Evaluation of the Complexity of Fully Homomorphic Encryption Schemes in Implementations of Programs

Dimitar Chakarov
dimitar_ch@hotmail.com
High School of Mathematics 'Acad. Kiril Popov'
Plovdiv, Bulgaria

Yavor Papazov
yavorpap@gmail.com
ESI – CEE
Sofia, Bulgaria

ABSTRACT

This paper shows a thorough examination of fully homomorphic encryption schemes and their performance in programs. We do an extensive analysis of one specific, widely-used scheme – the Gentry-Sahai-Waters scheme, and its variations – the libraries TFHE and FHEW. We aim to devise an abstraction level that enables us to assess the real-world speed performance of fully homomorphic operations without actually working with encrypted data. We propose an algorithm that uses the gathered statistical data combined with our mathematical model to evaluate the performance of fully homomorphic implementations of arbitrary computer programs.

CCS CONCEPTS

• **Security and privacy → Mathematical foundations of cryptography**; • **Theory of computation → Computational complexity and cryptography**; *Logic.*

KEYWORDS

cryptography, homomorphic encryption, analysis

ACM Reference Format:
Dimitar Chakarov and Yavor Papazov. 2019. Evaluation of the Complexity of Fully Homomorphic Encryption Schemes in Implementations of Programs. In *Proceedings of the 20th International Conference on Computer Systems and Technologies (CompSysTech '19), June 21–22, 2019, Ruse, Bulgaria.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3345252.3345292

1 INTRODUCTION

Fully homomorphic encryption schemes, originally called *privacy homomorphisms*, were introduced by Rivest, Adleman and Dertouzous shortly after the invention of RSA by Rivest, Shamir and Adleman. In short, fully homomorphic encryption preserves the arithmetic operations addition and multiplication over encrypted bits [17]. After 2009, a plethora of different algorithms, reliant on Gentry's construction [8], have been created ([9–14, 16]. Furthermore, in the recent years a number of dedicated libraries have emerged. A specifically efficient and developed library, which happens to be the stepping stone in our research as well, is the TFHE library [4, 5].

CompSysTech '19, June 21–22, 2019, Ruse, Bulgaria

ACM ISBN 978-1-4503-7149-0/19/06…$15.00
https://doi.org/10.1145/3345252.3345292

Fully homomorphic programs are secure yet inefficient in terms of speed and memory usage. This stems from the fact that every encrypted bit carries a certain amount of noise in it, which grows with each operation, so in order to minimize the noise and not corrupt the data, we need to perform a bootstrapping step that significantly increases the computational time. Therefore, it is convenient to know how much resources such a program is going to use.

We analyze one specific type of fully homomorphic encryption schemes and implement an abstraction level that determines the expected time for executing a program as if it was run fully homomorphically (performing the ordinary operations, not the fully homomorphic ones). It consists of an algorithmic part – a mathematical model for analysing fully homomorphic binary operations, and a 'fake' library that plays the role of a wrapper and executes the algorithm.

2 PRELIMINARIES

In this section we formally define fully homomorphic encryption and its variations.

Let us have an encryption scheme (K, E, D, P, C). With K we denote the key space. E and D are respectively the encryption and decryption algorithms. P is the plaintext and C is the cyphertext, such that $C = E_k(P)$, where $k \in K$ is a either a secret key (in the secret-key cryptosystem) or a public key (in the public-key cryptosystem).

Definition 1. A *somewhat homomorphic encryption (SHE) scheme* is an encryption scheme (K, E, D, P, C), such that the plaintext and the cyphertext form rings – $(P, \oplus_p, \otimes_p)$ and $(C, \oplus_c, \otimes_c)$, and

$$E_k(a \oplus_p b) = E_k(a) \oplus_c E_k(b)$$
$$E_k(a \otimes_p b) = E_k(a) \otimes_c E_k(b)$$

are true for all $a, b \in P$ and all $k \in K$. An encrypted bit can participate only in so many additions and multiplications, whose number depends on the nature of the scheme, but not on its parameters.

Definition 2. A *fully homomorphic encryption (FHE) scheme* is an extension of somewhat homomorphic encryption schemes, where addition and multiplication can be performed an arbitrary (indefinite) number of times on the same set with the homomorphic property still being valid.

Definition 3. A *levelled homomorphic encryption (LHE) scheme* is again an extension of somewhat homomorphic encryption schemes. The LHE scheme has its own depth parameter d, which tells us the maximal depth (in the possible number of multiplications) of an arbitrary circuit that can be executed.

We want all FHE schemes to preserve the two binary operations AND and XOR ($\otimes_p = \wedge$ and $\oplus_p = \oplus$) between the plaintext and

the cyphertext. Hence, an arbitrary circuit can be executed on the cyphertext and the decrypted result is as if the circuit was ran on the plaintext.

3 THE BOOTSTRAPPING PROCEDURE

In this section we describe thoroughly the essence of bootstrapping.

The bootstrapping starts with 'squashing' the decryption circuit, i.e. representing it as a low-degree polynomial in the bits of the cyphertext and the bits of the secret key. Then, instead of evaluating it on the cyphertext and the original secret key, we perform it fully homomorphically using the encryption of the secret key bits. Hence, we do not get the plaintext bit, but rather another encryption of that bit that has potentially less noise. The noise reduction depends solely on the depth of the decryption circuit, so the smaller the depth the bigger the reduction. To be exact, a SHE scheme that can evaluate its own decryption circuit is called *bootstrapable* and is essentially a fully homomorphic scheme. See figures 1 and 2 for visualisation. Bootstrapping by itself is a really

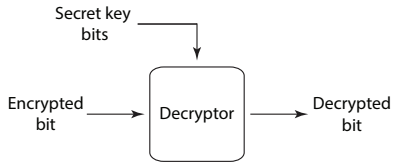

Figure 1: Normal decryption

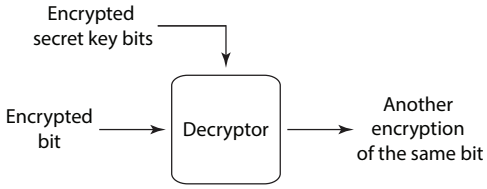

Figure 2: Bootstrapping

time-consuming operation. The implementation of Gentry's first fully homomorphic encryption algorithm performed bootstrapping on a single bit for about 30 minutes. Later this was improved to around 14 minutes, then 6 minutes, a couple of seconds, and, finally, less then 15 milliseconds in [5] (for detailed manually computed comparisons see [1]).

Depending on the fully homomorphic scheme one can differentiate three main categories of bootstrapping:

- *Fully Homomorphic Gate Bootstrapping* – A fast bootstrapping is performed on a single encrypted bit after a binary operation.
- *Fully Homomorphic Circuit Bootstrapping* – A similar bootstrapping step is done on a batch of encrypted bits after a single binary operation.
- *Leveled Fully Homomorphic Bootstrapping* – An encrypted bit can be used in at most d operations (d being the depth parameter of the FHE scheme), after which performing bootstrapping is mandatory, so as to make the scheme independent on the depth parameter d and enable the execution of infinite number of operations.

Currently, there do not exist FHE schemes that do not rely on a bootstrapping step, hence, we try to examine all three types of bootstrapping and devise an algorithm that measures their performance in program implementations, which one can think of as large arbitrary circuits.

4 FULLY HOMOMORPHIC SCHEMES WITH GATE AND CIRCUIT BOOTSTRAPPING

In this section we present an extensive description of FHE schemes based on the Gentry-Sahai-Waters scheme, which employ gate and circuit bootstrapping techniques.

4.1 Overview

We start with stating formally the underlying problem of the GSW scheme – the Learning with Errors problem.

Definition 4 (Learning with Errors)**.** Let $n, q \geq 1$ be integers and $\mathbf{s}$ be a uniformly distributed secret in $\mathbb{Z}_q^n$. With $\mathtt{LWE}_{n,q,\mathbf{s}}$ we denote the distribution over $\mathbb{Z}^n \times \mathbb{Z}$, obtained by sampling a number of pairs $(\mathbf{a}, b)$, where $\mathbf{a} \in \mathbb{Z}_q^n$ is chosen uniformly at random and $b = \mathbf{a} \cdot \mathbf{s} + e$. The error e is chosen from a gaussian distribution ('error' probability distribution) over $\mathbb{Z}_q^n$. The aim is to determined $\mathbf{s}$, based only on the information of the $\mathtt{LWE}_{n,q,\mathtt{s}}$ distribution.

In GSW-based schemes, a bit is encrypted as a LWE sample $(\mathbf{a}, b)$, represented as matrices. The fully homomorphic property holds, since the set of LWE samples behaves like a ring. The encryption and decryption steps are case-specific, therefore, we will not elaborate on them. However, the bootstrapping step being the polynomial representation of the decryption circuit is essentially a sequence of matrix multiplications and automorphisms. In the current research paper we limit ourselves to two specific FHE schemes – TFHE [4, 5] and FHEW [6]. They are two of the four implementations of fully homomorphic schemes, hence, we have the ability to analyse their behavior from a practical viewpoint, as well. Chronologically, FHEW precedes TFHE and is less developed, but we include it for completeness. In both implementations only the gate bootstrapping variant is presented with the circuit bootstrapping being a theoretical concept still (there exist only experimental proof-of-concept implementations) [2, 15].

4.2 Statistical Analysis

Two case specific tests were run in order to evaluate the expected execution time of the gate bootstrapping for both libraries. The first one performs operations on random bits on every trial and the second performs the same operations on the same bit set[1] on every trial. Every binary operation was executed 1024 times on 101 (random and predetermined) bits and the time for each execution was measured. The tests were run on an Intel Core i7 4700-HQ (2.40 GHz). For TFHE we used the *spqlios* engine, written with AVX assembly instructions, whereas, for FHEW we used the *FFTW3*

[1]The set was generated using the standard C++ `rand()` function

engine. Table 1 summarizes the average running time of a single gate along with the standard deviation (in seconds).

Gate	Random bits		Predetermined bits	
	Time	σ	Time	σ
TFHE	0.0144118	0.00011256	0.0142069	0.00005321
FHEW	0.2460612	0.00777669	0.2449758	0.00948597

Table 1: Performance test of the TFHE and FHEW libraries with random and predetermined bits

5 THE ALGORITHM

This section examines in depth our algorithm for evaluating the performance of fully homomorphic programs without actually performing the costly bootstrapping operations.

5.1 Overview

We define an abstraction level that mimics fully homomorphic operations.

Fact 1. *The time execution of a fundamental operation is constant with respect to the variables.*

Fact 2. *The bootstrapping procedure is defined as the polynomial form of the decryption circuit in the binary operations. In implementations it is equivalent to such polynomials.*

Considering that even the two fundamental operations are more time-expensive than normal ones [1, 17], we can conclude that in order to get a good enough estimation of the running time we need to keep track of the bootstrappings executed and the single binary operations. Since we work with statistical data for the average running time of a bootstrapping we take into account the deviation, as well. Now we will formally state the time complexity in terms bootstrapping executions. Let $\mathbb{B}$ denote a single bootstrapping. We have that $f_{\mathbb{B}}$ is the number of bootstrapping steps and $t_{\mathbb{B}}$ the average running time with deviation $\sigma_{\mathbb{B}}$. Hence, we get the following two equations for the approximate time t and the standard deviation σ

$$t = f_{\mathbb{B}} \cdot t_{\mathbb{B}} + t_{+}$$
$$\sigma = f_{\mathbb{B}} \cdot \sigma_{\mathbb{B}} + \sigma_{+},$$

where t_{+} and σ_{+} are the added time of the binary operations without bootstrapping.

We have derived the following theorem for the minimal number of bootstrappings.

Theorem 1. *Obtaining an optimal amount of bootstrapping procedures is equivalent to determining the smallest number of sets of independently-executable operations, so that all remaining operations depend on the said set.*

Proof. On one hand, we assume that a configuration with minimal bootstrapping procedures has two components of bits that are independent on one another at a given time. Since they are independent we can combine the bit sets and do a bootstrap on the new bigger set, thus reducing the number of bootstrappings. We get a contradiction.

On the other hand, if every two components are dependable on one another, then we cannot reduce the number of bootstrapping procedures as we risk compromising the correctness of the data. □

In layman terms, we do bootstrapping only when we are required by the program itself and cannot continue otherwise as we are risking security and bit corruption.

Corollary 1. *The minimal number of bootstrapping steps for Gate Bootstrapping schemes is the number of binary operations.*

Corollary 2. *The minimal number of bootstrapping steps for Circuit Bootstrapping schemes is the depth of the dependence tree of the encrypted bits in the circuit.*

Corollary 3. *For Levelled Bootstrapping we can reduce to Circuit bootstrapping by adding an artificial counter that records the depth of the bit. Hence, we get a modified dependence tree and its depth is the optimal number of bootstrapping operations.*

5.2 Possibilities and Limitations

We will make a slight diversion from the technicalities and talk about the possibilities and limitations of our analysis of fully homomorphic programs as well as the fully homomorphic encryption schemes as a whole.

Fundamentally, FHE schemes preserve binary manipulations over bits, i.e. all of them take as an input two (or in the case of the `MUX` gate – three) encrypted bits and output a single encrypted bit. What they are not capable of is preserving logical operations. Therefore, implementing an ordinary cyclic circuit (e.g. the `for`-cycle in C++) or an `if`-statement, which depend on a clearly defined condition – one that can be either `true` or `false` and not their encrypted counterparts, is nearly impossible.

On the other hand, because binary operations are defined for both the plaintext and the ciphertext, then we can implement all arithmetic operations along with others like showing if two bit sequences are the same or finding the minimal/maximal among two encrypted numbers. It is also possible to an extent to implement a less functional, yet usable if-else statement with the help of the `MUX` gate. What's more, implementing a cycle can be bypassed by taking the number of loops to be large enough and then by utilizing the `MUX` gate.

From another point of view, every operation should be performed fully homomorphically, so that the cryptosystem remains secure. Hence, even trivial operations such as increasing a counter have to be executed fully homomorphically (see Section 5.3).

5.3 Supported Operations

Here are presented all the supported operations by the software wrapper of our algorithm other than the mandatory binary ones. This subsection consists of deep analyses of the binary logic behind the operations as if they were implemented fully homomorphically. We have implemented all operations for the standard C++ integer types as well as for arbitrary bit sequences represented as `vectors`.

Equality Check Let us have two n-bit sequences a and b. In order to check if they are one and the same, we only need `XNOR` and `AND` gates. First, we will perform `XNOR` on all pairs of bits (a_i, b_i) for all $i = 0, 1, \ldots, n-1$ (1 if they are identical and 0 otherwise). Then to

see if all pairs are identical we will take their AND, thus if the result is a 1, it follows that $a = b$, and 0 otherwise.

We do n times XNOR and $n - 1$ times AND. Hence, we need $2n - 1$ binary operations and the time can be expressed as $t = n \cdot t_{\mathsf{XNOR}} + (n-1) \cdot t_{\mathsf{AND}}$.

Minimum and maximum between two values Let a and b be two n-bit sequences and let c be the result ($c = 0 \Leftrightarrow a < b$ and vice versa). We will look at the two sequences backwards, meaning the a_0 is in fact the last bit of a. We start at position 0 and build our way up. Here we will use only the XNOR and MUX gates. We take the XNOR of a_i and b_i. If the result is 1, we let $c = 0$ and carry on, however, if it is 0, then $c = a_i$ (if $a_i = 0$, then $a < b$, and if $a_i = 1$, then $a > b$). The final value of c shows which number is smaller (or bigger, depending on the case).

We perform n times XNOR for the bits, n times MUX for updating c, and n more times for returning the minimal value between the two. Therefore, we need $t = n \cdot t_{\mathsf{XNOR}} + 2n \cdot t_{\mathsf{MUX}}$.

Addition Let a and b be two n-bit sequences, c the resulting n-bit sequence and d the carry, which is initially set to 0. The resulting i-th bit can be expressed as $c_i = (a_i \ \mathsf{XOR} \ b_i) \ \mathsf{XOR} \ d$ and the new carry can be computed as $d = x \ \mathsf{XOR} \ (x \ \mathsf{AND} \ d)$, where $x = a_i \ \mathsf{AND} \ b_i$.

All in all, we do $2n$ times AND and $3n$ times XOR. Hence, $t = 2n \cdot t_{\mathsf{AND}} + 3n \cdot t_{\mathsf{XOR}}$.

Subtraction Let a and b be n-bit sequences. We have that $a - b$ is equivalent to
a + *the binary complement of* b. We do $3n$ times AND and $4n$ times XOR, for finding 2's complement of b and then to perform the addition. As a result, $t = 3n \cdot t_{\mathsf{AND}} + 4n \cdot t_{\mathsf{XOR}}$

Multiplication We have both implemented the pen-and-paper multiplication algorithm along with the Karatsuba multiplication algorithm in their pure versions. The latter being only for arbitrary long bit sequences as for numbers with less than 64 bits the performance difference is obsolete. We have not seen any significant discrepancy between the predicted performance difference of the two fully homomorphic versions and the real-life difference in the running time.

We have that $t = n^2 \cdot t_{\mathsf{AND}} + n \cdot t_{\mathsf{Add}}$ for the pen-and-paper method, whereas Karatsuba's complexity highly depends on the base we chose for switching back to the pen-and-paper method.

Division We implement the pen-and-paper method for dividing two integer numbers with a slight modification to ensure the security of the operation. Let a and b be two n-bit sequences, we want to compute a/b. On each step we try to subtract b shifted k places left (k ranges from n to 0) from a. If a is bigger than the shifted b, then we perform subtraction, otherwise, we subtract 0 from a. However, when we shift b for some numbers we may get a zero as we work with a fixed number of permitted. Hence, if the shifted number is zero we try to subtract from a the biggest possible number, namely $2^n - 1$, so as to avoid collisions.

For the complexity we get $t = n \cdot (t_{\mathsf{MUX}} + 2 \cdot t_{\mathsf{Min}} + 2 \cdot t_{\mathsf{Equality}} + t_{\mathsf{Sub}})$.

If statements As we already mentioned it, fully homomorphic encryption preserves operations, but not logic. Hence, the use of condition-dependent operations is limited. In order to implement an If-statement we use the MUX gate. It works similarly to a normal If – depending on the encrypted bit, it returns one of two values. It can be interpreted as $a \ ? \ b \ : \ c$, where a, b and c are encrypted bits (usually, $b = Enc(1)$ for true and $c = Enc(0)$ for false). Therefore, to mimick the work flow of an If, we will 'multiply' every operation by the result. If the result is an encrypted 0, we do nothing, and if we get an encrypted 1, we perform the operation.

Example. If we want to increase a counter by one given that two values are identical, we can do that in C++ in the following way:

```
if(a == b)
    count++;
```

However, fully homomorphically implemented it becomes:

```
count = count +
+ MUX(areEqual(a, b), Enc(1), Enc(0));
```

6 IMPLEMENTATION, TESTS AND APPLICATIONS

The code of our algorithm is written in C++ and can be found at https://github.com/dimitarch/FHE-Peformance-Project, where an up-to-date version of this paper is also uploaded. We have defined a class Computation that keeps track of the simulated operations and serves as a divider between multiple processes in a single program. It is equivalent to a single autonomous program. We implement the three simulated bits SimulatedGateBoostrappedBit, SimulatedCircuitBoostrappedBit and SimulatedLevelledBit that serve as wrappers for the normal bool type and the Computation class. We also implement a wrapper for normal encrypted bits – RealGateBootstrappedBit (currently only for TFHE). So far we have implemented a wrapper for 32-bit and 64-bit integers. A comprehensive set of tests are provided for each operation.

We did a number of tests on classical algorithms, such as searching for an element, counting occurrences and sorting an array (manual estimations can be found in [3, 7]). We note that searching for an element and counting occurrences have the same fully homomorphic complexity, therefore, it is sufficient to display only one of them. We have also run several of those tests with the normal fully homomorphic operations, so as to see how well our algorithm approximates the execution time. Tables 2 and 3 show the summarized results for basic searching and bubble sort (the TFHE library was used for the simulations).

Numbers	Time	σ	Real-time
40 16-bit numbers	18.4471	0.144076	18.4542
40 32-bit numbers	36.8942	0.288152	37.2048
300 16-bit numbers	138.353	1.08057	139.2811
300 32-bit numbers	276.706	2.16114	280.0223
1000 16-bit numbers	461.177	3.6019	Na
1000 32-bit numbers	922.354	7.20381	Na

Table 2: Searching for an element

Numbers	Time (in secs)	σ (in secs)
40 16-bit numbers	1798.59	14.0474
40 32-bit numbers	3597.18	28.0949
300 16-bit numbers	103419	807.727
300 32-bit numbers	206838	1615.45
1000 16-bit numbers	$1.15179 \cdot 10^6$	8995.76
1000 32-bit numbers	$2.30358 \cdot 10^6$	17991.5

Table 3: Sorting an array

It is evident that our algorithm gives good enough estimations of the running time compared to the manually executed programs. Homomorphic analysis of more classical algorithms follow.

Basic Search Let us have an array of length n and an element x we are searching for in this array. In the ordinary search algorithm we start by consecutively executing Equality check between x and the elements of the array. Then, we perform OR on the resulting bits to see if x is present. The fully homomorphic complexity is $t = n \cdot t_{\mathsf{XNOR}} + (n-1) \cdot (t_{\mathsf{AND}} + t_{\mathsf{OR}})$.

Binary Search Let us have a sorted array of length n and an element x we are searching for in this array. The algorithm starts by picking the middle element and comparing it with x. Then depending on which one is smaller we move onto one of the two halves of the array. However, as the binary search is fully homomorphically, we get the result of the comparison as an encrypted bit. Hence, by knowing which half we are supposed to move in we also get the plaintext version of the comparison result. Since we now know how a bit is encrypted, the fully homomorphic scheme is made vulnerable and impractical.

Bubble Sort Let us have an array of length n. Generally, Bubble sort has complexity of $O(n^2)$. Due to the nature of homomorphic operations, we are obliged to perform all homomorphic comparisons and element swaps. In its fully homomorphic form always perform $\frac{n(n-1)}{2}$ comparisons and $32n(n-1)$ MUX-gate executions.

Selection Sort Let us have an array of length n. Selection sort and Bubble sort have the same underlying idea. Since Selection sort requires us to find and extract the minimal element, we need to store the indices in encrypted form, as well. This makes the fully homomorphic complexity bigger than that of Bubble sort.

Merge Sort Let us have an array of length n. We begin with dividing the array into sub-arrays until we reach all sub-arrays of size 1. Then, we consecutively merge them. However, coming from a similar argument as in Bubble sort, we need to execute all comparisons and the accompanying element swaps. Hence, Merge sort's complexity is the same as that of Bubble sort.

Depth-First Search Let $G(V, E)$ be a graph with set of vertices V and set of edges E. Depth-First Search relies on recursively executing itself for every not-visited-yet vertex. In fully homomorphic form, however, we cannot execute the circuit only for the unvisited neighbours of a given vertex. Hence, we introduce an artificial vertex 0, which does not have any neighbours, and serves as the bottom of the recursive process.

Algorithm 1 Depth-First Search

```
procedure DFS(vertex u)
    u.visited = Enc(true)
    for v ∈ u.Neighbours do
        v = v AND (!v.visited)
        DFS(v)
```

Breadth-First Search Let $G(V, E)$ be a graph with set of vertices V and set of edges E. Breadth-First Search relies on storing the to-be-visited vertices in a queue both in its iterative and recursive versions. Thus, to ensure the security of the scheme, we again introduce an artificial vertex 0. In the ordinary version when we do not input a vertex in the queue as it has been visited, in the fully homomorphic version we input 0. However, to know when to stop the process, we need to know when the queue consists of only 0s. If we were to know such information explicitly (as normal `true` or `false`), then we have compromised the security of the FHE scheme. Hence, Breadth-First Search cannot be implemented fully homomorphically.

7 CONCLUSION AND FUTURE WORK

On balance, we have carried out a thorough analysis of fully homomorphic schemes. We have devised an algorithm along with a set of supported operations for approximating the time execution of fully homomorphic programs. We have run a number of test programs to see what exactly the expected time is and then compared them to manually computed execution times. However, our algorithm does not take into account FHE libraries with multithreading and specialised circuit (and levelled) bootstrapping as they are yet to be fully developed. We plan to expand the number of analysed libraries, for example the HELib, based on the BGV levelled fully homomorphic scheme [12–14]. Finally, our ultimate objective is the creation of a fully-operating C++ library for testing fully homomorphic programs.

ACKNOWLEDGMENTS

We would like to acknowledge the High School Science Institute in Mathematics and Informatics for hosting the 2018 Summer Research School and for making it possible for us to conduct this research.

REFERENCES

[1] Abbas Acar, Hidayet Aksu, A. Selcuk Uluagac, and Mauro Conti. 2017. A Survey on Homomorphic Encryption Schemes: Theory and Implementation. *CoRR* abs/1704.03578 (2017).
[2] Sergiu Carpov, Malika IzabachÃĺne, and Victor Mollimard. 2018. New techniques for multi-value homomorphic evaluation and applications. Cryptology ePrint Archive, Report 2018/622.
[3] Ayantika Chatterjee and Indranil Sengupta. 2015. Searching and Sorting of Fully Homomorphic Encrypted Data on Cloud. Cryptology ePrint Archive, Report 2015/981.
[4] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Cryptology ePrint Archive, Report 2016/870.
[5] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Improving TFHE: faster packed homomorphic operations and efficient circuit bootstrapping. Cryptology ePrint Archive, Report 2017/430.

[6] LÃĺo Ducas and Daniele Micciancio. 2014. FHEW: Bootstrapping Homomorphic Encryption in less than a second. Cryptology ePrint Archive, Report 2014/816.
[7] Nitesh Emmadi, Praveen Gauravaram, Harika Narumanchi, and Habeeb Syed. 2015. Updates on Sorting of Fully Homomorphic Encrypted Data. Cryptology ePrint Archive, Report 2015/995.
[8] Craig Gentry. 2009. *A fully homomorphic encryption scheme.* Ph.D. Dissertation. Stanford University.
[9] Craig Gentry and Shai Halevi. 2010. Implementing Gentry's Fully-Homomorphic Encryption Scheme. Cryptology ePrint Archive, Report 2010/520.
[10] Craig Gentry, Shai Halevi, and Nigel P. Smart. 2011. Better Bootstrapping in Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2011/680.
[11] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. Cryptology ePrint Archive, Report 2013/340.
[12] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. Cryptology ePrint Archive, Report 2014/106.
[13] Shai Halevi and Victor Shoup. 2014. Bootstrapping for HElib. Cryptology ePrint Archive, Report 2014/873.
[14] Shai Halevi and Victor Shoup. 2018. Faster Homomorphic Linear Transformations in HElib. Cryptology ePrint Archive, Report 2018/244.
[15] Daniele Micciancio and Jessica Sorrell. 2018. Ring packing and amortized FHEW bootstrapping. Cryptology ePrint Archive, Report 2018/532.
[16] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. 2009. Fully Homomorphic Encryption over the Integers. Cryptology ePrint Archive, Report 2009/616.
[17] Xun Yi, Russel Paulet, and Elisa Bertino. 2014. *Homomorphic Encryption and Applications.* Springer.