Introduction to Homomorphic Encryption and Schemes

Jung Hee Cheon, Anamaria Costache, Radames Cruz Moreno, Wei Dai, Nicolas Gama, Mariya Georgieva, Shai Halevi, Miran Kim, Sunwoong Kim, Kim Laine, Yuriy Polyakov, and Yongsoo Song

1 Introduction to Homomorphic Encryption

Homomorphic encryption (HE) enables processing encrypted data without decrypting it. This technology can be used, for example, to allow a public cloud to operate on secret data without the cloud learning anything about the data. Simply encrypt the secret data with homomorphic encryption before sending it to the cloud, have the

The original version of this chapter was revised: Revised Chapter 1 has been uploaded to Springerlink. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-77287-1_15

J. H. Cheon (✉)
Seoul National University, Seoul, Republic of Korea
e-mail: jhcheon@snu.ac.kr

A. Costache
Norwegian University of Science and Technology, Trondheim, Norway; Intel AI Research, San Diego, CA, USA

R. C. Moreno
Microsoft Research, Redmond, WA, USA
e-mail: radames.cruz@microsoft.com

W. Dai · K. Laine
Cryptography and Privacy Research Group, Microsoft Research, Redmond, WA, USA
e-mail: wei.dai@microsoft.com; kim.laine@microsoft.com

N. Gama · M. Georgieva
Inpher, Lausanne, Switzerland
e-mail: nicolas@inpher.io; mariya@inpher.io

S. Halevi
Algorand Foundation, Yorktown Heights, NY, USA
e-mail: shaih@alum.mit.edu

K. Lauter et al. (eds.), *Protecting Privacy through Homomorphic Encryption*,
https://doi.org/10.1007/978-3-030-77287-1_1

cloud process the encrypted data and return the encrypted result, and finally decrypt the encrypted result. Here is a simplistic "hello world" example using homomorphic encryption:

```
# Every encryption needs a secret key. Let's get one of those.
myEncryptionKey = generateEncryptionKey()

# Now we can encrypt some very secret data.
encrypted5  = encrypt(myEncryptionKey,  5)
encrypted12 = encrypt(myEncryptionKey, 12)
excrypted2  = encrypt(myEncryptionKey,  2)
# We have three ciphertexts now.

# We want the sum of the first two.
# Luckily we used homomorphic encryption, so we can actually
do this.
encrypted17 = addCiphertexts(encrypted5, encrypted12)

# Maybe we want to multiply the result by the 3rd ciphertext.
encrypted34 = multiplyCiphertexts(encrypted17, encrypted2)

# See that? We operated on ciphertexts without needing the key.

# But no matter what we compute, the result is always encrypted.
# To actually see the final result, we have to use the key.
decrypted34 = decrypt(myEncryptionKey, encrypted34)
print(decrypted34) # This should print '34'
```

Homomorphic encryption today falls into the following common categories: partially homomorphic (weakest notion), leveled fully homomorphic, and fully homomorphic encryption (strongest notion). Partially homomorphic encryption supports only one type of operation, e.g. addition or multiplication. Leveled fully homomorphic encryption supports more than one operation but only computations of a predetermined size (typically multiplicative depth). Fully homomorphic encryption (FHE) supports arbitrary computation on encrypted data and is the strongest notion of homomorphic encryption.

M. Kim
Department of Computer Science and Engineering, Ulsan National Institute of Science and Technology (UNIST), Ulsan, Republic of Korea
e-mail: mirankim@unist.ac.kr

S. Kim
University of Washington Bothell, Bothell, WA, USA
e-mail: sunwoong@uw.edu

Y. Polyakov
Duality Technologies, Newark, NJ, USA
e-mail: polyakov@njit.edu

Y. Song
Seoul National University, Seoul, Korea
e-mail: y.song@snu.ac.kr

1.1 Plaintexts and Operations

Computation on encrypted data in homomorphic encryption preserves the same computation on the underlying plaintext. In the example above, we encrypted integers and then added and multiplied them. There are other types of data that we may want to encrypt and other operations that we may want to perform. For example:

- Encrypt bits and perform logical AND, OR, XOR operations on the ciphertexts.
  $0\ AND\ 1 \rightarrow 0$, $0\ OR\ 1 \rightarrow 1$, $1\ XOR\ 1 \rightarrow 0$
- Encrypt small integers and perform addition and multiplication, as long as the result does not exceed some fixed bound, for instance, if the bound is 10,000
  $123 + 456 \rightarrow 579$, $12 \times 432 \rightarrow 5184$, $35 \times 537 \rightarrow$ *overflow*
- Encrypt 8-bit unsigned integers (between 0 and 255) and perform addition and multiplication modulo 256
  $128 + 128 \rightarrow 0$, $2 \times 129 \rightarrow 2$
- Encrypt fixed-point numbers and perform addition and multiplication with the result rounded to a fixed precision, for instance, two digits after the decimal point
  $12 + 42 + 1.34 \rightarrow 13.76$, $2.23 \times 5.19 \rightarrow 11.57$

Different homomorphic encryption schemes support different plaintext types and different operations on them.

1.2 Vectors and Special-Purpose Plaintext Data Types

Some homomorphic encryption schemes, such as BGV, BFV, and CKKS, support "packing" – or "batching" – many plaintexts into a single ciphertext. They encrypt vectors of elements, perform element-wise operations, and move elements around in the vector:

- Encrypt vectors of any of the above types and perform operations element-wise

$$(1, 0, 1, 0)\ \ AND\ \ (1, 0, 0, 1) \rightarrow (1, 0, 0, 0)$$

$$(1, 2) \times (3, 4) \rightarrow (3, 8)$$

$$(1.1, 2.2) + (5.5, 6.6) \rightarrow (6.6, 8.8)$$

- and rotation on element's positions

$$rotLeft1\ (\ (1, 2, 3, 4)\ ) \rightarrow (2, 3, 4, 1)$$

These element-wise and data-movement operations are often called *SIMD operations* (single-instruction multiple-data).

Many homomorphic encryption schemes also support various special-purpose plaintext data types. While not described in this document, we briefly list some of them below.

- Multi-precisions integers modulo a very large integer of the form $p^n + 1$;
- Vectors over finite fields (e.g. useful for evaluation of the AES cipher);
- Polynomials modulo $X^n + 1$(e.g. for convolution products).

Some of the most promising homomorphic encryption schemes today, such as BFV, BGV, CKKS, DM, and CGGI, are implemented in open-source libraries. All these schemes have unique advantages and drawbacks depending on the types of computation one wants to perform.

1.3 Ciphertexts

One thing that all contemporary homomorphic encryption schemes have in common is that in all of them each ciphertext is an *array of integers* of fairly high dimension (at least a few hundred integers, and sometimes many thousands).

1.4 Symmetric vs. Public-Key Homomorphic Encryption

In the example code from above we used the same key for encryption and decryption; this type of encryption is called *symmetric encryption*. In contrast, public-key encryption (also called asymmetric encryption), uses two different keys: a secret key for decryption and a public key associated to the secret key for encryption.

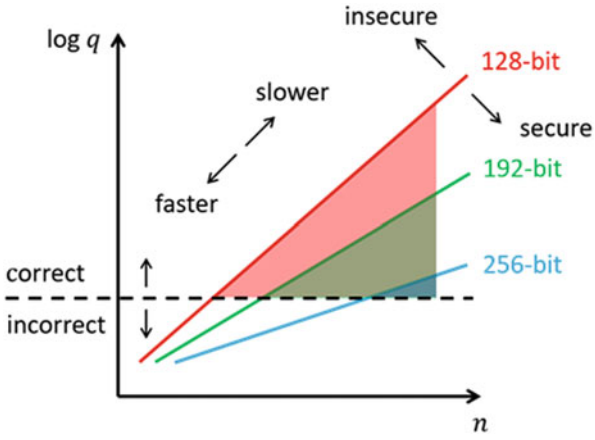
Homomorphic encryption can be instantiated as either symmetric encryption or public-key encryption, with encrypted computation capabilities. It provides the following fundamental operations:

Operation	Symmetric encryption	Public-key encryption
Key generation	secret key	secret key → public keys
Encryption	plaintext, secret key → ciphertext	plaintext, public key → ciphertext
Decryption	ciphertext, secret key → plaintext	ciphertext, secret key → plaintext
Operations	ciphertext (and plaintext) → ciphertext	

1.5 Parameters and Security

Instantiating any encryption scheme – homomorphic or otherwise – requires setting some parameters, for example, to determine the key size or the security level. For homomorphic encryption, the parameters influence not just security but also the

Fig. 1 Parameter selection



plaintext type and the computations that can be performed. The most prominent parameters that must be set for contemporary HE schemes are the following:

- Ciphertext dimension n corresponds roughly to the number of integers in each ciphertext;
- Ciphertext modulus q bounds the size of each integer in the ciphertext array.

In general, the security level *increases as n grows and decreases as q grows*. On the other hand, the larger q is, the more complex computations can be performed on ciphertexts of the encryption scheme: Ciphertexts in these encryption schemes contain a *noise component* (which is important for security), and that noise grows with each operation. The encrypted result can only be decrypted if the noise is smaller than q, hence using larger values of q imply that we can do more operations.

An illustration of the parameters n and q and their influence on the level of security is sketched in Fig. 1.

There are a few other parameters that influence security of lattice-based HE schemes, such as the distribution from which the secret key is selected (and a few others).

The security of lattice-based HE schemes is based on the hardness of a mathematical problem called *Learning with Errors* (LWE) or a variant of it called *Ring Learning-with-Errors* (RLWE). The (R)LWE problem is believed to be hard for both classical and quantum computers under appropriate parameters. The HE security document[1] contains tables indicating the security levels for various choices of n and q. Those tables are based on the best-known attacks against LWE.

The tables in the HE security document should be used as follows: Once the size of q is known (as well as the secret-key distribution), one needs to consult the

[1]Published in 2017 on http://homomorphicencryption.org/white_papers/security_homomorphic_encryption_white_paper.pdf. An updated version is included in Chapter "Homomorphic Encryption Standard".

table and find the smallest value of n that provides the desired security level for this q-size. For example, suppose we use a ternary secret-key distribution and want to achieve 128 bits of security with a 200-bit modulus q. The third part of Table 1 in the security document says that a value of $n = 8192$ can support q moduli of size up to 218 bits, but $n = 4096$ can only support q moduli of size up to 109 bits. Hence the smallest n that we can use is $n = 8192$.

2 The BGV and BFV Encryption Schemes

This section includes a simplified introduction to the *Brakerski-Gentry-Vaikuntanathan* (BGV) encryption scheme [34] and the encryption scheme due to *Brakerski and Fan-Vercauteren* (BFV) [3, 6]. For a more technical description of the schemes, we refer the reader to the Further Information subsection below.

BGV and BFV are homomorphic encryption schemes whose security is based on the hardness of the *Ring Learning with Errors* (RLWE) problem. The plaintext type in both schemes consists of vectors of integers, with modular SIMD operations as described below.

The BGV and BFV schemes involve several parameters that determine the security level, functionality, and the plaintext data type supported by the scheme. These parameters are:

- Plaintext modulus p;
- Ciphertext modulus q;
- Ciphertext dimension n.

The plaintext modulus $p$determines an upper bound for the integer components of the *plaintext vectors* that are encrypted in the BGV and BFV schemes. For example, setting the plaintext modulus to $p = 31$ means that computing the product of an encrypted 5 and an encrypted 7will overflow and produce the result $5 \times 7 - 31 = 4$.

The ciphertext modulus q is the main functional parameter that determines the encrypted computation capabilities of the scheme. A ciphertext in the BGV or BFV scheme consists of an array of $2n$ integers between 0 and $q - 1$. As explained in the introduction, the larger the parameter q of an instance is, the more operations can be performed on encrypted data in that instance.

For a given value of q, the ciphertext dimension n determines the security level of the scheme, with larger n meaning higher security. At the same time, the ciphertext dimension n also influences the size of the *plaintext vector* which is encrypted into each ciphertext. Often – but not always – the size of the plaintext vector is equal to n.

2.1 Homomorphic Operations

Operations over encrypted data preserve the same operations modulo p on vectors of integers and always produce a ciphertext as output. The main operations are:

Two-Argument Operations

- Ciphertext-Ciphertext addition;
- Ciphertext-Plaintext addition;
- Ciphertext-Ciphertext multiplication;
- Ciphertext-Plaintext multiplication;
- Ciphertext-Ciphertext subtraction;
- Ciphertext-Plaintext subtraction.

Unary Operations

- Negation;
- Vector rotation.[2]

2.2 Parameter Selection

Typically the first parameter to select is the plaintext modulus p, that determines the width of the plaintext data type. In some applications the plaintext modulus needs to be large enough to accommodate the desired computation without overflow, other times an overflow is desired. Selecting appropriate plaintext modulus depends on details of the application, and is beyond the scope for this document.

The next parameter to select is ciphertext modulus q, which is primarily determined by the multiplicative depth of the desired encrypted computation; a higher depth requires a larger ciphertext modulus, and is typically slower. Therefore, the computation should be made as low depth as possible. For example, computing a product of four encrypted numbers A, B, C, and D is better done as $(A * B) * (C * D)$ rather than $A * (B * (C * D))$, as the former has lower multiplicative depth, and hence requires a smaller ciphertext modulus.

Once q is determined, the ciphertext dimension n should be selected to achieve the desired security level, using the tables in [9]. The application developer is advised to use a library that implements the [9] standard. We note that choosing the right table from the [9] document requires knowing certain details of the implementation, such as the secret key distribution.

2.3 A BGV/BFV Hello World Example

```
# We first must set the parameters p,q, and n
p = 31
q = 65537
```

[2] In some cases we have more involved data-movement operations than just rotations. See the Further Information section for more details.

```
n = 16
# Warning: this setting is completely insecure!!
# To get any kind of security with q=65537 we need at least n=512

# With these parameters, the size of the plaintext vectors is 8

# Generate the keys for these parameters
myPublicKey, mySecretKey = generateBFVkey(n, p, q)

# Encrypt data, each plaintext is a vector of 8 elements
encrypted_a  = encrypt(myPublicKey, [5, 11, 2, 0, 20, 3, 8, 11])
encrypted_b = encrypt(myPublicKey, [12, 7, 14, 11, 1, 2, 3, 24])
excrypted_c  = encrypt(myPublicKey, [2, 10, 15, 13, 6, 3, 2, 1])
# We have three ciphertexts now.

# Compute the sum of the first two.
encrypted_d = addCiphertexts(myPublicKey, encrypted_a,
encrypted_b)
# Encryption of vector [17, 18, 16, 11, 21, 5, 11, 4]

# Maybe we want to multiply the result by the 3rd ciphertext.
encrypted_e = multiplyCiphertexts(myPublicKey, encrypted_c,
                                  encrypted_d)
# Encryption of vector [3, 25, 23, 19, 2, 15, 22, 4]

# Then rotate by 2 to the right
encrypted_f = rotateBy2(myPublicKey, encrypted_e)

# To actually see the final result we have to use the key.
decrypted = decrypt(mySecretKey, encrypted_f)
print(decrypted)
# This should print [22, 4, 3, 25, 23, 19, 2, 15]
```

2.4 Further Information

Maintenance Operations

The BGV and BFV schemes also include some operations that have no effect on the underlying plaintext, but are nonetheless sometimes needed for implementation reasons.

- Ciphertext-Ciphertext multiplications and cyclic vector rotations have a side-effect of requiring a different secret key to decrypt the result than what was needed before the operation. These operations are therefore followed by a *key switching* operation to restore the secret key back to the original one.[3] The

[3] In some applications, key switching operations are avoided or delayed for the sake of optimization.

key switching operation for Ciphertext-Ciphertext multiplication is also called *relinearization*;
- Bootstrapping, which "refreshes" a ciphertext and reduces the level of noise in it, to support more computations. This operation is very expensive, and hence it is not often used (and sometimes it is not even implemented).
- Modulus switching, which sometimes follows the multiplication operation. This is used more in BGV, where it is needed to control the level of noise in a ciphertext. (It is rarely used in BFV, except for bootstrapping.)

Evaluation Keys

The key switching operations require the evaluator to have access to special public *evaluation keys*. These evaluation keys are generated by the owner of the secret key. In the context of Ciphertext-Ciphertext multiplication, these keys are often called *relinearization keys*; and in the context of rotation, they are sometimes called *rotation* or *Galois keys*.

Data Encoding

Prior to encrypting data with the BGV or BFV scheme, a separate *encoding* operation is required, which transforms source data (e.g. vectors of integers) into a native plaintext format for the scheme. After decryption, a corresponding *decoding* operation is required.

Data Movement Operations

For some setting of the parameters p and n, the native data-movement operations supported by the scheme may differ from just cyclic rotations. For example, in some cases the plaintext elements are arranged in a matrix with 2 rows and $n/2$ columns, with native operations of row-rotate and column-rotate.[4] Even in these cases, it is always possible to implement cyclic rotations using the native row- and column-rotations, as described in [11].

References for the BFV Encryption Scheme

The BFV scheme is a ring variant of the scale-invariant LWE scheme proposed by Brakerski [3]. The "textbook" (multi-precision integer arithmetic) variant of BFV is described in [6, 13]. Note that [13] provides tighter noise constraints than the original paper [6].

[4] For some parameters we get even higher-dimension hypercube formats.

The most efficient variants of BFV used in practice represent large integers in the Residue Number System (RNS). The RNS representation has a number of practical advantages over the conventional multi-precision positional number system (PNS) representation:

1. RNS works with native (machine-word size) integers: faster (up to 5–10x) than PNS.
2. Runtime in RNS scales (quasi-)linearly with integer size.
3. RNS dramatically improves memory locality.
4. Computations are easily parallelizable, and hence RNS supports efficient GPU/FPGA hardware implementations.

Two RNS variants of BFV are known in literature: [2] (based on integer arithmetic) and [10] (based on both integer and floating-point arithmetic). A comparison of the RNS variants is provided in [4].

The encoding of vectors of integers into a BFV plaintext is described in Appendix A of [13]. This batching/packing encoding technique is discussed at a more advanced level in [7].

The bootstrapping for BFV is described in [5]. Note that BFV bootstrapping is rarely used in practice, and is not currently supported by any open-source homomorphic encryption library.

The following libraries have open-source implementations of BFV (the variants are indicated in parentheses):

- Microsoft SEAL [2]
- PALISADE [2, 6, 10]
- Lattigo [14]

References for the BGV Encryption Scheme

The BGV encryption scheme was first described in [34], improving on a previous construction from [35]. Here too it is desirable to represent large integers in the Residue Number System (RNS), for the same reason as for the BFV encryption scheme. This implementation was described in [7]. The BGV encryption scheme is implemented in the HElib and PALISADE libraries. Bootstrapping for BGV was described in [1, 8, 12], and is implemented in HElib.

3 The CKKS Encryption Scheme

This section includes a simplified introduction to the *Cheon, Kim, Kim, and Song (CKKS)* encryption scheme [15]. For a more technical description of the scheme, we refer the reader to the Further Information section below.

CKKS is a homomorphic encryption scheme whose security relies on the hardness of the *Ring Learning with Errors* (RLWE) problem. The plaintexts are vectors of real numbers, represented as a fixed-point type. The scheme natively supports fixed-point arithmetic between these vectors in a SIMD manner.

The CKKS scheme involves several parameters that determine the security level, functionality, and precision supported by the scheme. These parameters are:

- Number of fractional bits f, corresponding to the accuracy of the computation;
- (Maximal) Ciphertext modulus q;
- Ciphertext dimension n.

We assume that every plaintext value is represented as a binary fixed-point number which has f fractional bits after the radix point. The value of f for a ciphertext can be adjusted after performing computations using a so-called *rescaling* procedure, which is a distinctive feature of CKKS.

The ciphertext modulus q is the main functional parameter that determines the encrypted computation capabilities of the scheme. A ciphertext of the CKKS scheme consists of an array of $2n$ integers modulo q. The larger the parameter q is, the more operations can be performed on encrypted data and at a higher precision. For a given value of q, the ciphertext dimension n determines the security level of the scheme, with larger n meaning higher security. We refer the reader to the parameter selection section for more details.

As noted above, CKKS allows us to encrypt multiple fixed-point numbers in a single ciphertext. The ciphertext dimension n also determines the size of the plaintext vectors, which is $n/2$.

3.1 Homomorphic Operations

All computations involving at least one encrypted input produce encrypted outputs. The main operations are:

Two-Argument Operations

- Ciphertext-Ciphertext addition;
- Ciphertext-Plaintext addition;
- Ciphertext-Ciphertext multiplication;
- Ciphertext-Plaintext multiplication;
- Ciphertext-Ciphertext subtraction;
- Ciphertext-Plaintext subtraction.

Ciphertext-Ciphertext and Ciphertext-Plaintext multiplications return a ciphertext whose scaling factor is explicitly the product of scaling factors of inputs.

Ciphertext-Ciphertext and Ciphertext-Plaintext additions require the scaling factors of the inputs to match.

Unary Operations

- Negation;
- Cyclic vector rotation;
- Rescaling.

Rescaling, which almost always follows the multiplication operation, is a unary operation that divides the scaling factor of input ciphertext by a specific factor. It controls the magnitude of scaling factors during homomorphic computation. Ciphertext modulus decreases after the rescaling operation, and further multiplication is not allowed if a ciphertext modulus is too small.

3.2 Parameter Selection

The number of fractional bits and the supported depth of the encryption scheme are main parameters to be considered. Encrypted evaluation of a circuit can be performed if the circuit depth does not exceed the bound determined by the parameters.

Precision loss and overflow are two major issues of fixed-point arithmetic. Ciphertexts in CKKS have inherent error after encryption or computation, which is controlled by the parameter f. A larger f means more accurate result, but the computational cost grows as f increases. At the same time, the magnitude of encrypted values must be kept sufficiently smaller than the ciphertext modulus q to ensure that no overflow occurs during computation.

The maximal ciphertext modulus q is primarily determined by the multiplicative depth of the desired circuit to be evaluated, and by the accuracy parameter f; higher depth and larger accuracy require a larger ciphertext modulus, and is typically slower. Therefore, a common optimization technique is to represent a computational task as a circuit with minimal depth. For example, computing a product of four encrypted numbers A, B, C, and D is better done as $(A * B) * (C * D)$ rather than $A * (B * (C * D))$, as the former has lower multiplicative depth, and hence requires a smaller ciphertext modulus.

Once q is determined, a lower bound on the ciphertext dimension n is now determined to achieve a desired security level, using the tables in [9]. The application developer is advised to use a library that implements the [9] standard and automatically selects the correct table, as choosing the right table requires knowing certain details of the implementation, such as the secret key distribution.

3.3 A CKKS Hello World Example

```
# We first must set the parameters f,q, and n
f = 2
q = 65537
n = 8
# We use a decimal representation with f = 2 fractional digits

# Warning: this setting is completely insecure!!
# To get any kind of security with q=65537 we need at least n=512
# With these parameters, the plaintext vectors have size 4

# Generate the keys for these parameters
myPublicKey, mySecretKey = generateCKKSkey(n, q)

# Encrypt data, each plaintext is a vector of 4 elements
encrypted_a  = encrypt(myPublicKey, [1.53, -11.53, 0.02, -3.32])
encrypted_b = encrypt(myPublicKey, [12.29, 7.52, -14.47, 11.01])
excrypted_c  = encrypt(myPublicKey, [2.64, 10.78, -15.30, 13.34])
# We have three ciphertexts now.

# We want the sum of the first two.
# Luckily we used homomorphic encryption, so we can actually.
 do this.
encrypted_d = addCiphertexts(myPublicKey,
 encrypted_a, encrypted_b)
# encrypting the vector [13.82, -4.01, -14.45, 7.69]

# Maybe we want to multiply the result by the 3rd ciphertext.
encrypted_e = multiplyCiphertexts(myPublicKey, encrypted_c,
                                  encrypted_d)
# encrypting the vector [36.48, -43.23, 221.09, 102.58]

# Then rotate by 2 to the right
encrypted_f = rotateBy2(myPublicKey, encrypted_e)

# To actually see the final result, we have to use the key.
decrypted = decrypt(mySecretKey, encrypted_f)
print(decrypted)
# This should print [221.09, 102.58, 36.48, -43.23]
```

3.4 Further Information

Data Encoding

Prior to encrypting data with the CKKS scheme, a separate *encoding* operation is required. The CKKS encoding incurs some loss of precision, hence the plaintext vector must first be multiplied by a scaling factor (which is determined by the parameters of the scheme), to ensure that the encoded value retains enough

precision. Then, the scaled vector is converted into a native plaintext format for the scheme. Ciphertexts implicitly store the scaling factor which may change during homomorphic computation. After decryption, a corresponding *decoding* operation is required.

The ciphertext modulus determines an upper bound for the components of the underlying *encoded plaintext* to guarantee its correct decryption. For example, setting the ciphertext modulus to $q = 1024$ means that an encryption of 12.34 with scaling factor of 32 is correctly decryptable but encrypting the same value with scaling factor 256 will result in overflow.

Maintenance Operations

The CKKS scheme also uses some operations that do not change the underlying plaintext (beyond some possible precision loss) but are nonetheless needed for implementation reasons.

- Ciphertext-Ciphertext multiplication and cyclic vector rotation have a side-effect of requiring a different secret-key to decrypt the result than what was needed before the operation. These operations are therefore followed by a *key switching* operation to convert the secret key back to the original one. The key switching operation for Ciphertext-Ciphertext multiplication is also called *relinearization*.
- *Bootstrapping*, which "refreshes" a ciphertext and raises the ciphertext modulus in it, to support more computations. This operation is expensive, and hence it is not often used (and sometimes it is not even implemented).

Evaluation Keys

The key switching operations require the evaluator to have access to special public *evaluation keys*. The evaluation key generation must be done by the secret key owner. In the context of Ciphertext-Ciphertext multiplication, these keys are often called *relinearization keys*; and in the context of rotation, they are sometimes called *rotation* or *Galois keys*. The bootstrapping procedure also requires such evaluation keys.

References for the CKKS Scheme

The CKKS encryption scheme was first proposed in [15]. For the same reason as for the BFV scheme, it is desirable to represent large integers in the RNS. Several RNS variants of the CKKS scheme have been proposed and implemented, including [17, 19, 20], and [21]. Modern HE libraries typically implement a combination of these RNS variants and often add their own optimizations/usability improvements. Bootstrapping for CKKS was described in [16, 18, 21].

Reference Implementations

The following libraries have open-source implementations of CKKS (the variants are indicated in parentheses):

- HEAAN/RNS-HEAAN
- HElib
- Lattigo
- Microsoft SEAL
- PALISADE

4 The DM (FHEW) and CGGI (TFHE) Schemes

4.1 Basic Concepts

This section includes a simplified introduction to the *Ducas-Micciancio* (*DM)* and *Chillotti-Gama-Georgieva-Izabachene CGGI schemes, based on* [25–29, 31, 32]. The DM scheme is often referred to as the FHEW scheme in literature, and the CGGI scheme is often referred to as the TFHE scheme. To distinguish the underlying schemes from their implementations in the FHEW and TFHE libraries, we adopt the naming convention based on authors' initials. For a more technical description of these schemes, we refer the reader to the Further Information subsection below.

DM and CGGI are homomorphic encryption schemes based on the *Learning with Errors* (LWE) problem and its ring variant, the *Ring Learning with Errors* (RLWE) problem. Common use-cases for these schemes are the encrypted evaluation of decision diagrams, comparisons, lookup tables and circuits.

The schemes can be used in two different modes: simple (automated) and advanced (manual). The simple mode automatically performs bootstrapping after each gate operation, providing the ability to evaluate arbitrary (typically Boolean) circuits. The simple mode is easy to configure (requires only one parameter) but can be less efficient than the advanced mode, especially when the circuit is known in advance. In the advanced mode, the user decides when to perform bootstrapping or other maintenance operation, and even has an option not to perform bootstrapping at all.

The simple mode is easy to use, it suffices to generate or compile a small Boolean circuit that corresponds to the application, and evaluate it gate by gate on encrypted inputs. The homomorphic evaluation time is proportional to the plaintext evaluation of the same circuit. If the application can be written in terms of binary decision diagrams and lookup tables, the developer will often achieve much better performance by using the advanced mode. Some speed-ups using advanced mode are illustrated in [26] (lookup table and comparison circuit).

These schemes involve the following parameters:

- Bits of security λ (main parameter in all modes);
- Ciphertext-specific computation budget measure (only in advanced mode).

The bits of security parameter λ are related to the ciphertext modulus q and ciphertext dimension n for other schemes. In the simple mode, all the parameters can be derived from λ only.

In the advanced mode, the computation budget serves as a measure for the number of homomorphic operations that can be run on a ciphertext before a bootstrapping is required. Once all computations for a given ciphertext are performed, the user has to manually call bootstrapping to reset the computation budget for further computations on the ciphertext.

4.2 Homomorphic Operations

Computations over encrypted data always produce a ciphertext as output. The simple mode supports operations on Boolean circuits. The advanced mode supports operations on Boolean circuits, integers, and fixed-precision fractional numbers.

Simple Mode Plaintext Space and Operations

In the simple mode, the plaintext is just a Boolean value, and the main operations for Boolean circuits are:

- Constants
  - ZERO/ONE
- Unary gate
  - NOT
- Binary gates
  - AND/NAND
  - OR/NOR
  - XOR/XNOR
  - ORNOT/ANDNOT
- Ternary gates
  - MUX
  - Majority/Minority

In all these gates, the inputs and outputs are ciphertexts only. There is no direct support for mixed plaintext/ciphertext inputs because a Boolean gate that takes a plaintext as an input can always be simplified: e.g. x AND 1 = x, x AND 0 = 0, x XOR 1 = NOT x.

A DM/CGGI Hello World Example (Using Simple Mode)

```
# We first must set the bits of security
lambda = 128

# Generate the keys for these parameters
myPublicKey, mySecretKey = generateKeys(lambda)

# Encrypt data, each plaintext is a boolean value
encrypted_a = encrypt(myPublicKey, 1)
encrypted_b = encrypt(myPublicKey, 1)
excrypted_c = encrypt(myPublicKey, 0)
# We have three ciphertexts now.

# Compute the AND of the first two.
encrypted_AND = EvalGate("AND", myPublicKey, encrypted_a,
                         encrypted_b)
# Encryption of 1 AND 1 = 1

# Maybe we want to compute OR of this with the 3rd ciphertext.
encrypted_ANDOR = EvalGate("OR", myPublicKey, encrypted_AND,
                           encrypted_c)
# Encryption of (1 AND 1) OR 0 = 1

# To actually be able to see the final result we have to use
      the key.
decrypted = decrypt(mySecretKey, encrypted_ANDOR)
print(decrypted)
# This should print 1
```

Advanced Mode Plaintext Space and Operations

In the advanced mode, the scheme supports different plaintext types, as well as elementary operations across these plaintext spaces, and each ciphertext carries its own computation budget. The combination of plaintext types and computation budgets determines whether an operation is allowed, and at which performance it will be executed.

The main plaintext structure is a vector of n elements, which supports vector additions and some other vector operations (see below), but vector multiplications are not supported. In this section, we explain the plaintext arithmetic and give a few examples.

The main plaintext structure is a vector of n fixed-point numbers between -0.5 and 0.5 (i.e., modulo 1), given with a precision $\pm\alpha$ (each ciphertext has its own precision). This vector is encrypted in an RLWE[5] ciphertext with noise rate α. For instance, a ciphertext that encrypts a coefficient 0.0042 with precision

[5]Here RLWE is represented mod 1, with all coefficients divided by q used in BFV, BGV, CKKS and DM.

$\alpha = 10^{-9}$ means that it can be decrypted as any number between $0.0042 - 10^{-9}$ and $0.0042 + 10^{-9}$. This inherent error is analogous to the error in floating-point arithmetic (in the case of approximate arithmetic) or can be eliminated by post-decryption rounding if the message space is discretized (in the case of exact arithmetic).

If the coefficient 0.4942 was encrypted in a ciphertext with a much higher noise rate $\alpha = 10^{-2}$, it could be decrypted as any value between 0.4842 and 0.5042, and the second one would appear as -0.4958 modulo 1. This overflow is similar to the BGV/BFV case (with *mod p* plaintext space), but with respect to real numbers *mod* 1. If such overflow is not explicitly wanted by the application, it probably means that the noise is too large, or that the inputs should be scaled down.

The supported homomorphic operations are:

- Element-wise addition and subtraction: $x + y$, $x - y$
  - $(0.0042, 0.0011, 0.0034) + (0.0074, 0.0089, 0.0011) \rightarrow (0.0116, 0.0100, 0.0045)$

  The result is always reduced modulo 1 (it can either be viewed as an expected behavior, or as an overflow condition which is mitigated by downscaling the space until every coefficient is much smaller than 1 like in the above example).
- Multiplication by a small public integer constant: noted $a * x$
  - $3 \times (0.0042, 0.0011, 0, 0034) \rightarrow (0.0126, 0.0033, 0.0102)$, for $a = 3$;
  - $123 * (0.0042, 0.0011, 0, 0034) \pm (\alpha = 10^{-5}) \rightarrow (0.517, 0.135, 0.418) \pm (\alpha = 10^{-3})$, for $a = 123$.

  Here, the factor 123 is rather large, if the input noise was $\pm 10^{-5}$, only 3 decimal digits of precision remain after scaling, since the noise amplitude also increases by a factor 123.
- (Anticyclic) shift by k positions: noted $rot_k(x)$, with k a public value
  - $rot_0(\ (0.0042, 0.0011, 0, 0034, 0, 0, 0)\ ) \rightarrow (0.0042, 0.0011, 0, 0034, 0, 0, 0)$, for $k = 0$
  - $rot_2(\ (0.0042, 0.0011, 0, 0034, 0, 0, 0)\ ) \rightarrow (0, 0, 0.0042, 0.0011, 0, 0034, 0)$, for $k = 2$
  - $rot_3(\ (0.0042, 0.0011, 0, 0034, 0, 0, 0)\ ) \rightarrow (0, 0, 0, 0.0042, 0.0011, 0, 0034)$, for $k = 3$

  Any coefficient that vanishes to the right of the vector appears back on the left side with the opposite sign, so with the same example, if $n = 6$.
  - $rot_4(\ (0.0042, 0.0011, 0, 0034, 0, 0, 0)\ ) \rightarrow (-0.0034, 0, 0, 0, 0.0042, 0.0011)$, for $k = 4$
  - $rot_5((0.0042, 0.0011, 0, 0034, 0, 0, 0)) \rightarrow (-0.0011, -0.0034, 0, 0, 0, 0.0042)$, for $k = 5$
- There are also more involved operations, such as selecting only one particular position, or all odd or even positions and canceling all other positions.

Any combination of the above addition/scaling/rotations is possible: e.g. $x + 2 * rot_1(x) + 5 * rot_2(y)$ means $x +$ twice x rotated by one position $+5$ times y rotated by two positions. The same expression can equivalently be factorized as $(rot_0 + 2rot_1).(x) + (5rot_2).(y)$, which isolates the linear transformations applied on each ciphertext. Applying a linear transformation on a ciphertext increases its noise (and hence the output error) by the norm of all rotation coefficients: the bigger the coefficients, the larger is the resulting noise, and the application designer must always ensure that the resulting noise remains small enough to decrypt the output.

Having all these definitions and constraints in mind, the user is free to use any plaintext vector, encrypted as an RLWE ciphertext, which can represent a real or fractional number (mod 1).[6]

If the application cannot be expressed in terms of element-wise addition, public scaling or rotation with public index and we need non-linear operations, we should use a different encryption scheme called RGSW (vector of RLWE ciphertexts). It provides the possibility to evaluate any secret linear transformation (homomorphically encrypted).

- *Homomorphic action (external product):* Given an RGSW ciphertext that encrypts a linear transformation f, and an RLWE ciphertext that encrypts a real vector x, obtain the encryption of $f(x)$.

The most useful applications of this concept are essentially:

- *BlindRotation:* Given a RGSW ciphertext that encrypts rot_k, and a RLWE ciphertext that encrypts x, obtain a RLWE encryption of $rot_k(x)$, where k remains secret.
- *PrivateSelection* (CMUX): Given a RGSW ciphertext that encrypts $c = 1$ or 0,[7] and two RLWE ciphertexts that encrypt x and y, obtain an RLWE encryption of the selection $c?x:y$ (written like in C), which is equal to x when $c = 1$ and y when $c = 0$.

The CMUX is a building block for the evaluation of any binary decision diagrams or deterministic finite automata (DFA), like the lookup table or the automata in Figs. 2 and 3, where each selector is one CMUX gate.

These two operations above only add a constant amount of noise on top of the input RLWE ciphertexts, allowing to chain a large amount of these operations with negligible noise growth, and hence to build complex decision diagrams or deterministic automata. Many arithmetic circuits correspond to simple decision diagrams, the most famous of them is the decryption function, which is used to bootstrap ciphertexts in the simple mode.

[6] Fractional number mod 1 corresponds to an integer mod p, as in BGV/BFV, divided by p

[7] 1 is the identity function

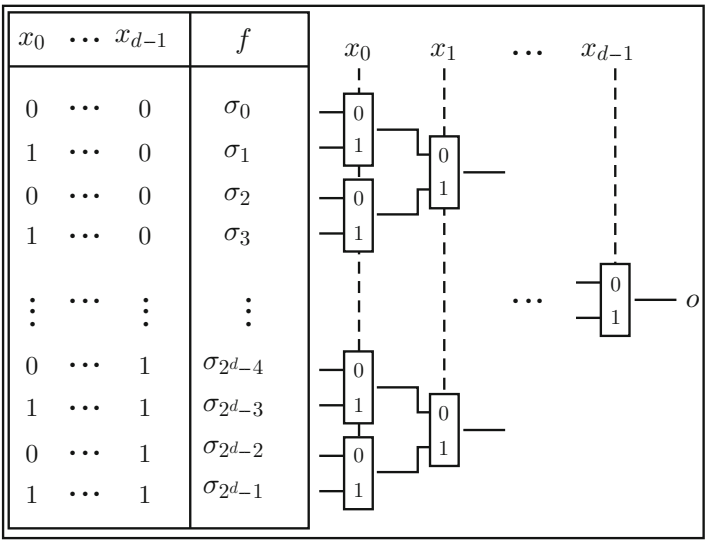
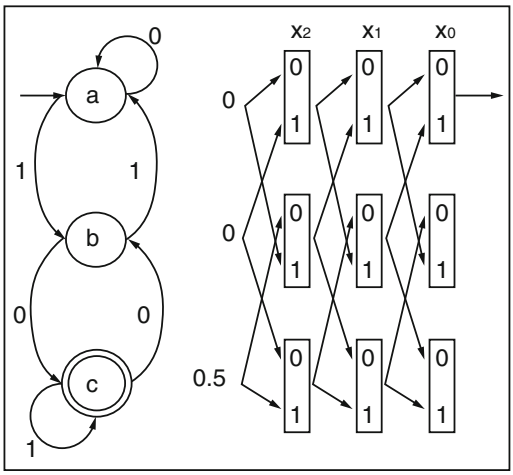


Fig. 2 Evaluation of lookup table.V

Fig. 3 Evaluation of DFA



Advanced-Mode CGGI Hello World Example (Corresponds to the DFA in Fig. 1)

```
# We first must set the bits of security and the noise-rate
lambda = 128
alpha = 2^-15

# Generate the keys for these parameters
myPublicKey,mySecretKey = generateKeys(lamda, alpha)
```

```
# encrypt each letter with RGSW
encrypted_x = [ encryptRGSW(myPublicKey, 0),
                encryptRGSW(myPublicKey, 1),
                encryptRGSW(myPublicKey, 0) ]

# the initial state values are trivial RLWE ciphertexts
a = RLWE(0)
b = RLWE(0)
c = RLWE(0.5)

For i = 3 to 1
    # evaluate each transition
    newA = CMux(encrypted_x[i],b,a)
    newB = CMux(encrypted_x[i],a,c)
    newC = CMux(encrypted_x[i],c,b)
    (a,b,c) = (newA,newB,newC)
EndFor
# To actually see the final result, we have to use the key.
decrypted = decrypt(mySecretKey, a)
Return decrypted
```

4.3 Further Information

Advanced Notes on Parameters

The computation budget can be equivalently expressed as the standard deviation of the ciphertext noise (noise rate α) for implementations of CGGI in the TFHE library, or to the modulus q for modular instantiations of DM and CGGI. The correspondence between the modulus q described in the security tables and the noise rate α is $q = 3.2/\sqrt{2\pi}\ \alpha$. This means, the noise grows at each operation until it reaches critical levels, or q can be rescaled down after each operation, until it reaches its minimum.

In the advanced mode, there are many sets of parameters n and α (q), each corresponding to a specific (sub-)circuit that gets bootstrapped. The details of setting the computational budget are implementation-specific.

Some More Advanced Operations Are Supported

- *Addition and composition of transformations:* Given two RGSW ciphertexts encoding f and g, we can homomorphically obtain single RGSW ciphertexts

that encrypt respectively $f + g$ and $f \circ g$. This is in line with the original ring operations described in the GSW scheme [31], and it is used in DM bootstrapping [29].
- *Multiplication by the secret key s:* This allows the evaluation of polynomials in s, and by extension, unlocks a variant of BFV and CKKS schemes supporting homomorphic element-wise addition/multiplication either modulo p or on fixed point-numbers, and that can be combined with the above circuit operations (see Fig. 3 at the end of the section). This is discussed at a more advanced level in the Chimera extension of CGGI in [28].

Maintenance Operations (and More)

The DM/CGGI schemes also support some maintenance operations:

- (both DM/CGGI)
  - *Gate bootstrapping,* which "refreshes" the noise of a binary gate ciphertext. Unlike BFV, BGV and CKKS, the gate bootstrapping is fast, in the order of a few milliseconds [25, 29], and is always applied after each boolean gate in the simple mode.
  - *(Functional) Key switching* allows to switch between scalar and polynomial message spaces, and to apply any linear combination with small integer coefficients.
- (CGGI-specific)
  - *Circuit bootstrapping*, which "converts" a LWE ciphertext from the space {0,1/2} into a RGSW ciphertext of 0 or 1. The circuit bootstrapping is applied in the advanced mode to compose binary decision diagrams, and runs in 137 milliseconds [26].
  - *Functional bootstrapping*, allows to approximate homomorphically a non-linear real-valued function [23, 27].
- (DM-specific)
  - *Modulus Rescaling*, which follows almost all multiplication operations on representations modulo q. This operation simplifies the ciphertext to obtain shorter equivalent representation, with a fixed noise amplitude. This operation is implicit on representations modulo 1, where the precision of the ciphertext representation is always of the order of α.

Advanced Functionality in the CGGI Encryption Scheme

The CGGI scheme in advanced mode proposes two methods to operate on batched data to decrease the ciphertext expansion and to optimize the evaluation of look-up tables and arbitrary functions. The batching techniques provide the possibility to

use the computation slots at their maximal capacity, even if the function itself is not SIMD, or has very few bits of output.

CGGI also extends the automata logic, to the leveled evaluation of deterministic weighted finite automata (WFA). These improvements speed up the evaluation of most arithmetic functions in a batched advanced mode, with a noise overhead that remains additive (more information is described in [26]).

Difference Between DM and CGGI

The CGGI scheme supports the both simple and advanced modes (with a special circuit bootstrapping proposed in [26]), DM currently supports just the simple mode, but can be generalized.

The main difference between the CGGI and DM schemes in the simple mode is in the bootstrapping procedure used for refreshing the ciphertexts [32]. CGGI uses the Gama–Izabachene–Nguyen–Xie bootstrapping procedure [30] based on CMUX gates while CGGI uses the Alperin-Sherif–Peikert bootstrapping procedure [22] via the composition of GSW ciphertexts.

Variants of DM/CGGI

The original CGGI scheme was constructed using the *Torus (Ring) Learning With Errors* (TLWE/TRLWE) problem, which is a generalization of LWE/RLWE rescaled over the *real Torus* (a set of real numbers modulo 1) [25]. The original DM scheme was constructed using LWE/RLWE [29]. The CGGI scheme was subsequently instantiated using LWE/RLWE in a unified framework including both DM and CGGI [32].

Scheme Switching Using CGGI

The Chimera framework described in [28] unifies CGGI (TFHE), BFV and CKKS under the TLWE/TRLWE/TRGSW problems and allows using the three schemes in the same computation. Figure 4 provides a schematic of scheme switching in the Chimera framework.

Reference Implementations

The following libraries have open-source CPU implementations of DM and CGGI:

- TFHE (CGGI scheme in simple and advanced modes using binary secret distribution)

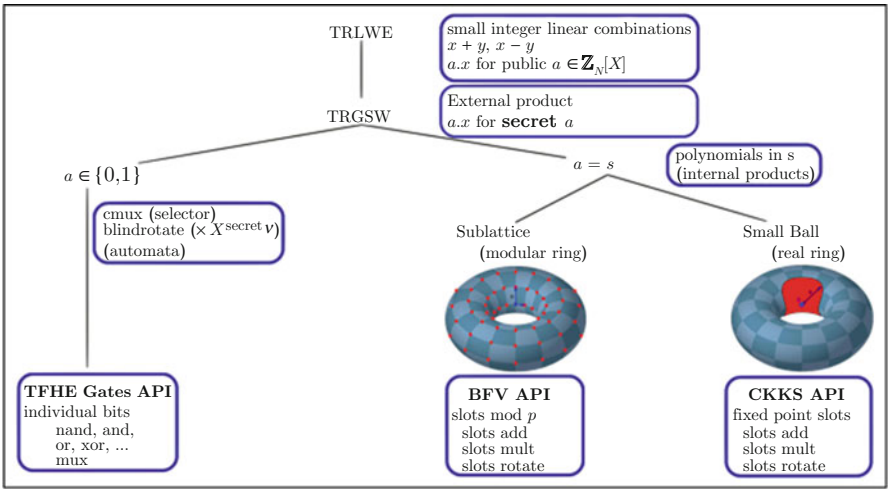


Fig. 4 The Chimera framework

- FHEW (DM scheme in the simple mode using binary distribution)
- PALISADE (DM and CGGI in simple mode using ternary secret distribution)

Please note that the parameters for binary secret distributions are not currently included in the HE Security Standard [9] but are being considered for inclusion in the standard.

The following libraries have GPU implementations of CGGI:

- cuFHE (GPU version of the CGGI scheme in the simple mode)
- nuFHE (GPU version of the CGGI scheme in the simple mode)

A proof of concept implementation of scheme switching between CGGI and CKKS (using the HEAAN library) is publicly accessible [24].

References

1. J. Alperin-Sheriff, and C. Peikert. *Faster Bootstrapping with Polynomial Error*. In CRYPTO 2014. Pages 297–314.
2. J.C. Bajard, J. Eynard, M.A. Hasan, and V. Zucca, V. *A Full RNS Variant of FV Like Somewhat Homomorphic Encryption Schemes*. In SAC 2016. Pages 423–442.
3. Z. Brakerski. *Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP*. In CRYPTO 2012. Pages 868–886.
4. A. Al Badawi, Y. Polyakov, K.M.M. Aung, B. Veeravalli, and K. Rohloff. *Implementation and Performance Evaluation of RNS Variants of the BFV Homomorphic Encryption Scheme*. IEEE Transactions on Emerging Topics in Computing (2019). https://eprint.iacr.org/2018/589.
5. H. Chen and K. Han. *Homomorphic Lower Digits Removal and Improved FHE Bootstrapping*. In EUROCRYPT 2018. Pages 315–337.

6. J. Fan and F. Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive. Report 2012/144, 2012. http://eprint.iacr.org/2012/144.
7. C. Gentry, S. Halevi, and N. P. Smart. *Homomorphic Evaluation of the AES Circuit*. In CRYPTO 2012. Pages 850–867.
8. C. Gentry, S. Halevi, and N. P. Smart. *Better Bootstrapping in Fully Homomorphic Encryption*. In Public Key Cryptography 2012. Pages 1–16.
9. M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, J. Hoffstein, K. Lauter, S. Lokam, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. *Security of Homomorphic Encryption*. http://homomorphicencryption.org/white_papers/security_homomorphic_encryption_white_paper.pdf
10. S. Halevi, Y. Polyakov, and V. Shoup. *An Improved RNS Variant of the BFV Homomorphic Encryption Scheme*. In CT-RSA 2019. Pages 83–105.
11. S. Halevi and V. Shoup. *Algorithms in HElib*. In CRYPTO 2014. Pages 554–571.
12. S. Halevi and V. Shoup. *Bootstrapping for HElib*. In EUROCRYPT 2015. Pages 641–670.
13. T. Lepoint and M. A. Naehrig. *A Comparison of the Homomorphic Encryption Schemes FV and YASHE*. In AFRICACRYPT 2014. Pages 318–335.
14. C. Mouchet, J. Troncoso-Pastoriza and J.-P. Hubaux. *Multiparty Homomorphic Encryption: From Theory to Practice*. Cryptology ePrint Archive, Report 2020/304, 2020. http://github.com/ldsec/lattigo
15. J. H. Cheon, A. Kim, M. Kim, Y. Song, *Homomorphic Encryption for Arithmetic of Approximate Numbers*. In ASIACRYPT 2017. Pages 409–437.
16. J. H. Cheon, K. Han, A. Kim, M. Kim, Y. Song, *Bootstrapping for Approximate Homomorphic Encryption*. In EUROCRYPT 2018. Pages 360–384.
17. J. H. Cheon, K. Han, A. Kim, M. Kim, Y. Song, *A Full RNS Variant of the Approximate Homomorphic Encryption*. In SAC 2018. Pages 347–368.
18. H. Chen, I. Chillotti, Y. Song, *Improved Bootstrapping for Approximate Homomorphic Encryption*. In EUROCRYPT 2019. Pages 34–54.
19. M. Blatt, A. Gusev, Y. Polyakov, K. Rohloff, V. Vaikuntanathan, *Optimized Homomorphic Encryption Solution for Secure Genome-Wide Association Studies*, BMC Medical Genomics, 2020.
20. M. Kim, Y. Song, B. Li, D. Micciancio, *Semi-Parallel Logistic Regression for GWAS on Encrypted Data*, BMC Medical Genomics, 2020.
21. K. Han, D. Ki, *Better Bootstrapping for Approximate Homomorphic Encryption*. In CT-RSA 2020. Pages 364–390.
22. J. Alperin-Sheriff and C. Peikert. Faster Bootstrapping with Polynomial Error. CRYPTO 2014.
23. F. Bourse, M. Minelli, M. Minihold, P. Paillier: Fast Homomorphic Evaluation of Deep Discretized Neural Networks. CRYPTO (3) 2018: 483-512.
24. https://github.com/DPPH/chimera-iDash2018
25. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster Fully Homomorphic Encryption Bootstrapping in Less Than 0.1 Seconds. In Asiacrypt 2016 (Best Paper), pages 3–33.
26. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE. ASIACRYPT (1) 2017: 377–408.
27. I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast Fully Homomorphic Encryption over the Torus. Journal of Cryptology 2019.
28. C. Boura, N. Gama, M. Georgieva and D. Jetchev: CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes. IACR Cryptology ePrint Archive 2018: 758 (2018) (NutMic, submitted to Journal of Mathematical Cryptology 2019).
29. L. Ducas and D. Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. EUROCRYPT 2015.
30. N. Gama, M. Izabachene, P. Q. Nguyen, and X. Xie. Structural Lattice Reduction: Generalized Worst-Case to Average-Case Reductions and Homomorphic Cryptosystems. EUROCRYPT 2016.
31. C. Gentry, A. Sahai, and B. Waters. Homomorphic Encryption From Learning With Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. CRYPTO 2013.

32. D. Micciancio and Y. Polyakov. Bootstrapping in FHEW-like Cryptosystems. Cryptology ePrint Archive. Report 2020/086, 2020. http://eprint.iacr.org/2020/086.
33. https://tfhe.github.io/tfhe/
34. Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. ACM Transactions on Computation Theory (TOCT), 6(3):1-36, 2014.
35. Z. Brakerski and V. Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. SIAM Journal on Computing, 43(2):831-871, 2014.