OpenFHE: Open-Source Fully Homomorphic Encryption Library*

Ahmad Al Badawi[1], Jack Bates[1], Flavio Bergamaschi[2], David Bruce Cousins[1], Saroja Erabelli[1], Nicholas Genise[1], Shai Halevi[3], Hamish Hunt[2], Andrey Kim[4], Yongwoo Lee[4], Zeyu Liu[1], Daniele Micciancio[1,5], Ian Quah[1], Yuriy Polyakov†[1], Saraswathy R.V.[1], Kurt Rohloff[1], Jonathan Saylor[1], Dmitriy Suponitsky[1], Matthew Triplett[1], Vinod Vaikuntanathan[1,6], and Vincent Zucca[7,8]

[1]Duality Technologies
[2]Intel Corporation
[3]Algorand Foundation
[4]Samsung Advanced Institute of Technology
[5]University of California, San Diego
[6]Massachusets Institute of Technology
[7]DALI, Université de Perpignan Via Domitia
[8]LIRMM, University of Montpellier

July 15, 2022

Abstract

Fully Homomorphic Encryption (FHE) is a powerful cryptographic primitive that enables performing computations over encrypted data without having access to the secret key. We introduce OpenFHE, a new open-source FHE software library that incorporates selected design ideas from prior FHE projects, such as PALISADE, HElib, and HEAAN, and includes several new design concepts and ideas. The main new design features can be summarized as follows: (1) we assume from the very beginning that all implemented FHE schemes will support bootstrapping and scheme switching; (2) OpenFHE supports multiple hardware acceleration backends using a standard Hardware Abstraction Layer (HAL); (3) OpenFHE includes both user-friendly modes, where all maintenance operations, such as modulus switching, key switching, and bootstrapping, are automatically invoked by the library, and compiler-friendly modes, where an external compiler makes these decisions. This paper focuses on high-level description of OpenFHE design, and the reader is pointed to external OpenFHE references for a more detailed/technical description of the software library.

*This work is supported in part by DARPA through HR0011-21-9-0003 and HR0011-20-9-0102. The views, opinions, and/or findings expressed are those of the author(s) and should not be interpreted as representing the official views or policies of the Department of Defense or the U.S. Government.

†Corresponding author; email: ypolyakov@openfhe.org

Contents

1 Introduction

Fully Homomorphic Encryption (FHE) is a powerful cryptographic primitive that enables performing computations over encrypted data without having access to the secret key. There are several open-source software libraries implementing FHE schemes, including HElib [39], PALISADE [50], SEAL [52], HEAAN [22], FullRNS-HEAAN [20], Concrete [25], and FHEW [30]. This paper introduces a new open-source FHE library designed by (some of) the authors of PALISADE, HElib, HEAAN, and FHEW libraries. Similar to PALISADE, which was used as the starting point for the design of the new library, the OpenFHE library supports all common FHE schemes. Note that several new scheme variants recently proposed in [43, 44, 45], which are not part of PALISADE, are also implemented in OpenFHE. Moreover, OpenFHE introduces several new design features:

- It is assumed that all supported FHE schemes will eventually support bootstrapping and scheme switching;
- The library can support multiple hardware acceleration backends using a standard Hardware Abstraction Layer (HAL);
- OpenFHE includes both user-friendly modes, where maintenance operations (such as modulus switching, key switching, and bootstrapping) are automatically invoked by the library, and compiler-friendly modes where an external compiler makes these decisions.

OpenFHE design also supports "importing" functionality from its predecessors, such as advanced plaintext encoding based on general cyclotomic rings and static noise estimation from HElib, mixed multiprecision-Residue-Number-System arithmetic from HEAAN, and other new capabilities that are yet to be developed.

This paper presents a high-level description of OpenFHE design, focusing on new functionality and design ideas and concepts. For a more detailed and technical description of the library, the reader is referred to the documentation sources listed in Section 5 of this paper.

2 Cryptographic Capabilities

OpenFHE includes the implementation of all common FHE schemes for integer, real-number, and Boolean arithmetic. All of these schemes are based on the hardness of the ring variant of the Learning With Errors (LWE) problem, which will be denoted by RLWE. Note that currently only power-of-two cyclotomic rings are supported. The library also includes multiparty extensions to support scenarios with multiple secret keys or secret shares. OpenFHE is designed under the assumptions that bootstrapping is available for all implemented schemes and switching ciphertexts between schemes is possible. Note that there are still open research problems related to some scenarios of scheme switching, but we assume that these problems will eventually be solved and the corresponding solutions will be implemented in OpenFHE.

2.1 FHE Schemes

Common FHE schemes are typically grouped into three classes based on the data types they support computations on. The first class supports modular arithmetic over finite fields, which typically correspond to vectors of integers mod t, where t is a prime or a prime power, that is commonly

called as the plaintext modulus. This class is also sometimes used for small-integer arithmetic. The class includes Brakerski-Gentry-Vaikuntantan (BGV) and Brakerski/Fan-Vercauteren (BFV) schemes [13, 14, 31]. The second class primarily works with Boolean circuits and decision diagrams and includes the Ducas-Micciancio (DM) and Chillotti-Gama-Georgieva-Izabachene (CGGI) schemes [24, 30], originally implemented in the FHEW and TFHE libraries, respectively. The third, and most recent, class supports approximate computations over vectors of real and complex numbers, and is represented by the Cheon-Kim-Kim-Song (CKKS) scheme [23]. All these schemes are based on the hardness of the RLWE problem, where noise is added during encryption and key generation to achieve the hardness properties. The noise grows as encrypted computations are performed, and the main functional parameter in all these schemes, the ciphertext modulus Q, needs to be large enough to accommodate the noise growth, or a special bootstrapping procedure may be used to reset the noise and keep the value of Q relatively small. For a high-level introduction to common FHE schemes, the reader is referred to [18].

From the scheme design perspective, the first and third classes share a common design, which was originally introduced for BGV. The design supports a relatively large number of multiplications without bootstrapping, using the technique of modulus switching, and performs homomorphic operations over vectors of integers or real numbers in a Single Instruction, Multiple Data (SIMD) manner, e.g., many multiplications are performed using a single homomorphic multiplication. We will refer to this scheme design pattern as BGV-like.

The DM and CGGI schemes also share a common design, as explained in [47]. In this design, first introduced by the DM scheme in [29], messages are small integers modulo t (for a very small t, e.g. $t = 4$) and are encrypted as simple LWE ciphertexts modulo q, which directly support only homomorphic linear operations (e.g., addition). Non-linear operations are implemented via a form of fast functional bootstrapping (under 100 ms), making internal use of RLWE ciphertexts and a larger modulus Q. This design allows to work with small moduli q, Q, typically 32-bit or 64-bit numbers, (vs. many hundreds of bits for the other schemes). On the other hand, unlike BGV-like schemes, homomorphic multiplications cannot be performed in a SIMD-like manner in DM/CGGI schemes. The CGGI scheme replaces standard LWE ciphertexts modulo q, with LWE ciphertexts "over the torus" (i.e., with real coordinates in $[0, 1)$), and makes use of a different bootstrapping procedure, but is otherwise identical to DM. In OpenFHE, we always use standard LWE ciphertexts modulo q, and adapt the CGGI bootstrapping procedure to work directly on DM ciphertexts. Hence, we will refer to this scheme design pattern as DM-like or the DM scheme.

2.1.1 BGV-like schemes

OpenFHE implements BGV, BFV, and CKKS schemes. Multiple variants of each scheme are supported. For efficiency, only the Residue Number System (RNS) variants are currently supported. RNS is used to efficiently perform operations on large integers by decomposing them into small numbers that fit in machine words, e.g., in 64-bit integers. However, multiprecision or mixed multiprecision-RNS variants of the schemes can be added in the future without changing the current design.

BGV scheme. OpenFHE implements both the original BGV scheme (with unscaled messages) [14] and the Gentry-Halevi-Smart (GHS) variant (with scaled messages) [33]. The main advantage of the GHS variant is that the RNS moduli q_i do not need to satisfy the congruence relation

$q_i \equiv 1 \bmod t$ to perform modulus switching. OpenFHE currently supports only the static noise estimation method [44] to choose the size of RNS moduli.

Four modes for BGV are currently implemented in OpenFHE (these modes are distinguished by the way the modulus switching is performed):

1. FIXEDMANUAL: original BGV variant [14] with RNS optimizations from [33, 44] where modulus switching is manually done by the user.

2. FIXEDAUTO: original BGV variant [14] with RNS optimizations from [33, 44] where modulus switching is automatically done right before a homomorphic multiplication (except for the first multiplication).

3. FLEXIBLEAUTO: GHS variant [33] with RNS optimizations from [44] where modulus switching is automatically done right before a homomorphic multiplication (except for the first multiplication).

4. FLEXIBLEAUTOEXT: GHS variant [33] with RNS optimizations from [44] where modulus switching is automatically done right before a homomorphic multiplication (including the first multiplication).

For a more detailed discussion of the algorithms for all these modes, the reader is referred to [44].

We suggest the following guidelines when choosing the modulus switching method:

- The FLEXIBLEAUTOEXT mode requires the smallest ciphertext modulus Q, but is somewhat slower for most cases than other options (typically less than 1.5x than the fastest mode). However, FLEXIBLEAUTOEXT is the fastest when a smaller ring dimension N can be chosen for its smaller Q to satisfy the same level of security.

- The FIXEDMANUAL and FIXEDAUTO methods are often the fastest (when the ring dimension N needed to achieve the desired level of security is the same for all four modes). Note that FIXEDMANUAL can yield better performance than FIXEDAUTO if it is tailored by an FHE expert or a compiler for a given application.

- The FLEXIBLEAUTO mode can be selected in relatively rare cases where the ring dimension N is smaller than for FIXEDAUTO (note that FLEXIBLEAUTO is often faster than FLEXIBLEAUTOEXT for the same ring dimension N).

- The AUTO modes are much easier to use than the FIXEDMANUAL mode as modulus switching is done automatically by OpenFHE.

- The FLEXIBLEAUTOEXT mode supports larger plaintext moduli than other modes.

Note that the default method is FLEXIBLEAUTOEXT as it easy to use and supports largest plaintext moduli. Other modes can be considered when there is a need to reduce the runtime (typically by no more than 1.5x).

BFV scheme. OpenFHE implements four different RNS variants of the BFV scheme. These variants differ in the way the homomorphic multiplication is performed. There are also some differences in evaluating the decryption operation for some of the variants. These four variants are:

- HPS: the homomorphic multiplication and decryption are implemented using the RNS procedures proposed by Halevi, Polyakov, and Shoup [37]. These RNS procedures use a mix of integer and floating-point operations.
- BEHZ: the homomorphic multiplication and decryption are implemented using the RNS procedures described by Bajard, Eynard, Hasan, and Zucca [6]. These RNS procedures are based on integer arithmetic.
- HPSPOVERQ: the HPS variant where the homomorphic encryption is optimized using the technique described in [44].
- HPSPOVERQLEVELED: the HPSOVERQ variant where modulus switching is applied inside homomorphic encryption to further reduce the computational complexity [44].

Note that all four methods use the modified BFV encryption method proposed in [44], which has smaller noise than the original BFV encryption method [31].

The HPSPOVERQLEVELED method is the fastest when floating-point arithmetic is available. The BEHZ method can be used when floating-point arithmetic is not available (it is slightly slower than HPS, typically by no more than 1.2x). The other two modes, namely, HPS and HPSOVERQ, are available mostly for academic purposes. For a more detailed comparison of the HPS and BEHZ variants, the reader is referred to [1, 7]. The default method for BFV in OpenFHE is HPSPOVERQLEVELED.

OpenFHE also provides two different options for BFV encryption: STANDARD and EXTENDED. For the STANDARD option, the encryption is done using fresh modulus Q. For the EXTENDED setting, a larger modulus is used for encryption by employing auxiliary moduli available for homomorphic multiplication and then modulus switching to Q is executed. The EXTENDED option requires a slightly smaller value of Q (around 5 bits less in the case of public key encryption) but makes encryption more computationally expensive. The STANDARD option is used as the default.

CKKS scheme. OpenFHE implements two RNS variants of the CKKS scheme (they are further split into four modes based on how rescaling is done). The first RNS variant assumes the same scaling factor 2^p for all levels and sets RNS moduli $q_i \approx 2^p$ for all i corresponding to multiplicative levels (all RNS moduli except for the first and possibly last ones, depending on the mode). This method was indepedently proposed in [21] and [8]. The second RNS variant uses a different scaling factor for each level [43].

The following rescaling modes are implemented in OpenFHE (labeled the same way as for the BGV scheme as there are a lot of similarities between CKKS and BGV):

1. FIXEDMANUAL: the RNS variant [8, 21] where modulus switching is manually done by the user.
2. FIXEDAUTO: the RNS variant [8, 21] where rescaling is automatically done right before a homomorphic multiplication (except for the first multiplication).

3. FLEXIBLEAUTO: the RNS variant [43] where rescaling is automatically done right before a homomorphic multiplication (except for the first multiplication).

4. FLEXIBLEAUTOEXT: the RNS variant [43] where rescaling is automatically done right before a homomorphic multiplication (including the first multiplication).

For a more detailed discussion of the algorithms for all these modes, the reader is referred to [43].

We suggest the following guidelines when choosing the rescaling method:

- The FLEXIBLEAUTOEXT mode provides the highest precision for the same parameters. The computational complexity is typically up to 1.5x higher, as compared to the fastest approach (FIXEDMANUAL or FIXEDAUTO) for the same parameters.

- The FLEXIBLEAUTO mode provides a precision that is about 3-4 bits smaller than FLEXIBLEAUTOEXT, but can be slightly faster.

- The FIXEDMANUAL and FIXEDAUTO modes incur additional precision loss of about 3-4 bits as compared FLEXIBLEAUTO, but have smaller computational complexity. Note that FIXEDMANUAL can yield better performance than FIXEDAUTO if it is tailored by an FHE expert or a compiler for a given application.

- The AUTO modes are much easier to use than the FIXEDMANUAL mode as rescaling is done automatically by OpenFHE.

- If the goal is to minimize the ciphertext modulus Q for the same precision, then the FLEXIBALEAUTOEXT mode is the best option. In some scenarios, the decrease in Q may also result in reduced ring dimension for the same security level, yielding better performance for FLEXIBLEAUTOEXT as compared to all other modes.

Note that the default method is FLEXIBLEAUTOEXT as it easy to use and achieves highest precision. Other modes can be considered when there is a need to reduce the runtime (typically by no more than 1.5x).

2.1.2 DM-like schemes

OpenFHE implements the DM scheme [29] with optimizations described in [47]. The library also implements the CGGI bootstrapping method [24] with a ternary CMUX optimization [10]. As explained in [47], the DM and CGGI (functional) bootstrapping procedures are ring versions of bootstrapping methods first proposed by Alperin-Sheriff and Peikert (AP) [3] and Gama, Izabachene, Nguyen, and Xie (GINX) [32]. See [47] for details.

Both DM and CGGI schemes are referred to as DM in OpenFHE, as they are implemented using LWE ciphertexts modulo q as originally proposed in [30]. The DM scheme is labeled as the AP mode of DM, while the GINX mode is used to denote the DM scheme with CGGI bootstrapping. The GINX mode is set as the default as it achieves smaller runtime and bootstrapping key size for ternary secret distribution, as compared to the AP mode [47].

2.2 Multiparty Extensions

OpenFHE supports multiparty FHE extensions for the BGV, BFV, and CKKS schemes: threshold FHE and Proxy ReEncryption (PRE).

The threshold FHE extension is based on additive secret sharing instantiated using additive key homomorphism properties of the FHE schemes. Threshold FHE follows the design introduced in [5]. The key generation and decryption in single-key FHE are replaced with their distributed versions while the computation is still performed the same way as in single-key FHE. Typically the computational overhead of threshold FHE is relatively small, and is primarily determined by a slightly increased norm of the secret key polynomial.

The PRE extension allows delegating an existing ciphertext to another party, which has a different secret key. OpenFHE uses the design proposed in [49]. The main idea is to use FHE key switching to perform proxy reencryption of ciphertexts with the reencryption key, which represents an encryption of the old secret key using the public key for the new secret key. Pre-randomization is applied as part of re-encryption to satisfy the security under honest reencryption attacks (HRA) [26], which is a stronger notion than regular Chosen-Plaintext Attack (CPA) security. The HRA security is often needed in practical applications of PRE.

2.3 Design for BGV-like FHE Schemes

To support scheme switching and simplify the maintenance of BGV-like schemes, we developed a design that includes

1. an abstract base scheme that provides the same API for various instantiations of BGV-like scheme, including full RNS, multiprecision, and hybrid RNS-multiprecision variants;
2. a parent RNS scheme that implements the common functionality for all BGV-like schemes instantiated in RNS;
3. concrete scheme C++ classes that implement scheme-specific functionality.

2.3.1 Common functionality

All BGV-like schemes support the same key switching methods. Hence all RNS variants share the common key switching implementation in OpenFHE. Two switching methods are supported: the Brakerski-Vaikuntanathan [15] digit decomposition method and the Gentry-Halevi-Smart [33] "hybrid" method (adapted to the RNS design in [41]). Both key switching methods and their RNS instantiations are described in detail in Appendix A of the ePrint version for [44].

Most of homomorphic operations, such as additions and rotations, are also performed the same way: they operate on pairs of polynomials, separately working with each polynomial. The only exception is homomorphic multiplication. These common operations are implemented at the level of either the base scheme or parent RNS scheme.

2.3.2 Differences between BGV-like schemes

The differences in BGV-like schemes can be summarized as follows:

- Homomorphic multiplication in BFV includes extending to a larger modulus and scaling down to the original modulus, which is different from BGV and CKKS.

- BGV and CKKS require operations of modulus switching / rescaling that control the noise / approximation error growth after homomorphic multiplications and other operations.
- Decryption in BFV requires special RNS procedures.
- There are some differences in encoding and encryption.

These scheme-specific procedures are implemented in the C++ classes for concrete schemes.

2.4 Design for DM-like FHE Cryptosystems

The DM-like cryptosystems encrypt messages as simple LWE ciphertexts, which support only additions and scalar multiplications, i.e., the scheme additively homomoprhic. Any other functions are evaluated using functional bootstrapping, which makes internal use of a different RLWE-based encryption scheme. More specifically, bootstrapping of DM-like ciphertexts is implemented using RingGSW, an efficient variant of the Gentry-Sahai-Waters (GSW) [34] scheme, adapted to the ring setting in [29]. For a more detailed information on the schemes and bootstrapping procedure, the reader is referred to [29, 47].

The original DM cryptosystem [29] and the original CGGI cryptosystem [24] were designed for evaluating arbitrary Boolean circuits, i.e., mod 2 arithmetic. OpenFHE supports this basic functionality ($t = 2$). The library also supports operations for larger plaintext moduli. Arbitrary function evaluation is supported for $t = 8$, and can be extended to larger moduli by using the homomorphic digit decomposition procedure [45]. The homomorphic comparison/sign evaluation can be performed for plaintext moduli up to $t = 2^{21}$ (a current limitation of the implementation), with the underlying algorithm supporting arbitrary precision. To implement these features, OpenFHE uses the large-precision algorithms proposed in [45]. The development of these capabilities was funded by the DARPA Cooperative Secure Learning (CSL) program.

2.5 Bootstrapping

We expect OpenFHE to support bootstrapping for all core FHE schemes. Bootstrapping is required to enable arbitrary computations while working within a single scheme, and often needed in scenarios where switching between different FHE schemes is beneficial.

The current scheme-specific information about the availability of bootstrapping for core FHE schemes in OpenFHE is as follows:

- Bootstrapping is currently implemented for DM and CGGI schemes using the designs described in [47].
- Approximate bootstrapping is implemented for the CKKS scheme using the RNS design described in [11, 16, 19].
- There is a prototype implementation of thin BGV bootstrapping based on the design from [17, 38], which was built as part of the DARPA Data Protection in Virtual Environments (DPRIVE) program, but it is not available yet in the main repositories of OpenFHE.
- The bootstrapping for BFV will be added together with BGV bootstrapping because both are based on the same design, with minor scheme-specific differences.

2.6 Noise Estimation

The current implementation in OpenFHE does not include noise estimation: The user specifies the multiplicative depth (and for some schemes also the maximum number of additions/key -switching operations), and OpenFHE selects all the necessary parameters, such as the number of bits needed for each multiplicative level. Later, the library performs operations on the ciphertexts, applying scaling/modulus-switching for some schemes, without trying to estimate the noise level in each ciphertext. It is up to the user to ensure that the bounds that were specified ahead of time are respected.

In future versions, we plan to add a noise estimator, similar to HElib [39, 40], that will carry a noise bound with each ciphertext, updating it with the homomorphic computations. In a little more detail, the noise bound will be a heuristic high-probability bound on the l_∞-norm of the canonical embedding of the term "secret-key$\times$ciphertext" that is calculated during decryption. Various quantities that are chosen during key generation and encryption are similarly tagged with a bound on their norm (either a high-probability bound or a probability-one bound). Given all these bounds on the inputs to homomorphic operations, one can derive corresponding bounds on the outputs of those operations.[1]

This method allows the library to evolve the noise bound with every low-level homomorphic operation. The OpenFHE library may also implement APIs that allow users to provide their own noise-evolution bounds for higher-level operations. (For example, certain homomorphic polynomial-evaluation routines may induce smaller noise growth than what can be deduced from the raw multiplications and additions.)

For CKKS, the noise bounds also provide a bound on the approximation error in an encrypted message. We note, however, that this requires also to keep tight bounds on the magnitude of the CKKS plaintext, and obtaining these tight bounds without leaking information about the plaintext takes some care. Moreover, maintaining these tight bounds through the homomorphic operations often relies on help from the user.

Once noise-estimation is implemented, it can be used to make automatic decisions for rescaling, modulus switching, and the HPSPOVERQLEVELED mode of BFV, allowing tighter parameters and simplifying the homomorphic evaluation of deep circuits.

2.7 Scheme Switching

There are many applications that benefit from scheme switching. For instance, in many machine learning applications CKKS is best suited for polynomial evaluation while DM/CGGI works best for comparisons and other discontinuous functions [46]. This requires bridging between schemes, often using procedures based on bootstrapping, e.g., see [12] for an earlier design for switching between CGGI, CKKS, and BFV schemes.

2.7.1 Classes of scheme switching operations

We expect OpenFHE to support a wide range of scheme switching operations, making different encryption schemes inter-operable. Scheme switching encompasses a number of different operations and techniques, that goes well beyond the simple operation of changing the encryption scheme under

[1] For example, the noise growth due to modulus-switching and key-switching operations are specified in Eqs. (28) and (31) of [40].

which a message is encrypted, and it is useful to classify scheme switching operations along several axes:

- **Message space:** Different encryption scheme families use different message spaces: e.g., $\mathbb{Z}_t^n$ for BGV/BFV, $\mathbb{Z}_2$ for DM/CGGI, and integer vectors $\mathbb{Z}^n$ with bounded coordinates for CKKS. Mapping between schemes with different message spaces must necessarily specify also a map between the corresponding messages. For example, a BGV ciphertext encrypting a message $\vec{m} \in \mathbb{Z}_t^n$ may be mapped to a vector of $n \log t$ DM ciphertexts encrypting the coordinates of $\vec{m}$ bit by bit. In its most general form, scheme switching includes a function $f: M_0 \to M_1$ between the message space of the source and target scheme, which is applied to the message. When f is a non-trivial function, the resulting operation may be referred to as *functional* scheme-switching.

- **Key space:** Scheme switching may also change the key under which a message is encrypted. This is clearly required when switching between schemes with different key spaces. Changing the key under which a message is encrypted is a useful operation also when the encryption scheme stays the same. In OpenFHE, key switching is just a special case of scheme switching, where the source and target encryption schemes are the same.

- **Key material:** Some simple forms of scheme switching may be publicly computed without the need of any key material. This is often the case when the encryption key remains unchanged (or is undergoing only "simple changes"). In the most general case, switching between two arbitrarily chosen keys requires some key material, which we call the "evaluation key". (When switching from S_0 to S_1, that key material is often some form of encryption of S_0 under S_1.)

  OpenFHE provides APIs for generating the evaluation key, which can then be used in conjunction with the corresponding key switching algorithm. The plan is for OpenFHE to include a general interface that on input S_0 and S_1 (and the associated encryption schemes) produces a corresponding switching key.

2.7.2 High-level design for scheme switching

Here we describe the design we are planning to implement in future versions of OpenFHE. In its most general form, scheme switching is parameterized by:

- A source and target encryption schemes Enc_0, Enc_1
- A function f between the respective message spaces
- An evaluation key, which may depend on the encryption schemes, function f, and secret keys of the two schemes.

Focusing for simplicity on the case where f is the identity, scheme/key switching is essentially just recryption, performing the decryption function of the first scheme homomorphically under the second. When switching from (Enc^0, S_0) to (Enc^1, S_1), the evaluation key is roughly an encryption of S_0 under S_1, $ek = Enc^1_{S_1}(S_0)$. Given this evaluation key and a ciphertext $C_0 = Enc^0_{S_0}(m)$, one can evaluate the decryption function $S_0 \mapsto Dec_{S_0}(C_0)$ on ek, using the Enc^1 homomorphic operations, to obtain a ciphertext $C_1 = Enc^1_{S_1}(m)$.

The decryption function of (R)LWE-based encryption schemes consists of two steps: the evaluation of a linear function of the secret key, followed by a rounding step to remove the encryption noise. Sometimes when performing scheme switching, it is enough to evaluate only the linear portion of the decryption function without the rounding, in those cases switching can be done much faster than homomorphic rounding. Other times we have to perform full recryption, including the rounding part, in these cases switching also serves as a noise control mechanism. For certain pairs of encryption schemes, full recryption is the only known switching method, even when noise reduction is not required, e.g., switching from BGV to CKKS requires full recryption as noted in Section 2.7.3.

We plan to eventually provide switching implementations between any two schemes in OpenFHE. Internally, the complexity of providing all these switching implementations is managed by using "(R)LWE encryption" as an intermediate representation. In all the schemes that are supported by OpenFHE, ciphertexts can be described as a collection of (R)LWE instances relative to the secret key, where extracting these instances from the ciphertext is simple. Hence, rather than implementing k^2 combinations of switching between all pairs, we will only implement $O(k)$ operations, equipping each FHE scheme with an (R)LWE extraction and an LWE embedding operation.

Extraction is usually a very simple operation that does not require key material, e.g., selecting part of the ciphertext or some simple encoding. Embedding is where the actual scheme switching happens, possibly using an evaluation key, but is simplified by starting from an (R)LWE instance rather than a full ciphertext. The use of LWE or RLWE depends on whether the encryption scheme supports scalar messages (e.g. DM), or vectors (e.g. BGV). Conversion between LWE and RLWE is done using ciphertext packing and unpacking operations.

We note that all the transformations above are internal to OpenFHE, from the user's perspective OpenFHE will just be providing a general scheme switching procedure, where the user specifies a pair of encryption schemes, and OpenFHE generates the evaluation key (if needed) and implements the actual switching.

2.7.3 Scheme-specific remarks

The topic of scheme switching still has open research questions. Here, we summarize the current understanding for various scenarios:

- Switching between BGV and BFV is trivial and does not require any keys or bootstrapping (see Appendix A of [2] for details).
- No scheme switching between DM and CGGI is needed as they differ only in the bootstrapping procedure. The outer scheme implementation is the same in OpenFHE.
- Switching from CKKS to DM/CGGI requires a linear transform to decode CKKS ciphertexts to coefficient encoding and extraction of the numbers in each slot into LWE ciphertexts, applying key and modulus switching where needed (see [46] for more details). This procedure typically does not require bootstrapping.
- Switching from DM/CGGI to CKKS requires applying a linear transformation to perform CKKS encoding, typically followed by a lightweight version of CKKS bootstrapping to support further computations using CKKS (see [46] for details).

- Switching from/to BGV/BFV to/from DM/CGGI follows the same high-level procedures as the switching from/to CKKS to/from DM/CGGI [12]. The main difference is that BGV/BFV bootstrapping is used instead of lightweight CKKS bootstrapping.
- Switching between BGV/BFV and CKKS in either direction appears to require bootstrapping due to the difference in underlying message spaces. This is considered as an open research question.

There is a prototype implementation of switching between CKKS and DM/CGGI and back, which was built as part of the DARPA CSL program, but it is not available yet in the main repositories of OpenFHE. The scheme switching capabilities will be added in future versions of OpenFHE.

3 Hardware Acceleration Support

FHE is a compute-bound privacy-preserving technology, and hence hardware acceleration is crucial for practical use of FHE. OpenFHE is designed to support different hardware acceleration technologies, such as AVX, GPU, FPGA, and ASIC. The computational bottlenecks of FHE are centered around the polynomial arithmetic used for implementing the FHE schemes. This section first provides a brief introduction to polynomial arithmetic, discusses the memory bandwidth limitation of FHE, and then presents the OpenFHE Hardware Abstraction Layer (HAL) to support hardware acceleration of the polynomial arithmetic. We then discuss the Intel HEXL library backend as an example of HAL instantiation in OpenFHE.

The development of HAL and related features has been funded in part by the DARPA DPRIVE program. These features are expected to support the ASIC-based hardware acceleration of cryptographic protocols implemented in OpenFHE, which is one of the goals of DARPA DPRIVE.

3.1 Brief Introduction to Polynomial Arithmetic used in FHE

FHE schemes based on RLWE work with polynomials that contain thousands of large bounded integers. The bound for integers is ciphertext modulus Q. The polynomials curently supported by OpenFHE are defined as $\mathbb{Z}_Q[X]/(X^N+1)$, i.e., they are obtained by applying modulo Q to the coefficients and then computing the polynomial remainder w.r.t. X^N+1. Here, N is a power-of-two ring dimension that is typically in the range from 2^{10} to 2^{17}. Efficient multiplication of such polynomials is achieved by using a Number Theoretic Transform (NTT) operation, which is similar to Fast Fourier Transform but works with modular arithmetic instead of complex numbers. The NTT allows reducing the computational complexity of polynomial multiplication from $O(N^2)$ to $O(N \log N)$. Other common operations for this polynomial arithmetic include component-wise addition and multiplication, which can be interpreted as modular vector addition and multiplication with the computational complexity of $O(N)$.

In the case of BGV-like schemes, such as BGV, BFV, and CKKS, the ciphertext modulus Q is often much larger than the standard machine word size of 64 bits. To support efficient operations with such large numbers, the Residue Number System (RNS) representation is used where Q is expressed as a product of small RNS moduli q_i (each smaller than the machine word size) and large numbers are split into "small" residues modulo q_i. Multiplication of large integers in RNS is performed by computing the products of component-wise residues modulo corresponding RNS

moduli. The RNS arithmetic also includes several special operations that work with multiple RNS moduli at the same time, e.g., basis extension and scaling [37].

Mathematically, the NTT representation for polynomials and RNS representation of large integers are based on the Chinese Remainder Theorem (CRT), and the representation of polynomials in both NTT and RNS formats is often referred to as the double-CRT representation of polynomials [33].

NTT, modular vector addition and multiplication, and RNS basis extension and scaling are the main bottleneck operations for polynomial arithmetic used in FHE, and hence they are the primary target for hardware acceleration.

3.2 Memory Bandwidth as Main Hardware Acceleration Challenge

Recent theoretical and practical hardware acceleration studies suggest that deep FHE computations, which often leverage bootstrapping, are not compute-bound but are rather memory-bound. For instance, the theoretical study [28] shows that secure implementations of CKKS bootstrapping exhibit a low arithmetic intensity (< 1 Op/byte) and require large caches (> 100 MB), making it heavily bound by the main memory bandwidth. An FHE hardware acceleration study [51] describes data movement as the main bottleneck in FHE hardware acceleration of deep homomorphic computations based on BGV and CKKS, and proposes a design that minimizes the data movement. Both studies suggest that key switching is the main memory bandwidth challenge as it works with large evaluation keys and requires a series of alternating NTT and RNS operations with different data access patterns. We consider minimizing data movement as one of main design requirements for OpenFHE hardware acceleration support.

3.3 Hardware Abstraction Layer

Similar to its predecessor PALISADE, OpenFHE has a modular design with multiple layers, which is illustrated in Figure 1 (this is an updated version of Figure 1 in the PALISADE manual [50]). All the algorithms for polynomial arithmetic are implemented in the math and lattice/polynomial layers, and the main goal of HAL is to support multiple instantiations of the bottleneck polynomial and RNS operations in these layers. This is achieved by defining C++ abstract classes for the functionality that can have backend-specific implementation.

In the primitive math layer, we define abstract classes for integers, modular vectors, and NTT transformations. These classes describe the interface for NTT, vector addition, vector multiplications and other operations needed to support polynomial arithmetic for the integers that fit in the machine words.

In the lattice/polynomial layer, we define abstract classes for polynomials, focusing on the double-CRT polynomial respresentation. The RNS subroutines, such as basis extension and scaling, are part of the double-CRT abstract class specification. More complex RNS subroutines that are called by key switching and modulus switching in BGV-like schemes are also defined at the double-CRT class level.

The goal of the design is to provide a backend-aware implementation for integers, modular vectors, NTT transformations, and polynomials while using the same implementation of cryptographic capabilities for all backends.

Multiple backends are supported by using distinct git repositories. A hardware-specific backend implementation is housed in a separate repository labeled as OPENFHE-<BACKEND-NAME>. There

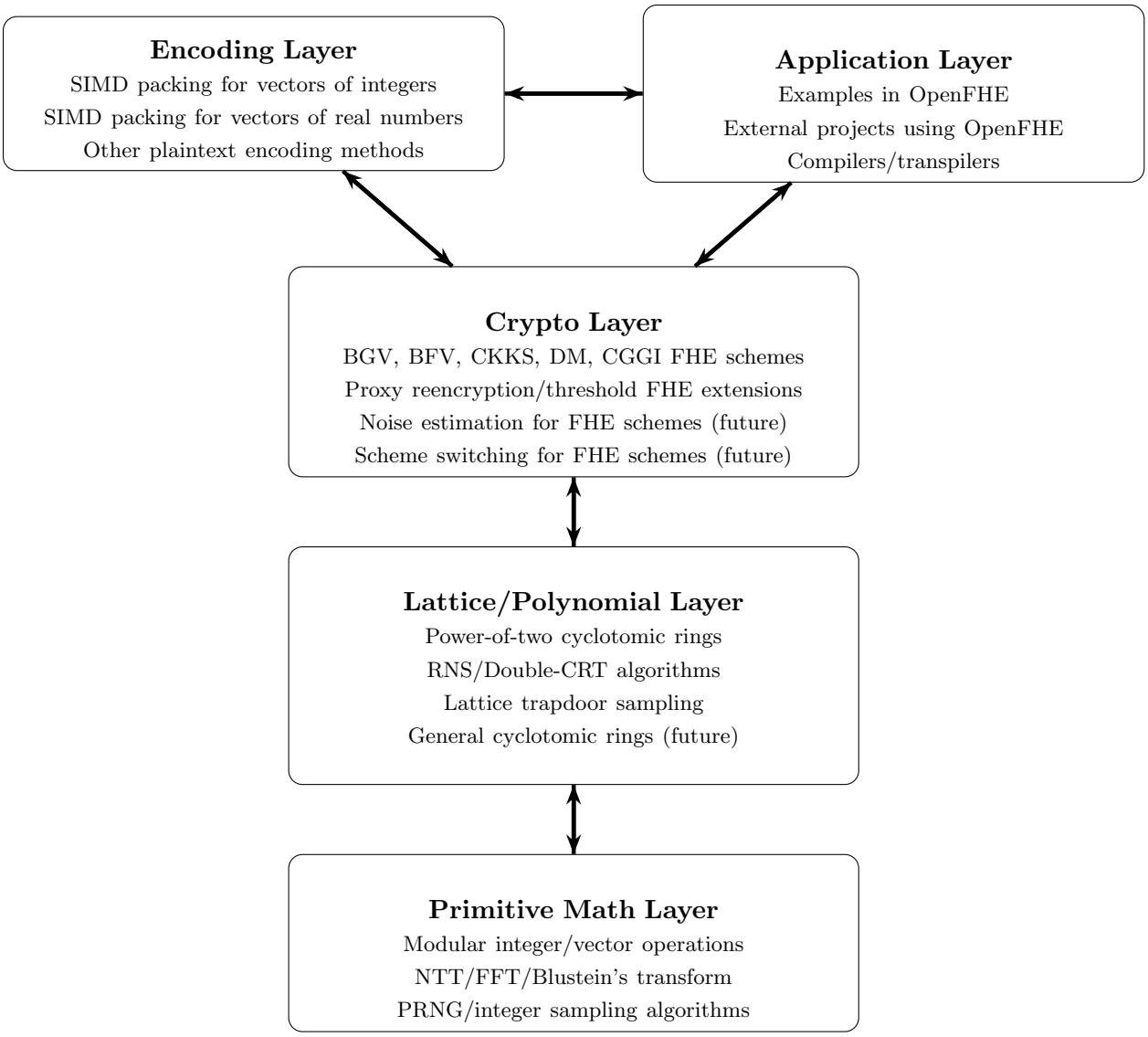

Figure 1: Layers in OpenFHE

is also a special OPENFHE-CONFIGURATOR repository that automatically builds OpenFHE for a selected backend by merging the code from one of the main OpenFHE repositories, i.e., OPENFHE-DEVELOPMENT if using a preview version or OPENFHE if using the stable version of OpenFHE, with the code in OPENFHE-<BACKEND-NAME>.

3.4 Intel® Homomorphic Encryption Acceleration Library (HEXL)

OpenFHE includes a hardward backend implementation for the *Intel Homomorphic Encryption Acceleration Library (HEXL)*, which is available in the OPENFHE-HEXL github repository under the OpenFHE github organization [48].

HEXL is an open-source C++ library which provides efficient implementations of integer arithmetic on finite fields. It is available at https://github.com/intel/hexl under the Apache 2.0 license.

HEXL uses the Intel® Advanced Vector Extensions 512 (Intel® AVX512) instruction set to implement polynomial operations with word-sized primes on 64-bit Intel processors. In particular, HEXL takes advantage of the Intel® Advanced Vector Extensions 512 Integer Fused Multiply Add (Intel® AVX512-IFMA52) [42] instructions introduced in the 3rd Gen Intel® Xeon® Scalable Processors to provide significant speedup on primes below 50 bits.

The primary functionality of HEXL is to provide optimized Intel® AVX512-DQ and Intel AVX512-IFMA52 implementations for the forward and inverse NTT, element-wise vector-vector multiplication, and element-wise vector-scalar multiplication. The Intel AVX512-IFMA52 implementations are valid on prime moduli less than 50 bits, while the Intel AVX512-DQ implementations allow moduli up to 62 bits. HEXL also provides a reference native C++ implementation for each kernel, which has reduced performance but ensures HEXL is compatible with non-AVX512 processors. The choice of implementation is determined at runtime based on the CPU feature availability.

HEXL is designed to intercept HE libraries at the polynomial layer with polynomials in RNS form. Since the majority of each HE operation's runtime lies at the polynomial layer, speedup at the polynomial layer typically propagates to higher-level HE operations.

The algorithms used by, and the implementation of HEXL are discussed in detail by Boemer *et al.* [9].

3.5 Pseudo Random Number Generators (PRNG)

In key generation, encryption, proxy re-encryption, and sometimes in decryption, sampling using a random distribution is needed. The current implementation of OpenFHE supports only software PRNGs based on random seeds. However, in the future we plan to extend HAL to support hardware-specific PRNGs. The current software engineering design already supports multiple PRNGs.

4 Usability Enhancements

One of the goals of OpenFHE is to make FHE schemes more usable for application developers. Some of the FHE schemes are already relatively simple to use. For instance, BFV does not need explicit modulus switching and key switching calls by the user, and has no special constraints on the selection of RNS moduli. DM and CGGI automatically call bootstrapping for every gate (except for negation), and do not require any special logic besides minimizing the number of gates. However, BGV and CKKS are considered much more challenging. For instance, BGV requires careful noise estimation and invocation of modulus switching in appropriate locations. CKKS requires a careful analysis of approximation error and invocation of rescaling in appropriate locations to truncate least significant bits.

4.1 Automation for BGV and CKKS

OpenFHE provides several different modes for BGV and CKKS where the decisions about modulus/key switching and level adjustments are automatically made by the library. These modes, which include AUTO in the name, are described in Section 2.1.1.

In the case of CKKS, the goal of automated logic is to minimize the approximation error, which under the hood requires a complex handling of ciphertexts at different levels, each with a different scaling factor; while not exposing this complicated logic to the user.

4.2 Compiler Support

While the built-in automation in OpenFHE helps a user get relatively simple examples up and running without requiring much FHE expertise, more advanced examples and better performance can be enabled for non-experts using external FHE compilers [4, 27]. These compilers address such challenges as FHE parameter selection, e.g., setting the multiplicative depth, the injection of modulus switching and key switching in a way that optimizes the performance, and dealing with the complicated encoding of input data into FHE-compatible plaintexts. One of the earliest works [4] provided a RAMPARTS compilation framework where programmers in the Julia language can write FHE applications the same way as regular plaintext applications are written. A later project EVA [27] provided an FHE compiler for CKKS with SIMD support.

To support various compilers, OpenFHE includes CKKS and BGV modes without any automation (see the modes with MANUAL in the name in Section 2.1.1) along with more granular operations, e.g., where key switching is broken down into several subroutines and homomorphic multiplication without relinearization is available. These options are exposed either for compilers or FHE experts developing higher-level FHE functions.

4.2.1 Google Transpiler

Since the initial version of OpenFHE, the Google Transpiler [36], which enables compiling FHE applications from C++ code, supports OpenFHE as an FHE backend [35]. The Google Transpiler uses the CGGI implementation in OpenFHE to evaluate arbitrary Boolean circuits, applying various optimizations to generate efficient circuits from the original C++ user code. Additional FHE capabilities of OpenFHE are expected to be supported in future versions of Google Transpiler as a result of collaboration between the OpenFHE and Transpiler teams.

5 Further Information

This paper provides a high-level description of OpenFHE design. More detailed up-to-date information is available from the following sources:

- OpenFHE website: https://openfhe.org
- ReadTheDocs documentation for OpenFHE:
  https://openfhe-development.readthedocs.io/en/latest/
- OpenFHE development repository:
  https://github.com/openfheorg/openfhe-development
- OpenFHE github organization where various OpenFHE-dependent projects are housed:
  https://github.com/openfheorg

References

[1] 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*, 9(2):941–956, 2021. 6

[2] J. Alperin-Sheriff and C. Peikert. Practical bootstrapping in quasilinear time. Cryptology ePrint Archive, Paper 2013/372, 2013. https://eprint.iacr.org/2013/372. 12

[3] J. Alperin-Sheriff and C. Peikert. Faster bootstrapping with polynomial error. In *CRYPTO 2014*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314, 2014. 7

[4] D. W. Archer, J. M. Calderón Trilla, J. Dagit, A. Malozemoff, Y. Polyakov, K. Rohloff, and G. Ryan. Ramparts: A programmer-friendly system for building homomorphic encryption applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, WAHC'19, page 57–68, New York, NY, USA, 2019. Association for Computing Machinery. 17

[5] G. Asharov, A. Jain, A. López-Alt, E. Tromer, V. Vaikuntanathan, and D. Wichs. Multiparty computation with low communication, computation and interaction via threshold fhe. In D. Pointcheval and T. Johansson, editors, *Advances in Cryptology – EUROCRYPT 2012*, pages 483–501, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. 8

[6] J.-C. Bajard, J. Eynard, M. A. Hasan, and V. Zucca. A full RNS variant of FV like somewhat homomorphic encryption schemes. In *International Conference on Selected Areas in Cryptography*, pages 423–442. Springer, 2016. 6

[7] J. C. Bajard, J. Eynard, P. Martins, L. Sousa, and V. Zucca. Note on the noise growth of the RNS variants of the BFV scheme. Cryptology ePrint Archive, Report 2019/1266, 2019. https://eprint.iacr.org/2019/1266. 6

[8] M. Blatt, A. Gusev, Y. Polyakov, K. Rohloff, and V. Vaikuntanathan. Optimized homomorphic encryption solution for secure genome-wide association studies. *BMC Medical Genomics*, 13(7):1–13, 2020. 6

[9] F. Boemer, S. Kim, G. Seifu, F. D. M. de Souza, and V. Gopal. Intel hexl: Accelerating homomorphic encryption with intel avx512-ifma52. Cryptology ePrint Archive, Paper 2021/420, 2021. https://eprint.iacr.org/2021/420. 16

[10] C. Bonte, I. Iliashenko, J. Park, H. V. L. Pereira, and N. P. Smart. Final: Faster fhe instantiated with ntru and lwe. Cryptology ePrint Archive, Report 2022/074, 2022. https://ia.cr/2022/074. 7

[11] J.-P. Bossuat, C. Mouchet, J. Troncoso-Pastoriza, and J.-P. Hubaux. Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys. In A. Canteaut and F.-X. Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 587–617, Cham, 2021. Springer International Publishing. 9

[12] C. Boura, N. Gama, M. Georgieva, and D. Jetchev. Chimera: Combining ring-lwe-based fully homomorphic encryption schemes. *Journal of Mathematical Cryptology*, 14(1):316–338, 2020. 10, 13

[13] Z. Brakerski. Fully homomorphic encryption without modulus switching from classical GapSVP. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012. 4

[14] Z. Brakerski, C. Gentry, and V. Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):1–36, 2014. 4, 5

[15] Z. Brakerski and V. Vaikuntanathan. Fully homomorphic encryption from ring-LWE and security for key dependent messages. In *Annual cryptology conference*, pages 505–524. Springer, 2011. 8

[16] H. Chen, I. Chillotti, and Y. Song. Improved bootstrapping for approximate homomorphic encryption. In Y. Ishai and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 34–54, Cham, 2019. Springer International Publishing. 9

[17] H. Chen and K. Han. Homomorphic lower digits removal and improved fhe bootstrapping. In J. B. Nielsen and V. Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018*, pages 315–337, Cham, 2018. Springer International Publishing. 9

[18] J. H. Cheon, A. Costache, R. C. Moreno, W. Dai, N. Gama, M. Georgieva, S. Halevi, M. Kim, S. Kim, K. Laine, Y. Polyakov, and Y. Song. Introduction to homomorphic encryption and schemes. In K. Lauter, W. Dai, and K. Laine, editors, *Protecting Privacy through Homomorphic Encryption*, pages 3–28, Cham, 2021. Springer International Publishing. 4

[19] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018. 9

[20] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. FullRNS-HEAAN, 2018. https://github.com/KyoohyungHan/FullRNS-HEAAN. 3

[21] J. H. Cheon, K. Han, A. Kim, M. Kim, and Y. Song. A full rns variant of approximate homomorphic encryption. In C. Cid and M. J. Jacobson Jr., editors, *Selected Areas in Cryptography – SAC 2018*, pages 347–368, Cham, 2019. Springer International Publishing. 6

[22] J. H. Cheon, A. Kim, M. Kim, and Y. Song. HEAAN, 2016. https://github.com/snucrypto/HEAAN. 3

[23] J. H. Cheon, A. Kim, M. Kim, and Y. Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017. 4

[24] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In J. H. Cheon and T. Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 3–33, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg. 4, 7, 9

[25] I. Chillotti, M. Joye, D. Ligier, J.-B. Orfila, and S. Tap. Concrete: Concrete operates on ciphertexts rapidly by extending tfhe. In *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, volume 15, 2020. 3

[26] A. Cohen. What about bob? the inadequacy of cpa security for proxy reencryption. In D. Lin and K. Sako, editors, *Public-Key Cryptography – PKC 2019*, pages 287–316, Cham, 2019. Springer International Publishing. 8

[27] R. Dathathri, B. Kostova, O. Saarikivi, W. Dai, K. Laine, and M. Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 546–561, New York, NY, USA, 2020. Association for Computing Machinery. 17

[28] L. de Castro, R. Agrawal, R. Yazicigil, A. Chandrakasan, V. Vaikuntanathan, C. Juvekar, and A. Joshi. Does fully homomorphic encryption need compute acceleration? Cryptology ePrint Archive, Paper 2021/1636, 2021. https://eprint.iacr.org/2021/1636. 14

[29] L. Ducas and D. Micciancio. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology – EUROCRYPT 2015*, pages 617–640, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. 4, 7, 9

[30] L. Ducas and D. Micciancio. FHEW, 2017. https://github.com/lducas/FHEW. 3, 4, 7

[31] J. Fan and F. Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptol. ePrint Arch.*, 2012:144, 2012. 4, 6

[32] N. Gama, M. Izabachène, P. Q. Nguyen, and X. Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In *EUROCRYPT 2016*, volume 9666 of *Lecture Notes in Computer Science*, pages 528–558, 2016. 7

[33] C. Gentry, S. Halevi, and N. P. Smart. Homomorphic evaluation of the AES circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012. 4, 5, 8, 14

[34] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In R. Canetti and J. A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, pages 75–92, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. 9

[35] FHE C++ transpiler, 2022. https://github.com/google/fully-homomorphic-encryption. 17

[36] S. Gorantala, R. Springer, S. Purser-Haskell, W. Lam, R. Wilson, A. Ali, E. P. Astor, I. Zukerman, S. Ruth, C. Dibak, P. Schoppmann, S. Kulankhina, A. Forget, D. Marn, C. Tew, R. Misoczki, B. Guillen, X. Ye, D. Kraft, D. Desfontaines, A. Krishnamurthy, M. Guevara, I. M. Perera, Y. Sushko, and B. Gipson. A general purpose transpiler for fully homomorphic encryption. Cryptology ePrint Archive, Paper 2021/811, 2021. https://eprint.iacr.org/2021/811. 17

[37] S. Halevi, Y. Polyakov, and V. Shoup. An improved RNS variant of the BFV homomorphic encryption scheme. In *Cryptographers' Track at the RSA Conference*, pages 83–105. Springer, 2019. 6, 14

[38] S. Halevi and V. Shoup. Bootstrapping for helib. Cryptology ePrint Archive, Paper 2014/873, 2014. https://eprint.iacr.org/2014/873. 9

[39] S. Halevi and V. Shoup. HElib, 2014. https://github.com/homenc/HElib. 3, 10

[40] S. Halevi and V. Shoup. Design and implementation of HElib: a homomorphic encryption library. Cryptology ePrint Archive, Report 2020/1481, 2020. https://eprint.iacr.org/2020/1481. 10

[41] K. Han and D. Ki. Better bootstrapping for approximate homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 364–390. Springer, 2020. 8

[42] Intel Corporation. Intel intrinsics guide, 2021. https://software.intel.com/sites/landingpage/IntrinsicsGuide/#avx512techs=AVX512IFMA52. 16

[43] A. Kim, A. Papadimitriou, and Y. Polyakov. Approximate homomorphic encryption with reduced approximation error. In S. D. Galbraith, editor, *Topics in Cryptology – CT-RSA 2022*, pages 120–144, Cham, 2022. Springer International Publishing. 3, 6, 7

[44] A. Kim, Y. Polyakov, and V. Zucca. Revisiting homomorphic encryption schemes for finite fields. In M. Tibouchi and H. Wang, editors, *Advances in Cryptology – ASIACRYPT 2021*, pages 608–639, Cham, 2021. Springer International Publishing. 3, 5, 6, 8

[45] Z. Liu, D. Micciancio, and Y. Polyakov. Large-precision homomorphic sign evaluation using fhew/tfhe bootstrapping. Cryptology ePrint Archive, Paper 2021/1337, 2021. https://eprint.iacr.org/2021/1337. 3, 9

[46] W.-j. Lu, Z. Huang, C. Hong, Y. Ma, and H. Qu. Pegasus: Bridging polynomial and non-polynomial evaluations in homomorphic encryption. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1057–1073, 2021. 10, 12

[47] D. Micciancio and Y. Polyakov. Bootstrapping in fhew-like cryptosystems. In *WAHC 2021–9th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, page 17–28, New York, NY, USA, 2021. Association for Computing Machinery. 4, 7, 9

[48] OpenFHE HEXL backend, 2022. https://github.com/openfheorg/openfhe-hexl. 15

[49] Y. Polyakov, K. Rohloff, G. Sahu, and V. Vaikuntanathan. Fast proxy re-encryption for publish/subscribe systems. *ACM Trans. Priv. Secur.*, 20(4), sep 2017. 8

[50] Y. Polyakov, R. Rohloff, G. W. Ryan, and D. Cousins. PALISADE Lattice Cryptography Library (release 1.11.5). https://palisade-crypto.org/, 2021. https://gitlab.com/palisade/palisade-release/-/blob/master/doc/palisade_manual.pdf. 3, 14

[51] N. Samardzic, A. Feldmann, A. Krastev, S. Devadas, R. Dreslinski, C. Peikert, and D. Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, page 238–252, New York, NY, USA, 2021. Association for Computing Machinery. 14

[52] Microsoft SEAL, 2020. https://github.com/Microsoft/SEAL. 3