Le démonstrateur

Sommaire

Ce chapitre est une preuve de concept que la cryptographie homomorphe peut être déployée dans la pratique et peut être utilisée pour assurer la sécurité et le traitement de données des clients avec du matériel existant. Nous présentons un cas concret d'utilisation de la cryptographie homomorphe qui n'a pas été cité auparavant dans la littérature, en utilisant le cryptosystème de Fan et Vercauteren (FV) qui est le plus adapté au cicruit de petite taille comme cité dans le chapitre précédent, et nous proposons une implémentation pratique de ce schéma et son déploiement dans le cadre de l'IoT. Ce cas d'usage est décrit dans la section suivante.

4.1 Cas d'usage

Le chiffrement homomorphe est une solution pour résoudre les principaux problèmes de l'IoT : la securité, le stockage et les calculs délégués sur le cloud. Dans l'IoT, des dispositifs matériels détectent des phénomènes physiques tels que la lumière, la chaleur, le mouvement, l'humidité ou la pression et envoient généralement des signaux qui sont convertis en affichage lisible par l'homme sur le dispositif lui-même ou transmit électroniquement sur un réseau pour traitement ultérieur. C'est dispositifs sont appelés des capteurs.

Considérons un cas d'usage dans l'IoT où différents capteurs envoient des données au cloud à travers plusieurs passerelles. Le but de l'utilisation du cloud est de stocker les données et faire des traitements dessus. À chaque fois que la passerelle reçoit des messages des capteurs, elle les chiffre de façon homomorphe et les envoie au cloud. Pour notre cas, le cloud est en mesure de stocker les données et effectue quelques calculs basés sur l'addition et la multiplication des données collectées à diff ts moments et dans divers contextes géographiques.
Le protocole utilisé pour l'envoi de messages des capteurs aux passerelles est LORA [2] (Long Range Wireless Protocol). Le cloud comprend un serveur MQTT (publish/subscribe) [1], ainsi pour stocker une donnée dans le cloud, la passerelle envoie une commande "publish" pour publier la donnée dans le cloud et pour récupérer une donnée du cloud, la machine de l'administrateur envoie une commande "subscribe" pour se souscrire à un sujet. Ce scénario est illustré par la Figure 4.1.

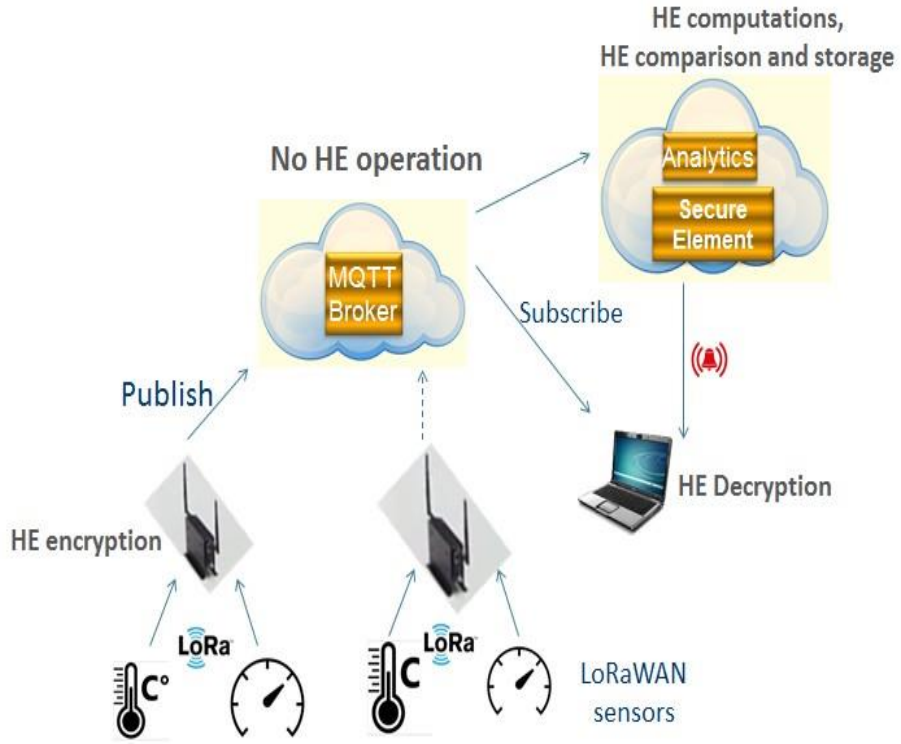

FIGURE 4.1 – Cas d'usage IoT

4.1.1 Cas d'usage des images

Le chiffrement homomorphe peut être utilisé pour assurer la sécurité et le traitement d'image dans le cloud tels que la transformation de couleur, la modification de la saturation et le changement de luminosité. Dans notre modéle, chaque image est représentée par une matrice de pixels, et chiffrer une image revient à chiffrer l'ensemble des pixels. De plus, les opérations du traitement d'image sont basées sur des produits et des sommes de matrices. Prenons l'exemple d'une image de 1024 × 1024 pixels, l'utilisateur envoi l'image à la passerelle qui chiffre chaque pixel par le cryptosystème homomorphe FV.

Chaque chiffré de pixel est de taille 32 koctets, donc la taille globale de l'image chiffrée est 32 Goctets. Si nous utilisons un Ethernet de $0.1 Moctes/s$ nous aurons besoin de 100 heures pour envoyer l'image chiffrée de la passerelle vers le Cloud. Ce cas d'usage n'est pas très efficace et c'est pour cette raison qu'on va se contenter de décrire des cas d'usage où la taille des données à chiffrer est plus petite et plus adaptées aux contraintes de la cryptographie homomorphe.

4.1.2 Cas d'usage plus réalistes

Imaginons un cas d'usage dans un port maritime dans lequel il est très utile d'accélérer les expéditions de produits, ce besoin peut être résolu en prévoyant suffisamment de camions pour transporter la marchandise. Les compagnies maritimes doivent vérifier en temps réel la marchandise transportée sans révéler des informations sur les clients. D'autant plus que les armateurs sont engagés pour protéger le secret professionnel. Les entreprises vérifient la valeur de marchandises, leur coût global, leur contenu et leur poids afin d'anticiper leur arrivée au port maritime, en prévoyant des camions ou des trains pour le transport des marchandises et le passage des douanes.
Dans ce cas d'usage, chaque compagnie possède plusieurs navires et plusieurs conteneurs. De plus, chaque navire possède une passerelle et chaque conteneur possède un capteur. Les conteneurs envoient des données à la passerelle du navire qui les chiffre avant de les envoyer au cloud. En pratique, lorsque le cloud reçoit les données il peut effectuer des calculs sur les données chiffrées sans avoir à les déchiffrer comme c'est décrit dans la Figure 4.2, il peut calculer la somme des conteneurs détenant les produits de certaines entreprises dans le port maritime en augmentant le nombre de conteneurs. Nous pouvons aussi calculer la valeur des marchandises dans ces conteneurs et la somme des poids de conteneurs avant de les charger sur les navires en faisant plusieurs additions. De plus, nous pouvons effectuer une conversion des devises des prix des produits en calculant le produit du prix de chaque produit et la devise du pays. Ce calcul nécessite un niveau de multiplication.

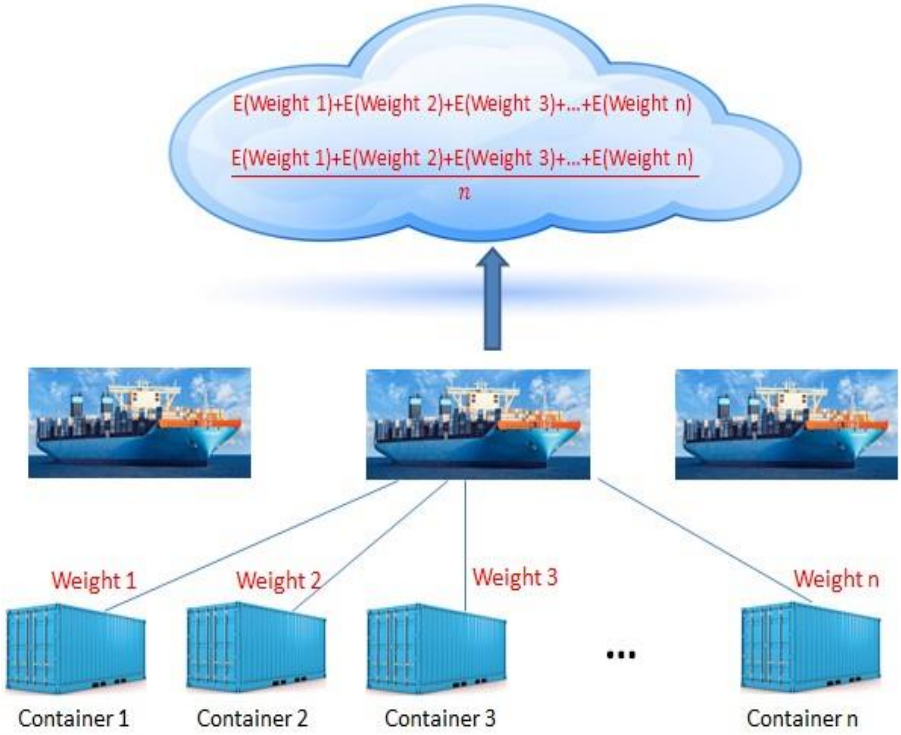

FIGURE 4.2 – Application de la cryptographie homomorphe dans un port maritime

On peut aussi décrire un scénario dans lequel différents supermarchés de différentes entreprises ont besoin de stocker et de calculer le nombre des ventes de produits dans le cadre de la gestion des stocks. L'objectif de ces entreprises est de stocker dans le cloud le chiffrement de ces données sans révéler leur identité en raison de la concurrence. Dans le cloud, nous pouvons calculer la somme de vente de chaque produit afin d'approvisionner le stock des supermarchés si nécessaire.

Prenons l'exemple d'un autre cas d'usage dans le transport et en particulier dans les trains. Ce cas d'usage va nous permettre de vérifier le niveau d'eau dans les toilettes. Supposons que plusieurs trains de plusieurs entreprises possèdent une passerelle par train et plusieurs capteurs pour chaque wagon, chaque passerelle reçoit des messages des capteurs, les chiffre et ensuite les envoie vers le Cloud, celui-ci regroupe les données et effectue des calculs dessus. Le cloud détient un secure element qui permet de prendre des décisions et en particulier faire des comparaisons sur les données chiffrées sur les données regroupées, en déchiffrant les données et en comparant les textes en clairs. La méthode de comparaison dans un secure element est décrite en détail dans la Section 4.3. Les appareils envoient systématiquement le niveau d'eau au cloud via la passerelle et à l'intérieur du cloud, ensuite le secure element compare le niveau d'eau au niveau minimum acceptable et par conséquent envoie des alertes à l'administrateur en cas de manque d'eau dans les toilettes. Le chiffrement homomorphe garantit la confidentialité, le respect de la vie privée et l'anonymisation de l'origine des conteneurs, tout en permettant le traitement et le stockage des données dans un cloud publique.

Nous avons choisi d'utiliser le système de cryptographie homomorphe FV dans les cas d'usage décrit ci-dessus, parce qu'il le plus adapté aux circuits de petite profondeur. De nombreuses bibliothèques [43], [31] implémentent le cryptosystème FV, mais elles ne sont pas portables dans la passerelle. La taille de la bibliothèque SEAL est 5 Moctets et la taille de la bibliothèque FV-NFLIB plus la bibliothèque NFLIB nécessaire pour la faire tourner est 5 Moctets aussi. Nous avons choisi de mettre en œuvre notre propre API pour l'intégrer dans la passerelle, le cloud et dans la machine administrateur.

4.2 Implémentation

Nous avons implémenté le cryptosystème FV en langage C. Il a été intégré dans les modules IoT suivants : la passerelle, le cloud et la machine de l'administrateur. Il s'agit d'un code portable, autonome et indépendant des autres bibliothèques. Cette implémentation peut être considérée comme une preuve de concept où nous n'utilisons que la bibliothèque mathématique standard "math.h" sans parallélisme SIMD et sans traitement multi-coeur. Nous utilisons un ensemble de paramètres permettant d'évaluer des circuits d'un seule niveau de multiplication, avec un niveau de sécurité de 128 bits, un degré 2048 pour le polynôme cyclotomique et 56 bits pour les coefficients polynômes, on a choisi l'anneau R_2 comme distribution de clé secrète χ_{key}, $\sigma = 3.1$, et χ_{err} une distribution gaussienne bornée par $B = 31$ pour choisir les coefficients du bruit. Nous avons choisi ces paramètres parce qu'ils sont les plus petits permettant de construire un schéma capable d'évaluer des circuits d'une multiplication avec un niveau de sécurité correct. Ce schéma permet d'évaluer un niveau de multiplication et chiffre des plaintexts de 16 bits. Le code est intégré et exécuté sur une machine d'administrateur qui détient un processeur Intel Core i5 à 2,4 GHz, une passerelle avec un processeur Intel Atom à 1,91 GHz, et une machine virtuelle utilisant un processeur Intel Xeon à 2,40 GHz au niveau du cloud.

4.2.1 Description de l'implémentation

Nous avons implémenté notre propre code à partir de zéro afin d'obtenir une implémentation homogène, et portable dans la passerelle. Dans cette implémentation, les coefficients des polynômes sont calculés modulo q et stockés dans une structure de type long long capable de contenir 64 bits. Si q est une puissance de 2, par exemple $q = 2^{56}$, le calcul de modulo q correspond à une opération AND. Soit N et k deux entiers où $q = 2^k$, l'opération de module vérifie l'équation suivante : $N \bmod 2^k = N \,\&\, 2^k - 1$.
Dans notre code, nous avons développé notre propre arithmétique 128 bits afin de permettre le stockage du résultat d'une multiplication de deux long long. Soit A et B deux entiers de 64 bits, de type long long, on note $A = A_1|A_2$ et $B = B_1|B_2$ où A_1, A_2, B_1, B_1, B_2 sont des entiers de 32 bits chacun. Pour calculer

le produit de A et B nous utilisons l'Algorithme 52.

Algorithme 52 *L'arithmétique 128 bits*

Entrées $A = A_1|A_2$ and $B = B_1|B_2$.
Sortie Le produit $A \times B$.

fonction MULTIPLIER DEUX LONG LONG(A, B)
Calculer $A_2 \cdot B_2 = r_1|r_2$.
Calculer $A_1 \cdot B_2 = r_3|r_4$.
Calculer $A_2 \cdot B_1 = r_5|r_6$.
Calculer $A_1 \cdot B_1 = r_7|r_8$.
Calculer $S_1 = r_1 * 2^{32} + r_4 + r_6$.
7: Calculer $M_1 = (r_1 * 2^{32} + r_4 + r_6) \bmod 2^{32}$.
Calculer $S_2 = r_4 * 2^{32} + r_5 * 2^{32} + r_8 + M_1$.
Calculer $M_2 = (r_4 * 2^{32} + r_5 * 2^{32} + r_8 + M_1) \bmod 2^{32}$.
Calculer $S_3 = r_7 * 2^{32} + M_2$.
retourner $A \cdot B = S_3|S_2|S_1|r_2$.
fin fonction

Dans cette implémentation on a repris l'algorithme de Fan et Vercauteren en omettant l'étape de la relinéarisation, parce que nous avons besoin d'assurer une seule multiplication. Autrement dit, le résultat de la multiplication des chiffrés est un texte chiffré de trois éléments, et nous pouvons facilement faire des additions entre des textes chiffrés de trois éléments et des additions entre un texte chiffré de trois éléments et un texte chiffré de deux éléments. La relinéarisation augmente le niveau du bruit dans le chiffré à cause de la multiplication par la clé de relinéarisation, et donc nous ne pouvons même pas effectuer une seule multiplication avec les paramètres choisis ($\lambda = 128$, $d = 2048$, $|q| = 56$). Par conséquent, il faut augmenter les paramètres pour pouvoir déchiffrer correctement. En revanche, en augmentant les paramètres les coefficients des polynômes ne peuvent plus être stockés dans une structure long long. Donc on a choisi de garder les chiffrés tels qu'ils sont, avec trois éléments, après la multiplication et implémenter le cryptosystème FV avec les paramètres : $\lambda = 128$, $d = 2048$, et $|q| = 56$. On présente ci-dessous les fonctions de l'algorithme de FV que nous avons implémenté.

4.2.2 Les algorithmes implémentés

Soient λ le paramètre de sécurité, $q > 1$ le module des coefficients, et $t > 1$ le module du texte en clair, tel que $t < q$. On note par R_q et R_t l'espace des chiffrés et des plaintexts. $R_q[x] = Z_q[x]/f(x)$ où $Z_q[x]$ est l'anneau de polynôme à coefficients modulo q, $f(x) = x^d + 1$ et $d = 2^n$. Les éléments de $R_q[x]$ sont des polynômes de degré inférieur à d et à coefficients modulo q.
Les éléments de l'anneau $R_q[x]$ sont notés en minuscule ($a \in R_q[x]$), on note par

$[a]_q$ les éléments de R obtenus en calculant tous les coefficients modulo q.
Pour tout $x \in R_q$ on note par lxl un nombre arrondi à l'entier le plus proche, fxl et lxJ des nombres arrondis à l'entier supérieur et inférieur. La notation $x \leftarrow D$ est utilisée pour tirer aléatoirement x d'une distribution D, et $x \xleftarrow{\$} D$ pour tirer uniformément x de D.
Soit χ une distribution gaussienne sur R_s avec une déviation standard σ. Les deux distributions χ_{err} et χ_{key} sont utilisées pour tirer les erreurs et la clé secrète du schéma. Ces distributions χ_{err} et χ_{key} sont bornées avec une borne B où $B = 10 \cdot \sigma$. On a choisi l'anneau R_2 comme distribution de clé secrète χ_{key}, $\sigma = 3.1$, et χ_{err} une distribution gaussienne bornée par $B = 31$. Soit $params$ l'ensemble de paramètres du schéma, $params = (R, d, q, t, \chi_{err}, \chi_{key})$.
L'Algorithme 53 décrit les étapes de génération de la clé privée s_k et la clé publique $p_k = (p_k[0], p_k[1])$.

Algorithme 53 Génération des clés

Entrées $params = (R, d, q, t, \chi_{err}, \chi_{key})$.
Sortie $s_k et p_k$.

fonction GÉNÉRATION DE CLÉS($params$)
$s_k \leftarrow \chi_{key}$
$a \xleftarrow{\$} R_q$
$e \leftarrow \chi_{err}$
$p_k = ([-(a \cdot s_k + e)]_q, a)$
retourner s_k et p_k.

fin fonction

L'Algorithme 54 décrit la procédure de chiffrement d'un message m en utilisant la clé publique $p_k = (p_k[0], p_k[1])$.

Algorithme 54 Chiffrement d'un message

Entrées $m \in R_t$ et p_k.
Sortie $E(m) = (c[0], c[1])$.

fonction CHIFFREMENT(m, p_k)
$\delta = l\frac{q}{t}J$
$u \leftarrow \chi_{key}$
$e_1 \leftarrow \chi_{err}$
$e_2 \leftarrow \chi_{err}$
$c[0] = [p_0 \cdot u]_q$
$r_0 = [c[0] + e_1]_q$
$c[0] = [r_0 + \delta \cdot m]_q$
$r_0 = [p_1 \cdot u]_q$
$c[1] = [r_0 + e_2]_q$
$E(m) = (c[0], c[1])$
retourner $E(m)$
fin fonction

L'Algorithme 55 décrit la procédure de déchiffrement d'un chiffré $(c[0], c[1])$ en utilisant la clé secrète s_k.

Algorithme 55 déchiffrement d'un chiffré $(c[0], c[1])$

Entrées $C = (c[0], c[1])$ et s_k.
Sortie $D(C)$.

fonction DÉCHIFFREMENT(C, s_k)
$r_0 = c[1] \cdot s_k$
$D(C) = [c[0] + r_0]_q$
$r_0 = t \cdot D(C)$
$D(C) = [l\frac{r_0}{q}l]_t$
retourner $D(C)$
fin fonction

L'Algorithme 56 décrit la procédure de déchiffrement d'un chiffré $(c[0], c[1], c[2])$, qui a subi une multiplication, en utilisant la clé secrète s_k.

Algorithme 56 déchiffrement d'un chiffré $(c[0], c[1], c[2])$

Entrées $C = (c[0], c[1], c[2])$ et s_k.
Sortie D(C).

fonction DÉCHIFFREMENT(C, s_k)
$r_0 = c[1] \cdot s_k$
$s_2 = s_k \cdot s_k$
$r_1 = c[2] \cdot s_2$
$D(C) = [c[0] + r_0 + r_1]_q$
$r_0 = t \cdot D(C)$
$D(C) = [l\frac{r_0}{q}l]_t$
retourner D(C)
fin fonction

L'Algorithme 57 décrit le calcul de la somme de deux chiffrés $c(c[1][0], c[1][1])$ et $(c[2][0], c[2][1])$.

Algorithme 57 Addition de chiffrés $c(c[1][0], c[1][1])$ et $(c[2][0], c[2][1])$

Entrées $c[1] = (c[1][0], c[1][1])$ et $c[2] = (c[2][0], c[2][1])$.
Sortie $S = c[1] + c[2]$.

fonction ADDITION$(c[1], c[2])$
$S[0] = c[1][0] + c[2][0]$
$S[1] = c[1][1] + c[2][1]$
retourner $S = (S[0], S[1])$.
fin fonction

L'Algorithme 58 décrit le calcul de la somme d'un chiffré résultant d'une multiplication $c(c[1][0], c[1][1], c[1][2])$ et un chiffré qui n'a jamais subi de multiplication $(c[2][0], c[2][1])$.

Algorithme 58 Addition de chiffrés $c(c[1][0], c[1][1], c[1][2])$ et $(c[2][0], c[2][1])$

Entrées $c[1] = (c[1][0], c[1][1], c[1][2])$ et $c[2] = (c[2][0], c[2][1])$.
Sortie $S = c[1] + c[2]$.

fonction ADDITION$(c[1], c[2])$
$S[0] = c[1][0] + c[2][0]$
$S[1] = c[1][1] + c[2][1]$
$S[2] = c[1][2]$
retourner $S = (S[0], S[1], S[2])$.
fin fonction

L'Algorithme 59 décrit le calcul de la somme de deux chiffrés issus d'une multiplication $c(c[1][0], c[1][1], c[1][2])$ et $(c[2][0], c[2][1], c[2][2])$.

Algorithme 59 Addition de chiffrés $c(c[1][0], c[1][1], c[1][2])$ et $(c[2][0], c[2][1], c[2][2])$

Entrées $c[1] = (c[1][0], c[1][1])$ et $c[2] = (c[2][0], c[2][1])$.
Sortie $S = c[1] + c[2]$.

fonction ADDITION($c[1]$, $c[2]$)
$S[0] = c[1][0] + c[2][0]$
$S[1] = c[1][1] + c[2][1]$
$S[2] = c[1][2] + c[2][2]$
retourner $S = (S[0], S[1], S[2])$.
fin fonction

Cet Algorithme 58 décrit le calcul du produit de deux chiffrés $c(c[1][0], c[1][1])$ et $(c[2][0], c[2][1])$. Rappelons que notre implémentation permet d'évaluer uniquement des circuits d'un seul niveau de multiplication, d'où le besoin d'une fonction permettant d'effectuer le produit sur des chiffrés qui n'ont jamais subi de multiplication.

Algorithme 60 Multiplication de chiffrés $c(c[1][0], c[1][1])$ et $c[2] = (c[2][0], c[2][1])$

Entrées $c[1] = (c[1][0], c[1][1])$ et $c[2] = (c[2][0], c[2][1])$.
Sortie $M = c[1] \cdot c[2]$.

fonction MULTIPLICATION($c[1]$, $c[2]$)
$ct_0 = t \cdot (c[1][0] \cdot c[2][0])$
$r_0 = \frac{ct_0}{q}$
$ct_0 = [lr_0l]_q$

$ct_1 = c[1][0] \cdot c[2][1]$
$r_0 = ct_1 + c[1][1] \cdot c[2][0]$
$ct_1 = t \cdot r_0$
$r_0 = \frac{ct_1}{q}$
$ct_1 = [lr_0l]_q$

$ct_2 = t \cdot (c[1][1] \cdot c[2][1])$
$r_0 = \frac{ct_2}{q}$
$ct_2 = [lr_0l]_q$
retourner $M = (ct_0, ct_1, ct_2)$
fin fonction

Notre implémentation peut évaluer tous les circuits qui peuvent être de la forme décrite dans les figures Figure 4.3 et Figure 4.4 .

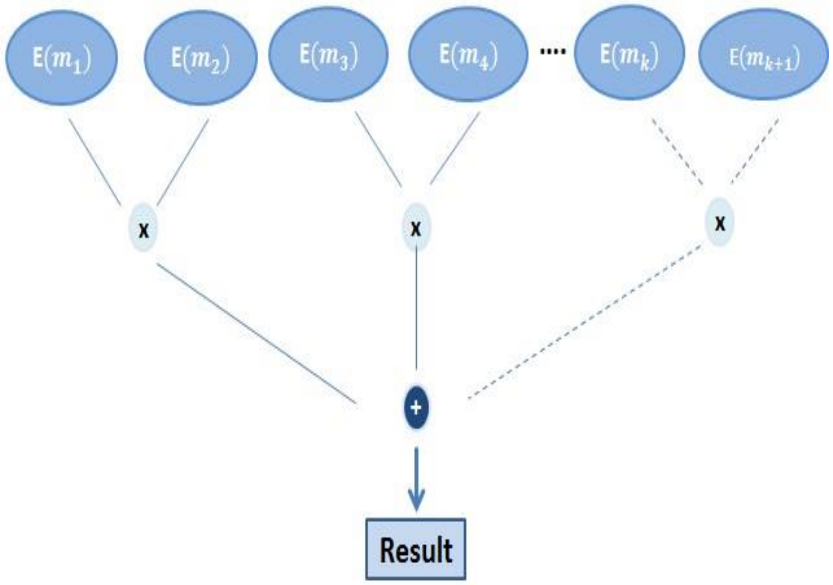

FIGURE 4.3 – Circuit permettant de calculer une moyenne pondérée

Un cas d'utilisation pratique du circuit de la Figure 4.3 est le calcul de la moyenne pondérée, ou le produit de matrices utilisé dans le traitement d'images pour transformer les couleurs, modifier la saturation et changer la luminosité. Ce circuit peut être utilisé dans le contexte du port maritime pour convertir la valeur de la marchandise de l'euro en yens par exemple.

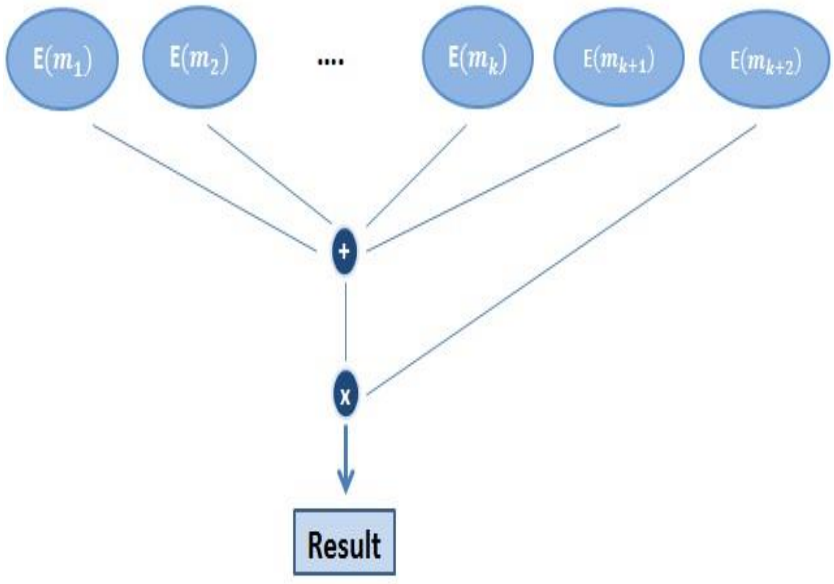

FIGURE 4.4 – Circuit permettant de calculer une moyenne

Le circuit de la figure Figure 4.4 peut être utilisé pour calculer une moyenne, ce calcul est pratique dans le cas d'usage du port maritime où les armateurs ont besoin de calculer le poids moyen des conteneurs présent dans le port d'une certaine compagnie en calculant la somme des poids et ensuite diviser par le nombre de conteneurs, cette division peut être écrite sous forme de multiplication $\frac{1}{sum}$.

4.2.3 Performances

Dans ce paragraphe, nous présentons les performances de notre implémentation sous différentes plateformes. Le même programme a été embarqué dans la passerelle, le cloud et la machine administrateur, nous avons ajouté quelques options pour spécifier quelle fonction du code nous voulons exécuter. Les fonctions de génération de clés et de chiffrement peuvent être exécutées dans la passerelle (processeur Intel Atom à 1.91GHz). Dans le cloud public (processeur Intel Xeon à 2.40GHz), nous ne devons exécuter que les fonctions d'addition et de multiplication de chiffrés. En outre la fonction de déchiffrement ne peut être exécutée que sur la machine de l'administrateur. Nous commençons par donner la taille des textes chiffrés et des clés de notre implémentation qui manipule des textes clair de 16 bits. La taille du texte chiffré est 32 koctets, la taille de la clé secrète est de 2048 bits et la taille de la clé publique est 32 koctets. Les temps d'exécution de toutes les fonctions à l'intérieur des différents modules de notre cas d'usage sont décrits dans le tableau Table 4.1. Nos résultats prouvent que nous pouvons utiliser le chiffrement homomorphe dans un cadre réel. Le timing pourrait probablement être amélioré, mais notre premier objectif était de déployer une implémentation homogène du cryptosystème FV entre différentes plateformes et un cloud.

Opération	Temps d'exécution dans la machine de l'administrateur	Temps d'exécution dans le Cloud	Temps d'exécution dans la passerelle
Génération de la clé secrète(*ms*)	0.122	—	0.151
Génération de la clé publique(*ms*)	70.994	—	175.946
Chiffrement (*ms*)	94.598	—	233.007
Déchiffrement d'un texte chiffré normal(*ms*)	18.934	—	—
Déchiffrement d'un texte chiffré avec 3 éléments (*ms*)	62.177	—	—
Addition (*ms*)	0.085	0.022	—
Multiplication (*ms*)	177.553	203.626	—

TABLEAU 4.1 – Temps d'exécution des fonctions

4.3 Comparaison des textes chiffrés homomorphes

Le chiffrement homomorphe est très prometteur pour l'industrie. Cependant, il est impossible de prendre une décision dans le cloud, basée sur la comparaison de deux chiffrés homomorphes. Le résultat d'une comparaison de deux chiffrés est un booléen chiffré, par conséquent, on ne peut pas prendre des décisions ou envoyer des alertes depuis le cloud en se basant sur ce résultat. L'idée présentée dans cette section consiste à utiliser un secure element dans le cloud pour déchiffrer en toute sécurité les chiffrés pour pouvoir comparer les plaintexts correspondants afin de pouvoir prendre des décisions, comme décrit précédemment dans l'exemple du niveau d'eau dans les toilettes des trains.

4.3.1 Description de la méthode

Nous avons proposé un système où nous utilisons un secure element à l'intérieur du Cloud afin de déchiffrer les chiffrés et comparer les plaintexts correspondants pour pouvoir prendre des décisions et puis l'envoi d'alertes si nécessaire.
La fonction de déchiffrement $D(C) = [l^{\frac{t \cdot [c[0] + c[1] \cdot s_k]_q}{q}} l]_t$ sera embarquée dans le secure element. Mais la contrainte qui se pose est que le calcul de cette fonction est très coûteux au niveau de la mémoire nécessaire pour le stockage des variables intermédiaires et en particulier le calcul de $(c[1] \cdot s_k) \bmod q$.
Nous avons proposé alors d'éviter de calculer le produit $(c[1] \cdot s_k) \bmod q$, qui nécessite des ressources mémoires et un temps d'exécution importants. Ainsi, le calcul de ce produit sera délégué au CPU, sans révéler la clé secrète. On commence par générer à l'intérieur du secure element un polynôme aléatoire $Random \in R_2$ avec des coefficients binaires. Nous calculons le masque de la clé secrète $M(s_k) = (s_k \oplus Random) \bmod q$, qui est la somme coefficient par coefficient des polynômes s_k et $Random$. Ensuite on envoit le masque $M(s_k)$ et le polynôme $c[1]$ au CPU. De son coté, le CPU calcule $(c[1] \cdot M(s_k)) \bmod q$ et envoie le résultat au secure element. Ce dernier extrait la partie aléatoire du résultat reçu du CPU, en calculant :

$$[(c[1] \cdot M(s_k)) \mod q - (c[1] \cdot Random) \mod q)] \mod q = (c[1] \cdot s_k) \mod q.$$

Notons que le calcul du produit $(c[1] \cdot Random) \bmod q$ nécessite moins de mémoire que le calcul du produit $(c[1] \cdot s_k) \bmod q$. En effet, si on choisit $|q| = 56$ et $s_k \leftarrow R_q$ alors c_1 et s_k sont des polynômes de degré 2048 avec des coefficients de taille 56 bits. D'autre part $Random$ est un polynôme de degré 2048 avec des coefficients de un bit. Donc le produit de c_1 et $Random$ revient à faire que des sommes des coefficients de type long long, contrairement au produit de c_1 et $Random$ qui nécessite de faire des produits des entiers de type long long suivi de sommes.
L'Algorithme 61 présente notre méthode pour déléguer le produit au CPU. Les lignes 2, 3, 4, et 5 de l'algorithme sont exécutés une seule fois au démarrage du secure element. En outre, le même polynôme aléatoire et le masque de clé secrète sont utilisés à chaque fois qu'une délégation du produit au CPU est faite.

Algorithme 61 L'algorithme de délégation du produit de polynômes au CPU

Entrées $(c[1], s_k)$.
Sortie $c[1] \cdot s_k$.

1: **fonction** DÉLÉGATION$(c[1], s_k)$
2: Le secure element génère un polynôme random $Random \in R_2$.
3: Le secure element calcule $A = c[1] \cdot Random$.
4: Le secure element calcule $M(s_k) = s_k \oplus Random$.
5: Le secure element envoi $M(s_k)$ au CPU.
6: Le CPU calcule $B = c[1] \cdot M(s_k)$.
7: Le CPU envoi le résultat B au secure element.
8: Le secure element calcule $B - A = c[1] \cdot s_k$.
9: **retourner** $c[1] \cdot s_k$
10: **fin fonction**

4.3.2 Analyse de sécurité

Pour permettre l'envoi sécurisé de la clé secrète, nous avons proposé ci-dessus d'envoyer le masque $M(s_k) = s_k \oplus Random$ au CPU. Chaque coefficient de $M(s_k)$ est calculé en ajoutant le coefficient correspondant de la clé secrète s_k et le polynôme aléatoire $Random$, $M(s_k)[i] = (s_k[i] \oplus Random[i]) \mod q$. La valeur de chaque $M(s_k)[i]$ est soit $s_k[i]$ soit $s_k[i] + 1$, car $Random[i] \in \{0, 1\}$, alors l'attaque exhaustive sur $M(s_k)[i]$ est une attaque sur un polynôme de degré d à coefficients binaires. Comme d est égal ou supérieur à 1024, pour un niveau de sécurité 128, l'attaque en force brute sur le masque n'est pas possible.

4.4 Synthèse sur le démonstrateur

Dans ce chapitre, nous avons présenté notre implémentation pratique et autonome du cryptosystème FV en langage C pour utiliser le chiffrement homomorphe dans la vie réelle. Cette implémentation a été embarquée dans une passerelle et un cloud, permettant d'évaluer des circuits d'un seul niveau et assurant une sécurité de 128 bits. Les armateurs peuvent utiliser cette implémentation dans le port maritime pour chiffrer les données des conteneurs dans les passerelles et effectuer des calculs dans le cloud, comme la somme pondérée des conteneurs, le poids des conteneurs avant leur chargement sur le navire, ou pour effectuer la conversion monétaire des marchandises. Grâce à notre implémentation, les armateurs peuvent manipuler des données chiff tout en respectant la concurrence entre les entreprises et sans impact sur la sécurité, la confidentialité et l'anonymat. Nous avons également décrit un cas d'utilisation dans le contexte de la gestion des stocks où la comparaison est nécessaire pour l'approvisionnement des stocks des supermarchés. En conséquence, nous avons proposé une méthode permettant de prendre des décisions basées sur la compa-

raison des chiffrés homomorphes. Cette méthode consiste à déchiff les textes chiffrés de FV et à comparer les textes en clairs dans un secure element. En utilisant notre contribution, l'exécution de la fonction du déchiffrement à l'intérieur du secure element devient possible en déléguant la multiplication polynomiale au CPU.