”Performing Operations on Encrypted Data using Homomorphic Encryption”

Shamili Tetali
G# 01225749
ECE 646 Applied Cryptography-Fall 2021
stetali@gmu.edu

I. Abstract

Although Homomorphic encryption (HE) is a new domain of cryptography, it has come long way demonstrating the enhanced mode of information protection by allowing arbitrary computations on encrypted data. With the rapid transformation of IT practices into cloud-based solutions the need for FHE like schemes have advanced in a pace that would enable users to develop secure cloud computation over sensitive data. The first FHE scheme was proposed by Craig Gentry in 2009, and although it was not a practical implementation, his theory laid the foundations to many schemes that exists today. The basic idea in Fully Homomorphic research is the creation of a library that allows users consume the technology securely without much knowledge of the underlying mathematical complexities of FHE. In this study, we will present the concepts behind FHE, together with the introduction of three open-source FHE libraries, in order to understand the details of its functions and capabilities offered in each. Our implementation begins with FHE Framework setup, the minimum requirements to accomplish a successful evaluation procedure for homomorphic encryption scheme, results extraction and finally derive conclusions.

II. Introduction

Homomorphic encryption is a cryptographic primitive that enables computation directly on encrypted data. This technology has the potential to change the way that we protect arbitrary computation or storage on shared infrastructure environments such as in Cloud Computing where privacy of sensible data is one of most important security issues. Moreover, it may be judiciously applied to special-purpose problems that requires stronger security guarantees than are not possible with traditional cryptography methods. Fully Homomorphic Encryption FHE schemes are considered as the next generation algorithms for cryptography, it is a type of smart encryption cryptosystems that support arbitrary computations on ciphertexts without ever needing to decrypt or reveal it. The first adequate Fully Homomorphic Encryption scheme is achieved by Gentry in 2009 by proposing bootstrapping theorem which provides a technique to allow refreshment of ciphertexts and reduce noise after its growth in encrypted functions.

The last few years have seen substantial research into the design of homomorphic encryption algorithms. Additionally, a few of the algorithms have been implemented in open-sourced software. Evaluations of these software implementations is allowing research communities to determine the feasibility of the theory and applications that could benefit most from targeted use of homomorphic encryption technology. However, prior evaluations were often tedious to conduct and were ad hoc in nature.

Contributions: The main goal of this study is to provide a generic understanding of these open-source framework and testing of the available homomorphic encryption schemes. The study here describes the design implementation methodology of the FHE frameworks, and use the results of the implementation work for understanding the efficiencies in Palisade, SEAL and HElib software packages.

Project outline: The rest of this report is decomposed as follows: In Section III, a brief introduction to Homomorphic Encryption and the types were discussed. In Section IV, Different underlying schemes used in this project were discussed. In Section V, Libraries and Environmental setup for running our programs were given in detail with prerequisites and commands. In Section VI Programs Results were discussed. In Section VII, Comparison table based on time taken for encryption, decryption, parameter generation and evaluation were shown and concluded in Section VIII.

III. TYPES OF HOMOMORPHIC ENCRYPTION

Homomorphic Encryption[5][2] is a special type of encryption scheme that gained significant momentum in recent years. When data is encrypted under a traditional encryption scheme, there are only two operations that can be performed on the data which is storage and retrieval. If any computation or analysis needs to be performed on the encrypted data, it must be decrypted first. This requirement comes at a cost of time, resources, and more importantly privacy. Homomorphic encryption schemes provide an alternative to regular cryptographic schemes by offering mechanisms to perform operations upon encrypted data without having to decrypt it first.

The HE Schemes basically operates on a circuit level, meaning that the functions used in homomorphic encryption must consist only of binary operations like AND and XOR

For example, addition can be represented as the multiplication of two bits:

AND(b1, b2) = b1 b2 and

XOR can be seen as the addition of two bits modulo 2:

XOR(b1, b2) = (b1 + b2) mod 2

Example:

Enc(a),Enc(b) $\rightarrow Enc(aXORb) and Enc(aANDb)$

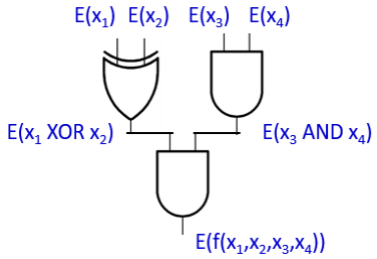


Fig 2.1: Sample circuit implementing HE

With increasing concerns in data security posture and privacy leakage of sensitive data, the conventional encryption methods are not sufficient enough to provide necessary security from an intermediary service like cloud servers. The Homomorphic encryption is envisioned as an encryption mechanism that can resolve the security and privacy issues in cases like cloud.

Unlike the public key encryption, which has three security procedures, i.e., key generation, encryption and decryption; there are four procedures in HE schemes, including the evaluation algorithm. HE allows the third-party service providers to perform certain type of operations on the users encrypted data without decrypting, therefore maintaining the privacy of the users data without having to bring it clear in memory during computation. In homomorphic encryption, if the user wants to query some information on the computing device, he first encrypts the data and stores it for later use. The follow-on operations involve user sending queries to extract data from storage, running prediction algorithms on encrypted data using HE and performs necessary computations without knowing the contents of the encrypted data. The system them returns the encrypted prediction back to the user where the decryption takes place in trusted zone by using the users secret key.

There are three main types of homomorphic encryption based on the number of mathematical operations on the encrypted message.

1. Partial Homomorphic Encryption
2. Somewhat Homomorphic Encryption
3. Fully Homomorphic Encryption

1. Partial Homomorphic Encryption (PHE):

In PHE scheme, only one type of mathematical operation is allowed on the encrypted message, i.e., either addition or multiplication operation, with unlimited number of times.

Example: RSA is an asymmetric encryption scheme where each user must have a public and private key in order to carry out encryption and decryption.

Let e, n be some public key. The encryption of message m is given by:

$E(m) = m^e mod\ n$

Consider two messages m1 and m2. RSA has a multiplicative homomorphic property as the product of two ciphertexts is equal to the encryption of the product of the two messages.

$E(m_1)E(m_2) = m_1^e m_2^e mod\ n = (m_1 m_2)^e mod\ n = E(m1m2)$

Since RSA only has multiplicative homomorphism, it is termed as a partial homomorphic encryption scheme (PHE).

2. Somewhat Homomorphic Encryption (SHE):

In SHE, both addition and multiplication operation is allowed but with only a limited number of times.

In addition to being a valid encryption scheme, this scheme is also homomorphic because by adding, subtracting, or multiplying the ciphertext, the underlying messages are actually added, subtracted, or multiplied modulo 2. Despite this being a valid homomorphic scheme, a significant problem exists. As operations are carried out, the noise of the system grows. Eventually it grows to a point where decryption no longer returns the correct result. This means that the scheme can only support some functions, namely functions where the accrued noise does not grow over p/2, Where p is random odd integer.

Since SHE is limited to functions that are not too complicated (e.g. functions that only contain low-degree polynomials), one may wonder if there is any benefit to using SHE over FHE when FHE can compute any function. In fact, there are cases in which SHE may be sufficient or even preferred over FHE. In the case where only a simple function needs to be computed, the overhead of SHE is much lower than the overhead of FHE.

SHE schemes could not handle their own decryption function without significant modifications, meaning that these types of schemes are not inherently bootstrappable. If the SHE scheme can be bootstrapped, it can then be used to construct a FHE scheme.

3. Fully Homomorphic Encryption (FHE):

FHE allows a large number of different types of evaluation operations on the encrypted message with unlimited number of times.

While SHE schemes only allow the computation of some functions, fully homomorphic encryption (FHE) schemes enable the computation of arbitrary functions on encrypted data. This property makes FHE the most sophisticated homomorphic encryption scheme.

Bootstrapping: Bootstrapping[7] refreshes a ciphertext by running the decryption function on it homomorphically. The complexity of bootstrapping is at least the complexity of the decryption times the bit-length of the individual ciphertexts that are used to encrypt the bits of the private key.

Suppose that is bootstrappable with plaintext space P which is 0,1, and the circuit C is Boolean. And we have a cipher-text 1 that encrypts under pk1 which we want to refresh. Suppose that we can decrypt it homomorphically, and we also have Sk1 the secret key for pk1. S_k1is encrypted under a second public key pk2: let sk_{1J} be the encrypted secret key bits. Consider the following algorithm. Recrypt(p_{k2}, D, $\overline{sk_{1j}}$, $\overline{\psi_{iJ}}$).

$$Set\overline{\psi_{1_J}} \overset{R}{\leftarrow} Encrypt\ (pk_2, \psi_{1j})$$

Output $\psi_2 \leftarrow Evaluale_{\varepsilon}(pk_2,\ D_{\varepsilon},\ \langle\langle\overline{sk_{1J}}\rangle, \langle\overline{\psi_{1J}}\rangle\rangle)$

Modulus Switching: The main idea of modulus switching is to use an evaluator, which determines the bound on the length of the private key, but not to private key itself. It is used to transform a ciphertext c mod q into a different ciphertext $c^1 mod\ p$ without jeopardizing correctness of the scheme.

IV. UNDERLYING SCHEMES

4.1 BGV: Brakerski-Gentry-Vaikuntanathan :

The BGV scheme was proposed in 2012 in the paper[8] Fully Homomorphic Encryption without Bootstrapping by Brakerski, Gentry, and Vaikuntanathan. BGV is a levelled FHE (LFHE) scheme, meaning that the parameters of the scheme depend (polynomially) on the depth of the circuits that the scheme is capable of evaluating using modulus switching. BGV is not practical for real world applications due to storage overhead[7].

The basic BGV scheme can be broken down into following functions

ParamGen(λ, μ, b), SecKeyGen(params), PubKeyGen(params), Enc($params, m, p_k$), Dec(s_k, c), EvalAdd(c_1, c_2), EvalMult(c_1, c_2)

where

λ, μ are input security parameters,

bit b 0, 1 to set the LWE version or RLWE,

c_1, c_2 are Ciphertexts,

s_k, p_k are secret key and public key respectively

4.2 BFV: Brakerski-Fan-Vercauteren:

In 2012, Fan and Vercauteren modified Brakerskis fully homomorphic encryption scheme based on LWE to work under the security assumption of RLWE[2]. They make use of relinearization, but BFV has a more efficient approach. They also simplify the bootstrapping procedure, by introducing modulus switching.

The BFV scheme can be broken down into following functions

PrivateKeyGen(λ), PublicKeyGen(s_k), EvaluationKeyGen(s_k, T), Encrypt(p_k, m), Decrypt(s_k, c), Add(c_1, c_2), Multiply(c_1, c_2)

where

λ is input security parameter,

c_1, c_2 are Ciphertexts,

s_k, p_k are secret key and public key respectively

4.3 CKKS: Cheon-Kim-Kim-Song:

CKKS was proposed by Cheon, Kim, Kim, and Song in the paper Homomorphic Encryption for Arithmetic of Approximate Numbers[9] released in 2017. CKKS solves this problem by allowing computation on complex numbers with limited precision by treating the encryption noise as part of the error that occurs naturally during approximate computations.

The CKKS scheme can be broken down into following functions

KeyGen(λ), Enc(m, p_k), Dec(c, s_k), Add($c_1, c2$), Mult(c_1, c_2), Relinearize

where

λ is input security parameter,

c_1, c_2 are Ciphertexts,

s_k, p_k are secret key and public key respectively

V. LIBRARIES AND ENVIRONMENTAL SETUP

5.1 PALISADE:

PALISADE[4] was released in 2017 and is currently supported by a team at the New Jersey Institute of Technology along with the backing of several partners and collaborators in academia (MIT, UCSD, WPI, ...) and industry (Raytheon, IBM Research, Galois, etc.,). It is released under the BSD 2 clause and has cross platform support for Windows, Linux, MacOS, and Android environments. It is written in C++ and the objects that are created by and manipulated within PALISADE are instances of C++ classes.

The layers in PALISADE are as follows:

1. Application: All programs that call the PALISADE library services are in this layer.

2. Encoding: All implementations of methods to encode data are at this layer.

3. Crypto: All implementation of cryptographic protocols are at this layer.

4. Lattice Operations: All higher-level lattice-crypto mathematical building blocks are in this layer.

5. Primitive Math: All low-level generic mathematical operations, such as multi-precision arithmetic implementations, are at this layer.

5.1.1 Implementation

1. Install pre-requisites: g++, CMake, Make, and autoconf

```
# sudo apt-get install buildessential

# sudo apt-get install cmake

# sudo apt-get install autoconf
```

2. Clone the PALISADE git repository to the local machine

```
# git clone https://gitlab.com/palisade /palisade-release.git
```

3. Change directories to the file where the cloned Palisade repository is and download the sub-modules

```
# cd PALISADE

# git submodule sync –recursive

# git submodule update –init recursive
```

4. Create a build directory where the binaries will be built

```
# mkdir build

# cd build

# cmake ..
```

5. Build and install PALISADE. In the build folder: Run unit tests to make sure all capabilities operate as expected

```
# make testall
```

6. Run the sample example

```
# cd bin/examples/pke

# ./simple-integers
```

5.2 Microsoft SEAL:

Microsoft SEAL(Simple Encryption Arithmetic Library) is a homomorphic encryption library that allows additions and multiplications to be performed on encrypted integers or real numbers. Other operations, such as encrypted comparison, sorting, or regular expressions, are in most cases not feasible to evaluate on encrypted data using this technology. Therefore, only specific privacy-critical cloud computation parts of programs should be implemented with Microsoft SEAL.

Microsoft SEAL comes with two different homomorphic encryption schemes with very different properties. The BFV scheme allows modular arithmetic to be performed on encrypted integers. The CKKS scheme allows additions and multiplications on encrypted real or complex numbers, but yields only approximate results. In applications such as summing up encrypted real numbers, evaluating machine learning models on encrypted data, or computing distances of encrypted locations CKKS is going to be by far the best choice. For applications where exact values are necessary, the BFV scheme is the only choice.

5.2.1 Implementation:

1. Install the pre-requisites: CMake, g++

```
# sudo apt-get install build - essential

# sudo apt-get install cmake
```

2. Clone the SEAL github repository to the local machine

```
# mkdir SEAL

# cd SEAL

# git clone https://github.com/dnat112/SEAL.git –recurse submodules
```

3. Build SEAL

```
# cd native/src

# cmake .

# make
```

4. Build SEAL examples

```
# cd native/examples

# cmake .

# make
```

5. Build unit tests

```
# cd native/tests

# cmake .

# make
```

6. Install SEAL

```
# cd native/src

# cmake .

# make

# sudo make install
```

7. To run the tests, navigate to bin directory

```
# ./sealtest
```

5.3 HElib:

HElib is an open-source (Apache License v2.0) software library that implements homomorphic encryption (HE). Currently available schemes are the implementations of the Brakerski-Gentry-Vaikuntanathan (BGV) scheme with bootstrapping and the Approximate Number scheme of Cheon-Kim-Kim-Song (CKKS), along with many optimizations to make homomorphic evaluation run faster, focusing mostly on effective use of the Smart-Vercauteren ciphertext packing techniques and the Gentry-Halevi-Smart optimizations. See this report for a description of a few of the algorithms using in this library.

HElib supports an ”assembly language for HE”, providing low-level routines (set, add, multiply, shift, etc.), sophisticated automatic noise management, improved BGV bootstrapping, multi-threading, and also support for Ptxt (plaintext) objects which mimics the functionality of Ctxt (ciphertext) objects. HElib is written in C++17 and uses the NTL mathematical library. HElib is distributed under the terms of the Apache License v2.0.

5.3.1 Implementation:

1. Helib build on Ubuntu 20.04 with general prerequisites as follows:

GNU make ≥ 4.2
g++ $\geq 10.3.0$
cmake ≥ 3.16

2. Installation of prerequisites : Cmake, make, g++, pthreads, git, patchelf and m4

```
# sudo apt - get install build - essential cmake make libpthread -stubs0 - dev git patchelf m4
```

3. Making g++ - 10 version as default

sudo apt y install g++-10

sudo update-alternatives install /usr/bin/g++ g++ /usr/bin/g++-10 10

sudo update-alternatives config g++(select g++-10 option to make it default)

4. Cloning Helib from Git repository

git clone https://github.com/homenc/HElib.git

5. Create a build directory.
Run the CMAKE configuration.

cmake DPACKAGE_BUILD=ON DENABLE TEST=ON ..

6. Compile the install and test the compilation

Make -j8

7. Run the Install

sudo make install

8. Helib comes with 3 examples:

BGV_binary_arithmetic

BGV_country_db_lookup

BGV_packed_arithmetic

9. To run the examples, create a Build directory.
Move it into examples folder
$cmake[-DhelibDIR = /usr/local/helib\_pack/share/$ cmake/helib] /home/stetali/Helib/examples ..

10. Compile and Run the install
To run the examples, move to bin folder
Select an example and run

./example_name h

VI. IMPLEMENTATION RESULTS

The Study is based on homomorphic evaluation of an arbitrary function e= y(x+z). The function here is selected to suffice the need to evaluate HE frameworks, functionality and provide insights not only on how the libraries work, but also gives us a ability to perform baseline comparisons and understand their performance metrics.[3] We further explored the possibilities of using batching/packing or any other optimizations available. The target security level for all of the tests is 128 bits, meaning that it should take an attacker 2 128 operations to break the scheme. In order to show the comparison between the different libraries, we will implement the same example program in each library for the schemes it supports. We have the results tabulated and compared the schemes based on timing and by considering run time for parameter generation, key generation, encryption, evaluation, and decryption. All tests were performed on separate VMs running Ubuntu 20.04 LTS and programs are in C++. The complied code along with the source is made available on https://github.com/stetali23/Homomorphic-Encryption.git

6.1 Implementation of BGV Scheme:

The BGV scheme is essentially available in HElib and PALISADE among the libraries considered in this study. In HElib, we were able to obtain 2760 elements in each vector and performed calculations to evaluate the resultant equation. Whereas, In PALISADE we could generate values on smaller vector size and its was due to complexities involved in choosing the parameters.

Apparently the BGV setting in PALISADE would require several additional parameters compared to the other available schemes. Regardless, based on the other PALISADE experiments and obtained results it is evident that the PALISADE implementation of BGV would run faster than HElibs to compute the same function. Figures 6.1 , 6.2 show the outputs of the BGV test for PALISADE and HElib

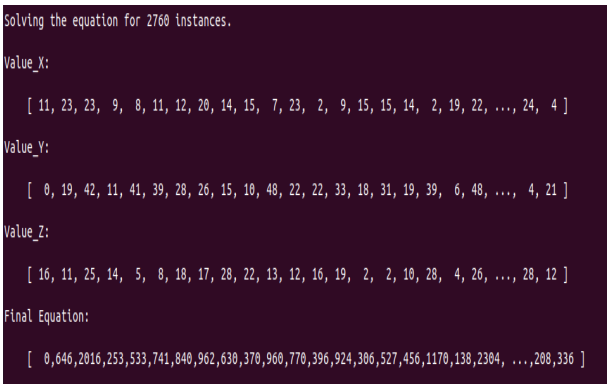


Fig 6.1: Final Equation Calculation implemented with BGV on Helib

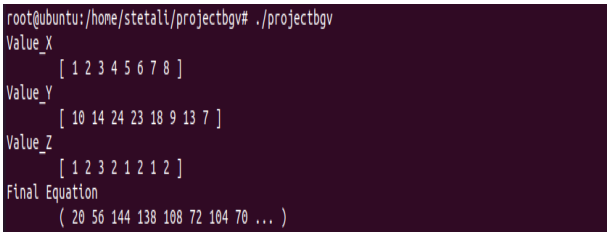


Fig 6.2: Final Equation Calculation implemented with BGV on palisade

6.2 Implementation of BFV Scheme:

SEAL and PALISADE libraries are fundamentally incorporated in BFV Schemes and therefore provide the scope to study its function in HE frameworks. The vector sizes considered were kept alike with 2670 elements long and all of the elements were generated randomly. Operations were performed element wise, starting first with the addition of x and z vectors followed by the multiplication of vector y to calculate the final equation. While the time factor for parameter generation were fairly close in two libraries, the time taken in other areas were vastly different with SEAL

always performing faster and therefore reducing the aggregate time taken to evaluate the arbitrary function. See Figures 6.3, 6.4 for the outputs of the experiments.

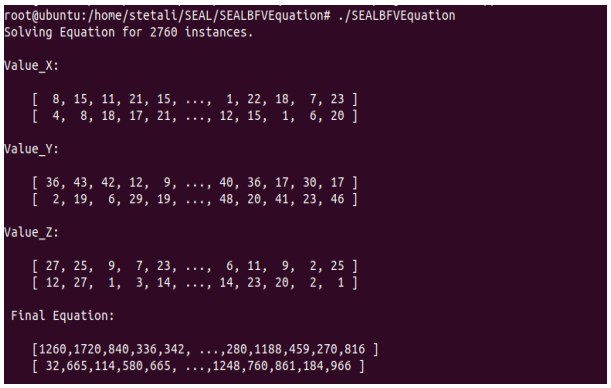


Fig 6.3: Final Equation Calculation implemented with BFV on SEAL

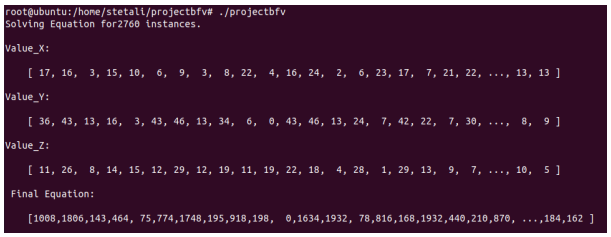


Fig 6.4: Final Equation Calculation implemented with BFV on palisade

6.3 Implementation of CKKS Scheme:

The CKKS scheme is inherently made available in SEAL and PALISADE libraries and to keep our evaluation unbiased, the Vectors sizes were essentially chosen the same as in other tests which is 2760 elements and considering that all elements were randomly generated. Operations were performed at element level, starting first with the addition of vectors (x, z) as similar to tests conducted in BFV tests. PALISADE was able to do addition and multiplication of the vectors with no steps in between, but SEAL required re-linearization and re-scaling steps between the two math operations. Comparing the time taken for various attributes, its determined that except for Parameter generation all other remaining attributes (like Key Gen, Enc, Eval, Decry) performed significantly faster under SEAL CKKS implementation. Figures 6.5, 6.6 show the outputs of these tests.

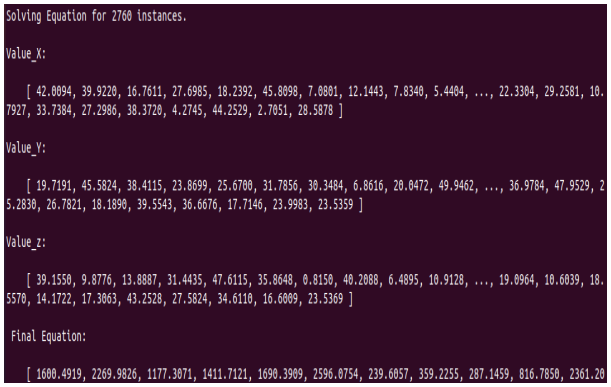


Fig 6.5: Final Equation Calculation implemented with CKKS on SEAL

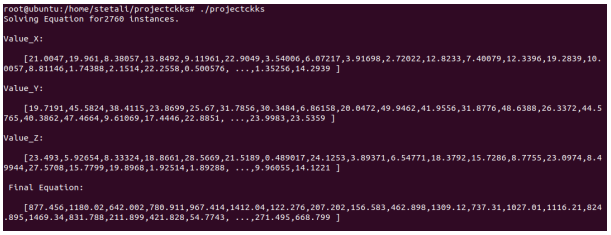


Fig 6.6: Final Equation Calculation implemented with CKKS on palisade

VII. Comparison of Results:

Comparison Table shown below depicts the difference of elapsed timing variations (in milliseconds) that each corresponding library took to evaluate essentials functions and to carry out Fully Homomorphic Encryption

Schemes	Libraries	Param Gen	Key Gen	Encryption	Evaluation	Decryption	Total Time
BGV	PALISADE	11.351	503.454	219.291	31.797	29.491	795.384
	Helib	14162.4	4393.89	221.522	288.121	994.884	20060.82
BFV	PALISADE	30.964	87.478	90.632	77.812	1.106	287.992
	SEAL	52.848	30.626	20.889	26.758	3.777	134.898
CKKS	PALISADE	13.076	91.191	82.08	50.75	57.704	294.801
	SEAL	15.276	42.513	35.136	17.062	0.322	110.309

Table 1: Comparison of time taken for different parameters during Homomorphic Evaluation

The tabulated results derived from BFV and CKKS schemes, we can clearly determine that the time taken to generate paramaters is fairly close for the two libraries, PALISADE and SEAL, whereas the time taken to determine other attributes have outperformed in SEAL compared to PALISADE libraries. Therefore, its clearly evident that SEAL performed computation on choosen arbitarry function faster than other counterparts.

Also, the BGV implementation in both PALISADE and Helib demonstrated sluggish performance amongst all available schemes.

VIII. Conclusion:

Based on Homomorphic Encryption Frameworks available today, this study demonstrated clean implementation of various FHE schemes with derived experimental results. The project began by considering an arbitrary function that could suffice the need to thoroughly validate FHE schemes by performing operations on encrypted data. Each scheme was explored to understand the fundamentals involved to achieve the possibilities of computing on encrypted data.

The study illustrates feasibility of computations on Encrypted Data using various types of Homomorphic schemes. Furthermore, the paper enlightens the differences in today's

freely available Homomorphic Libraries by making comparisons based on execution time.

REFERENCES

[1] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A guide to fully homomorphic encryption," *IACR Cryptol. ePrint Arch.*, vol. 2015, p. 1192, 2015.

[2] A. Carey, "On the explanation and implementation of three open-source fully homomorphic encryption libraries," 2020.

[3] C. Gentry, "Computing arbitrary functions of encrypted data," *Commun. ACM*, vol. 53, no. 3, p. 97105, mar 2010. [Online]. Available: https://doi.org/10.1145/1666420.1666444

[4] K. R. Yuriy Polyakov1 and G. W. Ryan1, "Palisade lattice cryptography library user manual," December 29, 2017.

[5] M. Will and R. Ko, *A guide to homomorphic encryption*, 12 2015, pp. 101–127.

[6] A. Viand, P. Jattke, and A. Hithnawi, "Sok: Fully homomorphic encryption compilers," 2021.

[7] K. Hariss, M. Chamoun, and A. E. Samhat, "On dghv and bgv fully homomorphic encryption schemes," in *2017 1st Cyber Security in Networking Conference (CSNet)*, 2017, pp. 1–9.

[8] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(leveled) fully homomorphic encryption without bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ser. ITCS '12. New York, NY, USA: Association for Computing Machinery, 2012, p. 309325. [Online]. Available: https://doi.org/10.1145/2090236.2090262

[9] J. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic encryption for arithmetic of approximate numbers," 11 2017, pp. 409–437.

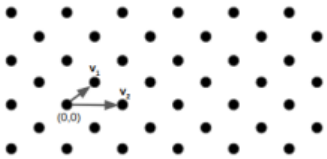
APPENDIX

Lattice Definitions

An n-dimensional lattice L is the set of all linear combinations of n linearly independent vectors v1, v2, ..., vn.

L = a1v1 + a2v2 + ... + anvn—ai Z

In other words, lattices are mathematical structures that consist of points in an n dimensional space, with some periodic structure. For example, consider a lattice with n = 2:



The linearly independent vectors v1, ..., vn that make up the lattice are called the basis B of the lattice. Generally the basis vectors are organized into an n n matrix where each vector becomes a column:

$$\mathbf{B} = \begin{pmatrix} \mathbf{v}_{1_0} & \mathbf{v}_{2_0} & \cdots & \mathbf{v}_{n_0} \\ \mathbf{v}_{1_1} & \mathbf{v}_{2_1} & \cdots & \mathbf{v}_{n_1} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{v}_{1_{n-1}} & \mathbf{v}_{2_{n-1}} & \cdots & \mathbf{v}_{n_{n-1}} \end{pmatrix}$$

The basis in matrix form can be used to represent the lattice in the following way:

$$\mathcal{L} = \mathcal{L}(\mathbf{B}) = \sum_{i=1}^{n} a_i \mathbf{v}_i : a_i \text{ are integers}$$

Random lattices

1. Operations on large matrices (e.g., 532x840)
2. Mostly matrix-vector multiplication modulo $q \leq 2^{32}$
3. Large public keys (e.g., 532x840 matrix)

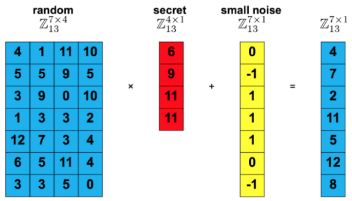
Ideal lattices

1. Operations on polynomials with 256 or 512 coefficients
2. Mostly polynomial multiplication modulo $q \leq 2^{32}$
3. Public keys are one (or two) polynomials with 256 or 512 coefficients

Learning With Errors

LWE is a quantum robust method of cryptography. Initially we create a secret key value (s) and another value (e). Next we select a number of values (A[]) and calculate B[] = A[] x s + e. The values of A[] and B[] become our public key. If s is a single value, A and B are one dimensional matrices. If we select s to be a one-dimensional matrix, A will be a two-dimensional matrix, and B will be a one-dimensional matrix.

It involves the difficulty of finding the values which solve: B=As+e where you know A and B. The value of s becomes the secret values (or the secret key), and A and B can become the public key.



Ring Learning With Errors

In this method we perform a similar method to the Diffie Hellman method, but use Ring Learning With Errors (RLWE). With RLWE we use the coefficients of polynomials and which can be added and multiplied within a finite field (F_q) and where all the coefficients will be less than q.