Universidade Estadual de Campinas
Instituto de Computação

Antonio Carlos Guimarães Junior

Accelerating FHE for arbitrary computation

Acelerando FHE para computação arbitrária

CAMPINAS
2023

Antonio Carlos Guimarães Junior

Accelerating FHE for arbitrary computation

Acelerando FHE para computação arbitrária

Tese apresentada ao Instituto de Computação da Universidade Estadual de Campinas como parte dos requisitos para a obtenção do título de Doutor em Ciência da Computação.

Thesis presented to the Institute of Computing of the University of Campinas in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Supervisor/Orientador: Prof. Dr. Edson Borin
Co-supervisor/Coorientador: Prof. Dr. Diego de Freitas Aranha

Este exemplar corresponde à versão final da Tese defendida por Antonio Carlos Guimarães Junior e orientada pelo Prof. Dr. Edson Borin.

CAMPINAS
2023

Ficha catalográfica
Universidade Estadual de Campinas
Biblioteca do Instituto de Matemática, Estatística e Computação Científica
Ana Regina Machado - CRB 8/5467

G947a

Guimarães Junior, Antonio Carlos, 1994-
Accelerating FHE for arbitrary computation / Antonio Carlos Guimarães Junior. – Campinas, SP : [s.n.], 2023.

Orientador: Edson Borin.
Coorientador: Diego de Freitas Aranha.
Tese (doutorado) – Universidade Estadual de Campinas, Instituto de Computação.

1. Criptografia homomórfica. I. Borin, Edson, 1979-. II. Aranha, Diego de Freitas, 1982-. III. Universidade Estadual de Campinas. Instituto de Computação. IV. Título.

Informações Complementares

Título em outro idioma: Acelerando FHE para computação arbitrária
Palavras-chave em inglês:
Homomorphic encryption
Área de concentração: Ciência da Computação
Titulação: Doutor em Ciência da Computação
Banca examinadora:
Edson Borin [Orientador]
Julio César López Hernández
Marcos Antônio Simplício Júnior
Ricardo Dahab
Jeroen Antonius Maria van de Graaf
Data de defesa: 01-09-2023
Programa de Pós-Graduação: Ciência da Computação

Identificação e informações acadêmicas do(a) aluno(a)
- ORCID do autor: https://orcid.org/0000-0001-5110-6639
- Currículo Lattes do autor: http://lattes.cnpq.br/3952604251815458

Universidade Estadual de Campinas
Instituto de Computação

Antonio Carlos Guimarães Junior

Accelerating FHE for arbitrary computation

Acelerando FHE para computação arbitrária

Banca Examinadora:

- Prof. Dr. Edson Borin
  IC/UNICAMP
- Prof. Dr. Julio Cesar Lopez Hernandez
  IC/UNICAMP
- Prof. Dr. Marcos Antonio Simplicio Junior
  PCS/USP
- Prof. Dr. Ricardo Dahab
  IC/UNICAMP
- Prof. Dr. Jeroen Antonius Maria van de Graaf
  DCC/UFMG

A ata da defesa, assinada pelos membros da Comissão Examinadora, consta no SIGA/Sistema de Fluxo de Dissertação/Tese e na Secretaria do Programa da Unidade.

Campinas, 01 de setembro de 2023

Agradecimentos

Agradeço aos meus pais, orientadores, colegas, amigos, e a todos que de alguma forma contribuíram para a realização desse trabalho.

Este trabalho foi financiado por:

- processo nº 2013/08293-7, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)
- processo nº 2019/12783-6, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)
- processo nº 2021/09849-5, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP)
- Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001

Resumo

Permitir a avaliação de computação arbitrária sobre dados criptografados elevou a criptografia totalmente homomórfica (FHE, do inglês *Fully Homomorphic Encryption*) a um lugar de destaque dentre as tecnologias de preservação da privacidade. A princípio, entretanto, o feito teve um impacto diminuto na prática, já que a maioria das aplicações não suportava a sobrecarga no desempenho que ela introduzia. A literatura então evoluiu em torno das necessidades de casos de uso específicos, muitas vezes abrindo mão da computação arbitrária em nome do desempenho. Atualmente, o estado da arte em desempenho para FHE é representado por esquemas que se especializam em aritmética rápida ao mesmo tempo em que relegam funções arbitrárias a aproximações polinomiais. Do outro lado dessa questão, o esquema TFHE (do inglês FHE over the Torus) [Chillotti et al., 2016] provê computação arbitrária enquanto luta por um nível de desempenho competitivo. Este trabalho é dedicado a melhorá-lo.

O TFHE possibilita a avaliação de funções arbitrárias através de uma técnica chamada *functional bootstrap*, mas seu custo cresce superlinearmente com a precisão da função, o que, a princípio, o tornava adequado apenas para funções com pequena precisão. Neste trabalho, apresentamos alguns dos primeiros métodos para permitir uma avaliação mais eficiente de funções arbitrárias com alta precisão usando TFHE. Nossos métodos permitiram ganhos de velocidade de até 3,2 vezes em relação à literatura anterior usando o bootstrap funcional e de até 8,74 vezes em comparação com outros métodos de avaliação. Também avançamos o TFHE otimizando e propondo novas técnicas para alguns de seus principais procedimentos e suas implementações. Entre essas melhorias, destacamos uma otimização em sua aritmética básica que atinge uma aceleração de até 2 vezes em relação às implementações anteriores e um novo método para avaliar o bootstrap funcional com várias funções ao mesmo tempo (empacotamento MISD, do inglês *Multi-Instruction Single-Data*). Por fim, com foco no lado prático de FHE, testamos nossas contribuições em um cenário do mundo real implementando a avaliação homomórfica de um algoritmo de inferência em dados do genoma humano.

Abstract

Enabling the evaluation of arbitrary computation over encrypted data raised fully homomorphic encryption (FHE) to the spotlight among privacy-preserving technologies. Initially, however, the feat had a deminute impact in practice, as most applications could not bear the performance overhead it introduced. The literature then evolved around the needs of specific use cases, often forfeiting arbitrary computation in the name of performance. Currently, state-of-the-art performance on FHE is represented by schemes that specialize in fast arithmetic while relegating arbitrary functions to polynomial approximations. On the other side of this issue, the TFHE scheme (FHE over the Torus) [Chillotti et al., 2016] upholds arbitrary computation while fighting for a competitive performance level. This work is dedicated to improving it.

TFHE enables the evaluation of arbitrary functions through a technique called *functional bootstrapping*, but its cost grows superlinearly with the function precision, which, at first, made it only suitable for functions with small precision. In this work, we introduced some of the first methods to allow a more efficient evaluation of arbitrary functions with high precision using TFHE. Our methods enabled speedups of up to 3.2 times over previous literature using the functional bootstrapping, and of up to 8.74 times compared to other evaluation methods. We also advanced TFHE by optimizing and proposing new techniques for some of its core procedures and their implementations. Among these improvements, we highlight an optimization on its basic arithmetic that achieves a speedup of up to 2 times over previous implementations and a new method for evaluating the functional bootstrapping with several functions at once (MISD, Multi-Instruction Single-Data, batching). Finally, with a focus on the practical side of FHE, we tested our contributions in a real-world scenario by implementing the homomorphic evaluation of an inference algorithm on human genome data.

List of Figures

List of Tables

List of Symbols

$\mathbb{Z}$	The set of integer numbers
$\mathbb{R}$	The set of real numbers
$\mathbb{Q}$	The set of rational numbers
$\mathbb{T}$	The real torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ (real numbers modulo 1)
$\mathbb{S}_q$	The set of elements in some set $\mathbb{S}$ modulo q, which is also in $\mathbb{S}$
$\mathbb{S}_q[X]$	The set of polynomials over the variable X with coefficients in $\mathbb{S}_q$
$\Phi_N(X)$	The N-th cyclotomic polynomial over the variable X
R_q	The ring of polynomials $\mathbb{Z}_q[X]/(X^N + 1)$ over the variable X with modulus $\Phi_{2N}(X) = X^N + 1$ and coefficients in $\mathbb{Z}_q$
σ	The standard deviation
σ^2	The variance
ℓ	The number of levels in a digit decomposition
β	The base of a digit decomposition
Δ	A scaling factor

List of Algorithms

Contents

Chapter 1

Introduction

Cryptography has always been an important tool when handling sensitive information. It can provide formal guarantees of privacy and is broadly used to protect data at rest or in transit. During processing, on the other hand, data might be equally vulnerable while protecting it presents many additional challenges. At first, one could consider processing only in trusted environments as a solution for this problem. However, while that may be a solution for some applications in specific contexts, many could benefit from outsourcing computation to the public cloud, and others rely on multi-party computation protocols. In both cases, data needs to be (partially or fully) processed in untrusted environments, thus requiring some way of protection. In this context, an ideal solution is in the use of fully homomorphic encryption (FHE), which allows performing computation over encrypted data.

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.* [61], but for decades proposed solutions only achieved partial homomorphism. In 2009, Gentry [36] 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 [59] and its variants [13, 15]. 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.

Current FHE schemes are often divided into two branches. The first is characterized for providing efficient arithmetic in a SIMD-like[1] manner (*i.e.*, they evaluate a single arithmetic operation over multiple data, at once, in an efficient way). BGV [14] and BFV [12, 33] were some of the first schemes providing such capabilities and remain the state-of-the-art when considering exact arithmetic. More recently, the CKKS scheme [20] introduced a new way of encoding data for performing SIMD-like operations by relying on approximate arithmetic. It was proposed considering specifically the homomorphic evaluation of neural network algorithms, a major use case for FHE. These algorithms

[1]SIMD: Single Instruction, Multiple Data

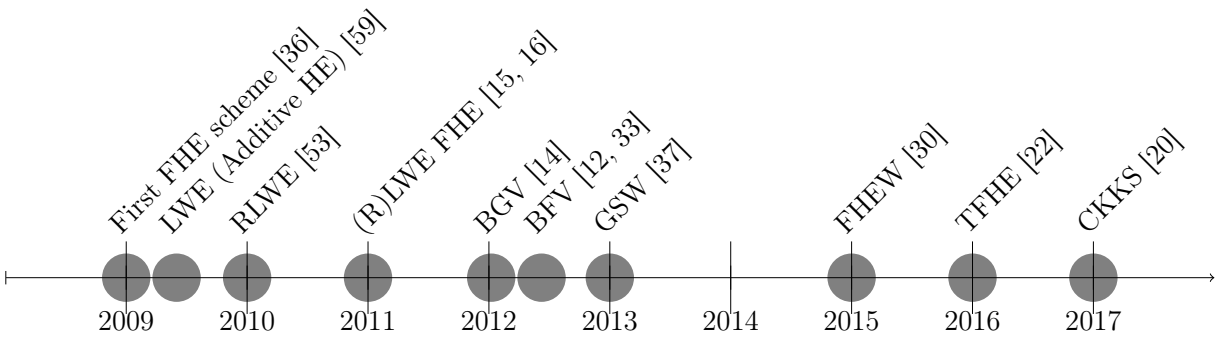


Figure 1.1: Chronology of the main HE schemes since 2009.

require a high throughput of arithmetic operations and are capable of correctly operating even with relatively large imprecisions [9]. In this way, CKKS quickly became one of the most efficient solutions for homomorphic evaluation. Its efficiency, however, restricts functionality, as the scheme is inherently approximate and needs to rely on further arithmetic approximations for arbitrary functions. The cost of evaluating these functions might grow exponentially with the desired precision [52], 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.

On the other branch of modern FHE, we have GSW-based [36] schemes such as FHEW [30] and TFHE [22]. Instead of focusing on providing fast SIMD arithmetic (although they still enable some methods for doing so), they represent applications as combinations 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, and they achieve performance by minimizing the latency of evaluating these components. However, large applications require a great number of them, and further performance improvements are still necessary for making these schemes practical for many applications. Another major advantage of these schemes is their capability for evaluating arbitrary functions. Providing larger arithmetic precision for these evaluations is however still a challenge, as existing solutions would often lead to exponential costs.

1.1 Contributions

In this work, we focus on the TFHE scheme and address some of the main aspects limiting its applicability.

Improving precision

Evaluating arbitrary (nonlinear) functions is generally a challenge for FHE schemes. TFHE is not an exception to that, but it provides methods for efficiently evaluating arbitrary functions with small precision. In our paper titled “Revisiting the Functional Bootstrap in TFHE” [38], published at the IACR Transactions on Cryptographic Hardware and Embedded Systems (CHES 2021), we present two methods for enabling the

efficient evaluation of arbitrary functions with high-precision using TFHE. We also introduce a method for minimizing noise at homomorphic multiplications and a few other side contributions. Compared to previous literature using TFHE, our methods are up to 2.49 times faster than the lookup table evaluation of Carpov *et al.* [17] and up to 3.19 times faster than the 32-bit integer comparison of Bourse *et al.* [11]. We also achieve speedups of up to 6.98, 8.74, and 3.55 times over 8-bit implementations of the ReLU, Addition, and Maximum functions, respectively.

Optimizing performance

From the very basic arithmetic procedures up to the high-level functions that compose the building blocks of the scheme, we present several contributions to accelerate TFHE, concerning both implementation and algorithmic aspects. In our paper titled "MOSFHET: Optimized Software for FHE over the Torus" [39], currently under consideration for publication in a journal, 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. We also introduce novel improvements to several of them and new approaches to implement some commonly used procedures. Furthermore, we 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 rotate unfolding and a *Faster-than-memory seed expansion*, which introduces speedups of up to 2 times to basic arithmetic operations.

Real-world applications

Using our novel techniques as well as the improvements we build over previous literature, we design solutions for using TFHE in applications that are typical use cases for FHE. In our paper "Homomorphic evaluation of large look-up tables for inference on human genome data in the cloud" [40], published at the 2nd Workshop on Cloud Computing, we propose and analyze a candidate we implemented for the homomorphic encryption track of iDash 2022, an annual competition for creating new solutions to tackle the challenges of securing human genome processing in untrusted environments. We focus on different approaches for optimizing its homomorphic evaluation using some of the techniques and implementations previously developed in our project. As a result, we not only show the practicability of our solution in the context of iDash but also provide key insights on the practical issues of employing popular homomorphic encryption techniques, such as LUT evaluation, in a real-world scenario.

1.2 Structure

The rest of this document is organized into five chapters. Chapter 2 introduces the basic notation and the core concepts necessary for understanding LWE-based cryptography, fully homomorphic encryption, and the TFHE scheme. Chapter 3 presents our first set

of contributions, in which we focus on improving the performance of TFHE for the evaluation of functions with high precision. Chapter 4 introduces MOSFHET, our optimized library implementing TFHE, and the main proposals that have been presented so far for improving it in the literature. It also presents the first results we were able to produce with the library, including novel techniques that we developed for the scheme. Chapter 5 presents the results of using our library for implementing an inference algorithm on human genome data, which is a typical use case for FHE and a good representative of real-world applications. Finally, Chapter 6 presents our conclusions and introduces future work.

Chapter 2

Theoretical basis

In this chapter, we introduce the main concepts necessary for the understanding of this work. The chapter was designed to be as self-contained as possible, referring to additional literature only for tangential or very basic concepts. It is important to note that this work does not address the security aspects of fully homomorphic encryption or LWE-based cryptography. We define the basic security problems only to help clarify the understanding of the homomorphisms they provide, but we do not approach the security reductions on which they rely.

2.1 Notation and basic arithmetic definitions

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 $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{B}$, and $\mathbb{C}$ the sets of integer, real, binary, and complex 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 $\mathbb{Z}_q$. In FHE in general, q is typically but not necessarily a prime. For this work specifically, we often choose q as a power of 2 for improving performance. For details on the basic arithmetic in these sets, refer to Sections 1.1 and 1.2 of Reference [63].

Rounding and modular reduction We denote $\lceil r \rfloor_t$ 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.

Vector interpretation and operations Given a polynomial $p \in \mathcal{R}_q$, its *vector interpretation* is a vector $v \in \mathbb{Z}_q^N$ given by the list of coefficients of p, *i.e.*, $p = \sum_{i=0}^{N-1} p_i X^i \mapsto v = [p_0, p_1, \ldots, p_{N-1}]$. Angle brackets $\langle a, b \rangle$ denote the inner product between vectors a and b. Arithmetic operations, *e.g.* additions and multiplications, occur element-wise.

Functions We present functions by first enunciating their domain and co-domain and, then, defining their exact map. For example, we denote the square function over the integers by $f : \mathbb{Z} \mapsto \mathbb{Z} = f(x) \mapsto x^2$.

Lookup tables Lookup tables (LUTs) are extensively used in this work as a way of representing discretized functions. A LUT is a vector $L = [l_0, l_1, \ldots, l_{n-1}]$ with n *elements* encoding a function f iff $l_i = f(i)$ for all $i \in [\![0, n-1]\!]$. A *lookup* is a procedure that receives a LUT L and a *selector* i and returns the *element* (or *value*) in the i-th position of L. Figure 2.1 shows the evaluation of a square function using a look-up table.

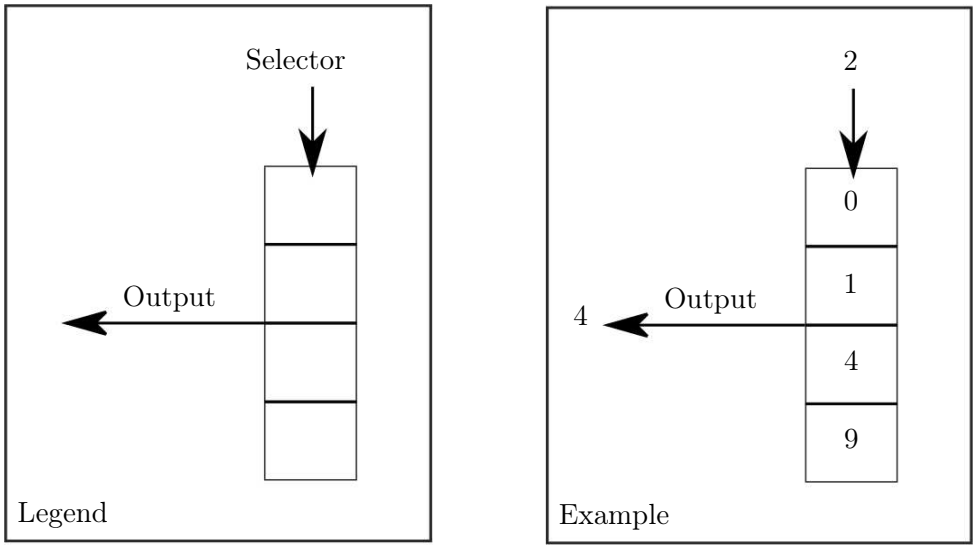


Figure 2.1: Lookup table (LUT) example.

2.2 Fully-homomorphic encryption (FHE)

The notion of homomorphic encryption is as old as public key encryption [61], and the same can be said about its first basic construction, as the RSA cryptosystem [60] itself already presented homomorphic properties. For decades, however, no solution was capable of providing a full homomorphism, *i.e.*, of being functionally complete. Still, significant progress was made throughout these years through the presentation of partially homomorphic encryption (PHE) schemes. These proposals achieved group homomorphisms, which we could informally define as being capable of evaluating just one type of arithmetic operation. Examples of partially homomorphic encryption (PHE) schemes include the Paillier cryptosystem [58], which allows only for additions, and the ElGamal cryptosystem [32], which allows only for multiplications between ciphertexts. It was only in 2009 that Gentry [36] presented the first Fully Homomorphic encryption (FHE) scheme. His scheme was based on ideal lattices and enabled arbitrary computation through the evaluation of `nand` gates. Figure 2.2 illustrates the use of FHE for protecting computation in untrusted environments. Overbar indicates the encrypted version of the input (*in*) and output (*out*) data, and f' is the implementation of f using logic gates that can be homomorphically evaluated.

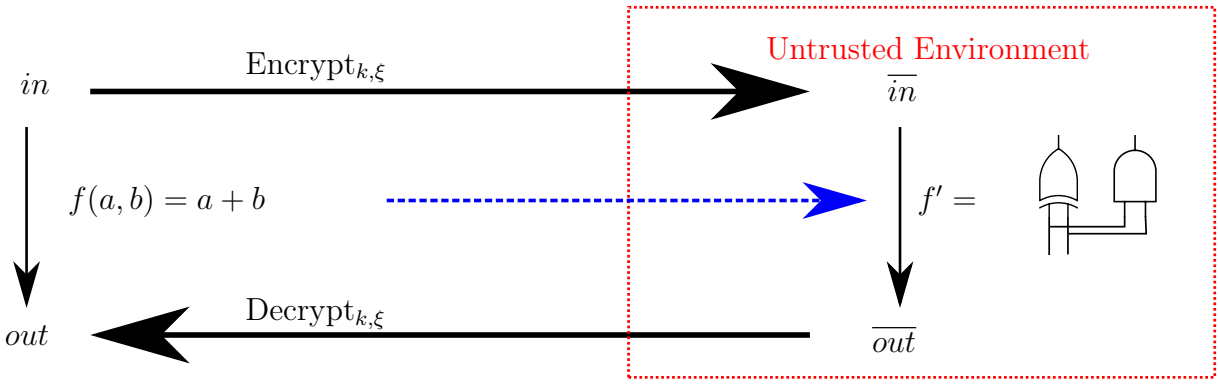


Figure 2.2: Illustration of FHE using binary gates.

Gentry [36] also revisited the definition of Homomorphic Encryption, which we reproduce in Definition 1.

Definition 1 (Homomorphic Encryption, [36]). *Let $c_i = Encrypt_{\xi,s}(m_i)$ be a set of ciphertexts encrypting a set of messages m_i, for $i \in [\![0, n]\!]$, with secret key s and a cryptosystem ξ. If ξ is fully homomorphic, it provides a function $Eval_\xi$ with which anyone knowing the ciphertexts (but not necessarily s) should be able to evaluate an arbitrary computationally solvable function f over the messages and obtain its encrypted result $\overline{c_f} = Eval_\xi(f, c_0, c_1, ..., c_n)$. Furthermore, ξ should provide the following properties:*

- **Correctness:** $f(m_0, m_1, ..., m_n) = Decrypt_{\xi,s}(Eval_\xi(f, c_0, c_1, ..., c_n))$
- **Security:** Definitions for chosen-ciphertext attacks (CCA) and semantic security games for HE schemes are the same as for general public-key cryptography and are detailed by Gentry [36]. Particularly, HE schemes should be semantically secure and secure against non-adaptive CCA.
- **Compactness:** The complexity of the decryption algorithm should depend only on the security parameters, and not on the evaluated function. It also implies that the size of $\overline{c_f}$ should not depend on the complexity of f.

This first scheme presented by Gentry was generally considered overly complex and relied on security assumptions that were not well-studied [16], such as the sparse subset-sum problem. However, it also established a blueprint for building FHE schemes, which was later used to build schemes based on other problems and assumptions. Most of the modern fully homomorphic encryption schemes are based on the Learning With Errors (LWE) problem or some of its variants [53], following the work of Brakerski and Vaikuntanathan [16]. The problem was introduced by Regev [59] and has security based on the worst-case of the Shortest Vector Problem on arbitrary lattices.

Figure 2.3 shows a toy example of one of the main challenges faced by modern HE schemes: noise management. It also exemplifies how Gentry's blueprint proposes dealing with it. All currently known fully homomorphic encryption schemes rely on noisy ciphertexts for security, *i.e.*, the encryption process adds a small error (noise) to the message. In Figure 2.3, we can see how message and noise behave during evaluation. In the first step,

we are showing the encryption of two 6-bit messages in 16-bit words. The messages are in the most significant bits while the 3-bit noises are the least significant bits, following the *encoding* adopted by cryptosystems such as BFV and TFHE. For now, we are just taking the addition of noise as a security necessity, Section 2.3 will further discuss the exact security problem we are relying on. In step 2, the words are added, which also adds the messages in $\mathbb{Z}_{2^6}$ and the noise in $\mathbb{Z}$. Notice that the noise has grown from 3 to 4 bits. It continues to grow as we perform more arithmetic operations. Eventually (step 3), it reaches a point at which significant bits of the message could be affected if we perform more arithmetic. At this point, there are three alternative approaches to follow.

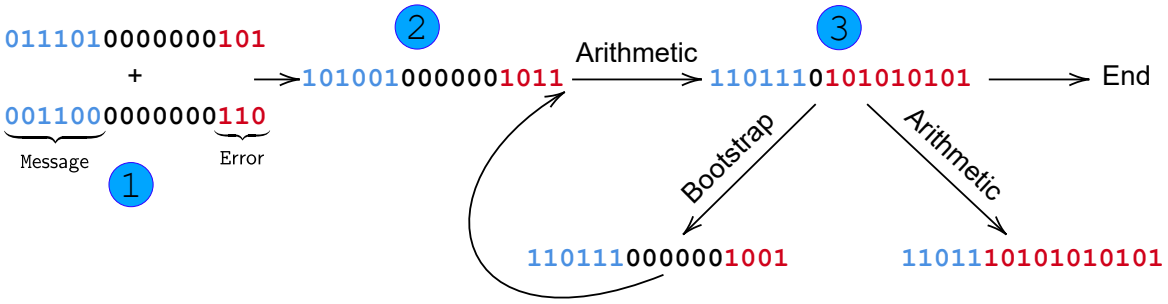


Figure 2.3: Diagram depicting noise behavior in modern HE schemes.

The first is to simply continue to perform arithmetic and treat messages as approximate results. In the encoding we adopt for this example, we could only do it for so long before the message is completely lost. Other schemes, such as CKKS, adopt encodings that allow for both the message and noise to grow, thereby enabling the possibility of a much larger number of approximate arithmetic operations. The second approach would be to end the evaluation there, which is often used with schemes such as BFV and BGV. These first two approaches are known as *leveled setting*. Its main restriction is in the number of operations it can evaluate, which needs to be defined when setting up the scheme. Specifically, it is necessary to estimate *parameters* that enable evaluating the desired number of operations. Large applications generally require large parameters, which could introduce a significant performance overhead on applications. Nonetheless, schemes following these approaches are the current state-of-the-art for efficiency in homomorphic evaluation. They present a very high throughput of operations by packing arithmetic (SIMD), but they also present no efficient ways of directly evaluating arbitrary functions, thus relying on approximations for them.

The third approach is the *bootstrapped setting*, where schemes reset the noise by using a *bootstrap procedure*. This allows schemes to evaluate an unbounded number of operations, but the *bootstrap* is usually very expensive in schemes such as BGV, BFV, and CKKS. Most schemes following this approach, *e.g.* TFHE and FHEW, enable fast bootstraps by using very small parameters that only allow for the evaluation of small logic components, such as logic gates and lookup tables. They can represent entire applications as combinations of such components. Therefore, they do not require approximate computing and enable straightforward implementations of any application with almost no algorithmic modifications. Scaling performance for larger applications is, however, a

problem, as larger applications may require a vast number of logic components.

2.3 LWE-based cryptography

First introduced by Regev in 2009 [59], the Learning With Errors problem has been extensively used in FHE, and several variants have been presented so far. When considering a specific scheme, definitions and notations generally tend to also be specific to the requirements of such scheme. Therefore, we start with the LWE definition presented by the TFHE scheme, our main focus in this work. This variant (with adjusted parameters) is at least as hard as solving the standard LWE problem [55].

Definition 2 (Binary-Secret Scale-invariant LWE (from TFHE [22]))**.** *Let an LWE sample be a pair* $(a, b) \in \mathbb{Z}_q^{n+1}$*, where* a *is uniformly sampled from* $\mathbb{Z}_q^n$*,* $b = \langle a, s \rangle + e \in \mathbb{Z}_q$*,* $n > 1 \in \mathbb{Z}$*, and* $\langle\ ,\ \rangle$ *denotes the inner product. The secret key* s *is uniformly sampled from* $\mathbb{B}^n$ *and the error* e *is sampled from a discretized Gaussian distribution over* $\mathbb{Z}$ *with mean* 0 *and standard deviation* σ*. Given a polynomially 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 $\mathbb{Z}_q^{n+1}$.

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).

2.4 Fully Homomorphic Encryption over the Torus (TFHE)

TFHE [22] is a fully homomorphic encryption scheme based on the Learning With Errors (LWE) problem [59] and its ring variant [53]. In this section, we describe its algebraic structures as well as its basic functioning for homomorphically evaluating arithmetic and arbitrary functions. TFHE was originally proposed using Torus notation, but we start 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.

2.4.1 Encryption scheme

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

LWE sample The sample itself is exactly as in Definition 2, but we reproduce it for convenience. An LWE sample[1] 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 discretized 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 $m = \lceil \mathsf{phase}(c)/\Delta \rfloor$.

RLWE sample Let $\mathcal{R}_q = \mathbb{Z}_q[X]/(X^N + 1)$ as defined in Section 2.1, 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 discretized 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.

RGSW sample An RGSW sample is a vector of 2ℓ RLWE samples. Algorithm 1 shows the encryption process, where ℓ and β are *gadget decomposition* parameters, and $\mathrm{RLWE\_Enc}(S, x)$ denotes the RLWE encryption of some message x with key S. 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 $\mathrm{RGSW}_{s,k}(\mathrm{m})$. Arithmetically, let C be a vector of 2ℓ RLWE samples of 0 (hence, it is a matrix in $\mathcal{R}_q^{2\ell\times 2}$), an RGSW encryption C' of a message m is given by $C' \leftarrow C + m \cdot G$, where $G \in \mathcal{R}_q^{2\ell\times 2}$ is the gadget decomposition matrix in Equation 2.1.

$$G = \begin{pmatrix} \beta^0 & 0 \\ \vdots & \vdots \\ \beta^{\ell-1} & 0 \\ 0 & \beta^0 \\ \vdots & \vdots \\ 0 & \beta^{\ell-1} \end{pmatrix} \tag{2.1}$$

[1] Formally, $(a, b) \in \mathbb{Z}_q^n \times \mathbb{Z}_q$, but, to ease the notation, we consider a flattened interpretation of (a, b) given by $f : \mathbb{Z}_q^n \times \mathbb{Z}_q \mapsto \mathbb{Z}_q^{n+1} = f(a, b) \mapsto (a_0, a_1, \ldots, a_{n-1}, b)$

Algorithm 1: RGSW Encryption (`RGSW_Enc`)

Input : An encryption key $S \in \mathcal{R}_q$
Input : A message $m \in \mathcal{R}_q$
Input : Decomposition parameters ℓ and $\beta \in \mathbb{Z}^+$
Output: An RGSW sample $c \in \mathrm{RGSW}_S(m)$

$C \leftarrow []$
for $i \in [\![0, \ell)$ **do**
  $C_i \leftarrow \mathrm{RLWE\_Enc}(S, 0)$
  $C_{\ell+i} \leftarrow C_{\ell+i} + (0, m\beta^i)$
  $C_i \leftarrow C_i + (m\beta^i, 0)$; `// Equivalent to:` $C_i \leftarrow C_i + (0, -K \cdot m\beta^i)$
return C

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 [4]. In this case, the Torus abstraction is not used.

2.4.1.1 Evaluating arithmetic

(R)LWE samples are in an $\mathfrak{R}$-module. Therefore, we have well-defined additions between samples and multiplications with elements from some ring $\mathfrak{R}$. In both cases, operations are pair-wise: Let $c_i = (a_i, b_i) \in \mathrm{(R)LWE}(m_i)$ for $i \in \{0, 1\}$ be two (R)LWE samples encrypting messages m_i. The sum of them is given by $c_{\mathsf{sum}} = (a_0 + a_1, b_0 + b_1) \in \mathrm{(R)LWE}_s(m_1 + m_2)$ while $c_{\mathsf{scale}} = (a_0 \cdot z, b_0 \cdot z) \in \mathrm{(R)LWE}_s(m_1 \cdot z)$ encrypts the scaling by $z \in \mathfrak{R}$, where $\mathfrak{R}$ is a ring (typically, $\mathbb{Z}$ or $\mathcal{R}$). For RGSW samples, these operations occur element-wise on their vector of RLWE samples.

(R)LWE samples also support external products by *RGSW samples*. Algorithm 2 describes the process. The decomposition of an RLWE sample in base β and ℓ levels is defined as a *gadget decomposition*. TFHE also defines an *approximate gadget decomposition* [22] that ignores the least significant bits of the RLWE sample. In this way, the multiplication is able to run with $\ell < \log_\beta(q)$, improving performance in exchange for some additional noise.

2.4.2 Building blocks for arbitrary computation

As we scale, add, or multiply samples, the Gaussian error in the component b increases. The bootstrap procedure, as first defined by Gentry [36], is the technique used for resetting this error to a default value established by the parameter set.

In TFHE, the bootstrap can be used not only for resetting the error but also to implement arbitrary (nonlinear) functions. For implementation, it defines three main building blocks, which we describe in this section.

Algorithm 2: RGSW × RLWE External Product

Input : An RLWE sample $c = (a, b) \in \text{RLWE}_S(m_1)$ encrypting message m_1
Input : An RGSW sample $C \in \text{RGSW}_S(m_2)$ encrypting message m_2
Input : Decomposition parameters ℓ and $\beta \in \mathbb{Z}^+$
Output: An RLWE sample $c' \in \text{RLWE}_S(m_1 m_2)$ encrypting the product of the messages, $m_1 m_2$

1 Let $\tilde{a}$ be the decomposion of a, such that, $a = \sum_{i=0}^{\ell-1} \tilde{a}_i \cdot \beta^i$
2 Let $\tilde{b}$ be the decomposion of b, such that, $b = \sum_{i=0}^{\ell-1} \tilde{b}_i \cdot \beta^i$
3 **return** $\sum_{i=0}^{\ell-1} C_i \cdot \tilde{a}_i + \sum_{i=0}^{\ell-1} C_{\ell+i} \cdot \tilde{b}_i$

2.4.2.1 Public and private key switching

The idea behind a key-switching algorithm is the homomorphic evaluation of the phase of a ciphertext. Let $c = (a, b) \in \text{(R)LWE}_{s,k}(m)$ be a (R)LWE sample encrypting m, the keyswitch algorithm uses an encryption of the secret key s, defined as $\text{KS}_i \in \text{(R)LWE}_{s',k'}(s_i)$, to calculate the $\mathsf{phase}(c) = b - \langle a, \text{KS} \rangle$. The result of this operation is $c' \in \text{(R)LWE}_{s',k'}(m)$, 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 3 shows the Public Key Switching algorithm from TFHE. Notice 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 3: Public Functional Key Switching (PublicKeySwitch) [22]

Input : p LWE samples $c^{(z)} = (a^{(z)}, b^{(z)}) \in \text{LWE}_s(\mu_z)$, $z \in [\![1, p]\!]$
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 $\text{KS}_{i,j} \in \text{(R)LWE}_{s'}(s_i 2^j)$, for $i \in [\![1, n]\!]$ and $j \in [\![1, t]\!]$
Output: an (R)LWE sample $c' \in \text{(R)LWE}_{s'}(f(\mu))$

1 **for** $i \in [\![1, n]\!]$ **do**
2 $\quad a_i \leftarrow f(a_i^{(1)}, a_i^{(2)}, ..., a_i^{(p)})$
3 $\quad \tilde{a}_i \leftarrow \left\lceil a_i \frac{2^t}{q} \right\rfloor$
4 $\quad$ Decompose $\tilde{a}_i$, such that $\tilde{a}_i = \sum_{j=1}^{t} \hat{a}_{i,j} \cdot 2^j$
5 **Return** $(0, f(b_i^{(1)}, b_i^{(2)}, ..., b_i^{(p)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \hat{a}_{i,j} \cdot \text{KS}_{i,j}$

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.*, $\text{KS} \in \text{(R)LWE}_{s',k'}(f(s))$. This version is especially useful when f depends on secret information, such as the key.Algorithm 4 shows the private key switching from TFHE.

Algorithm 4: Private Functional Key Switching (PrivateKeySwitch) [22]

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 $\mathrm{KS}_{i,j,z} \in (\mathrm{R})\mathrm{LWE}_{s'}(f_z(s)_i 2^j)$, for $i \in [\![1, n]\!]$, and $\mathrm{KS}_{n+1,j,z} \in (\mathrm{R})\mathrm{LWE}_{s'}(f(-1)2^j)$, for $j \in [\![1, t]\!]$ and $z \in [\![1, p]\!]$, where f_z are linear morphisms
Output: a (R)LWE sample $c_{out} \in (\mathrm{R})\mathrm{LWE}_{s'}(f(\mu))$

for $z \in [\![1, p]\!]$ **do**
  for $i \in [\![1, n]\!]$ **do**
    $\tilde{a}_{z,i} \leftarrow \left\lceil a_{z,i} \frac{2^t}{q} \right\rfloor$
    Decompose $\tilde{a}_{z,i}$, such that $\tilde{a}_{z,i} = \sum_{j=1}^{t} \hat{a}_{z,i,j} \cdot 2^j$
Return $-\sum_{z=1}^{p} \sum_{i=1}^{n+1} \sum_{j=1}^{t} \hat{a}_{z,i,j} \cdot \mathrm{KS}_{z,i,j}$

2.4.2.2 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 BlindRotate 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 5: Blind Rotation (BlindRotate) [22]

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 $c' \in \mathrm{RLWE}_S(X^{\lceil \mathsf{phase}(c) 2N/q \rfloor} \cdot v)$

ACC $\leftarrow X^{-\left\lceil b \frac{2N}{q} \right\rfloor} \cdot tv$
for $i \leftarrow 1$ ***to*** n **do**
  $\tilde{a}_i \leftarrow \left\lceil a_i \frac{2N}{q} \right\rfloor$
  ACC $\leftarrow \mathrm{CMUX}(C_i, X^{\tilde{a}_i} \cdot \mathrm{ACC}, \mathrm{ACC})$
return ACC

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

2.4.2.3 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.*, $\mathrm{SampleExtract}_j(c) \in \mathrm{LWE}_{s'}(m_j)$, where s' is the vector interpretation of s.

2.5 Lookup table (LUT) evaluation

Look-up tables are a convenient way of representing and evaluating functions using homomorphic encryption. There are several ways of doing so. In this section, we detail the basic procedures introduced by and used in schemes such as FHEW [30] and TFHE [22].

2.5.1 Leveled setting

Let L be a look-up table (LUT) of size l. A LUT evaluation on L with input (selector) x consists of looking up for the x-th element of L. Homomorphically, the simplest way of evaluating it is to encrypt L in an RLWE sample, with the value of each element of L being encrypted in a monomial of m. For example:

$$[m_0, m_1, m_2, m_3] \rightarrow \text{RLWE}(m_3 X^3 + m_2 X^2 + m_1 X^1 + m_0)$$

Then, the selector x is encrypted in the exponent of an RGSW sample as follows:

$$x \rightarrow \text{RGSW}(X^{2N-x})$$

To perform the look-up, we multiply the two encryptions. Let $x = 1$, the result of the multiplication would be:

$$\text{RLWE}(-m_0 X^3 + m_3 X^2 + m_2 X^1 + m_1)$$

Finally, we use the *SampleExtract* for extracting an LWE sample encrypting the constant term of the polynomial encrypted in the RLWE sample. In this way, we get the LUT evaluation result:

$$\begin{aligned} \mathit{SampleExtract}_0(\text{RLWE}(-m_0 X^3 + m_3 X^2 + m_2 X^1 + m_1)) \\ \rightarrow \text{LWE}(m_1) \end{aligned}$$

This is a toy example where we consider $l = N = 4$. Real parameters are typically much larger with $N > 1024$. While this solution is very efficient, with one RGSW $\times$ RLWE multiplication taking just a few tens of microseconds, it has some limitations. Firstly, LUTs are limited by the size of N. Secondly, the selector is encrypted in the exponent of an RGSW sample, which is often expensive to obtain in real-world scenarios [21]. These limitations are mitigated by techniques such as the *vertical packing* [22], which are based on the evaluation of CMUXes.

2.5.1.1 CMUX-based evaluation

As we defined in Algorithm 5, a CMUX [22] is a small multiplexer that receives two RLWE samples a and b, and an RGSW sample C encrypting 0 or 1. It returns a if $C = 0$, and b otherwise. Algorithm 6 shows the VerticalPacking [22], which essentially evaluates LUTs using a tree of CMUXes. In this method, the selector is encrypted bit by bit, which is usually easier or cheaper to obtain in real-world applications. It also enables evaluating

LUTs larger than N, as one can encrypt it in several RLWE samples and select the desired ones through a sequence of CMUX gates.

Algorithm 6: Vertical Packing (VERTICALPACKING) [22]

Input : a lookup table $L \in \mathbb{Z}_q^l$ of size $l = N^k$
Input : a list of samples $C_i \in \text{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$ encrypting a k-bit selector $s = \sum_{i=0}^{k-1} s_i 2^i$, bit by bit, for $k = \log_2(l)$
Output: a LWE sample $c' \in \text{LWE}_{S'}(L[s])$ encrypting the lookup result, where S' is the vector interpretation of S.

1 $f : \mathbb{Z}_q^N \mapsto \mathcal{R}_q$ given by $f(m) \mapsto \sum_{i=0}^{N-1} m_i X^i$
`/* Pack the LUT in RLWE samples */`
2 $v \leftarrow []$
3 **foreach** $i \in [\![0, l/N)$ **do**
4 $\quad v_i \leftarrow (0, f([L_{iN}, L_{iN+1}, \ldots, L_{(i+1)N-1}]))$
`/* Select the correct RLWE sample using a tree of CMUXes */`
5 **for** $i \leftarrow 0$ **to** $k - \log_2(N) - 1$ **do**
6 $\quad h \leftarrow \frac{l}{N2^{i+1}}$
7 $\quad$ **for** $j \leftarrow 0$ **to** $h - 1$ **do**
8 $\quad\quad v_j \leftarrow \text{CMUX}(C_{k-i-1}, v_j, v_{j+h})$
`/* Rotate the selected RLWE sample */`
9 **for** $i \leftarrow 0$ **to** $\log_2(N) - 1$ **do**
10 $\quad r \leftarrow v_0 \cdot X^{2N-2^i}$
11 $\quad v_0 \leftarrow \text{CMUX}(C_i, v_0, r)$
`/* Extract and return` $L[s]$ `*/`
12 **return** $\text{SAMPLEEXTRACT}_0(v_0)$

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

2.5.2 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.* [9] formalized the idea of a *functional bootstrap*, both for TFHE and other cryptosystems. In 2020, Chillotti *et al.* [23] introduced a discretized version of TFHE and defined the programmable bootstrapping (PBS), a formalization of the functional bootstrap specific to TFHE.

Algorithm 7 shows the functional bootstrap of TFHE.

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 2.2 details this process. The *Base* B is a precision parameter.

Algorithm 7: Functional Bootstrap (FUNCTIONALBOOTSTRAP) [9, 38, 23]

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 a vector (LWE) 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 $\text{C} \leftarrow \text{BLINDROTATE}((0, v), (a', b' + \frac{2N}{4B}), \text{BK})$
4 **return** $\text{SAMPLEEXTRACT}_0(\text{C})$

A single RLWE sample could encode up to N entries of a lookup table. However, when using the bootstrap to evaluate the LUT, only a small fraction of N will actually be available if the goal is exact computation. The first step of the bootstrap is to scale the ciphertext by $2N/q$ and round it to an integer. This process introduces a significant error variance, which is additive to the variance of the Gaussian error. To prevent these lookup errors, it is necessary to map each position of a LUT to a sequence of many consecutive coefficients of the *test vector* v and to adjust the selector to lookup positions in the middle of these sequences. For example, with $B = 4$ and $N = 1024$, an integer LUT $L = [l_1, l_2, l_3, l_4] \in \mathbb{Z}^4_B$ is mapped to $\sum_{i=0}^{255} l_1 X^i + \sum_{i=256}^{511} l_2 X^i + \sum_{i=512}^{767} l_3 X^i + \sum_{i=768}^{1023} l_4 X^i$ and we add a *precision offset* of $\frac{2N}{4B} = \frac{2N}{16}$ to b'.

$$L = [l_1 = f(1), ..., l_B = f(B)] \mapsto \Delta \sum_{i=0}^{N-1} l_{\lfloor \frac{iB}{N} \rfloor} X^i \tag{2.2}$$

The negacyclic property The table lookup is performed by using the BLINDROTATE to multiply the test vector by $X^{-\phi(c)2N/q}$, where ϕ denotes the phase. 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. In integer notation, this restricts $m\Delta \in \mathbb{Z}_q$ such that $0 < \Delta m < q/2$.

Evaluating encrypted LUTs and private functions Algorithm 7 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 an RLWE 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 [38]. 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.

2.5.3 Multi value bootstrap

The most expensive procedure in the LUT evaluation using the functional bootstrap of TFHE is the BlindRotate. The multi-value bootstrap is a technique that allows the evaluation of multiple LUTs with the same selector using just one BlindRotate. Suppose we want to evaluate z LUTs $(L_0, L_1, ..., L_z)$ with the same selector $c \in \text{LWE}_s(m)$ and we want to minimize the number of blind rotations. The most straightforward solution for implementing the multi-value bootstrap would be using the BlindRotate to calculate just $c' \in \text{RLWE}(X^{\lceil \mathsf{phase}(c)2N/q \rfloor})$ and, then, multiply it by each LUT (encoded in polynomials). This approach requires just one BlindRotate, but the error variance grows quadratically with the square norm of each LUT.

Carpov *et al.* [17] introduced a multi-value bootstrap scheme that allows for a much smaller error growth. As Algorithm 8 shows, the test vector is set to $(0, \sum_{i=0}^{N-1} \cdot \frac{q}{4N} \cdot \tau X^i)$, where τ is a scaling factor usually set as the `gcd` among the coefficients of all LUTs. After the blind rotation, each LUT (represented as a polynomial in $TV_{F_i} \in \mathcal{R}_q$) is divided by v, before being multiplied by the accumulator (ACC). This division greatly reduces the square norm of the LUTs and, hence, the error growth. Carpov *et al.* also introduced a very efficient way of calculating $\frac{TV_{F_i}}{v}$ that only requires a subtraction between each pair of consecutive coefficients of each LUT.

Algorithm 8: Multi-Value Functional Bootstrap (MVFB) [17]

Input : an LWE sample $c = (a, b) \in \text{LWE}_s(m)$, $m \in \mathbb{Z}_{2N}$
Input : a scale factor τ
Input : z LUTs encoded in polynomials $\mathsf{TV}_{F_i} \in \mathcal{R}_q$, for $i \in [\![1, z]\!]$
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 $i = 1, ..., z$, where $S' \in \mathbb{B}^N$ is a vector interpretation of S

$b' \leftarrow \lfloor b2N/q \rceil$ and $a' \leftarrow \lfloor a2N/q \rceil$
$v \leftarrow \sum_{i=0}^{N-1} \cdot \frac{q}{4N} \cdot \tau X^i$
$\text{ACC} \leftarrow \text{BlindRotate}((0, v), (a', b' + \frac{2N}{4B}), \text{BK})$
foreach $i \in [\![1, z]\!]$ **do**
  $c'_i = \text{SampleExtract}_0(\frac{\mathsf{TV}_{F_i}}{v} \cdot \text{ACC})$
return c'

Chapter 3

Evaluating arbitrary functions with high precision

The functional bootstrap is a great solution for evaluating low-precision functions, taking just a few milliseconds to evaluate, for example, the SIGN[1] function, as used by Bourse *et al.* [10] and Izabachène *et al.* [43]. Applications requiring higher precision, on the other hand, need to increase the parameters of the cryptosystem to keep the evaluation correct, which leads to deteriorated performance. For example, a 6-bit-to-6-bit LUT takes 1.5 seconds to be evaluated using TFHE's functional bootstrap, as implemented by Carpov *et al.* [17].

Considering this, in this work, we revisited the Functional Bootstrap of TFHE. In this section, we show how to evaluate functions with high precision using multiple functional bootstraps, but without increasing (too much) the parameters of the cryptosystem. This approach results in a much smaller impact on performance. The following contributions are presented.

- We introduce two new methods to combine multiple functional bootstraps in TFHE:
  - A tree-based one that allows the easy implementation of arbitrary functions, as well as a tree optimization based on particular properties of each function. We leverage the multi-value bootstrap of Carpov *et al.* [17] to lower the number of bootstraps in this method (asymptotically) from exponential to linear in the size of the input.
  - A chaining one that presents a better error rate growth behavior but is more intricate to implement depending on the target function.
- We perform an error variance analysis, including experimental validation, and a comparison between the aforementioned methods.
- We present optimizations to the building blocks used in our methods, which are also contributions of independent interest.

[1] $f : \mathbb{Z} \mapsto \mathbb{Z} = f(x) \mapsto 1$ if $x > 0, -1$ otherwise

  - We introduce a *multi-value extract* procedure that produces multiple LWE samples encrypting the same value with independent errors. It enables improving the error growth on ciphertext scaling from quadratic to linear with little performance and memory overhead. It also improves the error variance growth in the multi-value bootstrap [17] from quadratic to linear in the output base.
  - We introduce a *"base-aware"* Key Switching to pack $B < N$ LWE samples in an RLWE sample, where N is the polynomial size. In this work, B is the base of our integer encoding (thus, "base-aware"), but the technique enables gains of up to $\lfloor \frac{N}{B} \rfloor$ times for any $B < N$.

- We present implementations[2] of several relevant functions and compare their performance with state-of-the-art implementations from the literature.

Our methods speed up their evaluation up to 2.49 times, and, for specific functions, we also show possibilities of optimizations over the generic LUT evaluation. Compared to implementations using logic gates, we achieve speedups of up to 8.74 times in simple and useful functions, such as integer addition.

This chapter is organized as follows: Section 3.1 introduces the new methods we developed for combining functional bootstraps; Section 3.2 analyzes their error variance behavior; Section 3.3 presents optimizations to their building blocks; Section 3.4 presents a performance analysis of our methods, including comparison with the literature; Section 3.5 summarizes the related work; Finally, Section 3.6 discusses our results.

3.1 Combining functional bootstraps

To evaluate large LUTs considering the limitations we describe in Section 2.5.2, we either need to increase N (which increases the bootstrap time superlinearly [11]) or to lookup in multiple RLWE samples. In this section, we follow this latter approach and introduce two methods to combine multiple functional bootstraps to evaluate a single large LUT. Figure 3.1 illustrates the evaluation of an 8-bit parity function using them. In both methods, we decompose messages in base B with d digits and encrypt each digit in a different LWE sample. In this way, we are able to encrypt unbounded integers without increasing the parameters of the cryptosystem.

3.1.1 Tree-based method

Our first method to evaluate functions using multiple functional bootstraps is structured as a convergence tree and uses the output of a lookup to construct a new LUT. Algorithm 9 shows its final version. Let $c_i \in \mathrm{LWE}_s(m_i)$ for $i \in [\![0, d)$ be the selector, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the integer m in base B with d digits, and L be a B^d-sized LUT encoded in B^{d-1} RLWE samples. Notice that following the encoding we described in Section 2.5.2, each RLWE sample stores B elements of the LUT. Our first step is to

[2]The source code is available at `https://github.com/antoniocgj/FBT-TFHE`.

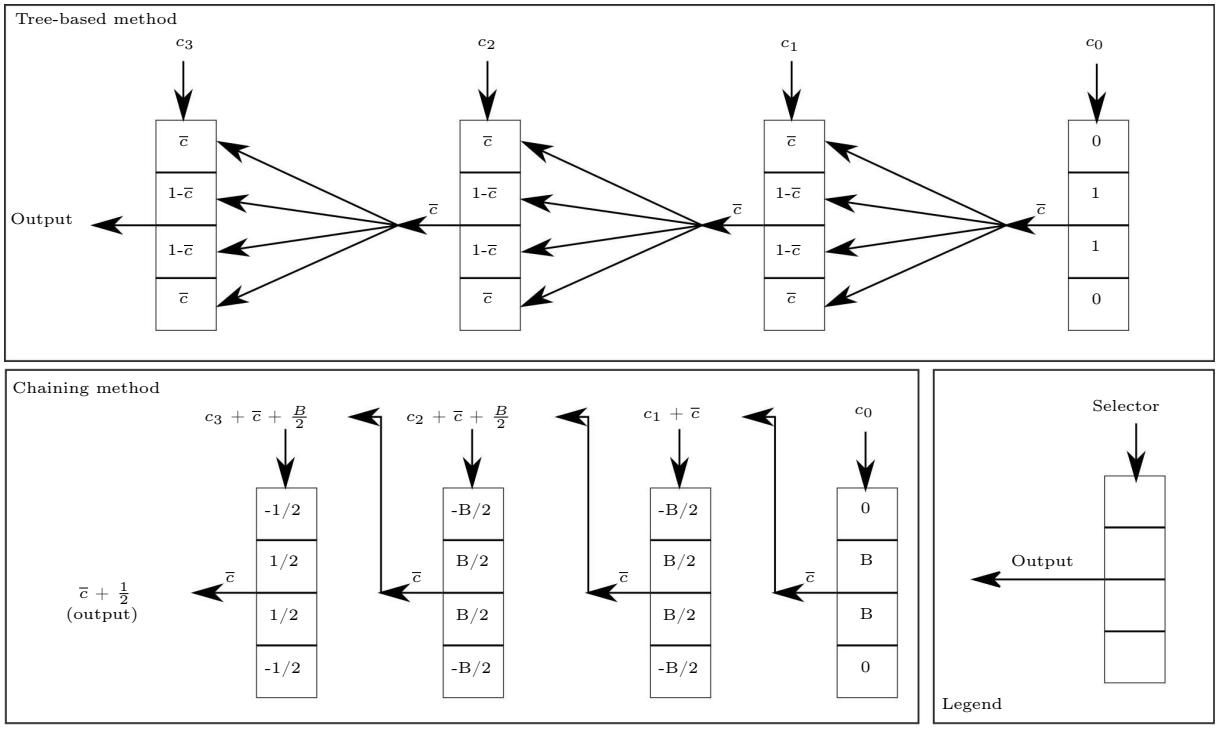


Figure 3.1: Evaluation of an 8-bit parity function using the tree-based and the chaining methods using base $B = 4$.

perform a functional bootstrap using the same selector c_0 on each of the B^{d-1} RLWE samples. This process results in B^{d-1} LWE samples. If $d = 1$, the evaluation is finished. Otherwise, we use a packing (LWE to RLWE) key switching to pack them in B^{d-2} RLWE samples. With that, we reduced our problem to the evaluation of a B^{d-1}-sized LUT encoded in B^{d-2} RLWE samples with selector $c_i \in \mathrm{LWE}_s(m_i)$ for $i \in [\![1, d)$ and we can recursively repeat this process until we have a single RLWE sample (as in $d = 1$). Figure 3.2a illustrates it for $B = 2^2 = 4$ and a LUT encoding the sigmoid function with 8 bits of precision (thus, $d = 4$ such that $B^d = 2^8$, achieving the 8-bit precision). Each rectangle represents a LUT, the arrows indicate the flow of data and c_i for $i \in [\![0, 3]\!]$ is the selector.

Without the multi-value bootstrap, the complexity of this process (measured in the number of functional bootstraps) would be exponential in the number of digits d. However, each level of the tree performs B^{d-1-i} bootstraps using the same selector c_i, thus allowing us to replace them with a single multi-value bootstrap, which reduces the complexity to linear in the number of digits. This complexity improvement results in similar performance gain asymptotically, but it faces practical problems for not-so-large LUTs. As we described in Section 2.5.3, the multi-value bootstrap relies on multiplications between the accumulator (ACC in line 5 of Algorithm 8) and the LUTs encoded as polynomials in $\mathcal{R}_q$. In the first level of our tree, this is not a problem since the LUTs are cleartext and can be encoded as polynomials in $\mathcal{R}_q$. Starting from level 1, the LUTs are now encrypted as RLWE samples. Arithmetically, this is not an obstacle, but the multiplication between ACC and the LUT is now a multiplication between two RLWE samples, which is

Algorithm 9: Tree-based Functional Bootstrap (TreeFB) [38]

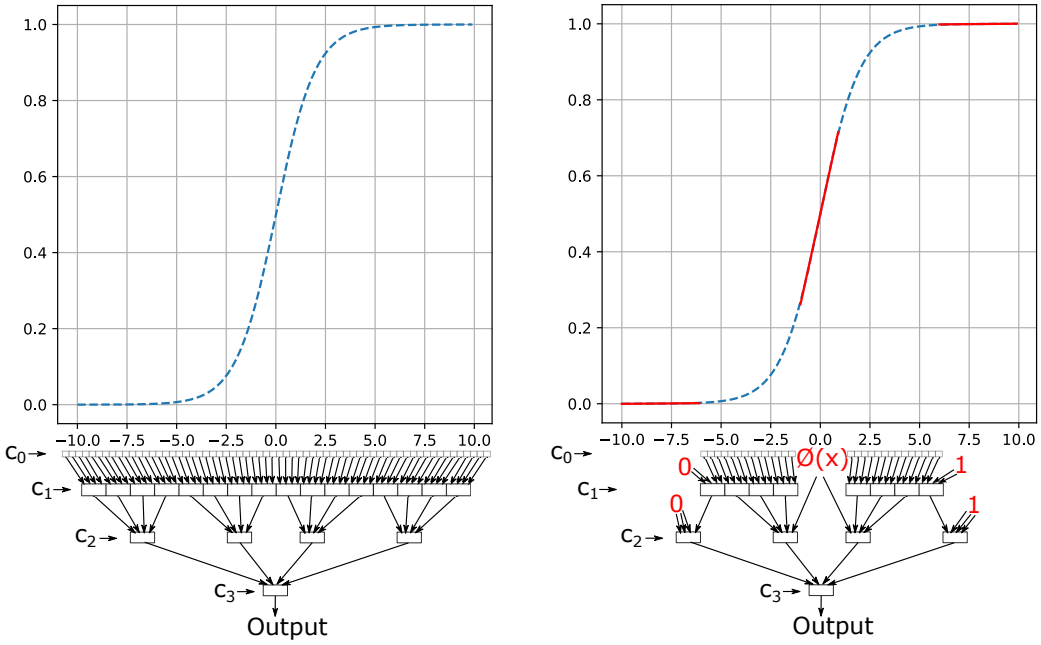
Input : a set of LWE samples $c_i \in \mathrm{LWE}_s(m_i)$, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the integer m in base B with d digits
Input : a set L of B^d polynomials $\in \mathcal{R}_q$ encoding the lookup table of an arbitrary function F
Input : a bootstrapping key $\mathrm{BK}_i \in \mathrm{RGSW}_S(s_i)$, for $i \in [\![1, n]\!]$
Input : a packing Key Switching key $\mathrm{KS}_{S' \mapsto S}$, where $S' \in \mathbb{B}^N$ is a vector (LWE) interpretation of S
Output: an LWE sample $c' \in \mathrm{LWE}_{S'}(F(m))$

```
TV ← L
f : Z_q^B ↦ R_q, given by f(m_0, ..., m_B) ↦ Σ_{i=0}^{N-1} m_{⌊iB/N⌋} X^i
for i ← 0 to d − 1 do
    c' ← MVFB(c_i, TV, BK)
    foreach j ∈ [[0, B^{d−i−2}) do
        TV_{j−1} ← PublicKeySwitch((c'_{(j−1)B}, ..., c'_{jB}), f, KS)
return c'_0
```

not directly performed in TFHE. We would need to convert ACC from RLWE to RGSW using techniques such as a Circuit Bootstrap [22], which would only be worth it if the number of Functional Bootstraps we are replacing is very large. In this chapter, most of our examples are 8-bit functions, and, therefore, we only use the multi-value bootstrap in the first level of the tree.

Circuit bootstrap implementation At the time we developed the techniques presented in this chapter, there was only an experimental implementation of the LWE-to-RGSW circuit bootstrap, and there was no RLWE-to-RGSW bootstrap. Naively, we could have adapted TFHE's algorithm to implement an RLWE-to-RGSW circuit bootstrap, which would have been N times more expensive. However, we could not consider such implementation a representative of RLWE-to-RGSW conversion performance. Practical implementations on this were mostly an open problem, and there were many recent developments in the literature [10, 19, 56] that could be used to achieve much more efficient conversions. The original (32-bit) implementation of TFHE does not support the circuit bootstrap, and, if we worked on another implementation of TFHE, it would be hard to compare with results from the literature. Considering all of that, we decided to leave the implementation of an RLWE-to-RGSW conversion as future work.

Optimizing the tree One of the main advantages of our method is its versatility. We can not only evaluate any function but also optimize the tree by considering the particular characteristics of each function. The sigmoid function, for example, has three intervals in its domain that could be linearly evaluated or approximated: $[\![-\infty, -6]\!] \approx 0$, $[\![6, \infty]\!] \approx 1$, and $[\![-1, +1]\!] \approx 0.24x + 0.5$. Figure 3.2b illustrates this optimization. $\varnothing(x)$ is the linear combination to calculate $f(x) = 0.24x + 0.5$ using integers (fixed-point representation). The output error of the optimized tree is not increased by approximations using cleartext literals (intervals $[\![-\infty, -6]\!] \approx 0$, $[\![6, \infty]\!] \approx 1$). The linear combination $\varnothing(x)$ increases the



(a) Full-tree (b) Optimized tree

Figure 3.2: Evaluation of an 8-bit sigmoid function using the tree-based with base $B = 4$.

error if its results are not bootstrapped, which is something to consider when composing functions to make an application. Considering security, the implications of optimizing the tree are limited to secondary aspects that are not in our scope. For example, the full-tree design enables us to easily achieve circuit privacy by simply encrypting the initial LUT, whereas tree optimizations could give partial information about the function.

3.1.2 Chaining method

This second method is a generalized version of the integer comparison algorithm presented by Bourse *et al.* [11]. It is much more functionally restricted than the first method, but it usually presents a smaller error growth. Its main characteristic is that the output of a lookup is used to construct the selector of the next lookup, whereas, in the tree-based approach, the output is used to construct the next LUT. This difference has deep implications for error propagation and overall functionality. Consider a selector $c_i \in \mathrm{LWE}_s(m_i)$ for $i \in [\![0, d)$ with d digits. The first functional bootstrap uses c_0 as the selector and outputs $\bar{c}_0$. From then on, each evaluation uses the selector $\oslash(c_i, \bar{c}_{i-1})$ for $i \in [\![1, d)$, where $\oslash$ is a linear combination. We can define this method as being functionally capable of evaluating any function that can be encoded in LUTs such that, for each digit, either the output of $\oslash(c_i, \bar{c}_{i-1})$ is smaller than the size of the LUT, B, or the function being encoded follows a B-anti-cyclic [9] logic. Although it is hard to generically define functions with such restrictions, this method seems to be especially good for functions that require carry-like logic, such as additions and multiplications, as we show in Section 3.4.1.3.

3.2 Error analysis

In this section, we analyze the error variance growth of each algorithm used to perform a functional bootstrap and we calculate the overall probability of error based on the final error variance. We start by reproducing two equations from Chillotti *et al.* [22], which we adapt from integer notation. Equation 3.1 and 3.2 show the variance resultant of the key switching and the gate bootstrap procedures, respectively. The underbar indicates input variables; ϑ_{KS} and ϑ_{BK} are, respectively, the variance of the bootstrap and key switching keys, and $\epsilon = \frac{q}{2\beta^\ell}$ and β are the gadget decomposition quality and base, also respectively. All other variables were introduced at the beginning of Section 2.4.

$$Var(Err(c)) \leq R^2 Var(Err(\underline{c})) + \underline{n}tN\vartheta_{KS} + q\underline{n}2^{-2(t+1)} \tag{3.1}$$

$$Var(Err(c)) \leq \underline{n}(k+1)\ell N \left(\frac{\beta}{2}\right)^2 \vartheta_{BK} + \underline{n}(1+kN)\epsilon^2 \tag{3.2}$$

In the chaining method, functional bootstraps have the same output error variance as the gate bootstrap (Equation 3.2) since both use noiseless *test vectors*. In the tree-based method, the *test vector* is an RLWE sample that might be encrypting the result of previous table lookups and, hence, we need to consider its error variance, which is additive with the one introduced by the bootstrap, giving us Equation 3.3. Considering the multi-value bootstrap, Carpov *et al.* presents Equation 3.4. Up to line 3 of Algorithm 8 the variance is the same as the single value bootstrap, but the multiplication $\frac{TV_{F_i}}{v} \cdot \text{ACC}$ in line 5 multiplies the variance of the bootstrap by $\|TV_f\|_2^2 \leq s(q-1)^2$, where s and q are the input and output bases, respectively.

$$Var(Err(c)) \leq Var(Err(TV)) + \underline{n}(k+1)\ell N \left(\frac{\beta}{2}\right)^2 \vartheta_{BK} + \underline{n}(1+kN)\epsilon^2 \tag{3.3}$$

$$Var(Err(c)) \leq \|TV_f\|_2^2(\underline{n}(k+1)\ell N \left(\frac{\beta}{2}\right)^2 \vartheta_{BK} + \underline{n}(1+kN)\epsilon^2) \tag{3.4}$$

As for the key switching algorithm, Chillotti *et al.* [22] only analyzes it using binary decomposition and TFHE only implements the LWE-to-LWE key switching. In this work, we use both the LWE-to-LWE key switching of TFHE and a packing (LWE-to-RLWE) key switching to pack LWE samples in an RLWE sample. This packing key switching is a core algorithm of our tree-based approach and, to improve efficiency, we need to use greater bases for decomposition. Considering that, we reanalyzed the key switching error variance introduction. Equation 3.1 presents three terms. The first term, $R^2 Var(Err(\underline{c}))$, comes from possible scalings performed by the linear function f (we only use 1-Lipschitz functions, therefore $R^2 = 1$). The second term, $\underline{n}tN\vartheta_{KS}$, comes from the addition of $(\underline{n}tN)$ (R)LWE samples (the summation in line 5 of Algorithm 3), each of them with variance ϑ_{KS}. The third term, $q\underline{n}2^{-2(t+1)}$, is the variance introduced by the rounding of the binary decomposition. Chillotti *et al.* defines the error variance starting from the error amplitude: Each of the $\underline{n}$ elements of the vector $\underline{a}$ of the LWE input $\underline{c}$ are rounded

to the closest multiple of $q/2^t$, which introduces an error of at most $q|\frac{2^{-t}}{2}| = q2^{-(t+1)}$. The variance is then calculated by squaring this amplitude and multiplying it by $\underline{n}$, *i.e.*, $|q\frac{2^{-t}}{2}|^2 \cdot \underline{n} \equiv q\underline{n}2^{-2(t+1)}$. To change the decomposition base, we can replace 2 with an arbitrary base β_{ks}, which gives us $\frac{q}{4}\underline{n}\beta_{ks}^{-2t}$. Albeit correct, this arithmetic approach is not as tight as desired.

The vector $\underline{a}$ of the input LWE sample is generated from a uniform distribution. Hence, the error inserted by rounding each of its elements to $q/2^t$ is also uniform and varies from $-\frac{q\beta_{ks}^{-t}}{2}$ to $+\frac{q\beta_{ks}^{-t}}{2}$. The variance of a uniform distribution varying from a to b is $\frac{1}{12}(a-b)^2$ and the sum of $\underline{n}$ uniformly distributed variables are a (scaled) Irwin-Hall distribution with variance $\underline{n} \cdot \frac{1}{12}(a-b)^2$. Applying these equations to our case, we have that the error variance introduced by the decomposition is $\underline{n} \cdot \frac{1}{12}(-\frac{q\beta_{ks}^{-t}}{2} - \frac{q\beta_{ks}^{-t}}{2})^2 = \frac{1}{12}q\underline{n}\beta_{ks}^{-2t}$. Irwin-Hall distributions are very good approximations for Gaussian distributions and we can just replace the third term of Equation 3.1, which results in Equation 3.5.

$$Var(Err(c)) \leq R^2 Var(Err(\underline{c})) + \underline{n}tN\vartheta_{KS} + \frac{1}{12}q\underline{n}\beta_{ks}^{-2t} \tag{3.5}$$

3.2.1 Error rate

Knowing the error variance is useful for approximate computation, but, for exact execution, we need to calculate the probability of such error affecting significant bits of the message. In the decryption, a failure occurs if the rounding procedure rounds to the wrong integer, which will happen if the absolute value of the error is greater than half of the least significant bit of the message in the Torus representation. Equation 3.6 gives us the probability of this error occurring for an LWE sample x, with standard deviation σ_x and the least significant bit encoded in the torus with value $(2 \cdot \mathsf{interval})$. erf is the Gaussian error function.

$$P(|err(x)| > \mathsf{interval}) = 1 - erf\left(\frac{\mathsf{interval}}{\sigma_x\sqrt{2}}\right) \tag{3.6}$$

The bootstrap may also fail since the error affects the accumulator blind rotation amount. The failure occurs if the accumulator is rotated to a polynomial in which the coefficient of the constant term is different from the desired one. In this way, we need the error to be within the $\mathsf{interval}$ of half the scaling factor $\Delta = \frac{q}{2B}$ we are working with, which is $\frac{1}{2} \cdot \frac{q}{2B}$. Besides the LWE error variance, we also need to consider the rounding error introduced when selecting the $log_2(2N)$ most significant bits of each position of a. The discarded bits of each position are uniformly distributed with values ranging from $-\frac{q}{4N}$ to $+\frac{q}{4N}$. The variance of a uniform distribution is $\frac{1}{12}$ times the square of its amplitude, thus $\sigma_{a_i}^2 = \frac{1}{12} \cdot \frac{q}{2N}^2$, for each $a_i \in a$. In the worst case, n positions will be added to calculate the phase, leading to variance $\sigma^2_{\sum_i^n a_i} = n \cdot \sigma^2_{a_i}$. This variance is additive with the one of the Gaussian noise, and we can obtain the error rate using Equation 3.6.

The bootstrap is much more susceptible to failures than the decryption. Therefore, the error variance of LWE samples that are input for bootstraps ends up being the main component to control the error rate. Given a function that can be evaluated by both of our methods, we can define which one presents the better error rate in function of such

variance. Let $\oslash$ be the linear operation of the chaining method and s be its error variance scaling, *i.e.*, how many times the error variance of the output of $\oslash$ is greater than the one of the inputs. Figure 3.3 shows the error rate in function of the LWE input error variance for 8-bit functions in base 4 for $s \in \{2, 4, 6, 10\}$. We found few practical cases with $s > 2$, but, in all of them, the tree-based approach required more bootstraps, and s could be lowered to 2 by increasing the number of bootstraps of the chaining one. In this way, we conclude that the chaining method is usually the better choice for the functions it can evaluate. However, as we noted before, its functionality is very restricted. Table 3.1 summarizes the main characteristics of our two methods.

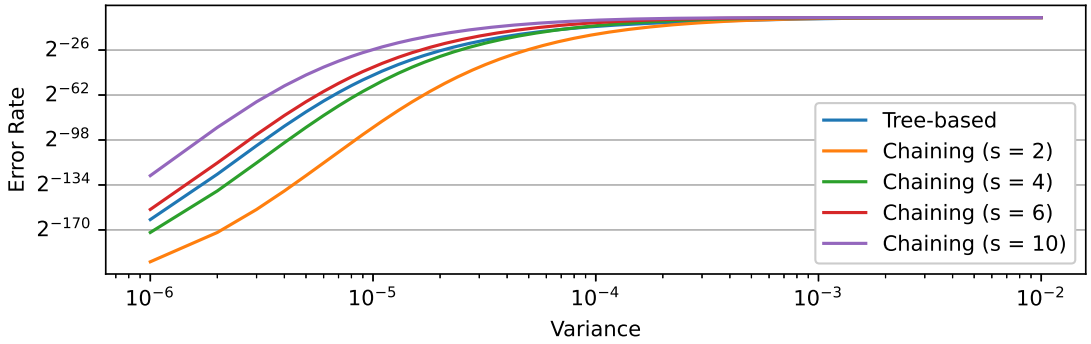


Figure 3.3: Comparison between the error rates of the tree-based and chaining methods.

3.3 Improving building blocks

3.3.1 Base-aware LWE-to-RLWE key switching

A regular packing (LWE-to-RLWE) key switching packs N LWE samples in one RLWE sample in $\mathcal{R}_q$. However, as we described in Section 2.5.2, we want to pack B LWE samples with each one being mapped to a sequence of consecutive coefficients in the RLWE sample. In Algorithm 10, we exploit the fact that $B < N$ to both accelerate the key switching and to reduce the error variance growth in $\frac{N}{B}$ times compared to the regular LWE-to-RLWE Key Switching. Comparing with Algorithm 3, we obtain speedups by replacing multiplications between N-sized polynomials (line 5 of Algorithm 3) by inner products between a B-sized vector of binary digits and a B-sized vector of RLWE samples (line 7 of Algorithm 10). The Key Switching key, however, is B times bigger.

3.3.2 Multi-value extract

The error variance growth of adding two variables x and y is defined as $\sigma^2_{x+y} = \sigma^2_x + \sigma^2_y + 2\rho\sigma_x\sigma_y$, where σ^2 is the variance and ρ is the correlation between the normally distributed variables. When this correlation is polynomial, ρ may be defined as its degree. If the variables are completely independent, then $\rho = 0$ and $\sigma^2_{x+y} = \sigma^2_x + \sigma^2_y$. If they are the same variable, then $\rho = 1$ and $\sigma^2_{x+x} = \sigma^2_x + \sigma^2_x + 2\sigma_x\sigma_x = 4\sigma^2_x$. Defining the multiplication as a sequence of additions of the same variable, we have that $\sigma^2_{n\times x} = n^2 \times \sigma^2_x$, for $n \in \mathbb{Z}$.

Table 3.1: Comparison between the chaining and tree-based methods. We use overbars to indicate output variables, $\oslash$ to represent linear transforms, and ":" to denote digit slicing. The function *prob* is given by Equation 3.6, d is the number of digits, s is the scaling factor of the error variance in a linear transform (*i.e.*, how much the linear transform increases the error variance), and σ^2 represents error variances (specifically, σ^2_{BT} and σ^2_{KS} are the output error variance of the bootstrap and RLWE Key switching, respectively).

	Chaining	Tree-Based
Supported Functions	(recursive definition) Let f be a function over an operand x with d digits. For each digit x_i for $i \in [0, d)$, there shall exist a linear combination $\oslash_i$ and a LUT L_i, s. t.: if $i = 0$, $\overline{x}_0 = L_i(x_0) = \oslash_f(f(x_0))$, if $i \geq 1$, $\overline{x}_i = L_i(\oslash_{i-1}(\overline{x}_i - 1, x_i))$ $= \oslash_f(f(x_i : x_0))$.	Any.
Suitable Functions	Most functions following carry-like (*e.g.* addition) or test (*e.g.* sign) logics.	All other functions.
Error var. σ^2_{out}	$s_{\oslash_f} \cdot \sigma^2_{BT}$	$(d-1) \cdot \sigma^2_{KS} + d \cdot \sigma^2_{BT}$
Success Rate	$\prod_{i=0}^{d-1} prob(\sigma^2_{in} \cdot s_{\oslash_i})$	$prob(\sigma^2_{out}) \cdot prob(\sigma^2_{in})^d$
Possible improvements and tradeoffs	The scaling factors, s, can be lowered by bootstrapping intermediate results of the linear combinations.	Optimizations in intervals of the function domain which are mapped to constant values or linear combinations.
Complexity in number of bootstraps	Linear in d.	Linear in d (Assymtoptically).

To avoid the quadratic variance growth at multiplications, we could implement them as sequences of additions of independent LWE samples encrypting the same number. We obtain these independent encryptions by extracting multiple coefficients from the accumulator (ACC) at the end of a bootstrap procedure. We call this process *Multi-Value Extract*. Recall that, in our LUT encoding (Section 2.5.2), each LUT position is mapped to a sequence of $\frac{N}{B}$ coefficients. Therefore, after the bootstrap, we should have $\frac{N}{B}$ independent encryptions of each number, which we can use to perform the multiplication as shown in Algorithm 11. The additional extracts on the ACC reduce the interval used to calculate the error rate in Equation 3.6 from $\frac{q}{4B}$ to $(\frac{q}{4B} - \frac{qz}{4N})$. However, we find this reduction to have a negligible impact on the error rate for the values we tested. We sustain the independence between coefficients of the ACC on the Independence Heuristic [22] (Definition 3).

Definition 3 (Independence Heuristic [22])**.** *The error of the coefficients of RLWE samples (including RGSW samples) and all linear combinations of them considered in TFHE are independent and concentrated.*

We tried to experimentally validate the error variance of the multiplication using the multi-value extract, and we obtained the results in Figure 3.4a. We noticed that the error variance is still growing quadratically, which indicates that the coefficients are not independent. To obtain formal guarantees of independence (instead of a heuristic),

Algorithm 10: Base-aware LWE-to-RLWE Public Functional Key Switching

Input : B LWE samples $\underline{c}^{(z)} = (\underline{a}^{(z)}, \underline{b}^{(z)}) \in \mathrm{LWE}_{\underline{s}}(\mu_z)$, $z \in [\![1, B]\!]$
Input : a precision parameter $t \in \mathbb{Z}$
Input : a Key Switching key $\mathrm{KS}_{i,j,b} \in \mathrm{RLWE}_{\overline{S}}(\underline{s}_i \beta_{ks}^j \cdot \sum_{q=bN/B}^{(b+1)N/B-1} X^q)$, for $i \in [\![1, n]\!]$, $j \in [\![1, t]\!]$, and $b \in [\![0, B)$.
Output: a RLWE sample $\overline{c} \in \mathrm{RLWE}_{\overline{S}}(f(\mu_z))$, for $z \in [\![1, B]\!]$.

$f : \mathbb{Z}_q^B \mapsto \mathcal{R}_q$, given by:
$f(m_0, ..., m_B) \mapsto \sum_{i=0}^{N-1} m_{\lfloor \frac{iB}{N} \rfloor} X^i$
foreach $i \in [\![1, n]\!]$ **do**
  foreach $b \in [\![1, B]\!]$ **do**
    $\tilde{a}_{i,b} \leftarrow \lceil \underline{a}_i^{(b)} 2^t / q \rfloor$.
  Decompose $\tilde{a}_{i,b}$, such that $\tilde{a}_{i,b} = \sum_{j=0}^{t-1} \hat{a}_{i,j,b} \cdot \beta_{ks}^j$
Return $(0, f(\underline{b}_i^{(1)}, \underline{b}_i^{(2)}, ..., \underline{b}_i^{(B)})) - \sum_{i=1}^{n} \sum_{j=1}^{t} \langle \hat{a}_{i,j}, \mathrm{KS}_{i,j} \rangle$

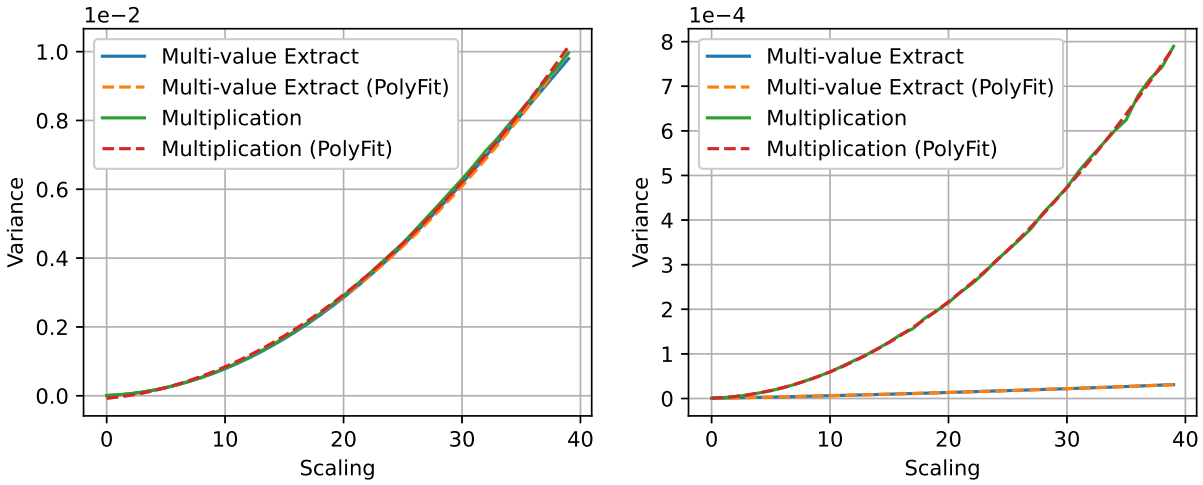
Algorithm 11: Multiplication (Scaling) using the Multi-Value Extract (MultiValueExtractScaling) [38]

Input : an RLWE sample $c \in \mathrm{RLWE}_S(\mathrm{p})$, which is the accumulator (ACC) of a previous functional bootstrap, and a cleartext scalar $z \in \mathbb{Z}$
Output: an LWE sample $c' \in \mathrm{LWE}_{S'}(z \cdot p_0)$, where p_0 is the constant term of p, and $S' \in \mathbb{B}^N$ is a vector interpretation of S

$c' \leftarrow \mathrm{LWE}_S(0)$
for $i \leftarrow 0$ **to** $\lceil \frac{z}{2} \rceil - 1$ **do**
  $c' \leftarrow c' + \mathrm{SampleExtract}_i(p)$
for $i \leftarrow N - \lfloor \frac{z}{2} \rfloor$ **to** $N - 1$ **do**
  $c' \leftarrow c' - \mathrm{SampleExtract}_i(p)$
Return c'

Chillotti *et al.* [22] points out that we could perform the gadget decomposition in a probabilistic way. We implemented the probabilistic gadget decomposition proposed by Genise *et al.* [35], but we obtained no improvements over the deterministic algorithm. We were only able to obtain the linear growth by lowering the error variance introduced by the gadget decomposition. We increased the size of the decomposition base $log_2(\beta)$ from 4 to 5, which improves its precision $\ell log_2(\beta)$ from 20 to 25 bits, a reasonably high value for a 32-bit implementation. Figure 3.4b shows the results.

From this experiment, we conclude that, although the independence heuristic holds for the Gaussian error, we could not verify the same behavior for the error introduced by the gadget decomposition. We also measured that the error variance introduced by the decomposition is bigger than the estimates using the right term of Equation 3.2, $\underline{n}(1 + kN)\epsilon^2$. We consider it as an indication that this dependency between coefficients might affect not only our multi-value extract but also the bootstrap itself. These conclusions are, however, specific to the original implementation of TFHE, and further research would be necessary to determine whether these claims apply to the scheme itself. The use of a probabilistic gadget decomposition with higher entropy might be an alternative solution



(a) 20-bit precision ($\ell = 5$ and $log_2(\beta) = 4$))

(b) 25-bit precision ($\ell = 5$ and $log_2(\beta) = 5$)

Figure 3.4: Comparison between the variance of scaling using the multi-value extract and direct multiplication.

to the increase of ℓ or β. However, in our case, it would require us to increase other parameters, which would impact performance.

Impact on the multi-value bootstrap The multi-value bootstrap of Carpov *et al.* [17] increases the bootstrap error variance in $\|TV_f\|_2^2 \leq s(q-1)^2$ times, where s and q are the input and output bases, respectively. Carpov *et al.* uses $q = 2$ (the binary base) to lower the error, but it needs to convert from base q to s to use the output in another function. It does so by using a key switching at the beginning of each functional bootstrap to perform a base composition, which is s-Lipschitz, and, therefore, introduces the previously avoided quadratic error. In summary, composable circuits often need the input and output bases to be the same, and, in these cases, solely outputting in the binary base would just transfer the complexity to the base composition. However, with the introduction of the multi-value extract, we can perform the scaling required by the base composition linearly and, thus, reduce the complexity of error variance from $s(q-1)^2$ to $s(q-1)$.

3.4 Experimental results

Our first step to define practical parameters was to survey the literature on non-linear functions that are usually implemented using exact computation with TFHE. Then, based on the required precision, we manually searched for parameters aiming at an error rate of at most 2^{-30}. Table 3.2 shows two of the most promising sets we found. A more methodological search could likely yield parameters with a better error rate or efficiency. The security level is defined by the (R)LWE dimensions (n and N) and the standard deviations of the errors. The LWE Estimator [3] reports a cost of $2^{127.1}$ on the primal attack via uSVP and $2^{139.6}$ on the dual-lattice attack, both using the `BKZ.sieve` cost model. Although we could increase the security level by increasing the parameters, we use these values since they are the same used by TFHE and most of the previous literature.

Table 3.2: Sets of parameters used to evaluate the performance of our implementations.

Name	Security Level	B	LWE		RLWE			Bootstrap		Key Switch	
			n	σ/q	N	k	σ/q	ℓ	$log_2(\beta)$	β	t
5_5_6_2	127	4	630	2^{-15}	1024	1	2^{-25}	5	5	2^6	2
6_4_6_3								6	4		3

We estimated the error rate for parameters using the equations of Section 3.2, and we experimented to validate their results. We find this validation to be necessary mainly because we based our error analysis on equations designed for TFHE to work with binary digits and logic gates. Once we introduced larger bases with arbitrary LUT evaluation and tightened some of the variance estimates, we could no longer support our statistics on the experimental validation provided by Chillotti *et al.* [22]. Table 3.3 shows the results of our experiments. We measure the variance of 2^{14} samples and calculated a 95%-confidence interval using the Chi-square distribution. Due to computational limits, we could only validate the error rate for higher variances. We performed 15,438,720 bootstraps over LWE samples with σ^2 = 1.70E-04. The output was wrong in only 39 of them. In both cases, the experiments showed that our estimates are reasonably close upper bounds for the actual values. Although we cannot extrapolate the results for different variances, the experiments provide some evidence of correctness for our equations on the considered parameters.

Table 3.3: Experimental validation for the variance and error rate for two sets of parameters.

Params.	Measurement		Estimative	Experiment		
				Measured	95% Conf. Interval	
5_5_6_2	Variance	Bootstrap	1.47E-06	4.94E-07	4.84E-07	5.05E-07
		RLWE KeySwitch	5.09E-06	2.53E-06	2.47E-06	2.58E-06
	Error rate (log_2)	σ^2 = 1.70E-04	-18.001	-18.595		
6_4_6_3	Variance	Bootstrap	4.41E-07	1.70E-07	1.67E-07	1.74E-07
		RLWE KeySwitch	1.25E-09	6.38E-10	6.25E-10	6.52E-10

3.4.1 Performance

We benchmarked our methods by using them to implement a set of relevant functions from the literature. We selected two functions from previous literature on the functional bootstrap of TFHE and three functions that are building blocks for neural network algorithms. We set the precision of our implementations to match the ones we are comparing to. We executed our experiments using an Intel i7-7700 processor at 4.20 GHz running Ubuntu 18.04. We compiled our code using GCC 7.3.0 with flags `-O3 -std=c++11 -funroll-all-loops -march=native -ltfhe-spqlios-fma -lm`, and we used the *"optim"* build of TFHE. Each result is an average of 100 executions. We tried to compile and execute previous implementations on the same environment. When we could not reproduce them (either because the authors did not provide the source code or because their source code did not compile in our environment), we report the results presented by

the authors and add an observation about the differences between machines. Fortunately, most authors report the execution time for the default gate bootstrap of TFHE, which we can use as a "TFHE benchmark score" and adjust the speedup values accordingly. In our machine, the default gate bootstrap of TFHE runs in 9ms and 13ms using the old (80-bit security) and the new (127-bit security) set of parameters, respectively. We estimated the error rates based on the equations provided in Section 3.2 using at most 500 bits of precision. Errors rates that could not be estimated within this precision are labeled as negligible.

3.4.1.1 Lookup table evaluation

Lookup tables are very commonly used in homomorphic circuits, and the implementation of Carpov *et al.* [17] is one of the most recent and efficient of them. It introduces the multi-value bootstrap of TFHE, which we explore in our tree-based method. Table 3.4 compares their implementation for a 6-bit-to-6-bit LUT with one using our tree-based method (Algorithm 9). Both implementations are based on the functional bootstrap of TFHE. The difference is that the implementation of Carpov *et al.* [17] performs a single functional bootstrap with base $B = 2^6$, whereas our implementations perform several functional bootstraps with base $B = 2^2$ and combine them using the tree-based method.

Table 3.4: Comparison between implementations of a 6-bit-to-6-bit LUT.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[17]	≥128	≈ 8 GB	-26.94	1570[a]	1.00
5_5_6_2	127	≈ 4.3 GB	-59.59	378.2	2.49
6_4_6_3	127	≈ 6.5 GB	-134.84	457.9	2.06

[a] Result provided by the authors, who executed experiments on a machine 1.67 times slower than ours. The speedup was adjusted accordingly.

3.4.1.2 32-bit integer comparison

Integer comparison is an extremely useful function in computing, but it presents some challenges to be implemented in homomorphic circuits, especially for unbounded integers. Bourse *et al.* [11] presented a very efficient implementation using the functional bootstrap of TFHE. The chaining method we presented in Section 3.1.2 is a generalization of their technique. To compare with them, we implemented a unary tree to evaluate the integer comparison. Algorithm 12 describes our implementation. Table 3.5 compares them with the integer comparison implemented by Zhou *et al.* [67] using logic gates.

We calculated the speedups using as reference the slowest implementation, which, in this case, is the second implementation of Bourse *et al.* [11]. However, adjusting the speedups to consider the difference in speed between machines, we can see that the implementation of Zhou *et al.* [67] using logic gates is up to 1.75 times slower than the one of Bourse *et al.* [11] and up to 5.6 times slower than ours. The chaining and tree-based methods perform the same number of bootstraps and should present a very similar performance. Nonetheless, we were able to achieve significant speedups thanks to our tighter variance and error rate estimates, which enabled a better choice of parameters.

Algorithm 12: Integer comparison algorithm using the tree-based method.

Input : two sets of LWE samples $c_{j,i} \in \text{LWE}_s(m_{j,i})$, such that $\sum_{i=0}^{d-1} m_{j,i}B^i = m_j$ encodes each number m_j for $j \in \{0,1\}$ and base $B = 4$ with d digits.
Input : a bootstrapping key BK and an RLWE Key Switching key KS
Output: an LWE sample $\overline{c} \in \text{LWE}_{\overline{S}}(\overline{m})$, where $\overline{S}$ is a vector (LWE)

$$\text{interpretation of } S \text{ and } \overline{m} = \begin{cases} 1, & \text{if } m_0 > m_1, \\ 0, & \text{if } m_0 = m_1, \\ -1, & \text{otherwise.} \end{cases}$$

$f : \mathbb{Z}_q^B \mapsto \mathcal{R}_q$, given by $f(m_0, ..., m_B) \mapsto \sum_{i=0}^{N-1} m_{\lfloor \frac{iB}{N} \rfloor} X^i$
TV $\leftarrow [0,1,1,1]\Delta$
for $i \leftarrow 0$ **to** $d-1$ **do**
  $c_i \leftarrow c_{0,i} - c_{1,i}$
  $\overline{c} \leftarrow$ FunctionalBoostrap$(c_i, \text{TV}, \text{BK})$
  TV $\leftarrow$ PublicKeySwitch$([\overline{c}, (0,1), (0,1), (0,1)], f, \text{KS})$
return $\overline{c}$

Table 3.5: Comparison between implementations of a 32-bit integer comparison.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[11]	90	≈ 1.2 GB	-50[b]	2232[a]	1.75
	109	≈ 3.4 GB	-47[b]	3902[a]	1.00
	211	≈ 4.6 GB	-89[b]	3840[a]	1.02
[67]	80	≈ 0.3 GB	Negligible	1143.2	0.93
	127	≈ 0.3 GB	Negligible	1867.2	0.57
5_5_6_2	127	≈ 4.3 GB	-26.51	334.1	3.19
6_4_6_3	127	≈ 6.5 GB	-129.58	396.4	2.68

[a] Execution time provided by the authors, who executed experiments on a machine 3.67 times slower than ours. The speedup was adjusted accordingly.
[b] Error Rate provided by the authors. We speculatively estimate it to be much lower, but we do not have sufficient data to calculate.

3.4.1.3 Neural network functions

We implemented three functions that are building blocks for Neural Network implementations and that are provided by SHE [52], an implementation of secure neural network inference based on TFHE that achieves state-of-the-art inference accuracy. SHE presents very fast arithmetic (thanks to the use of Lognet), but it relies on logic gates to implement non-linearities. We also compare our results with the implementations of Zhou *et al.* [67], which are generally faster than SHE but present worse inference accuracy.

ReLU The Rectified Linear Unit (ReLU) is a neural network activation function broadly used due to its simple implementation and non-linear properties. Algorithm 13 shows our implementation using the Functional Bootstrap. The logic is similar to a multiplexer. Table 3.6 compares it with implementations using logic gates. For the 127-bit security level, our implementations are up to 6.98 times faster than the one of Lou and Jiang [52] and 1.19 times faster than the one of Zhou *et al.* [67]. Although the logic gates of TFHE generally introduce error rates much smaller than the functional bootstrap, the error rate

of our implementation is 2^{-137}, which is also negligible compared to the security level.

Algorithm 13: ReLU implementation using the tree-based method.

Input : a set of LWE samples $c_i \in \text{LWE}_s(m_i)$, such that $\sum_{i=0}^{d-1} m_i B^i = m$ encodes the integer m in base $B = 4$ with d digits.
Input : a bootstrapping key BK and an RLWE Key Switching key KS
Output: A set of LWE sample $\overline{c}_i \in \text{LWE}_{\overline{S}}(\overline{m}_i)$, where $\overline{S}$ is a vector (LWE) interpretation of S and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer

$$\overline{m} = \begin{cases} m, & \text{if } m > 0, \\ 0, & \text{otherwise.} \end{cases}$$

1 $f : \mathbb{Z}_q^B \mapsto \mathcal{R}_q$, given by $f(m_0, ..., m_B) \mapsto \sum_{i=0}^{N-1} m_{\lfloor \frac{iB}{N} \rfloor} X^i$
2 **for** $i \leftarrow 0$ **to** $d - 1$ **do**
3 $\quad$ TV $\leftarrow$ PublicKeySwitch$((c_i, c_i, 0, 0), f, \text{KS})$
4 $\quad$ $\overline{c}_i \leftarrow$ FunctionalBoostrap$(c_{d-1}, \text{TV}, \text{BK})$
5 **return** $\overline{c}$

Table 3.6: Comparison between implementations of an 8-bit ReLU function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[52]	80	≈ 0.3 GB	Negligible	380	1.59
	127	≈ 0.3 GB	Negligible	603.1	1.00
[67]	80	≈ 0.3 GB	Negligible	64.8	9.31
	127	≈ 0.3 GB	Negligible	103.1	5.85
5_5_6_2	127	≈ 4.3 GB	-137.092	86.4	6.98
6_4_6_3	127	≈ 6.5 GB	-180.973	103.6	5.82

Maximum Our implementation of the maximum function is, at first, similar to the integer comparison followed by a multiplexer. However, in this context, we have to also consider signed numbers, which leads us to Algorithm 14. Table 3.7 shows performance results.

Addition We implemented this function using the chaining method since it presents a carry-like logic. Our implementations using the tree-based approach were more expensive since, although we can produce a linear combination for which the carry follows a B-anti-cyclic logic, we could not do the same for the addition itself. Algorithm 15 describes our implementation and Table 3.8 compares it with implementations using logic gates. We used the multi-value extract to perform the scaling in line 6 of Algorithm 15.

3.4.1.4 Additional estimates

Using the results of this section, we can estimate the performance gains our methods could bring to full applications. Let us take, for example, the binarized convolutional neural network (CNN) of Zhou *et al.* [67], which can be implemented using the neural network functions we have implemented. This CNN classifies images in the MNIST

Algorithm 14: Maximum algorithm using the tree-based method.

Input : two sets of LWE samples $c_{j,i} \in \text{LWE}_s(m_{j,i})$, such that $\sum_{i=0}^{d-1} m_{j,i}B^i = m_j$ encodes each number m_j for $j \in \{0,1\}$ and base $B = 4$ with d digits.
Input : a bootstrapping key BK
Input : a packing key switching key KS
Output: A LWE sample $\overline{c} \in \text{LWE}_{\overline{S}}(\overline{m})$, where $\overline{S}$ is a vector (LWE) interpretation of S, and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer

$$\overline{m} = \begin{cases} m_0, & \text{if } m_0 > m_1, \\ m_1, & \text{otherwise.} \end{cases}$$

$f : \mathbb{Z}_q^B \mapsto \mathcal{R}_q$, given by $f(m_0, ..., m_B) \mapsto \sum_{i=0}^{N-1} m_{\lfloor \frac{iB}{N} \rfloor} X^i$
TV $\leftarrow [0, 1, 1, 1]\Delta$
for $i \leftarrow 0$ **to** $d - 1$ **do**
  $c_i \leftarrow c_{0,i} - c_{1,i}$
  $\tilde{c} \leftarrow \text{FunctionalBoostrap}(c_i, \text{TV}, \text{BK})$
  TV $\leftarrow \text{PublicKeySwitch}((\tilde{c}, 1, 1, 1), f, \text{KS})$
TV $\leftarrow \text{PublicKeySwitch}((\tilde{c}, \tilde{c}, -\tilde{c}, -\tilde{c}), f, \text{KS})$
$\tilde{c} \leftarrow \text{FunctionalBoostrap}(c_{0,d-1}, \text{TV}, \text{BK})$
TV $\leftarrow \text{PublicKeySwitch}((\tilde{c}, \tilde{c}, -\tilde{c}, -\tilde{c}), f, \text{KS})$
$\tilde{c} \leftarrow \text{FunctionalBoostrap}(c_{1,d-1}, \text{TV}, f, \text{BK})$
$\tilde{c} \leftarrow \tilde{c} + (0, \frac{q}{2B})$
for $i \leftarrow 0$ **to** $d - 1$ **do**
  TV $\leftarrow \text{PublicKeySwitch}((c_{1,i}, c_{1,i}, c_{0,i}, 0), f, \text{KS})$
  $\overline{c}_i \leftarrow \text{FunctionalBoostrap}(\tilde{c}, \text{TV}, \text{BK})$
return $\overline{c}$

dataset and is composed of 3 binarized convolutional layers, 3 max-pooling layers, and two fully connected layers. We counted the number of operations on each one of them and estimated their execution time using the results of our previous experiments. Table 3.9 shows the results. Although this is a basic estimation, our execution times are reasonably close to the ones reported by Zhou *et al.* [67], especially considering the differences in execution environments. A real implementation would likely present better performance for our methods since they allow for further optimizations, such as using the multi-value bootstrap to batch multiple operations. Nonetheless, the current estimation indicates a speedup of up to 4.9 times, which is within the expected considering the execution time of each function, where the speedup over basic operations for this same implementation ranged from 1.19 (ReLU) to 6.77 (Addition) times.

We can also estimate the performance impact of changing the size of the keys. The main reason we use large keys (compared to implementations using logic gates) is the use of decomposition bases greater than 2 in the LWE-to-RLWE key switching. Using base 64, the Key Switching keys take 4.0 GiB and 6.0 GiB for the parameters **5_5_6_2** and **6_4_6_3**, respectively. Decreasing the base would also linearly decrease the size of these keys, but, to avoid increasing the error, we would need to logarithmically increase the value of t and, consequently, the key-switching execution time. For example, using base 16 instead of 64, the parameter **5_5_6_2** would need a 1 GiB key, but t would need

Table 3.7: Comparison between implementations of an 8-bit max function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[52]	80	≈ 0.3 GB	Negligible	379.3	2.14
	127	≈ 0.3 GB	Negligible	593.1	1.37
[67]	80	≈ 0.3 GB	Negligible	483.1	1.68
	127	≈ 0.3 GB	Negligible	810.6	1.00
5_5_6_2	127	≈ 4.3 GB	-50.0874	228.3	3.55
6_4_6_3	127	≈ 6.5 GB	-156.744	276.8	2.93

Algorithm 15: Addition algorithm using the chaining method.

Input : two sets of LWE samples $c_{j,i} \in \text{LWE}_s(m_{j,i})$, such that $\sum_{i=0}^{d-1} m_{j,i}B^i = m_j$ encodes each number m_j for $j \in \{0, 1\}$ and base $B = 4$ with d digits.
Input : a bootstrapping key BK and a TRLWE Key Switching key KS
Output: A LWE sample $\overline{c} \in \text{LWE}_{\overline{S}}(\overline{m})$, where $\overline{S}$ is a vector (LWE) interpretation of S and $\sum_{i=0}^{d-1} \overline{m}_i B^i = \overline{m}$ encodes the integer $\overline{m} = m_0 + m_1 \bmod B^d$

$\text{TV} \leftarrow [\frac{q}{4B}, \frac{q}{4B}, \frac{q}{4B}, \frac{q}{4B}]$
$\overline{c}_0 \leftarrow 0$
for $i \leftarrow 0$ **to** $d - 1$ **do**
  $\overline{c}_i \leftarrow \overline{c}_i + c_{0,i} + c_{1,i}$
  $\tilde{c} \leftarrow \text{FunctionalBoostrap}(\overline{c}_i, \text{TV}, \text{BK})$
  $\overline{c}_i \leftarrow \overline{c}_i + B \cdot \tilde{c}$
  $\overline{c}_i \leftarrow \overline{c}_i - (0, \frac{q}{4})$
  if $i < d - 1$ **then**
    $\overline{c}_{i+1} \leftarrow (0, \frac{q}{4B})$
    $\overline{c}_{i+1} \leftarrow \overline{c}_{i+1} - \tilde{c}$
return $\overline{c}$

to be increased from 2 to 3, which would increase the execution time of the key switching from 6 ms to 9 ms. At first, this tradeoff seems promising for many functions, especially the ones that make little use of the LWE-to-RLWE key switching. However, increasing the value of t also increases the second term of Equation 3.5 and, hence, might affect the output error variance negatively. For simplicity, we chose two sets of parameters that fit all functions we implemented. A more targeted search for parameters would likely yield better results, and the methods we introduced allow for easily changing parameters even within a single function evaluation.

3.5 Comparison to related work

The literature on using lookup tables (LUT) in homomorphic circuits dates back to the first fully homomorphic encryption schemes presented and has been used with most of the modern FHE schemes [28, 51, 57]. LUTs are a simple and powerful technique to represent arbitrary functions, but problems with latency and precision had also been reported [51, 57]. The introduction of LUT evaluations within the bootstrap by FHEW [30] started a

Table 3.8: Comparison between implementations of an 8-bit addition function.

	Security	Key Size	Error Rate (log_2)	Time (ms)	Speedup
[52]	80	≈ 0.3 GB	Negligible	585	1.21
	127	≈ 0.3 GB	Negligible	708.9	1.00
[67]	80	≈ 0.3 GB	Negligible	338	2.10
	127	≈ 0.3 GB	Negligible	548.7	1.29
5_5_6_2	127	≈ 4.3 GB	-124.7	81.1	8.74
6_4_6_3	127	≈ 6.5 GB	-176.139	94.8	7.48

Table 3.9: Estimation of an inference on the Binarized CNN of Zhou *et al.* [67].

	Security Level	Execution time per layer (h)			Total (h)	Speedup
		Bin. Conv.	Max-Pool.	Fully Conn.		
[67] (reported)	80	19.20	0.67	21.35	41.22	1.88
[67]	80	20.18	0.96	26.87	48.01	1.62
	127	32.46	1.56	43.62	77.64	1.00
5_5_6_2	127	7.39	0.53	7.91	15.83	4.90
6_4_6_3	127	8.19	0.61	9.20	18.00	4.31

new line of work on this topic. Bonnoron *et al.* [7] introduced techniques to evaluate gates with a large number of input bits using a single bootstrap of FHEW and implemented a 6-bit-to-6-bit LUT that runs in less than 10 seconds. Based on their work, Carpov *et al.* [17] presented the multi-value bootstrap of TFHE and lowered the execution time of the 6-bit-to-6-bit LUT to only 1.5 seconds. Our work makes extensive use of the multi-value bootstrap, but we focus more on accelerating the evaluation of non-linear functions than improving the multi-value bootstrap itself. Nonetheless, we did present some contributions towards it, such as the reduction of the complexity of its error growth. Moreover, our combination methods also introduce two new ideas to this line of work: how to use the multi-value bootstrap to accelerate single (instead of multiple) LUT evaluations; and how to improve the LUT evaluation based on particular properties of the encoded function.

Tree-based approach The most similar strategy to our tree-based evaluation is the vertical packing of Chillotti *et al.* [22], which suggests the use of a CMUX tree to choose among multiple RLWE samples and, then, uses a single BlindRotate to perform a final lookup. Similarly to ours, their method also allows some optimizations based on the encoded function (although they did not present nor explore this idea itself). On the other hand, our method constructs LUTs on-the-fly using results of previous lookups, which allows optimizations even within a single LUT. The main difference between them, however, is in their use of TFHE's building blocks. The vertical packing is directly based on CMUX gates and, hence, requires the selector to be encrypted in the binary base in RGSW samples. This makes it a very good choice in the leveled setting, but it requires one circuit bootstrap per bit in the fully homomorphic one. Chillotti *et al.* [22] reports an execution time of around 800ms to evaluate a 6-bit-to-6-bit LUT in the fully homomorphic setting with 80 bits of security. Correcting for the security level and difference between machines, the implementation of Carpov *et al.* is already slightly faster. Bouse *et al.* [11]

also presents a "tree-based" technique for integer comparison using the functional bootstrap, but, besides the name, by our definitions, it is equivalent to the chaining method. More specifically, it rearranges the chaining evaluation in a tree-like fashion (specific to the integer comparison logic) so that it can be parallelized in multiple threads. The output of each LUT is still used to construct the selector of the next one (which defines the chaining) and the technique has no further similarities with our "tree-based" approach.

Evaluation of neural network inference using the Functional Bootstrap Using the functional bootstrap of TFHE in neural network inference seems to be a recent (and promising) trend. As previously cited, Bourse *et al.* [10] and Izabachène *et al.* [43] implemented the sign activation function using it. Boura *et al.* [9] simulated the error propagation of several activation functions and introduced the term *Functional bootstrap*, which we adopt in this work. Klemsa *et al.* [46] presented a Ruby version of TFHE, which includes the functional bootstrap, targeted at neural network implementations. From all previous literature, the only one we found to combine multiple functional bootstraps to evaluate a single function is the integer comparison of Bourse *et al.* [11].

3.6 Discussion

In this chapter, we presented two methods to combine multiple functional bootstraps in TFHE; we performed a thorough error variance and error rate analysis on our methods and on the functional bootstrap itself, including experimental validation; we introduced a multi-value extract procedure to improve the error behavior on scalings and, especially, on the multi-value bootstrap; we introduced a "base-aware" LWE-to-RLWE Key Switching to speedup the LWE packing; and, finally, we selected practical parameters and benchmarked our methods using relevant functions from the literature. We achieved speedups of up to 3.19 times compared to previous literature on the functional bootstrap of TFHE, and of up to 8.7 times compared to implementations using logic gates.

Arbitrary LUTs are inherently exponential to evaluate, which gives more importance to the possibility of optimizing them based on the function particularities, a feature that our methods introduce. In our practical experiments, we limited ourselves to working with the original implementation of TFHE and with precision levels that were previously defined in the literature. Our results demonstrate efficient evaluations with a precision of up to 6 bits for completely arbitrary functions, and up to 32 bits for functions with enough opportunities for optimizations. The techniques themselves can be easily extended to work with higher parameters and are already defined to efficiently exploit the circuit bootstrap. To test it in practice, however, we would need to move to more optimized versions of TFHE (with at least 64-bit Torus precision) and implement an efficient version of the circuit bootstrap, which is mostly a practical open problem. Nonetheless, the speedups we achieved are certainly a good measurement of improvement over previous literature, and some of our contributions, such as the multi-value extract, go beyond the context of functional bootstrap implementations and are useful even for pure arithmetic.

3.6.1 New developments in the literature

This work was published by the IACR Transactions on Cryptographic Hardware and Embedded Systems (TCHES) journal in February 2021. Since then, several works have been published in the literature using, improving, or providing alternatives to the contributions we presented in this work. We summarize some of them in the following.

- Chillotti *et al.* [24] showed that our methods allow evaluating multivariate functions and lift several previously concerning limitations of the functional bootstrap. They also introduced an improved version of the *programmable bootstrapping* (which we discuss in Section 4.1.1) that improves our tree-based approach by allowing messages to be decomposed before using it.

- Clet *et al.* [26] showed that our chaining 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.

- Bergerat *et al.* [5] showed that the bootstrapped version of the vertical packing is generally faster than the tree-based approach for big enough messages. The performance of this technique is mostly reliant on the circuit bootstrap, which we discuss in Section 4.1.5.

Chapter 4

On the efficient implementation of TFHE

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), 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 the first part of this work, we implement the core functionalities of TFHE and the following techniques.

- The Functional [9] or Programmable [23] Bootstrap and its improved version [24];
- The Circuit Bootstrap [21] and its optimizations [24];
- The multi-value bootstrap [17, 24] and its optimizations [38];
- The Key Switching and its optimizations [19];
- The BlindRotate Unfolding [68] and its optimizations [10];
- The Full RGSW bootstrap [37];
- Three different approaches [47, 64, 24] for evaluating the Full-Domain Functional Bootstrap (FDFB);
- Public Key compression using randomness seed, or *seeded (R)LWE*;
- BFV-like multiplication [24]; and
- Bootstrap using Galois Automorphism [50].

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 [5]. 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 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[1] instructions to speed the execution up to 1.5 times.

- We generalize the BlindRotate Unfolding (as suggested by Bourse *et al.* [10]) 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 BlindRotate 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.

[1] Single Instruction Multiple Data

The remainder of this chapter is organized as follows: Section 4.1 presents the techniques implemented in our library and the improvements upon them; Section 4.2 introduces our novels methods; Section 4.3 presents the experimental results; and, finally, Section 4.4 concludes the chapter.

4.1 Implementing existing techniques

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 [30], GSW [37], or even CKKS [20], 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.

4.1.1 The Improved Programmable Bootstrap

In 2021, Chillotti *et al.* [24] 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 3.1.

Algorithm 16 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 [24] (but functionally equivalent).

Algorithm 16: Improved Programmable Bootstrap (PBS) [24]

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 κ and θ
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** FunctionalBootstrap(c', L, BK)

4.1.2 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 4.1.5) and the Tree-Based Functional Bootstrap (Section 3.1). The multi-value functional bootstrap technique introduced by Carpov *et al.* [17] (described in Section 2.5.3) and our improvements over the original technique (Section 3.3.2) are efficient solutions for it, which introduce a noise given by Equation 4.1, 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.

$$Var(Err(c)) \leq s(q-1)\sigma_{FB} \tag{4.1}$$

When performing this estimate, however, both works 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 11) 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 $2^1 + (-1)^2 + 1^2 = 4$.

Bootstrapping many LUTs In 2021, Chillotti *et al.* [24] 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 17 describes the Bootstrap Many LUTs procedure.

Algorithm 17: Bootstrap ManyLUT (BML) [24]

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 $S' \in \mathbb{B}^N$ is a vector interpretation of S

$r \leftarrow \frac{N}{zB}$
$b' \leftarrow \lfloor b2N/q \rceil$ and $a' \leftarrow \lfloor a2N/q \rceil$
$v \leftarrow \sum_{i=0}^{B-1} \sum_{j=0}^{z-1} \sum_{k=0}^{r-1} L_{j,i} X^{(iq+j)r+k}$
ACC $\leftarrow$ BlindRotate$((0, v), (a, b + \frac{q}{4Bz}), \text{BK})$
foreach $i \in [\![0, z)$ **do**
  $c'_i \leftarrow$ SampleExtract$_{ir}$(ACC)
return c'

4.1.3 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 [20, 33]. In 2021, Chillotti *et al.* [24] showed that it is possible to implement the BFV-like [33] tensor product using TFHE parameters. They also showed how it can be used to perform a multiplication between LWE samples. Algorithm 18 describes the RLWE product, and Algorithm 19 shows the multiplication between LWE samples.

Algorithm 18: RLWE Product (RLWEProd) [24]

Input : two RLWE samples $c_i = (a_i, b_i) \in \text{RLWE}_{s,\Delta}(p_i)$, for $p_i \in \mathcal{R}_q$ and $i \in \{0, 1\}$, with scaling factor Δ
Input : a relinearization key $\text{RLK}_i \in \text{RLWE}_s(s^2\beta^j)$, for $j \in [\![0, t)$
Output: $c' \in \text{RLWE}_s(p_0 \cdot p_1)$

$T \leftarrow \left[\left\lfloor \frac{A_1 \cdot A_2}{\Delta} \right\rceil \right]_q$
$A' \leftarrow \left[\left\lfloor \frac{A_1 \cdot B_2 + B_1 \cdot A_2}{\Delta} \right\rceil \right]_q$
$B' \leftarrow \left[\left\lfloor \frac{B_1 \cdot B_2}{\Delta} \right\rceil \right]_q$
$T' \leftarrow \lfloor T\beta^t/q \rceil$
Decompose T', s. t. $T' = \sum_{j=0}^{t-1} T'_j \cdot \beta^j$
return $(A', B') + \sum_{i=0}^{t-1} T'_i \cdot \text{RLK}_i$

Algorithm 19: LWE Multiplication (LWEMult) [24]

Input : two LWE samples $c_i = (a_i, b_i) \in \text{LWE}_s(m_i)$, for $m_i \in \mathbb{Z}_B$ and $i \in \{0, 1\}$
Input : a relinearization key $\text{RLK}_i \in \text{RLWE}_S(S^2\beta^j)$, for $j \in [\![0, t)$
Input : a Packing Key Switching key $\text{KSK}_{s \mapsto S}$
Output: $c' \in \text{LWE}_{S'}(m_0 \times m_1)$

$f : \mathbb{Z}_q \mapsto \mathcal{R}_q = m \mapsto mX^0$
$C_0 \leftarrow \text{PublicKeySwitch}(c_0, f, \text{KSK})$
$C_1 \leftarrow \text{PublicKeySwitch}(c_1, f, \text{KSK})$
$C_{\mathsf{mul}} \leftarrow \text{RLWEProd}(C_0, C_1, \text{RLK})$
return $\text{SampleExtract}_0(C_{\mathsf{mul}})$

4.1.4 Full-Domain Functional Bootstrap (FDFB)

As detailed in Section 2.5.2, 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 [24, 26, 47, 64], 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.

4.1.4.1 The tensor product method

Chillotti *et al.* [24] were the first to present a full-domain functional bootstrap for TFHE or, as they defined, a without-padding programmable bootstrap (WoP-PBS). Algorithm 20 shows their technique, proposed in 2021.

Algorithm 20: Full-Domain Functional Bootstrap based on LWEMult (FDFB-CLOT21) [24]

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 $i \in [\![1, n]\!]$
Input : a relinearization key $\text{RLK}_i \in \text{RLWE}_S(S^2\beta^j)$, for $j \in [\![0, t)$
Input : a Packing Key Switching key $\text{KSK}_{s \mapsto S}$
Output: $c' \in \text{LWE}_{S'}(L[m])$, where $S' \in \mathbb{B}^N$ is the vector interpretation of S

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

4.1.4.2 The PubMux method

In 2021, Kluczniak and Schild [47] proposed a technique for the FDFB based on the definition of a public version of TFHE's C multiplexer (CMUX, Algorithm 5). 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 21 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.

4.1.4.3 The chaining method

The FDFB presented by Chillotti *et al.* [24] 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 22 describes it. Despite requiring two functional bootstraps, the algorithm combines them using the chaining method [38], which provides the lowest output error variance and does not require larger

Algorithm 21: Full-Domain Functional Bootstrap based on PubMux (FDFB-KS21) [47]

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 $\quad c_{\mathsf{sign},i} \leftarrow \mathrm{FunctionalBootstrap}(c, [\frac{-q}{2\mathfrak{B}^i}, ..., \frac{-q}{2\mathfrak{B}^i}], BK)$
5 $\quad c_{\mathsf{sign},i} \leftarrow c_{\mathsf{sign},i} + \frac{q}{2\mathfrak{B}^i}$
6 $v \leftarrow \mathrm{PubMux}(c_{\mathsf{sign}}, p_1, p_2)$
7 **return** $\mathrm{FunctionalBootstrap}(c, v, BK)$

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

parameters. This method was first presented in the *FullFBS* algorithm by Yang *et al.* [64] 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.* [64] by replacing line 1 of Algorithm 22 with line 2 of Algorithm 7. We developed Algorithm 22 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 22: 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 $c_{\mathsf{sign}} \leftarrow \mathrm{FunctionalBootstrap}(c, [\frac{q(B+1)}{4B}, ..., \frac{q(B+1)}{4B}], BK)$
3 $c' \leftarrow c + c_{\mathsf{sign}} - \frac{q(B+1)}{4B}$
4 **return** $\mathrm{FunctionalBootstrap}(c', tv, BK)$

4.1.5 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 tech-

niques [5, 22, 38] require samples to be encrypted as RGSW samples. In this context, the *Circuit Bootstrap* [21], 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 23 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 4.1.2, to accelerate the computation (as suggested by Chillotti *et al.* [24]) at the cost of a slightly increased error rate.

Algorithm 23: Circuit Bootstrap algorithm (CircuitBootstrap) [22]

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: $c' \in \mathrm{RGSW}_{S,(\ell,\mathfrak{B})}(L[m])$

$f : \mathbb{Z}_q \mapsto \mathcal{R}_q = m \mapsto mX^0$
foreach $i \in [\![0, \ell)$ **do**
  $\tilde{c} \leftarrow$ FunctionalBootstrap$(c, L\mathfrak{B}^i, BK)$
  $c'_i \leftarrow$ PrivateKeySwitch$(\tilde{c}, \mathrm{KSKA})$
  $c'_{\ell+i} \leftarrow$ PublicKeySwitch$(\tilde{c}, f, \mathrm{KSKB})$
return c'

4.1.6 The full RGSW bootstrap

The TreeFB algorithm (Section 3.1) 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.* [17] MVFB (Algorithm 8) does not operate over encrypted test vectors and Chillotti *et al.* [24] BML (Algorithm 17) supports a limited number of LUTs. In Chapter 3, we suggested using the CircuitBootstrap to employ the MVFB over encrypted LUTs, but we 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 we discussed [38]. 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 BlindRotate executions as the CircuitBootstrap, but saves time by avoiding key switchings.

The full RGSW bootstrap (RGSW_Bootstrap) is similar to the functional bootstrap described in Algorithm 7, 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 4.1.2.

4.1.7 (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.* [19] 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 $q' = \frac{q}{2N}$.

4.1.8 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 parts, in turn, are the multiplications by RGSW samples, which encrypt s_i. In 2018, Zhou *et al.* [68] showed how to reduce the number of multiplications by unfolding the BlindRotate loop. Equation 4.2 shows their proposal. In the same year, Bourse *et al.* [10] 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.

$$\begin{aligned} X^{as+a's'} &= ss'X^{a+a'} + s(1-s')X^{a} \\ &\quad + (1-s)s'X^{a'} + (1-s)(1-s'). \end{aligned} \tag{4.2}$$

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 is given by $\frac{2^u}{u}$, where u is the unfolding size. Algorithm 24 shows the unfolded blind rotation, where r is the key expansion.

Algorithm 24: 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 $\mathrm{ACC} \leftarrow X^{-\lceil b\frac{2N}{q} \rfloor} \cdot tv$
2 $r \leftarrow \frac{n2^u}{u}$
3 **for** $i \leftarrow 0$ **to** $n-1$ **by** u **do**
4 $\quad C' \leftarrow C_{ir}$
5 $\quad$ **for** $j \in [\![1, r)$ **do**
6 $\quad\quad a' \leftarrow 0$
7 $\quad\quad$ **for** $k \leftarrow 0$ **to** $u-1$ **do**
8 $\quad\quad\quad$ **if** $j \wedge 1$ **then** // Bitwise AND
9 $\quad\quad\quad\quad a' \leftarrow a' + a_{i+u}$
10 $\quad\quad\quad j \leftarrow \lfloor j/2 \rfloor$
11 $\quad\quad \tilde{a} \leftarrow \left\lceil a' \frac{2N}{q} \right\rfloor$
12 $\quad\quad C' \leftarrow C' + C_{ir+j} \cdot X^{\tilde{a}}$
13 $\quad \mathrm{ACC} \leftarrow \mathrm{ACC} \cdot C'$
14 **return** ACC

4.1.9 State-of-the-art summary

Table 4.1 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 is to implement all the techniques in a single highly optimized software library, which is publicly available in our GitHub repository[2].

4.2 Novel techniques

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

4.2.1 UBR multi-value bootstrap

Although a theoretically promising technique, several factors limit the impact of the BlindRotate unfolding in practical performance, with the main one being the exponential blow-up in the number of polynomial additions and multiplications. Nonetheless, in this Section, we introduce another use for the Unfolded BlindRotate (UBR): a new method for multi-value bootstrapping.

We start from the observation that lines 5 to 12 in Algorithm 24, which contain the

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

Table 4.1: Summary of the techniques described in this chapter and our contributions to each.

Procedures	**Section**	**Literature**	**Improvements in this work**
PBS	4.1.1	[24]	-
MVFB	4.1.2	[17, 38]	We modify the composition algorithm to treat a corner case on error growth.
		[24]	-
	4.2.1	This work	New technique
BFV-like multiplication	4.1.3	[24]	-
FDFB	4.1.4.1	[24]	Accelerated using the BLM (as suggested in [24])
	4.1.4.2	[47]	Accelerated using the BLM
	4.1.4.3	[64] and this work	New technique, extending the method of [64]
TreeFB	3.1	[38]	We use the RGSW bootstrap or the MVFB-UBR to provide MVFB for all levels of the tree.
ChainingFB			-
CircuitBootstrap	4.1.5	[22]	Accelerated using the BLM (as suggested in [24])
RGSW_Bootstrap	4.1.6	[37]	-
KeySwitching	2.4.2.1, 4.1.7	[22, 19]	Accelerated using FTM-SE
UBR	4.1.8	[68, 10]	Method generalized for large unfoldings (as suggested in [10])
FTM-SE	4.2.2	[22]	We show how to exploit fast PRNGs to improve performance

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 24 and store the values produced for C' in a n/u-sized array of RGSW samples $\tilde{C}$.
2. Run Algorithm 25 using $\tilde{C}$.

Compared to other existing techniques for multi-value bootstrap, namely the ones we described in Section 4.1.2, 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.* [17], described in Algorithm 8,

Algorithm 25: Multi-Value Functional Bootstrap based on the UBR algorithm (MVFB-UBR)

Input : an LWE sample $c = (a, b) \in \text{LWE}_s(m)$, $m \in \mathbb{Z}_{2N}$
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 $c'_i \in \text{LWE}_{S'}(F_i(m))$ for $i = 1, ..., z$, where $S' \in \mathbb{B}^N$ is a vector interpretation of S

```
b' ← ⌊b2N/q⌉ + n/(2B)
foreach i ∈ [[1, z]] do
    ACC ← L_i
    for j ← 0 to n/u − 1 do
        ACC ← ACC · C̃_j
    c'_i ← SampleExtract_0(ACC)
return c'
```

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.* [24], described in Algorithm 17, 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.

4.2.2 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 BlindRotate 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.* [22] 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 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).

Algorithm 26 shows the core RLWE subtraction algorithm used in the key switching. We could use any PRNG to implement it, but SHAKE256 [31] 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 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.

Algorithm 26: RLWE subtraction using SHAKE256

Input : a compressed RLWE sample $c_0 = (seed_{a_0}, b_0) \in \text{RLWE}_s(p_0)$
Input : an RLWE sample $c_1 = (a_1, b_1) \in \text{RLWE}_s(p_1)$
Output: $c' = (a', b') \in \text{RLWE}_s(p_0 - p_1)$

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

To solve this problem, we implement the key switching as shown in Algorithm 27. There are two main changes to note in this version:

1. We replace SHAKE256 with Xoroshiro [6], 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 [8, 6, 41]. We chose Xoshiro for its performance, but if a secure PRNG is required even for public data, viable alternatives may be found in Lightweight Cryptography(LWC) [62].

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 27: RLWE subtraction using Xoshiro

Input : a compressed RLWE sample $c_0 = (seed_{a_0}, b_0) \in \text{RLWE}_s(p_0)$
Input : an RLWE sample $c_1 = (a_1, b_1) \in \text{RLWE}_s(p_1)$
Output: $c' = (a', b') \in \text{RLWE}_s(p_0 - p_1)$

$state \leftarrow seed_{a_i}$
for $i \leftarrow 0$ **to** $N - 1$ **do**
  $a'_i \leftarrow \text{XOROSHIRO128PP\_NEXT}(state) - a_{1,i}$
  $b'_i \leftarrow b_{0,i} - b_{1,i}$
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 speedups 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.

4.2.2.1 VAES version

On newer computer architectures, we can also use AVX512-VAES instructions [29] 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 great security guarantees.

4.3 Experimental results

We implement all algorithms presented in Sections 4.1 and 4.2 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.

4.3.1 Parameters

We use the parameter sets reproduced in Table 4.2. All of them are extracted from previous literature. We select parameter sets 1 to 3 from Bergerat *et al.* [5] as representatives of some of the most commonly used parameter sizes in TFHE. Meanwhile, we adapted parameter set 4 from TFHEpp [54] since it allows evaluating all techniques described in this work.

Table 4.2: Parameter sets. Noise is chosen based on the dimension for obtaining a 128-bit security level. bs and ks stand for bootstrapping and key switching, respectively.

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

4.3.2 Basic arithmetic

Most of the optimized implementations of TFHE rely on the Fast Fourier Transform (FFT) for providing fast polynomial arithmetics [25]. Our library presents two options for FFT implementations: the SPQLIOS [34] library[3], and the FFNT library [45] 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 4.3 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

[3]SPQLIOS was presented by Nicolas Gama *et al.* [34] with TFHE [22]. It was adapted by TFHEpp [54] for their C++ code, which we adapted to pure C.

reductions on floating-point registers. Listing 4.1 shows the modular reduction originally used by SPQLIOS, which relies on a sequence of bitwise operations. Listing 4.2 shows our vectorized version implemented as a sequence of scalings and reductions over floating-point registers. 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.

Table 4.3: FFT implementation performance for polynomials of size $N = 2048$. Time in microseconds.

Implementation	Source	FFT	IFFT
FFNT	[45]	8.43	11.12
FFNT (AVX2/FMA)	This work	3.58	7.49
SPQLIOS (FMA)	[34]	2.68	4.73
SPQLIOS (AVX512)	This Work	2.58	2.65

Listing 4.1: Floating-point modular reduction from SPQLIOS [34].

```
uint64_t* vals = (uint64_t*) input;
uint64_t valmask0 = 0x000FFFFFFFFFFFFFul;
uint64_t valmask1 = 0x0010000000000000ul;
uint16_t expmask0 = 0x07FFu;
for (int i=0; i<N; i++) {
    uint64_t val = (vals[i]&valmask0)|valmask1; // mantissa
    uint16_t expo = (vals[i]>>52)&expmask0; // exponent
    int16_t trans = expo-1075;
    uint64_t val2 = trans>0?(val<<trans):(val>>-trans);
    output[i]=(vals[i]>>63)?-val2:val2;
}
```

Listing 4.2: Floating-point modular reduction using AVX512 instructions.

```
__m512d * in512 = (__m512d *) input;
__m512i * out512 = (__m512i *) output;
__m512d modc = {64, 64, 64, 64, 64, 64, 64, 64};
for (size_t i = 0; i < N/8; i++) {
    __m512d tmp = _mm512_scalef_pd (in512[i], -modc); //
        tmp ← in · 2^-64
    tmp = _mm512_reduce_pd (tmp, 0); // tmp ← tmp − round(tmp)
    tmp = _mm512_scalef_pd (tmp, modc); // tmp ← tmp · 2^64
    out512[i] = _mm512_cvtpd_epi64 (tmp);
}
```

4.3.3 Memory accesses impact

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 2.6μs

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 4.1 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.

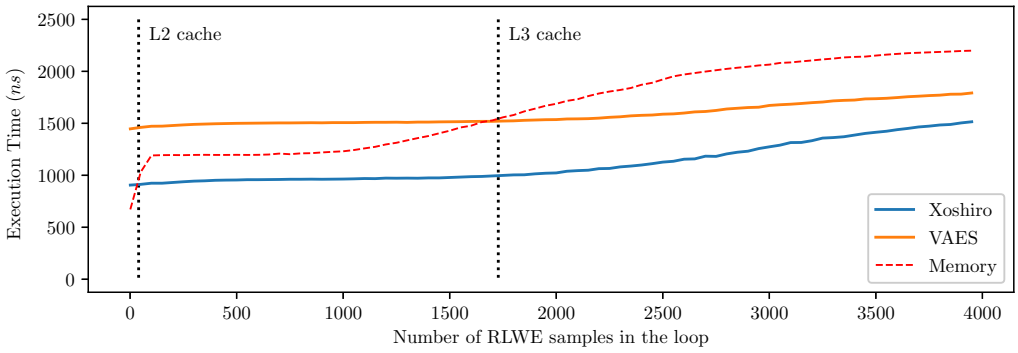


Figure 4.1: Execution time of a single RLWE addition in a summation of RLWE samples. Vertical lines represent the number of samples required to fill the data cache. VAES and Xoshiro curves are the results of using the FTM-SE with the respective PRNG algorithm.

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.

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 guaranteed.

4.3.4 FTM-SE

Table 4.4 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.

Table 4.4: 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

4.3.5 Bootstrap

Table 4.5 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.

Table 4.5: 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

4.3.5.1 Comparison against other libraries

TFHEpp [54] 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 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. Compared to previous versions of TFHEpp, our implementation was 20% faster on average.

There are also several commercial libraries implementing optimized versions of TFHE, such as OpenFHE [4] and TFHE-rs [66]. Their purpose and scope are, however, very differ-

ent 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 NTT. 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`) that TFHE-rs uses for providing their benchmark results [65], which also include data for several other libraries and should, hence, facilitate such analysis as future work.

4.4 Discussion

As we tried to extend the results of Chapter 3 and develop new techniques for TFHE, the limitations of the original TFHE implementation became apparent to us. It did not support most of the techniques being proposed at the time, and authors would need to rely on their own implementations to test their contributions. This led us to two significant issues. Firstly, the new techniques were now scattered across many different libraries, as each group of authors would choose a different base implementation and produce their own variation of it. Secondly, most authors choose to work with implementations that provide high-level interfaces for basic arithmetic procedures, such as PALISADE [2]. This choice is natural, as they facilitate the implementation, but it comes with a significant performance penalty, as these implementations are generally far from providing competitive performance levels.

To continue our research, it was necessary to have a unified software platform over which contributions and improvements to TFHE could be easily developed and tested in efficient ways. In this context, we decided to introduce our library. MOSFHET is fully portable and self-contained, with optional optimizations using AVX2 and AVX512 instructions. In this chapter, we presented both the library itself and the initial results we obtained from it. By combining existing techniques, we were able, for example, to accelerate the CircuitBoostrap up to 2.76 times. The suggestions of combining these techniques were not new, but, as far as we know, we were the first to do so to this extent for TFHE (considering the number of techniques) in a highly-optimized library.

The novel contributions we introduced in Section 4.2 also have a direct impact on our previously developed techniques. Our UBR-MVFB, for example, enables accelerating the TreeFB significantly (up to 8 times considering the results of Table 4.5). 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.

Chapter 5

Securing human genome inference

The implications of human genome data sharing are a growing concern for the scientific community, which is working towards the development of collaboration standards and frameworks [48]. Privacy is one of the utmost issues, as the consequences of unrestricted sharing can be disastrous considering the sensitive nature of the source data as well as the inferences that can be made by combining it with other databases [49]. Furthermore, any data disclosures affect not only individual data owners but also their blood relatives. This problem is further aggravated by the increasing use of cloud computing for providing scalable genomic data processing, as pointed out by the USA National Institutes of Health [42].

On the other hand, several research fields could leverage large genomic databases to advance human health technologies. In Personalized Medicine [18], it is possible to correlate genotypic and phenotypic data for providing targeted diagnoses and treatments for diseases. This process, however, requires not only the patients to share their genomic sequencing information but also the establishment of large databases to carry out statistical analyses. Data inference models trained from these databases are also sensitive and may reveal information about the training dataset. In this context, Fully Homomorphic Encryption (FHE) [36] comes as an ideal solution, allowing the evaluation of these models without revealing any information about the user input or the model parameters.

In 2014, the iDash competition [1] was created among the efforts of the scientific community towards seeking new solutions for securing genome data processing in cloud environments. In 2022, its homomorphic encryption track proposes an instantiation of genotype-to-phenotype inference over encrypted data as a challenge. In particular, the inference must be performed from high throughput genomic data without any cleartext preprocessing. In this way, the challenge goes beyond the training of an inference model and its homomorphic evaluation. It is also necessary to handle a relatively large amount of encrypted data, which, as we show in this chapter, presents issues on its own.

Contributions

In this work, we present our submission for the homomorphic encryption track of iDash 2022 and analyze it considering different aspects and execution environments. Our focus is mainly on the homomorphic evaluation and handling of encrypted data. We first trained

a simple inference model based on decision trees of up to depth 5. Then, we investigated efficient ways of homomorphically evaluating it. While our model itself might not be on par with other more elaborate solutions, we evaluate it in a generic way using look-up tables (LUT), which allow us to test, in a real-world application, some of the solutions we developed in Chapters 3 and 4. Moreover, since we are using only generic techniques, the results we obtain and the conclusions we draw from them are not limited to our model. Instead, they can be applied to many other models in the same context, including those that are not based on decision trees. We analyze three different methods for data encryption and their trade-off in terms of storage and execution time in our inference model, and we evaluate the impact of the cloud environment on the methods. All of our experiments are in the context of the iDash competition and consider the restrictions imposed by the challenge.

The rest of this chapter is organized as follows: Section 5.1 introduces the iDash competition, its current challenge, and an overview of our solution; Section 5.2 details our solution; Section 5.3 presents the experimental results; and Section 5.4 summarizes the chapter.

5.1 The iDash competition

The iDASH Healthcare Privacy Protection Challenge [1] is an annual competition promoted by healthcare-related government agencies, the industry, and the scientific community in an effort to, as they define, *"evaluate the performance of state-of-the-art methods that ensure rigorous data confidentiality during data analyses in a cloud environment"* [1]. Recent editions are split into three tracks: blockchain, homomorphic encryption, and confidential computing.

5.1.1 The 2022 Homomorphic Encryption Track

The task for the Homomorphic Encryption Track of iDash 2022 [44] is to perform phenotype prediction from high throughput genomic data. Essentially, we are provided with genotypes for 200 individuals, each of them containing 20390 features, and we need to predict values for 5 different phenotypes in 30 minutes. We are free to choose any inference model, which we can train on a dataset with 3000 individuals. Each feature is a value in $\{0, 1, 2\}$, the first three phenotypes are floating-point values, and the last two are binary values. The challenge requires schemes to provide 128-bit security and the entire solution to run in less than 30 minutes while taking at most 500 GB of storage. The evaluation system is also limited to 4 CPUs and 32 GB of RAM. No pre or post-computation is allowed. All input data and all parameters of the trained model must be encrypted. Each solution should present the following workflow:

1. *Client*: Setups the cryptosystem, generates and saves the keys, and encrypts the input data.

2. *Modeler*: Using the client's private key, it generates and encrypts the model parameters.

3. *Evaluator*: Receives the client's encrypted input, and the modeler's encrypted parameters, and evaluates the model inference, producing the encrypted output. It only has access to encrypted data and public information (*e.g.*, public evaluation keys). Feature selection (*i.e.*, selecting just a subset of the input features to work over) is allowed to be performed using public information.

4. *Client*: Finally decrypts the results.

Furthermore, the Client, Modeler, and Evaluator are required to be separate processes communicating through files.

5.1.2 Our model

Challenges in iDash always come in two parts: *"How to get the best inference model"* and *"How to evaluate this model homomorphically"*. The scope of our work is limited to the second. Therefore, we considered a basic inference model based on relatively shallow (depth 5) decision trees. Figure 5.1 illustrates a small decision tree. Our model achieves accuracy (measured in 1-NRMSE, 1 minus the normalized root mean squared error, and AUC, Area under the ROC Curve, as defined by the challenge [44]) from 81% up to 96%, which might not beat more elaborate models designed for the problem. However, when homomorphically evaluating the model, we preserve bit precision throughout the entire execution, so that it represents any model of similar size. Our homomorphic evaluation approach treats the tree decisions as just 32 bits of data, without any assumptions on the type of model data.

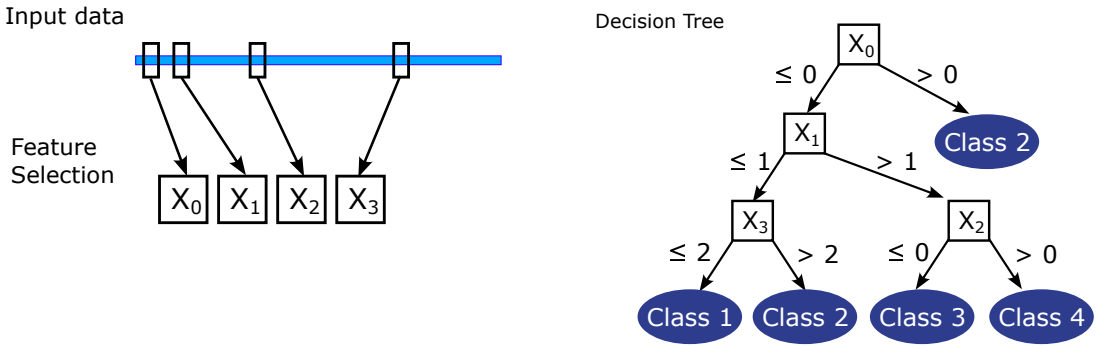


Figure 5.1: Classification using a decision tree.

5.2 Homomorphic evaluating our solution

As each of the input features is represented in 2 bits, and our 5-depth decision trees look at most 11 features, we encoded the entire tree evaluation as a LUT with 2^{22} slots, each one storing a 32-bit floating-point number. Notice that there are two very obvious optimizations that we could have done at this point:

- At the output: A 5-depth tree has at most 32 possible decisions. So we did not need to keep 32-bit floats as the output of the tree. This is also true (and even more drastic) for the binary phenotypes, which can only be 0 or 1.

- At the input: Each input feature can only assume values in $\{0, 1, 2\}$, and the tree does not necessarily check for all of them. In this way, a big part of the LUT is taken by positions that are never looked up.

We could have implemented such optimizations, but they would make our solution specific to the chosen inference model, which is something we are trying to avoid as we want our solution to be as generic as possible within the context of iDash. Notice that, in the way we designed it, the results we present in this chapter represent not only other decision-tree-based models of the same size but all inference models that could be represented as a large LUT evaluation for the competition.

5.2.1 LUT evaluation

We evaluate the LUT using the vertical packing method [22], which we briefly described in Section 2.5.1. This method is typically the most efficient way of evaluating LUTs in a leveled setting, as the RGSW $\times$ RLWE multiplications take just tens of microseconds and even relatively large LUTs require a small amount of them. In a bootstrapped setting, we could consider using the Tree-Based Functional Bootstrap approach (Section 3.1.1) for smaller LUTs, but, for the sizes required by our model, the bootstrapped version of the vertical packing would also be the best choice [5]. As we mentioned in Section 5.2, we want our results to cover any model in iDash that could be represented as a LUT of similar size. Therefore, we also did not consider methods that target specifically the evaluation of decision trees, such as the SortingHat [27], which could present much better results.

For each of the 200 individuals, our evaluator works as follows:

1. It receives the encrypted input data and selects which features are relevant for the decision tree. From the 20390 features, only around 10 of them are considered for inference.

2. It generates an array of RGSW samples, a process that we further discuss in the next section. Each sample encrypts a bit of the selected features.

3. It receives the encrypted model. The model is already represented as a LUT encrypted in an array of RLWE samples.

4. Using the results of steps 2 and 3, it performs the evaluation itself, using vertical packing.

5. It returns the result.

Encryption Approaches

The only undefined part of this workflow is also the most challenging part of our approach: How to generate an array of RGSW samples, each encrypting a bit of the selected features. We implemented three approaches for that. In all of them, the client encrypts the features bit by bit:

1. **RGSW-single.** Each bit is encrypted alone in an RGSW sample. This seems to be the ideal solution for optimizing execution time at the tree evaluation, but it requires the client to generate more than 8 million RGSW samples, which might require hundreds of gigabytes to be stored.

2. **RLWE-packed.** Each bit is encrypted in a monomial of an RLWE sample. Since each RLWE sample has N monomials and N is usually around 2^{10} to 2^{14}, the client would need just around 4 to 10 thousand RLWE samples to encrypt everything. Depending on the parameters, this method would require just a few hundred megabytes of storage. On the other hand, the evaluator would need to extract the bits from the RLWE samples (which can be done using the inexpensive SampleExtract procedure) and transform them into RGSW samples. This transformation requires a Circuit Bootstrap [21], which, as we discuss in Section 4.1.5, is a very expensive procedure.

3. **RGSW-packed.** Each bit is encrypted in a monomial of an RGSW sample. This is a middle-ground solution between the two other approaches. Since the features are already in RGSW samples, it is not necessary to perform a Circuit Bootstrap. However, it is still necessary to isolate a monomial from the others (a single RGSW sample is encrypting N bits of the features, and we want just one of them). This is accomplished by a sequence of key switching procedures [22] that zeroes the unselected monomials and reconstructs the RGSW sample. These key switchings are faster than a Circuit Bootstrap, but this method also requires 2ℓ times more storage.

5.3 Experimental Results

Table 5.1 shows the two execution environments we experimented with and the reference specifications provided by the iDash competition. We chose `d3.xlarge` and `i4i.xlarge` instances as they are the closest to the reference machine we could find on AWS. Table 5.2 shows the cryptosystem parameters for each approach. In all cases, we built our implementations using our MOSFHET library [39].

Table 5.1: Execution environments. The storage type (HDD/SSD) was not specified for the reference machine.

	iDash Reference	`d3.xlarge`	`i4i.xlarge`
Processor	Intel 8180	Intel 8259CL	Intel 8375C
Frequency	2.5 GHz	2.5 GHz	2.9 GHz
CPUs	4	4	4
RAM	32 GB	32 GB	32GB
Storage	500 GB	5940 GB - HDD	937 GB - SSD

Table 5.2: Parameters for TFHE.

	LWE		RLWE		RGSW	
	n	σ/q	N	σ/q	$\log_2(\beta)$	ℓ
RGSW-single	-	-	1024	2^{-25}	8	2
Others	632	2^{-15}	2048	2^{-44}	9	4

5.3.1 Storage

Table 5.3 shows the storage requirements of each approach. We did not include the encrypted model, since it does not vary according to the approach we choose to encrypt the input. Notice that the RGSW-single approach almost reaches the 500GB limit specified by the challenge, whereas the packed approaches take just around 1% of the available storage.

Table 5.3: Storage requirements of each approach.

	RGSW-single	**RLWE-packed**	**RGSW-packed**
Private Key	8 KB	16 KB	16 KB
Eval. Key	0	5.1 GB	3.8 GB
Genotype	497.8 GB	124.4 MB	497.8 MB

5.3.2 Performance

Tables 5.4 and 5.5 present the execution time, in seconds, for each of our test machines. Each result is the average of 5 executions, for which we calculated a 95%-confidence interval. We compiled our code using the default compiling options from MOSFHET [39] and measured time using Linux wall-clock timestamps (`date +%s`). Client and Modeler are single-thread processes while the evaluator is parallelized in 5 processes, one for each phenotype inference. Although the slightly higher clock frequency of the `i4i.xlarge` may have been an advantage, the differences in storage technologies are certainly a dominant factor in the comparison between machines.

Table 5.4: Execution time, in seconds, in a `d3.xlarge`.

	RGSW-single	**RLWE-packed**	**RGSW-packed**
Client	3664.4 ± 13.2	69.2 ± 11.4	63.6 ± 1.6
Modeler	14.6 ± 1.0	12.0 ± 3.0	10.2 ± 1.3
Evaluator	102.4 ± 0.5	1804.0 ± 13.8	1300.0 ± 0.9

RGSW-single approach

While disk IO was the bottleneck for the client on the `d3.xlarge`, the same did not happen on the `i4i.xlarge`. The average storage writing speed was 281 MBps, significantly below the NVME writing capacity of more than 1 GBps. Considering this result, we could have parallelized client encryption and reduced the client execution time by up to 3 times. This

Table 5.5: Execution time, in seconds, in a `i4i.xlarge`.

	RGSW-single	**RLWE-packed**	**RGSW-packed**
Client	1812.4 ± 37.4	36.2 ± 0.7	32.4 ± 2.5
Modeler	4.6 ± 0.5	4.4 ± 0.5	4.2 ± 0.4
Evaluator	113.8 ± 5.8	1841.8 ± 57.5	1332.0 ± 50.7

would certainly make the RGSW-single approach significantly faster than the alternatives. However, as the storage technology of the iDash reference machine was not specified, we could not follow this optimization. In the `d3.xlarge`, for example, parallelization would not help as the client was already writing at 139 MBps, near the limit of the HDD writing bandwidth.

RLWE-packed approach

As expected, client encryption became much faster, as each RLWE sample is now encrypting $N = 2048$ bits of the features. The evaluator execution time, however, grew considerably, with a slowdown of 17.6 times on the `i4i.xlarge` and of 16.2 times on the `d3.xlarge`.

RGSW-packed approach

The $\ell = 4$ times increment in the encrypted genotype size (compared to RLWE-packed) was compensated by a 1.38 times speedup in the evaluator, without observable slowdowns on client encryption. At first, the 4 times increment in storage could be seen as a significant drawback. However, in practice, it requires just an additional 373.4 MB of storage, which even the HDD of the `d3.xlarge` can load in a bit less than 3 seconds. The 1.38 times speedup in execution time, on the other hand, represents at least 504 seconds, which is more than a hundred times the possible slowdown caused by the additional use of storage.

5.4 Summary

In this chapter, we introduced a proposal for the homomorphic encryption track of iDash 2022 and addressed the challenges of homomorphically evaluating it in a cloud environment. We chose homomorphic evaluation methods that are generic enough for our results to apply to other models and applications in a similar context. In our final submission to the competition, we used the RGSW-packed approach for encrypting data, as its entire execution takes around 23 minutes to run, well within the 30-minute limit of the competition. Nonetheless, we note that the performance enabled by RGSW-single encryption could be leveraged by other applications, as each LUT takes less than 0.1s to be evaluated on average, even considering all the overhead introduced by IO operations.

Chapter 6

Conclusion

As FHE evolves, different programming paradigms are adopted for improving performance in specific use cases. The evaluation of look-up tables and, more specifically, the programmable bootstrap gained traction recently as one of the most promising ways of evaluating arbitrary functions. Our first contributions to it, as detailed in Chapter 3, came at a time when few applications were using it, and most of them were limited to low-precision functions. In fact, not even the term itself had been coined yet (thus why we refer to it by the generic naming of functional bootstrap). Our work was one of the first to address the problem of evaluating functions with high precision using it. It presented some of the first methods for doing so efficiently and paved the way for several other methods that have been developed since then, helping it become the established approach it is today.

Our follow-up contributions to the method (Chapter 4) came in the form of a comprehensive library implementing almost all the techniques available for TFHE at the time in a single highly-optimized implementation. Our primary goal with this was to provide a unified source for comparing the many new techniques that were been proposed for TFHE. Notwithstanding, we also introduced several new contributions and provided insightful analyses of the performance of core FHE procedures. Some of them targeted specifically at improving our previous contributions, such as our new method for Multi-value bootstrap (Section 4.2.1), which enables accelerating the tree-based approach. Others are more generic and have a broader application to several procedures, as is the case of the FTM-SE. It is also worth noting that some of our contributions, such as the improvements to SPQLIOS, are already being used by other projects [54].

As a final contribution to close this work, we experimented with our contributions in a real-world application in the context of the 2022 iDash competition. Instead of engaging in the typical machine learning optimizations that are common to solutions submitted iDash, we chose a basic inference model and evaluated it homomorphically as it was. In this way, we see this work as a real-world benchmark for our techniques and implementations, which allowed us not only to address performance issues but also other aspects such as the storage and loading of large encrypted machine learning models.

6.1 Future work

Our library, MOSFHET, brings many opportunities for improvements to state-of-the-art techniques for TFHE. By providing a broad range of efficiently implemented procedures, it allows one to conduct their own analyses and propose new improvements for the scheme. As a practical example of future work, we do not perform a parameter optimization for the techniques we present in this work since, as we mentioned in Chapter 4, parameter optimization is generally an intricate and often application-dependent task. However, it is also a fundamental step towards improving the performance of FHE evaluation, especially for specific use cases. Another example of future work on MOSFHET could be addressing the practical challenges of implementing public-key encryption, such as protection against side-channel attacks.

On the algorithmic side, many of our contributions already have an impact on the literature, and much of the future work we suggested throughout this document is already being done by other authors. Nonetheless, there is still a long way to improve FHE performance to a point in which it is practical for any application. When we consider the evaluation of functions with high precision, which was one of the main topics of this work, there are many applications that require much larger precision than what is currently practical with FHE. The use of lookup tables for evaluating functions, as efficient as it may be, has its limitations in the asymptotic complexity of a LUT evaluation, which is inherently exponential in the input size. While there is still room for improving the homomorphic evaluations of LUTs, as many recent works have shown, the development of new paradigms for homomorphic evaluation seems to be necessary for making FHE practical beyond the typical use cases currently considered by the community. There are also many aspects that affect FHE performance and require special considerations in terms of security. For instance, multi-key and threshold FHE are currently hot topics of research, but current solutions still incur a non-negligible performance overhead to be secure, often impairing its widespread adoption.

Bibliography

[1] About - IDASH PRIVACY & SECURITY WORKSHOP 2022 - secure genome analysis competition. URL: http://www.humangenomeprivacy.org/2022/about.html.

[2] PALISADE Lattice Cryptography Library (release 1.11.5), 2021. URL: https://palisade-crypto.org/.

[3] Martin R. Albrecht, Rachel Player, and Sam Scott. On the concrete hardness of Learning with Errors. *Journal of Mathematical Cryptology*, 9(3):169 – 203, October 2015. Place: Berlin, Boston Publisher: De Gruyter. URL: https://www.degruyter.com/view/journals/jmc/9/3/article-p169.xml, doi:https://doi.org/10.1515/jmc-2015-0016.

[4] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V, Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. OpenFHE: Open-Source Fully Homomorphic Encryption Library, 2022. Report Number: 915. URL: https://eprint.iacr.org/2022/915.

[5] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter Optimization & Larger Precision for (T)FHE, 2022. Published: Cryptology ePrint Archive, Paper 2022/704. URL: https://eprint.iacr.org/2022/704.

[6] David Blackman and Sebastiano Vigna. Scrambled Linear Pseudorandom Number Generators. *ACM Transactions on Mathematical Software*, 47(4):36:1–36:32, September 2021. doi:10.1145/3460772.

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

[8] Joppe W. Bos, Simon Friedberger, Marco Martinoli, Elisabeth Oswald, and Martijn Stam. Fly, you fool! Faster Frodo for the ARM Cortex-M4, 2018. Report Number: 1116. URL: https://eprint.iacr.org/2018/1116.

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

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

[11] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved Secure Integer Comparison via Homomorphic Encryption. In Stanislaw Jarecki, editor, *Topics in Cryptology – CT-RSA 2020*, pages 391–416, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-40186-3_17.

[12] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology – CRYPTO 2012*, Lecture Notes in Computer Science, pages 868–886, Berlin, Heidelberg, 2012. Springer. doi:10.1007/978-3-642-32009-5_50.

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

[14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, January 2012. Association for Computing Machinery. doi:10.1145/2090236.2090262.

[15] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*, pages 97–106, October 2011. ISSN: 0272-5428. doi:10.1109/FOCS.2011.12.

[16] Zvika Brakerski and Vinod Vaikuntanathan. Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, Lecture Notes in Computer Science, pages 505–524, Berlin, Heidelberg, 2011. Springer. doi:10.1007/978-3-642-22792-9_29.

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

[18] Isaac S. Chan and Geoffrey S. Ginsburg. Personalized Medicine: Progress and Promise. *Annual Review of Genomics and Human Genetics*, 12(1):217–244, 2011. _eprint: https://doi.org/10.1146/annurev-genom-082410-101446. doi:10.1146/annurev-genom-082410-101446.

[19] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient Homomorphic Conversion Between (Ring) LWE Ciphertexts, 2020. Report Number: 015. URL: https://eprint.iacr.org/2020/015.

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

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

[22] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: Fast Fully Homomorphic Encryption Over the Torus. *Journal of Cryptology*, 33(1):34–91, January 2020. URL: http://link.springer.com/10.1007/s00145-019-09319-x, doi:10.1007/s00145-019-09319-x.

[23] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks. In Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander Schwarzmann, editors, *Cyber Security Cryptography and Machine Learning*, Lecture Notes in Computer Science, pages 1–19, Cham, 2021. Springer International Publishing. doi:10.1007/978-3-030-78086-9_1.

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

[25] Eleanor Chu and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, November 1999. Google-Books-ID: 30S3kRiX4xgC.

[26] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping, 2022. Report Number: 149. URL: https://eprint.iacr.org/2022/149.

[27] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering, 2022. Published: Cryptology ePrint Archive, Paper 2022/757. URL: https://eprint.iacr.org/2022/757.

[28] Jack L. H. Crawford, Craig Gentry, Shai Halevi, Daniel Platt, and Victor Shoup. Doing Real Work with FHE: The Case of Logistic Regression. In *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, WAHC '18, pages 1–12, New York, NY, USA, 2018. Association for Computing Machinery. event-place: Toronto, Canada. doi:10.1145/3267973.3267974.

[29] Nir Drucker, Shay Gueron, and Vlad Krasnov. Making AES Great Again: The Forthcoming Vectorized AES Instruction. In Shahram Latifi, editor, *16th International Conference on Information Technology-New Generations (ITNG 2019)*, Advances in Intelligent Systems and Computing, pages 37–41, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-14070-0_6.

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

[31] Morris J. Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions. Technical Report NIST FIPS 202, National Institute of Standards and Technology, July 2015. URL: https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf, doi:10.6028/NIST.FIPS.202.

[32] Taher Elgamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, 31(4):469–472, 1985. doi:10.1109/TIT.1985.1057074.

[33] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption, 2012. Report Number: 144. URL: https://eprint.iacr.org/2012/144.

[34] Nicolas Gama and others. Spqlios FFT Library, 2016. URL: https://github.com/tfhe/tfhe/tree/master/src/libtfhe/fft_processors/spqlios.

[35] Nicholas Genise, Daniele Micciancio, and Yuriy Polyakov. Building an Efficient Lattice Gadget Toolkit: Subgaussian Sampling and More. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019*, pages 655–684, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-17656-3_23.

[36] Craig Gentry. *A fully homomorphic encryption scheme*. PhD Thesis, Stanford University, 2009.

[37] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based. In

Ran Canetti and Juan A. Garay, editors, *Advances in Cryptology – CRYPTO 2013*, Lecture Notes in Computer Science, pages 75–92, Berlin, Heidelberg, 2013. Springer. doi:10.1007/978-3-642-40041-4_5.

[38] Antonio Guimarães, Edson Borin, and Diego F. Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2021(2):229–253, February 2021. URL: https://tches.iacr.org/index.php/TCHES/article/view/8793, doi:10.46586/tches.v2021.i2.229-253.

[39] Antonio Guimarães, Edson Borin, and Diego F. Aranha. MOSFHET: Optimized Software for FHE over the Torus, 2022. Published: Cryptology ePrint Archive, Paper 2022/515. URL: https://eprint.iacr.org/2022/515.

[40] Antonio Guimarães, Leonardo Neumann, Fernanda A. Andaló, Diego F. Aranha, and Edson Borin. Homomorphic evaluation of large look-up tables for inference on human genome data in the cloud. In *2022 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 33–38, November 2022. doi:10.1109/SBAC-PADW56527.2022.00015.

[41] François Gérard and Mélissa Rossi. An Efficient and Provable Masked Implementation of qTESLA. In Sonia Belaïd and Tim Güneysu, editors, *Smart Card Research and Advanced Applications*, Lecture Notes in Computer Science, pages 74–91, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-42068-0_5.

[42] National Institutes of Health and others. NIH security best practices for controlled-access data subject to the NIH Genomic Data Sharing (GDS) policy. *NIH Office of Science Policy*, 2015.

[43] Malika Izabachène, Renaud Sirdey, and Martin Zuber. Practical Fully Homomorphic Encryption for Fully Masked Neural Networks. In Yi Mu, Robert H. Deng, and Xinyi Huang, editors, *Cryptology and Network Security*, pages 24–36, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-31578-8_2.

[44] Miran Kim, Arif Harmanci, and Xiaoqian Jiang. Task 2 - Competition tasks - IDASH PRIVACY & SECURITY WORKSHOP 2022 - secure genome analysis competition. URL: http://www.humangenomeprivacy.org/2022/competition-tasks.html.

[45] Jakub Klemsa. Fast and Error-Free Negacyclic Integer Convolution using Extended Fourier Transform, 2021. Report Number: 480. URL: https://eprint.iacr.org/2021/480.

[46] Jakub Klemsa and Martin Novotný. WTFHE: neural-netWork-ready Torus Fully Homomorphic Encryption. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–5, June 2020. ISSN: 2637-9511. doi:10.1109/MECO49872.2020.9134331.

[47] Kamil Kluczniak and Leonard Schild. FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption, 2021. Report Number: 1135. URL: https://eprint.iacr.org/2021/1135.

[48] Bartha Maria Knoppers. Framework for responsible sharing of genomic and health-related data. *The HUGO journal*, 8(1):1–6, 2014. Publisher: Springer.

[49] Tsung-Ting Kuo, Xiaoqian Jiang, Haixu Tang, XiaoFeng Wang, Tyler Bath, Diyue Bu, Lei Wang, Arif Harmanci, Shaojie Zhang, Degui Zhi, and others. iDASH secure genome analysis competition 2018: blockchain genomic data access logging, homomorphic encryption on GWAS, and DNA segment searching, 2020.

[50] Yongwoo Lee, Daniele Micciancio, Andrey Kim, Rakyong Choi, Maxim Deryabin, Jieun Eom, and Donghoon Yoo. Efficient FHEW Bootstrapping with Small Evaluation Keys, and Applications to Threshold Homomorphic Encryption, 2022. Report Number: 198. URL: https://eprint.iacr.org/2022/198.

[51] Qian Lou, Bo Feng, Geoffrey Charles Fox, and Lei Jiang. Glyph: Fast and Accurately Training Deep Neural Networks on Encrypted Data. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 9193–9202. Curran Associates, Inc., 2020. URL: https://proceedings.neurips.cc/paper_files/paper/2020/file/685ac8cadc1be5ac98da9556bc1c8d9e-Paper.pdf.

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

[53] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors over Rings. In Henri Gilbert, editor, *Advances in Cryptology – EUROCRYPT 2010*, Lecture Notes in Computer Science, pages 1–23, Berlin, Heidelberg, 2010. Springer. doi:10.1007/978-3-642-13190-5_1.

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

[55] Daniele Micciancio and Yuriy Polyakov. Bootstrapping in FHEW-like Cryptosystems, 2020. Report Number: 086. URL: https://eprint.iacr.org/2020/086.

[56] Daniele Micciancio and Jessica Sorrell. Ring Packing and Amortized FHEW Bootstrapping. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 100:1–100:14, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISSN: 1868-8969. URL: http://drops.dagstuhl.de/opus/volltexte/2018/9104, doi:10.4230/LIPIcs.ICALP.2018.100.

[57] Karthik Nandakumar, Nalini Ratha, Sharath Pankanti, and Shai Halevi. Towards Deep Neural Network Training on Encrypted Data. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 40–48, June 2019. ISSN: 2160-7516. doi:10.1109/CVPRW.2019.00011.

[58] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology — EUROCRYPT '99*, pages 223–238, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. doi:10.1007/3-540-48910-X_16.

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

[60] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, February 1978. URL: https://dl.acm.org/doi/10.1145/359340.359342, doi:10.1145/359340.359342.

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

[62] Markku-Juhani O. Saarinen. Exploring NIST LWC/PQC Synergy with R5Sneik: How SNEIK 1.1 Algorithms were Designed to Support Round5, 2019. Report Number: 685. URL: https://eprint.iacr.org/2019/685.

[63] Nigel P. Smart. *Cryptography Made Simple*. Springer Publishing Company, Incorporated, 1st edition, 2015. doi:10.5555/2927491.

[64] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security, 2021. Report Number: 1347. URL: https://eprint.iacr.org/2021/1347.

[65] Zama. Benchmarks - TFHE-rs. URL: https://docs.zama.ai/tfhe-rs/getting-started/benchmarks.

[66] Zama. TFHE-rs: Pure Rust implementation of the TFHE scheme for boolean and integers FHE arithmetics., May 2023. URL: https://github.com/zama-ai/tfhe-rs.

[67] Junwei Zhou, Junjiong Li, Emmanouil Panaousis, and Kaitai Liang. Deep Binarized Convolutional Neural Network Inferences over Encrypted Data. In *2020 7th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/2020 6th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, pages 160–167, August 2020. doi:10.1109/CSCloud-EdgeCom49738.2020.00035.

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