Overview of open source libraries for Fully Homomorphic Encryption (FHE) [*]

Guest contribution by
Oussama Amine[‡]

September 2019

Abstract

Several open-source (F)HE libraries are available, with each implementing one or several of the state of the art schemes. In order to summarize, to a certain extent, the state of affairs, we have written a short report on some of the important libraries in which we look at certain high-level features that constitute the differences between them.

1 Introduction

1.1 Nature of the report

This report focuses mostly on a subset of the publicly available libraries for F/HE. We do not cover all the available libraries but only those that have a relatively large development basis. We also neither go into the technical details of the implementations nor give a general description of their workings. We present the big picture, describing their building blocks, how these blocks contribute to the implementation of the F/HE schemes in question and the external libraries on which they depend.

1.2 Context

Big data, cloud computing and the need for privacy are three pillars that can help anchor our understanding of the problem that homomorphic encryption (HE) contributes to solving. HE is a form of encryption that has the additional property of being compatible with a set of computational operators, for example $\{ADD(x,y), MUL(x,y)\}$.

[*]Research supported by the Fantom Foundation

[†]PhD candidate in Mathematics, University of Oslo

[‡]The author would like to thank Patricia Thaine for her feedback on earlier versions of this report as well as for many fruitful and insightful discussions.

That is:

$$Decrypt(Evaluate(f, Encrypt(x))) == f(x)$$

where f is a function often represented as a set of arithmetic gates i.e. a circuit.

The first plausible construction of a fully homomorphic encryption scheme was proposed in Gentry (2009) [9]. Prior to Gentry's breakthrough, there had been many attempts to solve this problem, but all of these can be said to belong to the category of *Partially HE schemes*. These are HE schemes that support only one type of computational operator (i.e. either Add or Mult). An exception to this is that developed by Bonch et al. (2005) [1], which can be categorized as a *Somewhat HE scheme (SWHE)*, namely a scheme that supports the homomorphic evaluation of only a subset of the set of all possible circuits. In Bonch et al. [1], the circuits are allowed to have an arbitrary number of additions but only one multiplication.

Gentry's work [9] was the first to show that it was possible to realize Fully HE (FHE) both theoretically and practically. Gentry's approach, also sometimes called Gentry's blueprint, is based on three ingredients:

1. A SWHE scheme
2. Squashing of Decrypt
3. Bootstrapping.

It is important to mention that this three-ingredient blueprint played a defining role for all of the available practical implementations of HE schemes and that improvements and trade-offs between these three ingredients are to a large extent what separates them into the three-generation classification:

1. First generation: This is essentially Gentry's 2009 proposal and its practical implementation in Gentry-Halevi (2011) [10]. These schemes suffer from the problem of exponential growth of the so-called noise in the number of multiplication gates. The noise can be understood as a quantity that helps "drown" out data and thus makes it hard to infer anything meaningful about it. The rapid growth of noise made bootstrapping difficult, and additional transformations at the level of the Decrypt function where necessary, i.e. squashing. It is also worth mentioning that this squashing is based on certain non-standard extra hardness assumptions, and thus the security of the scheme is affected.
2. Second generation: The problem of exponential noise growth makes the bootstrapping step from a SWHE to a FHE scheme difficult although still feasible through the squashing procedure of the Decrypt circuit. Second-generation schemes are the results of various techniques aimed at better noise management, thus enabling the construction of practical SWHE schemes that support the evaluation of general circuits (of polynomial size) with only a restriction on the number of multiplications. Thus, while in the first generation the SWHE was constructed with the ultimate aim of evaluating a (squashed if necessary) Decrypt circuit plus one NAND gate, the SWHE in the second generation can evaluate any general circuit of predetermined size. This distinction is emphasized by referring to these as Leveled FHE (LFHE) schemes.
3. Third generation: These schemes do away with the two most important techniques that make up the second generation, namely key switching and modulus switching. This improves performance but at the expense of changing the representation of the ciphertext into one based on matrices. This makes many of the optimization developed for generation-two schemes incompatible with this generation, e.g. packing.

2 The libraries

The following are common to all libraries:

- All are written in C++;
- All are based on RLWE for speed and efficiency (this does not apply to Palisade as it is more of a framework for general lattice cryptography);
- All are second-generation with SIMD support, except for TFHE and its predecessor FHEW.

2.1 SEAL

The SEAL library[1], made open source by Microsoft towards the end of 2018, is a library with no dependencies that implements two (levelled) second-generation Homomorphic schemes, namely:

1. The scale-invariant B/FV scheme of Brakerski (2012) [2], and its implementation in Fan et al. (2012) [8], where modulus switching is avoided by the use of a slightly different encryption function than the one in Brakerski et al. (2014) [3]. Other than this, [3] and [8] are similar in spirit.
2. The CKKS scheme, as introduced in Cheon et al. (2017) [5]. This scheme is the first to provide native support for fixed-point arithmetics. This, however, comes at the cost of giving only approximate results. Nevertheless, for real-world applications, CKKS is a strong choice, as demonstrated by the results in [14].

The SEAL library has support for SIMD-type operations for both schemes. SEAL does not support bootstrapping and does not have external dependencies.

2.2 HElib

The HElib library[2] is designed and maintained by IBM. The first implementation included one for the Brakerski et al. (2014) scheme [3] and provided support for data manipulation instructions, e.g. the packing techniques as introduced in Smart et al. (2014) [16]. As of the 1.0.0 beta release, HElib also includes partial support for the CKKS scheme. The HElib library is broadly composed of four parts:

1. The bottom layer: Implements the mathematical structures and various other utilities needed by the top layers. This layer makes heavy use of the NIT [15] and GMP [11] external libraries.
2. The second layer: Implements the Double-CRT polynomial representation. This is the most important building block for HElib from an implementation point of view.
3. The third layer: Implements the scheme itself, i.e. Decrypt, Encrypt, (native) Evals and other methods related to the modulus/key switching techniques.
4. The top layer: Provides interfaces that permit arrays of plaintext to be worked on simultaneously while using the packed representation.

A diagram detailing the different components can be found on page 4 of [13]

2.3 PALISADE

PALISADE[3] is a general-purpose library providing implementations of various building blocks for lattice-based cryptography along with implementations of advanced lattice-based cryptographic protocols such as public-key encryption and homomorphic encryption. This modular design approach makes it possible to achieve both implementations of standard protocols that can be used

[1]https://github.com/microsoft/SEAL
[2]https://github.com/homenc/HElib
[3]https://git.njit.edu/palisade/PALISADE

out of the box for building applications on top and as a platform for more advanced users, allowing experimentation and the possibility to combine their specific implementations with those provided by the library. This modular approach, combined with the multi-level abstraction design approach, makes PALISADE a versatile library. PALISADE is composed of the following layers that sit on top of one another:

1. At the lowest and most basic level sits the **Primitive Math layer**, which supports basic modular arithmetic operations and multi-precision arithmetic. This layer includes efficient implementations of several number-theoretic algorithms (e.g. number-theoretic transform and Fermat-theoretic transform) as well as samplers of the discreet Gaussian distribution and other utilities. The interface for the math functionality is provided by both custom multi-precision libraries and external ones such as NTL and GMP.
2. On top of the previous layer sits the **Lattice Operation layer**. This layer supports all lattice constructs and is used to provide implementations of the various **Poly** classes, which are in turn used to build the **Plaintext** and **Ciphertext** classes. This layer is used to work in polynomial rings, which are the native elements that the crypto layer works with.
3. The **Crypto layer** provides the cryptographic functionalities (e.g. Encryption, decryption methods and container classes for parameters specific to those schemes) required for each specific cryptographic protocol (e.g. PKE, SWHE ...) This layer manipulates **Plaintext** and **Ciphertext** objects and uses the former to communicate with the Encoding layer.
4. The **Encoding layer** sits on top of the lattice operations layer and provides all the classes and methods that can be used to transform the raw plaintext message into a **Plaintext** object and back.

In terms of HE scheme capabilities, PALISADE most notably implements the second generation B/FV[8] and BGV [3] schemes and some of their variants.

2.4 TFHE

This scheme and the library with the same name that implements it are part of the third generation (some call it the fourth generation and reserve the term third generation for Gentry et al. (2013) [12]), and it is marked, in contrast to first-generation schemes, by its philosophy of making bootstrapping a part of the scheme from the get-go.

This is part of a new wave of schemes that one can call bootstrapped schemes. These schemes are marked by their integration of bootstrapping as a key component of the scheme. This bootstrapping or, to be more precise, refreshing procedure aims at keeping the noise at a certain appropriate level before every (homomorphic) evaluation. Moreover, the refresh procedure is applied to the output of each gate instead of at its input.

More precisely, in FHEW [7], which is the predecessor to TFHE [6], it is shown that if one works with the universal set of gates consisting of the NAND gate then one can implement this operation homomorphically, namely HomoNAND, as a simple combination of additions, albeit it in a different setting than the original one. A requirement for the correctness of this implementation, i.e. of HomoNAND, is that the two inputs (i.e. ciphertexts) are of a certain form, and this form is the result of a so-called refresh process. This refresh process makes up the bulk of the computational work in the homomorphic evaluation and, in fact, it can be shown that the cost of HomoNAND is essentially the cost of a single refresh.

In Ducas et al. (2015) [7], it is claimed that the evaluation of this HomoNAND gate (obviously, with the refresh) is achieved in under a second on a personal computer. In Chillotti et al. (2016) [6], this result is improved to under 0.1 second per gate. Both TFHE and FHEW rely on efficient implementations of FFTW.

References

[1] Boneh, Dan, Eu-Jin Goh, and Kobbi Nissim. ”Evaluating 2-DNF formulas on ciphertexts.” Theory of Cryptography Conference. Springer, Berlin, Heidelberg, 2005.

[2] Brakerski, Zvika. ”Fully homomorphic encryption without modulus switching from classical GapSVP.” Annual Cryptology Conference. Springer, Berlin, Heidelberg, 2012.

[3] Brakerski, Zvika, Craig Gentry, and Vinod Vaikuntanathan. ”(Leveled) fully homomorphic encryption without bootstrapping.” ACM Transactions on Computation Theory (TOCT) 6.3 (2014): 13.

[4] Brakerski, Zvika, and Vinod Vaikuntanathan. ”Efficient fully homomorphic encryption from (standard) LWE.” SIAM Journal on Computing 43.2 (2014): 831-871.

[5] Cheon, Jung Hee, et al. ”Homomorphic encryption for arithmetic of approximate numbers.” International Conference on the Theory and Application of Cryptology and Information Security. Springer, Cham, 2017.

[6] Chillotti, Ilaria, et al. ”Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds.” International Conference on the Theory and Application of Cryptology and Information Security. Springer, Berlin, Heidelberg, 2016.

[7] Ducas, Léo, and Daniele Micciancio. ”FHEW: bootstrapping homomorphic encryption in less than a second.” Annual International Conference on the Theory and Applications of Cryptographic Techniques. Springer, Berlin, Heidelberg, 2015.

[8] Fan, Junfeng, and Frederik Vercauteren. ”Somewhat Practical Fully Homomorphic Encryption.” IACR Cryptology ePrint Archive 2012 (2012): 144.

[9] Gentry, Craig: A fully homomorphic encryption scheme. Vol. 20. No. 09. Stanford: Stanford University, 2009.

[10] Gentry, Craig, and Shai Halevi. ”Implementing gentry’s fully-homomorphic encryption scheme.” Annual international conference on the theory and applications of cryptographic techniques. Springer, Berlin, Heidelberg, 2011.

[11] https://gmplib.org/

[12] Gentry, Craig, Amit Sahai, and Brent Waters. ”Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based.” Annual Cryptology Conference. Springer, Berlin, Heidelberg, 2013.

[13] http://people.csail.mit.edu/shaih/pubs/he-library.pdf

[14] http://www.humangenomeprivacy.org/2018/index.html

[15] https://www.shoup.net/ntl/

[16] Smart, Nigel P., and Frederik Vercauteren. ”Fully homomorphic SIMD operations.” Designs, codes and cryptography 71.1 (2014): 57-81.