Bootstrapping in FHEW-like Cryptosystems*

Daniele Micciancio and Yuriy Polyakov

Duality Technologies

March 3, 2020

Abstract

FHEW and TFHE are fully homomorphic encryption (FHE) cryptosystems that can evaluate arbitrary Boolean circuits by bootstrapping after each gate evaluation. The FHEW cryptosystem was originally designed based on standard (Ring) LWE assumptions, and its initial implementation was able to run bootstrapping in less than 1 second. The TFHE cryptosystem used somewhat stronger assumptions, such as LWE over torus and binary secret distribution, and applied several other optimizations to reduce the bootstrapping runtime to less than 0.1 second. Up to now, the gap between the underlying security assumptions prevented a fair comparison of the cryptosystems for same security settings.

We present a unified framework that includes the original and extended variants of both FHEW and TFHE cryptosystems, and implement it in PALISADE using modular arithmetic. Our analysis shows that the main distinction between the cryptosystems is the bootstrapping procedure used: Alperin-Sherif–Peikert (AP) for FHEW vs. Gama–Izabachene–Nguyen–Xie (GINX) for TFHE. All other algorithmic optimizations in TFHE equally apply to both cryptosystems. We extend the GINX bootstrapping to ternary uniform and Gaussian secret distributions, which are included in the HE community security standard. Our comparison of the AP and GINX bootstrapping methods for different secret distributions suggests that the TFHE/GINX cryptosystem provides better performance for binary and ternary secrets while FHEW/AP is faster for Gaussian secrets. We make a recommendation to consider the variants of FHEW and TFHE cryptosystems based on ternary and Gaussian secrets for standardization by the HE community.

1 Introduction

Bootstrapping, i.e, the homomorphic evalution of a decryption circuit on the encryption of a secret key, is a central component of all fully homomorphic encryption (FHE) schemes since Gentry's pioneering work [13], and the main efficiency bottleneck in their implementation. The most common approach to mitigate the high cost of bootstrapping (e.g., see [14, 22, 16]) is to simultaneously refresh several messages during a single bootstrapping computation, and to carefully apply the bootstrapping algorithm only when strictly required. This allows to reduce the amortized (per message and per homomorphic operation) cost of bootstrapping, but results in a rather involved programming model where basic operations cannot be arbitrarily composed. An alternative approach was put

*This work was funded solely by Duality Technologies.

forward by the FHEW encryption cryptosystem [11] which demonstrated (building on techniques from [15, 4] and some novel technical ingredients) that an elementary bootstrapped computation (on a single encrypted bit) could be carried out in practice in a fraction of a second, for typical values of security. The running time was further improved by the TFHE cryptosystem [8], which introduced a number of optimizations over [11] in conjunction with an alternative bootstrappting strategy [12]. Two distinguishing features of the TFHE cryptosystem, that set it apart from most other homomorphic encryption proposals, are the use of the Ring LWE problem with binary secrets, and the use of arithmetic operations over the continuous interval $[0, 1)$ (the so-called Torus, from which TFHE derives its name) instead of the more common modular integer arithmetic. These are legitimate choices in the exploratory context of proposing a new FHE cryptosystem, but also make it harder both to compare TFHE to other cryptosystems (and FHEW in particular) and to consider the cryptosystem within the ongoing *Homomorphic Encryption Standardization* process [2].

Secret distribution. Theoretical work on the LWE and Ring LWE problems [17, 21] suggests that secret vectors should have random Gaussian entries with standard deviation $\sigma \approx \sqrt{n}$ or larger (where n is the underlying lattice dimension or security parameter). Small secrets (with binary entries) have received some attention, but theoretical results supporting their hardness [19] only apply to the inefficient (non-ring) LWE setting. In fact, this is the reason why the FHEW cryptosystem [11] used a combination of standard LWE encryption with binary secrets together with Ring LWE with arbitrary (or large Gaussian) secrets. As a pragmatic choice to support the use of more efficient schemes, the *Homomorphic Encryption* standardization document [2] considers the use of Gaussians over a smaller interval $\{\pm 8\}$, or even "ternary" secrets with entries in $\{-1, 0, 1\}$. However, it does not currently support the extreme choice of binary secrets with $\{0, 1\}$ entries. Even if binary secrets were to be included in [2], any concrete efficiency comparison between schemes should carefully take into account how the use of different secret distributions affects the concrete security level, possibly calling for different values of the key size.

Our work. In order to better understand the relative merits of the bootstrapping procedures employed by FHEW and TFHE, and facilitate the inclusion of these cryptosystems in the standard [2], we implemented several variants and generalizations of these cryptosystems within a uniform framework (using the PALISADE lattice cryptography library), supporting the use of different secret distributions as well as a range of time/memory trade-offs. More specifically, we started from FHEW as a base cryptosystem, for its use of standard modular integer arithmetic and arbitrary secrets. We reimplemented it within the PALISADE library, including the algorithmic optimizations from TFHE that are applicable to both cryptosystems, but without sacrificing the use of modular integers or the support for larger secret key distributions.

As we will explain, there are two main approaches to arithmetic bootstrapping, first suggested in the (non-ring, inefficient) LWE setting by Alperin-Sheriff and Peikert [4] and Gama, Izabachene, Nguyen and Xie [12]. The FHEW cryptosystem is essentially an instantiation of [4] for the Ring LWE setting, while TFHE proposes a similar Ring LWE adaptation of [12]. So, we will refer to these two bootstrapping procedures as AP/FHEW and GINX/TFHE. This is the main algorithmic difference between FHEW and TFHE, and the reason why TFHE requires binary secrets: while the AP bootstrapping algorithm can be equally applied to secrets of any size, the GINX one is directly applicable only to binary secrets, and extending it to arbitrary secrets carries a substantial cost.

To further facilitate the comparison between the two bootstrapping algorithms, we propose (and implement) a simple method to adapt GINX/TFHE bootstrapping to non-binary secrets. For example, using the linearity of the bootstrapping procedure, one can express ternary secrets as the difference $\mathbf{s}-\mathbf{s}' \in \{-1,0,1\}^n$ between two bits $\mathbf{s},\mathbf{s}' \in \{0,1\}^n$, carry out two TFHE computations on binary secrets $\mathbf{s},\mathbf{s}'$, and then take the difference between the two outputs. This provides a simple method to adapt GINX/TFHE techniques to parameter sets covered by the standard [2], and compare its performace with AP/FHEW bootstrapping and other schemes in a uniform security setting. For the sake of comparison, we also considered instantiations of FHEW with binary secrets, as well as variants of FHEW that offer trade-offs between running time and bootstrapping key size.

Comparison highlights. Our results suggest that different performance between FHEW and TFHE can be almost entirely explained by the choice of a different bootstrapping algorithm. Moreover, how the cryptosystems compare with each other is highly sensitive to the secret key distribution. In summary, when the secret is binary (in $\{0,1\}$), TFHE is faster than FHEW roughly by a factor 2 (once FHEW has been improved with algorithmic optimizations that apply equally to both cryptosystems). However, already for ternary secrets $\{-1,0,1\}$, the two cryptosystems have essentially the same running time, with FHEW being slightly faster, but at the cost of a much larger bootstrapping key. But as one moves to larger secrets (e.g., Gaussian with standard deviation $\sigma = 3.2$ or $\sigma \approx \sqrt{n}$ as supported by [2] and theoretical work), FHEW outperforms TFHE in terms of running time. So, the performance advantages of TFHE seem to be specific to the choice of binary secrets. In terms of memory usage, the TFHE bootstrapping key is always smaller than FHEW, but this can be largely mitigated by FHEW's time/memory trade-off: when instantiated to optimize space, the FHEW bootstrapping key is only about twice as large as TFHE, while still providing much better running times than TFHE in the Gaussian secret settings.

A more detailed description of previous work related to FHEW and TFHE and our contribution is provided in the next subsections.

1.1 Historical Background

The first FHE scheme based on the hardness of approximating lattice problems within polynomial factors was proposed by Brakerski and Vaikuntanathan in [7]. More specifically, that paper showed how to use the homomorphic encryption scheme of Gentry, Sahai and Waters [15] to evaluate polynomial-size branching programs, while keeping the lattice approximation factor polynomial. Since LWE ciphertexts can be decrypted by log-depth circuits, and log-depth circuits can be expressed by polynomial-size branching programs, this allowed, for the first time, to base Gentry's bootstrapping procedure on the polynomial hardness of LWE.

However, decrypting by reduction to log-depth circuits and general branching programs is rather impractical. More efficient methods to bootstrap using the GSW cryptosystem were later proposed by Alperin-Sheriff and Peikert [4] and Gama, Izabachene, Nguyen and Xie [12]. Similar to [7], these works can homomorphically compute the LWE decryption function (and therefore, implement bootstrapping) based on the hardness of LWE with polynomial modulus q. However, the use of the special structure of the LWE decryption function results in a much simpler procedure than generic reductions to branching programs.

Still, the practical efficiency of [4, 12] was rather limited because those works employed general lattices. A big step in improving the speed of bootstrapping was taken by the FHEW cryptosystem [11], which demonstrated for the first time that a fully bootstrapped homomorphic evaluation (of an elementary operation) could be performed in a fraction of a second. The FHEW cryptosystem introduced two important technical innovations:

- The observation that the evaluation of an arbitrary function can be split into the computation of a linear function (which can be directly implemented by the LWE encryption scheme), followed by a table look-up, which can be easily performed during bootstrapping essentially at no additional cost.

- A ring version of the GSW cryptosystem (based on the Ring LWE problem), which can be used to provide an efficient implementation of a cryptographic accumulator, as needed by the bootstrapping procedure of [4].

The first contribution allows to realize fully homomorphic encryption by bootstrapping a very simple LWE-based encryption scheme. (Previous schemes required to bootstrap a scheme capable of evaluating at least one multiplication operation. This is not the case in FHEW, where the basic scheme is only required to support addition.) Then, this simple scheme is bootstrapped using the method of [4], but implemented using the efficient accumulators based on the Ring GSW cryptosystem.

Finally, [8, 9] proposed TFHE, a homomorphic encryption cryptosystem "over the torus", which replaces integer arithmetics modulo q with real arithmetic over the unit interval $[0, 1]$, and introduced several optimizations to the FHEW cryptosystem. The most important difference between TFHE and FHEW is that TFHE uses the FHEW accumulators to implement a ring variant of the bootstrapping procedure of [12], rather than [4].

The performance improvement of TFHE over FHEW is substantial. However, TFHE also makes a number of technical choices (e.g., the use of real arithmetic over $[0, 1]$, and the use of Ring LWE with binary secrets) that are somehow non-standard, making it harder for the cryptosystem to be considered by the current Homomorphic Encryption Standardization process [2].

The purpose of this paper is twofold:

- Producing a "standard-compliant" version of the FHEW/TFHE cryptosystem, within the PALISADE lattice cryptography library, to enable the comparison of FHEW with the other main FHE schemes (BGV and BFV) currently considered for standardization.

- Implement (Ring LWE versions of) both bootstrapping procedures [4, 12] within the same (integer-based) FHEW cryptosystem, in order to better understand the relative merits of the two bootstrapping methods, and the differences between FHEW and TFHE.

We remark that the TFHE library described in [8, 9] provides several other procedures beside single gate computations and bootstrapping. Since these auxiliary functions can arguably be implemented both on top of FHEW and TFHE, we see these extensions as orthogonal to the standardization and comparison of FHEW/TFHE, and focus on the core functionality of the cryptosystems.

Our contributions. Our contributions can be summarized as follows:

- We extend the GINX bootstrapping [12] to ternary uniform and Gaussian secret key distributions, which are included in the HE Security Standard [2]. The original GINX bootstrapping supports only binary secret key distribution, which is not being considered for standardization.

- We present a variant of the TFHE cryptosystem that is compliant with the HE Security Standard [2]. Our variant does not require any "non-standard" assumptions, such as LWE over torus or binary secret key distribution.

- We present a theoretical comparison of the AP and GINX bootstrapping procedures for common secret key distributions. Our analysis suggests that the GINX bootstrapping is more efficient for binary and ternary secret key distributions while the AP bootstrapping provides better computational complexity for Gaussian secret key distributions.

- We provide an open-source implementation of the extended FHEW and TFHE cryptosystems in PALISADE.

Other developments The FHEW cryptosystem studied the problem of fast bootstrapping by considering the evaluation of the simplest possible gate: a Boolean NAND operation, or other binary gate. This has been extended to larger gates in [5, 6]. In a different direction, [20] showed how to simultaneously bootstrap n FHEW gates at a cost comparable to a single FHEW bootstrapping, thereby the reducing the amortized cost of bootstrapping by a factor (close to) n. Both improvements are currently mostly of theoretical interest, as they introduce a substantial overhead that makes them unattractive in practice. Finding more practical implementations of the ideas in [5, 6, 20] is a theoretically interesting, and practically important open problem.

2 Ring LWE Encryption

Let $R = \mathbb{Z}[X]/(X^n+1)$ be the $2n$th cyclotomic ring, for $n = 2^k$, and $R_q = R/qR \equiv \mathbb{Z}_q[X]/(X^n+1)$. We identify ring elements with the corresponding coefficient vectors in $\mathbb{Z}^n$ and $\mathbb{Z}_q^n$.

We recall the construction of homomorphic encryption schemes from the Ring LWE problem following the modular approach of [18]. The basic Ring LWE symmetric encryption scheme encrypts (the encoding of) a message $\tilde{m} \in R_q$ under key $s \in R$ as

$$\mathsf{RLWE}_s(\tilde{m}) = (a, as + e + \tilde{m}),$$

where $a \leftarrow R_q$ is chosen uniformly at random, and $e \leftarrow \chi_\sigma^n$ is chosen from a discrete Gaussian distribution of parameter σ. We write $\mathsf{RLWE}_s(\tilde{m}; a, e)$ or $\mathsf{RLWE}_s(\tilde{m}; e)$ when we want to make the randomness explicit or emphasize the error term e. The secret key s may be chosen from the uniform distribution over R_q, or from some other distribution with support over "short" elements, e.g., $s \leftarrow \chi_\sigma^n$ or $s \leftarrow \{0, 1, -1\}^n$. A ciphertext $(a, b) \in R_q^2$ is decrypted by computing

$$\mathsf{RLWE}_s^{-1}(a, b) = b - as = \tilde{m} + e$$

and then evaluating an appropriate decoding function to correct the error e and recover the message encoded by $\tilde{m}$. The simplest type of encoding is to use R_t as a message space (for a message modulus t much smaller than q), and encode $m \in R_t$ as $\tilde{m} = (q/t)m$, i.e., by scaling m by a factor (q/t). Assuming the error distribution χ is concentrated on vectors with entries bounded by $q/(2t)$ in absolute value, the decoding function

$$f(x) = \lfloor (t/q)x \rceil \pmod t$$

recovers the message by rounding each coordinate to the closest multiple of q/t.

The error of a Ring LWE ciphertext (a, b) encrypting message (encoding) $\tilde{m}$ under key s is defined as

$$\mathsf{Err}_s((a,b);\tilde{m}) = b - as - \tilde{m}.$$

If (a, b) is computed using the RLWE encryption algorithm, then $\mathsf{Err}_s((a,b);\tilde{m}) = e$ is precisely the error term $e \leftarrow \chi_\sigma^n$. We will use the function Err to keep track of errors also when computing homomorphically on ciphertexts.

RLWE encryption is linearly homomorphic, with operations defined as

$$\begin{aligned}(a_0, b_0) + (a_1, b_1) &= (a_0 + a_1, b_0 + b_1)\\ d \cdot (a_0, b_0) &= (d \cdot a_0, d \cdot b_0)\end{aligned}$$

where $\mathbf{c}_0 = (a_0, b_0) = \mathsf{RLWE}_s(\tilde{m}_0)$, $\mathbf{c}_1 = (a_1, b_1) = \mathsf{RLWE}_s(\tilde{m}_1)$ are ciphertexts and $d \in R_q$. The error growth during homomorphic computations is given by

$$\begin{aligned}\mathsf{Err}_s(\mathbf{c}_0 + \mathbf{c}_1; \tilde{m}_0 + \tilde{m}_1) &= \mathsf{Err}_s(\mathbf{c}_0;\tilde{m}_0) + \mathsf{Err}_s(\mathbf{c}_1;\tilde{m}_1)\\ \mathsf{Err}_s(d \cdot \mathbf{c}_0; d \cdot \tilde{m}_0) &= d \cdot \mathsf{Err}_s(\mathbf{c}_0;\tilde{m}_0).\end{aligned}$$

Notice that the multiplication operation

$$(\cdot)\colon R \times \mathsf{RLWE} \to \mathsf{RLWE}$$

defined above can only be used with "small" multipliers d, otherwise the error could grow beyond bounds. In order to support multiplication by arbitrary ring elements, one defines a new encryption scheme

$$\mathsf{RLWE}'_s(m) = (\mathsf{RLWE}_s(m), \mathsf{RLWE}_s(Bm), \mathsf{RLWE}(B^2m), \ldots, \mathsf{RLWE}_s(B^{k-1}m))$$

using the same keys $s \in R_q$, and where ciphertexts consist of $k = \log_B q$ basic encryptions produced by the original RLWE scheme. The base B can be set differently to achieve various time/space trade-offs. For simplicity, we assume q is a power of B, but the scheme can be easily adapted to other values. Mixed base variants are also possible, where B^i is replaced by a product $B_1 \cdots B_i$. Incidentally, we note that these ciphertexts allow to recover the message m exactly, even without encoding/scaling, by first decrypting $\mathsf{RLWE}(B^{k-1}m)$ to recover $(m \bmod B)$. Then subtracting $B^{k-2} \cdot (m \bmod B)$ from $\mathsf{RLWE}_s(B^{k-2}m)$ to recover $(m \bmod B^2)$, and so on. More importantly, RLWE' ciphertexts support multiplication by any constant $d \in R_q$ by first expressing $d = \sum_i B^i d_i$ with B-bounded component polynomials d_i, and then computing

$$d \odot (\mathbf{c}_0, \ldots, \mathbf{c}_{k-1}) = \sum_i d_i \cdot \mathbf{c}_i.$$

This provides a multiplication operation

$$(\odot)\colon R \times \mathsf{RLWE}' \to \mathsf{RLWE}$$

with much smaller error growth than the basic $R \times \mathsf{RLWE}$ product. Specifically, the error is only logarithmic in q, rather than linear. It is also possible to define a multiplication operation

$$(\odot')\colon R \times \mathsf{RLWE}' \to \mathsf{RLWE}' \qquad d \odot' \mathbf{C} = (d \odot \mathbf{C}, (B \cdot d) \odot \mathbf{C}, \ldots, (B^{k-1} \cdot d) \odot \mathbf{C}).$$

with result in RLWE'.

The RLWE and RLWE' schemes only support multiplication by constant values. In order to support multiplication by ciphertexts, we define one more cryptosystem, which is equivalent to a ring variant of the encryption scheme proposed in [15]. The scheme is built from RLWE' using the same keys, and, conceptually, it can be defined as

$$\mathsf{RGSW}_s(m) = (\mathsf{RLWE}'_s(-s \cdot m), \mathsf{RLWE}'_s(m)). \tag{1}$$

We said "conceptually" because, as shown below, there is a simpler and more direct way to compute these ciphertexts. The security of the scheme is based on the fact that RLWE (and therefore also RLWE') is circular secure, because an encryption of s can be trivially computed as the ciphertext $(a, b) = (-1, 0)$, which, by construction, decrypts to $b - as = 0 - (-1) \cdot s = s$. This same property also provides a method to compute the first component of (1) without explicitly including the secret key s as part of the message:

$$\mathsf{RLWE}'_s(-s \cdot m) = \mathsf{RLWE}'_s(0) - m \cdot (-1, 0) = (a + m, as + e).$$

Readers familiar with the original GSW cryptosystem [15] (or, more precisely, the Ring LWE version [11] of the simplified variant proposed in [4]) will immediately notice how $\mathsf{RGSW}_s(m)$ ciphertexts can be equivalently written as

$$\mathsf{RGSW}_s(m) = (\mathsf{RLWE}_s(0), \ldots, \mathsf{RLWE}_s(0)) + m\mathbf{G} \tag{2}$$

where $\mathbf{G} = \mathbf{I} \otimes (1, B, B^2, \ldots, B^{k-1})$ is the powers-of-B "gadget matrix". For simplicity, PALISADE implements the RGSW cryptosystem directly using equation (2), but multiplication between RGSW ciphertexts is best analyzed using the modular definition (1) from [18].

Multiplication between RLWE ciphertexts is supported by computing the RLWE decryption function homomorphically. Specifically, given $(a, b) = \mathsf{RLWE}_s(m_0; e_0)$ and $(\mathbf{c}, \mathbf{c}') = \mathsf{RGSW}_s(m_1)$ one computes

$$\begin{aligned}
(a, b) \diamond (\mathbf{c}, \mathbf{c}') &= \big\langle (a, b), (\mathbf{c}, \mathbf{c}') \big\rangle \\
&= a \odot \mathbf{c} + b \odot \mathbf{c}' \\
&= a \odot \mathsf{RLWE}'_s(-s \cdot m_1) + b \odot \mathsf{RLWE}'_s(m_1) \\
&= \mathsf{RLWE}_s(-s \cdot a \cdot m_1) + \mathsf{RLWE}_s(b \cdot m_1) \\
&= \mathsf{RLWE}_s(b \cdot m_1 - a \cdot s \cdot m_1) \\
&= \mathsf{RLWE}_s((b - as) \cdot m_1) \\
&= \mathsf{RLWE}_s(m_0 m_1 + e_0 m_1).
\end{aligned}$$

This is an encryption of the product $m_0 m_1$, with an additional error term $e_0 m_1$. For this operation to be effective, RGSW encryption is usually restricted to small messages m_1, typically $m_1 = \pm X^v$, so that $\|e_0 m_1\| = \|e_0\|$. This multiplication operation has type

$$(\diamond)\colon \mathsf{RLWE} \times \mathsf{RGSW} \to \mathsf{RLWE}$$

but is easily extended component-wise

$$\begin{aligned}
&(\diamond)\colon \mathsf{RLWE}' \times \mathsf{RGSW} \to \mathsf{RLWE}' \qquad & (\mathbf{c}_0, \ldots, \mathbf{c}_{k-1}) \diamond \mathbf{C} &= (\mathbf{c}_0 \diamond \mathbf{C}, \ldots, \mathbf{c}_{k-1} \diamond \mathbf{C}) \\
&(\diamond)\colon \mathsf{RGSW} \times \mathsf{RGSW} \to \mathsf{RGSW} \qquad & (\mathbf{c}, \mathbf{c}') \diamond \mathbf{C} &= (\mathbf{c} \diamond \mathbf{C}, \mathbf{c}' \diamond \mathbf{C})
\end{aligned}$$

to operate on RLWE' or RGSW ciphertexts. The last operation provides a way to multiply RGSW ciphertexts with each other $\mathsf{RGSW}(m_0) \cdot \mathsf{RGSW}(m_1) = \mathsf{RGSW}(m_0 \cdot m_1)$, and it is perhaps the most convenient operation to design protocols. But whenever one needs only a RLWE (or RLWE') encryption of the result, efficiency savings can be obtained by only using a component of $\mathsf{RGSW}(m_0)$ (which is a ciphertext of the form $\mathsf{RLWE}(m_0)$,) and computing the product $\mathsf{RLWE}(m_0)\cdot\mathsf{RGSW}(m_1) = \mathsf{RLWE}(m_0 \cdot m_1)$, e.g., as done in the TFHE scheme, and optimized implementations of FHEW.

3 Bootstrapping

We recall that LWE ciphertexts have the form $(\mathbf{a}, b)$ where $\mathbf{a} \in \mathbb{Z}_q^n$ and $b \in \mathbb{Z}_q$, and keys are vectors $\mathbf{s} \in \mathbb{Z}_q^n$, possibly chosen from a subset of $\mathbb{Z}_q^n$. Ciphertext $(\mathbf{a}, b)$ is decrypted by first computing the linear term $d = b - \langle \mathbf{a}, \mathbf{s} \rangle \in \mathbb{Z}_q$, and then "rounding" d to a message $m = f(d) \in \mathbb{Z}_t$, where $f \colon \mathbb{Z}_q \to \mathbb{Z}_t$ is an appropriate function. (Usually, $f(d) = \lceil mt/q \rfloor$ rounds d to the closest multiple of q/t.) The goal of bootstrapping is to compute this decryption function homomorphically, given an encryption $E(\mathbf{s})$ of the secret key, so that the result of the computation is again an encryption $E(m)$ of the message, but with smaller noise. In particular, the noise of the output ciphertext $E(m)$ only depends on the noise of $E(\mathbf{s})$ (and the complexity of the decryption procedure, which is fixed), but not on the noise of the ciphertext $(\mathbf{a}, b)$ which is "decrypted away".

We remark that the secret key $E(\mathbf{s})$ may be encrypted under a different scheme than LWE. Moreover, it is often convenient to consider bootstrapping procedures where the final output $\tilde{E}(m)$ is encrypted under still another encryption scheme.

A common feature of [4, 12] is that they both directly implement bootstrapping as an *arithmetic* procedure. Since arithmetic computations are well supported by lattice based encryption, this carries much lower overhead than homomorphic computing via branching programs. More specifically, using the terminology of [11], bootstrapping is implemented by means of a cryptographic *accumulator* ACC holding values from $\mathbb{Z}_q$ and supporting the following operations:

1. **Initialize:** $\mathsf{ACC} \leftarrow b$, setting the content of ACC to any known value $b \in \mathbb{Z}_q$
2. **Update:** $\mathsf{ACC} \xleftarrow{+} c \cdot E(s)$, modifying the content of the accumulator from $\mathsf{ACC}[v]$ to $\mathsf{ACC}[v + c \cdot s]$, where $c, s \in \mathbb{Z}_q$, and s is given encrypted under E.
3. **Extract:** $f(\mathsf{ACC})$, returning an encryption $\tilde{E}(f(v))$ of function f applied to the current content of the accumulator $\mathsf{ACC}[v]$.

Using this cryptographic data structure with

$$\mathrm{ek} = E(\mathbf{s}) = (E(s_1), \ldots, E(s_n))$$

as a bootstrapping (also called "evaluation" or "refreshing") key, the bootstrapping procedure is easily implemented by the pseudo-code in Figure 1. The exact computations performed by this bootstrapping procedure depend on the details of how the accumulator operations are implemented:

- The AP bootstrapping procedure [4] supports basic updates $\mathsf{ACC} \xleftarrow{+} E(s)$ for arbitrary $s \in \mathbb{Z}_q$. Then, $\mathsf{ACC} \xleftarrow{+} c \cdot E(s)$ is implemented by providing (in the bootstrapping key) encryptions $E(2^i s)$ of multiples of the secret key elements s, taking the binary expansion of $c = \sum_i 2^i c_i$, and then executing $\mathsf{ACC} \xleftarrow{+} E(2^i s)$ for all i such that $c_i = 1$.

```
Bootstrap(ek = (E(s_i))_i,(a, b)):
  ACC ← b
  for i = 1, ..., n do
    c_i = −a_i (mod q)
    ACC +← c_i · ek_i
  return f(ACC)
```

Figure 1: Arithmetic bootstrapping using a generic cryptographic accumulator ACC and *rounding function* f.

- The GINX bootstrapping procedure [12] supports basic updates $\mathsf{ACC} \xleftarrow{+} c \cdot E(s)$ where $c \in \mathbb{Z}_q$ is arbitrary, but $s \in \{0, 1\}$ is a single bit. Then, for an arbitrary secret $s = \sum_i 2^i s_i \in \mathbb{Z}_q$, one can execute $\mathsf{ACC} \xleftarrow{+} (2^i c) \cdot E(s_i)$ for all i.

The AP scheme can be generalized to offer a space/time trade-off, where $a = \sum_i {B_{\mathrm{r}}}^i a_i$ is written to a base $B_{\mathrm{r}} \geq 2$ (with digits $a_i \in \mathbb{Z}_{B_{\mathrm{r}}}$), and the bootstrapping key includes encryptions $E({B_{\mathrm{r}}}^i as)$ for all $i \leq \log_{B_{\mathrm{r}}} q$ and $a \in \mathbb{Z}_{B_{\mathrm{r}}}$. This speeds up bootstrapping by a factor $\log_2 B_{\mathrm{r}} \leq \log_2 q$, but at the cost of increasing the size of the bootstrapping key by a factor $B_{\mathrm{r}}/\log_2 B_{\mathrm{r}} \leq q/\log_2 q$.

No such trade-off is offered by the GINX scheme [12], which only supports binary expansion of the secret key $s = \sum_i 2^i s_i$. But efficiency can be improved at the cost of a stronger security assumption, by using a small secret key $\mathbf{s}$ to start with. This is why [12, 8] make the non-standard assumption that the key $\mathbf{s}$ has binary entries, which results is the highest possible performance improvement $\log_2 q = O(\log n)$ (over the same scheme with arbitrary secrets $s \in \mathbb{Z}_q$, or even "small" secrets $|s| = O(\sqrt{n})$ following the error distribution.)

In this paper, we consider standard instantiations of GINX, e.g., using ternary secrets $s_i \in \{-1, 0, +1\}$ [2] or gaussian secrets $s_i = O(\sqrt{n})$ [17, 21], which offer better (or at least better studied) security, at a modest performance penalty.

3.1 Ring LWE accumulators

We now describe how to implement the cryptographic accumulators ACC required by the bootstrapping procedure. Since, for efficiency's sake, we will use ring lattices, we focus on the Ring LWE-based accumulators proposed in [11] and further refined/optimized in [8].

The main idea of the FHEW accumulators [11] (building on a suggestion of [4]) is to work in a cyclotomic ring of order q, and represent values $v \in \mathbb{Z}_q$ as $X^v \in \mathbb{Z}[X]/\Phi_q(X)$, where $\Phi_q(X)$ is the qth cyclotomic polynomial. Since X has multiplicative order q in $\mathbb{Z}[X]/\Phi_q(X)$, addition in the exponent of X^v is performed modulo q. Following [11], we assume $q = 2^k$ is a power of 2, and use the ring $R_Q = \mathbb{Z}_Q[X]/(X^{q/2} + 1)$ for a sufficiently larger modulus Q, but generalizations to other cyclotomic rings are possible, e.g., see [5, 6]. The ring R_Q is used to implement the $\mathsf{RLWE}, \mathsf{RLWE}'$ and RGSW encryption schemes described in Section 2. The smaller modulus q is only used for the LWE ciphertext given as input to the bootstrapping procedure.

Note that in the implementation, we use N larger than $q/2$ to achieve a desired security level for the $\mathsf{RLWE}/\mathsf{RGSW}$ schemes. The requirement in this case is that q divides $2N$ so that we could embed the ring $\mathbb{Z}_Q[X]/(X^{q/2}+1)$ into $\mathbb{Z}_Q[X]/(X^N+1)$. Please see the PALISADE implementation [1] for more details.

FHEW accumulators store the value $v \in \mathbb{Z}_q$ as $\mathsf{ACC}[v] = \mathsf{RGSW}(X^v)$, and support the computation of any function f (during extraction) such that

$$f(v + (q/2)) = -f(v). \tag{3}$$

Two optimizations are possible, and will be used in this paper:

- If the function f is known in advance, the value v can be alternatively stored as

$$\mathsf{ACC}[v] = \mathsf{RGSW}(\sum_{i=0}^{q/2-1} f(v-i) \cdot X^i).$$

  This leads to a simpler and more efficient extraction procedure.

- If only an LWE ciphertext is needed at the end of the computation, one can set the accumulator

$$\mathsf{ACC}[v] = \mathsf{RLWE}(X^v)$$

  to a simple RLWE ciphertext, i.e., a single component of a RGSW ciphertext.

We remark that the more complex accumulators may be useful, e.g., to support multiple functions f, and more advanced bootstrapping methods, like [20]. But for the main purpose of this paper, both optimizations can be used, and we set

$$\mathsf{ACC}[v] = \mathsf{RLWE}(\sum_{i=0}^{q/2-1} f(v-i) \cdot X^i).$$

We write ACC_f to emphasize that the value of the extraction function is fixed at initialization time.

The initialization and extraction operations are easily implemented:

- **Initialize:** $\mathsf{ACC}_f \leftarrow v$ simply sets ACC_f to a noiseless encryption $\mathsf{RLWE}(m) = (0, m)$ of the polynomial

$$m(X) = \sum_{i<q/2} f(v-i) \cdot X^i.$$

- **Extract:** if $\mathsf{ACC}_f = (a, b)$ is the RLWE ciphertext with component polynomials $a(X) = \sum_{i<q/2} a_i X^i, b(X) = \sum_{i<q/2} b_i X^i$, the extraction operation outputs the LWE ciphertext

$$f(\mathsf{ACC}_f) = (\mathbf{a}, b_0),$$

  where $\mathbf{a} = (a_0, \ldots, a_{q/2-1})$ is the coefficient vector of $a(X)$.

The pseudo-code of the initialization and extraction procedures is given is Figure 2.

The initialization procedure is clearly correct, and it has the useful feature of not introducing any noise. For the extraction procedure, recall that

$$b(X) = z(X) \cdot a(X) + e(X) + m(X) \bmod (Q, X^{q/2}+1)$$

$$\mathsf{ACC}_f[v] = \mathsf{RLWE}\left(\sum_{i=0}^{q/2-1} f(v-i)\cdot X^i\right) \in R_Q^2$$

$\mathsf{Init}_f(v)$:
  for $i = 0,\ldots,q/2-1$
    $m_i = f(v-i)$
  $\mathbf{m} = \sum_{i<q/2} m_i X^i = (m_0,\ldots,m_{q/2-1})$
  return $(\mathbf{0},\mathbf{m})$

$\mathsf{Extract}_f(\mathbf{a},\mathbf{b})$:
  let $(b_0,\ldots,b_{q/2-1}) = \mathbf{b}$
  return $(\mathbf{a}, b_0)$

Figure 2: FHEW accumulators and initialization/extraction procedures for any fixed function $f\colon \mathbb{Z}_q \to \mathbb{Z}_Q$ such that $f(v+q/2) = -f(v)$. Init takes as input a value $v\in\mathbb{Z}_q$, and outputs an accumulator $\mathsf{ACC}_f[v]$ holding it. The $\mathsf{Extract}$ function takes as input an RLWE ciphertext representing an accumulator $\mathsf{ACC}_f[v]$ for a given function f and key $z(X)\in R_Q$, and outputs an LWE encryption (modulo Q) of $f(v)\in\mathbb{Z}_Q$ with respect to key $\mathbf{z} = (z_0,-z_{q/2-1},\ldots,-z_1)\in\mathbb{Z}_Q^{q/2}$. In the implementation, it is more convenient to compute the transpose of $\mathbf{a}$ to get an encryption of the result under the original (rather than permuted) secret.

for some secret key polynomial $z(X)\in R_Q$ (typically different from the input LWE secret key $\mathbf{s}$) and small noise polynomial $e(X)$. So, the constant term $b_0 = b(0)$ of this polynomial equals

$$\begin{aligned} b_0 &= z_0\cdot a_0 + \sum_{i=1}^{q/2-1} z_i a_{q/2-i}\cdot(-1) + e(0) + m(0) \\ &= \langle \mathbf{a},\mathbf{z}\rangle + e_0 + m(0), \end{aligned}$$

where $\mathbf{z} = (z_0,-z_{q/2-1},\ldots,-z_1)$ is a signed permutation of the coefficients of the secret key polynomial $z(X)$. Notice that $m(0) = f(v)$. So, $(\mathbf{a}, b_0)$ is precisely an LWE encryption of $f(v)$ under key $\mathbf{z}$ with small noise $e_0 = e(0)$. Notice that the resulting LWE ciphertext uses a different dimension $q/2$ and modulus Q than the input LWE ciphertext given to the bootstrapping procedure. An LWE encryption of $f(v)$ under key $\mathbf{s}$, in dimension n and modulus q, can be obtained using a standard key-switching operation.

We now move to the update operation $\mathsf{ACC} \overset{+}{\leftarrow} c\cdot E(s)$. This operation is implemented differently in the AP/FHEW and GINX/TFHE bootstrapping procedures, and using different methods to encrypt the secret values $E(s_i)$. But both methods operate on the same FHEW accumulators described in Figure 2, and support he same initialization and extraction procedures.

- In AP/FHEW, each secret value $s\in\mathbb{Z}_q$ is encrypted as

  $$E(s) = \{\mathbf{Z}_{j,v} = \mathsf{RGSW}(X^{vB_\mathrm{r}{}^j\cdot s}) \mid i < \log_{B_\mathrm{r}} q, v\in\mathbb{Z}_{B_\mathrm{r}}\}.$$

  Then, the update operation $\mathsf{ACC}[v] \overset{+}{\leftarrow} c\cdot E(s)$ is computed by writing $c = \sum_i B_\mathrm{r}{}^i c_i$ in base B_r, and sequentially updating $\mathsf{ACC} := \mathsf{ACC}\diamond\mathbf{Z}_{j,c_j}$ for $j = 0,\ldots,\log_{B_\mathrm{r}} q - 1$, using the $\mathsf{RLWE}\times\mathsf{RGSW}\to\mathsf{RLWE}$ multiplication operation. The pseudo-code is given in Figure 3.

- In GINX/TFHE, each secret value $s\in\mathbb{Z}_q$ needs to be expressed as a subset-sum $s = \sum_{u\in U} u\cdot x_i$ (with $x_i\in\{0,1\}$) where $U\subset\mathbb{Z}_q$ is an appropriate subset of $\mathbb{Z}_q$. For example, for binary

$$E(s) = \{\mathbf{Z}_{j,v} = \mathsf{RGSW}(X^{v B_r{}^j \cdot s}) \mid j < \log_{B_r} q, v \in \mathbb{Z}_{B_r}\} \in \mathsf{RGSW}^{B_r \log q / \log B_r}.$$

```
Update(c,{Z_{j,v}}_{j,v} = E(s)):
  for j = 0, ..., log_{B_r} q − 1
    c_j = ⌊c/B_r^j⌋ mod B
    if c_j > 0
      ACC ← ACC ⋄ Z_{j,c_j}
  return ACC
```

Figure 3: Encryption function E and accumulator update method for the FHEW bootstrapping procedure.

secrets one can simply set $U = \{1\}$. Arbitrary elements of $\mathbb{Z}_q$ can be expressed using $U = \{1, 2, 4, \ldots, 2^{k-1}\}$. For ternary secrets one can use $U = \{1, -1\}$, or $U = \{1, -2\}$ to make the representation unambiguous.

For any such U, the secret encryption function is defined as

$$E'(s) = \{\mathbf{Z}_u = \mathsf{RGSW}(x_u) \mid \text{for some } \mathbf{x} \in \{0,1\}^U \text{ such that } \textstyle\sum_u u \cdot x_u = s\}.$$

Then, the update operation $\mathsf{ACC}[v] \xleftarrow{+} c \cdot E(s)$ is computed by sequentially updating

$$\mathsf{ACC} := \mathsf{ACC} \diamond \bar{\mathbf{Z}}_u + (X^{u \cdot c} \cdot \mathsf{ACC}) \diamond \mathbf{Z}_u$$

for $u \in U$, where

$$\bar{\mathbf{Z}}_u = \mathbf{G} - \mathbf{Z}_u = \mathsf{RGSW}(1) - \mathbf{Z}_u$$

is the (homomorphic) logical negation of the encrypted bit $\mathbf{Z}_u$.

When implementing the GINX/TFHE bootstrapping procedure, it is more efficient to write the update operation as

$$\mathsf{ACC} := \mathsf{ACC} + (X^{u \cdot c} - 1)\,(\mathsf{ACC} \diamond \mathbf{Z}_u)\,.$$

This form requires only a single $\mathsf{RLWE} \times \mathsf{RGSW}$ product.

See Figure 4 for the pseudo-code.

3.2 Rounding functions and Boolean gates

The full bootstrapping procedure is obtained by implementing the algorithm in Figure 1 using the accumulator described in Figure 2 and the update procedure from either Figure 3 or Figure 4.

We conclude with a discussion of the function f used by the extraction function at the end of bootstrapping. In its simplest instantiation, the FHEW encryption scheme uses ciphertexts of the form $\mathsf{LWE}_{\mathbf{s}}(m \cdot (q/4)) = (\mathbf{a}, b)$, for message bit $m \in \{0,1\}$ with error $e = b - \langle \mathbf{a}, \mathbf{s} \rangle - m \cdot (q/4)$ bounded by $|e| < q/16$. The NAND of two ciphertexts $(\mathbf{a}_0, b_0) = \mathsf{LWE}(m_0(q/4))$ and $(\mathbf{a}_1, b_1) = \mathsf{LWE}(m_1(q/4))$ is computed by first adding them up, to obtain an encryption $(\mathbf{a}, b) = (\mathbf{a}_0 + \mathbf{a}_1, b_0 + b_1)$ of $m_0 + m_1 \in \{0, 1, 2\}$ with noise less than $|e_0 + e_1| < q/8$. Then, this ciphertext $(\mathbf{a}, b) = \mathsf{LWE}((m_0 + m_1)(q/4))$ is homomorphically decrypted using the bootstrapping procedure from Figure 1. This requires a function f that

$$E'\left(t = \sum T\right) = \{\mathbf{Z}_u = \mathsf{RGSW}(I_T(u)) \mid u \in U\}$$

$\mathsf{Update}'(c, \{\mathbf{Z}_u\}_u = E'(t))$:
  for $u \in U$
    $\mathsf{ACC} \leftarrow \mathsf{ACC} + (X^{u \cdot c} - 1)\,(\mathsf{ACC} \diamond \mathbf{Z}_u)$
  return ACC

Figure 4: Encryption function E' and accumulator update method for the TFHE bootstrapping procedure. The function E' encrypts values $t \in \mathbb{Z}_q$ that can be written as a subset sum $t = \sum T$ for some $T \subseteq U$. $I_T(u) \in \{0, 1\}$ is the indicator function of set T, i.e., $I_T(u) = 1$ if $u \in T$, and $I_T(u) = 0$ otherwise.

- rounds $m + e = (m_0 + m_1)(q/4) + (e_0 + e_1)$ to the closest multiple of $q/4$, then
- maps $0 \mapsto 1$, $1 \mapsto 1$, $2 \mapsto 0$, (and, for completeness, also $3 \mapsto 0$), and
- finally multiplies the resulting bit by the scaling factor $Q/4$. (Remember that the extraction procedure returns an LWE ciphertext modulo Q. A ciphertext modulo q can be obtained using key/modulus switching.)

In summary, the function f should map the set $[0, q/4] \pm (q/8) = (-q/8, 3q/8) \subset \mathbb{Z}_q$ to $q/4$, and its complement $[2q/4, 3q/4] \pm (q/8) = (3q/8, 7q/8)$ to 0. However, this mapping does not satisfy the requirement $f(v + q/2) = -f(v)$ imposed by the bootstrapping procedure. This issue is addressed using a function f that maps $(-q/8, 3q/8)$ to $q/8$ and $(3q/8, 7q/8)$ to $-q/8$. Using this function, the result of the bootstrapping procedure is an LWE encryption of $m = \neg(m_0 \wedge m_1) \in \{0, 1\}$, encoded as $\mathsf{LWE}((2m-1) \cdot (q/8))$. Adding a noiseless encryption $(0, q/8) = \mathsf{LWE}(q/8)$ of $q/8$ to this ciphertext, yields an encryption of m under the standard encoding

$$\mathsf{LWE}((2m - 1) \cdot (q/8) + (q/8)) = \mathsf{LWE}(m \cdot (q/4)).$$

For reference, the code of the full NAND operation is given in Figure 5, using the AP/FHEW bootstrapping procedure, and Figure 6 using the GINX/TFHE update operations.

The same approach can be used to encode other Boolean gates. The homomorphic operations and mapping ranges for some common binary and ternary Boolean gates are listed in Table 1. Each of these gates requires a single bootstrapping operation.

The only Boolean operation that does not require bootstrapping is NOT. For an LWE ciphertext $(\mathbf{a}, b)$, the NOT gate is evaluated as $(-\mathbf{a}, -b + q/4)$.

4 Theoretical Comparison of FHEW/AP and TFHE/GINX

To compare the computational complexity and bootstrapping key size of FHEW/AP and TFHE/GINX, we use the following parameters:

- q, small (LWE) modulus;
- n, lattice parameter for the LWE scheme;

$\mathsf{EvalKeyGen}(z, \mathbf{s})$:

for $i = 1, \ldots, n$

  for $j = 0, \ldots, \log_{B_r} q - 1$

    for $v \in 0, \ldots, B_r - 1$

      $\mathbf{Z}_{i,j,v} = \mathsf{RGSW}_z(X^{v B_r{}^j \cdot s_i})$

$\mathrm{ek} = \{\mathbf{Z}_{i,j,v}\}_{i,j,v}$

return ek

$\mathsf{NAND}(\mathrm{ek}, \mathbf{c}_0, \mathbf{c}_1)$:

$(\mathbf{a}, b) = \mathbf{c}_0 + \mathbf{c}_1$

$p = 2N/q$

for $j = 0, \ldots, q/2 - 1$:

  if $(b - j) \bmod q \in [3q/8, 7q/8)$

  then $m_{j \cdot p} = -Q/8$

  else $m_{j \cdot p} = Q/8$

$\mathbf{m} = (m_0, \ldots, m_{(q/2-1) \cdot p})$

$\mathsf{ACC} \leftarrow (\mathbf{0}, \mathbf{m})$

for $i = 1, \ldots, n$

  $c = -a_i \bmod q$

  for $j = 0, \ldots, \log_{B_r} q - 1$

    $c_j = \lfloor c / B_r^j \rfloor \bmod B_r$

    if $c_j > 0$

      $\mathsf{ACC} \leftarrow \mathsf{ACC} \diamond \mathbf{Z}_{i,j,c_j}$

$(\mathbf{a}', \mathbf{b}') = \mathsf{ACC}$

$b'_0 = \mathbf{b}'(0)$

return $(\mathbf{a}', b'_0 + Q/8 \bmod Q)$

Figure 5: FHEW-style encrypted NAND computation. The input values $\mathbf{c}_b = \mathsf{LWE}_{\mathbf{s}}((q/4) \cdot m_b)$ are LWE ciphertexts encrypting message bits $m_b \in \{0, 1\}$ for $b = 0, 1$ under key $\mathbf{s} \in \mathbb{Z}_q^n$ with noise bounded by $q/16$. The bootstrapping key ek encrypts $\mathbf{s}$ under $z \in R_Q$. The output is an $\mathsf{LWE}_{\mathbf{z}}((Q/4)m)$ encryption of the NAND $m = \neg(m_0 \wedge m_1)$ under key $\mathbf{z} = (z_0, -z_{q/2-1}, \ldots, -z_1) \in \mathbb{Z}_Q^{2^k}$. Key-switching (not shown) can be used to turn the output into an LWE encryption under $\mathbf{s}$, and perform additional NAND operations.

$\mathsf{EvalKeyGen}(z, \mathbf{s})$:

  for $i = 1, \ldots, n$

    pick $T \subseteq U$ such that $s_i = \sum T \pmod q$

    for $u \in U$

      if $u \in T$

      then $\mathbf{Z}_{i,u} = \mathsf{RGSW}_z(1)$

      else $\mathbf{Z}_{i,u} = \mathsf{RGSW}_z(0)$

  $\mathrm{ek} = \{\mathbf{Z}_{i,u}\}_{i,u}$

  return ek

$\mathsf{NAND}(\mathrm{ek}, \mathbf{c}_0, \mathbf{c}_1)$:

  $(\mathbf{a}, b) = \mathbf{c}_0 + \mathbf{c}_1$

  $p = 2N/q$

  for $j = 0, \ldots, q/2 - 1$:

    if $(b - j) \bmod q \in [3q/8, 7q/8)$

    then $m_{j \cdot p} = -Q/8$

    else $m_{j \cdot p} = Q/8$

  $\mathbf{m} = (m_0, \ldots, m_{(q/2-1)\cdot p})$

  $\mathsf{ACC} \leftarrow (\mathbf{0}, \mathbf{m})$

  for $i = 1, \ldots, n$

    for $u \in U$

      $\mathsf{ACC} \leftarrow \mathsf{ACC} + (X^{u \cdot c} - 1)\,(\mathsf{ACC} \diamond \mathbf{Z}_{i,u})$

  $(\mathbf{a}', \mathbf{b}') = \mathsf{ACC}$

  $b'_0 = \mathbf{b}'(0)$

  return $(\mathbf{a}', b'_0 + Q/8)$

Figure 6: Encrypted NAND computation with GINX/TFHE-style accumulator updates. Input and output are as in Figure 5.

Table 1: Additive homomorphic computations and mappings for Boolean gates; $\mathbf{c}_i$ is an encryption of i-th Boolean input.

Gate	Computation	maps to $q/8$	maps to $-q/8$
AND	$\mathbf{c}_1 + \mathbf{c}_2$	$[3q/8, 7q/8)$	$[-q/8, 3q/8)$
NAND	$\mathbf{c}_1 + \mathbf{c}_2$	$[-q/8, 3q/8)$	$[3q/8, 7q/8)$
OR	$\mathbf{c}_1 + \mathbf{c}_2$	$[q/8, 5q/8)$	$[-3q/8, q/8)$
NOR	$\mathbf{c}_1 + \mathbf{c}_2$	$[-3q/8, q/8)$	$[q/8, 5q/8)$
XOR	$2\,(\mathbf{c}_1 - \mathbf{c}_2)$	$[q/8, 5q/8)$	$[-3q/8, q/8)$
XNOR	$2\,(\mathbf{c}_1 - \mathbf{c}_2)$	$[-3q/8, q/8)$	$[q/8, 5q/8)$
Majority	$\mathbf{c}_1 + \mathbf{c}_2 + \mathbf{c}_3$	$[3q/8, 7q/8)$	$[-q/8, 3q/8)$

- Q, RLWE/RGSW modulus;
- N, ring dimension for RLWE/RGSW;
- B_g, gadget base for digit decomposition in each accumulator update, which breaks integers $\bmod Q$ into d_g digits;
- B_ks, gadget base used for key switching, which breaks integers $\bmod Q$ into d_ks digits;
- B_r, gadget base in FHEW/AP (not used in TFHE/GINX), which breaks integers $\bmod q$ into d_r digits.

Note that we have applied all other TFHE algorithmic optimizations [8, 9, 10] to both FHEW/AP and TFHE/GINX. These include the precomputation of f before extraction, which reduces the noise growth, and treating ACC as an RLWE rather than a RGSW ciphertext, which improves the runtime of bootstrapping by a factor of $2d_\mathrm{g}$. Both of these optimizations were not included in the original FHEW implementation [11]. All other TFHE optimizations described in [8, 9] are specific to the floating-point arithmetic implementation of FFT/torus operations, and hence ignored in our analysis.

4.1 Computational complexity

The bottleneck operation in both FHEW/AP and TFHE/GINX is the Number Theoretic Transform (NTT) that is used to switch a ring element in $R_Q = \mathbb{Z}_Q[X]/(X^N+1)$ from coefficient to evaluation representation and back. NTT operations are called inside each digit decomposition in the RLWE $\times$ RGSW products of the accumulator update steps for both bootstrapping methods.

The pseudocode in Figures 5-6 and the corresponding implementation in PALISADE can be used to estimate the number of NTTs needed for each method:

- FHEW/AP: $2\left(1-\frac{1}{B_\mathrm{r}}\right) n d_\mathrm{r}(d_\mathrm{g}+1)$ NTTs.
- TFHE/GINX: $2n|U|(d_\mathrm{g}+1)$ NTTs, where $|U|$ is the cardinality of U, i.e., the number of bits needed to represent the full range of secret key samples with a given (very high) probability.

Note that the accumular updates also require $2\left(1-\frac{1}{B_\mathrm{r}}\right) n d_\mathrm{r} d_\mathrm{g}$ and $2n|U|(d_\mathrm{g}+1)$ vector component-wise modular multiplications for FHEW/AP and TFHE/GINX, respectively. However, the contribution of these vector multiplications is relatively small compared to the NTTs, and hence can be excluded from our analysis for simplicity.

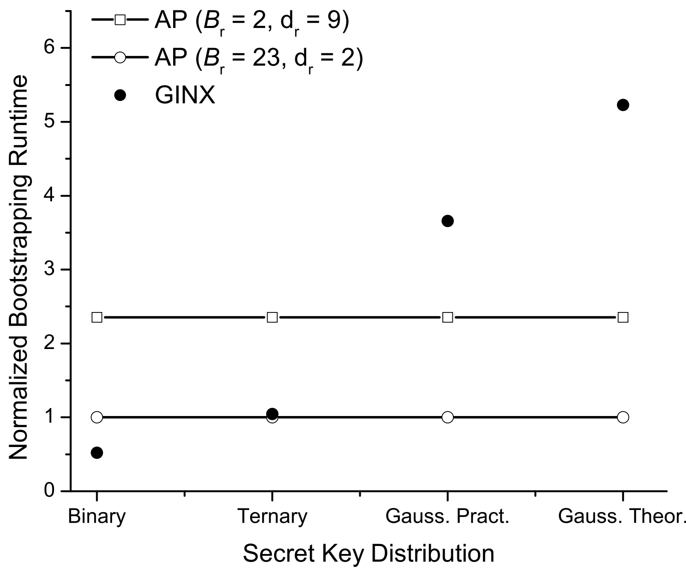

Figure 7: Comparison of boostrapping computational complexity of TFHE/GINX and FHEW/AP methods. All estimates of computational complexity are normalized to the AP complexity for $B_r = 23, d_r = 2, q = 512$ considered in [11]. "Gauss Pract." corresponds to the Gaussian secret key distribution with standard deviation $\sigma = 3.19$; "Gauss. Theor." to the case of $\sigma = \sqrt{n}$; both secret key distributions are assumed to be bounded by 12σ.

The ratio of runtimes for TFHE/GINX and FHEW/AP can be expressed as $|U|/\left(1 - \frac{1}{B_\mathrm{r}}\right) d_\mathrm{r}$. Figure 7 compares the computational complexity of FHEW/AP and TFHE/GINX for different secret key distributions, ranging from binary secret key distribution used in TFHE to theoretically secure Gaussian distribution corresponding to the standard RLWE assumption. For FHEW/AP, we include the cases of $B_r = 2$ (classical AP bootstrapping) and $B_r = 23$ (optimized setting considered in [11]). We assume that $q = 512$, which correspods to typical practical settings (as discussed in Section 5). Although the value of q may increase as we go from binary/ternary secret key distributions to Gaussian secrets, the ratio between GINX and AP is still expected to be quite accurate.

We see that the TFHE/GINX botstrapping is roughly twice faster than AP with $B_r = 23$ for binary secret distribution, which is used in the TFHE implementation [8]. Once we switch to the ternary secret distribution, the runtimes become about the same (with FHEW/AP being slightly faster than THFE/GINX due to the $1 - {B_\mathrm{r}}^{-1}$ factor). When the norm of secret key distribution is further increased, which corresponds to Gaussian distribution, the runtime of the FHEW/AP bootstrapping procedure becomes significantly smaller. The FHEW/AP bootstrapping at $B_r = 2$ is also faster than GINX for Gaussian secret distributions.

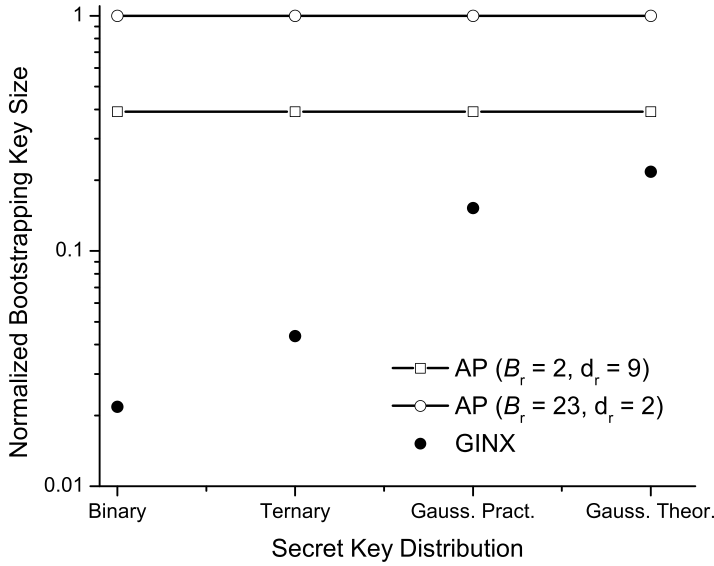

Figure 8: Comparison of boostrapping key size for TFHE/GINX and FHEW/AP methods. All estimates of bootstrapping key size are normalized to the AP key size for $B_r = 23, d_r = 2, q = 512$ considered in [11]. "Gauss Pract." corresponds to the Gaussian secret key distribution with standard deviation $\sigma = 3.19$; "Gauss. Theor." to the case of $\sigma = \sqrt{n}$; both secret key distributions are assumed to be bounded by 12σ.

4.2 Bootstrapping key size

One of the practical limitations of the original FHEW implementation [11] is the bootstrapping key size. The key sizes can be estimated as:

- AP/FHEW: $4nNd_{\mathrm{r}}B_{\mathrm{r}}d_{\mathrm{g}}\log_2 Q$ bits;
- TFHE/GINX: $4nN|U|d_{\mathrm{g}}\log_2 Q$ bits.

It is convenient to consider the ratio of $|U|/\left(d_{\mathrm{r}}B_{\mathrm{r}}\right)$. Figure 8 illustrates the comparison of key sizes for FHEW/AP and TFHE/GINX at different secret key distributions. For any practical secret key distribution, TFHE/GINX bootstrapping requires a significantly smaller key size. However, the key size improvement becomes less pronounced as the norm of secret key distribution is increased. For instance, the key size for TFHE/GINX bootstrapping is only 2 times smaller than for FHEW/AP at $B_r = 2$, which implies that this AP setting may be preferred over GINX for Gaussian secret distributions due to better computational complexity of the bootstrapping.

In addition to the bootstrapping key, both bootstraping procedures require a switching key to switch LWE ciphertexts to the original secret key. The size of switching key for both bootstrapping methods is the same, and is equal to $nNB_{\mathrm{ks}}d_{\mathrm{ks}}\log_2 Q$ bits.

4.3 Recommendation for ternary secret key distribution

The most efficient case included in the HE Security Standard [2] is based on ternary secret key distribution. For this setting, our analysis shows that the runtimes of FHEW/AP and TFHE/GINX bootstrapping procedures are roughly the same while the bootstrapping key size is more than one order of magnitude smaller for TFHE/GINX. Therefore, the TFHE/GINX method is preferred for this setting. However, the AP/FHEW bootstrapping at $B_r = 2$ becomes a more efficient option than TFHE/GINX when Gaussian secret key distributions are considered.

5 Implementation Results

5.1 Parameter sets

In PALISADE, we added several parameter sets corresponding to various levels of security, including MEDIUM, STD128, STD192, STD256, STDQ128, STD192Q, and STD256Q (see Figure 2). All parameters sets are for ternary secret distribution. The parameter sets starting with "STD" use the tables in the HE security standard [2]. The numbers correspond to the estimated bits of security. The suffix "Q" stands for the quantum attack estimates.

The correctness of the parameters was checked using both numerical experiments and theoretical noise estimates. In numerical experiments, for each parameter set we recorded the actual values of the error/noise for a relatively large sample (about 500 bootstrapping runs), and then estimated the mean and standard deviation of the error. Assuming normal distribution of the error, we estimated the decryption failure probability, i.e., the probabilty of the error exceeding $Q/8$ or $q/8$, for both FHEW/AP and TFHE/GINX cryptosystems. We observed that the decryption failure probability after bootstrapping is always slightly lower for FHEW/AP: the average value of error is smaller by about half bit (around 1.5x), which is due to extra additions in the TFHE/GINX case for ternary secret distributions. For all parameter sets except for MEDIUM and STD256Q, the decryption failure probability due to additions in LWE (we estimated it to be about 2^{-85} for $q = 512$) was higher than for the bootstrapping operation.

In this work, we provide implementation results for the MEDIUM and STD128 parameter sets.

The MEDIUM parameter set corresponds to 108 bits of security and 100 bits of security for classical and quantum computers, respectively, based on the estimates obtained using the LWE estimator [3]. We ran the LWE security estimator[1] (commit b0a23f4) [3] to find the lowest security levels for the uSVP, decoding, and dual attacks following the HE Security Standard recommendations [2]. We selected the least value of the number of security bits λ for all 3 attacks on classical computers based on the estimates for the BKZ sieve reduction cost model. The "MEDIUM"" security level is roughly the same (in terms of bits of security) as in prior FHEW/TFHE implementations [11, 8, 9].

The STD128 parameter set corresponds to 128 bits of security for classical computers, and is the main parameter set recommended for use in practice.

5.2 Software implementation

We implemented both bootstrapping methods in PALISADE v1.8.0. Our implementation is publicly available [1]. The evaluation environment was a commodity desktop computer system with an

[1] https://bitbucket.org/malb/lwe-estimator

Table 2: Parameter sets for ternary secret distribution; P_{AP} and P_{GINX} are estimated upper bounds for decryption failure probabilities of FHEW/AP and TFHE/GINX, respectively.

Parameter Set	n	q	N	$\log_2 Q$	B_{ks}	B_{g}	B_{r}	P_{AP}	P_{GINX}
MEDIUM	256	512	1024	32	25	2^{16}	23	2^{-65}	2^{-30}
STD128	512	512	1024	27	25	2^{9}	23	2^{-85}	2^{-85}
STD192	512	512	2048	37	25	2^{19}	23	2^{-85}	2^{-85}
STD256	1024	1024	2048	29	25	2^{10}	32	2^{-100}	2^{-100}
STD128Q	512	512	1024	25	25	2^{7}	23	2^{-85}	2^{-85}
STD192Q	1024	1024	2048	32	25	2^{11}	32	2^{-100}	2^{-100}
STD256Q	1024	1024	2048	27	25	2^{9}	32	2^{-50}	2^{-30}

Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz and 64 GB of RAM, running Ubuntu 18.04 LTS. The compiler was g++ (GCC) 7.4.0.

5.3 Runtime results

Table 3 summarizes the runtime results for both bootstrapping and key switching operations (the key switching runtime was included in the bootstraping runtime). These runtime results were obtained using PALISADE v1.9.1.

Table 3: Single-threaded timing results for gate evaluation (bootstrapping)

Parameter Set	AP [ms]	GINX [ms]	KeySwitch [ms]
MEDIUM	74	79	6.6
STD128	195	207	10.8

The runtime of AP/FHEW is about 6% smaller for both MEDIUM and STD128. The gap is due to a slightly smaller number of NTTs and smaller number of multiplications for AP.

6 Concluding Remarks

We present a theoretical comparison of the FHEW/AP and TFHE/GINX cryptosystems for common secret key distributions. Our analysis suggests that the TFHE/GINX cryptosystem is more efficient for binary and ternary secret key distributions while the AP bootstrapping provides better computational complexity for Gaussian secret key distributions. We also provide an open-source implementation of both cryptosystems in PALISADE.

Our implementation in PALISADE does not use any AVX extensions and barely uses any assembly-level optimizations. As a result, it is significantly slower than the TFHE results reported in [9]. If we want to estimate how the runtime reported for the TFHE library increases when we switch to a standardized HE setting (ignoring for simplicity the runtime differences between floating-point operations/FFTs and modular operations/NTTs), we should expect a 2.7x slowdown, i.e., 13 ms would change to 35 ms. A factor of 2x is introduced by going from binary secret distribution to the ternary one. Additional factor of 4/3 is because we have to use a smaller Q, 27 vs. 32 bits, which requires 3 digits to avoid decryption failures, i.e., $d_g = 3$ vs. $d_g = 2$ in the TFHE setting. For more accurate performance results, an AVX-optimized implementation based on NTTs and modular arithmetic would need to be evaluated.

As the FHEW/AP and TFHE/GINX cryptosystems based on ternary uniform and Gaussian secret distributions (described in this paper and implemented in PALISADE) satisfy all requirements of the HE Security Standard [2], we make a recommendation to consider these variants for standardization by the HE community.

References

[1] PALISADE Lattice Cryptography Library (release 1.8.0). `https://palisade-crypto.org/`, Jan. 2020.

[2] M. Albrecht, M. Chase, H. Chen, J. Ding, S. Goldwasser, S. Gorbunov, S. Halevi, J. Hoffstein, K. Laine, K. Lauter, S. Lokam, D. Micciancio, D. Moody, T. Morrison, A. Sahai, and V. Vaikuntanathan. Homomorphic encryption security standard. Technical report, HomomorphicEncryption.org, Toronto, Canada, November 2018.

[3] M. Albrecht, S. Scott, and R. Player. On the concrete hardness of learning with errors. *Journal of Mathematical Cryptology*, 9(3):169–203, 10 2015.

[4] J. Alperin-Sheriff and C. Peikert. Faster bootstrapping with polynomial error. In J. A. Garay and R. Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 297–314. Springer, 2014.

[5] J. Biasse and L. Ruiz. FHEW with efficient multibit bootstrapping. In K. E. Lauter and F. Rodríguez-Henríquez, editors, *Progress in Cryptology - LATINCRYPT 2015 - 4th International Conference on Cryptology and Information Security in Latin America, Guadalajara, Mexico, August 23-26, 2015, Proceedings*, volume 9230 of *Lecture Notes in Computer Science*, pages 119–135. Springer, 2015.

[6] G. Bonnoron, L. Ducas, and M. Fillinger. Large FHE gates from tensored homomorphic accumulator. In A. Joux, A. Nitaj, and T. Rachidi, editors, *Progress in Cryptology - AFRICACRYPT 2018 - 10th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 7-9, 2018, Proceedings*, volume 10831 of *Lecture Notes in Computer Science*, pages 217–251. Springer, 2018.

[7] Z. Brakerski and V. Vaikuntanathan. Lattice-based FHE as secure as PKE. In M. Naor, editor, *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 1–12. ACM, 2014.

[8] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *ASIACRYPT (1)*, volume 10031 of *Lecture Notes in Computer Science*, pages 3–33, 2016.

[9] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. Faster packed homomorphic operations and efficient circuit bootstrapping for TFHE. In T. Takagi and T. Peyrin, editors, *Advances in Cryptology - ASIACRYPT 2017 - 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I*, volume 10624 of *Lecture Notes in Computer Science*, pages 377–408. Springer, 2017.

[10] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène. TFHE: Fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33:34–91, 2020.

[11] L. Ducas and D. Micciancio. FHEW: bootstrapping homomorphic encryption in less than a second. In *EUROCRYPT (1)*, volume 9056 of *Lecture Notes in Computer Science*, pages 617–640. Springer, 2015.

[12] N. Gama, M. Izabachène, P. Q. Nguyen, and X. Xie. Structural lattice reduction: Generalized worst-case to average-case reductions and homomorphic cryptosystems. In M. Fischlin and J. Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 528–558. Springer, 2016.

[13] C. Gentry. Fully homomorphic encryption using ideal lattices. In *STOC*, pages 169–178. ACM, 2009.

[14] C. Gentry, S. Halevi, and N. P. Smart. Fully homomorphic encryption with polylog overhead. In *EUROCRYPT*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012.

[15] C. Gentry, A. Sahai, and B. Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 75–92. Springer, 2013.

[16] S. Halevi and V. Shoup. Bootstrapping for helib. In E. Oswald and M. Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 641–670. Springer, 2015.

[17] V. Lyubashevsky, C. Peikert, and O. Regev. On ideal lattices and learning with errors over rings. *J. ACM*, 60(6):43:1–43:35, 2013.

[18] D. Micciancio. Fully homomorphic encryption from the ground up. Invited Talk, Eurocrypt 2019, 2019.

[19] D. Micciancio and C. Peikert. Hardness of SIS and LWE with small parameters. In *CRYPTO (1)*, volume 8042 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 2013.

[20] D. Micciancio and J. Sorrell. Ring packing and amortized FHEW bootstrapping. In I. Chatzigiannakis, C. Kaklamanis, D. Marx, and D. Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming, ICALP 2018, July 9-13, 2018, Prague, Czech Republic*, volume 107 of *LIPIcs*, pages 100:1–100:14. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2018.

[21] O. Regev. On lattices, learning with errors, random linear codes, and cryptography. *J. ACM*, 56(6):34:1–34:40, 2009.

[22] N. P. Smart and F. Vercauteren. Fully homomorphic SIMD operations. *Des. Codes Cryptogr.*, 71(1):57–81, 2014.