MOSFHET: Optimized Software for FHE over the Torus

Antonio Guimarães[1*], Edson Borin[2] and Diego F. Aranha[3]

[1]IMDEA Software Institute, Madrid, Spain.
[2]Institute of Computing, University of Campinas, Campinas, São Paulo, Brazil.
[3]Department of Computer Science, Aarhus University, Aarhus, Denmark.

*Corresponding author(s). E-mail(s): antonio.guimaraes@imdea.org;
Contributing authors: edson@ic.unicamp.br; dfaranha@cs.au.dk;

Abstract

Homomorphic encryption is one of the most secure solutions for processing sensitive information in untrusted environments, and there have been many recent advances toward its efficient implementation for the evaluation of approximated arithmetic as well as linear and arbitrary functions. 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. 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. Among the techniques we introduce, we highlight a new method for *multi-value bootstrapping* based on *blind rotation* unfolding and a *Faster-than-memory seed expansion*, which introduces speedups of up to 2 times to basic arithmetic operations.

Keywords: Homomorphic Encryption, TFHE, Functional Bootstrap, Programmable Bootstrap, Efficient Implementation

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 by Rivest *et al.* [1], but for decades proposed solutions only achieved partial homomorphism. In 2009, Gentry [2] 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 [3] and its variants [4, 5]. Many of these follow-up works presented significant improvements efficiency-wise, but the literature generally evolved around the needs of specific use 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 [6], which was proposed aiming specifically at the homomorphic evaluation of neural network algorithms, a major use case for FHE.

These algorithms require a high throughput of linear arithmetic operations and are capable of correctly operating even with relatively large imprecisions [7]. Considering that, CKKS offers a very efficient homomorphic evaluation of approximate arithmetic in a SIMD-like[1] 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 [8], 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 some of them.

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 [9] 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 [10] and lookup tables [11].

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, MOSFHET (**O**ptimized **S**oftware for **FHE** over the **T**orus) [12], 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 [7] or Programmable [11] Bootstrap and its improved version [13];
- The Circuit Bootstrap [14] and its optimizations [13];
- The multi-value bootstrap [13, 15] and its optimizations [16];
- The Key Switching and its optimizations [17];
- The BlindRot Unfolding [18] and its optimizations [19];
- The Full RGSW bootstrap [20];
- Three different approaches [13, 21, 22] for evaluating the Full-Domain Functional Bootstrap (FDFB);
- Public Key compression using randomness seed, or *seeded (R)LWE*;
- BFV-like multiplication [13]; and
- Bootstrap using Galois Automorphism [23].

It is important to note that our focus is to provide optimized implementations of these techniques, but comparing competing techniques would also require careful consideration of the choice of cryptosystem parameters. While our library provides all the necessary implementations for enabling such analyses, conducting them is beyond the scope of this project, as parameter optimization is generally an intricate and often application-dependent task [24]. Nonetheless, we benchmark all techniques with 4 different parameter sets from the literature taking note of which techniques would require larger parameter sets.

From this baseline implementation, we found several opportunities for improvements in core procedures as well as for combining existing techniques to yield better performance or error growth behavior. We also developed new methods to implement some commonly used techniques. As a result, we present the following contributions:

- We introduce *faster-than-memory seed expansion (FTM-SE): 'Seeded RLWE'* is a sample and public key compression technique so far used for memory or storage optimizations. In

[1] Single Instruction Multiple Data

this work, we show how to use it to accelerate the execution time of basic arithmetic procedures by up to 2 times.

- We also provide several other contributions to the basic arithmetic procedures (*e.g.*, FFTs and complex multiplication):
  - We analyze and characterize the impact of memory accesses (intensified by larger keys) on the performance of individual operations, with and without FTM-SE.
  - We show how the relation between key size and arithmetic performance represents a major practical challenge for techniques that should, in theory, greatly improve performance.
  - We optimize two FFT implementations using SIMD instructions to speed the execution up to 1.5 times.
- We generalize the blind rotation unfolding (as suggested by Bourse *et al.* [19]) and show that it does not achieve the expected gains on large parameters. We partially explain this behavior based on our arithmetic microbenchmarks.
- We introduce a new procedure for multi-value bootstrapping based on the blind rotation unfolding. Although significantly more expensive than existing techniques, we show that our method has unique properties and complements existing techniques (instead of competing with them).
- We optimize several techniques by combining them with others and with our aforementioned improvements. In some cases, we also add new functionalities through these combinations.

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 introduces novel techniques for TFHE; Section 5 presents the experimental results; and, finally, Section 6 concludes the paper.

2 Fully Homomorphic Encryption over the Torus (TFHE)

TFHE [9] is a fully homomorphic encryption scheme based on the Learning With Errors (LWE) problem [3] and its ring variant [25]. In this section, we describe its algebraic structures as well as its basic functioning for homomorphically evaluating linear arithmetic and arbitrary functions.

Notation

We denote as $\mathbb{S}_q^n$ the set of vectors with n elements, each of them in some set $\mathbb{S}$ modulo q. We use subscript to index elements of a vector, *i.e*, $s_i \in \mathbb{S}$ is the i-th element of $s \in \mathbb{S}^n$. We denote by $\mathbb{Z}$, $\mathbb{R}$, and $\mathbb{B}$ the sets of integer, real, and binary numbers. The real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ is the set of real numbers modulo 1. 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 define $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ as the ring of polynomials over the variable X with modulus $\Phi_{2N}(X) = X^N + 1$ and coefficients in $\mathbb{Z}_q$. Additionally, $\langle a, b\rangle$ denotes the inner product between vectors a and b, $\lceil r \rfloor_t$ the rounding of r to the closest multiple of t, and $[r]_p$ its reduction modulo p. If omitted, $t = 1$. If r is a polynomial, rounding and modular reduction are applied to each of its coefficients. Similarly, if r is a vector, rounding and modular reduction are applied to each of its elements. The *vector interpretation* of a polynomial is the list of its coefficients, in order, starting with the coefficient of the monomial with degree 0.

TFHE was originally proposed using Torus notation, but we start this paper with a generic definition over $\mathbb{Z}_q$, more common in the FHE literature. We introduce the Torus abstraction and show how it maps to $\mathbb{Z}_q$ and to $\mathcal{R}_q$ later in this section.

Encryption scheme

TFHE works with scalar and polynomial messages and encrypts them, respectively, in LWE and RLWE samples.

- An **LWE** sample is a pair $(a, b) \in \mathbb{Z}_q^{n+1}$, where a is uniformly sampled from $\mathbb{Z}_q^n$, $b = \langle a, s\rangle + e \in \mathbb{Z}_q$, and $n \geq 1 \in \mathbb{Z}$. The binary secret key

s is sampled from a uniform distribution over $\mathbb{B}^n$, and the error e is sampled from a Gaussian distribution over $\mathbb{Z}_q$ with mean 0 and standard deviation σ.

  - Encryption: We encrypt a message $m \in \mathbb{Z}_{q/\Delta}$ by adding $(0, m\Delta)$ to a fresh LWE sample, where Δ is a scaling factor meant to separate message from noise. We denote the set of LWE samples encrypting the message m with key s and parameters $k = (n, \sigma, \Delta)$ by $c \in \mathrm{LWE}_{s,k}(m)$. Textually, we refer to $c \in \mathrm{LWE}_{s,k}(m)$ as a *"sample of m"*. We omit the parameters if they are not relevant to the context and whenever it is possible to unequivocally infer them from the key or context.
  - Decryption: we first use the secret key s to calculate the phase of a sample $\mathsf{phase}(c) = b - \langle a, s\rangle$, which is the message plus the noise. Then, we round it to remove the noise and get the message $m = \lceil \mathsf{phase}(c)/\Delta \rfloor$.

- An **RLWE** sample is a pair $(a, b) \in \mathcal{R}_q^2$, where a is uniformly sampled from $\mathcal{R}_q$, and $b \in \mathcal{R}_q$ is given by $b = a \cdot S + e$, for a binary secret key S sampled from a uniform distribution over $\mathcal{R}_2$, and an error e sampled from a Gaussian distribution over $\mathcal{R}_q$ with mean 0 and standard deviation σ. Encryption and decryption are similar as described for LWE samples. We denote the set of RLWE samples encrypting the message m with key S and parameters $k = (N, \Delta, \sigma)$ by $\mathrm{RLWE}_{S,k}(\mathrm{m})$. Again, we omit the parameters whenever it is possible.

The Torus representation

Implementation-wise, TFHE usually works in $\mathbb{Z}_{2^p}$ and $\mathcal{R}_{2^p}$, where p is the word size of the implementation. The original 32-bit implementation of TFHE uses $p = 32$ whereas its experimental branch as well as most newer implementations use $p = 64$. When defined over the Torus, it relies on the map $\mathbb{T} \xrightarrow{\sim} \mathbb{Z}_{2^p}$ given by $x \mapsto x \cdot 2^p$ to be implemented because floating-point arithmetic modulo 1 is considered to be more expensive compared to just working with word-sized integers (where modular reductions for $p = 2^k$ are an architecture feature). Implementations of TFHE with prime moduli are also common in some libraries [26]. In this case, the Torus abstraction is not used.

Evaluating arithmetic

(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 $c_i = (a_i, b_i) \in$ (R)LWE(m_i) for $i \in \{0, 1\}$ be two (R)LWE samples encrypting messages m_i. The sum of them is given by $c_{\mathsf{sum}} = (a_0 + a_1, b_0 + b_1) \in$ (R)LWE$_s(m_1 + m_2)$ while $c_{\mathsf{scale}} = (a_0 \cdot z, b_0 \cdot z) \in$ (R)LWE$_s(m_1 \cdot z)$ encrypts the scaling by $z \in \mathfrak{R}$, where $\mathfrak{R}$ is a ring (typically, $\mathbb{Z}$ or $\mathcal{R}$).

(R)LWE samples also support external products by *RGSW samples*, which are sets of 2ℓ RLWE samples. They are rarely used to encrypt messages but are necessary for creating evaluation keys, which are self-encryptions of the LWE and RLWE secret keys necessary for providing fully homomorphic evaluation. We denote the set of RGSW samples encrypting the message $m \in \mathcal{R}_q$ with key $s \in \mathcal{R}_2$ and parameters $k = (n, N, \sigma, \ell)$ by $\mathrm{RGSW}_{s,k}(\mathrm{m})$. For the most part of this paper, we can consider RGSW 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 [2], 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. It defines it using 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 $c = (a, b) \in$ (R)LWE$_{s,k}(m)$ be a (R)LWE sample encrypting m, the keyswitch algorithm uses an encryption of the secret key s, defined as $\mathrm{KS}_i \in$ (R)LWE$_{s',k'}(s_i)$, to calculate the $\mathsf{phase}(c) = b - \langle a, \mathrm{KS}\rangle$. The result of this operation is $c' \in$ (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 $\mathsf{phase}(f(c)) = f(\mathsf{phase}(c))$. We should note that, by this definition, f can be a linear combination of several (R)LWE samples, which allows us, for example, to pack LWE samples in RLWE 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) [9]

```
Input  : p LWE samples
         $c^{(z)} = (a^{(z)}, b^{(z)}) \in \mathrm{LWE}_s(\mu_z)$,
         $z \in [\![1, p]\!]$
Input  : a linear morphism $f : \mathbb{Z}_q^p \mapsto \mathcal{R}_q$
Input  : a precision parameter $t \in \mathbb{Z}$
Input  : a Key Switching key
         $\mathrm{KS}_{i,j} \in \mathrm{(R)LWE}_{s'}(s_i 2^j)$, for
         $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$
Output: an (R)LWE sample
         $c' \in \mathrm{(R)LWE}_{s'}(f(\mu))$
1 for $i \in [\![1, n]\!]$ do
2     $a_i \leftarrow f(a_i^{(1)}, a_i^{(2)}, ..., a_i^{(p)})$
3     $\tilde{a}_i \leftarrow \left\lceil a_i \frac{2^t}{q} \right\rfloor$
4     Decompose $\tilde{a}_i$, s.t. $\tilde{a}_i = \sum_{j=1}^{t} \hat{a}_{i,j} \cdot 2^j$
5 Return $(0, f(b_i^{(1)}, b_i^{(2)}, ..., b_i^{(p)})) -$
     $\sum_{i=1}^{n} \sum_{j=1}^{t} \hat{a}_{i,j} \cdot \mathrm{KS}_{i,j}$
```

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.*, KS $\in$ (R)LWE$_{s',k'}(f(s))$. This version is especially useful when f depends on secret information, such as the secret key. Algorithm 2 shows the private key switching from TFHE.

2.1.2 Blind rotation

Given an LWE sample $c = (a, b) \in \mathrm{LWE}_s(m)$ and an RLWE sample $t \in \mathrm{RLWE}_{s'}(v)$, the BLINDROT procedure computes $t' = \mathrm{RLWE}_{s'}(v \cdot X^{-\lceil \mathsf{phase}(c) 2N/q \rfloor})$. Since this multiplication occurs modulo the $2N$-th cyclotomic polynomial, the operation works as a negacyclic rotation of the polynomial $v \in \mathcal{R}_q$ by an amount defined by the phase of c (thus, a *blind rotation*).

Algorithm 2: Private Functional Key Switching (PRIVATEKEYSWITCH) [9]

```
Input  : p LWE samples
         $c_z = (a_z, b_z) \in \mathrm{LWE}_s(\mu_z)$, $z \in [\![1, p]\!]$
Input  : a precision parameter $t \in \mathbb{Z}$
Input  : a Key Switching key
         $\mathrm{KS}_{i,j,z} \in \mathrm{(R)LWE}_{s'}(f_z(s)_i 2^j)$, for
         $i \in [\![1, n]\!]$, and
         $\mathrm{KS}_{n+1,j,z} \in \mathrm{(R)LWE}_{s'}(f(-1)2^j)$,
         for $j \in [\![1, t]\!]$ and $z \in [\![1, p]\!]$, where
         $f_z$ are linear morphisms
         representing a function f
Output: a (R)LWE sample
         $c' \in \mathrm{(R)LWE}_{s'}(f(\mu))$
1 for $z \in [\![1, p]\!]$ do
2     for $i \in [\![1, n]\!]$ do
3         $\tilde{a}_{z,i} \leftarrow \left\lceil a_{z,i} \frac{2^t}{q} \right\rfloor$
4         Decompose $\tilde{a}_{z,i}$, such that
            $\tilde{a}_{z,i} = \sum_{j=1}^{t} \hat{a}_{z,i,j} \cdot 2^j$
5 Return $-\sum_{z=1}^{p} \sum_{i=1}^{n+1} \sum_{j=1}^{t} \hat{a}_{z,i,j} \cdot \mathrm{KS}_{z,i,j}$
```

Algorithm 3: Blind Rotation (BLINDROT) [9]

```
Input  : a sample
         $c = (a_1, ..., a_n, b) \in \mathrm{LWE}_s(m)$
Input  : a sample $tv \in \mathrm{RLWE}_S(v)$
Input  : a list of samples $C_i \in \mathrm{RGSW}_S(s_i)$,
         for $i \in [\![1, n]\!]$
Output: an RLWE sample of
         $c' \in \mathrm{RLWE}_S(X^{\lceil \mathsf{phase}(c) 2N/q \rfloor} \cdot v)$
1 ACC $\leftarrow X^{-\left\lceil b \frac{2N}{q} \right\rfloor} \cdot tv$
2 for $i \leftarrow 1$ to $n$ do
3     $\tilde{a}_i \leftarrow \left\lceil a_i \frac{2N}{q} \right\rfloor$
4     ACC $\leftarrow$ CMUX$(C_i, X^{\tilde{a}_i} \cdot \mathrm{ACC}, \mathrm{ACC})$
5 return ACC

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

2.1.3 Sample extraction

Given an RLWE sample $c \in \mathrm{RLWE}_s(p = \sum_{i=0}^{N-1} m_i X^i)$, the SAMPLEEXTRACT procedure extracts an LWE sample encrypting a coefficient from the polynomial p, *i.e.*, $\textsc{SampleExtract}_j(c) \in \text{LWE}_{s'}(m_j)$, where s' is the vector interpretation of s.

2.1.4 The functional bootstrap

In its first version, TFHE's bootstrap was defined for evaluating only binary logic gates in a procedure called *Gate Bootstrapping*, which was later generalized for evaluating arbitrary functions discretized in Lookup Tables (LUTs). In 2019, Boura *et al.* [7] formalized the idea of a *functional bootstrap*, both for TFHE and other cryptosystems. In 2021, Chillotti*et al.* [11] introduced a discretized version of TFHE and defined the programmable bootstrapping (PBS), a formalization of the functional bootstrap specific to TFHE.

Algorithm 4 shows the functional bootstrap of TFHE.

Algorithm 4: Functional Bootstrap (FBS) [7, 11, 16]

```
Input  : an LWE sample
         $c = (a, b) \in \text{LWE}_s(m)$, 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
         $\text{BK}_i \in \text{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output: $c' \in \text{LWE}_{S'}(L[m])$, where $S' \in \mathbb{B}^N$
         is the vector interpretation of $S$
1 $b' \leftarrow \lceil b2N/q \rfloor$ and $a' \leftarrow \lceil a2N/q \rfloor$
2 $v \leftarrow \sum_{i=0}^{N-1} \Delta \cdot l_{\lfloor \frac{iB}{N} \rfloor} X^i$
3 $\text{C} \leftarrow \textsc{BlindRot}((0, v), (a', b' + \frac{2N}{4B}), \text{BK})$
4 return $\textsc{SampleExtract}_0(\text{C})$
```

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 precision parameter.

$$L = [l_1 = f(1), ..., l_B = f(B)] \mapsto \Delta \sum_{i=0}^{N-1} l_{\lfloor \frac{iB}{N} \rfloor} X^i \tag{1}$$

The negacyclic property

The table lookup is performed by using the $\textsc{BlindRot}$ to multiply the test vector, v by $X^{-\lceil \mathsf{phase}(c)2N/q \rfloor}$. 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 4 receives a LUT represented as an array of integers in $\mathbb{Z}_{B'}^B$, but it could receive directly the *test vector* v polynomial (calculated in line 2) or even an RLWE sample encrypting v. 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 [16]. 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 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 should note that we do not include proposals made for other cryptosystems. Although many could be adapted from schemes such as FHEW [27], GSW [20], or even CKKS [6], we decided 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 improved programmable bootstrap

In 2021, Chillotti *et al.* [13] presented an improved version of the programmable bootstrap. This version introduced new parameters that allow for slicing and selecting just part of the input to evaluate the function over. Let $c = (a, b) \in \mathrm{LWE}(m)$ be an LWE sample encrypting m and let $\tilde{m}$ be the binary vector representation of m, *i.e.* $m = \sum_{i=0}^{\lfloor \log_2 m \rfloor} 2^i \tilde{m}_i$. 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 into digits 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 3.5.

Algorithm 5 describes the improved version of the programmable bootstrap using the functional bootstrap of TFHE. Notice that our definition of κ and θ is different from [13] (but functionally equivalent).

Algorithm 5: Improved Programmable Bootstrap (PBS) [13]

```
Input : an LWE sample
        $c = (a, b) \in \mathrm{LWE}_s(m)$, 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 $\kappa$ and $\theta$
Input : a bootstrapping key
        $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output: $c'' \in \mathrm{LWE}_{S'}(L[\tilde{m}])$, where
        $\tilde{m} = [\lfloor m/\theta \rceil]_\kappa$ and $S' \in \mathbb{B}^N$ is the
        vector interpretation of $S$
1 $b' \leftarrow [\lfloor b/\theta \rceil]_\kappa$ and $a' \leftarrow [\lfloor a/\theta \rceil]_\kappa$
2 Let $c' = (a', b') \in \mathrm{LWE}_s([\lfloor m/\theta \rceil]_\kappa)$
3 return $\mathrm{FBS}(c', L, \mathrm{BK})$
```

3.2 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 3.6) and the Tree-Based Functional Bootstrap (Section 3.5). 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 BLINDROT to calculate just $c' \in \mathrm{RLWE}(X^{-\lceil \mathsf{phase}(c) 2N/q \rfloor})$ and, then, multiply it by each LUT (encoded in polynomials). In 2019, Carpov *et al.* [15] proposed a better method based on decomposing the polynomials that encoded the LUTs to achieve a better error output. Algorithm 6 describes it.

Algorithm 6: Multi-Value Functional Bootstrap (MVFB) [15]

```
Input : an LWE sample
        $c = (a, b) \in \mathrm{LWE}_s(m)$, $m \in \mathbb{Z}_{2N}$
Input : a scale factor $\tau$
Input : $z$ LUTs encoded in polynomials
        $\mathsf{TV}_{F_i} \in \mathcal{R}_q$, for $i \in [\![1, z]\!]$
Input : a bootstrapping key
        $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output: An array of LWE samples
        $c_i' \in \mathrm{LWE}_{S'}(F_i(m))$ for $i = 1, ..., z$,
        where $S' \in \mathbb{B}^N$ is the vector
        interpretation of $S$
1 $b' \leftarrow \lfloor b2N/q \rceil$ and $a' \leftarrow \lfloor a2N/q \rceil$
2 $v \leftarrow \sum_{i=0}^{N-1} \cdot \frac{q}{4N} \cdot \tau X^i$
3 $\mathrm{ACC} \leftarrow \mathrm{BLINDROT}((0, v), (a', b' + \frac{2N}{4B}), \mathrm{BK})$
4 foreach $i \in [\![1, z]\!]$ do
5     $c_i' \leftarrow \mathrm{SAMPLEEXTRACT}_0(\frac{\mathsf{TV}_{F_i}}{v} \cdot \mathrm{ACC})$
6 return $c'$
```

This method is a significant improvement over the straightforward version, but it still introduces significantly more noise than the single-value counterpart. Carpov *et al.* [15] 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, and s and q are the input and output bases, respectively.

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

In 2021, Guimarães *et al.* [16] improved the method by introducing a base composition with linear error growth, based on the scaling algorithm described in Algorithm 7. Equation 3 shows the final output error variance. Both works consider 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. Some corner cases, however, require additional treatment for that. Let us take, for example, a 4-slot LUT with values $L = [1, 0, 1, 1]$ representing some function f_L, 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 $s(q-1)^2 = 4(2-1)^2 = 4$. As recently pointed out by Sergiu Carpov[2], this can be solved by applying simple linear transforms to the LUT. One could, for example, generate a new lookup table $L' = L + 1 = [0, 1, 0, 0]$, and evaluate the original function as a linear function of the result: $f_L(x) = L'[x] - 1$. This process adds no noise, and we have that the norm of the factorized version of L' is smaller than $s(q-1)^2$. Alternatively, one could also solve this problem by applying the same scaling algorithm used in the base composition (Algorithm 7) 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.* [13] presented a new method for the multi-value bootstrap. Different from the previous ones, their method does not incur additional noise nor affect performance. On the other hand, the number of LUTs evaluated in each bootstrap is limited by the cryptosystem parameters, and increasing it requires reducing message precision or working with a higher probability of failure. Algorithm 8 describes the Bootstrap Many LUTs procedure.

[2]Personal communication, Jan. 22, 2025

Algorithm 7: Multiplication (Scaling) using the Multi-Value Extract (MultiValueExtractScaling) [16]

```
Input : an RLWE sample $c \in \text{RLWE}_S(\text{p})$, which is the accumulator (ACC) of
        a previous functional bootstrap, and a cleartext scalar $z \in \mathbb{Z}$
Output: an LWE sample $c' \in \text{LWE}_{S'}(z \cdot p_0)$, where $p_0$ is the constant term of $p$,
        and $S'$ is the vector interpretation of $S$
1 $c' \leftarrow \text{LWE}_S(0)$
2 for $i \leftarrow 0$ to $\lceil \frac{z}{2} \rceil - 1$ do
3     $c' \leftarrow c' + \text{SampleExtract}_i(p)$
4 for $i \leftarrow N - \lfloor \frac{z}{2} \rfloor$ to $N - 1$ do
5     $c' \leftarrow c' - \text{SampleExtract}_i(p)$
6 Return $c'$
```

Algorithm 8: Bootstrap ManyLUT (BML) [13]

```
Input : an LWE sample $c = (a, b) \in \text{LWE}_s(m)$, $m \in \mathbb{Z}_{2N}$
Input : a set L of z lookup tables, each represented by an array $L_i \in \mathbb{Z}_q^B$
        encoding a function $F_i$, for $i \in [\![0, z)$
Input : a bootstrapping key $\text{BK}_i \in \text{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Output: An array of LWE samples $c'_i \in \text{LWE}_{S'}(F_i(m))$ for $i \in [\![0, z)$,
        where $S'$ is the vector interpretation of $S$
1 $r \leftarrow \frac{N}{zB}$
2 $b' \leftarrow \lfloor b2N/q \rceil$ and $a' \leftarrow \lfloor a2N/q \rceil$
3 $v \leftarrow \sum_{i=0}^{B-1} \sum_{j=0}^{z-1} \sum_{k=0}^{r-1} L_{j,i} X^{(iq+j)r+k}$
4 ACC $\leftarrow \text{BlindRot}((0, v), (a, b + \frac{q}{4Bz}), \text{BK})$
5 foreach $i \in [\![0, z)$ do
6     $c'_i \leftarrow \text{SampleExtract}_{ir}(\text{ACC})$
7 return $c'$
```

3.3 Tensor product

As first defined, TFHE did not introduce direct multiplications between (R)LWE samples. However, there are several FHE schemes also based on the RLWE problem presenting tensorial multiplications [6, 28]. In 2021, Chillotti *et al.* [13] showed that it is possible to implement the BFV-like [28] tensor product using TFHE parameters. They also showed how it can be used to perform a multiplication between LWE samples. Algorithm 9 describes the RLWE product, and Algorithm 10 shows the multiplication between LWE samples.

Algorithm 9: RLWE Product (RLWEProd) [13]

```
Input  : two RLWE samples
         $c_i = (a_i, b_i) \in \mathrm{RLWE}_{s,\Delta}(p_i)$, for
         $p_i \in \mathcal{R}_q$ and $i \in \{0,1\}$, with scaling
         factor $\Delta$
Input  : a relinearization key
         $\mathrm{RLK}_i \in \mathrm{RLWE}_s(s^2\beta^j)$, for $j \in [\![0, t)$
Output: $c' \in \mathrm{RLWE}_s(p_0 \cdot p_1)$
1 $T \leftarrow \left[\left\lfloor \frac{A_1 \cdot A_2}{\Delta} \right\rceil\right]_q$
2 $A' \leftarrow \left[\left\lfloor \frac{A_1 \cdot B_2 + B_1 \cdot A_2}{\Delta} \right\rceil\right]_q$
3 $B' \leftarrow \left[\left\lfloor \frac{B_1 \cdot B_2}{\Delta} \right\rceil\right]_q$
4 $T' \leftarrow \lfloor T\beta^t / q \rceil$
5 Decompose $T'$, s. t. $T' = \sum_{i=0}^{t-1} \hat{T}_i \cdot \beta^j$
6 return $(A', B') + \sum_{i=0}^{t-1} \hat{T}_i \cdot \mathrm{RLK}_i$
```

Algorithm 10: LWE Multiplication (LWEMult) [13]

```
Input  : two LWE samples
         $c_i = (a_i, b_i) \in \mathrm{LWE}_s(m_i)$, for
         $m_i \in \mathbb{Z}_B$ and $i \in \{0,1\}$
Input  : a relinearization key
         $\mathrm{RLK}_i \in \mathrm{RLWE}_S(S^2\beta^j)$, for
         $j \in [\![0, t)$
Input  : a Packing Key Switching key
         $\mathrm{KSK}_{s \mapsto S}$
Output: $c' \in \mathrm{LWE}_{S'}(m_0 \times m_1)$
1 $f : \mathbb{Z}_q \mapsto \mathcal{R}_q = m \mapsto mX^0$
2 $C_0 \leftarrow$ PublicKeySwitch$(c_0, f, \mathrm{KSK})$
3 $C_1 \leftarrow$ PublicKeySwitch$(c_1, f, \mathrm{KSK})$
4 $C_{\mathsf{mul}} \leftarrow$ RLWEProd$(C_0, C_1, \mathrm{RLK})$
5 return SampleExtract$_0(C_{\mathsf{mul}})$
```

3.4 Full-Domain Functional Bootstrap (FDFB)

As detailed in Section 2.1.4, the negacyclic property restricts the functionality of the functional bootstrap. Specifically, it is capable of evaluating arbitrary functions only if the input is in the first half of the torus, *i.e.*, in integer notation, when $m\Delta < q/2$. 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 [13, 21, 22, 29], 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 the main solutions that are not purely based on high-level function pre-processing. Specifically, we implement those that require modifications to or introduce new building blocks to the cryptosystem. The following sections discuss them.

3.4.1 The tensor product method

Chillotti *et al.* [13] were the first to present a full-domain functional bootstrap for TFHE or, as they defined, a without-padding programmable bootstrap (WoP-PBS). Algorithm 11 shows their technique, proposed in 2021.

Algorithm 11: Full-Domain Functional Bootstrap based on LWEMult (FDFB-CLOT21) [13]

```
Input  : an LWE sample
         $c = (a, b) \in \mathrm{LWE}_s(m)$, 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
         $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Input  : a relinearization key
         $\mathrm{RLK}_i \in \mathrm{RLWE}_S(S^2\beta^j)$, for
         $j \in [\![0, t)$
Input  : a Packing Key Switching key
         $\mathrm{KSK}_{s \mapsto S}$
Output: $c' \in \mathrm{LWE}_{S'}(L[m])$, where $S' \in \mathbb{B}^N$
         is the vector interpretation of $S$
1 $c_a \leftarrow \mathrm{FBS}(c, L[0 : \frac{B}{2}], \mathrm{BK})$
2 $c_b \leftarrow \mathrm{FBS}(c, L[\frac{B}{2} : B], \mathrm{BK})$
3 $c_s \leftarrow \mathrm{FBS}(c, [\frac{q}{2B}, ...., \frac{q}{2B}], \mathrm{BK})$ ;        // sign
4 $\hat{c}_a \leftarrow$ LWEMult$(c_a, c_s + (0, \frac{q}{2B}), \mathrm{RLK}, \mathrm{KSK})$
5 $\hat{c}_b \leftarrow$ LWEMult$(c_b, c_s - (0, \frac{q}{2B}), \mathrm{RLK}, \mathrm{KSK})$
6 return $\hat{c}_a + \hat{c}_b$
```

3.4.2 The PubMux method

In 2021, Kluczniak and Schild [21] proposed a technique for the FDFB based on the definition of a public version of TFHE's C multiplexer (CMUX, Algorithm 3). In this version, the inputs are polynomials (instead of (R)LWE samples), and the selector is an LWE sample (instead of an RGSW sample). Algorithm 12 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.

Algorithm 12: Full-Domain Functional Bootstrap based on PubMux (FDFB-KS21) [21]

```
   Input  : an LWE sample
            $c = (a, b) \in \mathrm{LWE}_s(m)$, 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
            $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
   Input  : precision parameters $\ell, \mathfrak{B} \in \mathbb{N}$
   Output: $c' \in \mathrm{LWE}_{S'}(L[m])$, where $S'$ is the
            vector interpretation of $S$
1  $p_1 \leftarrow \sum_{i=0}^{N-1} l_{\lfloor \frac{iB}{2N} \rfloor} X^i$
2  $p_2 \leftarrow \sum_{i=0}^{N-1} -l_{\lfloor \frac{B}{2} + \frac{iB}{2N} \rfloor} X^i$
3  foreach $i \in [\![0, \ell)$ do
4      $c_{\mathsf{sign},i} \leftarrow \mathrm{FBS}(c, [\frac{-q}{2\mathfrak{B}^i}, ...., \frac{-q}{2\mathfrak{B}^i}], BK)$
5      $c_{\mathsf{sign},i} \leftarrow c_{\mathsf{sign},i} + \frac{q}{2\mathfrak{B}^i}$
6  $v \leftarrow \mathrm{PubMux}(c_{\mathsf{sign}}, p_1, p_2)$
7  return $\mathrm{FBS}(c, v, BK)$

1  Procedure PubMux(C, A, B)
2      $M \leftarrow B - A$
3      $M' \leftarrow \lfloor M\mathfrak{B}^\ell / q \rceil$
4      Decompose $M'$, s.t. $M' = \sum_{i=0}^{\ell-1} M'_i \cdot \mathfrak{B}^i$
5      return $A + \sum_{i=0}^{\ell-1} C_i \cdot M'_i$
```

3.4.3 The chaining method

The FDFB presented by Chillotti *et al.* [13] takes just one multi-value bootstrap, but it still introduces significantly more noise than the original FB, as it selects between the bootstrap lookup results using the RLWE product. In this section, we introduce another method for performing the full-domain functional bootstrap that provides the same error variance output as the basic (half-domain) FB. Algorithm 13 describes it. Despite requiring two functional bootstraps, the algorithm combines them using the chaining method [16], which provides the lowest output error variance and does not require larger parameters. This method was first presented in the *FullFBS* algorithm by Yang *et al.* [22] for their cryptosystem (TOTA). Their method, however, only removes the negacyclicity, without addressing full-domain evaluation specifically. One can obtain the original technique from Yang *et al.* [22] by replacing line 1 of Algorithm 13 with line 2 of Algorithm 4. We developed Algorithm 13 independently, but it can also be seen as an extension of TOTA's *FullFBS*. This method has also been independently presented and used several times in the literature since these first presentations.

Algorithm 13: Full-Domain Functional Bootstrap based on Chaining (FDFB-C)

```
   Input  : an LWE sample
            $c = (a, b) \in \mathrm{LWE}_s(m)$, for $m \in \mathbb{Z}_B$
   Input  : an integer LUT
            $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_B^B$
   Input  : a bootstrapping key
            $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
   Output: $c' \in \mathrm{LWE}_{S'}(L[m])$, where $S' \in \mathbb{B}^N$
            is the vector interpretation of $S$
1  $tv \leftarrow$
     $\sum_{i=0}^{\frac{B}{2}-1} \sum_{j=0}^{1} \sum_{k=0}^{\frac{N}{B}-1} \Delta l_{\frac{jB}{2}+i} X^{(2i+j)\frac{N}{B}+k}$
2  $c_{\mathsf{sign}} \leftarrow \mathrm{FBS}(c, \left[\frac{q(B+1)}{4B}, ...., \frac{q(B+1)}{4B}\right], \mathrm{BK})$
3  $c' \leftarrow c + c_{\mathsf{sign}} - \frac{q(B+1)}{4B}$
4  return $\mathrm{FBS}(c', tv, \mathrm{BK})$
```

3.5 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.

3.5.1 Tree-based and Chaining methods

One way of evaluating large lookup tables is to decompose the message into several ciphertexts and combine the evaluation of several small LUTs over the decomposed digits. In 2021, Guimarães *et al.* [16] introduced two methods for evaluating large LUTs. Algorithm 14 describes the tree-based functional bootstrap.

Algorithm 14: Tree-based functional bootstrap (TreeFB) [16]

```
Input : a set of LWE samples $c_i \in \mathrm{LWE}_s(m_i)$, 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 $\in \mathcal{R}_q$ encoding the lookup table of an arbitrary function F
Input : a bootstrapping key $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Input : a packing Key Switching key $\mathrm{KS}_{S' \mapsto S}$, where $S' \in \mathbb{B}^N$ is the vector interpretation of $S$
Output: an LWE sample $c' \in \mathrm{LWE}_{S'}(F(m))$
1 TV ← L
2 $f : \mathbb{Z}_q^B \mapsto \mathcal{R}_q$, given by:
    $f(m_0, ..., m_B) \mapsto \sum_{i=0}^{N-1} m_{\lfloor \frac{iB}{N} \rfloor} X^i$
3 for $i \leftarrow 0$ to $d-1$ do
4     $c' \leftarrow \mathrm{MVFB}(c_i, \mathrm{TV}, \mathrm{BK})$
5     foreach $j \in [\![0, B^{d-i-2})$ do
6         $\mathrm{TV}_{j-1} \leftarrow$
              $\textsc{PublicKeySwitch}(({c'}_{(j-1)B}, ..., {c'}_{jB}), f, \mathrm{KS})$
7 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 13 exemplifies it. Guimarães *et al.* [16] 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.* [29] showed that the method is capable of evaluating any function by using a digit composition as a 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.

3.5.2 (Bootstrapped) Vertical Packing method

The *vertical packing* is a technique introduced with TFHE [9] that enables efficiently evaluating larger LUTs in the leveled setting. However, this technique requires circuit bootstraps for being composable. In 2022, Bergerat *et al.* [24] showed that the bootstrapped version of the vertical packing is generally faster than the tree-based approach. The performance of this technique is mostly reliant on the circuit bootstrap, which we discuss in Section 3.6.

3.6 The circuit bootstrap

Working with (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 [9, 16, 24] require samples to be encrypted as RGSW samples. In this context, the *Circuit Bootstrap* [14] is a technique for producing an RGSW sample from an RLWE one. Since it is based on the functional bootstrap, the content of the fresh sample can be arbitrarily defined by a function. Algorithm 15 defines the circuit bootstrap based on the functional bootstrap. We are presenting a functional version of it (*i.e.*, the LUT L is a parameter), but originally it just evaluates the identity function. Since it requires the evaluation of several functions over the same input, we can use the BML algorithm, presented in Section 3.2, to accelerate the computation (as suggested by Chillotti *et al.* [13]) at the cost of a slightly increased error rate.

3.7 The full RGSW bootstrap

The TreeFB algorithm (Section 3.5) 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 RLWE samples (instead of encoded in clear-text polynomials). Carpov *et al.* [15] MVFB (Algorithm 6) does not operate over encrypted test vectors and Chillotti *et al.* [13] BML (Algorithm 8) supports a limited number of LUTs. Guimarães *et al.* [16] suggests using the CircuitBootstrap to employ the MVFB over encrypted LUTs, but they did not implement the technique as it would require

Algorithm 15: Circuit Bootstrap algorithm (CircuitBootstrap) [9]

```
Input  : an LWE sample
         $c = (a, b) \in \mathrm{LWE}_s(m)$
Input  : an integer LUT
         $L = [l_0, l_1, ..., l_{B-1}] \in \mathbb{Z}_{B'}^{B}$
Input  : a bootstrapping key
         $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Input  : a Private Key Switching key KSKA
         that evaluates
         $f : \mathbb{Z}_q \mapsto \mathcal{R}_q = f(m) \mapsto m \cdot -S$
Input  : a Packing Key Switching key
         $\mathrm{KSKB}_{s \mapsto S}$
Output: $c' \in \mathrm{RGSW}_{S,(\ell,\mathfrak{B})}(L[m])$
1 $f : \mathbb{Z}_q \mapsto \mathcal{R}_q = m \mapsto mX^0$
2 foreach $i \in [\![0, \ell)$ do
3     $\tilde{c} \leftarrow \mathrm{FBS}(c, L\mathfrak{B}^i, BK)$
4     $c'_i \leftarrow$ PrivateKeySwitch($\tilde{c}$, KSKA)
5     $c'_{\ell+i} \leftarrow$ PublicKeySwitch($\tilde{c}, f$, KSKB)
6 return $c'$
```

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 [16]. We note, however, that instead of executing the regular FB plus a CircuitBootstrap, we can just directly perform a *full RGSW bootstrap*, which uses the same number of BlindRot executions as the CircuitBootstrap, but saves time by avoiding key switchings.

The full RGSW bootstrap is similar to the functional bootstrap described in Algorithm 4, but the accumulator vector is an RGSW sample instead of an RLWE sample. In this way, the external products become internal products between RGSW samples, which are at least ℓ times more expensive but have the same output error variance. The result produced by the algorithm, encrypting $X^{-\lceil \mathsf{phase}(c)2N/q \rfloor}$, is also an RGSW sample and can, therefore, be multiplied by the decomposed LUTs in Carpov *et al.* method, even when they are encrypted in RLWE samples. Different from the original MVFB, the output error variance of such multiplication depends on the square norm of the RGSW 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 3.2.

3.8 (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 LWE samples. For RLWE 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 increased output noise. In 2021, Chen *et al.* [17] presented several algorithms that allow performing LWE Key Switching using RLWE Key Switching methods. For LWE-to-RLWE 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 are mapped to $\mathbb{Z}_{2^k}$, therefore N does not have a modular 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 by employing bootstraps, before encryption, or by switching to a different modulus $q' = \frac{q}{2N}$.

3.9 The blind rotation unfolding

The blind rotation (BlindRot, Algorithm 3) is the most expensive operation in TFHE's bootstrap. Its most expensive part, in turn, is the multiplication by RGSW samples, which encrypt s_i and is used to compute the encryption of $X^{\sum_{i=1}^{n} s_i \tilde{a}_i}$. In 2018, Zhou *et al.* [18] showed how to reduce the number of multiplications by unfolding the blind rotation loop. Equation 4 shows their proposal. In the same year, Bourse *et al.* [19] 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 with sizes 2, 4, and 8.

$$\begin{aligned} X^{as+a's'} &= ss'X^{a+a'} + s(1-s')X^{a} \\ &\quad + (1-s)s'X^{a'} + (1-s)(1-s'). \qquad (4) \end{aligned}$$

Notice that the unfolding increases the key size exponentially. In Equation 4, for example, we need to store values for ss', $s(1-s')$, $(1-s)s'$, and $(1-s)(1-s')$ instead of just s and s'. Specifically, the key expansion factor is given by $\frac{2^u}{u}$, where u is the unfolding size. Algorithm 16 shows the unfolded blind rotation, where r is the size of the expanded key.

Algorithm 16: Unfolded Blind Rotation (UBR)

```
Input : a sample $c = (a_1, ..., a_n, b) \in \mathrm{LWE}_s(m)$
Input : an unfolding level $u \in N$
Input : a sample $tv \in \mathrm{RLWE}_S(v)$
Input : a list of samples $C_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![0, \frac{n2^u}{u})$
Output: an RLWE sample of $c' \in \mathrm{RLWE}_S(X^{\lceil \mathsf{phase}(c)2N/q \rfloor} \cdot v)$
1  ACC ← $X^{-\left\lceil b\frac{2N}{q}\right\rfloor} \cdot tv$
2  $r \leftarrow \frac{n2^u}{u}$
3  for $i \leftarrow 0$ to $n-1$ by $u$ do
4      $C' \leftarrow C_{ir}$
5      for $j \in [\![1, r)$ do
6          $a' \leftarrow 0$
7          for $k \leftarrow 0$ to $u-1$ do
8              if $j \wedge 1$ then      // Bitwise AND
9                  $a' \leftarrow a' + a_{i+u}$
10             $j \leftarrow \lfloor j/2 \rfloor$
11         $\tilde{a} \leftarrow \left\lceil a'\frac{2N}{q}\right\rfloor$
12         $C' \leftarrow C' + C_{ir+j} \cdot X^{\tilde{a}}$
13     ACC ← ACC · $C'$
14 return ACC
```

3.10 State-of-the-art summary

Table 1 summarizes the techniques presented in this work 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, which is publicly available in our GitHub repository[3].

[3] https://github.com/antoniocgj/MOSFHET

4 Novel techniques

Besides the several improvements to existing techniques detailed in Section 3, we also developed entirely new procedures to accelerate core functionalities of TFHE as well as specific evaluation methods.

4.1 UBR multi-value bootstrap

The unfolded blind rotation (UBR, Section 3.9) is a theoretically promising technique, but there are several factors limiting its impact on practical performance. The main one is the exponential blow-up in the number of polynomial additions and multiplications, which is defined by the value of r in Algorithm 16. Nonetheless, in this section, we introduce another use for the unfolded blind rotation: a new method for multi-value bootstrapping.

We start from the observation that lines 5 to 12 in Algorithm 16, which contain the exponential blow-up, are not dependent on the lookup table, which is stored in ACC. Therefore, we can use the UBR to perform a multi-value bootstrap as follows:

1. Run Algorithm 16 and store the values produced for C' in an array of RGSW samples $\tilde{C}$ with size $\frac{n}{u}$.
2. Run Algorithm 17 using $\tilde{C}$.

Compared to other existing techniques for multi-value bootstrap, namely the ones we described in Section 3.2, our new method is significantly more expensive, but it introduces unique properties, and, instead of an alternative, it works as a complement to the other methods.

- Compared to the method introduced Carpov *et al.* [15], described in Algorithm 6, our method allows for the evaluation of encrypted LUTs, generally introduces less noise, and has no requirements for the format of the LUT.
- Compared to the BML method from Chillotti *et al.* [13], described in Algorithm 8, our method does not have limits on the number of LUTs it can evaluate, nor it affects the probability of failure. Notice that it essentially removes the main limitations from the BML and, hence, could work to complement the technique when needed.

Table 1: Summary of the techniques described in this work and our contributions to each.

Procedures	**Section**	**Literature**	**Improvements in this work**
PBS	3.1	[13]	-
MVFB	3.2	[15, 16]	We address a corner case on the error growth estimate.
		[13]	-
	4.1	This work	New technique
BFV-like multiplication	3.3	[13]	-
FDFB	3.4.1	[13]	Accelerated using the BLM (as suggested in [13])
	3.4.2	[21]	Accelerated using the BLM
	3.4.3	[22] and this work	New technique, extending the method of [22]
TreeFB	3.5	[16]	We use the RGSW bootstrap or the MVFB-UBR to provide MVFB for all levels of the tree.
ChainingFB			-
CircuitBootstrap	3.6	[9]	Accelerated using the BLM (as suggested in [13])
RGSW_Bootstrap	3.7	[20]	-
KeySwitching	2.1.1, 3.8	[9, 17]	Accelerated using FTM-SE
UBR	3.9	[18, 19]	Method generalized for large unfoldings (as suggested in [19])
FTM-SE	4.2	[9]	We show how to exploit fast PRNGs to improve performance

4.2 Faster-Than-Memory Seed Expansion (FTM-SE)

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 BlindRot unfolding in the bootstrap. In both cases, it is possible to achieve linear gains in performance with exponential growth in the key size. Techniques for compressing evaluation keys are broadly available in the literature. For TFHE, Chillotti *et al.* [9] suggest storing just the pseudo-random number generator (PRNG) seed used to generate the a component of RLWE samples and only generating a when necessary. This technique is known as *Seeded (R)LWE* and gives up to n times storage and memory usage improvements for LWE samples and up to 2 for RLWE samples. In this work, we not only implement this idea but also show how we can use it to improve execution time in the key switching algorithm and in basic arithmetic procedures.

Algorithm 18 shows an implementation of an RLWE sample subtraction using Seeded RLWE. We could use any PRNG to implement it, but SHAKE256 [30] 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 RLWE 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

```
Algorithm 17: Multi-Value Functional Bootstrap based on the UBR algorithm (MVFB-UBR)
Input : an LWE sample $c = (a, b) \in \mathrm{LWE}_s(m)$, $m \in \mathbb{Z}_{2N}$
Input : $z$ LUTs encoded in polynomials $\mathsf{L}_{F_i} \in \mathcal{R}_q$, for $i \in [\![1, z]\!]$
Input : a $n/u$-sized array of RGSW samples $\tilde{C}$ produced by the UBR
Output: An array of LWE samples $c'_i \in \mathrm{LWE}_{S'}(F_i(m))$ for $i = 1, ..., z$, where $S' \in \mathbb{B}^N$ is the vector interpretation of $S$
1 $b' \leftarrow \lfloor b2N/q \rceil + \frac{N}{2B}$
2 foreach $i \in [\![1, z]\!]$ do
3     $\mathrm{ACC} \leftarrow \mathsf{L}_i$
4     for $j \leftarrow 0$ to $\frac{n}{u} - 1$ do
5         $\mathrm{ACC} \leftarrow \mathrm{ACC} \cdot \tilde{C}_j$
6     $c'_i \leftarrow \mathrm{SampleExtract}_0(\mathrm{ACC})$
7 return $c'$
```

expanding the entire keys at loading time (which is typically done by most implementations), but we would lose the memory usage gains, which are one of the most important benefits of this technique.

```
Algorithm 18: RLWE subtraction using SHAKE256
Input : a seeded RLWE sample $c_0 = (seed_{a_0}, b_0) \in \mathrm{RLWE}_s(p_0)$
Input : an RLWE sample $c_1 = (a_1, b_1) \in \mathrm{RLWE}_s(p_1)$
Output: $c' = (a', b') \in \mathrm{RLWE}_s(p_0 - p_1)$
1 $a_0 \leftarrow \mathrm{SHAKE256}(seed_{a_0}, N)$
2 for $i \leftarrow 0$ to $N - 1$ do
3     $a'_i \leftarrow a_{0,i} - a_{1,i}$
4     $b'_i \leftarrow b_{0,i} - b_{1,i}$
5 return $c'$
```

To solve this problem, we implement the RLWE subtraction as shown in Algorithm 19. There are two main changes to note in this version:

1. We replace SHAKE256 with Xoshiro/Xoroshiro [31], 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 the a component of (R)LWE samples. For the security aspects of using Xoroshiro for generating a, we refer to previous literature [31–33]. If a secure PRNG is required even for public parameters, viable alternatives may be found in Lightweight Cryptography (LWC) [34].
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 (calculation of the a component) and memory (loading of the b component) intensive code portions are executed simultaneously by the processor.

```
Algorithm 19: RLWE subtraction using Xoshiro
Input : a seeded RLWE sample $c_0 = (seed_{a_0}, b_0) \in \mathrm{RLWE}_s(p_0)$
Input : an RLWE sample $c_1 = (a_1, b_1) \in \mathrm{RLWE}_s(p_1)$
Output: $c' = (a', b') \in \mathrm{RLWE}_s(p_0 - p_1)$
1 $state \leftarrow seed_{a_i}$
2 for $i \leftarrow 0$ to $N - 1$ do
3     $a'_i \leftarrow Xoroshiro128pp\_next(state) - a_{1,i}$
4     $b'_i \leftarrow b_{0,i} - b_{1,i}$
5 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 speedups over the non-compressed version. The vectorized version of Xoroshiro is a side contribution of this work.

4.2.1 VAES version

On newer computer architectures, we can also use AVX512-VAES instructions [35] to provide fast and cryptographically-secure PRNG. This is an interesting alternative to Xoroshiro and LWC-based PRNGs, as the VAES hardware support is able to introduce similar speed-ups while requiring little implementation effort and providing the security guarantees of a standard cryptographic algorithm (AES).

5 Experimental Results

We implement all algorithms presented in Section 3 in a single C library. The code is fully portable and self-contained, and includes optional optimizations for the Intel AVX2 and AVX512 Instruction Set Extensions.

We executed all experiments on a bare metal instance on AWS public cloud (`m6i.metal`), using an Intel Xeon Platinum 8375C (Ice Lake) CPU at 3.5GHz with 512GB of RAM running Ubuntu 22.04.4 LTS. Each measurement presented in this section is the average of at least 100 executions.

5.1 Parameters

We use the parameter sets reproduced in Table 2. All of them are extracted from previous literature. We select parameter sets 1 to 3 from Bergerat *et al.* [24] as representatives of some of the most commonly used parameter sizes in TFHE. Meanwhile, we adapted parameter set 4 from TFHEpp [36] since it allows evaluating all techniques described in this work.

Table 2: Parameter sets. Noise is chosen based on the dimension for obtaining a 128-bit security level.

Name	n	N	ℓ_{bs}	β_{bs}	ℓ_{ks}	β_{ks}
Set 1	585	1024	2	2^8	5	2^2
Set 2	744	2048	1	2^{23}	5	2^3
Set 3	807	4096	1	2^{22}	5	2^3
Set 4	632	2048	4	2^9	8	2^4

5.2 Basic arithmetic

Most of the optimized implementations of TFHE rely on the Fast Fourier Transform (FFT) for providing fast polynomial arithmetics [37]. Our library presents two options for FFT implementations: the SPQLIOS [38] library[4], and the FFNT library [39] for providing software portability. As an additional contribution, we optimize both libraries using AVX-512 instructions for SPQLIOS and AVX2/FMA instructions for FFNT. Table 3 shows the execution time of FFT and inverse FFT (IFFT) transforms. We note that our biggest speedups concern the inverse FFT not because of optimizations in the inner algorithm, but because AVX512 instructions enable us to perform significantly more efficient modular reductions for floating-point. We also optimized the product and the addition between polynomials, which took 516 and 314 ns, respectively. Their optimization is independent of the FFT implementation.

[4]SPQLIOS was presented by Nicolas Gama *et al.* [38] with TFHE [9]. It was adapted by TFHEpp [36] for their C++ code, which we adapted to pure C.

Table 3: FFT implementation performance for $N = 2048$. Time in microseconds.

Implementation	Source	FFT	IFFT
FFNT	[39]	8.43	11.12
FFNT (AVX2/FMA)	This work	3.58	7.49
SPQLIOS (FMA)	[38]	2.68	4.73
SPQLIOS (AVX512)	This Work	2.58	2.65

5.3 Memory impact on the basic arithmetic

At first, basic arithmetic operations such as polynomial addition are quite inexpensive compared to an FFT. In our AVX512 version of SPQLIOS, the forward FFT takes $2.6\mu s$ while a polynomial addition takes just $312ns$. However, this comparison only holds while all data is available in the processors' cache. Once we consider the evaluation of large summations, which is the typical use-case for additions (*e.g.* at the key switching), costs increase quickly. Figure 1 shows the relation between the cost of a single RLWE sample addition and the number of RLWE samples in a summation loop. It also shows the impact of our FTM-SE techniques. Each sample contains 2 polynomials and takes 32KB of memory. Our machine has 48KB, 1.25MB, and 54MB of L1, L2, and L3 cache, respectively.

The performance behavior observed in this experiment partially explains the difficulties of obtaining practical gains in techniques such as the blind rotate unfolding. A bootstrapping key using parameter Set 2 contains 1488 RLWE samples. With unfolding 2 and 4, this number increases to 2976 and 5952, respectively, which introduces a slowdown to the basic arithmetic of more than 1.25 times.

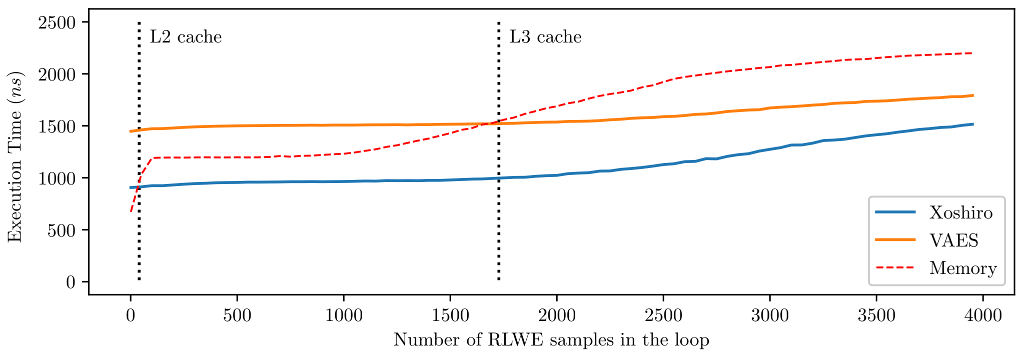

Fig. 1: Execution time of a single RLWE addition in a summation loop adding many RLWE samples. Vertical lines represent the number of samples required to fill the data cache. VAES and Xoshiro curves are the results of using the FTM-SE with the respective PRNG algorithm. Memory represents the result of adding samples without using Seeded RLWE (*i.e.*, loading everything from the cache or memory).

This experiment also allows us to compare the two algorithms we use for the FTM-SE. Xoroshiro would be certainly the best option if the issue is just performance, but VAES also provides significant gains while being a more conservative option for which security is well established.

5.4 FTM-SE

Table 4 shows the high-level functions that benefit from the FTM-SE. We use the parameter set 4, as it is the only one that evaluates all techniques with a reasonably low probability of failure. We note that this table could also be used to show the improvements we are able to achieve by combining existing techniques in the implementation. Our fastest version of the circuit bootstrap, for example, is 2.76 times faster than the most basic implementation. This number goes up to 4.3 times when we consider the FDFB methods. Notwithstanding, we must note that these results only showcase possible improvements enabled by this work, but they are not a comparison between the techniques, which would require further work on parameter optimization.

5.5 Bootstrap

Table 5 presents the execution times of the functional bootstrap and all techniques that could be used to implement the multi-value bootstrap. Dashes indicate that the technique cannot be run with the respective parameter set. Zeroes indicate that the phase (setup or LUT evaluation) is not required for the method. As we discussed in Section 4.1, the MVFB and BML methods are significantly faster than the UBR MVFB, but they also present several limitations that do not affect the UBR MVFB. On the other hand, the RGSW Bootstrap, which is also unaffected by the discussed limitations, requires much larger parameters and would only be faster than the UBR MVFB if there is a large number of LUTs to evaluate. For example, considering the UBR MVFB with unfolding $u = 4$ and supposing both methods use the same parameters (Set 4), it would be necessary at least 20 LUTs for the RGSW Bootstrap to be faster than the UBR MVFB. In practice, the UBR MVFB introduces significantly less noise, and most applications could use the UBR MVFB with Set 2 to achieve similar bootstrapping capabilities as the RGSW Bootstrap with Set 4. In this case, it would be necessary more than 70 LUTs for the RGSW Bootstrap to be faster than the UBR MVFB.

5.6 Comparison against other libraries

TFHEpp [36] is the only library to cover many of the techniques we consider in this work and, it would, at first, be a natural source for comparing the performance of our library. However, the most recent versions of TFHEpp are built using the

Table 4: High-level procedures using the FTM-SE. Execution time in microseconds using parameter set 4. Speedup over memory.

Technique	**Execution Time (μs)**			**Speedup**	
	Memory	**Xoshiro**	**VAES**	**Xoshiro**	**VAES**
PrivateKeySwitching	41,263	32,506	34,089	1.27	1.21
CircuitBootstrap	465,627	397,109	406,852	1.17	1.14
CircuitBootstrap + BML	364,231	294,620	303,325	1.24	1.20
CircuitBootstrap + BML + RLWE KS	198,911	163,244	168,722	1.22	1.18
FDFB-KS21	333,196	294,201	303,219	1.13	1.10
FDFB-KS21 + BML	233,375	194,629	199,968	1.20	1.17
FDFB-CLOT21	266,408	227,775	233,548	1.17	1.14
FDFB-CLOT21 + BML	199,055	160,928	166,489	1.24	1.20
FDFB-C	76,702	76,530	76,790	1.00	1.00

Table 5: Performance of multi-value bootstrapping methods. Execution time in microseconds.

Technique	**Set 1**		**Set 2**		**Set 3**		**Set 4**	
	Setup	**LUT**	**Setup**	**LUT**	**Setup**	**LUT**	**Setup**	**LUT**
Functional Bootstrap	0	7,552	0	12,756	0	31,531	0	33,066
BML	7,552	0	12,756	0	31,531	0	33,066	0
MVFB	7,483	3	12,692	6	31,382	13	33,022	6
RGSW Bootstrap	-	-	-	-	-	-	212,798	40
UBR MVFB (u=2)	10,987	3,528	16,184	5,683	39,036	13,476	54,766	14,612
UBR MVFB (u=4)	15,246	1,703	20,816	2,742	46,991	6,567	68,463	6,926
UBR MVFB (u=8)	106,576	838	133,139	1,413	279,332	3,308	431,239	3,284

arithmetic backend we developed in this project (specifically, the AVX512 version of SPQLIOS and the optimizations we introduce) and, hence, performance should be essentially the same.

There are also several commercial libraries implementing optimized versions of TFHE, such as OpenFHE [26] and TFHE-RS [40]. Their purpose and scope are, however, very different from ours, making it difficult to provide a fair comparison. For instance, applications and non-functional features such as maintainability and user-friendliness are completely outside of our scope. Conversely, they also do not implement the many experimental techniques and optimizations we implement in this work. Ultimately, one could obtain a superficial comparison by looking at core procedures, such as the functional bootstrap, or core arithmetic functions, such as the FTT or NTT implementations. For this work, we consider it would be misleading to present any claims over this comparison without further analysis. Nonetheless, we executed all our experiments in the same AWS instance (`m6i.metal`) TFHE-RS uses for providing their benchmark results [41], which also include data for several other libraries and should, hence, facilitate such analysis as future work.

6 Conclusion

One of our major goals in this work was to present a software platform to facilitate the development and testing of efficient methods and improvements to TFHE. MOSFHET achieves this goal, and the experimental results presented in this paper showcase its potential. As we stress throughout the paper, the library is fully portable, self-contained, and offers optional optimizations using AVX2 and AVX512 instructions, which enables efficient performance on the most commonly used CPUs while still supporting other architectures. As side contributions, it also introduces versions of Xoroshiro and SPQLIOS vectorized with AVX2 and AVX-512 ISEs.

Notwithstanding, the main contributions of this work are beyond the presentation of a library. We reviewed and implemented the main techniques presented so far for improving execution time or error behavior in the homomorphic evaluation using the TFHE scheme. While it was not within our scope to provide a comprehensive survey of the literature, our presentation of such techniques represents a contribution on its own, particularly in the form of a systematization of knowledge for TFHE. In terms of practical impact, having all the techniques presented and implemented in a single work provides consistency for comparing existing techniques as well as for developing new ones.

The novel techniques we introduced in Section 4 are also contributions of independent interest. The UBR-MVFB, for example, enables accelerating the TreeFB significantly (up to 8 times considering the results of Table 5). The improvements to basic procedures, such as our optimized version of FFT libraries and the novel FTM-SE, are more general, impacting a much broader range of techniques that are used not only in TFHE but also in most FHE implementations regardless of scheme.

As future work, we expect the library to be used not only for developing and evaluating new methods for TFHE but also in different contexts and applications.

Acknowledgments. We thank Sergiu Carpov for pointing out a better way of treating a corner case in the noise estimate of Section 3.2.

This work is partially supported by the São Paulo Research Foundation (FAPESP) under grants 2013/08293-7, 2019/12783-6, and 2021/09849-5; by the National Council for Scientific and Technological Development under grants 314645/2020-9 and 404087/2021-3; by the Independent Research Fund Denmark (DFF) project no. 1026-00350B; and by the Smart Networks and Services Joint Undertaking (SNS JU) under the European Union's Horizon Europe research and innovation programme in the scope of the CONFIDENTIAL6G project under Grant Agreement 101096435. The contents of this publication are the sole responsibility of the authors and do not in any way reflect the views of the EU.

This work was partially performed when the first author (A.G.) was a Ph.D. student at University of Campinas, Brazil, and partially when he was visiting the Department of Computer Science at Aarhus University, Denmark.

The source code for reproducing the experiments is publicly available in our GitHub repository: https://github.com/antoniocgj/MOSFHET

References

[1] Rivest, R.L., Adleman, L., Dertouzos, M.L.: On data banks and privacy homomorphisms. Foundations of Secure Computation, Academia Press (1978)

[2] Gentry, C.: A fully homomorphic encryption scheme. PhD Thesis, Stanford University (2009)

[3] Regev, O.: On Lattices, Learning with Errors, Random Linear Codes, and Cryptography. J. ACM **56**(6) (2009) https://doi.org/10.1145/1568318.1568324 . Place: New York, NY, USA Publisher: Association for Computing Machinery

[4] Brakerski, Z., Gentry, C., Halevi, S.: Packed Ciphertexts in LWE-Based Homomorphic Encryption. In: Kurosawa, K., Hanaoka, G. (eds.) Public-Key Cryptography – PKC 2013, pp. 1–13. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-36362-7_1

[5] Brakerski, Z., Vaikuntanathan, V.: Efficient Fully Homomorphic Encryption from (Standard) LWE. In: 2011 IEEE 52nd Annual Symposium on Foundations of Computer Science, pp. 97–106 (2011). https://doi.org/10.1109/FOCS.2011.12 . ISSN: 0272-5428

[6] Cheon, J.H., Kim, A., Kim, M., Song, Y.: Homomorphic Encryption for Arithmetic of Approximate Numbers. In: Takagi,

T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, pp. 409–437. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_15

[7] Boura, C., Gama, N., Georgieva, M., Jetchev, D.: Simulating Homomorphic Evaluation of Deep Learning Predictions. In: Dolev, S., Hendler, D., Lodha, S., Yung, M. (eds.) Cyber Security Cryptography and Machine Learning, pp. 212–230. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-20951-3_20

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

[9] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: TFHE: Fast Fully Homomorphic Encryption Over the Torus. Journal of Cryptology **33**(1), 34–91 (2020) https://doi.org/10.1007/s00145-019-09319-x . Accessed 2022-11-14

[10] Bonnoron, G., Ducas, L., Fillinger, M.: Large FHE Gates from Tensored Homomorphic Accumulator. In: Joux, A., Nitaj, A., Rachidi, T. (eds.) Progress in Cryptology – AFRICACRYPT 2018, pp. 217–251. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-89339-6_13

[11] Chillotti, I., Joye, M., Paillier, P.: Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In: Dolev, S., Margalit, O., Pinkas, B., Schwarzmann, A. (eds.) Cyber Security Cryptography and Machine Learning. Lecture Notes in Computer Science, pp. 1–19. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-78086-9_1

[12] Guimarães, A.: MOSFHET: Optimized Software for FHE over the Torus. original-date: 2022-04-26T12:41:21Z (2023). https://github.com/antoniocgj/MOSFHET Accessed 2023-05-16

[13] Chillotti, I., Ligier, D., Orfila, J.-B., Tap, S.: Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In: Tibouchi, M., Wang, H. (eds.) Advances in Cryptology – ASIACRYPT 2021. Lecture Notes in Computer Science, pp. 670–699. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92078-4_23

[14] Chillotti, I., Gama, N., Georgieva, M., Izabachène, M.: Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE. In: Takagi, T., Peyrin, T. (eds.) Advances in Cryptology – ASIACRYPT 2017, pp. 377–408. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-70694-8_14

[15] Carpov, S., Izabachène, M., Mollimard, V.: New Techniques for Multi-value Input Homomorphic Evaluation and Applications. In: Matsui, M. (ed.) Topics in Cryptology – CT-RSA 2019, pp. 106–126. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-12612-4_6

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

[17] Chen, H., Dai, W., Kim, M., Song, Y.: Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts. Report Number: 015 (2020). https://eprint.iacr.org/2020/015 Accessed 2023-06-29

[18] Zhou, T., Yang, X., Liu, L., Zhang, W., Li, N.: Faster Bootstrapping With Multiple Addends. IEEE Access **6**, 49868–49876 (2018) https://doi.org/10.1109/ACCESS.2018.2867655 . Conference Name: IEEE Access

[19] Bourse, F., Minelli, M., Minihold, M., Paillier, P.: Fast Homomorphic Evaluation of Deep Discretized Neural Networks. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018, pp. 483–512. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96878-0_17

[20] Gentry, C., Sahai, A., Waters, B.: Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In: Canetti, R., Garay, J.A. (eds.) Advances in Cryptology – CRYPTO 2013. Lecture Notes in Computer Science, pp. 75–92. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40041-4_5

[21] Kluczniak, K., Schild, L.: FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption. Report Number: 1135 (2021). https://eprint.iacr.org/2021/1135 Accessed 2023-05-16

[22] Yang, Z., Xie, X., Shen, H., Chen, S., Zhou, J.: TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security. Report Number: 1347 (2021). https://eprint.iacr.org/2021/1347 Accessed 2023-05-16

[23] Lee, Y., Micciancio, D., Kim, A., Choi, R., Deryabin, M., Eom, J., Yoo, D.: Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption. Report Number: 198 (2022). https://eprint.iacr.org/2022/198 Accessed 2023-04-10

[24] Bergerat, L., Boudi, A., Bourgerie, Q., Chillotti, I., Ligier, D., Orfila, J.-B., Tap, S.: Parameter Optimization & Larger Precision for (T)FHE. Published: Cryptology ePrint Archive, Paper 2022/704 (2022). https://eprint.iacr.org/2022/704

[25] Lyubashevsky, V., Peikert, C., Regev, O.: On Ideal Lattices and Learning with Errors over Rings. In: Gilbert, H. (ed.) Advances in Cryptology – EUROCRYPT 2010. Lecture Notes in Computer Science, pp. 1–23. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-13190-5_1

[26] Badawi, A.A., Bates, J., Bergamaschi, F., Cousins, D.B., Erabelli, S., Genise, N., Halevi, S., Hunt, H., Kim, A., Lee, Y., Liu, Z., Micciancio, D., Quah, I., Polyakov, Y., R.V, S., Rohloff, K., Saylor, J., Suponitsky, D., Triplett, M., Vaikuntanathan, V., Zucca, V.: OpenFHE: Open-Source Fully Homomorphic Encryption Library. Report Number: 915 (2022). https://eprint.iacr.org/2022/915 Accessed 2023-05-16

[27] Ducas, L., Micciancio, D.: FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second. In: Oswald, E., Fischlin, M. (eds.) Advances in Cryptology – EUROCRYPT 2015, pp. 617–640. Springer, Berlin, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46800-5_24

[28] Fan, J., Vercauteren, F.: Somewhat Practical Fully Homomorphic Encryption. Report Number: 144 (2012). https://eprint.iacr.org/2012/144 Accessed 2023-05-21

[29] Clet, P.-E., Zuber, M., Boudguiga, A., Sirdey, R., Gouy-Pailler, C.: Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping. Report Number: 149 (2022). https://eprint.iacr.org/2022/149 Accessed 2023-05-21

[30] Dworkin, M.J.: SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology (July 2015). https://doi.org/10.6028/NIST.FIPS.202 . https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf Accessed 2023-05-21

[31] Blackman, D., Vigna, S.: Scrambled Linear Pseudorandom Number Generators. ACM Transactions on Mathematical Software **47**(4), 36–13632 (2021) https://doi.org/10.1145/3460772 . Accessed 2023-05-16

[32] Bos, J.W., Friedberger, S., Martinoli, M., Oswald, E., Stam, M.: Fly, you fool! Faster Frodo for the ARM Cortex-M4. Report Number: 1116 (2018). https://eprint.iacr.

org/2018/1116 Accessed 2023-05-21

[33] Gérard, F., Rossi, M.: An Efficient and Provable Masked Implementation of qTESLA. In: Belaïd, S., Güneysu, T. (eds.) Smart Card Research and Advanced Applications. Lecture Notes in Computer Science, pp. 74–91. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-42068-0_5

[34] Saarinen, M.-J.O.: Exploring NIST LWC/PQC Synergy with R5Sneik: How SNEIK 1.1 Algorithms were Designed to Support Round5. Report Number: 685 (2019). https://eprint.iacr.org/2019/685 Accessed 2023-05-21

[35] Drucker, N., Gueron, S., Krasnov, V.: Making AES Great Again: The Forthcoming Vectorized AES Instruction. In: Latifi, S. (ed.) 16th International Conference on Information Technology-New Generations (ITNG 2019). Advances in Intelligent Systems and Computing, pp. 37–41. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-14070-0_6

[36] Matsuoka, K.: TFHEpp: pure C++ implementation of TFHE cryptosystem (2020). https://github.com/virtualsecureplatform/TFHEpp

[37] Chu, E., George, A.: Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms. CRC Press, ??? (1999). Google-Books-ID: 30S3kRiX4xgC

[38] Gama, N., et al.: Spqlios FFT Library (2016). https://github.com/tfhe/tfhe/tree/master/src/libtfhe/fft_processors/spqlios

[39] Klemsa, J.: Fast and Error-Free Negacyclic Integer Convolution using Extended Fourier Transform. Report Number: 480 (2021). https://eprint.iacr.org/2021/480 Accessed 2023-05-22

[40] Zama: TFHE-rs: Pure Rust implementation of the TFHE scheme for boolean and integers FHE arithmetics. Zama (2023). https://github.com/zama-ai/tfhe-rs Accessed 2023-05-16

[41] Zama: Benchmarks - TFHE-rs. https://docs.zama.ai/tfhe-rs/getting-started/benchmarks Accessed 2023-05-16