USE CASE

A PRACTICAL USE CASE OF HOMOMORPHIC ENCRYPTION

POSSIBILITIES START HERE kontron S&T Group

CONTENTS

A homomorphic encryption is a cryptographic method that has homomorphic properties, allowing calculations to be performed on the ciphertext corresponding to mathematical operations on the corresponding plaintext.

Author:
Amina BEL KORCHI,
KONTRON modular computers
Nadia EL MRABET,
Ecole nationale sup´erieure des mines de Saint-Etienne

This paper is a proof of concept that homomorphic encryption can be deployed in practice and can be used by the industry to ensure security and computation of customer data.

In this paper, we present a concrete use case of homomorphic encryption that is not considered in literature, using Fan and Vercauteren (FV) cryptosystem, and we propose a practical implementation of FV cryptosystem and its deployment in an IoT use case.
Keywords: IoT use case, cloud security, anonymity, homomorphic encryption

I. INTRODUCTION

Nowadays, data security has become a very important subject for the industries to improve business. They need to manipulate data, while ensuring data protection, privacy and anonymization.

Homomorphic encryption responds to this challenge and enables calculations on encrypted data without decryption. Let $E(a)$ and $E(b)$ be the encryption of a and b using an homomorphic cryptosystem, $E(a)$ and $E(b)$ verify the following properties: $E(a) \oplus E(b) = E(a \oplus b)$ and $E(a) \times E(b) = E(a \times b)$.

Two variants of homomorphic encryption exist:
Fully Homomorphic Encryption (FHE) and Somewhat Homomorphic Encryption (SWHE). FHE is a fully homomorphic encryption allowing the evaluation of an arbitrary circuit, as to SWHE, it can evaluates circuits of constant depth. The circuit depth is the number of multiplication that can be performed using a given scheme. Exceeding this depth, decryption can not be done correctly due to the noise that appears during the encryption of plaintexts. This noise grows after every ciphertext multiplication until we reach a level where we can not decrypt correctly.

Gentry[11] has invented the first FHE cryptosystem in 2009 using a bootstrapping[11] procedure to transform a SWHE cryptosystem into a FHE cryptosystem. The security of Gentry's scheme is based on ideal lattices[18].

The bootstrapping technique transforms a ciphertext resulting from a circuit to a new ciphertext with a noise similar to the one in a ciphertext freshly encrypted.

Numerous schemes have been proposed following Gentry's cryptosystem[22] [8] [12], basing their security on different hardness assumptions.

Before the apparition of the technique to turn bootstrapping in less than 0.1 seconds[9], the inconvenient of FHE schemes was the time to turn the boostrapping, this is the reason why different SWHE schemes as[7] [10] [6] have been developed with practical depths to use homomorphic encryption in practice.

Those practical security schemes are based on LWE[19] (Learning with errors) and RLWE[16] (Ring Learning with Errors) problems.

In this paper, we give a practical use case of homomorphic encryption, we validate our scheme using a home made implementation of FV cryptosystem and provide the detailed description of this use case and the description of our implementation.

In the first section of the paper, we define the use case. Then, in the second section, we give the description of FV cryptosystem. In the third section, we describe existing implementations of FV, and finally we introduce our implementation with its performances inside the cloud, the gateway and inside an administrator machine.

II. IOT USE CASE

Homomorphic encryption is a solution to solve the main problems of IoT[1]: security, storage and computations.

Let's picture a use case in IoT where we have different devices, several gateways and a cloud with multiple servers to store and manage data. Each gateway receives several messages from sensors, encrypts messages homomorphically and sends them to the cloud. For our case the cloud will store those ciphertexts and makes some calculations based on addition and multiplication of collected data at different time and in various geographies.

The protocol used for sending messages from sensors to the gateway is LORA[3] (Long Range Wireless Protocol). The cloud includes a MQTT[2] server (Publish/ Subscribe protocol). To store data in the cloud, the gateway sends a publish command, and to receive a data from the cloud, the calculation server sends a subscribe command. This scenario is shown in Figure 1.

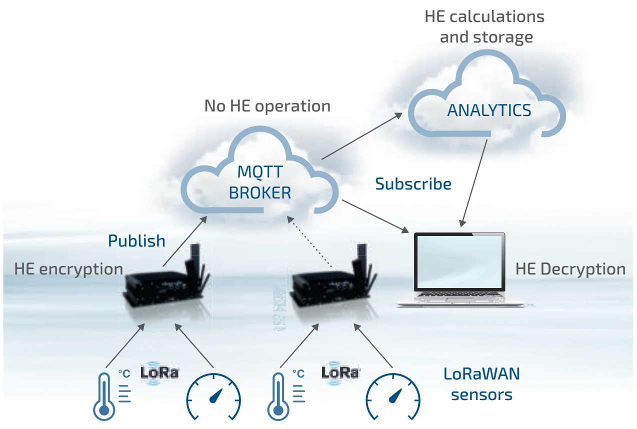
Let's take a scenario where different supermarkets of different companies need to store and compute the number of product sales in the context of stock management. The goal of these companies is to store in the cloud the encryption of this data without revealing their identities due to the competition. In the cloud we can compute the sum of sales of each product in order to supply the stock of supermarkets if necessary.

Let's picture a use case in seaport to accelerate the shipments of products through customs by expecting enough trucks to transport the goods. Shipping companies need to verify in real time the transported merchandise without revealing information about the clients. Shipowners are engaged with clients to protect the professional secrecy. The companies will verify the value of goods, their overall cost, their content and the weight of goods in order to anticipate the arrival of goods to the seaport, the trucks or train to transport the merchandise and the passage of customs.

In this use case each company have several ships and containers, each ship owns a gateway, and each container holds a sensor. Containers send data to the gateway which will encrypts them and send them to the cloud. In practice when the cloud receives data we can compute the sum of containers holding products of some companies in the seaport by incrementing the number of containers. We can compute the value of goods of those containers, and the sum of weights of containers before loading them on ships by doing several additions. Moroever, we can compute the currency conversion of goods prices by computing the price of each product and the country currency. This calculation needs one level of multiplication.

Homomorphic encryption will ensure confidentiality, privacy and anonymization of container origin while allowing publishing an aggregate figure.

We choose to use the homomorphic cryptosystem FV in the use cases above, because it is well suited for circuits of small depth. Many libraries[14) 20)] implement the FV cryptosystem, but they are not portable in the gateway. the size of SEAL library is 5 Mbytes, which has no external dependencies, FV-NFLIB library uses the NFLIB library, and to turn it inside the gateway we need the full module of 5 Mbytes. We choose to implement our own API to be embedded in the gateway, the cloud and the administrator machine.



// Figure 1: IoT use case: the Edge/Cloud solution

III. FV CRYPTOSYSTEM

A. Notations

The algebraic structure used by Fan and Vercauteren scheme is the polyomial ring $R = Z[x]/f(x)$, where $Z[x]$ describes the polynomial ring with coefficients in Z and $f(x)$ is a cyclotomic polynomial of degree d. In practice $f(x) = x^d+1$ and $d = 2^n$. Elements of R are polynomials of degree less than d and coefficients in Z. Let q denotes coefficients modulus, $R_q[x] = Z_q[x]/f(x)$ where $Z_q[x]$ is the polynomial ring with coefficients modulo q. Elements of $R_q[x]$ are polynomials of degree less than d and coefficients modulo q.

Elements of the ring $R_q[x]$ are noted in lowercase ($a \in R_q[x]$), we denote by $[a]_q$ the elements in R obtained by computing all its coefficients modulo q.

For $x \in R_q$ we denote by $[x]$ rounding to the nearest integer, $\lceil x \rceil$ and $\lfloor x \rfloor$ rounding up and down. Let D denotes a distribution, the notation $x \leftarrow D$ is used to sample randomly x from the distribution D, and $x \overset{\$}{\leftarrow} D$ is used to sample uniformly x from D.

B. Definition of FV

The cryptosystem of Fan and Vercauteren (FV)[10)] is a somewhat homomorphic encryption scheme, developed in 2012, its security is based on the hardness of the RLWE problem[19)].

Let λ denotes the security parameter, $q > 1$ denotes the coefficients modulus used to define the polynomial ring R_q, and $t > 1$ an integer used to denote the plaintext modulus, where $t < q$. R_q is the ciphertext space and the plaintext space is R_t.

Let χ be a Gaussian distribution over R with a standard deviation σ. We will use two distributions χ_{err} and χ_{key} to sample errors and the secret key of the scheme. The distributions χ_{err} and χ_{key} are B bounded distributions where $B = 10 \cdot \sigma$. In practice we can choose χ_{key} as the ring $R_2, \sigma = 3.1$, and χ_{err} is a Gaussian distribution bounded by $B = 31$.

We denote by params the set of the scheme parameters, $params = (R, d, q, t, \chi_{err}, \chi_{key})$.

KeyGen$(params)$:
FV cryptosystem is a public key scheme, the generation of keys will return a public key p_k and a secret key s_k. Let's start by generating the secret key which is a polynomial in χ_{key}, $s_k \leftarrow \chi_{key}$ is randomly sampled from χ_{key} , in practice we choose $\chi_{key} = R_2$ and s_k a polynomial of degree less than d with binary coefficents.

To compute the public key we need to sample a from the ring R_q $a \xleftarrow{\$} R_q$, and a random error e from χ_{err} $e \xleftarrow{\$} \chi_{err}$. The public key p_k is a couple of two polynomials,

$$p_k = ([-(a \cdot s_k + e)]q, a).$$

Let rlk be the relinearization key used to reduce the size of ciphertext after multiplication.

For $i = 0 \ldots l$ we sample a_i from R_q $a_i \xleftarrow{\$} R_q$, e_i from χ_{err} $e_i \xleftarrow{\$} \chi_{err}$, and compute $rlk = ([-(a_i \cdot s_k + e_i) + T^i \cdot s^2]_q, a_i)$, where T is a random integer independent from the plaintext modulus t and $l = \lfloor log_T(q) \rfloor$. The generation of key is described in Algorithm 1.

▶ ALGORITHM 1 Generate keys

```
Input $params = (R, d, q, t, \chi_{err}, \chi_{key})$.
Output $s_k, p_k, rlk$.
1: function keyGeneration($params$)
2:     $s_k \leftarrow \chi_{key}$.
3:     $a \xleftarrow{\$} R_q$.
4:     $e \leftarrow \chi_{err}$.
5:     $pk = ([-(a \cdot s_k + e)]_q, a)$.
6:     Choose $T$ an integer independent from $t$.
7:     Compute $l = \lfloor log_T(q) \rfloor$.
8:     for $i = 1$ to $l$ do
9:         $a_i \xleftarrow{\$} R_q$.
10:        $e_i \xleftarrow{\$} \chi_{err}$.
11:        $rlk[i] = ([-(a \cdot s_k + e) + T^i \cdot s_k^2]_q, a_i)$.
12: return $s_k$, $p_k$ and $rlk$.
```

Encryption $(m, params, pk = (p_0; p_1))$:
To encrypt a message $m \in R_t$, we start by computing $\delta = \lfloor \frac{q}{t} \rfloor$, we sample $u \leftarrow \chi_{key}$, and $e1, e2 \leftarrow \chi_{err}$.
The encryption of m is a ciphertext of two polynomials defined as follows:

$$E(m) = ([p0 \cdot u + e1 + \delta \cdot m]_q, [p1 \cdot u + e2]_q). \quad (1)$$

One can notice that the ciphertext is a couple of data, where we can see the first element as the ciphertext, and the second as the noise. The encryption step is described in Algorithm 2.

▶ ALGORITHM 2 Encrypt a message

```
Input $m \in 2\ R_t$ and $p_k$.
Output $E(m) = (c[0]; c[1])$.
    function Encryption($m, p_k$)
        $\delta = \lfloor \frac{q}{t} \rfloor$
        $u \leftarrow \chi_{key}$
        $e_1 \leftarrow \chi_{err}$
        $e_2 \leftarrow \chi_{err}$
        $c[0] = [p_0 \cdot u]_q$
        $r_0 = [c[0] + e_1]_q$
        $c[0] = [r_0 + \delta \cdot m]_q$
        $r_0 = [p_1 \cdot u]_q$
        $c[1] = [r_0 + e_2]_q$
        $E(m) = (c[0], c[1])$
    return $E(m)$
```

Decryption $(C, params, s_k)$:
To compute the decryption of a ciphertext $C = (c0, c1)$ we evaluate the following equation:

$$D(C) = [\lfloor \frac{t \cdot [c[0] + c[1] \cdot s_k]_q}{q} \rceil]_t,$$

as presented in Algorithm 3.

▶ ALGORITHM 3 Decrypt a ciphertext

```
Input $C = (c[0], c[1])$ and $s_k$.
Output $D(C)$.
    function Decryption($C, s_k$)
        $r_0 = [c[01] \cdot s_k$
        $D(C) = [c[0] + r_0]_q$
        $r_0 = t \cdot D(C)$
        $D(C) = [\lfloor \frac{r_0}{t} \rceil]_t$
    return $D(C)$
```

Addition$(c[0], c[2])$:
The addition of two ciphertexts $c[1] = (c[1][0], c[1][1])$ and $c[2] = (c[2][0], c[2][1])$ is achieved by computing the addition of polynomials of ciphertexts with modulo reduction.
The result of addition is

$$c[1] + c[2] = ([c[1][0] + c[2][0]]_q, [c[1][1] + c[2][1]]_q) \quad (2)$$

The addition can be computed using Algorithm 4. The following equations shows that the addition of two ciphertexts is equal to the encryption of the addition of corresponding plaintexts.

$$[c[1][0]+c[2][0]]_q = ([p_0(u_1+u_2)+(e_1+e'_1)+\delta \cdot (m_1+m_2)]_q,$$

$$[c[1][1] + c[2][1]]_q = [p_1 \cdot (u_1 + u_2) + (e_2 + e'_2)]_q).$$

▶ ALGORITHM 4 Add ciphertexts

```
Input $c[1] = (c[1][0], c[1][1])$ and $c[2] = (c[2][0], c[2][1])$.
Output $S = c[1] + c[2]$.
    function ADDITION $(c[1], c[2])$
        $S[0] = c[1][0] + c[2][0]$
        $S[1] = c[1][1] + c[2][1]$
    return $S = (S[0], S[1])$.
```

Multiplication $(c[1], c[2])$**:**
The multiplication of two ciphertexts $c[1]$ and $c[2]$ consists in computing the tensor product of $c[1] = (c[1][0], c[1][1])$ and $c[2] = (c[2][0], c[2][1])$ and scaling by t/q. The multiplication will increase the ciphertext length.
We can simply define

$$(c[1][0], c[1][1]) \cdot (c[2][0], c[2][1]) = (ct_0; ct_1; ct_2)$$

where

$$ct_0 = [\lfloor \frac{t \cdot (c[1][0] \cdot c[2][0])}{q} \rceil]_q,$$

$$ct_1 = [\lfloor \frac{t \cdot (c[1][0] \cdot c[2][1] + c[1][1] \cdot c[2][0])}{q} \rceil]_q,$$

$$ct_2 = [\lfloor \frac{t \cdot (c[1][1] \cdot c[2][1])}{q} \rceil]_q.$$

The result of a multiplication is a triplet (ct_0, ct_1, ct_2), as illustrated in Algorithm 5. To transform this ciphertext of three elements (ct_0, ct_1, ct_2) into a ciphertext of two elements (ct'_0, ct'_1), we have to use the relinearization technique.

▶ ALGORITHM 5 Multiply ciphertexts

```
Input $c[1] = (c[1][0], c[1][1])$ and $c[2] = (c[2][0], c[2][1])$.
Output $M = c[1] \cdot c[2]$.
    function MULTIPLICATION $(c[1], c[2])$
        $ct_0 = t \cdot (c[1][0] \cdot c[2][0])$
        $r_0 = \frac{ct_0}{q}$
        $ct_0 = [\lfloor r_0 \rceil]_q$

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

        $ct_2 = t \cdot (c[1][1] \cdot c[2][1])$
        $r_0 = \frac{ct_2}{q}$
        $ct_2 = [\lfloor r_0 \rceil]_q$
    return $M = (ct_0, ct_1, ct_2)$
```

Relinearization$(params, rlk, (ct_0, ct_1, ct_2))$**:**
The relinearization step reduces the number of ciphertext elements by transforming a ciphertext of three elements to a ciphertext of two elements, we call the relinearization step after each multiplication. First we write ct_2 in the base T, $ct_2 = \sum_{i=0}^{l} c[2]^{(i)} T^i$ where $c[2]^{(i)} \in R_T$, and we use the relinearization key to compute the new ciphertext $(ct'_0; ct'_1)$.

$$ct'_0 = [ct_0 + \sum_{i=0}^{l} rlk[i][0] \cdot c[2](i)]_q,$$

$$ct'_1 = [ct_1 + \sum_{i=0}^{l} rlk[i][1] \cdot [2](i)]_q.$$

The noise of the ciphertext grows after the relinearization step and we can avoid this step if the circuit depth is one by keeping the ciphertext as (ct_0, ct_1, ct_2) and computing the decryption directly using this ciphertext $D(ct_0, ct_1, ct_2) = [\lfloor \frac{t \cdot [ct_0 + ct_1 \cdot s_k + ct_2 \cdot s_k^2]_q}{q} \rceil]_t$, we can compute also the addition of two ciphertexts of three elements, or the addition of a ciphertext of three elements and a ciphertext of two elements using the equation (3) and (4).

▶ ALGORITHM 6 Relinearize a ciphertext

```
Input $(ct_0, ct_1, ct_2)$.
Output $(ct'_0, ct'_1)$.
    function RELINEARIZATION $(rlk, (ct_0, ct_1, ct_2))$
        Write $ct_2$ in the base $T$.
        $ct_2 = \sum_{i=0}^{l} c[2]^{(i)} T^i$ where $c[2]^{(i)} \in R_T$ .
        $ct'_0 = [ct_0 + \sum_{i=0}^{l} rlk[i][0] \cdot c[2]^{(i)}]_q$
        $ct'_1 = [ct_1 + \sum_{i=0}^{l} rlk[i][1] \cdot c[2]^{(i)}]_q$
    return $(ct'_0, ct'_1)$
```

IV. PARAMETERS GENERATION FOR FV CRYPTOSYSTEM

In this section, we provide a naive method to generate parameters for the FV scheme. This method is intended for engineers who don't have the necessary mathematical background for understanding the FV cryptosystem. The security of parameters (λ, q, d, L, t) is based on RLWE problem, which is a difficult problem to solve. So let's start by explaining the RLWE problem.

A. RLWE problem

Ring Learning with Errors (RLWE)[19] is a very difficult problem to solve. This problem is used for the foundation of several homomorphic cryptosystems designed to withstand attacks by quantum computers.

The decision version of this problem is described in the mathematical ring formed by degree d polynomials over a finite field such as the integers modulo a prime number q.

Let $\varnothing(x)$ be a cyclotomic polynomial of degree d, and $q \geq 2$ a modulus depending on a security level λ. For a random $s \in R_q$ and a distribution $\chi = \chi\ (x)$ over R. The problem consists in distinguishing (a, $[a \cdot e + s]_q$) from a random pair sampled uniformly from $R_q * R_q$, where a is a random element of R_q and e a noise term from χ.

Solving the $RLWE_{q,\chi,m}$ problem is at least as hard as quantumly solving the $SVP\gamma$ problem on arbitrary ideal lattices in R, for some $\gamma = poly(n)/\alpha$.

B. Parameters

The generation parameters of FV cryptosystem consists in generating (λ, σ, q, d) using LWE estimator and deriving L and t to define all parameters $(\lambda, \sigma, q, d, L, t)$. In the next subsection we describe a method to verify the security of parameters using a SageMath module[21]. In a nutshell it consists in choosing some q and d using [4] and computing the noise of operations specially the noise of multiplication using[15], then we can derive the level L and t.

C. LWE etimator

It is a SageMath module[4] developed in 2015 by Martin Albrecht to estimate the concrete security of Learning with errors instances. It's an easy way to choose parameters resisting to known attacks. The designer should give the degree d of the cyclotomic polynomial $x^d + 1$ used to define the polynomial ring $Z[x]/x^d+1$, the coefficients modulus q of polynomials in $Z[x]/x^d+1$ and the error rate α, as inputs. The estimator will return the number of bits of operations to attack the parameters given, the memory requirements and the number of calls to the LWE oracle.
To verify the security of parameters we use the following algorithm.

▸ **ALGORITHM 7 LWE estimator**

```
Input $\lambda$, $d = 2^n$, $\alpha = \frac{8}{q}$ and $h$ where $(q = 2^h)$.
Output $\lambda, d, h$.
    function KeyGeneration ($\lambda$, $d = 2^n$, $\alpha = \frac{8}{q}$ and $h$)
        Run the LWE estimator to determine the best attack for those parameters.
        while (The best attack costs less than $\lambda$) do
            Decrease h.
        return $\lambda, d, \alpha$ and $h$.
```

Some practical parameters:

The main problem of homomorphic encryption is the huge size of ciphertexts, which causes a transmission and a storage challenge.

The size of the ciphertext is defined by d and q. Table I, Table II, Table III and Table IV show the impact of other parameters (λ and L) on d and q, for different plaintext modulus and security levels. The degree d and the modulus q depend on the security level λ and the error rate α. The circuit depth L depends on d, q and the plaintext modulus. Increasing λ for a given depth L (Table I and Table III) will increase d and q, and increasing the plaintext modulus for a given security level (Table I and Table II) will decrease L.

d: Degree of polynomial	q: modulus size of coefficients	L: depth of circuit
1024	29	0
2048	56	1
4096	110	2
8192	219	6
16384	441	11

// TABLE I: Parameters for a security 128 and a plaintext modulus of 16 bit

d: Degree of polynomial	q: modulus size of coefficients	L: depth of circuit
1024	29	0
2048	56	0
4096	110	1
8192	219	3
16384	441	4

// TABLE II: Parameters for a security 128 and a plaintext modulus of 30 bit

d: Degree of polynomial	q: modulus size of coefficients	L: depth of circuit
2048	39	0
4096	77	1
8192	153	4

// TABLE III: Parameters for a security 192 and a plaintext modulus of 16 bit

d: Degree of polynomial	q: modulus size of coefficients	L: depth of circuit
2048	39	0
4096	77	0
8192	153	1

// TABLE IV: Parameters for a security 192 and a plaintext modulus of 30 bit

V. EXISTING IMPLEMENTATIONS

A. FV-NFLlib library

FV-NFLlib[20] is a software library implementing FV cryptosystem, developed in 2016 in C++, and based on NFLlib[17], a library specially designed for ideal lattices cryptography.

B. SEAL library

SEAL[14] is an Open source library, developed in 2015 in C++ and C# by a Microsoft team. It implements FV cryptosystem with no external library dependencies.

In SEAL, the user chooses the security level: 128 or 192, the plaintext modulus (no limit for the choice) and the degree: 1024, 2048, 4096, 8192 or 16384. The library returns the coefficients modulus size and the noise budget to know if we can make another computation or not.

C. SEAL-RNS library

RNS version of FV: The RNS variant[5] of FV cryptosystem is an improvement of the FV scheme, which enables RNS representation of coefficients. It represents each element of the ring R_q as a vector of several elements in R_{qi} , where q_i are small modulus. This technique allows the optimization of coefficients storage and the acceleration of computations.

The RNS representation consists in the composition of the coefficients modulus q into a product of multiple small co-prime modulus.

Let q be a product of q_i, $q = q_1\ q_2 * ... * q_n$, and c_k be the coefficient of index k of a polynomial in R_q. Using the CRT theorem c_k can be represented as below:

$$c_k = \begin{cases} c'_i & \text{mod} \quad q_i \\ & \quad\quad ... \\ c'_n & \text{mod} \quad q_n \end{cases}$$

SEAL-RNS library: This version of SEAL[13] implements the FullRNS[5] variant of FV cryptosystem. In this version the coefficients modulus are replaced with several small modulus, in order to accelerate computations and to optimize the storage of coefficients. It uses the same parameters as the initial version of SEAL.

D. Comparison of libraries

In SEAL and SEAL-RNS, the decryption of a multiplication result can be done by two different methods. The standard method, where we multiply two ciphertexts, we relinearize and decrypt, and the second method which consists in decrypting directly the result of a multiplication. Let $(c[0], c[1], c[2])$ be the product of two ciphertexts and s a secret key, to decrypt this ciphertexts product compute $c[0]+c[1]\cdot s+c[2]\cdot s^2$. The first method is used when the circuit depth is one and the second method is used when we need to evaluate a circuit of more than one multiplication.

We note that FV-NFLIB library is faster than SEAL libraries, but FV-NFLIB does not support circuits with high levels, the maximum level supported by this library is 6. Thus we recommend to use FV-NFLIB for small circuits, and to use SEAL to evaluate large circuits.

For our use case, we need to use the FV cryptosystem in order to protect data and perform computations on encrypted data. In practice, one of those libraries will be embedded inside the gateway and the cloud, but we have to be careful about the code size to accomodate limited ressources in the gateway and to facilitate security audits.

VI. OUR IMPLEMENTATION

A. Setup:

We implement the FV cryptosystem in C language to be embedded in IoT modules, in particular the gateway. This implementation is portable, self-contained and independent from other libraries. It can be considered as a proof of concept where we only use the standard mathematics library *math.h* without SIMD-parallelism and multi-core processing.

We use a set of practical parameters, to enable the evaluation of circuits of one level, a security level of 128 bits, a degree 2048 for the cyclotomic polynomial and 56 bits for the polynomials coefficients. We choose those parameters because they are the smallest that enable to do one multiplication. The scheme enables 1 level of multiplication and encrypts plaintexts of 16 bits.

The code is embedded and run under an administrator machine, which holds an Intel® Core™ i5 processor at 2.4 GHz, a gateway holding an Intel Atom® processor at 1.91 GHz, and a virtual machine using an Intel® Xeon® processor at 2.40 GHz in a public cloud.

B. Description:

We implement our own code from scratch in order to get a homogeneous implementation, and portable in the gateway. In this implementation, polynomials coefficients are computed modulo q that is stored in a long long type capable of containing 64 bits. If q is a power of 2 $q = 2^{56}$, computing modulo q corresponds to an AND operation. Let's N and k be two integers where $q = 2^k$, the modulus operation verify the following equation: N mod $2^k = N \,\&\, 2^k - 1$.

In our code, we developed our own 128-bit arithmetic in order to enable the storage of the result of two long long multiplication. Let A and B be two long long, $A = A_1/A_2$ and $B = B_1/B_2$ where A_1, A_2, B_1, B_2 are integers of 32 bits. To compute the product of A and B we use the algorithm 8.

▶ ALGORITHM 8 128-bit arithmetic

```
Input $A = A_1|A_2$ and $B = B_1|B_2$.
Output $A \cdot B$.
    function PRODUCT OF TWO LONG LONG $(A, B)$
        Compute $A_2 \cdot B_2 = r_1|r_2$.
        Compute $A_1 \cdot B_2 = r_3|r_4$.
        Compute $A_2 \cdot B_1 = r_5|r_6$.
        Compute $A_1 \cdot B_1 = r_7|r_8$.
        compute $S_1 = r_1 * 2^{32} + r_4 + r_6$.
7:      compute $M_1 = (r_1 * 2^{32} + r_4 + r_6) \ mod \ 2^{32}$.
        compute $S_2 = r_4 * 2^{32} + r_5 * 2^{32} + r_8 + M_1$.
        compute $M_2 = (r_4 * 2^3 + r_5 * 2^{32} + r_8 + M_1) \bmod 2^{32}$.
        compute $S_3 = r_7 * 2^3 + M_2$.
    return $A \cdot B = S_3|S_2|S_1|r_2$.
```

In our implementation we don't use the relinearization step because we need to ensure only one multiplication depth. In other words, the result of ciphertexts multiplication is a ciphertext of three elements, and we can do additions between ciphertexts of three elements or between a ciphertext of three elements and a ciphertext of two elements. If we use relinearization, we cannot perform even one depth with the chosen parameters, because the noise of the ciphertext will be huge after this step due to the multiplication by the relinearization key and we cannot decrypt correctly. Therefore, we need to increase parameters, and then, the polynomials coefficients cannot be stored in a long long structure.

C. Algorithm:

We implement the FV algorithm described below without relinearization, we add some functions to decrypt a ciphertext of three elements and to do addition between ciphertexts.

Decrypt$(s_k, c[1], c[2], c[3])$: This function computes the decryption of a ciphertext resulting from a multiplication and return

$$D(c[1], c[2], c[3]) = [\lfloor \frac{t \cdot [c[1] + c[2] \cdot s_k + c[3] \cdot s_k^2]_q}{q} \rceil]_t,$$

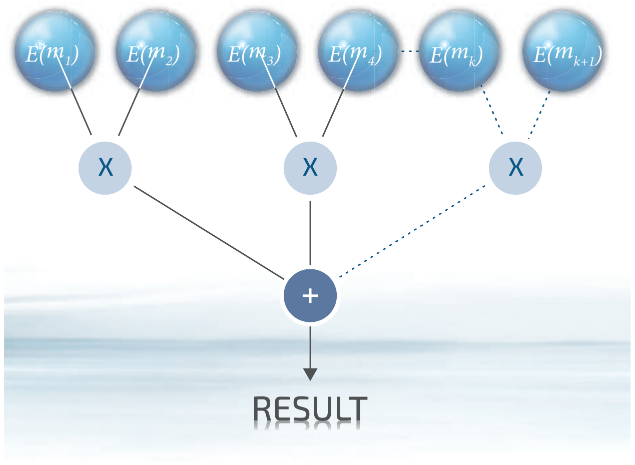
Addition $(c[1], c[2], c[3], c[1]', c[2]')$: This function does the addition of a ciphertext of three elements $(c[1], c[2], c[3])$ and a normal ciphertext $(c[1]', c[2]')$:

$$(c[1], c[2], c[3]) + (c[1]', c[2]') = (c[1] + c[1]', c[2] + c[2]', c[3]). \quad (3)$$

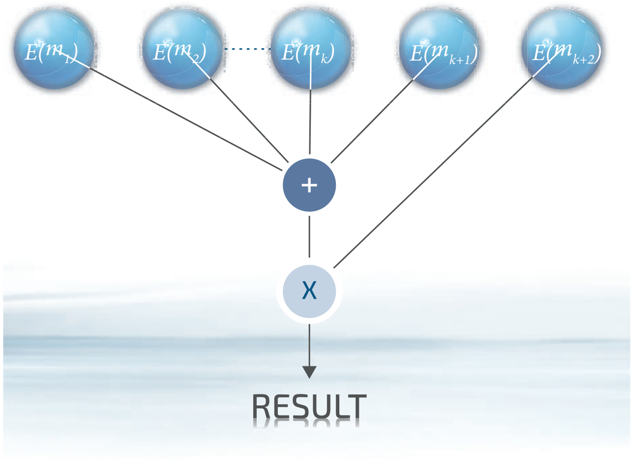
Addition $(c[1], c[2], c[3], c[1]', c[2]', c[3]')$: This function does the addition of ciphertexts of three elements.

$$(c[1], c[2], c[3]) + (c[1]', c[2]', c[3]') = (c[1] + c[1]', c[2] + c[2]', c[3] +, c[3]'). \quad (4)$$

Our implementation can perform all the circuits that can be described as in Figure 2 and 3. We describe the algorithms in such a way that we minimize the number of internal registers. A practical use case of this circuit (Figure 2) is matrix product, which is used for example in image processing to do color transformation, modifying saturation and changing brightness. The circuit of Figure 3 can be used to compute average.

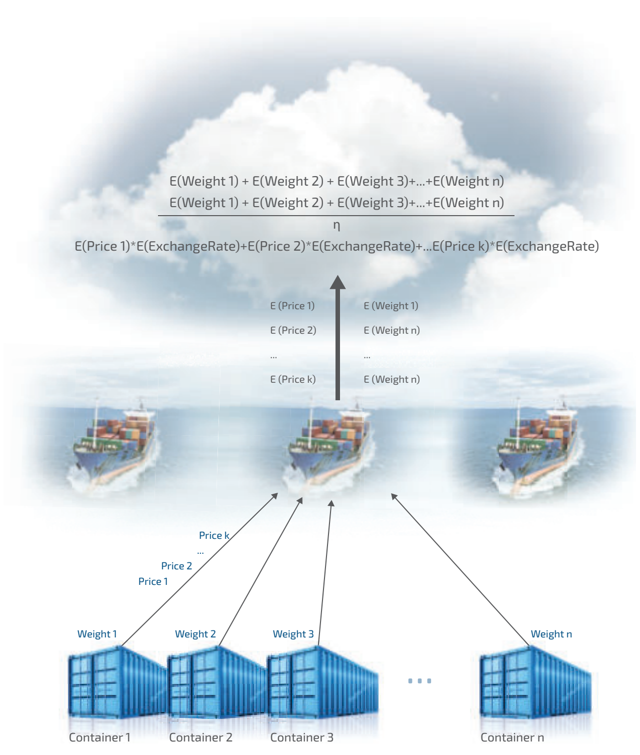


// Figure 2: A computation of weighted average



// Figure 3: Example of possible computation

For our use cases, we need to compute the sum of containers in the seaport and the sum of the containers weights before loading them on the ship, the circuit of Figure 2 is adapted to this use case, also the circuit of Figure 3 can be used to compute currency conversion. The figure 4 illustrates this use cases, it shows that shipowners can do currency conversion of goods from Euro to Yen for example, compute the weight of containers, and the average weight of containers, by computing the sum of weights and dividing by the number of containers, the division can be written as a multiplication by $\frac{1}{sum}$.


E(Weight 1) + E(Weight 2) + E(Weight 3)+...+E(Weight n)
E(Weight 1) + E(Weight 2) + E(Weight 3)+...+E(Weight n)
η
E(Price 1)*E(ExchangeRate)+E(Price 2)*E(ExchangeRate)+...E(Price k)*E(ExchangeRate)
E (Price 1)
E (Weight 1)
E (Price 2)
E (Weight n)
...
...
E (Price k)
E (Weight n)
Price k
...
Price 2
Price 1
Weight 1
Weight 2
Weight 3
Weight n
Container 1
Container 2
Container 3
Container n

// Figure 4: Application of the homomorphic encryption in the Seaport

D. Performances:

In this paragraph, we present the performances of our implementation under different platforms. The same program is embedded in the gateway, the cloud and the administrator machine, we add some options to specify which function of the code we want to run. In the gateway (Intel Atom® processor at 1.91 GHz), we run the GenerateKey and Encrypt functions. In the public cloud (Intel® Xeon® processor at 2.40 GHz), we run the addition and multiplication functions, and the decryption function is run in the administrator machine (Intel® Core™ i5 processor at 2.4 GHz). Let's start with the size of ciphertexts and keys for our implementation which handles 16 bits of plaintext. The ciphertext size is 32 kbytes, the secret key size is 2048 bits and the public key size is 32 kbytes. The timings of the execution of all the functions inside the different modules of our use case are described in Table V. Our results prove that we can use homomorphic encryption for a real use case. The timing could probably be improved, but our first goal was to deploy a compact home-made implementation of the FV cryptosystem between different platforms and a private cloud.

Operation	Inside the administrator machine	Inside the cloud	Inside the gateway
Time to generate the secret key(ms)	0.041	-	0.151
Generation of the public key(ms)	56.59	-	175.946
Encryption(ms)	69.52	-	233.007
Decryption of a normal ciphertext (ms)	6.52	-	-
Decryption of a ciphertext with 3 elements(ms)	62.177	-	-
Addition(ms)	0.031	0.022	-
Multiplication(ms)	146.704	203.626	-

// TABLE V: The execution time of functions

VII. CONCLUSION

In this work, we provide a practical and self-contained implementation of the FV cryptosystem in C language to apply homomorphic encryption in real life. This implementation has been run under a gateway and a cloud, allowing to evaluate circuits of one level and ensuring a security of 128 bits. Shipowners can use this implementation in the Seaport to encrypt data of containers in the gateways and compute inside the cloud, the weighted sum of containers, the weight of containers before loading them on the ship, or to do currency conversion of goods. Using our implementation, shipowners can manipulate encrypted data while respecting the competition between companies and without impacting the security, the privacy and the anonymization.

▶ For more information please contact
support.KFR@kontron.com

REFERENCES

[1] Internet of things (IoT).
https://internetofthingsagenda.techtarget.com/definition/Internet-of-Things-IoT.

[2] MQTT (MQ Telemetry Transport).
https://internetofthingsagenda. techtarget.com/definition/MQTT-MQ-Telemetry-Transport.

[3] what is lora?
https://www.semtech.com/lora/what-is-lora.

[4] Martin R. Albrecht, Rachel Player, and Sam Scott.
On the concrete hardness of learning with errors. *J. Mathematical Cryptology*, 9(3): 169–203, 2015.

[5] Jean-Claude Bajard, Julien Eynard, M. Anwar Hasan, and Vincent Zucca.
A full RNS variant of FV like somewhat homomorphic encryption schemes. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers, volume 10532 of Lecture Notes in Computer Science, pages* 423–442. Springer, 2016.

[6] Joppe W. Bos, Kristin E. Lauter, Jake Loftus, and Michael Naehrig.
Improved security for a ring-based fully homomorphic encryption scheme. In Martijn Stam, editor, *Cryptography and Coding - 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings, volume 8308 of Lecture Notes in Computer Science, pages* 45–64. Springer, 2013.

[7] Zvika Brakerski.
Fully homomorphic encryption without modulus switching from classical gapsvp. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings, volume 7417 of Lecture Notes in Computer Science, pages* 868–886. Springer, 2012.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan.
(leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabach`ene.
Faster Fully Homomorphic Encryption: Bootstrapping in Less than 0.1 Seconds. *Asiacrypt 2016*, 10031:3–33, 2016.

[10] Junfeng Fan and Frederik Vercauteren.
Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, 2012:144, 2012.

[11] Craig Gentry.
A fully homomorphic encryption scheme. phd thesis, stanford university. 2009.

[12] Craig Gentry, Amit Sahai, and Brent Waters.
Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology - CRYPTO 2013 - 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I, volume 8042 of Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[13] Zhicong Huang Amir Jalali Hao Chen, Kyoohyung Han and Kim Laine.
Simple encrypted arithmetic library - seal (v2.3.0). Online.

[14] Hao Chen Kim Laine and Rachel Player.
Simple encrypted arithmetic library - seal (v2.2).

[15] Kim Laine and Rachel Player.
Simple encrypted arithmetic library - seal (v2.0).

[16] Vadim Lyubashevsky, Chris Peikert, and Oded Regev.
On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[17] Carlos Aguilar Melchor, Joris Barrier, Serge Guelton, Adrien Guinet, Marc-Olivier Killijian, and Tancr`ede Lepoint.
Nfllib: Ntt-based fast lattice library. In Kazue Sako, editor, *Topics in Cryptology - CT-RSA 2016 - The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science*, pages 341–356. Springer, 2016.

[18] Carlos Aguilar Melchor, Guilhem Castagnos, and Philippe Gaborit.
Lattice-based homomorphic encryption of vector spaces. In Frank R. Kschischang and En-Hui Yang, editors, *2008 IEEE International Symposium on Information Theory, ISIT 2008, Toronto, ON, Canada, July 6-11, 2008*, pages 1858–1862. IEEE, 2008.

[19] Oded Regev.
The learning with errors problem (invited survey). In *Proceedings of the 25th Annual IEEE Conference on Computational Complexity, CCC 2010, Cambridge, Massachusetts, USA, June 9-12, 2010*, pages 191–204. IEEE Computer Society, 2010.

[20] Victor Shoup Shai Halevi.
Design and implementation of a homomorphic-encryption library.
https://researcher.watson.ibm.com/researcher/files/us-shaih/helibrary. pdf.

[21] The Sage Developers.
SageMath, the Sage Mathematics Software System (Version 8.1), 2017. https://www.sagemath.org.

[22] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan.
Fully homomorphic encryption over the integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings, volume 6110 of Lecture Notes in Computer Science*, pages 24–43. Springer, 2010.

About Kontron – Member of the S&T Group

Kontron is a global leader in IoT/Embedded Computing Technology (ECT). As a part of technology group S&T, Kontron, together with its sister company S&T Technologies, offers a combined portfolio of secure hardware, middleware and services for Internet of Things (IoT) and Industry 4.0 applications. With its standard products and tailor-made solutions based on highly reliable state-of-the-art embedded technologies, Kontron provides secure and innovative applications for a variety of industries. As a result, customers benefit from accelerated time-to-market, reduced total cost of ownership, product longevity and the best fully integrated applications overall.

For more information, please visit: **www.kontron.com**

About the Intel® Internet of Things Solutions Alliance

From modular components to market-ready systems, Intel and the 400+ global member companies of the Intel® Internet of Things Solutions Alliance provide scalable, interoperable solutions that accelerate deployment of intelligent devices and end-to-end analytics. Close collaboration with Intel and each other enables Alliance members to innovate with the latest IoT technologies, helping developers deliver first-in-market solutions.

Intel and Atom are registered trademarks of Intel Corporation in the U.S. and other countries.

IoT Solutions Alliance
Premier

GLOBAL HEADQUARTERS

KONTRON S&T AG

Lise-Meitner-Str. 3-5
86156 Augsburg, Germany
Tel.: + 49 821 4086-0
Fax: + 49 821 4086-111
info@kontron.com

www.kontron.com