de base par des circuits équivalents en fonction de la fonction NONET. Sur cet environnement avec des spécifications techniques faibles, le potentiel du schéma de [9] a été prouvé promoteur pour l'avenir.

La présente implémentation a été effectuée dans un mode mono-tâche. Pour plus de performance, une implémentation parallèle par CPU ou GPU avec des algorithmes de multiplication optimisés produirait des résultats plus meilleurs pour des applications de la vie courante à grande échelle.

Le traitement parallèle par Central Processing Unit ou par Graphics Processing Unit pourra permettre d'implémenter ces opérations pour des entiers plus grands. Elle ouvrira une possibiliste pour des implémentations des entiers signés et des réels binaires.

REFERENCES

[1] Brakerski Z.: Fully homomorphic encryption without modulus switching from classical GapSVP. In: Safavi-Naini R., Canetti R., (eds) CRYPTO 2012, LNCS, vol. 7417, pp. 868-886. Springer, Berlin (2012);.

[2] Brakerski Z., Gentry C., Vaikuntanathan V., (Leveled) fully Homomorphic encryption without bootstrapping. In: Goldwasser, S. (ed) ITCS 2012, pp. 309-325. ACM, New York (2012);.

[3] Brakerski Z., Vaikuntanathan V., Fully encryption from ring-LWE and security for key dependent messages. In: Rogaway, P. (ed) CRYPTO 2011, LNCS, vol. 6841, pp. 505-524. Springer, Berlin (2011);.

[4] Chen, Y., Gong, G.: Integer arithmetic over ciphertext and homomorphic data aggregation. In: Proceedings of 2015 IEEE Conference on Communications and Network Security, pp. 628-632. IEEE, Piscataway, NJ (2015);.

[5] Chen X., Jingwei C., Wenyuan W., Yong F.: Homomorphically Encrypted Arithmetic Operations over Integer Ring.;.

[6] Cheon J.H., Coron J.S., Kim J., Lee M.S., Lepoint T., Tibouchi M., Yun A.: Batch fully Homomorphic encryption over integers. In: Johanson, T. Nguyen, P.Q. (eds) EUROCRYPT 2013, LNCS, vol 7881, pp. 315-335. Springer, Berlin 2013;.

[7] Coron J.S., Mandal A., Naccache D, Tibouchi M.,: Fully homomorphic encryption over the integers with shorter public keys. In: Rogaway, P. (ed) CRYPTO 2011, LNCS, vol 6841, pp. 487-504. Springer, Berlin 2011;.

[8] Van Dijk M., Gentry C., Halevi S., Vaikuntanathan V.: Fully Homomorphic encryption over the integers. In: Gilbert, H. (ed) EUROCRYPT 2010, LNCS, vol 6110, pp. 24-43, Springer, Berlin (2010);.

[9] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: EUROCRYPT 2015, Part I. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. LNCS. Springer, Heidelberg, Apr. 2015, pp. 617–640. doi: 10.1007/978-3-662-46800-5_24;.

[10] Gentry C.,: A Fully Homomorphic Encryption Scheme. PhD, thesis, Stanford University, Stanford (2009);.

[11] Gentry C.: fully Homomorphic encryption using ideal lattices. In: Mitzenmacher M. (ed) SOC 2009, pp. 169-178. ACM, New York (2009);.

[12] Gentry C., Halevi S., Smart N.P.: Fully Homomorphic encryption with polylog overhead. In: Pointcheval D., Johansson T. (eds) EUROCRYPT 2012, LNCS, vol. 7237, pp. 465482. Springer, Berlin (2012);.

[13] Genty C., Sahai A., Waters B.: Homomorphic encryption from learning with errors: Conceptually – simpler, asymptotically-faster, attribute-based. In: Canetti R., Garay J.A., (eds) CRYPT 2013, Part I, LNCS, vol 8042, pp. 75-92, Springer, Berlin (2013);.

[14] Gentry C., Halevi S., Smart N.P.: Fully Homomorphic Encryption with polylog overhead. In: pointcheval, D., Johanson, T. (eds) EUROCRYPT 2012, LNCS, vol. 7237, pp. 465-482. Springer, Berlin (2012);.
[15] Halevi S., Shoup V.,: Helib: An Implementation of Homomorphic encryption https://github.com/shaih/Helib, accessed in June 2016;.
[16] Kolesnikov, V., Sadeghi, A.R. Scheinder, T.: Improved garbled circuit building blocks and application to auctions and computing minima. In: Garay, J.A., Miyaji, A, Otsuka, A. (eds) CANS 2009, LNCS, vol. 5888, pp. 1-20. Springer Berlin (2009);.
[17] Lyubashevsky, V., Peiker, C., Regev, O,: On ideal lattices and learning with errors over rings. In: Gilbert, H, (ed) EUROCRYPT 2010, LNCS, vol. 6110, pp1-23. Springer, Berlin (2010);.
[18] Regev O.: On lattices, learning with errors, random linear codes, and cryptography. In: Gabow, H.N.., Fagin, R. (eds) STOC 2005, pp. 84-93. ACM, New York (2005);.
[19] Rivest R., Adleman L., Dertrouzos M.,: On data banks and privacy Homomorphism. In: DeMillo, RA., Dobkin, D.P., Jones A.K., Lipton R.J., (eds) Foundations os Secure computation, pp. 165-179. Academic Press, Atlanta (1978);.
[20] Shoup V., NTL: A library for doing number theory. http://shoup.net/ntl/, accessed in June, 2016;.
[21] Smart N. P., Vercauteren F.,: Fully Homomorphic SIMD operations. Designs, Codes and Cryptography 71 (1), 57-81 (2014);.