24th International Congress on Modelling and Simulation, Sydney, NSW, Australia, 5 to 10 December 2021
mssanz.org.au/modsim2021

Implementing a Fully Homomorphic Encrypted Circuit in Workspace

Dang Nguyen [a] **and Damien Watkins** [a]

[a]*Data61 CSIRO, Research Way, Clayton, VIC 3168, Australia*

Email: dang-quan.nguyen@data61.csiro.au

Abstract: Fully homomorphic encryption (FHE) schemes are regularly referred to as solutions for both privacy and security aspects in cloud computing. They allow a user's data to be encrypted in such a way that the cloud provider can still blindly manipulate the encrypted data on the user's behalf, i.e. by running some computations (called "circuit") on the encrypted data. Since its discovery a decade ago, FHE has become an active field of research with many encryption schemes proposed. When compared to non-FHE cryptographic functions, most of the FHE schemes require much larger memory space and longer execution time for their basic operations such as encrypt, decrypt and recrypt. This makes designing and experimenting with FHE circuits that have more than a few basic operations a very challenging task.

We present in this paper an implementation of Gentry's lattice-based FHE scheme as a plugin in Workspace. Workspace is a world-leading workflow software that allows to implement computational functions as operations then to reuse and rearrange them in a workflow to achieve larger and more complex functions. The operations' input and output data can be easily "wired" between the operations and Workspace automatically takes care of referencing the data, thus avoiding unnecessarily copying it between functions. The operations themselves are executed on a need-to-run basis, i.e. only when their outputs are required by some other executing operations and their inputs have been changed since their last execution. This can lead to a significant improvement in terms of both execution time and memory space required if the transitional data is large as in many FHE schemes. We experiment with this implementation in Workspace by creating a few basic circuits such as the obfuscated `if ... then ... else ...` statement and the $\texttt{MIN}(a, b)$ and $\texttt{MAX}(a, b)$ functions on encrypted data.

Keywords: *Workspace, workflow, homomorphic encryption, FHE*

1 INTRODUCTION

A fully homomorphic encryption (FHE) scheme is one that allows to perform arbitrary computations on the encrypted data (ciphertext) without compromising the confidentiality of the data or the validity of the computations. FHE has many potential applications as it solves the problem of trust computing. The owner of the data can encrypt it and safely delegates the data processing task to a potentially untrustworthy third-party to run on the encrypted data. The third-party can process the data without being able to comprehend it. Only the rightful owner of the data who has the private key can decrypt the result of the processing task. Healthcare data processing and cloud computing are two immediate examples of FHE applications.

The concept of homomorphic encryption was introduced shortly after RSA was invented. It was noted that RSA is multiplicatively homomorphic: since the ciphertext is the cleartext raised to a fixed exponent (the public key), multiplying two ciphertexts together is the same as multiplying the two cleartexts and raised the product to the same exponent. Over the coming years, a number of cryptosystems are known to be either additively homomorphic (e.g. Paillier) or multiplicatively homomorphic (e.g. ElGamal) but not both. Many candidates to a fully-homomorphic encryption scheme have been shown to be insecure.

Three decades after the concept was introduced, Gentry published in his thesis Gentry [2009] in 2009 the first provably secure FHE scheme. The breakthrough of Gentry's scheme is in making a "weaker" somewhat homomorphic scheme able to homomorphically compute its own decryption function.

In 2011 Gentry and Halevi published a paper on an implementation of Gentry's FHE scheme based on ideal lattices Gentry and Halevi [2011]. The idea of using ideal lattices as ciphertext spaces is similar to one in an implementation by Smart and Vercauteren published in 2010 Smart and Vercauteren [2010]. Smart and Vercauteren's implementation was not able to support large enough parameters to become fully homomorphic. Gentry and Halevi's implementation introduced many optimisations and innovating techniques that made it capable of homomorphically evaluate its own decryption function. Hence it is considered the first fully-homomorphic encryption implementation.

There have been many successful implementations of FHE schemes over the years: NTRU-based, GSW, CKKS schemes. They bring various optimisations to, or either replace or eliminate some procedures of Gentry and Halevi's original implementation.

Our implementation presented in this paper follows Gentry and Halevi's original implementation in Gentry and Halevi [2011] as it allows for a comprehensive understanding of the challenges and optimisation techniques relevant to the early FHE schemes.

Recently a team at Google led by Shruthi Gorantala has developed a FHE transpiler Gorantala et al. [2021]; Google [2021]. It can convert a program written in C++ into a program in FHE-C++ that runs on encrypted inputs. The transpiler works independently of the underlying FHE scheme implementation. This separation is also reflected in our choice of implementing FHE in two separate components: a backend component that performs the basic cryptographic functions based on Gentry's ideal lattices and a frontend component that offers a user-friendly GUI to create homomorphic circuits on encrypted data.

We implement the frontend component as a CSIRO Workspace plugin. Workspace CSIRO [2021] is a powerful scientific workflow framework that lets users implement their own functions as Workspace operations. The users can then drag and drop those operations onto a canvas and connect their inputs and outputs to create a more complex workflow. Workspace optimises the execution of workflows by running or re-running the operations on a need-to-run basis, i.e. only when their outputs are required by some other executing operations and their inputs have been changed since their last execution. Workspace automatically references data so to avoid unnecessary copying large data between operations.

The rest of this paper is organised as follows. The details of our implementation is presented in Section 2 where we present the backend component and discuss various domain parameters. We also present our implementation of the frontend component and give some examples of homomophic circuits in Workspace. We conclude the paper in Section 3.

2 IMPLEMENTATION

We implement Gentry's fully-homomorphic encryption scheme as it is described in Gentry and Halevi [2011]. We add some extra functionalities such as homomorphic addition, substraction and multiplication of whole bytes. We also implement a homomorphic equivalence of the `if ...then ...else ...` statement and the `MIN`(a, b) and `MAX`(a, b) functions where a, b are bytes.

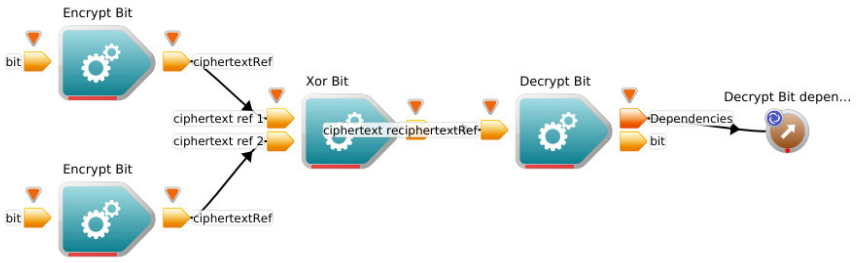

Figure 1. A circuit encrypting two bits, homomorphically xor'ing the ciphertexts then decrypting the result

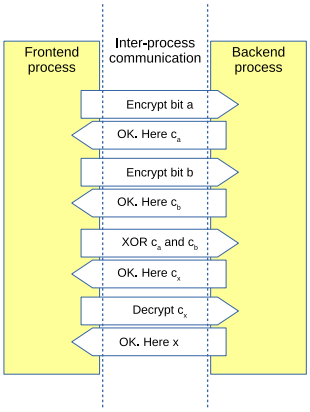

Figure 2. Communications between frontend and backend processes

The implementation uses C++. It is divided into two parts: the backend and the frontend. The backend is a collection of basic functions such as key generations, encrypt and decrypt bits, logical gates on bits, etc. The backend is open-source, released under LGPL v2.1 and available at Nguyen [2021].

The frontend is developed as a plugin for the CSIRO Workspace software. It aims at providing a user-friendly interface to create customised homomorphic circuits on encrypted data. The users can therefore use Workspace with this plugin to drag and drop operations representing the basic functions of the backend component onto the Workspace canvas, to connect the operations and form a complete homomorphic circuit. Figure 1 shows such a circuit that:

1. encrypts two bits a, b,
2. homomorphically XOR the encrypted data,
3. decrypt the ciphertext so the users can check the resulting bit is the equal $a \oplus b$.

The collection of operations in Figure 1 is called a workflow in Workspace. The users can tell Workspace to execute a workflow which will then run the homomorphic circuit contained in it.

Behind the scene, what actually happens is that the frontend and the backend are two separate processes. The frontend gives the order to the backend to run the corresponding basic functions and receives the result when they finish. The communication is done via some asynchronous inter-process communication APIs and is illustrated in Figure 2.

2.1 FHE backend

Our FHE backend component is implemented as a standalone application. It uses Shoup's NTL library Shoup [2021] to perform modular arithmetic on large numbers and high degree polynomials. The backend source code is available at Nguyen [2021] under the LGPL v2.1 license.

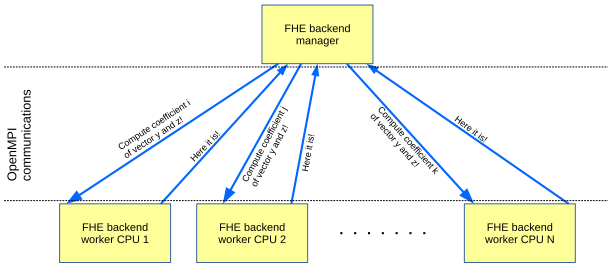

Figure 3. Paralleled calculation of vectors y, z using OpenMPI

The backend component lets the users set all domain parameters as described in Gentry and Halevi [2011]. In particular, we ran tests with lattices of dimensions $n = 512, 1024, 2048$ and with $f_n(x) = x^n + 1$. The vector coefficients are 380-bit integers. That is, every vector is an array of n numbers, each number is $N = 380$ bits.

Based on existing works of attacking the BDD problem, Gentry estimates that setting $n = 2048$ would offer a security guarantee of at least equivalent to that of RSA-1024, i.e. with a scheme complexity of 2^{80} on the lattice.

Another type of attack is the birthday attack (collision attack) on the coefficients of the noise vector r. It is known that if r has l bits of entropy then the birthday attack can recover the noise in $2^{l/2}$ time. Since we would like to keep the noise vector small on a new ciphertext, this leads to a tradeoff on choosing the number p of nonzero coefficients for r. Therefore with a noise vector having $p \log \binom{n}{p}$ bits of entropy it is unlikely that a collision occurs in time less than 2^{80} (equivalent to RSA-1024 security guarantee) if $p \geq 15$. In our implementation we choose random noise vectors having between 15 and 20 nonzero coefficients, in line with Gentry's recommendation.

We also implement Gentry's polynomial inverting algorithm that runs in time $O(N)$ compared to $O(nN)$ by FFT. This algorithm works because of the special form of the divisor $f_n(x) = x^n + 1$.

The procedures to compute public-private key pairs, encryption and decryption are relatively fast. They are done in under a few milliseconds on a single 2.5GHz CPU. The recryption Gentry and Halevi [2011] is by far the most time-consuming procedure. Recryption is done in two steps:

1. Computes two helper vectors y, z as described in the Squashing the Decryption Procedure in Gentry and Halevi [2011].
2. Evaluate homomorphically the decryption procedure using the pre-computed vectors y, z.

When running on a single 2.5GHz CPU core, a recryption of just one bit can take up to 36 minutes with a 512-dimensional lattice. About 80% of the time is spent in step 1. Step 2 takes only 20% of that time.

Fortunately, the computations of each coefficient of y, z are independent of each other. So we implement them in parallel with the OpenMPI library. This allows to significantly shorten the run time for both step 1 and 2. With this implementation being run on a 4-core 2.5GHz CPU with 8 hyperthreads, the recryption time is brought down to less than 10 seconds. We do not use any GPU acceleration capabilities, all modular arithmetic algorithms are implemented with software capabilities. Figure 3 shows the paralleled calculation of vectors y, z using OpenMPI.

The following basic cryptographic functions are implemented in the backend components:

- Public-private key generation.
- Encrypt, decrypt and recrypt one bit.
- Homomorphic `AND`, `XOR` two bits and `FLIP` one bit.
- Homomorphic `CHOOSE` one bit out of two bits.
- Encrypt and decrypt one byte.

Figure 4. A homomorphic logical NAND gate

- Homomorphic `ADD`, `SUB`stract and `MULT`iply two bytes.
- Homomorphic `MIN` and `MAX` of two bytes.

The `CHOOSE` function is a homomorphic equivalence of the `if ... then ... else ...` statement. Specifically, given three booleans a, b, c, the statement

$$\texttt{if } c \texttt{ then return } a \texttt{ else return } b$$

can be rewritten as

$$(c \texttt{ and } a) \texttt{ or } ((\texttt{not } c) \texttt{ and } b)$$

So if ψ_a, ψ_b, ψ_c are the ciphertexts of a, b, c then the above statement can be written homomophically as

$$\texttt{CHOOSE}(\psi_c, \psi_a, \psi_b) = (\psi_c \texttt{ AND } \psi_a) \texttt{ XOR } ((\texttt{FLIP } \psi_c) \texttt{ AND } \psi_b)$$

The homomorphic function `CHOOSE` illustrates an elementary branching in a homomorphic program. Its result is a new ciphertext that is decrypted to either a or b depending on the value of c but remains a secret to whomever is tasked with running that program.

The homomorphic `MIN` (or similarly `MAX`) function proceeds by homomorphically `SUB`stracting the two ciphertexts that represent the two bytes in cleartext. The function then `CHOOSE` the corresponding ciphertext depending on the (ciphertext of the) carrier bit resulting from the `SUB` function.

2.2 FHE frontend

We develop the frontend component to offer a user-friendly GUI to experiment with the basic cryptographic functions of the backend. In the GUI those functions are represented by operations that a user can drag and drop onto a canvas, then connect them together to form another homomorphic circuit. For example, the output of operation `AND` can be connected to the input of operation `FLIP` to form a homomorphic equivalence of the logical gate `NAND` as shown in Figure 4.

The frontend is implemented as a plugin in the CSIRO Workspace software.

CSIRO Workspace . Workspace CSIRO [2021] is a world-leading scientific workflow framework that allows to implement computational functions as operations then to reuse and rearrange them in a workflow to achieve more complex functions. The operations' input and output data can be easily "wired" between the operations and Workspace automatically takes care of referencing the data, thus avoiding unnecessarily copying it between functions. The operations themselves are executed on a need-to-run basis, i.e. only when their outputs are required by some other executing operations and their inputs have been changed since their last execution. This can lead to a significant improvement in terms of both execution time and memory space required if the transitional data is large as in many FHE schemes.

Workspace itself is developed in C++ on the Qt framework. It offers many complex and useful builtin capabilities such as running Python scripts within a workflow, executing operations in parallel, profiling the run time of operations or blocks of code individually. It also lets the users generate complete GUI applications based on any workflow. CSIRO [2021] contains a page listing some applications built on Workspace by teams from both inside and outside CSIRO.

Workspace FHE plugin . We develop the Workspace FHE plugin as a collection of Workspace operations. Each operation runs a basic cryptographic function of the backend component with the operation inputs mapped to the function inputs and the function outputs are mapped back to the operation outputs.

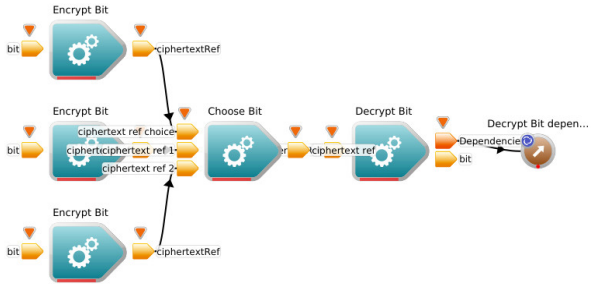

Figure 5. A homomorphic CHOOSE function of the if . . . then . . . else . . . statement

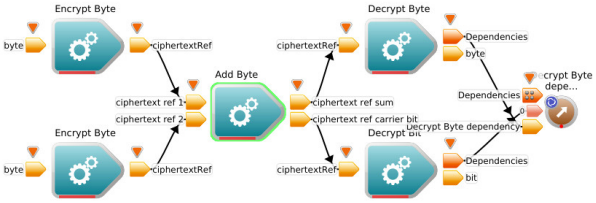

Figure 6. A homomorphic ADD function adding two bytes homomorphically

When Workspace executes an operation in the FHE plugin, the operation sends a command to the backend component via inter-process communication to request the execution of the corresponding function. When the backend function finishes its execution, its status and results are sent back to the frontend operation on the same communication channel. This communication is therefore asynchronous. The executing operation does not block Workspace while the backend function is being run. Subject to CPU resource availability, multiple operations can then be executed in parallel. Note that we did not use this high-level parallelisation to implement the parallel computations described in the backend, where the low-level implementation using OpenMPI is more suitable.

Figures 5, 6 and 7 illustrate three workflows using these operations.

The workflow in Figure 5 first encrypts three bits, their ciphertexts are given as inputs to the **Choose Bit** operation that will run the corresponding `CHOOSE` function in the backend. The resulting bit is then given as input to the **Decrypt Bit** operation. The final result is the output bit of that **Decrypt Bit** operation which is equal to second bit (from the top of the leftmost three bits in input) if the first bit is 1, or equal to the third bit if the first bit is 0.

Figure 6 shows a workflow that encrypts two bytes, homomorphically adds them up and then decrypts the result so that the user can check the validity of the operations. This time we note that the operation **Add Byte** has two outputs: the ciphertext sum and the ciphertext carrier bit because adding two bytes can result in a sum that exceeds 8 bits.

Figure 7 shows a workflow that encrypts two bytes, homomorphically sorts them in decreasing order by outputing the bigger byte to the top output and the smaller byte to the bottom output. The workflow then

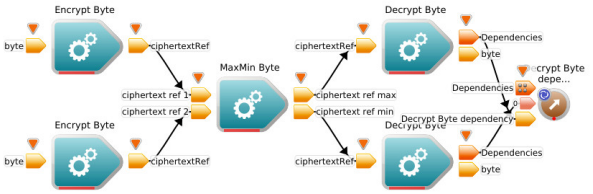

Figure 7. A homomorphic MAX MIN function sorting two bytes homomorphically

decrypts the outputs so the user can check the results.

3 CONCLUSION

We present in this paper our implementation of Gentry's fully homomorphic encryption scheme based on ideal lattices. The implementation is composed of a backend that performs the basic cryptographic functions and of a GUI frontend that offers a user-friendly interface developed on the CSIRO Workspace scientific workflow framework. Our implementation incorporates all optimisations suggested by Gentry and take advantage of multicore CPUs to significantly speed up the recryption procedure.

This implementation aims at providing some tools for experimenting with fully homomorphic encryption schemes. The users can drag and drop operations onto a canvas and connect them up to form different homomorphic circuits, check their validity and measure run time. The basic cryptographic functions in the backend can be reimplemented with another scheme without affecting the frontend logics.

In future works we may explore the possibility of implementing fully-homomorphic encryption on Edwards-Bernstein curves Bernstein and Lange [2007]. Since non-singular elliptic curves are not rational (see Perrin [2001]) we cannot apply the nice point-wise addition formulas for Edwards-Bernstein curves on the parameterisations as we would otherwise for unicursal curves. However we would like to know if it is possible to take a curve C that has some singular points (possibly at infinity, note that on Edwards-Bernstein curves those points are all finite Nguyen [2017]) so that its genus $g(C) = 0$, making it unicursal, then check if the point-wise addition formula can be made into a homomorphism wherever it is defined on C.

REFERENCES

Bernstein, D. J. and Lange, T. [2007], Faster addition and doubling on elliptic curves, *in* K. Kurosawa, ed., 'Advances in Cryptology – ASIACRYPT 2007', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 29–50.

CSIRO, D. [2021], 'Workspace', `https://research.csiro.au/workspace`. Accessed on 2021-09-02.

Gentry, C. [2009], A Fully Homomorphic Encryption Scheme, PhD thesis, Stanford University.

Gentry, C. and Halevi, S. [2011], Implementing gentry's fully-homomorphic encryption scheme, *in* K. G. Paterson, ed., 'Advances in Cryptology – EUROCRYPT 2011', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 129–148.

Google [2021], 'Fully homomorphic encryption (fhe)', `https://github.com/google/fully-homomorphic-encryption`. Accessed on 2021-08-25.

Gorantala, S., Springer, R., Purser-Haskell, S., Lam, W., Wilson, R., Ali, A., Astor, E. P., Zukerman, I., Ruth, S., Dibak, C., Schoppmann, P., Kulankhina, S., Forget, A., Marn, D., Tew, C., Misoczki, R., Guillen, B., Ye, X., Kraft, D., Desfontaines, D., Krishnamurthy, A., Guevara, M., Perera, I. M., Sushko, Y. and Gipson, B. [2021], 'A general purpose transpiler for fully homomorphic encryption', Cryptology ePrint Archive, Report 2021/811. `https://ia.cr/2021/811`.

Nguyen, D. [2017], Correspondence between elliptic curves in edwards-bernstein and weierstrass forms, Report, University of Ottawa.

Nguyen, D. [2021], 'Fully homomorphic encryption (fhe) library', `https://github.com/themanitou/fully-homomorphic-encryption`. Accessed on 2021-09-01.

Perrin, D. [2001], *Géométrie algébrique - Une introduction*, EDP Scences/CNRS Editions.

Shoup, V. [2021], 'Ntl: A library for doing number theory', `https://libntl.org`. Accessed on 2021-09-02.

Smart, N. P. and Vercauteren, F. [2010], Fully homomorphic encryption with relatively small key and ciphertext sizes, *in* P. Q. Nguyen and D. Pointcheval, eds, 'Public Key Cryptography – PKC 2010', Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 420–443.