MOSFHET: Optimized Software for FHE over the Torus

ANTONIO GUIMARÃES and EDSON BORIN, Institute of Computing, University of Campinas, Brazil
DIEGO F. ARANHA, DIGIT Centre and Department of Computer Science, Aarhus University, Denmark

Homomorphic encryption is one of the most secure solutions for processing sensitive information in untrusted environments, and there have been many recent advances towards its efficient implementation for the evaluation of linear functions and approximated arithmetic. However, the practical performance when evaluating arbitrary (nonlinear) functions is still a major challenge for HE schemes. The TFHE scheme [Chillotti *et al.*, 2016] is the current state-of-the-art for the evaluation of arbitrary functions, and, in this work, we focus on improving its performance. We divide this paper into two parts. First, we review and implement the main techniques to improve performance or error behavior in TFHE proposed so far. For many, this is the first practical implementation. Then, we introduce novel improvements to several of them and new approaches to implement some commonly used procedures. We also show which proposals can be suitably combined to achieve better results. We provide a single library containing all the reviewed techniques as well as our original contributions. Our implementation is up to 1.2 times faster than previous ones with a similar optimization level, and our novel techniques provide speedups of up to 2.83 times on algorithms such as the Full-Domain Functional Bootstrap (FDFB).

CCS Concepts: • **Mathematics of computing** → *Mathematical software performance*; • **Security and privacy** → **Public key encryption**; Mathematical foundations of cryptography; Privacy-preserving protocols; Management and querying of encrypted data.

Additional Key Words and Phrases: Homomorphic Encryption, TFHE, Functional Bootstrap, Programmable Bootstrap, Efficient Implementation

ACM Reference Format:
Antonio Guimarães, Edson Borin, and Diego F. Aranha. 2022. MOSFHET: Optimized Software for FHE over the Torus. 1, 1 (April 2022), 21 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The idea of performing computation over encrypted data was a long-chased goal in the Cryptography community. The concept was first defined in 1978 [35], but for decades proposed solutions only achieved partial homomorphism. In 2009, Gentry [22] presented the first Fully Homomorphic Encryption (FHE) scheme, based on ideal lattices, enabling arbitrary computation through the evaluation of logic gates. Efficiency was a problem from the start, but Gentry's work also established a blueprint later used to build more efficient FHE schemes based on the Learning With Errors (LWE) problem [34] and its variants [7, 8]. Many of these follow-up works presented significant improvements efficiency-wise, but the literature generally evolved around the needs of specific uses cases, leaving behind, in terms of performance, capabilities such as the evaluation of arbitrary (nonlinear) functions.

Currently, one of the most efficient solutions for homomorphic evaluation is the CKKS cryptosystem [11], which was proposed aiming specifically at the homomorphic evaluation of neural network algorithms, a major use case for

This work was partially performed when the first author was visiting the Department of Computer Science at Aarhus University. It was supported by the São Paulo Research Foundation under grants 2013/08293-7, 2019/12783-6, and 2021/09849-5; by the National Council for Scientific and Technological Development under grant 313012/2017-2; and by the Independent Research Fund Denmark (DFF) project no. 1026-00350B.

Authors' addresses: Antonio Guimarães, antonio.guimaraes@ic.unicamp.br; Edson Borin, edson@ic.unicamp.br, Institute of Computing, University of Campinas, Campinas, Brazil; Diego F. Aranha, dfaranha@cs.au.dk, DIGIT Centre and Department of Computer Science, Aarhus University, Aarhus, Denmark.

2022. Manuscript submitted to ACM

FHE. These algorithms require a high throughput of linear arithmetic operations and are capable of correctly operating even with relatively large imprecisions [5]. Considering that, CKKS offers a very efficient homomorphic evaluation of approximate arithmetic in a SIMD-like manner. Its efficiency, however, restricts functionality, as the scheme needs to rely on arithmetic approximations for nonlinear functions. The cost of evaluating such approximations might grow exponentially with the desired precision [30], and trusting the arithmetic robustness of the overlying application is not always possible. In this way, the scheme requires extensive modifications for some applications and is unfit for many of them. For example, it is currently not possible to achieve state-of-the-art accuracy levels on deep neural networks without employing unrealistically large parameters [30].

Schemes implementing exact computing, on the other hand, usually represent applications as compositions of very basic logic components, such as binary logic gates, finite automata, and lookup tables. Translating an application to such components is a straightforward process and works broadly. However, large applications require a great number of logic components, and evaluating each may take significant amounts of time. The TFHE cryptosystem [14] is the current state-of-the-art for arbitrary exact (non-approximate) homomorphic evaluation. It was originally designed to evaluate binary logic gates, but newer versions also enable evaluating multi-bit gates [3] and lookup tables [15].

1.1 Contributions

There were several recent proposals for improving TFHE, but most of them are built upon various different implementations of the scheme, making it hard to address and evaluate their impact on the cryptosystem. Many also remained purely theoretical contributions, with no practical implementation until now. Considering this, our first goal in this work is to unify all these proposals in a single highly-optimized library. In this way, we can not only measure their impact considering the use of modern implementation techniques and algorithms but also evaluate how combinations of optimizations affect performance. Our library, **M**OSFHET (**O**ptimized **S**oftware for **FHE** over the **T**orus) [26], is fully portable and self-contained with optional optimizations for the Intel AVX2, FMA, and AVX-512 Instruction Set Extensions (ISEs). We designed it specifically for enabling the efficient prototyping of improvements to TFHE. In this first part, we implement the core functionalities of TFHE and the following techniques.

- The Functional [5] or Programmable [15] Bootstrap and its improved version [16].
- The Circuit Bootstrap [13] and its optimizations [16].
- The multi-value bootstrap [9, 16] and its optimizations [25].
- The Key Switching [12] and its optimizations [10].
- The BlindRotate Unfolding [40] and its optimizations [6].
- The Full TRGSW bootstrap [23].
- Three different approaches [16, 29, 38] for evaluating the Full-Domain Functional Bootstrap (FDFB).
- Public Key compression using randomness seed [14].
- BFV-like multiplication [16].

From these, we highlight that this work is the first to implement the Evaluation Key Compression for TFHE and to experimentally compare multiple approaches for the full-domain functional bootstrap. We should also note that techniques such as the multi-value bootstrap [9] and the Full TRGSW bootstrap are only available on unmaintained implementations of TFHE. For other techniques, we compare our results with other publicly available libraries, such as TFHEpp [31, 32], Concrete [39], and PALISADE [1].

When implementing these techniques, we found several opportunities for improvements on them as well as for combining them to yield better performance or error growth behavior. We also developed new methods to implement some commonly used procedures. As result, we present the following contributions:

- We introduce a new method to implement the Full-Domain Functional Bootstrap (FDFB) that is up to 2.83 times (AVX-512 version) faster than previous approaches while being the first to maintain the same output error variance as the regular (half-domain) Functional Bootstrap.
- We exploit key compression techniques so far used mostly for memory or storage optimizations to achieve speedups of up to 1.44 times on the execution time of core procedures, such as the Key Switching Algorithm.
- We generalize the BlindRotate Unfolding (as suggested in [6]) and show that it does not achieve the expected gains on large parameters.
- We found that previous works [9, 25] on the multi-value bootstrap fail to consider a corner case when estimating the output noise variance for some array decompositions. Moreover, we show how to reduce noise variance on this corner case with a negligible impact on performance.
- We implement basic polynomial arithmetic multiplication procedures that are significantly slower than the negacyclic FFT convolution, but that enable the implementation of techniques requiring precision higher than 53 bits (the hardware limit for floating-point representation). We also accelerate the negacyclic FFT convolution itself by implementating a version of the SPQLIOS FFT library [21] using AVX-512 instructions. This optimization provides up to 1.5 times speedup over the original (FMA-optimized) SPQLIOS.

The remainder of this text is organized as follows: Section 2 introduces the basic notation and concepts of TFHE; Section 3 presents the techniques implemented in our library and the improvements upon them; Section 4 presents the experimental results; finally, Section 5 concludes the paper.

2 FULLY HOMOMORPHIC ENCRYPTION OVER THE TORUS (TFHE)

TFHE [12–14] is a fully homomorphic encryption scheme based on the Learning With Errors (LWE) problem [34] and its ring variant [7]. In this section, we describe its algebraic structures as well as its basic functioning for homomorphically evaluating linear arithmetic and arbitrary functions. We use superscript to denote the number of elements in a vector and subscript to denote modulus. In this way, $\mathbb{S}_q^n$ is the set of vectors with n elements, each of them in some set $\mathbb{S}$ modulo q. We denote a set of polynomials over the variable X with coefficients in $\mathbb{S}$ by $\mathbb{S}[X]$. For power-of-two cyclotomic polynomials, we describe their modulo by the degree, therefore $\mathbb{S}_q[X]_N^n$ (or $\mathbb{S}'_N[X]^n$ for $\mathbb{S}' = \mathbb{S}_q$) is the set of vectors with n elements, where each element is a polynomial over the variable X with modulus $\Phi_{2N}(X) = X^N - 1$ with each coefficient belonging to the set $\mathbb{S}$ modulo q. Additionally, $\langle a, b \rangle$ denotes the inner product between vectors a and b, and $\lceil l \rfloor_t$ denotes the rounding of a number l to the closest multiple of t. If omitted, $t = 1$. Hereafter, we start by describing the LWE variant used in TFHE, where $\mathbb{M}$ is some $\mathfrak{R}$-module.

Definition 2.1 (Binary-Secret Scale-invariant LWE, from TFHE [14]). Let an LWE sample be a pair $(a, b) \in \mathbb{M}^{n+1}$, where a is uniformly sampled from $\mathbb{M}^n$, $b = \langle a, s \rangle + e \in \mathbb{M}$, and $n \geq 1 \in \mathbb{Z}$. The binary secret key s is sampled from a uniform distribution over some $\mathfrak{B}^n$ and the error e is sampled from a Gaussian distribution over $\mathbb{M}$ with mean 0 and standard deviation σ. Given a polynomial bounded number of LWE samples using the same s, we define two versions of the LWE problem:

- **Search problem:** Find s.

- **Decision problem:** Distinguish with non-negligible advantage the LWE samples from vectors uniformly sampled from $\mathbb{M}^{n+1}$.

Encryption scheme. The basic idea behind an LWE-based cryptosystem is to encrypt messages by adding them to the b component of the LWE sample since it is indistinguishable from a vector sampled from the uniform distribution (LWE decision problem). TFHE works with scalar and polynomials messages and encrypts them, respectively, in TLWE and TRLWE samples. They are both as described in Definition 2.1, differing by the definition of $\mathbb{M}$ and $\mathfrak{B}$:

- For **TLWE** samples, $\mathbb{M}$ is the real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$, which is the set of real numbers modulo 1, and the secret key s is sampled from $\mathbb{B}^n$, which is the set of n-sized arrays with elements in the binary set $\mathbb{B} = \{0, 1\}$. To encrypt a message, we map it to $\mathbb{T}$ and add to the b component of a fresh TLWE sample. We denote the set of TLWE samples encrypting the message $m \in \mathbb{T}$ with key $s \in \mathbb{B}^n$ and parameters $k = (n, \sigma)$ by $c \in \text{TLWE}_{s,k}(\text{m})$. We omit the parameters whenever the key is enough to specify them. For decrypting, we first use the secret key s to calculate the phase of a sample $phase(c) = b - \langle a, s \rangle$, which is the message plus the Gaussian error. At this point, we can either consider the phase as an approximated result or we can discretize the torus and round the phase to remove the error and get the exact message $m = \lceil phase(c) \rfloor_t$, where t is a discretization parameter.
- For **TRLWE** samples, $\mathbb{M}$ is $\mathbb{T}_N[X]$, which is the set of polynomials modulo the $2N$-th cyclotomic polynomial $(X^N - 1)$ with coefficients in the real torus $\mathbb{T}$. The secret key s is sampled from $\mathbb{B}_N[X]^n$, which is the set of n-sized arrays of polynomials modulo the $2N$-th cyclotomic polynomial $(X^N - 1)$ with coefficients in binary set $\mathbb{B} = \{0, 1\}$. Encryption and decryption are similar as described for TLWE samples. We denote the set of TRLWE samples encrypting the message $m \in \mathbb{T}_N[X]$ with key $s \in \mathbb{B}_N[X]^n$ and parameters $k = (n, N, \sigma)$ by $\text{TRLWE}_{s,k}(\text{m})$. Again, we omit the parameters whenever the key is enough to specify them.

Implementation-wise, TFHE represents torus elements as integers in $\mathbb{Z}_{2^p}$ using the map $\mathbb{T} \xrightarrow{\sim} \mathbb{Z}_{2^p}$ given by $x \mapsto x \cdot 2^p$, where p is the bit precision used in the implementation.

Evaluating Arithmetic. T(R)LWE samples are in an $\mathfrak{R}$-module. Therefore, we have well-defined additions between samples and multiplications with other rings. In both cases, operations are pair-wise: Let $c_i = (a_i, b_i) \in \text{T(R)LWE}(m_i)$ for $i \in \{0, 1\}$ be two T(R)LWE samples encrypting messages m_i. The sum of them is given by $c_{sum} = (a_0 + a_1, b_0 + b_1) \in \text{T(R)LWE}_s(m_1 + m_2)$ while $c_{scale} = (a_0 \cdot z, b_0 \cdot z) \in \text{T(R)LWE}_s(m_1 * z)$ encrypts the scaling by $z \in \mathfrak{R}$, where $\mathfrak{R}$ is a ring (typically, $\mathbb{Z}$ or $\mathbb{Z}_N[X]$).

T(R)LWE samples also support external products by *TRGSW samples*, which are sets of ℓn TRLWE samples. They are rarely used to encrypt messages but are necessary for creating evaluation keys, which are self-encryptions of the TLWE and TRLWE secret keys (the cryptosystem assumes circular security) necessary for providing fully homomorphic evaluation. Contrary to the T(R)LWE samples, the set of TRGSW samples is a ring and supports both additions and multiplications. We can also perform an external product between T(R)LWE and TRGSW samples. We denote the set of TRGSW samples encrypting the message $m \in \mathbb{Z}_N[X]$ with key $s \in \mathbb{B}_N[X]^n$ and parameters $k = (n, N, \sigma, \ell)$ by $\text{TRGSW}_{s,k}(\text{m})$. For the most part of this paper, we can consider TRGSW encryption and decryption as black-box algorithms.

As we scale, add, or multiply samples, the Gaussian error in the component b increases. For being a fully homomorphic encryption scheme, and therefore allowing for the evaluation of an unbound number of consecutive operations, we need to have tools for controlling the error growth. The bootstrap procedure, as first defined by Gentry [22], is a technique that allows resetting the error to a default value established by the parameter set.

2.1 Bootstrapping

In TFHE, the bootstrap can be used not only for resetting the error but also to implement arbitrary (nonlinear) functions. In its first version, TFHE's bootstrap was capable of evaluating any logic gates with two-bit inputs. For implementing it, it defines three main building blocks, which we describe in this section.

2.1.1 Public and Private Key Switching. The idea behind a key-switching algorithm is the homomorphic evaluation of the phase of a ciphertext. Let $c = (a, b) \in \text{T(R)LWE}_{s,k}(m)$ be a T(R)LWE sample encrypting m, the keyswitch algorithm uses an encryption of the secret key s, defined as $\text{KS}_i \in \text{T(R)LWE}_{s',k'}(s_i)$, to calculate the $phase(c) = b - \langle a, \text{KS}\rangle$. The result of this operation is $c' \in \text{T(R)LWE}_{s',k'}(m)$, allowing us, therefore, to switch keys and parameters. This process also allows the evaluation of linear morphisms, *i.e.*, any function f for which $phase(f(c)) = f(phase(c))$. We should note that, by this definition, f can be a linear combination of several T(R)LWE samples, which allows us, for example, to pack TLWE samples in TRLWE samples, a process called *Packing Key Switching*. Algorithm 1 shows the Public Key Switching algorithm from TFHE. We should note that a_i is decomposed before being multiplied by the encryption of s (line 4) so that the error variance growth, which would be quadratic on the value of a_i, is now significantly reduced.

Algorithm 1: Public Functional Key Switching (PublicKeySwitch) [14]

Input : p TLWE samples $c^{(z)} = (a^{(z)}, b^{(z)}) \in \text{TLWE}_s(\mu_z)$, $z \in [[1, p]]$
Input : a public R-Lipschitz linear function $f : \mathbb{T}^T \mapsto \mathbb{T}_N[X]$
Input : a precision parameter $t \in \mathbb{Z}$
Input : a Key Switching key $\text{KS}_{i,j} \in \text{T(R)LWE}_{s'}(\frac{s_i}{2^j})$, for $i \in [[1, n]]$ and $j \in [[1, t]]$
Output : a T(R)LWE sample $c' \in \text{T(R)LWE}_{s'}(f(\mu_z))$, for $z \in [[1, p]]$

1 **for** $i \in [[1, n]]$ **do**
2 $\quad a_i \leftarrow f(a_i^{(1)}, a_i^{(2)}, ..., a_i^{(p)})$
3 $\quad$ Let $\tilde{a}_i = \lceil a_i \rfloor_{\frac{1}{2^t}}$ be the closest multiple of $\frac{1}{2^t}$ to a_i
4 $\quad$ Decompose each $\tilde{a}_i = \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot 2^{-j}$, where $\tilde{a}_{i,j} \in \mathbb{B}_N[X]$
5 **Return** $(0, f(b_i^{(1)}, b_i^{(2)}, ..., b_i^{(p)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \tilde{a}_{i,j} \cdot \text{KS}_{i,j}$

The private version of the function bootstrap is quite similar to the public one, differing by the fact that the function f is embedded in the key switching key, *i.e.*, $KSK \in \text{T(R)LWE}_{s',k'}(f(s))$. This version is especially useful when f depends on secret information, such as the key, as occurs in Section 3.7. Algorithm 2 shows the private key switching from TFHE.

2.1.2 Blind Rotate. Given a TLWE sample $c = (a, b) \in \text{TLWE}_s(m)$ and a TRLWE sample $t \in \text{TRLWE}_{s'}(v)$, the BlindRotate procedure computes $t' = \text{TRLWE}_{s'}(tv \cdot X^{\lceil phase(c)2N \rfloor})$. Since this multiplication occurs modulo the $2N$-th cyclotomic polynomial, the operation works as a negacyclic rotation of the polynomial $t \in \mathbb{T}_N[X]$ by an amount defined by the phase of c (thus, a *blind rotation*). It is important to note that the phase contains the Gaussian error and it is scaled by $2N$. In addition to that, the rounding of each element of a also introduces an error.

2.1.3 Sample Extract. Given a TRLWE sample $c \in \text{TRLWE}_s(p = \sum_{i=0}^{N-1} m_i X^i)$, it extracts a TLWE sample encrypting a coefficient from the polynomial p, *i.e.*, $\text{SampleExtract}_j(c) \in TLWE_{s'}(m_j)$, where s' is the TLWE interpretation of s.

2.1.4 Gate Bootstrapping. Using the previously defined building blocks, we can now define the gate bootstrap algorithm, shown in Algorithm 4. We can summarize the idea behind this algorithm in three steps:

Algorithm 2: Private Functional Key Switching (PrivateKeySwitch) [14]

Input : p TLWE samples $c_k = (a_k, b_k) \in \text{TLWE}_s(\mu_k)$, $k \in [\![1, p]\!]$
Input : a precision parameter $t \in \mathbb{Z}$
Input : a Key Switching key $\text{KS}_{i,j,k} \in \text{T(R)LWE}_{s'}(\frac{f_k(s)_i}{2^j})$, for $i \in [\![1, n]\!]$, and $\text{KS}_{n+1,j,k} \in \text{T(R)LWE}_{s'}(\frac{f(-1)}{2^j})$, for $j \in [\![1, t]\!]$ and $k \in [\![1, p]\!]$, where f_k are linear morphisms
Output : a T(R)LWE sample $c_{out} \in \text{T(R)LWE}_{s'}(f(\mu_k))$, for $k \in [\![1, p]\!]$

for $k \in [\![1, p]\!]$ **do**
  for $i \in [\![1, n]\!]$ **do**
    Let $\tilde{a}_{k,i} = \lceil a_{k,i} \rfloor_{\frac{1}{2^t}}$ be the closest multiple of $\frac{1}{2^t}$ to $a_{k,i}$
    Decompose each $\tilde{a}_{k,i} = \sum_{j=1}^{t} \tilde{a}_{k,i,j} \cdot 2^{-j}$, where $\tilde{a}_{k,i,j} \in \mathbb{B}_N[X]$
Return $-\sum_{k=1}^{p} \sum_{i=1}^{n+1} \sum_{j=1}^{t} \tilde{a}_{k,i,j} \cdot \text{KS}_{k,i,j}$

Algorithm 3: BlindRotate Algorithm [14]

Input : a sample $c = (a_1, ..., a_n, b) \in \text{TLWE}_s(m)$
Input : a sample $tv \in \text{TRLWE}_S(m)$
Input : a list of samples $C_i \in \text{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output : a TRLWE sample of $c' \in \text{TRLWE}_S(X^{\lceil phase(c)2N \rfloor} \cdot tv)$

ACC $\leftarrow X^{-\lceil b2N \rfloor} \cdot tv$
for $i = 1$ ***to*** p **do**
  ACC $\leftarrow$ CMUX$(C_i, X^{\lceil a_i 2N \rfloor} \cdot \text{ACC}, \text{ACC})$
return ACC

Procedure CMUX(C, A, B)
  return $C \cdot (B - A) + B$

(1) Set the accumulator vector, ACC, to be $\sum_{i=0}^{N} \frac{1}{4} X^i \in \mathbb{T}_N[X]$
(2) Use BlindRotate to calculate ACC $\cdot X^{-\phi(c)2N} \mod \Phi_{2N}$
(3) Use SampleExtract to extract the constant term of rotated ACC.

In this logic gate version of TFHE, all messages should have values in the set $\{\frac{-1}{4}, \frac{+1}{4}\}$. However, the accumulation of errors and the arithmetic part of the logic gate implementation cause these values to change. The gate bootstrapping essentially rounds them back to the expected values. To exemplify, a NAND logic gate would be implemented as $\text{NAND}(a, b) = \text{GateBootstrap}((0, \frac{5}{8}) - a - b)$.

3 STATE-OF-THE-ART ON TFHE AND IMPROVEMENTS

In this section, we describe the main proposals presented so far for improving core algorithms or functionalities of TFHE. We also present our own novel ideas and methods. We should note that we do not include proposals made for other cryptosystems. Although many could be adapted from schemes such as FHEW [18], GSW [23], or even CKKS [11], we had to limit our efforts at some point. We also do not consider optimizations for building applications or high-level functions with TFHE, as these are usually more specialized use cases.

3.1 The Functional Bootstrap

First defined by Boura *et al.* [5] in 2019, the idea of a functional bootstrap is to evaluate a function within the bootstrap procedure either in addition to or instead of resetting the error. For TFHE, the functional bootstrap is a generalization

Algorithm 4: GateBootstrap algorithm [14]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(m)$
Input : a constant $\mu \in \mathbb{T}$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Output : $c' \in TLWE_S(m' \cdot \mu)$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$ and

$$m' = \begin{cases} 1, & \text{if } m < 0.5 \\ -1, & \text{otherwise} \end{cases}$$

1 $b \leftarrow \lceil 2Nb \rfloor$ and $a_i \leftarrow \lceil 2Na_i \rfloor \in \mathbb{Z}_{2N}$ **for each** $i \in [[1, n]]$
2 $v \leftarrow (1, X, X^2, ..., X^{N-1}) \cdot \mu \in \mathbb{T}_N[X]$
3 ACC $\leftarrow$ BlindRotate$((0, v), (a_1, ..., a_n, b), (\text{BK}_1, ..., \text{BK}_n))$
4 **return** SampleExtract$_0$(ACC)

of the gate bootstrapping for arbitrary functions discretized in Lookup Tables (LUTs). This notion, we should note, is not new to TFHE and was first introduced with the FHEW cryptosystem [18, 33]. Algorithm 5 shows the functional bootstrap of TFHE.

Algorithm 5: FunctionalBootstrap algorithm [5, 15, 25]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(\frac{m}{2B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Output : $c' \in \text{TLWE}_S(\frac{L[m]}{2B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$
1 $b \leftarrow \lceil 2Nb \rfloor$ and $a_i \leftarrow \lceil 2Na_i \rfloor \in \mathbb{Z}_{2N}$ **for each** $i \in [[1, n]]$
2 $tv \leftarrow \sum_{i=0}^{N-1} \frac{1}{2B} \cdot l_{\lfloor \frac{iB}{N} \rfloor} X^i \in \mathbb{T}_N[X]$
3 ACC $\leftarrow$ BlindRotate$((0, tv), (a_1, ..., a_n, b + \frac{1}{4B}), (\text{BK}_1, ..., \text{BK}_n))$
4 **return** SampleExtract$_0$(ACC)

The first step for evaluating the arbitrary function is to discretize its domain, evaluate it in all discretized points, and store the results in a lookup table (LUT). The LUT, then, needs to be encoded as a polynomial (line 2). Equation 1 details this process. The *Base B* is a discretization parameter.

$$L = [l_1 = f(1), l_2 = f(2), ..., l_B = f(B)] \mapsto \sum_{i=0}^{N/B-1} \frac{l_1}{2B} X^i + \sum_{i=N/B}^{2N/B-1} \frac{l_2}{2B} X^i + ... + \sum_{i=(B-1)N/B}^{N-1} \frac{l_B}{2B} X^i \tag{1}$$

Likewise, messages also need to be encoded in a way that they map to integers, which can be achieved by discretizing the torus. At this step, two parameters can define the discretization: numeric base (B) or precision. The first refers to the numeric base in which messages are decomposed when working with messages encrypted in several samples. The second is the number of bits in each TLWE sample or in each coefficient of the polynomials in TRLWE samples that are considered part of the message. Typically, $Base = 2^{precision-1}$, due to the *negacyclic property*, or $Base = 2^{precision}$, when working with full-domain functional bootstraps (Section 3.5).

The negacyclic property. The table lookup is performed by using the BlindRotate to multiply the test vector by $X^{-\phi(c)2N}$. This multiplication occurs modulo the $2N$-th cyclotomic polynomial and, therefore, presents a negacyclic

property, *i.e.*, let p be a polynomial, $p \cdot X^N = -p$. This property restricts the use of the functional bootstrap to anti-symmetric functions, *i.e.*, functions f such that $f(x+N) = -f(x)$. For arbitrary functions, we avoid the negacyclic property by using only the first half of the torus to encode messages.

Evaluating encrypted LUTs and private functions. Algorithm 5 receives a LUT represented as an array of integers in $\mathbb{Z}_B^B$, but it could receive directly the test vector (tv) polynomial (calculated in line 2) or even a TRLWE sample encrypting tv. This last case is especially useful for evaluating private functions, but the error variance of the encrypted LUT is added to the output error variance of the algorithm. This version can also be used to evaluate multi-variable functions, as we can use the Packing Key Switch to create LUTs from function inputs [25]. In this case, the output error variance is always greater than at least one of the function inputs, limiting the bootstrap's error-reducing capabilities.

3.2 The Improved Programmable Bootstrap

The programmable bootstrap, proposed in 2020 by Chillotti *et al.* [15], is essentially the same technique as the functional bootstrap but defined using a discretized notation of TFHE. Nonetheless, a subsequent improved version of this technique [16], proposed in 2021, introduced new parameters that allow for slicing and selecting just part of the input to evaluate the function over. Let $c = (a, b) \in \text{TLWE}(\frac{m}{2B})$ be a TLWE sample encrypting m and let $\tilde{m}$ be the binary vector representation of m, *i.e.* $m = \sum_{i=0}^{\lfloor \log_2 m \rfloor} 2^i \tilde{m}$. The improved version of the Programmable Bootstrap allows to evaluate $f(\sum_{k=0}^{j-i} 2^k \tilde{m}_{k+i})$, for any $i \leq j \in [[0, \lfloor \log_2 m \rfloor]]$. In this way, it makes it possible to decompose messages and bootstrap decomposed digits separately. This feature can be leveraged by methods that work over decomposed messages for enabling the evaluation of large lookup tables representing functions with high precision. We further discuss them in Section 3.6.

Algorithm 6 describes the improved version of the programmable bootstrap using the functional bootstrap of TFHE.

Algorithm 6: Improved Programmable Bootstrap (PBS) [16]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(\frac{m}{2B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : message slicing parameters κ and θ
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Output : $c'' \in \text{TLWE}_S(\frac{L[m]}{2B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$

1 $b' \leftarrow \lfloor b \cdot 2^\kappa \rceil_{\frac{1}{2^\theta}} \in \mathbb{T}$
2 $a'_i \leftarrow \lfloor a_i \cdot 2^\kappa \rceil_{\frac{1}{2^\theta}} \in \mathbb{T}$ **for each** $i \in [[1, n]]$
3 Let $c' = (a', b) \in \text{TLWE}_s(\lfloor \frac{m}{2B} \cdot 2^\kappa \rceil_{\frac{1}{2^\theta}})$
4 **return** FunctionalBootstrap(c', L, BK)

3.3 The Multi-Value Functional Bootstrap (MVFB)

Evaluating several different functions over the same input is a necessity not only for high-level applications but even for core procedures of the cryptosystem, such as the Circuit Bootstrap (Section 3.7) and the Tree-Based Functional Bootstrap (Section 3.6). The multi-value functional bootstrap is a technique that allows these evaluations to occur at a much smaller cost than executing several (single-value) functional bootstraps. The most straightforward solution for implementing the multi-value bootstrap would be using the BlindRotate to calculate just $c' \in \text{TRLWE}(X^{\lceil phase(c) 2N \rfloor})$

and, then, multiplying it by each LUT (encoded in polynomials). In 2019, Carpov *et al.* [9] proposed a better method based on decomposing the polynomials that encoded the LUTs to achieve a better error output. Algorithm 7 describes it.

Algorithm 7: Multi-Value Functional Bootstrap algorithm (MVFB) [9]

Input : a TLWE sample $c = (a = [a_1, a_2, ..., a_n], b) \in \text{TLWE}_s(\frac{m}{2N})$, $m \in \mathbb{Z}_{2N}$
Input : a scale factor τ
Input : q LUTs encoded in polynomials $TV_{F_i} \in \mathbb{Z}_N[X]$, for $i \in [[1, q]]$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Output : An array of TLWE samples $c'_i \in \text{TLWE}_S(\frac{F_i(m)}{2N})$ for $i = 1, ..., q$, where $S \in \mathbb{B}^N$ is a vector interpretation of $S \in \mathbb{B}_N[X]$

$b \leftarrow \lfloor 2Nb \rceil, a_i \leftarrow \lfloor 2Na_i \rceil \in \mathbb{Z}_{2N}$ **for each** $i \in [[1, n]]$
$v \leftarrow \frac{1}{2} \sum_{i=0}^{N-1} X^i \cdot \frac{1}{2N} \cdot \tau \in \mathbb{T}_N[X]$
$\text{ACC} \leftarrow \text{BlindRotate}((0, v), (a_1, ..., a_n, b), (\text{BK}_1, ..., \text{BK}_n))$
$c'_i = \text{SampleExtract}_0(\frac{TV_{F_i}}{v} \cdot \text{ACC})$, **for each** $i \in [[1, q]]$
return c'

This method is a significant improvement over the straightforward version, but it still introduces significantly more errors than the single-value counterpart. Carpov *et al.* [9] estimates the error output variance of their multi-value bootstrap as given by Equation 2, where σ_{FB} is the output error variance of the (single-value) functional bootstrap.

$$Var(Err(c)) \leq s(q-1)^2 \sigma_{FB} \tag{2}$$

In 2021, Guimarães *et al.* [25] improved the method by introducing a base composition with linear error growth, based on the scaling algorithm described in Algorithm 8. Equation 3 shows the final output error variance. Both works, however, start from the assumption that the square norm of the polynomial representing the LUT, $\|TV_f\|_2^2$, is smaller than $s(q-1)^2$, where s and q are, respectively, the input and output bases. This equation is not true in all cases. Let us take, for example, a 4-slot LUT with values $[1, 0, 1, 1]$, input base 4, and output base 2. The factorized version would be $[2, -1, 1, 0]$, for which the square norm is $2^2 + -1^2 + 1^2 = 6$, which should be smaller or equal than $s(q-1)^2 = 4(2-1)^2 = 4$. This is a corner case for their error estimations, which, in this work, we solve by applying the same scaling algorithm used in the base composition (Algorithm 8) to the multiplication by the first element of the factorized LUT. In our example, while the square norm is still $2^2 + -1^2 + 1^2 = 6$, the variance growth is linear on the first element, thus presenting a final growth of $2^1 + -1^2 + 1^2 = 4$.

$$Var(Err(c)) \leq s(q-1)\sigma_{FB} \tag{3}$$

Bootstrapping Many LUTs. In 2021, Chillotti *et al.* [16] presented a new method for the multi-value bootstrap. Different from the previous ones, their method does not incur additional errors nor affect performance. On the other hand, the number of LUTs evaluated in each bootstrap is limited by the cryptosystem parameters. Algorithm 9 describes the Bootstrap Many LUTs procedure.

3.4 Tensor product

As first defined, T(R)LWE samples cannot be directly multiplied by one another. However, there are several FHE schemes also based on the RLWE problem presenting tensorial multiplications between samples [11, 20]. In 2021,

Algorithm 8: Multiplication (scaling) using the multi-value extract (MultiValueExtractScaling) [25]

Input : a TRLWE sample $c \in \text{TRLWE}_S(\text{p})$, which is the accumulator (ACC) of a previous functional bootstrap, and a cleartext scalar $b \in \mathbb{Z}$

Output: a TLWE sample $c' \in \text{TLWE}_S(b \cdot p_0)$, where p_0 is the constant term of p, and $S \in \mathbb{B}^N$ is a vector interpretation of $S \in \mathbb{B}_N[X]$

$c' \leftarrow \text{TLWE}_S(0)$

$c' \leftarrow c' + \text{SampleExtract}_i(p)$, **for each** $i \in [[0, \lceil \frac{b}{2} \rceil - 1]]$

$c' \leftarrow c' - \text{SampleExtract}_i(p)$, **for each** $i \in [[N - \lfloor \frac{b}{2} \rfloor, N - 1]]$

Return c'

Algorithm 9: Bootstrap ManyLUT (BML) [16]

Input : a TLWE sample $c = (a = [a_1, a_2, ..., a_n], b) \in \text{TLWE}_s(\frac{m}{2N})$, $m \in \mathbb{Z}_{2N}$

Input : a set L of q lookup tables, each represented by an array $L_i \in \mathbb{T}^B$ encoding a function F_i, for $i \in [[0, q)$

Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$

Output: An array of TLWE samples $c'_i \in \text{TLWE}_S(F_i(m))$ for $i \in [[0, q)$, where $S \in \mathbb{B}^N$ is a vector interpretation of $S \in \mathbb{B}_N[X]$

$r \leftarrow \frac{N}{qB}$

$b \leftarrow \lfloor 2Nb \rceil, a_i \leftarrow \lfloor 2Na_i \rceil \in \mathbb{Z}_{2N}$ **for each** $i \in [[1, n]]$

$v \leftarrow \sum_{i=0}^{B-1} \sum_{j=0}^{q-1} \sum_{k=0}^{r-1} L_{j,i} X^{(iq+j)r+k}$

$\text{ACC} \leftarrow \text{BlindRotate}((0, v), (a_1, ..., a_n, b + \frac{1}{4Bq}), (\text{BK}_1, ..., \text{BK}_n))$

$c'_i = \text{SampleExtract}_{ir}(\text{ACC})$, **for each** $i \in [[0, q)$

return c'

Chillotti *et al.* [16] showed that it is possible to implement the BFV-like [20] tensor product using TFHE parameters. They also showed how it can be used to perform a multiplication between TLWE samples. While their work presents all the necessary error estimations and parameters, it remained a purely theoretical contribution without practical implementations so far. In this work, we implement the BFV-like tensor product in TFHE for TRLWE samples with $n = 1$ (following the TRLWE definition of Chillotti *et al.* [16]). Algorithm 10 describes the tensor product, and Algorithm 11 shows the multiplication between TLWE samples.

Algorithm 10: TRLWE tensor product (TensorProd) [16]

Input : two TRLWE samples $c_i = (a_i, b_i) \in \text{TRLWE}_s(\frac{p_i}{B})$, for $p_i \in \mathbb{Z}_N[X]$ and $i \in \{0, 1\}$

Input : a relinearization key $\text{RLK}_i \in \text{TRGSW}_s(\frac{s^2}{\mathfrak{B}^j})$, for $j \in [[1, t]]$

Input : an integer precision parameter $q \in \mathbb{N}$

Output: $c' \in TRLWE_s(\frac{p_0 \cdot p_1}{B})$

$Q \leftarrow q^2, \Delta \leftarrow \log_2(q/B)$

$A_i \leftarrow qa_i, B_i \leftarrow qb_i$, for $i \in \{0, 1\}$

$T \leftarrow \lfloor \frac{\lfloor A_1 \cdot A_2 \rceil_Q}{\Delta} \rceil_q$

$A' \leftarrow \lfloor \frac{\lfloor A_1 \cdot B_2 + B_1 \cdot A_2 \rceil_Q}{\Delta} \rceil_q$

$B' \leftarrow \lfloor \frac{\lfloor B_1 \cdot B_2 \rceil_Q}{\Delta} \rceil_q$

Decompose T/q, such that $T/q = \sum_{j=1}^{t} T'_j \cdot \mathfrak{B}^{-j}$

return $(A'/q, B'/q) + \sum_{i=1}^{t} T'_i \cdot \text{RLK}_i$

Algorithm 11: TLWE multiplication (TLWEMult) [16]

Input : two TLWE samples $c_i = (a_i, b_i) \in \text{TLWE}_s(\frac{m_i}{B})$, for $m_i \in \mathbb{Z}_B$ and $i \in \{0, 1\}$
Input : a relinearization key $\text{RLK}_i \in \text{TRGSW}_s(\frac{s^2}{\mathfrak{B}^j})$, for $j \in [\![1, t]\!]$
Input : an integer precision parameter $q \in \mathbb{N}$
Input : a Key Switching key $\text{KSK}_{i,j} \in \text{T(R)LWE}_S(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$
Output : $c' \in TLWE_s(\frac{m_0 \times m_1}{B})$

$f : \mathbb{T} \mapsto \mathbb{T}_N[X] = m \mapsto mX^0$
$C_0 \leftarrow \text{PublicKeySwitch}(c_0, f, KSK)$
$C_1 \leftarrow \text{PublicKeySwitch}(c_1, f, KSK)$
$C_{mul} \leftarrow \text{TensorProd}(C_0, C_1, RLK, q)$
return $\text{SampleExtract}_0(C_{mul})$

Integer precision. In Algorithm 10, we use an integer precision parameter $q \in \mathbb{N}$ to map (by scaling) elements from the torus to $\mathbb{Z}_q$. This is not necessary for the definition of the algorithm, but it shows the required precision of the underlying polynomial arithmetic implementation. Typically, $q = 2^{32}$ or $q = 2^{64}$. In the first case, multiplications can be performed directly using the FFT without adding significant error. In the latter, the multiplication might require up to 128 bits of precision depending on the input base B. Considering that, in this work, we also implement a version of the TRLWE tensor product using 128-bit polynomial multiplication based on the Karatsuba algorithm [27].

3.5 Full-Domain Functional Bootstrap (FDFB)

The functional bootstrap is capable of evaluating arbitrary functions only if the input is in the first half of the torus, due to the negacyclic property (Section 3.1). Thus, it is a *half-domain functional bootstrap* (HDFB). It also is not able to perform modular (cyclic) arithmetic. The full-domain functional bootstrap (FDFB) is a variant that overcomes such restrictions and operates over the entire input domain following modular cyclic arithmetic. There are several techniques for implementing it [16, 17, 29, 38], and, in general, they evaluate an arbitrary function f by decomposing it into multiple sub-functions f_i and evaluating each f_i with an HDFB. In this work, we implement all solutions that are not purely based on high-level function pre-processing. Specifically, we implement all that require modifications to or introduce new building blocks to the cryptosystem. The following sections discuss them.

3.5.1 The Tensor Product method. Chillotti *et al.* [16] were the first to present a full-domain functional bootstrap for TFHE or, as they defined, a without-padding programmable bootstrap (WoP-PBS). Algorithm 12 shows their technique, proposed in 2021.

3.5.2 The PubMux Method. In 2021, Kluczniak and Schild [29] proposed a technique for the FDFB based on the definition of a public version of TFHE's C multiplexer (CMUX, Algorithm 4). In this version, the inputs are polynomials (instead of T(R)LWE samples), and the selector is a TLWE sample (instead of a TRGSW sample). Algorithm 13 presents their technique. It first calculates the input sign, then uses it to select, using the PubMux method, between LUTs encoding the subfunctions $f_0 = f$ and $f_1 = -f$. The result is used as a *test vector* for a regular functional bootstrap using the same input.

3.5.3 The Chaining Method. The FDFB presented by Chillotti *et al.* [16] is the state-of-the-art on performance, requiring just one multi-value bootstrap. However, it still introduces more errors than the original FB, as it selects between the bootstrap lookup results using the TRLWE tensor product. In this section, we introduce a novel method for performing

Algorithm 12: Full-Domain Functional Bootstrap based on TLWEMult (FDFB-CLOT21) [16]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(\frac{m}{B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$.
Input : a relinearization key $\text{RLK}_i \in \text{TRGSW}_s(\frac{s^j}{\mathfrak{B}^j})$, for $j \in [\![1, t]\!]$
Input : an integer precision parameter $q \in \mathbb{N}$
Input : a key switching key $\text{KSK}_{i,j} \in \text{T(R)LWE}_S(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$
Output : $c' \in TLWE_S(\frac{L[m]}{2B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$

1 $c_a \leftarrow$ FunctionalBootstrap$(c, L[0 : \frac{B}{2}], BK)$
2 $c_b \leftarrow$ FunctionalBootstrap$(c, L[\frac{B}{2} : B], BK)$
3 $c_{sign} \leftarrow$ FunctionalBootstrap$(c, [\frac{1}{2B}, ..., \frac{1}{2B}], BK)$
4 $c_{as} \leftarrow$ TLWEMult$(c_a, c_{sign} + (0, \frac{1}{2B}), q, \text{RLK}, \text{KSK})$
5 $c_{bs} \leftarrow$ TLWEMult$(c_b, c_{sign} - (0, \frac{1}{2B}), q, \text{RLK}, \text{KSK})$
6 **return** $c_{as} + c_{bs}$

Algorithm 13: Full-Domain Functional Bootstrap based on PubMux (FDFB-KS21) [29]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(\frac{m}{B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Input : precision parameters $\ell, \mathfrak{B} \in \mathbb{N}$
Input : a key switching key $\text{KSK}_{i,j} \in \text{T(R)LWE}_S(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$.
Output : $c' \in TLWE_S(\frac{L[m]}{B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$

1 $p_1 \leftarrow \sum_{i=0}^{\frac{B}{2}-1} \sum_{j=0}^{\frac{2N}{B}} \frac{l_i}{B} X^{\frac{2Ni}{B}+j}$
2 $p_2 \leftarrow \sum_{i=0}^{\frac{B}{2}-1} \sum_{j=0}^{\frac{2N}{B}} -\frac{l_{i+\frac{B}{2}}}{B} X^{\frac{2Ni}{B}+j}$
3 $c_{sign,i} \leftarrow$ FunctionalBootstrap$(c, [\frac{1}{2\mathfrak{B}^i}, ..., \frac{1}{2\mathfrak{B}^i}], BK) - \frac{1}{2\mathfrak{B}^i}$, for $i \in [\![0, \ell)$
4 $tv \leftarrow$ PubMux(c_{sign}, p_1, p_2)
5 **return** FunctionalBootstrap(c, tv, BK)

1 **Procedure** PubMux(C, A, B)
2 $BA \leftarrow B - A$
3 Let BA' be the decomposition of BA in base $\mathfrak{B}$, s.t. $BA = \sum_{i=0}^{\ell-1} BA'_i \cdot \mathfrak{B}^{-i}$
4 **return** $B + \sum_{i=0}^{\ell-1} C_i \cdot BA'_i$

the full-domain functional bootstrap that provides the same error variance output as the basic (half-domain) FB. Algorithm 14 describes it. Despite requiring two functional bootstraps, the algorithm combines them using the chaining method [25], which provides the lowest output error variance. We note that this method can be seen as an extension of the *FullFBS* presented by Yang *et al.* [38] for their cryptosystem (TOTA), although their method only removes the negacyclicity, without addressing full-domain evaluation specifically. One can obtain the original technique from Yang *et al.* [38] by replacing line 1 of Algorithm 14 with line 2 of Algorithm 5.

3.6 Evaluating large Lookup tables

All methods and variations of the functional bootstrap we presented so far have a common limitation: The message is encrypted in a single sample, and the size of the LUT is limited by the parameters of the cryptosystem. In practice, it is

Algorithm 14: Full-Domain Functional Bootstrap based on Chaining (FDFB-C)

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(\frac{m}{B})$, for $m \in Z_B$
Input : an integer LUT $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Output : $c' \in TLWE_S(\frac{L[m]}{B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$

$tv \leftarrow \sum_{i=0}^{\frac{B}{2}-1} \sum_{j=0}^{1} \sum_{k=0}^{\frac{N}{B}-1} \frac{1}{B} l_{\frac{jB}{2}+i} X^{(2i+j)\frac{N}{B}+k}$
$c_{sign} \leftarrow \text{FunctionalBootstrap}(c, [\frac{B+1}{4B}, ..., \frac{B+1}{4B}], BK) - \frac{B+1}{4B}$
return $\text{FunctionalBootstrap}(c + c_{sign}, tv, BK)$

not possible to efficiently evaluate functions with more than 6 bits of precision with these methods [16]. To evaluate large lookup tables, it is necessary to decompose the message into several ciphertexts and combine the evaluation of several small LUTs over the decomposed digits. In 2021, Guimarães *et al.* [25] introduced two methods for evaluating large LUTs. Algorithm 15 describes the tree-based functional bootstrap.

Algorithm 15: Tree-based functional bootstrap (TreeFB) [25]

Input : a set of TLWE samples $c_i \in \text{TLWE}_s(\frac{m_i}{2B})$, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the integer m in base B with d digits
Input : a set L of B^d polynomials $\in \mathbb{Z}_N[X]$ encoding the lookup table of an arbitrary function F
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_S(s_i)$, for $i \in [[1, n]]$
Input : a Key Switching key $\text{KS}_{i,j} \in \text{T(R)LWE}_s(\frac{s_i}{2^j})$, for $i \in [[1, n]]$ and $j \in [[1, t]]$
Output : A TLWE sample $c' \in \text{TLWE}_S(\frac{F(m)}{2B})$, where $S \in \mathbb{B}^N$ is a vector (TLWE) interpretation of $S \in \mathbb{B}_N[X]$

TV $\leftarrow$ L
$f : \mathbb{T}^B \mapsto \mathbb{T}_N[X] = (a_1, ..., a_B) \mapsto a_1 X^{N-1} + \ ... \ + a_B$
for $i \leftarrow 0$ **to** $d - 1$ **do**
  $c' \leftarrow \text{MVFB}(c_i, \text{TV}, \text{BK})$
  for $j \leftarrow 1$ **to** B^{d-i-2} **do**
    $\text{TV}_{j-1} \leftarrow \text{PublicKeySwitch}((c'_{(j-1)\times B}, ..., c'_{j\times B}), f, \text{KS})$
return c'_0

They also introduced a *chaining method* (ChainingFB) for combining multiple functional bootstraps, which is more intricate to implement but provides better error output variance. Its implementation is specific to each function. In summary, the method boils down to using linear combinations of an FB output as the selector for the next. Algorithm 14 exemplifies it. Guimarães *et al.* [25] remarks that, although more functionally restricted, the method is especially good for evaluating functions with carry-like or test logics. In 2022, Clet *et al.* [17] showed that the method is capable of evaluating any function by using a digit composition as linear combination, *i.e.*, $(a_0, a_1) \mapsto a_0 + a_1 \cdot B$, where B is the numeric base. This composition, however, requires quadratically larger parameters, and it is still unclear whether it would improve the evaluation of arbitrary high-level functions.

3.7 The Circuit Bootstrap

Working with T(R)LWE samples is usually the norm in TFHE, as computation is cheaper both for arithmetic and FB-based arbitrary function evaluation. However, several techniques require samples to be encrypted as TRGSW [14] samples. In this context, the *Circuit Bootstrap*, first proposed by Chillotti *et al.* in 2016, is a technique for producing

a TRGSW sample from a TRLWE one. Since it is based on the functional bootstrap, the content of the fresh sample can be arbitrarily defined by a function. Algorithm 16 defines the circuit bootstrap based on the functional bootstrap. Since it requires the evaluation of several functions over the same input, we can use the BML algorithm, presented in Section 3.3, to accelerate the computation (as suggested by Chillotti *et al.* [16]) at the cost of a slightly increased error rate.

Algorithm 16: CircuitBootstrap algorithm [14]

Input : a TLWE sample $c = (a, b) \in \text{TLWE}_s(m)$ for $m \in \{0, \frac{1}{4}\}$
Input : a constant $\mu \in \mathbb{T}$
Input : a bootstrapping key $\text{BK}_i \in \text{TRGSW}_{s'}(s_i)$, for $i \in [\![1, n]\!]$
Input : a Private Key Switching key $\text{KSA}_{i,j} \in \text{T(R)LWE}_{s'}(\frac{f(s)_i}{2^j})$, for $i \in [\![1, n]\!]$, and $\text{KS}_{n+1,j} \in \text{T(R)LWE}_{s'}(\frac{f(-1)}{2^j})$, for $j \in [\![1, t]\!]$ and $f : \mathbb{B} \mapsto \mathbb{B}_N[X] = (s) \mapsto s \cdot -s'$
Input : a Key Switching key $\text{KSB}_{i,j} \in \text{T(R)LWE}_{s'}(\frac{s_i}{2^j})$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$
Output : $c' \in \text{TRGSW}_{s',(\ell,\mathfrak{B})}(m' \cdot \mu)$, where $m' = \begin{cases} 1, & \text{if } m \in [\![0.25, 0.5) \\ 0, & \text{otherwise} \end{cases}$

$f : \mathbb{T} \mapsto \mathbb{T}_N[X] = m \mapsto m \cdot X^0$
for $i \leftarrow 1$ **to** ℓ **do**
  $\tilde{c} \leftarrow \text{FunctionalBootstrap}(c, [0, \mathfrak{B}^{-i}], BK)$
  $c'_i \leftarrow \text{PrivateKeySwitch}(\tilde{c}, \text{KSKA})$
  $c'_{\ell+i} \leftarrow \text{PublicKeySwitch}(\tilde{c}, f, \text{KSKB})$
return c'

3.8 The Full TRGSW bootstrap

The TreeFB algorithm (Section 3.6) supposes the use of the multi-value functional bootstrap for every level of the tree to achieve a linear number of bootstraps. However, after the first (base) level of the tree, LUTs are encrypted in TRLWE samples (instead of encoded in clear-text polynomials). Carpov *et al.* [9] MVFB (Algorithm 7) does not operate over encrypted test vectors and Chillotti *et al.* [16] BML (Algorithm 9) supports a limited number of LUTs. Guimarães *et al.* [25] suggests using the CircuitBootstrap to employ the MVFB over encrypted LUTs, but they did not implement the technique as it would require an implementation supporting 64-bit torus precision. Our library not only provides this precision level but also all optimizations for the CircuitBootstrap discussed by them [25]. We note, however, that instead of executing the regular FB plus a CircuitBootstrap, we can just directly perform a *full TRGSW bootstrap*, which uses the same number of BlindRotate executions as the CircuitBootstrap, but saves time by avoiding key switchings.

The full TRGSW bootstrap is similar to the functional bootstrap described in Algorithm 5, but the accumulator vector is a TRGSW sample instead of a TRLWE sample. In this way, the external products become internal products between TRGSW samples, which are at least ℓ times more expensive but have the same output error variance. The result produced by the algorithm, encrypting $X^{\lceil phase(c)2N \rfloor}$, is also a TRGSW sample and can, therefore, be multiplied by the decomposed LUTs in Carpov *et al.* method, even when they are encrypted in TRLWE samples. Different from the original MVFB, the output error variance of such multiplication depends on the square norm of the TRGSW samples, and not on the LUT. In this way, Carpov *et al.* decomposition presents no advantage anymore, and we can use the straightforward version of the multi-value bootstrap described at the beginning of Section 3.3.

3.9 T(R)LWE conversion

The Key Switching algorithm is one of the core procedures of TFHE, and its performance degrades rather fastly for large parameters when inputs are TLWE samples. For TRLWE samples, on the other hand, the Key Switching can be sped up by using the FFT to perform multiplications, which comes at the cost of an increased error rate. In 2021, Chen *et al.* [10] presented several algorithms that allow performing TLWE Key Switching using TRLWE Key Switching methods. For TLWE-to-TRLWE conversion, however, their algorithm multiplies the coefficients of the result by N (the modulo polynomial degree) as a side effect, since it is based on the Galois permutation. In the standard instantiation of TFHE, coefficients are in the real torus and N does not have an inverse. In this way, we implement their algorithms for completeness, but we did not find many cases in which it could be used efficiently. Specifically, it is possible to use such algorithms in cases in which the message can be divided by N using bootstraps or before encryption.

3.10 The BlindRotate Unfolding

The BlindRotate is the most expensive operation in TFHE's bootstrap. It calculates $X^{\sum_{i=1}^{n} s_i a_i}$. Its most expensive operations, in turn, are the multiplications by TRGSW samples, which encrypt s_i. In 2018, Zhou *et al.* [40] showed how to reduce the number of multiplications by unfolding the BlindRotate loop. Equation 4 shows their proposal. In the same year, Bourse *et al.* [6] improved the unfolding equation by calculating the last term from the first three. They also suggest the equation could be generalized to large unfoldings. In this work, we implemented this generalization and tested unfoldings of sizes 2, 4, and 8.

$$X^{as+a's'} = ss'X^{a+a'} + s(1-s')X^{a} + (1-s)s'X^{a'} + (1-s)(1-s'). \tag{4}$$

3.11 Public Key compression

The bootstrap operation and, to a lesser degree, the key switching algorithm are the most time-consuming procedures in TFHE. Both of them, however, can be sped up at the cost of larger keys. Specifically, one can increase the decomposition base of the key switching and the BlindRotate unfolding in the bootstrap. In both cases, it is possible to achieve linear gains on performance with exponential growth on the key size. Techniques for compressing evaluation keys are broadly available in the literature. For TFHE, Chillotti *et al.* [14] suggest storing just the pseudo-random number generator (PRNG) seed used to generate the a component of TRLWE samples and only generating a when necessary. This technique gives up to n times storage gain, but so far has not been implemented for TFHE. In this work, we not only implement the idea but also show how we can use it to improve execution time in the key switching algorithm.

Algorithm 17 shows the core TRLWE subtraction algorithm used in the key switching. We could use any PRNG to implement it, but SHAKE256 [19] was a convenient choice as we were already using it for the rest of the implementation, and it is a cryptographically secure PRNG. This version provides almost two times storage and memory usage reduction for TRLWE key switching keys and bootstrap keys. However, it slows down the execution by more than 10 times. We could minimize the impact of this slowdown by expanding the entire keys at loading time, but we would lose the memory usage gains, which are one of the most important benefits of this technique.

To solve this problem, we implement the key switching as shown in Algorithm 18. There are two main changes to note in this version:

(1) We replace SHAKE256 by Xoroshiro [2, 37], a much faster PRNG, but that is not considered cryptographically secure. There are several examples of using such generators for generating public information, as is the case of

Algorithm 17: TRLWE subtraction

Input : a compressed TRLWE sample $c_0 = (seed_{a_0}, b_0) \in \text{TRLWE}_s(\frac{p_0}{B})$, for $p_0 \in \mathbb{Z}_N[X]$
Input : a TRLWE sample $c_1 = (a_1, b_1) \in \text{TRLWE}_s(\frac{p_1}{B})$, for $p_1 \in \mathbb{Z}_N[X]$
Output : $c' = (a', b') \in \text{TRLWE}_s(\frac{p_0 - p_1}{B})$

$a_0 \leftarrow \text{SHAKE256}(seed_{a_0}, N)$
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $a'_i = a_{0,i} - a_{1,i}$
  $b'_i = b_{0,i} - b_{1,i}$
return c'

the a component of (R)LWE samples. For the security aspects of using Xoroshiro for generating a, we refer to previous literature [2, 4, 24]. If a secure PRNG is required even for public parameters, viable alternatives may be found in Lightweight Cryptography [36].

(2) We interleave the memory load of the b component with the expansion computation of the PRNG. In this way, we take advantage of instruction-level parallelism since CPU (a calculation) and memory (b loading) intensive code portions are executed simultaneously by the processor.

Algorithm 18: TRLWE subtraction

Input : a compressed TRLWE sample $c_0 = (seed_{a_0}, b_0) \in \text{TRLWE}_s(\frac{p_0}{B})$, for $p_0 \in \mathbb{Z}_N[X]$
Input : a TRLWE sample $c_1 = (a_1, b_1) \in \text{TRLWE}_s(\frac{p_1}{B})$, for $p_1 \in \mathbb{Z}_N[X]$
Output : $c' = (a', b') \in TRLWE_s(\frac{p_0 - p_1}{B})$

$state \leftarrow seed_{a_i}$
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $a'_i = Xoroshiro128pp\_next(state) - a_{1,i}$
  $b'_i = b_{0,i} - b_{1,i}$
return c'

At the implementation level, it was also necessary to vectorize Xoroshiro's code using AVX2 instructions. Ultimately, it was necessary to use a highly optimized version of an already very fast generator to have gains over the non-compressed version, but we were able to achieve speedups of up to 1.44 times. The vectorized version of Xoroshiro is a side contribution of this work.

3.12 State-of-the-art summary

Table 1 summarizes the techniques presented in this section as well as the improvements we presented for them. Besides the improvements listed in the table, we note that one of our main contributions in this work is to implement all the techniques in a single highly optimized software library.

4 EXPERIMENTAL RESULTS

We implement all algorithms presented in Section 3 in a single C library. The code is fully portable, self-contained, and includes optional optimizations for the Intel AVX2 Instruction Set Extension (ISE). In this section, we compare the execution times with implementations from TFHEpp [31], Concrete [15], and PALISADE [1, 29]. We use the parameter set defined by TFHEpp [31] and reproduced in Table 2.

Table 1. Summary of the techniques presented in Section 3 and our contribution to each

Procedures	Literature	Section	Improvements in this work
Functional Bootstrap	[5, 18]	3.1	-
Improved Programmable Bootstrap	[16]	3.2	-
Circuit Bootstrap	[14]	3.7	Accelerated using the BLM (as suggested in [16])
Multi-value Bootstrap	[9, 25]	3.3	We modified the composition algorithm to treat a corner case on error growth.
	[16]		-
Key Switching	[10, 14]	2.1.1, 3.9	Accelerated using public key compression
BlindRotate Unfolding	[6, 40]	3.10	Method generalized for large unfoldings (as suggested in [6])
TRGSW Bootstrap	[14]	3.8	-
TreeFB	[25]	3.6	We use the TRGSW bootstrap to provide multi-value bootstrap for all levels of the tree.
ChainingFB	[25]		-
Full-Domain Functional Bootstrap	[16]	3.5.1	Accelerated using the BLM (as suggested in [16])
	[29]	3.5.2	Accelerated using the BLM
	This work	3.5.3	New technique
Public key compression	[14]	3.11	We show how to exploit lightweight PRNG to improve performance
BFV-like multiplication	[16]	3.4	-

Table 2. Parameters from TFHEpp [31]

λ	TLWE		TRLWE			TRGSW		Key Switch	
	n	σ	n	N	σ	ℓ	$log_2(base)$	t	$log_2(base)$
127	632	2^{-15}	1	2048	2^{-44}	4	9	8	4

4.1 Execution Time

Table 3 shows the execution times of our algorithms, in microseconds, and compare with the TFHEpp library [31]. We executed all experiments on a bare metal instance on AWS public cloud (`m5zn.metal`) featuring an Intel Xeon Platinum 8252C CPU at 4.5GHz with 192GB of RAM running Ubuntu 20.04.4 LTS. We note that the AVX-512 ISE reduces the maximum processor frequency and might sometimes impact performance negatively compared with code using FMA instructions. We compiled both implementations using GCC 10.3.0 with similar optimization flags[1]. Both our library and TFHEpp (by default) use the same FFT library for fast polynomial arithmetics: the SPQLIOS [21] library with Intel FMA ISE optimizations[2]. In this work, we also developed (as a side contribution) a version of SPQLIOS optimized using AVX-512 instructions. For providing software portability, our library includes the FFNT library [28]. Each measurement in the table is the average of 1000 executions. We also measured standard deviation and calculated a 99% confidence interval for all metrics in our implementation. We omit them from the table as they are all negligible (smaller than 1% of the average).

For most procedures, we obtain speedups varying from 1.21 to 1.4 times over TFHEpp. The AVX-512 version offers a further speedup of up to 1.5 times over the FMA version, being up to 1.6 times faster than TFHEpp. The only case

[1]The complete compilation commands are available in the code repositories [26, 31]
[2]SPQLIOS was presented by Nicolas Gama *et al.* [21] with TFHE [12]. It was adapted by TFHEpp for their C++ code, which we adapted to pure C.

Table 3. Execution time, in microseconds, for each procedure. The speedup considers the FMA version only, as TFHEpp does not include AVX-512 acceleration.

Algorithm	Section	This work		TFHEpp	Speedup
		FMA	AVX-512	FMA	(FMA version)
Functional Bootstrap	3.1	39,710	29,848	47,942	1.21
Full TRGSW Bootstrap Setup:	3.8	274,178	183,298	-	-
Full TRGSW Bootstrap (cost per LUT):		61	45	-	-
MVFB setup	3.3	39,508	29,750	-	-
MVFB (cost per LUT):		12	11	-	-
TRLWE Key Switching	2.1.1	46,079	45,644	39,191	0.85
TRLWE Key Switching using Compression	3.11	31,871	34,196	-	1.23
TLWE Key Switching	2.1.1	8,880	9,240	2,802	0.32
TRLWE Key Switching CDKS21	3.9	559	557	665	1.19
Circuit Bootstrap	3.7	416,258	396,560	-	-
Circuit Bootstrap using BML		295,303	305,571	412,547	1.40
128-bit Tensor Product using Karatsuba:	3.4	12,867	12,910	-	-
64-bit Tensor Product using FFT:		91	88	-	-
32-bit Tensor Product using FFT:		-	-	30	-
Full Domain Functional Bootstrap KS21:	3.5.2	327,041	288,261	-	-
Full Domain Functional Bootstrap KS21 with BML:		207,804	197,710	-	-
Full Domain Functional Bootstrap CLOT21:	3.5.1	248,320	227,645	-	-
Full Domain Functional Bootstrap CLOT21 with BML:		167,956	167,427	-	-
Full Domain Functional Bootstrap (this work):	3.5.3	89,076	69,707	-	-
Functional Bootstrap Unfold=2	3.10	70,200	58,319	-	-
Functional Bootstrap Unfold=4		87,563	77,675	-	-
Functional Bootstrap Unfold=8		506,666	492,330	-	-

we observe slowdowns are in the TLWE KeySwitch, which they implement over a 32-bit version of the torus. Our implementation only uses 64-bit representations of the torus, as they are required by several algorithms.

Comparing the technique among themselves, we highlight the up to 1.44 times speedup of the optimized version of the key switching algorithms, achieved thanks to the evaluation key compression, which also reduces the key size by a factor of 2. In the AVX-512 version, our FDFB method is 2.83 times faster than the FDFB-KS21 method and 2.40 times faster than the FDFB based on TLWE multiplication (FDFB-CLOT21), despite requiring more bootstraps. This is explained by the cost of the TRLWE key switching procedures used in FDFB-CLOT21 (which uses Algorithm 11).

At first, we would also expect the have linear gains as we increase (exponentially) the unfolding. However, we observed significant slowdowns likely due to the increased size of keys. Although we tried to further optimize our unfolding implementation by specializing it in a specific unfolding, our best result was still slightly worse than without using the technique. In this way, we chose to keep and report just the generic algorithm, which still provides a trade-off on rounding errors and error variances and should bring gains for smaller parameters, as shown by Bourse *et al.* [6].

Comparison against other libraries. TFHEpp is the only library to cover many of the techniques we consider and present a similar level of optimization. Nonetheless, we also executed the Functional Bootstrap of Concrete and the FDFB of Kluczniak and Schild [29] in our environment. The former took 2.9 seconds to run on the same parameters, being 74 times slower than our library. The latter was built over the PALISADE library [1], and we could not test for the exact same parameter set as it instantiates TFHE mapping the torus to prime fields (which is common on other (R)LWE

cryptosystems but not on TFHE). Using the default parameters provided by the authors, the functional bootstrap took 2.7 seconds to execute (68 times slower than ours), and the full-domain functional bootstrap took 21.3 seconds, a 65 times slowdown compared to our non-optimized implementation of the same algorithm and a 102 times slowdown compared to the optimized version.

5 CONCLUSION

In this work, we reviewed and implemented the main techniques presented so far for improving execution time or error behavior in the homomorphic evaluation using the TFHE cryptosystem. We showed which proposals could be efficiently combined and introduced several novel contributions. Our implementation achieved speedups of up to 1.2 over previous ones with a similar optimization level, and our new methods achieved speedups of up to 2.83 times over previously employed techniques. We also presented, as side contributions, versions of Xoroshiro and SPQLIOS vectorized with AVX2 and AVX-512 ISEs, respectively.

One of our major goals in this work was to present a software platform over which contributions and improvements to TFHE could be easily developed and tested in efficient ways. In this context, we introduced MOSFHET, which is a fully portable and self-contained C-library. Thanks to optional optimizations using AVX2 instructions, it also offers efficient performance on the most commonly used CPUs. Compared to the implementation of Kluczniak and Schild [29], for example, we achieved speedups of up to 102 times. Their implementation is built upon PALISADE [1], a library that is broadly used for prototyping FHE schemes, but that is not specific to TFHE. In this way, it is not only far from offering a competitive performance level but also does not provide easy access to the state-of-the-art techniques available for the scheme. Our library comes to fulfill these two necessities. It implements all the newest techniques proposed in the literature and provides an environment for efficiently prototyping new contributions.

As future work, we intend to continuously update the library to follow the state-of-art on TFHE; seek better options for arithmetic with more than 64 bits of precision; adapt techniques from other cryptosystems that might seem interesting for TFHE; and implement an additional library covering methods for the implementation of commonly used high-level functions.

REFERENCES

[1] 2021. PALISADE Lattice Cryptography Library (release 1.11.5). https://palisade-crypto.org/. (Accessed on 03/12/2022).

[2] David Blackman and Sebastiano Vigna. 2021. Scrambled Linear Pseudorandom Number Generators. *ACM Trans. Math. Softw.* 47, 4, Article 36 (sep 2021), 32 pages. https://doi.org/10.1145/3460772

[3] Guillaume Bonnoron, Léo Ducas, and Max Fillinger. 2018. Large FHE Gates from Tensored Homomorphic Accumulator. In *Progress in Cryptology – AFRICACRYPT 2018*, Antoine Joux, Abderrahmane Nitaj, and Tajjeeddine Rachidi (Eds.). Springer International Publishing, Cham, 217–251. https://doi.org/10.1007/978-3-319-89339-6_13

[4] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. 2018. Fly, you fool! Faster Frodo for the ARM Cortex-M4. Cryptology ePrint Archive, Report 2018/1116. https://ia.cr/2018/1116.

[5] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. 2019. Simulating Homomorphic Evaluation of Deep Learning Predictions. In *Cyber Security Cryptography and Machine Learning*, Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung (Eds.). Springer International Publishing, Cham, 212–230. https://doi.org/10.1007/978-3-030-20951-3_20

[6] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. 2018. Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In *Advances in Cryptology – CRYPTO 2018*, Hovav Shacham and Alexandra Boldyreva (Eds.). Springer International Publishing, Cham, 483–512. https://doi.org/10.1007/978-3-319-96878-0_17

[7] Zvika Brakerski, Craig Gentry, and Shai Halevi. 2013. Packed Ciphertexts in LWE-Based Homomorphic Encryption. In *Public-Key Cryptography – PKC 2013*, Kaoru Kurosawa and Goichiro Hanaoka (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–13. https://doi.org/10.1007/978-3-642-36362-7_1

[8] Zvika Brakerski and Vinod Vaikuntanathan. 2011. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. 97–106. https://doi.org/10.1109/FOCS.2011.12

[9] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. 2019. New Techniques for Multi-value Input Homomorphic Evaluation and Applications. In *Topics in Cryptology – CT-RSA 2019*, Mitsuru Matsui (Ed.). Springer International Publishing, Cham, 106–126. https://doi.org/10.1007/978-3-030-12612-4_6

[10] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. 2021. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. In *Applied Cryptography and Network Security*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer International Publishing, Cham, 460–479. https://doi.org/10.1007/978-3-030-78372-3_18

[11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437. https://doi.org/10.1007/978-3-319-70694-8_15

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2016. Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds. In *Advances in Cryptology – ASIACRYPT 2016*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 3–33. https://doi.org/10.1007/978-3-662-53887-6_1

[13] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 377–408. https://doi.org/10.1007/978-3-319-70694-8_14

[14] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2020. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology* 33, 1 (2020), 34–91. https://doi.org/10.1007/s00145-019-09319-x

[15] Ilaria Chillotti, Marc Joye, and Pascal Paillier. 2021. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In *Cyber Security Cryptography and Machine Learning*, Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann (Eds.). Springer International Publishing, Cham, 1–19. https://doi.org/10.1007/978-3-030-78086-9_1

[16] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. 2021. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *Advances in Cryptology – ASIACRYPT 2021*, Mehdi Tibouchi and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 670–699. https://doi.org/10.1007/978-3-030-92078-4_23

[17] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. 2022. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. Cryptology ePrint Archive, Report 2022/149. https://ia.cr/2022/149.

[18] Léo Ducas and Daniele Micciancio. 2015. FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In *Advances in Cryptology – EUROCRYPT 2015*, Elisabeth Oswald and Marc Fischlin (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 617–640. https://doi.org/10.1007/978-3-662-46800-5_24

[19] Morris Dworkin. 2015. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. https://doi.org/10.6028/NIST.FIPS.202

[20] Junfeng Fan and Frederik Vercauteren. 2012. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2012/144. https://ia.cr/2012/144.

[21] Nicolas Gama et al. 2016. Spqlios FFT Library. https://github.com/tfhe/tfhe/tree/master/src/libtfhe/fft_processors/spqlios. (Accessed on 04/25/2022).

[22] Craig Gentry. 2009. *A fully homomorphic encryption scheme*. Ph.D. Dissertation. Stanford University. crypto.stanford.edu/craig.

[23] Craig Gentry, Amit Sahai, and Brent Waters. 2013. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In *Advances in Cryptology – CRYPTO 2013*, Ran Canetti and Juan A. Garay (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 75–92. https://doi.org/10.1007/978-3-642-40041-4_5

[24] François Gérard and Mélissa Rossi. 2020. An Efficient and Provable Masked Implementation of qTESLA. In *Smart Card Research and Advanced Applications*, Sonia Belaïd and Tim Güneysu (Eds.). Springer International Publishing, Cham, 74–91. https://doi.org/10.1007/978-3-030-42068-0_5

[25] Antonio Guimarães, Edson Borin, and Diego F. Aranha. 2021. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021, 2 (Feb. 2021), 229–253. https://doi.org/10.46586/tches.v2021.i2.229-253

[26] Antonio Guimarães, Edson Borin, and Diego F. Aranha. 2022. MOSFHET: Optimized Software for FHE over the Torus. https://github.com/antoniocgj/MOSFHET.

[27] Anatolii Alekseevich Karatsuba and Yu P Ofman. 1962. Multiplication of many-digital numbers by automatic computers. In *Doklady Akademii Nauk*, Vol. 145. Russian Academy of Sciences, 293–294.

[28] Jakub Klemsa. 2021. Fast and Error-Free Negacyclic Integer Convolution Using Extended Fourier Transform. In *Cyber Security Cryptography and Machine Learning*, Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann (Eds.). Springer International Publishing, Cham, 282–300. https://doi.org/10.1007/978-3-030-78086-9_22

[29] Kamil Kluczniak and Leonard Schild. 2021. FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Report 2021/1135. https://ia.cr/2021/1135.

[30] Qian Lou and Lei Jiang. 2019. SHE: A Fast and Accurate Deep Neural Network for Encrypted Data. In *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.). Curran Associates, Inc., 10035–10043. http://papers.nips.cc/paper/9194-she-a-fast-and-accurate-deep-neural-network-for-encrypted-data.pdf

[31] Kotaro Matsuoka. 2020. TFHEpp: pure C++ implementation of TFHE cryptosystem. https://github.com/virtualsecureplatform/TFHEpp.

[32] Kotaro Matsuoka, Ryotaro Banno, Naoki Matsumoto, Takashi Sato, and Song Bian. 2021. Virtual Secure Platform: A Five-Stage Pipeline Processor over TFHE. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 4007–4024. https://www.usenix.org/conference/usenixsecurity21/presentation/matsuoka

[33] Daniele Micciancio and Yuriy Polyakov. 2020. Bootstrapping in FHEW-like Cryptosystems. Cryptology ePrint Archive, Report 2020/086. https://eprint.iacr.org/2020/086.

[34] Oded Regev. 2009. On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. *J. ACM* 56, 6, Article 34 (Sept. 2009), 40 pages. https://doi.org/10.1145/1568318.1568324

[35] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. 1978. On data banks and privacy homomorphisms. *Foundations of Secure Computation, Academia Press* (1978).

[36] Markku-Juhani O. Saarinen. 2019. Exploring NIST LWC/PQC Synergy with R5Sneik: How SNEIK 1.1 Algorithms were Designed to Support Round5. Cryptology ePrint Archive, Report 2019/685. https://ia.cr/2019/685.

[37] Sebastiano Vigna and David Blackman. [n.d.]. xoshiro/xoroshiro generators and the PRNG shootout. https://prng.di.unimi.it/. (Accessed on 03/11/2022).

[38] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. 2021. TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security. Cryptology ePrint Archive, Report 2021/1347. https://ia.cr/2021/1347.

[39] zama. [n.d.]. zama-ai/concrete: Concrete Operates oN Ciphertexts Rapidly by Extending TfhE. https://github.com/zama-ai/concrete. (Accessed on 02/21/2022).

[40] Tanping Zhou, Xiaoyuan Yang, Longfei Liu, Wei Zhang, and Ningbo Li. 2018. Faster Bootstrapping With Multiple Addends. *IEEE Access* 6 (2018), 49868–49876. https://doi.org/10.1109/ACCESS.2018.2867655