**Czech Technical University in Prague**

Faculty of Electrical Engineering

Department of Telecommunication Engineering

# DOCTORAL THESIS

**Privacy-Preserving Data Collection and Utilization using Provable Cryptographic Primitives**

| | |
|---|---|
| Author: | Jakub Klemsa |
| Supervisor: | doc. Ing. Lukáš Vojtěch, Ph.D. |
| Study programme: | P2612 Electrical Engineering & Information Technology |
| Specialization: | 2601V013 Telecommunication Engineering |
| Year of submission: | 2023 |

# Declaration

# Abstract

*Fully Homomorphic Encryption* (FHE) is a technique that ensures data *confidentiality* and enables *processing* them at the same time. In other words, *anyone* can evaluate an arbitrary (computable) function over encrypted data, without ever decrypting it. After evaluation, only the legitimate holder of the secret key is capable of decryption – what she finds is the same result as if the function were evaluated over plain, unencrypted data.

The history of FHE dates back to 1978 when Rivest et al. proposed the existence of an FHE scheme as a challenge. After more than 30 years, the challenge was resolved by Gentry in 2009 who gives a positive answer with his first-ever (publicly known) FHE scheme. However, the first-generation FHE schemes suffer from rather impractical computational overhead: reported by Gentry et al., a primitive bit-wise operation took around 30 minutes to evaluate on ordinary hardware. Since then, FHE schemes, as well as their implementations, made a tremendous progress: with state-of-the-art implementations, a similar operation only takes lower tens of milliseconds on ordinary hardware or about 500 µs on a specialized hardware (as of Q2/2023). Despite those speed-ups, the computational overhead of homomorphic evaluation using FHE over plain evaluation still remains significant.

This thesis studies manifold aspects of fully homomorphic encryption. In particular, it aims at improving FHE: (i) *correctness-wise*: analyze the error growth which is intrinsic to most existing FHE schemes; (ii) *performance-wise*: enhance the performance of selected homomorphic operations; and (iii) *scalability-wise*: enable multiple holders of the secret keying material.

One of the main contributions of this thesis is *fast integer arithmetic* over encrypted data, using the *TFHE Scheme* (Chillotti *et al.*, Asiacrypt '16) which is one of the state-of-the-art FHE schemes. Integer arithmetic operations are among the most fundamental ones, implemented as part of most microprocessors' instruction set. Therefore, there is a strong motivation to push the latency of their homomorphic counterparts as low as possible. The proposed approach employs a certain non-standard integer representation for which there exists a parallel addition algorithm, upon which other arithmetic operations are constructed. The benefit of parallelization of arithmetic operations stems from the parallelization-unfriendliness of the most expensive operation of TFHE. On the one hand, this approach may reduce the latency significantly – in particular for long inputs and with a sufficient amount of parallel resources. On the other hand, certain computational overhead is imposed which implies spending more overall processor time.

Besides improving the *performance* of fundamental homomorphic operations, a *variant* of FHE where multiple parties hold shares of the secret key is studied in this thesis and a new scheme is proposed. Compared to previous attempts, the new scheme not only outperforms them in terms of *latency*, it is also capable of a practical instantiation with an order of hundreds of parties—which the previous ones were not—thanks to its *low error growth.* Last but not least, a new method for a practical error assessment is proposed: based on noise measurements and two types of evaluation errors, this method captures even very low or overlapping probability distributions of incorrect homomorphic evaluation.

***Keywords***—Fully homomorphic encryption; Secure cloud computing; TFHE scheme.

# Abstrakt

*Plně homomorfní šifrování* (FHE z anglického *Fully Homomorphic Encryption*) je technika, která zajišťuje důvěrnost dat a zároveň umožňuje jejich zpracování. Jinými slovy, *kdokoliv* může vyhodnotit libovolnou (vyčíslitelnou) funkci nad zašifrovanými daty, aniž by ta musela být dešifrována. Výstup vyhodnocení pak dokáže dešifrovat pouze legitimní držitel soukromého klíče – touto cestou obdrží stejný výsledek, jako kdyby byla tato funkce vyhodnocena nad nezašifrovanými daty.

Historie FHE se píše od roku 1978, kdy Rivest a kol. představili otázku existence FHE jakožto výzvu. V roce 2009 byla tato výzva po více jak 30 letech úspěšně vyřešena Gentrym, který představil úplně první (veřejně známé) FHE schéma. Nicméně první generace FHE schémat trpí spíše nepraktickými výpočetními nároky: dle reportu Gentryho a kol. zabere výpočet jednoduché bitové operace okolo 30 minut na běžné výpočetní technice. Od té doby doznala FHE schémata, včetně jejich implementací, významného pokroku: s využitím dnešních předních implementací zabere podobná operace pouhé nižší desítky milisekund na běžné výpočetní technice nebo okolo 500 µs na specializované technice (stav ke 2. čtvrtletí 2023). Navzdory těmto urychlením zůstávají výpočetní nároky homomorfního vyhodnocování za použití FHE značné.

Tato práce studuje rozličné aspekty FHE. Jmenovitě se zabývá vylepšením FHE z pohledů: (i) *korektnosti*: analýza růstu chyb, které jsou vlastní většině existujících FHE schémat; (ii) *výpočetní náročnosti*: vylepšení výkonu vybraných homomorfních operací; a (iii) *škálovatelnosti*: umožnění rozdělení soukromého klíče mezi více držitelů.

Jedním z hlavních přínosů této práce je *rychlejší celočíselná aritmetika* nad zašifrovanými daty používající schéma TFHE (Chillotti a kol., Asiacrypt '16), které je jedním ze současných předních FHE schémat. Operace celočíselné aritmetiky se řadí mezi ty nejzákladnější a jsou součástí instrukční sady většiny mikroprocesorů, proto je zde silná motivace snížit časovou náročnost jejich homomorfních protějšků jak jen to bude možné. Navržený přístup využívá jistou nestandardní číselnou reprezentaci, pro kterou existuje paralelní sčítací algoritmus, na základě kterého jsou postaveny další aritmetické operace. Přínos paralelizace aritmetických operací je umocněn nepříliš dobrou paralelizovatelností nejnáročnější operace v TFHE. Na jednu stranu může tento přístup značně snížit časovou náročnost, obzvláště pro dlouhé vstupy a při velkém množství paralelních zdrojů, na druhou stranu jsou vynuceny vyšší výpočetní nároky, což může znamenat více celkového procesorového času.

Vedle zlepšování *výkonu* základních homomorfních operací studuje tato práce *variantu* FHE, ve které drží podíly soukromého klíče více stran, pro což je navrženo nové schéma. V porovnání s předchozími návrhy takovýchto schémat je toto nové schéma překoná nejen v ohledu časové náročnosti, ale je schopné praktického sestavení s řádově stovkami stran—čehož předchozí schémata schopná nebyla—díky jeho *nízkému růstu šumu*. Navíc je navržena nová metoda pro praktické vyhodnocení množství chyb založená na měřeních šumu a dvouch typech vyhodnocovacích chyb. Tato metoda tak zvládá zachytit i velmi nízké nebo překrývající se pravděpodobnostní rozdělení nesprávného homomorfního vyhodnocení.

***Klíčová slova***—Plně homomorfní šifrování; Bezpečné výpočty v cloudu; TFHE schéma.

# Acknowledgements

I would like to express my thanks to all the people who helped me complete this thesis, in particular to: Lukáš Kencl, who suggested I start doctoral studies at the Department of Telecommunication Engineering and who provided me with initial ideas; Lukáš Vojtěch for his valuable remarks and assistance throughout my studies; special thanks go to Melek Önen for her warm welcoming me at EURECOM for my research stays, lasting 25 months altogether. Also, I would like to thank all my co-authors for their contribution to our joint research outcomes. Last but not least, I appreciate all the comments of anonymous referees of my/our papers, submitted to various venues.

# Contents

# Preface

## 1 Foreword

*Cryptography* is a discipline that aims at securing information from undesirable abuse by third parties. From primitive ciphers used already in ancient civilizations, cryptography has undergone great development and has become an essential part of modern life. Indeed, from phone calls, instant messaging, or Internet banking to government-level communications – all are secured with cryptographic techniques.

State-of-the-art cryptographic methods are believed to be practically unbreakable: even for the hypothetical presence of a powerful quantum computer (which is anticipated to become true in a more or less distant future), there already exist symmetric as well as asymmetric cryptographic schemes for which no quantum attack is known[1]. However, although we may rely on the practical strength of cryptography, we are witnessing ever-lasting leaks of sensitive data[2].

Why is that?

The answer is not unambiguous: e.g., there might occur—either intended or unintended—misconduct of a responsible individual, like information disclosure, software bug, etc., possibly induced by social engineering, or a sophisticated attack that involves cracking the physical implementation might be mounted. What such attacks have in common is that they *never* aim at breaking the underlying cryptography.

A possible way to mitigate such attacks is to deploy countermeasures that may range from organizational to technical. Another way of protection is to decrease the attack surface by keeping the data encrypted *end-to-end* – let us outline a typical data lifecycle in its entirety.

Assume that a user holds a piece of data which she sends to a cloud. This enables the user to access her data from multiple devices or locations. Now, what if the data is very sensitive and/or the cloud is not fully trusted? The user may employ traditional cryptographic tools to protect her data before sending it for storage in the cloud – this prevents the data from being read or manipulated without being detected.

This way, things work perfectly fine until the cloud is supposed to *process* that data as part of a service offered by that cloud – for instance, we may think of photo-editing tools or advanced genome analytics using neural network inference. Such a scenario poses an Achilles' heel of traditional cryptography: indeed, data protection is limited to *data at rest* and *data in transit* while one important phase of a typical life-cycle of data is missing – *data in use.*

*Fully Homomorphic Encryption* (FHE) resolves this particular issue: it allows a third party (a cloud) to evaluate an arbitrary computable function over encrypted data, without ever decrypting it. This means that only the holder of the secret key can decrypt the result, and what

[1]More on the *Post-Quantum Cryptography* (PQC) standardization process can be found at `https://csrc.nist.gov/projects/post-quantum-cryptography`. Accessed: 2023-06-08.

[2]A review of data breaches of 2022 by Verizon can be found at `https://www.verizon.com/business/resources/reports/dbir/2023/wrap-up/`. Accessed: 2023-06-08.

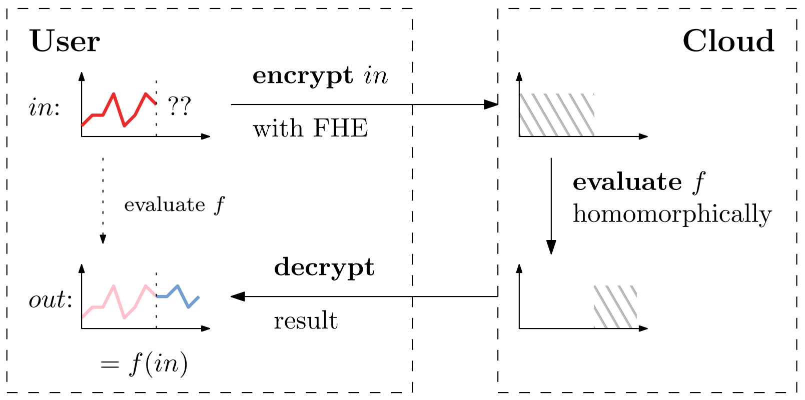


Figure 1: Example use-case of FHE: homomorphic evaluation of market prediction function $f$ over encrypted input data at Cloud. After User decrypts the result, she obtains the same as if the function $f$ was evaluated over the plain input *in* (see the dotted arrow).

she obtains is the same as if the function was evaluated right over the plain input data – provided that the cloud performs the computation honestly. Find an illustration of a basic use-case of FHE in Figure 1.

For the first time proposed as a challenge by Rivest et al. in 1978 [115], the existence of an FHE scheme remained an open question for more than 30 years until 2009 when Gentry presented his first-ever construction of an FHE scheme [58]. Although FHE would improve the security of various applications, the major obstacle that prevents FHE from massive deployment is its fairly high resource consumption and long evaluation times. From the initial attempts [59], which required around 30 minutes to evaluate a simple binary operation on ordinary hardware, FHE schemes and their implementations have been improved significantly over the years, pushing timing down to tens of milliseconds [132]. Additional speed-ups are expected, in particular with emerging attempts to design and build specialized hardware [126].

## 2 Basic Literature Overview

Below, we summarize the evolution of FHE schemes, followed by an overview of the state-of-the-art in a particular research branch relevant to this thesis.

### 2.1 Evolution of FHE Schemes

The evolution of FHE schemes is often divided into four generations, preceded by "pre-FHE" attempts.

#### Pre-FHE attempts

Before the discovery of the first *fully* homomorphic encryption scheme, there appear various schemes with certain (limited) homomorphic properties. Actually, the famous RSA scheme by Rivest et al. [116], presented in 1978, allows the multiplication of encrypted plaintexts[3]. This probably led Rivest et al. to suggest a hypothetical scheme [115] that would not only allow the multiplication of encrypted plaintexts but which would allow the evaluation of *any*

[3]Using a variant without padding, which is not semantically secure.

computable function over encrypted data. Note that (ring) multiplication is not sufficient: e.g., either (ring) addition *and* multiplication, or the Boolean gates AND *and* OR, or just the Boolean NAND gate is needed. Apart from RSA, other schemes satisfy a homomorphic property: e.g., ElGamal's cryptosystem [52] from 1985 which is multiplicatively homomorphic[4], or Paillier's cryptosystem [107] from 1999 which is additively homomorphic[5].

The first attempt towards the full homomorphism is a scheme by Boneh et al. [16] which enables the homomorphic evaluation of one level of multiplication, followed by any number of additions (represented by, e.g., a quadratic multi-variate polynomial). Other schemes by Sanders et al. [117] or by Ishai et al. [73] cover *branching programs* [12]. These schemes are referred to as *somewhat* homomorphic (SHE) and they paved the way towards the first actual fully homomorphic scheme.

**On the presence of noise** An important characteristic of SHE schemes—which is inherited by fully homomorphic schemes, too—is the presence of *noise*. In SHE, a small amount of noise, usually added to the encrypted message, is needed to achieve *security*. With each homomorphic operation, such noises accumulate – hence, on average, the noise grows. Ultimately, the noise may exceed a certain bound and damage the plaintext encoding, i.e., the result would not decrypt correctly. This is the main reason why the set of circuits that can be evaluated by SHE is fairly restricted. Also, the intrinsic presence of noise is the central difficulty that fully homomorphic schemes need to deal with.

### 1st generation

As outlined, the first publicly known construction of an FHE scheme was proposed by Gentry [58] in 2009, and it builds upon existing somewhat homomorphic schemes and ideal lattices. The paramount idea of Gentry's construction is to employ a somewhat homomorphic scheme that evaluates (i) a (couple of) homomorphic operation(s), followed by (ii) the decryption circuit. In the second step, the noise gets refreshed to an—on average—constant level. Such procedure is referred to as *bootstrapping*.

In an implementation report [59], authors present the performance of Gentry's scheme (including partial enhancements): for the presumably "most secure" set of parameters—which still achieves at most $\lambda = 72$ bits of security—one step comparable to the evaluation of a Boolean gate takes 30 minutes on an ordinary hardware[6]. A simplified variant of Gentry's scheme, which uses integers instead of ideal lattices in the underlying somewhat homomorphic scheme, was presented in 2011 by Van Dijk et al. (aka. DGHV; [127]).

### 2nd generation

Based on the *Learning With Errors* problem (LWE; [113]) and its *ring* variant (RLWE; [98]), a series of schemes have been proposed: most notably BGV [23] and B/FV [22, 53], which vastly improve the efficiency of FHE compared to the first generation. These schemes are still actively developed [4], in particular in the *leveled mode*—in which only a limited (multiplicative) depth of evaluated circuits is allowed—they remain competitive.

Other schemes of the 2nd generation build upon the NTRU cryptosystem [71] (e.g., LTV [97] or BLNN [19]), however, the underlying *overstretched NTRU assumption* has been shown vulnerable to attacks [6, 32].

[4]Shown to be IND-CCA1 secure [96] under a flavour of the decisional Diffie-Hellman assumption.
[5]Shown to be IND-CCA1 secure [10] under a flavour of the decisional composite residuosity assumption.
[6]Authors were using Intel Xeon E5450 processor (quad-core, introduced in 2007).

### 3rd generation

Another, (R)LWE-based scheme by Gentry et al. [61] (aka. GSW) laid the foundations for a branch of FHE schemes known for its fast bootstrapping. In particular, the FHEW scheme by Ducas et al. [50] is the first FHE scheme to achieve the bootstrapping time under a second, further reduced by the TFHE scheme [34, 35] which builds upon FHEW. With the state-of-the-art implementation [132] of TFHE, named `tfhe-rs`, bootstrapping only takes low tens of milliseconds (version `v0.2`).

There exist very recent results that aim at further improving the performance of bootstrapping; we choose to present two interesting lines of research that use fundamentally different approaches. First, the FINAL scheme [111], which is based on both LWE and NTRU[7], is anticipated to outperform TFHE by 28% in the bootstrapping time and by 45% in the key size[8]. Second, Lee et al. [95] suggest employing ring automorphisms which may lead to savings in any FHEW-based scheme, including TFHE. However, it has not been yet shown in a comparable setting whether any of these constructions practically outperforms the state-of-the-art `tfhe-rs` library.

### 4th generation

A scheme that directly implements (approximate) fixed-point arithmetic of real and complex numbers is known as CKKS [33], introduced in 2017. This is particularly useful for tasks of machine learning since such applications tolerate small errors. Due to its relatively costly bootstrapping procedure (introduced later in [30, 26]), CKKS is mostly used in the leveled mode. Another limitation is its relatively complex noise-propagation analysis, only given by Costache et al. [44] in 2022.

## 2.2 On the Comparison of FHE Schemes

Due to fundamental differences between FHE schemes across generations, there is no "absolute winner" FHE scheme. Indeed, each scheme has its strengths and weaknesses, and it usually depends on the particular use case which scheme is the most suitable. E.g., as already outlined, the CKKS scheme is useful when it comes to approximate arithmetic (e.g., in machine learning tasks), on the other hand, the TFHE scheme is useful for discrete computations when a bounded probability of errors is required (e.g., encrypted computations in a blockchain).

Besides that, it is hard to compare even similar schemes (or their implementations) *fairly*, for a variety of reasons:

- the performance of most FHE schemes depends heavily on the choice of parameters (there are usually tons of them), which is a non-trivial optimization task itself [13];
- FHE schemes (or their implementations) might offer different plaintext spaces and/or different homomorphic operations, e.g., Boolean plaintext space & the NAND gate [111] vs. a small additive group & look-up table evaluation [37];
- some schemes offer multiple plaintext "slots" in a single ciphertext (e.g., the CKKS scheme), which enables to process data in the SIMD manner (single instruction, multiple data);
- there might exist a highly optimized implementation for one scheme (e.g., an FPGA accelerator [126] for TFHE) which might not exist for any other scheme.

In this thesis, we primarily focus on the TFHE scheme.

[7]FINAL explicitly avoids the attacks on overstretched NTRU parameters (as outlined before).
[8]Authors compared both schemes with the binary plaintext space.

### 2.3 Limitations of Homomorphic Schemes

Below, we outline selected limitations of current homomorphic schemes:

**Slow & erroneous evaluation.** Compared to the evaluation over plaintext data, the computational overhead over encrypted data is still enormous, even though recent advances push it down significantly [132].

In addition, in selected implementations, there emerge implementation errors that might—under certain circumstances—exceed the inherent noise, in particular in TFHE. Namely, rounding errors are introduced by the use of a finite representation of real/complex numbers, which are employed within *Fast Fourier Transform* (FFT) that is used for fast modular polynomial multiplication.

**Limited support for multiple parties.** The construction of an FHE scheme that involves multiple holders of the private key (shares of the key) is not straightforward.

We discuss each topic in more detail in the next section.

## 3 Problem Statement

Currently, the TFHE scheme [34, 35] appears to be the most promising *general-purpose* FHE scheme: recall that TFHE is capable of evaluating *any* computable function (represented by an evaluation circuit of a certain form) over encrypted data. In particular, TFHE does not restrict the depth of the circuit—as opposed to leveled schemes—and it also achieves the best bootstrapping times among FHE schemes [68, 77][9]. On top of that, TFHE parameters can be practically tuned in such a way that the error probability of a single bootstrapping is as low as $2^{-50}$ [13]. We provide a comprehensive overview of the TFHE scheme in Chapter 1.

### 3.1 Erroneous Evaluation

As already outlined, selected TFHE implementations [105, 106, 124, 132] employ (a modified version of) FFT for fast modular polynomial multiplication. Since FFT works with real/complex numbers, their (finite) representation in a computer imposes rounding errors. To understand properly the *origin* and *propagation* of these rounding errors, we analyze the modified FFT algorithm in Chapter 2.

Note that besides FFT, some implementations [94, 125] employ *Number-Theoretic Transform* (NTT; [112]) instead of FFT (nuFHE [105] implements both). NTT is a derivative of FFT that works over finite fields which gives the advantage to represent the transformed image *without* loss of precision. Experiments show that FFT achieves better performance than NTT[10], however, optimizations might be introduced in the future to improve either approach.

### 3.2 Speeding-up Homomorphic Arithmetic (and more)

Although the performance of homomorphic evaluation of a function over encrypted data has been vastly improved since the first attempt by Gentry [59], fully homomorphic computations still consume a huge amount of resources. Indeed—roughly speaking—, within tens of milliseconds on an ordinary CPU, one may either run $10^8$ operations with 64-bit integers, or a single TFHE

[9] Summarized by Nigel Smart at `https://zama.ai/post/what-is-bootstrapping-homomorphic-encryption`. Accessed 2023-05-07.

[10] E.g., as measured by authors of nuFHE [105], the same setup with FFT is around $3\times$ faster than with NTT.

bootstrapping which is capable of processing only a couple of bits. Besides computation costs, which melt down to electricity bills and the price of hardware, possibly the most important measure of effectivity of homomorphic computations is the *latency*, i.e., how long time one needs to wait for the result. Therefore, there is a strong aim at parallelization whenever possible, even if this comes with a certain rise in computation costs.

Unfortunately, the central operation of TFHE's bootstrapping is *not* multi-thread friendly (except parallelization of polynomial multiplication itself), hence, it is worth considering parallelization at a higher level, running multiple bootstraps simultaneously. However, the same problem might re-emerge – the homomorphically evaluated function might not be multi-thread friendly. One such operation is among the most trivial ones, taught already at elementary schools: that is *addition with carry*.

In general, arithmetic operations with integers are among the most fundamental ones, supported by instruction sets of most microprocessors. This gives a strong motivation to implement them as efficiently as possible. However, using a standard (radix-based) representation, no algorithm for parallel multi-precision addition can exist [91]. Thankfully, in certain non-standard integer representations, parallel addition algorithm *does* exist, as shown by Avizienis [11] in 1961, later extended by Chow et al. [38].

Prior to our work [87], there were no attempts to employ a redundant representation to accelerate homomorphic arithmetic. Other works [124, 66, 132] suggest/implement the standard addition with carry while the `tfhe-rs` library [132] further implements arithmetic operations in a residual system based on the *Chinese Remainder Theorem* (CRT), their method is covered theoretically in [13].

In Chapter 3, we present a method that solves this particular problem: it employs the restricted "instruction set" of TFHE to implement parallel addition of (encrypted) multi-precision integers. Recall that TFHE is only capable of working with fairly short integers (practically up to 8 bits [13]), hence multi-precision is truly needed. On top of parallel addition, other arithmetic and selected comparison-based operations are proposed and implemented.

## 3.3 Multi-Key/Multi-Party Homomorphic Encryption

In the constructions discussed so far, the secret (decryption) key is held by a single party. To allow homomorphic evaluations over data encrypted with keys held by multiple parties—e.g., data may originate from multiple sources or the secret key is distributed in shares across multiple parties—, *multi-key* or *multi-party* homomorphic encryption comes into play.

- In the *multi-key* setup, defined by López-Alt et al. [97], parties may generate their keys independently of each other, and the result of the homomorphic evaluation is only encrypted with the keys of parties whose ciphertexts were involved in the computation.
- The *multi-party* scenario is a special case of *threshold* encryption where the threshold equals the number of parties. In this setup, parties need to collaborate to obtain the common public key, and also the result can only be decrypted using the secret keys of all parties.

First attempts to design an FHE scheme with multiple parties [40, 24, 29] are rather impractical (and never implemented). The first practical[11] multi-key scheme is proposed by Chen et al. [27]. Recently, a scheme by Kim et al. [79] improves the evaluation complexity from quadratic to quasi-linear in the number of involved parties.

In Chapter 4, we propose a new (somehow hybrid) multi-key scheme based on TFHE. Not only achieves our scheme *linear* evaluation time with a lower constant factor than [79], it also

[11] Proof-of-concept implementation is available at `https://github.com/ilachill/MK-TFHE`.

enjoys a low error growth which allows us to practically instantiate our scheme with up to 128 parties, compared to maximums of 16 and 32 for [27] and [79], respectively. Last but not least, we show that the scheme by Kim et al. suffers from an enormous error growth due to FFT which is used in their proof-of-concept implementation[12].

[12]Available at `https://github.com/SNUCP/MKTFHE`.

Chapter 1

# A Guide to the TFHE Scheme

Also referred to as the holy grail of cryptography, *Fully Homomorphic Encryption* (FHE) allows for arbitrary calculations over encrypted data. First proposed as a challenge by Rivest *et al.* in 1978, the existence of an FHE scheme has only been shown by Gentry in 2009. However, until these days, existing general-purpose FHE schemes suffer from a substantial computational overhead, which has been vastly reduced since the original construction of Gentry, but it still poses an obstacle that prevents a massive practical deployment of FHE. The *TFHE scheme* by Chillotti *et al.* represents the state-of-the-art among general-purpose FHE schemes. This chapter aims to serve as a thorough guide to the TFHE scheme, with a strong focus on the reliability of computations over encrypted data, and help researchers and developers understand the internal mechanisms of TFHE in detail. In particular, it may serve as a baseline for future improvements and/or modifications of TFHE or related schemes.

## 1.1 Introduction to (T)FHE

*Fully Homomorphic Encryption* (FHE), first discovered by Gentry [58] in 2009, enables the evaluation of an arbitrary (computable) function over encrypted data. As a basic use-case of FHE, we outline a secure cloud-aided computation: we describe how a user (**U**) may delegate a computation over her sensitive data to a semi-trusted cloud (**C**):

- **U** generates secret keys $\mathsf{sk}$, and (public) evaluation keys $\mathsf{ek}$, which she sends to **C**;
- **U** encrypts her sensitive data $d$ with $\mathsf{sk}$, and sends the encrypted data to **C**;
- **C** employs $\mathsf{ek}$ to evaluate function $f$, homomorphically, over the encrypted data (i.e., without ever decrypting it), yielding an encryption of $f(d)$, which it sends back to **U**;
- **U** decrypts the message from **C** with $\mathsf{sk}$, obtaining the desired result: $f(d)$ in plain.

We illustrate the concept of homomorphic evaluation of a function over the plain vs. over the encrypted data in Figure 1.1.

Among existing FHE schemes, two main means of homomorphic evaluation can be identified: (i) the *leveled* approach (e.g., [23, 33]), and (ii) the *bootstrapped* approach (e.g., [58, 35]), while a combination of both is also possible [30]. The leveled approach (i) is particularly useful for functions that are represented by an evaluation circuit of limited depth which is known in advance. After evaluating it, no additional operation can be performed with the result, otherwise, the data might be corrupted. Based on the circuit's depth and other properties, parameters must be

$$\begin{tikzcd}
 & \textbf{Plain data } d \arrow[r, "\text{FHE encryption}"] \arrow[d] & \textbf{Encrypted data } \mathsf{Encr}(d) \arrow[d] & \\
\text{User} & \text{evaluate } f \arrow[d] & \text{evaluate } f \text{ \textit{homomorphically}} \arrow[d] & \text{Cloud} \\
 & \textbf{Plain result } f(d) & \textbf{Encrypted result } \mathsf{Encr}\big(f(d)\big) \arrow[l, "\text{FHE decryption}"'] &
\end{tikzcd}$$

Figure 1.1: Illustration of an evaluation of function $f$ over the plain and over the encrypted data in the User's and in the Cloud's domain, respectively. In both ways, the same result is obtained.

chosen accordingly. On the other hand, for the bootstrapped approach (ii), there is no limit on the circuit depth, which also means that the circuit does not need to be known in advance. The TFHE scheme by Chillotti et al. [35] is currently considered as the state-of-the-art FHE scheme that follows the bootstrapped approach.

For a basic overview of the evolution of FHE schemes, we refer to a survey by Acar et al. [2] (from 2018; in particular for implementations, much progress has been made since then).

### 1.1.1 Basic Overview of TFHE

First, let us provide a high-level overview of TFHE and its abilities, and let us outline its structure.

Similar to many other FHE schemes, the TFHE scheme builds upon the *Learning With Errors* (LWE) encryption scheme, first introduced by Regev [113]. There are two important properties of LWE: *additive homomorphism*, and the *presence of noise*; let us comment on either:

**Additive homomorphism:** LWE ciphertexts, also referred to as *samples*, are represented by vectors of (additive) group elements. By the nature of LWE encryption, LWE samples are additively homomorphic, which means that—roughly speaking—for two samples $\mathbf{c}_1$ and $\mathbf{c}_2$, which encrypt respectively $\mu_1$ and $\mu_2$, it holds that $\mathbf{c}_1 + \mathbf{c}_2$ encrypts $\mu_1 + \mu_2$.

**Noise:** To achieve security, LWE samples need to contain a certain amount of noise. However, with each homomorphic addition, noises also add up, which may ultimately destroy the accuracy/correctness of the plaintext.

Like many other FHE schemes, TFHE deals with the noise growth by defining a procedure referred to as *bootstrapping*. Bootstrapping aims at resetting the noise to an—on average—fixed level. Otherwise, if the noise exceeded a certain bound, the probability of correct decryption would drop rapidly.

To sum up, TFHE offers two operations: (i) *homomorphic addition*, and (ii) *bootstrapping*. Homomorphic addition (i) is a very cheap operation, however, the noise accumulates. On the other hand, bootstrapping (ii) is a costly operation, but it refreshes the noise and—in case of TFHE—it is inherently capable of evaluating homomorphically a custom *Look-Up Table* (LUT), which can be moreover encrypted. These two operations are sufficient for the *full homomorphism*, i.e., the possibility to evaluate any computable function over encrypted data. In Figure 1.2, we introduce the TFHE *gate*, which comprises (i) homomorphic addition(s), grouped into a homomorphic *dot-product* with integer weights, followed by (ii) TFHE bootstrapping. For bootstrapping, we outline its internal structure that consists of four sub-operations: KeySwitch, ModSwitch, BlindRotate and SampleExtract.

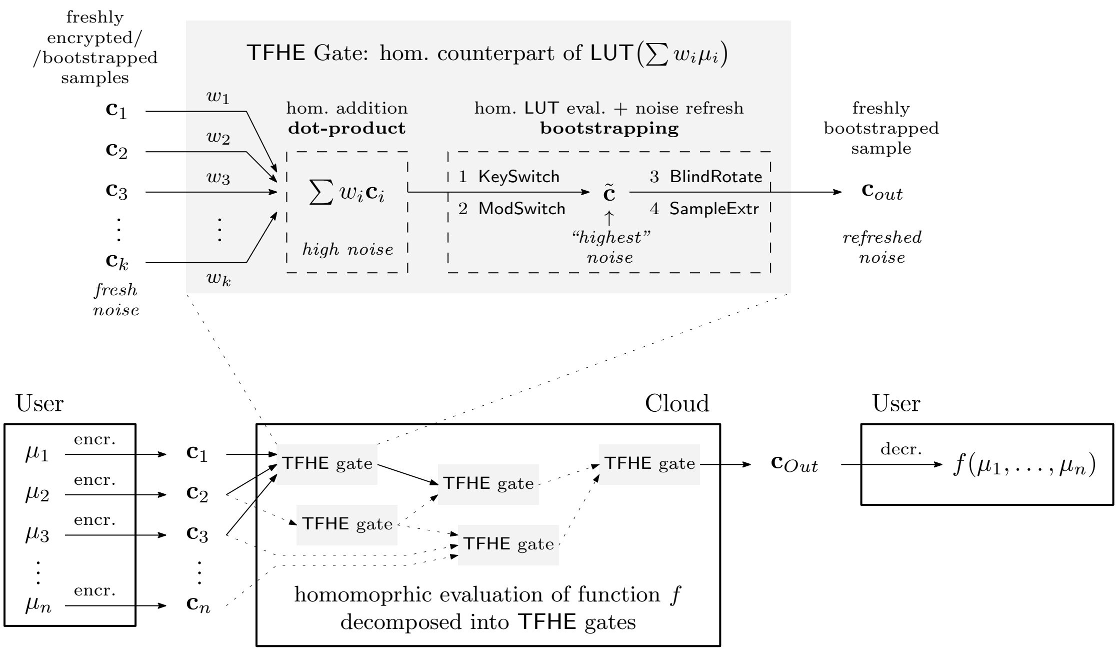

freshly encrypted/ /bootstrapped samples
TFHE Gate: hom. counterpart of $\mathsf{LUT}(\sum w_i\mu_i)$
hom. addition dot-product
hom. LUT eval. + noise refresh bootstrapping
freshly bootstrapped sample
$\mathbf{c}_1$ $\mathbf{c}_2$ $\mathbf{c}_3$ $\mathbf{c}_k$
$w_1$ $w_2$ $w_3$ $w_k$
$\sum w_i\mathbf{c}_i$
high noise
1 KeySwitch
2 ModSwitch
$\tilde{\mathbf{c}}$
3 BlindRotate
4 SampleExtr
"highest" noise
$\mathbf{c}_{out}$
refreshed noise
fresh noise
User
Cloud
User
$\mu_1$ $\mu_2$ $\mu_3$ $\mu_n$
encr.
$\mathbf{c}_n$
TFHE gate
$\mathbf{c}_{Out}$
decr.
$f(\mu_1,\dots,\mu_n)$
homomoprhic evaluation of function $f$ decomposed into TFHE gates

Figure 1.2: TFHE gate: homomorphic addition(s) and bootstrapping, which comprises four sub-operations. The sample $(b', \mathbf{a}')$ may proceed to another TFHE gate, or it may go to the output and decryption.

### 1.1.2 Aim of this Chapter

The aim of this chapter is to provide the reader—mainly intended for FHE researchers and developers—with a comprehensive and intelligible guide to the TFHE scheme. In particular, in this chapter, we thoroughly analyze the noise growth of TFHE's operations, which is decisive for the correctness and reliability of homomorphic evaluations in the wild. Besides that, we comment on the purpose of selected tricks that are intended to decrease the noise growth. Therefore, our TFHE guide is supposed to provide useful insights for any prospective improvements and/or design modifications to TFHE(-related schemes).

**Related Work** The original full paper on TFHE [35] is followed by other papers that recall or redefine TFHE in numerous ways [66, 37, 27, 93], while changing the notation as well as the approach to describe the bootstrapping procedure.

Joye [75] provides a SoK paper on TFHE, supported by many examples with concrete values. Our TFHE guide complements their SoK in particular by providing a thorough noise growth analysis, which is one of its pillars.

### 1.1.3 Chapter Outline

We introduce building blocks of the TFHE scheme in Section 1.2. Next, in Section 3.2.1, we describe the construction of TFHE in detail with a particular focus on noise propagation. We further focus on the correctness of homomorphic evaluation in Section 1.4. In Section 1.5, we briefly comment on implementation aspects of TFHE. We conclude this chapter in Section 4.6.

## 1.2 Building Blocks of TFHE

In this section, we first briefly outline flavors of LWE, we outline a technical notion, referred to as the *concentrated distribution*, and we provide a list of symbols and notation. Then, we introduce in detail a generalized variant of LWE, denoted GLWE, and we comment on its additive homomorphism, security and other properties. Finally, we define the *decomposition* operation that we use to build up a compound scheme called GGSW, which enables multiplicative homomorphism.

**The Torus and the Ring Variant of LWE** Internally, TFHE employs two variants of LWE, originally referred to as TLWE and TRLWE, which stand for *(Ring)* LWE *over the Torus*. In a nutshell, let us outline what *torus* and *ring* mean in this context.

The *torus* is the underlying additive group of LWE that is used in TFHE, denoted $\mathbb{T}$ and defined as $\mathbb{T} \coloneqq \mathbb{R}/\mathbb{Z}$ with the addition operation. The torus can be represented by the interval $[0, 1)$, with each addition followed by reduction $\bmod 1$, e.g., $0.3 + 0.8 = 0.1$. Since $\mathbb{T}$ is an abelian group, we may perceive $\mathbb{T}$ as an algebraic $\mathbb{Z}$*-module*, i.e., we further have scalar multiplication $\mathbb{Z} \times \mathbb{T} \to \mathbb{T}$, defined as repeated addition.

The *ring* variant of LWE, introduced by Lyubashevsky et al. [98], extends the module's ring to a ring of polynomials with a bounded degree. In TFHE, we will work with the ring $\mathbb{Z}[X]/(X^N + 1)$, denoted $\mathbb{Z}^{(N)}[X]$, with $N$ a power of two. Then, the underlying $\mathbb{Z}^{(N)}[X]$-module comprises torus polynomials modulo $X^N + 1$, denoted $\mathbb{T}^{(N)}[X]$.

**Concentrated Distribution** Unlike (scalar) multiplication, the division of a torus element by an integer cannot be defined without ambiguity, the same holds for the expectation of a distribution over the torus. However, this can be fixed for a *concentrated distribution* [35], which is a distribution with support limited to a ball of radius $1/4$, up to a negligible subset. For further details, we refer to [35].

**Symbols & Notation** Throughout this chapter, we use the following symbols & notation; we denote:

- $\mathbb{B} \coloneqq \{0, 1\} \subset \mathbb{Z}$ the set of binary coefficients,
- $\mathbb{T}$ the additive group $\mathbb{R}/\mathbb{Z}$, referred to as the *torus* (i.e., real numbers modulo 1),
- $\mathbb{Z}_n$ the quotient ring $\mathbb{Z}/n\mathbb{Z}$ (or its additive group),
- $\lfloor \cdot \rceil : \mathbb{R} \to \mathbb{Z}$ the standard rounding function,
- for vector $\mathbf{v}$, $v_i$ stands for its $i$-th coordinate,
- $\langle \mathbf{u}, \mathbf{v} \rangle$ the dot product of two vectors $\mathbf{u}$ and $\mathbf{v}$,
- $M^{(N)}[X]$ the additive group (or ring) of polynomials modulo $X^N + 1$ with coefficients from $M$, where $N \in \mathbb{N}$ is a power of two,
- for polynomial $p(X)$, $p^{(i)}$ stands for the coefficient of $p$ at $X^i$,
- for vector of polynomials $\mathbf{w}$, $w_i^{(j)}$ stands for the coefficient at $X^j$ of the $i$-th coordinate of $\mathbf{w}$,
- $\|p(X)\|_2^2$ the square of the $l^2$-norm of polynomial $p(X)$ (i.e., the sum of squared coefficients),
- $a \overset{\$}{\leftarrow} M$ the uniform draw of random variable $a$ from $M$,

- $a \overset{\alpha}{\leftarrow} M$ the draw of random variable $a$ from $M$ with distribution $\alpha$ (for $\alpha \in \mathbb{R}$, we consider the zero-centered /discrete/ Gaussian draw with standard deviation $\alpha$),
- $\mathrm{E}[X]$, $\mathsf{Var}[X]$ the expectation and the variance of random variable $X$, respectively.

### 1.2.1 Generalized LWE

First, we define a generalized variant of the LWE scheme, referred to as GLWE, which combines plain LWE with its ring variant. We define GLWE solely over the torus, although another underlying structure might be used, e.g., $\mathbb{Z}_q$ with prime $q$ that is taken in some other schemes.

**Definition 1.1** (GLWE Sample)**.** Let $k \in \mathbb{N}$ be the *dimension*, $N \in \mathbb{N}$, $N$ a power of two, be the *degree*, $\alpha \in \mathbb{R}_0^+$ be the *standard deviation* of the noise, and let the plaintext space $\mathcal{P} = \mathbb{T}^{(N)}[X]$, the ciphertext (sample) space $\mathcal{C} = \mathbb{T}^{(N)}[X]^{1+k}$ and the key space $\mathcal{K} = \mathbb{Z}^{(N)}[X]^k$. For $\mu \in \mathcal{P}$ and $\mathbf{z} \overset{\chi}{\leftarrow} \mathcal{K}$, where $\chi$ is a *key distribution*, we call $\bar{\mathbf{c}} = (b, \mathbf{a}) =: \mathsf{GLWE}_{\mathbf{z}}(\mu)$ the GLWE *sample* of message $\mu$ under key $\mathbf{z}$, if

$$b = \mu - \langle \mathbf{z}, \mathbf{a} \rangle + e, \tag{1.1}$$

where $\mathbf{a} \overset{\$}{\leftarrow} \mathbb{T}^{(N)}[X]^k$ and $e \overset{\alpha}{\leftarrow} \mathbb{T}^{(N)}[X]$.

If $\mathbf{a} = \mathbf{0}$, we call the sample *trivial*, and if $\mu = \mathbf{0}$, we call the sample *homogeneous.* We denote $\bar{\mathbf{z}} := (1, \mathbf{z}) \in \mathbb{Z}^{(N)}[X]^{1+k}$, referred to as the *extended key.* For $N = 1$, we have the (plain) LWE *sample* and we usually denote its dimension by $n$. We also generalize GLWE sampling to vector messages, yielding a matrix of $1 + k$ columns, with one GLWE sample per row.

GLWE sampling is actually *encryption*: in TFHE, plaintext data is encrypted using the plain LWE, while GLWE is used internally. To *decrypt*, we apply the GLWE *phase* function (followed by rounding if applicable).

**Definition 1.2** (GLWE phase)**.** Let $k$, $N$ and $\alpha$ be GLWE parameters as per Definition 1.1, and let $\bar{\mathbf{c}} = (b, \mathbf{a})$ be a GLWE sample of $\mu$ under GLWE key $\mathbf{z}$. We call the function $\varphi_{\mathbf{z}} \colon \mathbb{T}^{(N)}[X] \times \mathbb{T}^{(N)}[X]^k \to \mathbb{T}^{(N)}[X]$,

$$\varphi_{\mathbf{z}}(b, \mathbf{a}) = b + \langle \mathbf{z}, \mathbf{a} \rangle = \langle \bar{\mathbf{z}}, \bar{\mathbf{c}} \rangle, \quad (= \mu + e), \tag{1.2}$$

the GLWE *phase.* We call the sample $\bar{\mathbf{c}}$ *valid* iff the distribution of $\varphi_{\mathbf{z}}(\bar{\mathbf{c}})$ is concentrated. Finally, for valid sample $\bar{\mathbf{c}}$, we call $\mathsf{msg}_{\mathbf{z}}(\bar{\mathbf{c}}) := \mathrm{E}\big[\varphi_{\mathbf{z}}(\bar{\mathbf{c}})\big]$ the *message* of $\bar{\mathbf{c}}$, which equals $\mu$, since the noise is zero-centered and concentrated.

*Remark* 1.1. GLWE phase returns $\mu + e$, i.e., the original message with a small amount of noise. We may define GLWE decryption as either:

1. an erroneous decryption via GLWE phase – we accept some errors in the decrypted result, which might be considered harmless or even useful, e.g., in the context of differential privacy [51]; or
2. a correctable decryption – for this purpose, we need to control the amount of noise and follow GLWE phase by an appropriate rounding step (relevant for this chapter); or
3. an expectation of GLWE phase, i.e., $\mathsf{msg}_{\mathbf{z}}(\bar{\mathbf{c}})$ – this is useful for formal definitions and proofs.

In the following theorem, we state the additively homomorphic property of GLWE.

**Theorem 1.1** (Additive Homomorphism). *Let $\bar{\mathbf{c}}_1, \ldots, \bar{\mathbf{c}}_n$ be valid and independent* $\mathsf{GLWE}$ *samples under* $\mathsf{GLWE}$ *key* $\mathbf{z}$ *and let $w_1, \ldots, w_n \in \mathbb{Z}^{(N)}[X]$ be integer polynomials (weights). In case $\bar{\mathbf{c}} = \sum_{i=1}^{n} w_i \cdot \bar{\mathbf{c}}_i$ is a valid* $\mathsf{GLWE}$ *sample, it holds*

$$\mathsf{msg}_{\mathbf{z}}\Big(\sum_{i=1}^{n} w_i \cdot \bar{\mathbf{c}}_i\Big) = \sum_{i=1}^{n} w_i \cdot \mathsf{msg}_{\mathbf{z}}(\bar{\mathbf{c}}_i) \tag{1.3}$$

*and for the noise variance*

$$\mathsf{Var}[\bar{\mathbf{c}}] = \sum_{i=1}^{n} \|w_i\|_2^2 \cdot \mathsf{Var}[\bar{\mathbf{c}}_i]. \tag{1.4}$$

*If all samples $\bar{\mathbf{c}}_i$ have the same variance $V_0$, we have $\mathsf{Var}[\bar{\mathbf{c}}] = V_0 \cdot \sum_{i=1}^{n} \|w_i\|_2^2$ and we define*

$$\nu^2 \coloneqq \sum_{i=1}^{n} \|w_i\|_2^2, \tag{1.5}$$

*referred to as the* quadratic weights*. We refer to the operation* (1.3) *as the (homomorphic)* dot product ($\mathsf{DP}$)*.*

### Discrete-Valued Plaintext Space

As outlined in Remark 1.1, item 2, in this chapter, we focus on a variant of $\mathsf{GLWE}$ that restricts its messages to a discrete subspace of the entire torus plaintext space. Denoted by $\mathcal{M}$, we refer to the plaintext subspace as the *cleartext space*, leaving the term *plaintext space* for torus polynomials.

In this chapter, we only focus on the cleartext space of the form $\mathcal{M} = \frac{1}{2^\pi}\mathbb{Z}/\mathbb{Z} \subset \mathbb{T}$ (a subgroup of $\mathbb{T}$ isomorphic to $\mathbb{Z}_{2^\pi}$), where we refer to the parameter $\pi$ as the *cleartext precision.* In terms of Definition 1.2, if it holds for the noise $e$ that $|e| < 1/2^{\pi+1}$, then rounding of the value $\varphi_{\mathbf{z}}(b, \mathbf{a}) \in \mathbb{T}$ to the closest element of $\mathcal{M}$ leads to the correct decryption/recovery of $\mu$.

### Discretized Torus

For the sake of simplicity of the noise growth analysis, $\mathsf{TFHE}$ is defined over the continuous torus, whereas in implementation, a discretized finite representation must be used instead. To cover the unit interval uniformly, $\mathsf{TFHE}$ implementations use an integral type—usually 32- or 64-bit `(u)int`—to represent a torus element, where we denote the bit-precision by $\tau$. E.g., for $\tau = 32$-bit `uint32` type, $t \in$ `uint32` represents $t/2^{32} \in \mathbb{T} \sim [0, 1)$, where the denominator is usually denoted by $q = 2^\tau$ (in this case $q = 2^{32}$). Using such a representation, we effectively restrict the torus $\mathbb{T}$ to its submodule $\mathbb{T}_q \coloneqq q^{-1}\mathbb{Z}/\mathbb{Z} \subset \mathbb{T}$.

### Distribution of $\mathsf{GLWE}$ Keys

For the coefficients of $\mathsf{GLWE}$ keys, a ternary distribution $\chi_p \colon (-1, 0, 1) \to (p, 1 - 2p, p)$, parameterized by $p \in (0, 1/2)$, can be used. In particular, uniform ternary distribution is suggested by a draft of the homomorphic encryption standard [7], and it also is widely adopted by main FHE libraries like HElib [72], Lattigo [101], SEAL [99], or HEAAN [121], although they implement other schemes than $\mathsf{TFHE}$. With a fixed $\mathsf{GLWE}$ dimension and carefully chosen $p$, the distribution $\chi_p$ may achieve better security as well as lower noise growth than uniform binary $\mathcal{U}_2 \colon (0, 1) \to (1/2, 1/2)$. On the other hand, it is worth noting that for "small" values of $p$, such keys are also referred to as *sparse keys* (in particular with a fixed/limited Hamming weight), and there exist specially tailored attacks [31, 123]; we discuss security in the following paragraphs.

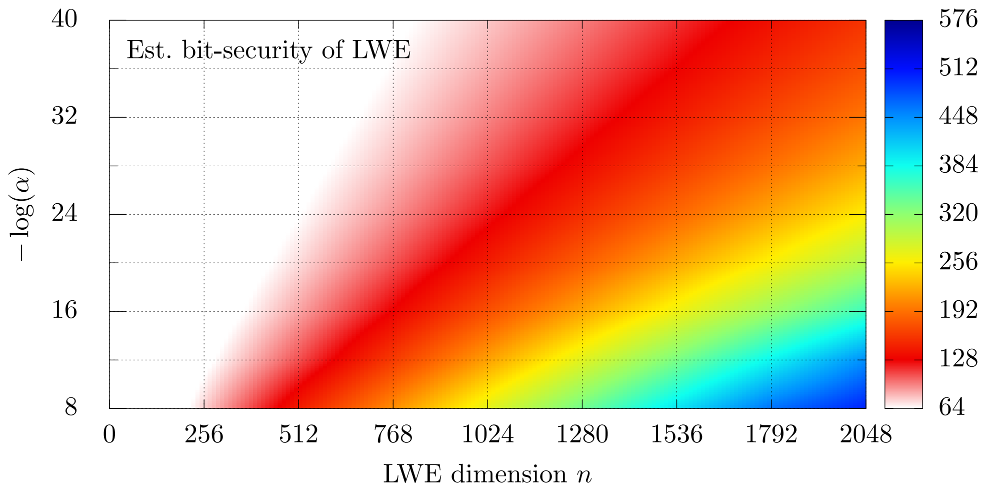


Figure 1.3: Bit-security of LWE as estimated by `lattice-estimator` by Albrecht et al. [8, 9] (commit ID `f9dc7c`), using underlying group size $q = 2^{64}$. Interpolated between grid points. Raw data can be found at `https://github.com/fakub/LWE-Estimates`.

*Note* 1.2. In TFHE, one instance of LWE and one of GLWE is employed. For LWE keys, usually, uniform binary distribution is used for technical reasons, although attempts to extend the key space can be found in the literature [76]. For GLWE keys, a ternary distribution can be used immediately.

### Security of (G)LWE

Estimation of the security of (G)LWE encryption is a complex task: it depends on (i) the size of the secret key (i.e., the dimension and/or the polynomial degree), (ii) the distribution of its coefficients, (iii) the distribution of the noise, which is usually given by its standard deviation, denoted by $\alpha$, and (iv) the underlying structure (usually integers modulo $q$). As a rule of thumb, it holds that *the longer key, the better security*, as well as *the greater noise, the better security.*

A state-of-the-art tool that implements an LWE security assessment is known as `lattice--estimator` – a tool by Albrecht et al. [8, 9]. Authors aim at considering all known relevant attacks on LWE, including those targeting sparse keys, as outlined previously. A plot that shows selected results of `lattice-estimator` can be found in Figure 1.3. A code example of the usage of `lattice-estimator` as well as raw data that were used to generate the figure can be found in our repository[1].

[1]`https://github.com/fakub/LWE-Estimates`

**Balancing Parameters**

The downside of increasing the key size (improves *security*) is longer evaluation time (reduces *performance*), similarly increasing the amount of noise (improves *security*) leads to an error-prone evaluation (reduces *correctness*). Therefore, the goal is to find the best balance within the triangle of somehow orthogonal goals:

$$\begin{tikzcd}
\textit{security} \arrow[rr, leftrightarrow] \arrow[dr, leftrightarrow] & & \textit{correctness} \arrow[dl, leftrightarrow] \\
& \textit{performance} &
\end{tikzcd}$$

The problem of finding such a balance is thoroughly studied by Bergerat et al. [13], who provide concrete results that aim at achieving the best *performance*, without sacrificing *security*, nor *correctness*.

## 1.2.2 Decomposition

To enable homomorphic multiplication and at the same time to reduce its noise growth, torus elements get decomposed into a series of integers. The operation is parameterized by (i) the *decomposition base* (denoted $B$; we only consider $B = 2^\gamma$ a power of two), and (ii) by the *decomposition depth* (denoted $d$). We further denote

$$\mathbf{g} \coloneqq (1/B, 1/B^2, \dots, 1/B^d), \tag{1.6}$$

referred to as the *gadget vector*. We define *gadget decomposition of* $\mu \in \mathbb{T} \sim [-1/2, 1/2) \subset \mathbb{R}$, denoted $\mathbf{g}^{-1}(\mu)$, as the base-$B$ representation of $\tilde{\mu} = \lfloor B^d \cdot \mu \rceil \in \mathbb{Z}$ (multiplied in $\mathbb{R}$) in the alphabet $[-B/2, B/2) \cap \mathbb{Z}$. Note that such decomposition is unique. For the *decomposition error*, it holds that

$$\left|\mu - \langle \mathbf{g}, \mathbf{g}^{-1}(\mu) \rangle\right| \leq 1/2B^d. \tag{1.7}$$

We denote

$$\varepsilon^2 \coloneqq \frac{1}{12B^{2d}} \quad \text{and} \tag{1.8}$$

$$V_B \coloneqq \frac{B^2+2}{12} \tag{1.9}$$

the variance of the decomposition error and the mean of squares of the alphabet $[-B/2, B/2) \cap \mathbb{Z}$ (n.b., we assume $B$ is even), respectively; for both we consider a uniform distribution. Note that with the alphabet $[0, B) \cap \mathbb{Z}$, the respective value of $V_B$ would have been higher, i.e., this is one of the little tricks to reduce later the noise growth.

For $k \in \mathbb{N}$, $k \geq 2$, we further denote

$$\mathbf{G}_k \coloneqq \mathbf{I}_k \otimes \mathbf{g}, \tag{1.10}$$

where $\mathbf{I}_k$ is identity matrix of size $k$ and $\otimes$ stands for the tensor product, i.e., we have $\mathbf{G}_k \in \mathbb{T}^{kd \times k}$, referred to as the *gadget matrix*.

We generalize $\mathbf{g}^{-1}$ to torus vectors and torus polynomials (and their combination) in a natural way: for vector $\mathbf{t} \in \mathbb{T}^n$, $\mathbf{g}^{-1}(\mathbf{t})$ is the concatenation of respective component-wise decompositions $\mathbf{g}^{-1}(t_i)$, for polynomial $t \in \mathbb{T}^{(N)}[X]$, $\mathbf{g}^{-1}(t)$ proceeds coefficient-wise, i.e., the output is a vector of integer polynomials. Finally, for a vector of torus polynomials, $\mathbf{g}^{-1}$ outputs a concatenation of respective vectors of integer polynomials.

### 1.2.3 GGSW & Homomorphic Multiplication

Unlike GLWE, which encrypts torus polynomials, GGSW encrypts integer polynomials. The main aim of GGSW is to allow homomorphic multiplication of a (GLWE-encrypted) torus polynomial by a (GGSW-encrypted) integer polynomial. The multiplicative homomorphic operation is referred to as the *External Product*, denoted by $\boxdot$: GGSW $\times$ GLWE $\to$ GLWE.

**Definition 1.3** (GGSW Sample)**.** Let $k$, $N$ and $\alpha$ be the parameters of a GLWE instance with key $\mathbf{z}$. We call $\bar{\mathbf{C}} = \bar{\mathbf{Z}} + m \cdot \mathbf{G}_{1+k}$, $\bar{\mathbf{C}} \in \mathbb{T}^{(N)}[X]^{(1+k)d,1+k}$, the GGSW *sample* of $m \in \mathbb{Z}^{(N)}[X]$ if rows of $\bar{\mathbf{Z}}$ are mutually independent, homogeneous GLWE samples under the key $\mathbf{z}$. We call the sample *valid* iff there exists $m \in \mathbb{Z}^{(N)}[X]$ such that each row of $\bar{\mathbf{C}} - m \cdot \mathbf{G}_{1+k}$ is a valid homogeneous GLWE sample.

**Definition 1.4** (External Product)**.** For GLWE sample $\bar{\mathbf{c}} = (b, \mathbf{a}) \in \mathbb{T}^{(N)}[X]^{1+k}$ and GGSW sample $\bar{\mathbf{A}}$ of corresponding dimensions, we define the *External Product*, $\boxdot$: GGSW $\times$ GLWE $\to$ GLWE, as

$$\mathbf{g}^{-1}(\bar{\mathbf{c}})^T \cdot \bar{\mathbf{A}} =: \bar{\mathbf{A}} \boxdot \bar{\mathbf{c}}. \tag{1.11}$$

In the following theorem, we state the multiplicative homomorphic property of the external product and we evaluate its excess noise.

**Theorem 1.2** (Correctness & Noise Growth of $\boxdot$)**.** *Given* GLWE *sample* $\bar{\mathbf{c}}$ *of* $\mu_c \in \mathbb{T}^{(N)}[X]$ *under* GLWE *key* $\mathbf{z}$ *and noise parameter* $\alpha$*, and* GGSW *sample* $\bar{\mathbf{A}}$ *of* $m_A \in \mathbb{Z}^{(N)}[X]$ *under the same key and noise parameters, external product returns* GLWE *sample* $\bar{\mathbf{c}}' = \bar{\mathbf{A}} \boxdot \bar{\mathbf{c}}$*, which holds excess noise* $e_\boxdot$*, given by* $\langle \bar{\mathbf{z}}, \bar{\mathbf{c}}' \rangle = m_A \cdot \langle \bar{\mathbf{z}}, \bar{\mathbf{c}} \rangle + e_\boxdot$*, for which it holds*

$$\begin{aligned}\mathsf{Var}[e_\boxdot] \approx {} & \underbrace{dNV_B\alpha^2(1+k)}_{\textit{amplified } \mathsf{GGSW} \textit{ noise}} + \\ & + \underbrace{\|m_A\|_2^2 \cdot \varepsilon^2(1+kNV_{\mathbf{z}})}_{\textit{decomp. errors}},\end{aligned} \tag{1.12}$$

*where* $V_{\mathbf{z}}$ *is the variance of individual coefficients of the* GLWE *key* $\mathbf{z}$ *and other parameters are as per previous definitions. If* $e_\boxdot$ *and the noise of* $\bar{\mathbf{c}}$ *are "sufficiently small",* $\bar{\mathbf{c}}'$ *encrypts* $\mathsf{msg}_{\mathbf{z}}(\bar{\mathbf{c}}') = m_A \cdot \mu_c$*, i.e., external product is indeed multiplicatively homomorphic.*

*Proof.* Find the proof in Appendix B.1. □

## 1.3 Constructing the TFHE Scheme

By far, there are two issues with (G)LWE:

1. As shown in Theorem 1.1, each additive homomorphic operation over (G)LWE samples leads to *noise growth* in the resulting aggregate sample, which limits the number of additions and which may also lead to incorrect results if the noise grows "too much".

2. Besides that, no homomorphic operation other than addition has been defined yet, which is not sufficient to achieve the *full homomorphism.*

The procedure referred to as *bootstrapping* aims at resolving them both at the same time: while *refreshing the noise* to a certain, constant-on-average level, bootstrapping also inherently *evaluates a function*, referred to as the *bootstrapping function*, represented by a *Look-Up Table* (LUT). This makes TFHE fully homomorphic.

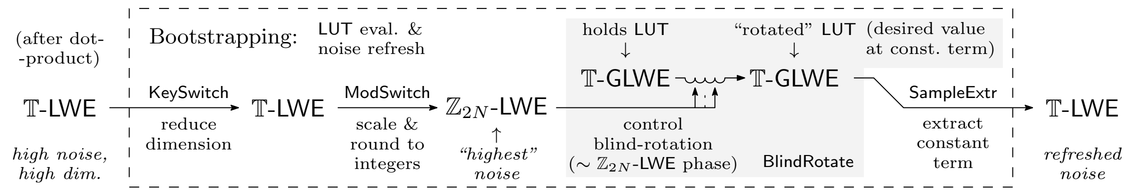


Figure 1.4: Outline of the internal structure of TFHE's bootstrapping. The aim of KeySwitch is to improve performance, whereas BlindRotate (inside the gray box) evaluates the LUT and refreshes the noise.

*Note* 1.3. An approach that clearly achieves the full homomorphism is presented in the original paper by Chillotti et al. [35], where authors define several logical gates, including $\neg$, $\vee$, and $\wedge$. We refer to this variant as the *binary* TFHE, however, in this chapter we rather focus on the variant of TFHE with a discrete multi-value cleartext space $\mathbb{Z}_{2^\pi}$, as outlined in Section 1.2.1. Binary TFHE might then be perceived as a special case.

As already outlined in Figure 1.2, TFHE gate is a combination of a homomorphic dot-product (cf. Theorem 1.1) and the bootstrapping procedure, which consists of four algorithms: KeySwitch, ModSwitch, BlindRotate and SampleExtract; find a more detailed illustration of bootstrapping in Figure 1.4. In the rest of this section, we discuss each algorithm in detail (except ModSwitch, which we cover together with BlindRotate) and we combine them into the Bootstrap algorithm.

According to the results of Bergerat et al. [13], it shows that in most cases, more efficient TFHE parameters can be found if dot-product is moved *before* key-switching (originally proposed by Bourse et al. [20]), as opposed to the original variant of TFHE [35]. I.e., in the new variant, key-switching appears at the beginning of bootstrapping, whereas in the original variant, key-switching is the last step. Authors of [13] also consider a variant that omits key-switching, but they do not find it more efficient either. Hence, we describe solely the new, re-ordered variant with key-switching in this chapter.

## 1.3.1 Key-Switching

The first step towards refreshing the noise, which happens in blind-rotate, is key-switching. Since blind-rotate is a demanding operation, the aim of key-switching is to reduce the dimension of the input sample. Attempts to omit key-switching were also tested by Bergerat et al. [13], however, achieving a poorer performance than the variant with key-switching.

The key-switching operation, denoted KeySwitch, effectively changes the encryption key of LWE sample $(b', \mathbf{a}')$ from LWE key $\mathbf{s}' \in \mathbb{B}^{n'}$ to LWE key $\mathbf{s} \in \mathbb{B}^n$. Besides the input LWE sample, KeySwitch requires a series of *key-switching keys*, while the $j$-th key is defined as

$$\mathsf{KS}_j := \mathsf{LWE}_{\mathbf{s}}(s'_j\, \mathbf{g}'), \quad j \in [1, n'], \tag{1.13}$$

where $\mathbf{g}'$ is a gadget vector given by decomposition base $B'$ and depth $d'$, and where each component of $s'_j\,\mathbf{g}'$ produces one LWE sample, independent from others. I.e., $\mathsf{KS}_j \in \mathbb{T}^{d',1+n}$ is interpreted as a matrix, where rows are actual LWE samples. We denote the set of key-switching keys from $\mathbf{s}'$ to $\mathbf{s}$ as $\mathsf{KS}_{\mathbf{s}'\to\mathbf{s}} := (\mathsf{KS}_j)_{j=1}^{n'}$. Note that key-switching keys consist of LWE samples and they can therefore be published as a part of evaluation keys.

Given LWE sample $(b', \mathbf{a}') \in \mathbb{T}^{1+n'}$ of $\mu$ under $\mathbf{s}'$, key-switching keys $\mathsf{KS}_{\mathbf{s}'\to\mathbf{s}}$, generated with gadget vector $\mathbf{g}'$, we define *key-switching* as

$$\mathsf{KeySwitch}_{\mathbf{s}'\to\mathbf{s}}(b',\mathbf{a}') = (b',\mathbf{0}) - \sum_{j=1}^{n'} \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j, \tag{1.14}$$

which returns an LWE sample of $\mu$ under $\mathbf{s}$. Note that in fact, KeySwitch homomorphically evaluates the phase function. In the following theorem, we evaluate the excess noise induced by KeySwitch.

**Theorem 1.3** (Correctness & Noise Growth of Key-Switching). *Given* LWE *sample* $\bar{\mathbf{c}}'$ *of* $\mu \in \mathbb{T}$ *under* LWE *key* $\mathbf{s}'$ *and key-switching keys* $\mathsf{KS}_{\mathbf{s}'\to\mathbf{s}}$*, encrypted with noise parameter* $\alpha'$*,* $\mathsf{KeySwitch}_{\mathbf{s}'\to\mathbf{s}}$ *returns* LWE *sample* $\bar{\mathbf{c}}$*, which holds excess noise* $e_{\mathsf{KS}}$*, given by* $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}} \rangle = \langle \bar{\mathbf{s}}', \bar{\mathbf{c}}' \rangle + e_{\mathsf{KS}}$*, for which it holds*

$$\mathsf{Var}[e_{\mathsf{KS}}] \approx \underbrace{n' V_{\mathbf{s}'} \varepsilon'^2}_{\textit{decomp. errors}} + \underbrace{n' d' V_{B'} \alpha'^2}_{\textit{amplif. } \mathsf{KS} \textit{ noise}}, \tag{1.15}$$

*where* $\varepsilon'^2$ *and* $V_{B'}$ *are as per* (1.8) *and* (1.9)*, respectively, with* $B'$ *and* $d'$*,* $V_{\mathbf{s}'}$ *is the variance of individual coefficients of the* LWE *key* $\mathbf{s}'$*, and other parameters are as per previous definitions. If* $e_{\mathsf{KS}}$ *and the noise of* $\bar{\mathbf{c}}'$ *are "sufficiently small", it holds* $\mu = \mathsf{msg}_{\mathbf{s}}(\bar{\mathbf{c}}) = \mathsf{msg}_{\mathbf{s}'}(\bar{\mathbf{c}}')$*, i.e.,* KeySwitch *indeed changes the key, without modifying the message.*

*Proof.* Find the proof in Appendix B.3. □

## 1.3.2 Blind-Rotate

The blind-rotate operation, denoted BlindRotate, is the cornerstone of bootstrapping since this is where the noise gets refreshed. It combines two ingredients: the *decryption (phase) function* $\varphi_{\mathbf{s}}(b,\mathbf{a}) = \mu + e$ (cf. (1.2)), and the *multiplicative homomorphism* of GGSW $\times$ GLWE samples (cf. Theorem 1.2).

Internally, blind-rotate evaluates a relation reminiscent of the phase function $\varphi_{\mathbf{s}}(b,\mathbf{a})$. However, as such, the phase function does not get rid of the noise – indeed, the error term remains present in the original amount; cf. (1.2). Therefore, a rounding step—as outlined in Section 1.2.1—must be included, too. Blind-rotate achieves rounding using a staircase LUT, i.e., a LUT that encodes a staircase function, where the "stairs" are responsible for rounding. Such a LUT is provided in a form of a (possibly encrypted) polynomial, which is referred to as the *test vector*, denoted by $tv(X)$, whose coefficients represent the LUT values.

Roughly speaking, we aim at multiplying $tv(X)$ by $X^{-M}$, where $M$ holds somehow the erroneous phase $\mu + e$. This "shifts" the coefficients of $tv(X)$ by $M$ positions towards lower powers of $X$ (we can think of discarding the coefficients that underflow for now). Then, evaluating $tv(X) \cdot X^{-M}$ at $X = 0$ (i.e., taking the constant term of the product) yields the originally $M$-th coefficient of $tv(X)$.

In the following paragraphs, we outline more concretely how a LUT can be encoded into a torus polynomial, which we further reduce modulo $X^N + 1$, so that it can be taken as a plaintext for a (possibly trivial) GLWE sample.

### Encoding a LUT into a Polynomial Modulo $X^N + 1$

The product of degree-$N$ polynomial $tv(X)$ and monomial $X^m$ (with $0 \le m < N$) holds the coefficients of $tv$ shifted by $m$ positions towards higher degrees. Reducing the product $tv(X) \cdot X^m$ modulo $X^N + 1$ brings the coefficients of powers higher than or equal to $N$ back to lower powers (namely by $N$ positions) while flipping their sign (e.g., $aX^{N+k}$ is reduced to $-aX^k$). Hence, multiplication of a polynomial by a monomial modulo $X^N+1$ yields a *negacyclic rotation.* These are the consequences for $\mathsf{LUT}$ evaluation in blind-rotate:

- multiplying $tv(X)$ by $X^{-m} \bmod X^N+1$ with $0 \leq m < N$ results in moving the $m$-th coefficient of $tv(X)$ to the constant position, which is then taken as a result of the $\mathsf{LUT}$;
- for $N \leq m < 2N$, the result equals to the $(m-N)$-th coefficient of $tv(X)$ with a flipped sign, due to the negacyclic rotation; and
- for greater $m$, it is worth noting that the period is $2N$.

Since we assume that $tv$ is a torus polynomial, we have $\mathsf{LUT}\colon \mathbb{Z}_{2N} \to \mathbb{T}$ and it holds

$$\mathsf{LUT}(N+m) = -\mathsf{LUT}(m), \quad m \in [0,N), \tag{1.16}$$

i.e., only the first $N$ values of a $\mathsf{LUT}$ need to be provided explicitly, while the other $N$ values are given implicitly by the negacyclic extension, and the rest is periodic with a period of $2N$. Encoded in a test vector $tv \in \mathbb{T}^{(N)}[X]$ as

$$tv^{(m)} = \mathsf{LUT}(m), \quad m \in [0,N), \tag{1.17}$$

the $\mathsf{LUT}$ is evaluated at $m \in \mathbb{Z}$ as

$$\begin{aligned} &\left(X^{-m} \cdot tv(X) \bmod X^N+1\right)^{(0)} = \\ &= (-1)^{\lfloor m/N \rfloor} \cdot tv(X)^{(m \bmod N)} = \\ &= \mathsf{LUT}(m \bmod 2N). \end{aligned} \tag{1.18}$$

We illustrate encoding of a $\mathsf{LUT}$ into a polynomial (test vector) $tv(X)$ in Figure 1.5. Next, we outline the overall idea of blind-rotate.

**Encoding the Stairs** Let us put forward explicitly the process of encoding the desired (negacyclic) bootstrapping function $\bar{f}\colon \mathbb{Z}_{2^\pi} \to \mathbb{Z}_{2^\pi}$ (which acts on cleartexts) into the respective $\mathsf{LUT}\colon \mathbb{Z}_{2N} \to \mathbb{T}$, represented by the test vector $tv(X)$, including the "stairs":

$$\mathsf{LUT}(k) = \bar{f}\left(\left\lfloor k \cdot \frac{2^\pi}{2N} \right\rceil\right), \quad k \in [0,2N). \tag{1.19}$$

We provide an illustration of such an encoding in Figure 1.6. We recall that only the $\mathsf{LUT}$ values for $k \in [0,N)$ are actually encoded into the test vector; cf. (1.17), (1.18) and Figure 1.5.

Recall that the "stairs" are supposed to be responsible for rounding, which in turn refreshes the noise. From 1.19, it follows that the width of such a stair is $1/2^\pi$. By $E_{\max}$, defined as

$$E_{\max} := \frac{1}{2^{\pi+1}}, \tag{1.20}$$

we denote the maximum of error magnitude that leads to the correct $\mathsf{LUT}$ evaluation; cf. Figure 1.6.

*Note* 1.4. Bootstrapping cannot be applied to only refreshing the noise, i.e., setting identity as the bootstrapping function, since identity is not negacyclic. A workaround must be made, with respect to a particular use case. Specifically, many existing implementations prepend an extra bit of padding, which they set to zero and do not use it. Note that such implementations need to somehow prevent possible overflows of homomorphic additions.

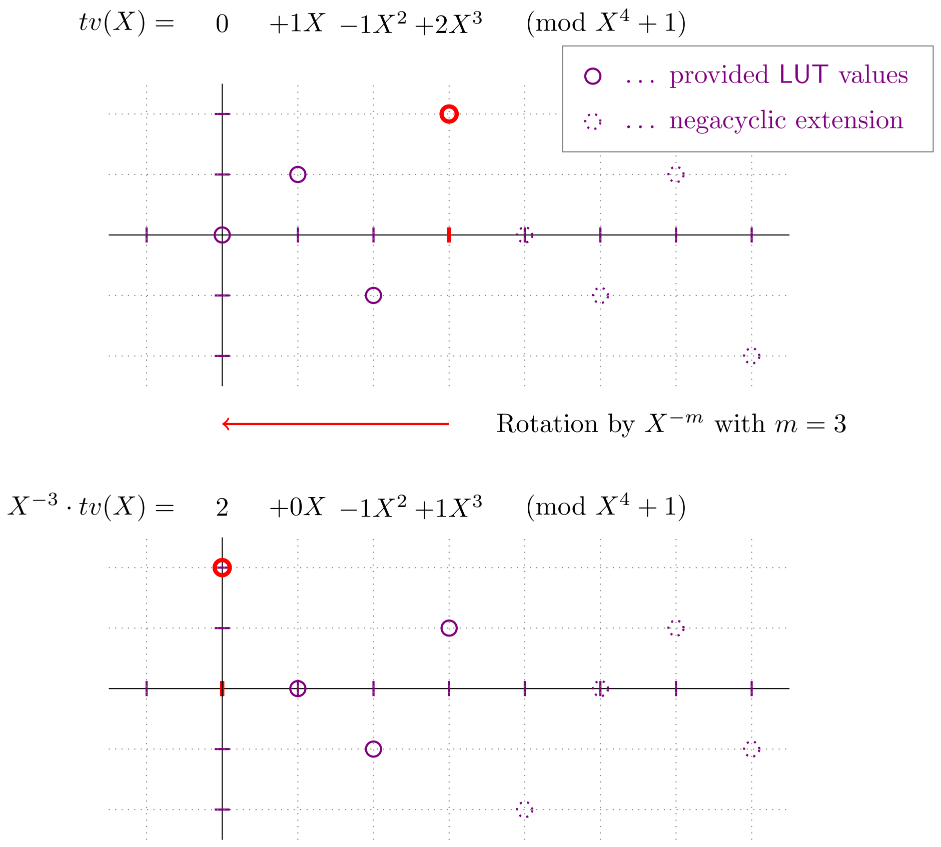
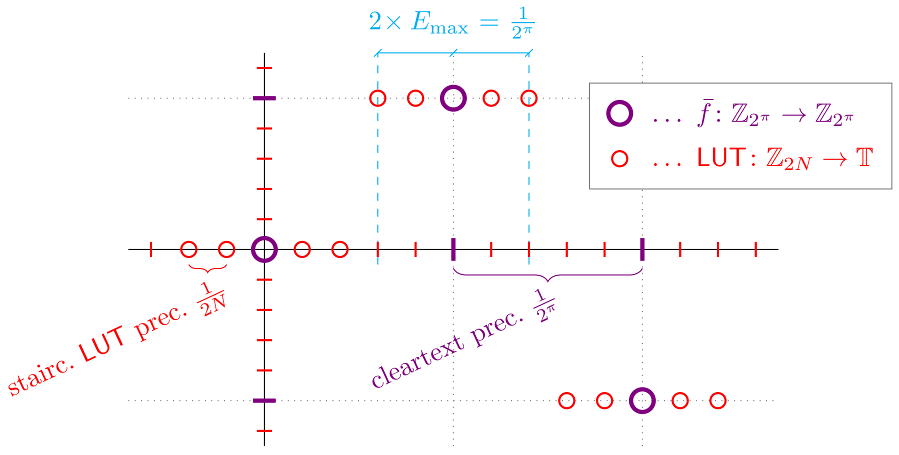

$tv(X) = \quad 0 \quad +1X \ -1X^2 + 2X^3 \quad (\text{mod } X^4 + 1)$
○ . . . provided LUT values
◌ . . . negacyclic extension
Rotation by $X^{-m}$ with $m = 3$
$X^{-3} \cdot tv(X) = \quad 2 \quad +0X \ -1X^2 + 1X^3 \quad (\text{mod } X^4 + 1)$

Figure 1.5: Illustration of encoding of a LUT into a polynomial mod $X^N + 1$ that is used in blind-rotate. We set $N = 4$ and we evaluate at $m = 3$, which means "rotation" by $X^{-3}$. The desired output value LUT(3) is emphasized in bold. N.b., in this illustration, we omit the "stairs" for simplicity.



$2\times E_{\max} = \frac{1}{2^\pi}$
... $\bar{f}\colon \mathbb{Z}_{2^\pi} \to \mathbb{Z}_{2^\pi}$
... $\mathsf{LUT}\colon \mathbb{Z}_{2N} \to \mathbb{T}$
stairc. LUT prec. $\frac{1}{2N}$
cleartext prec. $\frac{1}{2^\pi}$

Figure 1.6: Relation between a negacyclic bootstrapping function $\bar{f}$ and respective staircase LUT (illustrative). If the evaluated value does not leave its "stair", i.e., the input's error magnitude is lower than $E_{\max}$, LUT gets evaluated correctly.

### Idea of Blind-Rotate

First, let us get back to the phase function, which is responsible for decryption. Originally, with a LWE sample $(b, \mathbf{a})$ to be bootstrapped, $\varphi_{\mathbf{s}}(b, \mathbf{a})$ is (i) evaluated over the *torus*, and (ii) it is using *known bits* of the key $\mathbf{s}$.

For (i): based on previous observations, we first rescale & round the sample $(b, \mathbf{a}) \in \mathbb{T}^{1+n}$ to the $\mathbb{Z}_{2N}$ domain, which preserves periodicity. Therefore, we calculate the scaled and rounded value of the phase function as

$$\tilde{m} = \tilde{b} + \langle \mathbf{s}, \tilde{\mathbf{a}} \rangle, \tag{1.21}$$

where $\tilde{b} = \lfloor 2Nb \rceil$ and $\tilde{a}_i = \lfloor 2Na_i \rceil$, which is also referred to as *modulus switching*. As outlined previously, we aim at performing the evaluation of $\tilde{m}$ as per (1.21) in powers of $X$; cf. (1.18).

For (ii): the secret key $\mathbf{s}$ is clearly not known to the evaluator (the cloud). Instead, bits $s_i$ of the key are provided in a form of GGSW samples $\mathsf{BK}_i$, encrypted with GLWE key $\mathbf{z}$, and referred to as the *bootstrapping keys*, denoted by $\mathsf{BK}_{\mathbf{s}\to\mathbf{z}} = (\mathsf{BK}_i)_{i=1}^n$.

From (i) and (ii), it follows that given the sample $(b, \mathbf{a})$ and the encrypted bits of the key $\mathbf{s}$ (bootstrapping keys), we can apply homomorphic operations to obtain the (encrypted) monomial $X^{\tilde{m}}$ modulo $X^N + 1$, which we employ for LUT evaluation as per (1.18). I.e., the test vector gets *blindly rotated.*

In the following paragraphs, we provide a full technical overview of modulus-switching and blind-rotate, respectively.

### Modulus-Switching

As outlined, blind-rotate is preceded by a technical step, referred to as modulus-switching and denoted by ModSwitch, which is parameterized by $N \in \mathbb{N}$. $\mathsf{ModSwitch}_N$, which inputs LWE sample $(b, \mathbf{a}) \in \mathbb{T}^{1+n}$ under LWE key $\mathbf{s}$ and outputs LWE sample $(\tilde{b}, \tilde{\mathbf{a}}) \in \mathbb{Z}_{2N}^{1+n}$ under the same key, is defined as

$$\mathsf{ModSwitch}_N(b, \mathbf{a}) = \left(\lfloor 2Nb \rceil, \lfloor 2Na_i \rceil_{i=1}^n\right) =: (\tilde{b}, \tilde{\mathbf{a}}), \tag{1.22}$$

where multiplications of type $2N \cdot a_i$ are performed in $\mathbb{R}$ (using any unit interval for $\mathbb{T}$), then rounding brings result back to $\mathbb{Z}$, from where we easily obtain $\mathbb{Z}_{2N}$.

Due to rounding, additional noise is induced by $\mathsf{ModSwitch}$; we evaluate it in the following lemma.

**Lemma 1.4** (Noise Growth of Modulus-Switching). $\mathsf{ModSwitch}_N$ *induces an excess noise, given by* $1/2N \cdot \langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}}) \rangle = \langle \bar{\mathbf{s}}, (b, \mathbf{a}) \rangle + e_{\mathsf{MS}}$*, for which it holds*

$$\mathsf{Var}[e_{\mathsf{MS}}] = \frac{1 + n/2}{48N^2}, \tag{1.23}$$

*where* $n$ *is the* $\mathsf{LWE}$ *dimension.*

*Proof.* We write

$$\begin{aligned} e_{\mathsf{MS}} &= \langle (1, \mathbf{s}), (\tilde{b}/2N - b, \tilde{\mathbf{a}}/2N - \mathbf{a}) \rangle = \\ &= \underbrace{\tilde{b}/2N - b}_{\in(-1/4N, 1/4N]} + \sum s_i \cdot \underbrace{(\tilde{a}_i/2N - a_i)}_{\in(-1/4N, 1/4N]}, \end{aligned} \tag{1.24}$$

where each underbraced term is assumed to have a uniform distribution on $(-1/4N, 1/4N]$, i.e., the variance of $1/48N^2$. For $s_i$, we have $\mathrm{E}[s_i^2] = 1/2$. For independent variables with $\mathrm{E}[Y] = 0$, it holds $\mathsf{Var}[X \cdot Y] = \mathrm{E}[X^2] \cdot \mathsf{Var}[Y]$, which is this case for $X = s_i$ and $Y = \tilde{a}_i/2N - a_i$. The result follows. □

## Description of Blind-Rotate

A description of blind-rotate is given in Algorithm 1. In line 4, if $\mathsf{BK}_i$ encrypts $s_i = 0$, the line evaluates to (encrypted) $\mathsf{ACC} = X^{0 \cdot \tilde{a}_i} \cdot \mathsf{ACC}$, if $\mathsf{BK}_i$ encrypts $s_i = 1$, we obtain (encrypted) $X^{1 \cdot \tilde{a}_i} \cdot \mathsf{ACC}$. I.e., after blind-rotation, we obtain an encryption of $X^{\tilde{m}} \cdot tv$, where $\tilde{m} = \langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}}) \rangle$. Line 4 also mandates $s_i \in \mathbb{B}$, as outlined in Note 1.2, although generalization attempts exist [76].

**Algorithm 1** BlindRotate

**Input:** $\mathsf{LWE}$ sample $(b, \mathbf{a})$ of $\mu \in \mathbb{T}$ under $\mathsf{LWE}$ key $\mathbf{s} \in \mathbb{B}^n$, modulus-switched to $(\tilde{b}, \tilde{\mathbf{a}}) \in \mathbb{Z}_{2N}^{1+n}$,
**Input:** (usually trivial) $\mathsf{GLWE}$ sample $\bar{\mathbf{t}} \in \mathbb{T}^{(N)}[X]^{1+k}$ of $tv \in \mathbb{T}^{(N)}[X]$ (aka. *test vector*) under $\mathsf{GLWE}$ key $\mathbf{z} \in \mathbb{Z}^{(N)}[X]^k$,
**Input:** for $i \in [1, n]$, $\mathsf{GGSW}$ samples of $s_i$ under $\mathbf{z}$, referred to as *bootstrapping keys*, denoted $\mathsf{BK}_{\mathbf{s} \to \mathbf{z}} := (\mathsf{BK}_i)_{i=1}^n$.
**Output:** $\mathsf{GLWE}$ sample of $X^{\tilde{m}} \cdot tv$ under $\mathbf{z}$, where $\tilde{m} = \langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}}) \rangle \approx 2N\mu$.

```
1: $\mathsf{ACC} \leftarrow X^{\tilde{b}} \cdot \bar{\mathbf{t}}$                ▷ aka. accumulator
2: for $i \in [1, n]$ do
3:     $\mathsf{ACC} \leftarrow \mathsf{ACC} + \mathsf{BK}_i \boxdot (X^{\tilde{a}_i} \cdot \mathsf{ACC} - \mathsf{ACC})$
4: end for
5: return $\mathsf{ACC}$
```

*Remark* 1.5. During blind-rotate, the "old" noise is refreshed with a fresh noise, which comes from the bootstrapping keys and from the (possibly encrypted) test vector – the fresh noise *does not* depend on the noise of the input sample. Nevertheless, the "old" noise affects what value from the test vector is selected (gets rotated to), i.e., at which point the (staircase) $\mathsf{LUT}$ is evaluated; cf. Figure 1.6. We discuss two types of decryption errors later in Section 4.5.

In the following theorem, we evaluate the noise of the output of $\mathsf{BlindRotate}$ – i.e., the refreshed noise, which we denote $V_0$.

**Theorem 1.5** (Correctness & Noise Growth of Blind-Rotate)**.** *Given inputs of Algorithm 1, where bootstrapping keys are encrypted with noise parameter $\alpha$ and test vector is (possibly) encrypted with noise parameter $\alpha_t$ (i.e., $\alpha_t = 0$ or $\alpha$),* $\mathsf{BlindRotate}$ *returns the last-step* $\mathsf{ACC}$ *with noise variance given by*

$$\begin{aligned}\mathsf{Var}[\langle \bar{\mathbf{z}}, \mathsf{ACC}\rangle] \approx \alpha_t^2 + ndNV_B\alpha^2(1+k) + \\ + n\varepsilon^2(1+kNV_{\mathbf{z}}) =: V_0,\end{aligned} \tag{1.25}$$

*which we denote by $V_0$, other parameters are as per previous theorems and definitions. If the noise of $\langle \bar{\mathbf{z}}, \mathsf{ACC}\rangle$ is "sufficiently small", it holds $\mathsf{msg}_{\mathbf{z}}(\mathsf{ACC}) = X^{\tilde{m}} \cdot tv$, where $\tilde{m} = \langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}})\rangle \approx 2N\mu$, i.e.,* $\mathsf{BlindRotate}$ *indeed "rotates" the test vector by the approximate phase of $(b, \mathbf{a})$, scaled to $\mathbb{Z}_{2N}$.*

*Proof.* Find the proof in Appendix B.2. □

### 1.3.3 Sample-Extract

By far, $\mathsf{BlindRotate}$ outputs a $\mathsf{GLWE}$ sample of a polynomial (blindly-rotated test vector), which holds the desired value at its constant term and which is encrypted with a $\mathsf{GLWE}$ key. The goal of $\mathsf{SampleExtract}$ is to literally extract a partial $\mathsf{LWE}$ sample, which encrypts the constant term, out of the $\mathsf{GLWE}$ sample, which we denote by $(b, \mathbf{a}) \in \mathbb{T}^{(N)}[X]^{1+k}$ (the last-step $\mathsf{ACC}$ in Algorithm 1). Note that a similar thing happens with the key: the new $\mathsf{LWE}$ key is also an "extract" of the original polynomial $\mathsf{GLWE}$ key $\mathbf{z} \in \mathbb{Z}^{(N)}[X]^k$. Writing down the constant term of $\langle \bar{\mathbf{z}}, (b, \mathbf{a})\rangle$, which is a torus polynomial, we obtain

$$\begin{aligned}\big\langle \bar{\mathbf{z}}, (b, \mathbf{a})\big\rangle^{(0)} &= b^{(0)} + \sum_{i=1}^{k} \big(z_i(X) \cdot a_i(X)\big)^{(0)} = \\ &= b^{(0)} + \sum_{i=1}^{k} \Big\langle \underbrace{\big(z_i^{(0)}, -z_i^{(N-1)}, \dots, -z_i^{(1)}\big)}_{i\text{-th partial extr. LWE key } \mathbf{z}_i^*}, \\ &\qquad \underbrace{\big(a_i^{(0)}, a_i^{(1)}, \dots, a_i^{(N-1)}\big)}_{i\text{-th partial extr. LWE sample } \mathbf{a}_i^*} \Big\rangle,\end{aligned} \tag{1.26}$$

where we denote the $i$-th partial extracted $\mathsf{LWE}$ key and sample by $\mathbf{z}_i^* \in \mathbb{Z}^N$ and $\mathbf{a}_i^* \in \mathbb{T}^N$, respectively. We obtain the full extracted $\mathsf{LWE}$ key and sample as their concatenations, denoted respectively as $\mathbf{z}^* \in \mathbb{Z}^{kN}$ and $(b^{(0)}, \mathbf{a}^*) \in \mathbb{T}^{1+kN}$, where $b^{(0)}$ is prepended. Note that $\mathbf{a}^*$ is a simple serialization of polynomial coefficients of $\mathbf{a}$, whereas for $\mathbf{z}$, a rearranging is needed, together with negative signs. Finally, we have

$$\big\langle \bar{\mathbf{z}}, (b, \mathbf{a})\big\rangle^{(0)} = \big\langle \bar{\mathbf{z}}^*, (b^{(0)}, \mathbf{a}^*)\big\rangle, \tag{1.27}$$

while noise preserves. Note that the extracted key $\mathbf{z}^*$ plays the role of the $\mathsf{LWE}$ key $\mathbf{s}'$ that is supposed to be encrypted in key-switching keys – we compose the four algorithms and we provide further details in the next section.

### 1.3.4 TFHE Bootstrapping

Putting the four algorithms together, we obtain the $\mathsf{TFHE}$ *(Programmable) Bootstrapping* algorithm; find it as Algorithm 2, previously outlined in Figure 1.4. It is worth noting that $\mathsf{BlindRotate}$

(on line 3 of that algorithm) inputs a negative sample: this is due to the LUT encoding that we use; cf. (1.17) and (1.18), where a negative sign at $m$ is expected, although by Theorem 4.2, a positive sign appears in the power of $X$.

**Algorithm 2** Bootstrap

**Input:** LWE sample $(b^*, \mathbf{a}^*) \in \mathbb{T}^{1+kN}$ of $\mu = {}^m/_{2^\pi}$, $m \in \mathbb{Z}_{2^\pi}$, under LWE key $\mathbf{z}^* \in \mathbb{Z}^{kN}$, extracted from GLWE key $\mathbf{z} \in \mathbb{Z}^{(N)}[X]^k$,
**Input:** (possibly trivial) GLWE sample $\bar{\mathbf{t}} \in \mathbb{T}^{(N)}[X]^{1+k}$ of test vector $tv \in \mathbb{T}^{(N)}[X]$ under the key $\mathbf{z}$, where $tv$ encodes negacyclic bootstrapping function $\bar{f}\colon \mathbb{Z}_{2^\pi} \to \mathbb{Z}_{2^\pi}$ as per (1.17) and (1.19),
**Input:** key-switching keys $\mathsf{KS}_{\mathbf{z}^* \to \mathbf{s}}$ and bootstrapping keys $\mathsf{BK}_{\mathbf{s} \to \mathbf{z}}$.
**Output:** LWE sample of ${}^{\bar{f}(m)}/_{2^\pi}$ under key $\mathbf{z}^*$.

```
1: $(b, \mathbf{a}) \leftarrow \mathsf{KeySwitch}\big((b^*, \mathbf{a}^*), \mathsf{KS}_{\mathbf{z}^* \to \mathbf{s}}\big)$
2: $(\tilde{b}, \tilde{\mathbf{a}}) \leftarrow \mathsf{ModSwitch}_N(b, \mathbf{a})$
3: $(s, \mathbf{r}) \leftarrow \mathsf{BlindRotate}\big((-\tilde{b}, -\tilde{\mathbf{a}}), \bar{\mathbf{t}}, \mathsf{BK}_{\mathbf{s} \to \mathbf{z}}\big)$
4: return $(b', \mathbf{a}') \leftarrow \mathsf{SampleExtract}\big((s, \mathbf{r})\big)$
```

We provide a summary of parameters in Table 1.1. For an exhaustive technical overview of blind-rotate, preceded by modulus-switching and followed by sample-extract, we refer to Appendix B, Figure 9.

Table 1.1: Summary of parameters' notation. Parameters $\varepsilon^2$, $\varepsilon'^2$ and $V_B$, $V_{B'}$ are derived from respective decomposition parameters; cf. (1.8) and (1.9).

| | | | |
|---|---|---|---|
| LWE secret key | $\mathbf{s}$ | GLWE secret key | $\mathbf{z}$ |
| LWE dimension | $n$ | GLWE dimension | $k$ |
| | | GLWE polyn. degree | $N$ |
| LWE noise std-dev | $\alpha'$ | GLWE noise std-dev | $\alpha$ |
| KS decomp. base | $B'$ | BK decomp. base | $B$ |
| KS decomp. depth | $d'$ | BK decomp. depth | $d$ |

Recall that the noise gets refreshed in BlindRotate (cf. Remark 1.5) and it does not change in SampleExtract, i.e., the variance of a freshly bootstrapped sample is given by $V_0$ as per (1.25). N.b., at this point, we do not guarantee the correctness of the output – details will be given in Section 1.4, where we identify bounds that need to be satisfied so that the bootstrapping function is evaluated at the correct point.

## 1.4 Correctness of LUT Evaluation

In this section, we combine the noise growth estimates from the previous section and we derive a condition for the correct evaluation of the bootstrapping function. As outlined, the noise propagates throughout various operations, let us provide an overview first.

### Overview of Noise Propagation

The noise, which is present in every encrypted sample, evolves during the evaluation of a TFHE gate, which comprises (i) homomorphic *dot-product* and (ii) *bootstrapping* (with its four sub-operations). Let us comment on either operation:

**Dot-product:** Provided that the noise of involved samples is independent, the error variance of a weighted sum is additive with weights squared (cf. (1.4) in Theorem 1.1).

**Bootstrapping:** If the noise of the sample-to-be-bootstrapped is smaller than a certain bound, the blind-rotate step of bootstrapping evaluates the bootstrapping function correctly: i.e., the error of $\tilde{m}$ (as per Theorem 4.2 and Algorithm 1) is smaller than the bound $E_{\max}$; cf. Figure 1.6. The resulting sample then carries—on average—a fixed amount of noise (independent of the original sample), which solely depends on the TFHE parameters (cf. (1.25) in Theorem 4.2).

In Figure 1.7, we illustrate the error propagation throughout a TFHE gate, where $e_0^{(i)}$ denotes actual noise of the $i$-th, freshly bootstrapped sample. Note that the overall maximum of relative average noise is achieved within bootstrapping when the modulus-switched sample $(\tilde{b}, \tilde{\mathbf{a}})$ enters blind-rotate, which refreshes the noise; cf. Remark 1.5.

$$\underbrace{e_0^{(1)}, e_0^{(2)}, \dots}_{\substack{\text{freshly bootstrap-}\\\text{ped sample(s)}}} \xrightarrow[\text{dot-product}]{\text{homomorphic}} \underbrace{\sum_i e_0^{(i)}}_{\substack{\textsf{pre-Bootstrap}\\(e_{\textsf{DP}})}} \overbrace{\xrightarrow[\textsf{ModSwitch}]{\textsf{KeySwitch,}} \underbrace{e_{\textsf{DP}} + e_{\textsf{KS}} + e_{\textsf{MS}}}_{\substack{\textsf{pre-BlindRotate}\ (e_{\max}),\\ |e_{\max}| \overset{!}{<} E_{\max}}} \xrightarrow[\textsf{SampleExtract}]{\textsf{BlindRotate,}}}^{\textsf{Bootstrap}} \underbrace{e_0'}_{\substack{\text{bootstrapped}\\\text{sample}}}$$

Figure 1.7: Noise propagation from a bunch of freshly bootstrapped samples throughout a TFHE gate towards a new, freshly bootstrapped sample. If $|e_{\max}| < E_{\max}$, the bootstrapping function is evaluated correctly.

We denote the maximum error and its variance by $e_{\max}$ and $V_{\max}$, respectively, and we have

$$V_{\max} \approx \nu_{\max}^2 \cdot V_0 + V_{\textsf{KS}} + V_{\textsf{MS}}, \tag{1.28}$$

where $\nu_{\max}^2$ is the maximum of sums of squares of integer weights of dot-products (cf. (1.5)) across the entire computation, $V_0$, $V_{\textsf{KS}}$ and $V_{\textsf{MS}}$ are respectively the variance of a freshly bootstrapped sample (cf. (1.25), combined using (1.4)), the variance of the excess noise of key-switching (cf. (1.15)) and that of modulus-switching (cf. (1.23)). Note that $e_{\max}$ is the relative, torus-scaled error of $\tilde{m} \in \mathbb{Z}_{2N}$ that enters blind-rotate; cf. Algorithm 2. The magnitude of this error is decisive for the correctness of the bootstrapping function evaluation as per Figure 1.6.

*Note* 1.6. Let us outline an intuition that justifies the design where key-switching is moved to the beginning of bootstrapping (proposed in [20], experimentally shown to be more efficient in [13], presented in this chapter), as opposed to the original TFHE design [35], where key-switching is the last step of bootstrapping. The variance of the maximum error of the original variant writes

$$V_{\max} \approx \underbrace{(V_{\textsf{BR}} + V_{\textsf{KS}})}_{V_0^{(\text{or.})}} \cdot \nu^2 + V_{\textsf{MS}}, \tag{1.29}$$

whereas in the re-ordered variant, we have

$$V_{\max} \approx \underbrace{V_{\textsf{BR}}}_{V_0^{(\text{re.})}} \cdot \nu^2 + V_{\textsf{KS}} + V_{\textsf{MS}}, \tag{1.30}$$

where $V_{\textsf{BR}}$ is the variance of the output of BlindRotate. We may notice that the re-ordered variant is expected to achieve a lower noise growth, in particular for applications with greater $\nu^2$. Also, note that the re-ordered variant in fact only swaps key-switching and dot-product; let us outline both, starting after sample-extract:

$$\begin{aligned} \text{orig.:}\ & (\text{SE}) \to \mathsf{KS} \xrightarrow{V_0^{(\text{or.})}} \mathsf{DP} \xrightarrow{\text{to bs.}} \mathsf{MS} \to \dots \\ \text{reord.:}\ & (\text{SE}) \xrightarrow{V_0^{(\text{re.})}} \mathsf{DP} \xrightarrow{\text{to bs.}} \mathsf{KS} \to \mathsf{MS} \to \dots \end{aligned}$$

### 1.4.1 Correct Evaluation of the Bootstrapping Function

Let us define the quantity $\kappa$, which aims at quantifying the probability of correct evaluation of the bootstrapping function (i.e., $|e_{\max}| < E_{\max}$), as

$$\kappa := \frac{E_{\max}}{\sqrt{V_{\max}}}. \tag{1.31}$$

The aim of $\kappa$ is to tell how many times the standard deviation of the maximum error, denoted $\sigma_{\max} = \sqrt{V_{\max}}$, fits into the target interval of the size of $2E_{\max}$ around the expected value. The probability that a normally distributed random variable falls within the interval of $\kappa$ times its standard deviation can be looked-up from *standard normal tables* (aka. the *Z-tables*). Note that by the Central Limit Theorem, we assume a normal distribution for the value of $\tilde{m}$. E.g., for $\kappa = 3$, we have $\Pr[\cdot] \approx 99.73\% \approx 1/370$ (aka. rule of $3\sigma$), however, we recommend higher values of $\kappa$ (e.g., Bergerat et al. [13] provide their parameters with $\kappa = 4$, which gives error rate $\approx 1/15\,787$). N.b., also the size of the evaluated circuit as well as possible real-world consequences of an incorrect evaluation shall be taken into account.

From (4.15) and (1.20), we obtain the *fundamental condition* on the variance of the maximum error as

$$\boxed{V_{\max} \leq \frac{1}{\kappa^2 \cdot 2^{2\pi+2}},} \tag{1.32}$$

where $V_{\max}$ can be further broken down by (1.28) and other previous equalities. If the fundamental condition is satisfied, the (erroneous) value of $\tilde{m}$ does not leave its “stair” with high probability (related to $\kappa$), and the bootstrapping function $\bar{f}$ is evaluated correctly; cf. Figure 1.6.

#### Types of Decryption Errors

Correct blind-rotate does not itself guarantee the correctness of the result after decryption – indeed, there is a non-zero probability that the freshly bootstrapped sample (or a dot-product of them) decrypts incorrectly due to the intrinsic LWE noise. Therefore, we define two types of decryption errors that may occur after a dot-product followed by bootstrapping: one due to LWE noise (as just outlined), and one due to incorrect blind-rotate.

**Fresh Bootstrap Error** ($\mathsf{Err}_1$) First, let us assume that blind-rotate rotates the test vector correctly (i.e., $|e_{\max}| < E_{\max}$) and we denote the output LWE sample of bootstrapping as $\bar{\mathbf{c}}'$. Then, if the distance of $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}' \rangle$ from the expected value is greater than $E_{\max}$, we refer to this kind of error as the *type-1 error*, denoted $\mathsf{Err}_1$.

The probability of $\mathsf{Err}_1$ relates to the noise of a correctly blind-rotated, freshly bootstrapped sample, which can be estimated from $V_0$; see (1.25).

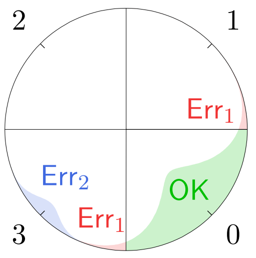


Figure 1.8: Illustration of type-1 and type-2 errors: LUT evaluates correctly and incorrectly to 0 and 3, respectively.

**Blind-Rotate Error** ($\mathsf{Err}_2$) Second, we consider the result of a TFHE gate, i.e., we take a dot-product of a bunch of independent, freshly bootstrapped samples, with $\nu^2 \leq \nu^2_{\max}$, and we bootstrap it. Then, if blind-rotate rotates the test vector incorrectly (i.e., $|e_{\max}| > E_{\max}$), we refer to this kind of error as the *type-2 error*, denoted $\mathsf{Err}_2$. Note that a combination of both error types may occur[2].

The probability of $\mathsf{Err}_2$ relates to the error of modulus-switched sample $(\tilde{b}, \tilde{\mathbf{a}})$ that appears inside bootstrapping, and it can be estimated from $V_{\max}$; see (1.28). We outline both error types in Figure 1.8.

**Corollary 1.6.** *For the probabilities of type-1 and type-2 errors, by* (1.28) *we have*

$$\Pr[\mathsf{Err}_1] < \Pr[\mathsf{Err}_2]. \tag{1.33}$$

For common choices of parameters, $\Pr[\mathsf{Err}_1]$ can be neglected. I.e., we may use the fundamental condition (1.32) to estimate the probability of incorrect evaluation of a single TFHE gate.

## 1.4.2 Parameter Constraints

Previously, we justified the use of the fundamental condition (1.32) to make error probability estimates. Next, we identify four high-level parameters that aim at characterizing the properties of an instance of TFHE. Finally, we combine the fundamental condition to obtain a relation between the four high-level parameters and actual TFHE parameters (like LWE dimension or noise amplitude).

### Characteristic Parameters

Given a usage scenario, an instance of TFHE can be characterized by the following four (input) parameters:

1. *cleartext space bit-precision*, denoted by $\pi$ (cf. Section 1.2.1);
2. *quadratic weights*, denoted by $\nu^2$ (cf. Theorem 1.1, we take maximum of computation);
3. *bit-security level*, denoted by $\lambda$; and
4. *bootstrapping correctness*, denoted by $\eta := \Pr[\mathsf{Err}_2]$ (cf. Corollary 1.6).

Let us provide more (practical) comments on each of the input parameters.

[2]The result may combine both kinds of errors and decrypt correctly at the same time – in such a case, we consider that both error types occur simultaneously.

**Cleartext Bit-Precision:** $\pi$ Regarding the choice of an appropriate cleartext bit-precision, we point out two things: First, it shows that the complexity of the TFHE bootstrapping grows roughly exponentially with the cleartext bit-precision – reasonable bootstrapping times can be achieved for up to about $\pi = 8$ bits, then, splitting the cleartext into multiple chunks comes into play. Second, bootstrapping is capable of evaluating a custom bootstrapping function $\bar{f}\colon \mathbb{Z}_{2^\pi} \to \mathbb{Z}_{2^\pi}$, however, such function must be negacyclic; cf. (1.16) and (1.19), unless a workaround is adopted as per Note 1.4. Both limitations must be carefully considered before choosing the right cleartext space bit-precision $\pi$: it might make sense to decrease the cleartext space size at the expense of additional, but cheaper bootstrapping.

**Quadratic Weights:** $\nu^2$ As outlined in (1.28), $\nu^2_{\max} := \max_g\{\nu_g^2\}$ is defined as the maximum of sums of squares of integer weights of dot-products across the whole circuit that comprises TFHE gates $g \in G$ (with $\nu^2$ defined in (1.5)). Note that $\log(\nu_g)$ expresses the number of bits of the standard deviation of the excess noise introduced by the dot-product in gate $g$.

**Security Level:** $\lambda$ We discuss LWE/GLWE security in Section 4.4.1. Recall that the higher $\lambda$ is requested, the higher LWE dimension and/or the lower noise must be present, and security also depends on the distribution of keys.

**Bootstrapping Correctness:** $\eta$ Introduced in Section 1.4.1, the parameter $\kappa$ characterizes the probability of erroneous blind-rotate. In Corollary 1.6, we use this probability to estimate the overall probability of correct evaluation of a TFHE gate. Hence, to quantify the probability of correct evaluation of a single TFHE gate, we take $\eta$, and by standard normal tables, we deduce the value of $\kappa$, which we use for the rest of the analysis. Recall that $\kappa$ relates to the correctness of a single TFHE gate, i.e., for a circuit that consists of multiple TFHE gates, the value of $\eta$ needs to be modified accordingly.

**Parameter Relations**

To make the fundamental condition (1.32) hold, we may combine (1.28) with (1.25), (1.15) and (1.23), and mandate

$$\begin{aligned} V_{\max} &\approx \nu^2 \cdot V_0 + V_{\mathsf{KS}} + V_{\mathsf{MS}} \approx \\ &\approx \nu^2 \cdot \big(\alpha_t^2 + ndNV_B\alpha^2(1+k) + \\ &\quad + n\varepsilon^2(1 + kNV_{\mathbf{z}})\big) + kNV_{\mathbf{z}}\varepsilon'^2 + \\ &\quad + kNd'V_{B'}\alpha'^2 + \frac{1 + {}^{n}\!/_{2}}{48N^2} \overset{!}{\leq} \\ &\overset{!}{\leq} \frac{1}{\kappa^2 \cdot 2^{2\pi+2}}, \end{aligned} \tag{1.34}$$

where the baseline parameters are summarized in Table 1.1, $\varepsilon^2$ and $V_B$ are defined in (1.8) and (1.9), respectively, and $V_{\mathbf{z}}$ stands for the variance of coefficients of the internal GLWE secret key $\mathbf{z}$.

In terms of the *security* $\leftrightarrow$ *correctness* $\leftrightarrow$ *performance* triangle given in Section 1.2.1, this inequality only provides a guarantee of *correctness*, which is given by $\eta$ (translated into $\kappa$), for prescribed plaintext precision $\pi$ and quadratic weights $\nu^2$. In particular, *security* is *not* addressed and it must be resolved separately, e.g., using `lattice-estimator` [8]. The combination of constraints on TFHE parameters makes it a complex task to generate a set of parameters, which is further supposed to achieve a good *performance*. Authors of [13] claim to have implemented a generator of efficient TFHE parameters and they provide a comprehensive list of TFHE parameters for many input setups. However, at the time of writing, the tool is not publicly available yet.

## 1.5 Implementation Remarks

In this section, we briefly comment on various implementation aspects of TFHE, namely

- (negacyclic) polynomial multiplication;
- additional errors (noise) that stem from particular implementation choices;
- estimated complexity & key sizes; and
- existing implementations of TFHE, including recent trends and advances.

### 1.5.1 Negacyclic Polynomial Multiplication

For performance reasons, modular polynomial multiplication in TFHE—which appears, e.g., in GLWE encryption (1.1) or in external product (1.11)—is implemented using *Fast Fourier Transform* (FFT). Recall that polynomials mod $X^N + 1$ rotate negacyclically when multiplied by $X^k$, unlike polynomials mod $X^N - 1$, which rotate cyclically. Note that in such a case, polynomial multiplication is equivalent to the standard cyclic convolution, which can be calculated using Fast Fourier Transform (FFT). However, for polynomials mod $X^N + 1$, other tricks need to be put into place; find a description of negacyclic polynomial multiplication, e.g., in [81].

### 1.5.2 Implementation Noise

Notably, FFT is the major source of additional errors (noise) that are *not* captured by the theoretical noise analysis given in Section 1.4. The magnitude of FFT errors depends particularly on the number representation that is used by selected FFT implementation; find a study on FFT errors in [81]. Although for commonly used parameters and FFT implementations, FFT errors are negligible compared to (G)LWE noises, they shall be kept in mind, in particular in non-standard constructions or new designs.

In addition, compared to the theoretical results of Section 1.4, torus elements are represented using a finite representation (as outlined in Section 1.2.1; e.g., with 64-bit integers), which also changes the errors slightly. However, as long as the torus precision (e.g., $2^{-64}$) is much smaller than the standard deviation of the (G)LWE noise—which is usually the case for common parameter choices—this contribution can be neglected.

### 1.5.3 Key Sizes & Bootstrapping Complexity

Below, we provide the sizes of key-switching and bootstrapping keys, as represented in a TFHE implementation with a $\tau$-bit representation of torus elements. The complexity of TFHE bootstrapping, which is the dominant operation of TFHE, is roughly proportional to the key sizes.

**Size of Key-Switching Keys** Using notation of Table 1.1, key-switching keys consist of $N$ sub-keys $\mathsf{KS}_j \in \mathbb{T}^{d',1+n}$; altogether we have

$$\left|(\mathsf{KS}_j)_{j=1}^{N}\right| = Nd'(1+n)\tau \text{ [bits]}. \tag{1.35}$$

Note that a common method to store/transmit key-switching keys, which can also be applied to bootstrapping keys, is to keep just a seed for a pseudo-random number generator (PRNG), instead of all of the randomness. I.e., for each LWE sample, just the value of $b$ is kept and the values of $\mathbf{a}$ can be re-generated from the seed.

**Size of Bootstrapping Keys**
Bootstrapping keys consist of $n$ GGSW samples $\mathsf{BK}_i \in \mathbb{T}^{(N)}[X]^{(1+k)d,1+k}$; altogether we have

$$\left|(\mathsf{BK}_i)_{i=1}^{n}\right| = n(1+k)^2 dN\tau \text{ [bits]}. \tag{1.36}$$

**Rough Estimate of Bootstrapping Complexity** Key-switching is dominated by $Nd'(1+n)$ torus multiplications, followed by $1+n$ summations of $Nd'$ elements, which makes key-switching $O\big(Nd'(1+n)\tau\big)$. Blind-rotate is dominated by $n(1+k)^2d$ degree-$N$ polynomial multiplications, followed by a similar number of additions/subtractions, which makes blind-rotate $O\big(n(1+k)^2dN\tau\big)$. Note that the entire calculation of BlindRotate (cf. Algorithm 1) can be performed in the Fourier domain – thanks to its linearity and pre-computed bootstrapping keys, i.e., the $O(\tau N \log N)$ term of FFT can be neglected. In this rough estimate, we neglect modulus-switching and sample-extract. Also, we do not distinguish the bit-length of $\tau$ for LWE and GLWE, as some implementations do [106].

### 1.5.4 Existing TFHE Implementations

The original (experimental) TFHE library [124] is not developed anymore, instead, there are other, more or less active implementations. Here we list selected implementations of TFHE:

`TFHE-rs` **[132]:** written in Rust, `TFHE-rs` implements latest findings by Bergerat et al. [13] (recently separated from the Concrete Library [42] that implements higher-level operations and interfaces);

`FPT` **[126]:** an experimental FPGA accelerator for TFHE bootstrapping (a benchmark of state-of-the-art implementations in software/GPU/FPGA/ASIC can also be found in [126]);

`nuFHE` **[105]:** a GPU implementation of TFHE.

TFHE is also implemented as a part of more generic libraries like OpenFHE [4], which is a successor of PALISADE [108] and which also attempts to incorporate the capabilities of HElib [72] and HEAAN [121]. There exist many other implementations that are not listed here.

## 1.6 Conclusion

We believe that our TFHE guide helps many researchers and developers understand the inner structure of TFHE, in particular probably the most mysterious operation – the negacyclic blind-rotate – thanks to a step-by-step explanation, which we support with an illustration. Not only do we provide an intelligible description of each sub-operation of bootstrapping, but we also highlight what tweaks can be put in place to limit the noise growth at little to no additional cost (e.g., the order of key-switch $\leftrightarrow$ dot-product or the signed decomposition alphabet). Last but not least, we provide a comprehensive noise analysis, supported by proofs, where we employ an easy-to-follow notation of the decomposition operation using $\mathbf{g}$ and $\mathbf{g}^{-1}$. Finally, we list selected implementation remarks that shall be kept in mind when attempting to implement TFHE and its variants or modifications.

## Appendix

## A Proofs

### A.1 Proof of Theorem 1.2

*Theorem* 1.2 (Correctness & Noise Growth of $\boxdot$). Given GLWE sample $\bar{\mathbf{c}}$ of $\mu_c \in \mathbb{T}^{(N)}[X]$ under GLWE key $\mathbf{z}$ and noise parameter $\alpha$, and GGSW sample $\bar{\mathbf{A}}$ of $m_A \in \mathbb{Z}^{(N)}[X]$ under the same key and noise parameters, external product returns GLWE sample $\bar{\mathbf{c}}' = \bar{\mathbf{A}} \boxdot \bar{\mathbf{c}}$, which holds excess noise $e_\boxdot$, given by $\langle \bar{\mathbf{z}}, \bar{\mathbf{c}}' \rangle = m_A \cdot \langle \bar{\mathbf{z}}, \bar{\mathbf{c}} \rangle + e_\boxdot$, for which it holds

$$\mathsf{Var}[e_\boxdot] \approx \underbrace{dNV_B\alpha^2(1+k)}_{\text{amplified GGSW noise}} + \\ + \underbrace{\|m_A\|_2^2 \cdot \varepsilon^2(1+kNV_\mathbf{z})}_{\text{decomp. errors}}, \tag{37}$$

where $V_\mathbf{z}$ is the variance of individual coefficients of the GLWE key $\mathbf{z}$ and other parameters are as per previous definitions. If $e_\boxdot$ and the noise of $\bar{\mathbf{c}}$ are "sufficiently small", $\bar{\mathbf{c}}'$ encrypts $\mathsf{msg}_\mathbf{z}(\bar{\mathbf{c}}') = m_A \cdot \mu_c$, i.e., external product is indeed multiplicatively homomorphic.

*Proof.* Let us denote $\bar{\mathbf{c}} = (b, \mathbf{a}) \in \mathbb{T}^{(N)}[X]^{1+k}$ and let us unfold the construction of $\bar{\mathbf{A}}$ as

$$\bar{\mathbf{A}} = \left(\begin{array}{c|c} -\mathbf{A}_0\mathbf{z}+\mathbf{e} & \mathbf{A}_0 \\ -\mathbf{A}_1\mathbf{z}+\mathbf{e} & \mathbf{A}_1 \\ \vdots & \vdots \\ -\mathbf{A}_k\mathbf{z}+\mathbf{e} & \mathbf{A}_k \end{array}\right) + \begin{pmatrix} m_A\,\mathbf{g} & \mathbf{0} & \dots & \mathbf{0} \\ \mathbf{0} & m_A\,\mathbf{g} & \dots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{0} & \mathbf{0} & \dots & m_A\,\mathbf{g} \end{pmatrix}, \tag{38}$$

where $\mathbf{A}_i \in \mathbb{T}^{(N)}[X]^{d,k}$ with $j$-th column denoted $\mathbf{A}_i^{(j)}$. Unfolding the construction of $\bar{\mathbf{A}}$ and that of external product, we obtain

$$\langle \bar{\mathbf{z}}, \bar{\mathbf{c}}' \rangle_{1+k} = \langle (1,\mathbf{z}), \mathbf{g}^{-1}(\bar{\mathbf{c}})^T \cdot \bar{\mathbf{A}} \rangle_{1+k} = \langle \mathbf{g}^{-1}(b), m_A\, \mathbf{g} \underbrace{-\mathbf{A}_0\mathbf{z}}_{-\blacklozenge} + \mathbf{e} \rangle_d + \sum_{i=1}^{k} \langle \mathbf{g}^{-1}(a_i), \underbrace{-\mathbf{A}_i\mathbf{z}}_{-\heartsuit} + \mathbf{e} \rangle_d +$$

$$+ \sum_{j=1}^{k} z_j \Big( \underbrace{\langle \mathbf{g}^{-1}(b), \mathbf{A}_0^{(j)} \rangle_d}_{+\blacklozenge} + \underbrace{\sum_{i=1}^{k} \langle \mathbf{g}^{-1}(a_i), \mathbf{A}_i^{(j)} \rangle_d}_{+\heartsuit} + \langle \mathbf{g}^{-1}(a_j), m_A\, \mathbf{g} \rangle_d \Big) = \tag{39}$$

$$= m_A \cdot \Big( \underbrace{\langle \mathbf{g}^{-1}(b), \mathbf{g} \rangle_d}_{\approx b} \pm b \Big) + m_A \sum_{j=1}^{k} z_j \Big( \underbrace{\langle \mathbf{g}^{-1}(a_j), \mathbf{g} \rangle_d}_{\approx a_j} \pm a_j \Big) + \langle \mathbf{g}^{-1}(b), \mathbf{e} \rangle_d +$$

$$+ \sum_{i=1}^{k} \langle \mathbf{g}^{-1}(a_i), \mathbf{e} \rangle_d =$$

$$= m_A \cdot \underbrace{(b + \langle \mathbf{z}, \mathbf{a} \rangle)}_{\langle \bar{\mathbf{z}}, \bar{\mathbf{c}} \rangle} + \underbrace{m_A \cdot \Big( \langle \mathbf{g}^{-1}(b), \mathbf{g} \rangle_d - b + \sum_{j=1}^{k} z_j \big( \langle \mathbf{g}^{-1}(a_j), \mathbf{g} \rangle_d - a_j \big) \Big) +}_{\text{decomp. errors: } \|m_A\|_2^2 \cdot \varepsilon^2 (1 + kNV_{\mathbf{z}})} \tag{40}$$

$$\underbrace{+ \langle \mathbf{g}^{-1}(b), \mathbf{e} \rangle_d + \sum_{i=1}^{k} \langle \mathbf{g}^{-1}(a_i), \mathbf{e} \rangle_d}_{\text{amplified GGSW noise: } dNV_B\alpha^2(1+k)}, \tag{41}$$

while in (39), terms denoted ♦ and ♥ cancel out. Next, in (40), we assume that each decomposition error term (cf. (1.7)) has a uniform distribution on $[-1/2B^d, 1/2B^d)$, hence variance of $\varepsilon^2$; cf. (1.8). Finally, in (41), we assume that the decomposition digits have a uniform distribution on $[-B/2, B/2) \cap \mathbb{Z}$, hence their mean of squares equals $V_B$; cf. (1.9). Evaluating the variance of each term, the result follows, while $\approx$ is due to the possible statistical dependency of variables across terms. Note that we indicate the length of inner products in lower indices. □

## A.2 Proof of Theorem 4.3

*Theorem* 4.3 (Correctness & Noise Growth of Key-Switching). Given LWE sample $\bar{\mathbf{c}}'$ of $\mu \in \mathbb{T}$ under LWE key $\mathbf{s}'$ and key-switching keys $\mathsf{KS}_{\mathbf{s}' \to \mathbf{s}}$, encrypted with noise parameter $\alpha'$, $\mathsf{KeySwitch}_{\mathbf{s}' \to \mathbf{s}}$ returns LWE sample $\bar{\mathbf{c}}$, which holds excess noise $e_{\mathsf{KS}}$, given by $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}} \rangle = \langle \bar{\mathbf{s}}', \bar{\mathbf{c}}' \rangle + e_{\mathsf{KS}}$, for which it holds

$$\mathsf{Var}[e_{\mathsf{KS}}] \approx \underbrace{n' V_{\mathbf{s}'} \varepsilon'^2}_{\text{decomp. errors}} + \underbrace{n' d' V_{B'} \alpha'^2}_{\text{amplif. KS noise}}, \tag{42}$$

where $\varepsilon'^2$ and $V_{B'}$ are as per (1.8) and (1.9), respectively, with $B'$ and $d'$, $V_{\mathbf{s}'}$ is the variance of individual coefficients of the LWE key $\mathbf{s}'$, and other parameters are as per previous definitions. If $e_{\mathsf{KS}}$ and the noise of $\bar{\mathbf{c}}'$ are "sufficiently small", it holds $\mu = \mathsf{msg}_{\mathbf{s}}(\bar{\mathbf{c}}) = \mathsf{msg}_{\mathbf{s}'}(\bar{\mathbf{c}}')$, i.e., KeySwitch indeed changes the key, without modifying the message.

*Proof.* Similar to the proof of Theorem 1.2, we write

$$
\begin{aligned}
\langle \bar{\mathbf{s}}, \bar{\mathbf{c}} \rangle &= \Big\langle (1,\mathbf{s}), (b', \mathbf{0}) - \sum_{j=1}^{n'} \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j \Big\rangle_{1+n} = \\
&= b' + \sum_{j=1}^{n'} \Big\langle (1,\mathbf{s}), \mathbf{g}'^{-1}(a'_j)^T \cdot \underbrace{\Big[ s'_j\, \mathbf{g}' - \mathbf{A}_j \mathbf{s} + \mathbf{e}_j \;\Big|\; \mathbf{A}_j \Big]}_{\mathsf{KS}_j} \Big\rangle_{1+n} = \\
&= \underbrace{b + \sum_{j=1}^{n'} s'_j \Big( \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{g}' \big\rangle_{d'} \pm a'_j \Big) - \overbrace{\sum_{j=1}^{n'} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{A}_j \mathbf{s} \big\rangle_{d'}}^{-\heartsuit} + \sum_{j=1}^{n'} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{e}_j \big\rangle_{d'}}_{1 \,\cdot\, \text{first element of } \sum \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j} + \\
&\quad + \underbrace{\sum_{j=1}^{n'} \big\langle \mathbf{s}, \mathbf{g}'^{-1}(a'_j)^T \cdot \mathbf{A}_j \big\rangle_n}_{+\heartsuit} = \\
&= \underbrace{b' + \langle \mathbf{s}', \mathbf{a}' \rangle_{n'}}_{\langle \bar{\mathbf{s}'}, \bar{\mathbf{c}}' \rangle} + \underbrace{\sum_{j=1}^{n'} s'_j \Big( \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{g}' \big\rangle_{d'} - a'_j \Big)}_{\text{decomp. errors: } n' V_{\mathbf{s}'} \varepsilon'^2} + \underbrace{\sum_{j=1}^{n'} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{e}_j \big\rangle_{d'}}_{\text{amplified } \mathsf{KS} \text{ noise: } n' d' V_{B'} \alpha'^2} \qquad (43)
\end{aligned}
$$

and the result follows. $\square$

## A.3 Proof of Theorem 4.2

*Theorem* 4.2 (Correctness & Noise Growth of Blind-Rotate). Given inputs of Algorithm 1, where bootstrapping keys are encrypted with noise parameter $\alpha$ and test vector is (possibly) encrypted with noise parameter $\alpha_t$ (i.e., $\alpha_t = 0$ or $\alpha$), BlindRotate returns the last-step ACC with noise variance given by

$$
\begin{aligned}
\mathsf{Var}[\langle \bar{\mathbf{z}}, \mathsf{ACC} \rangle] \approx \alpha_t^2 + ndNV_B\alpha^2(1+k) &+ \\
+ n\varepsilon^2(1 + kNV_{\mathbf{z}}) &=: V_0, \qquad (44)
\end{aligned}
$$

which we denote by $V_0$, other parameters are as per previous theorems and definitions. If the noise of $\langle \bar{\mathbf{z}}, \mathsf{ACC} \rangle$ is "sufficiently small", it holds $\mathsf{msg}_{\mathbf{z}}(\mathsf{ACC}) = X^{\tilde{m}} \cdot tv$, where $\tilde{m} = \langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}}) \rangle \approx 2N\mu$, i.e., BlindRotate indeed "rotates" the test vector by the approximate phase of $(b, \mathbf{a})$, scaled to $\mathbb{Z}_{2N}$.

*Proof.* The core of BlindRotate consists of the sample $\bar{\mathbf{t}}$ being gradually externally-multiplied by $\mathsf{BK}_i$'s (plus some other additions/multiplications). For $i$-th step with $a := \tilde{a}_i \in \mathbb{Z}_{2N}$ and $\mathsf{BK} := \mathsf{BK}_i$ that encrypts $s := s_i \in \{0,1\}$, we write:

$$
\begin{aligned}
&\big\langle \bar{\mathbf{z}}, \mathsf{ACC} + \mathsf{BK} \boxdot (X^a \cdot \mathsf{ACC} - \mathsf{ACC}) \big\rangle = \\
&= \big\langle \bar{\mathbf{z}}, \mathsf{ACC} \big\rangle + s \cdot \big\langle \bar{\mathbf{z}}, X^a \cdot \mathsf{ACC} - \mathsf{ACC} \big\rangle + e_{\boxdot}(s) = \qquad (45) \\
&= \big\langle \bar{\mathbf{z}}, X^{s \cdot a} \cdot \mathsf{ACC} \big\rangle + e_{\boxdot}(s), \qquad (46)
\end{aligned}
$$

where (45) is by Theorem 1.2 and the step towards (23) holds for $s \in \{0,1\}$. Hence, with each such a step, the noise grows by the additive term $e_{\boxdot}(s)$. The first step $X^{\tilde{b}} \cdot \bar{\mathbf{t}}$ retains the noise of $\bar{\mathbf{t}}$, hence the result follows. $\square$

## B Technical Overview

In Figure 9, we provide an exhaustive technical overview of blind-rotate, preceded by modulus-switching and followed by sample-extract.

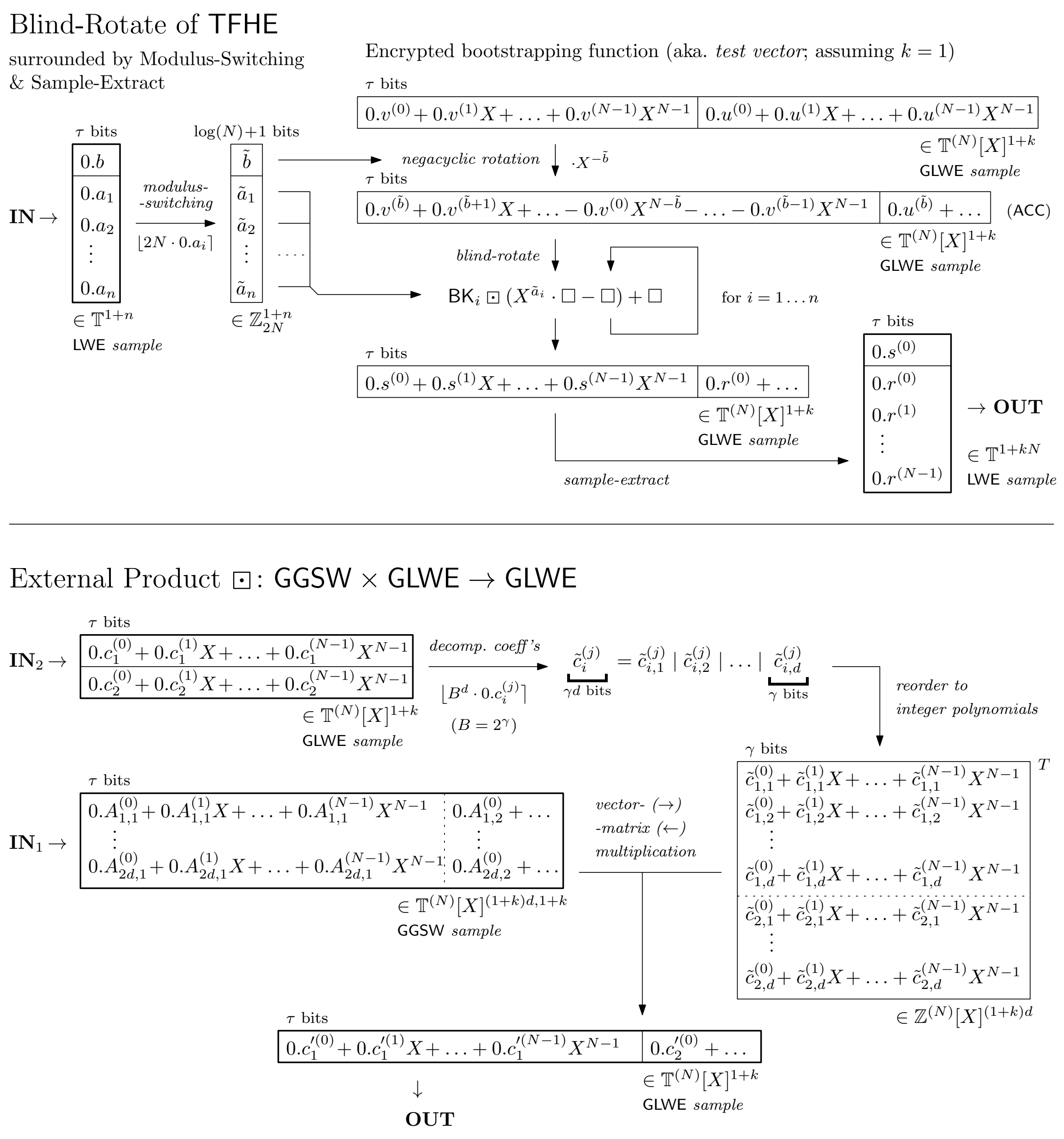

Blind-Rotate of TFHE
surrounded by Modulus-Switching & Sample-Extract
Encrypted bootstrapping function (aka. test vector; assuming $k = 1$)
$\tau$ bits
$0.v^{(0)} + 0.v^{(1)}X + \ldots + 0.v^{(N-1)}X^{N-1}$
$0.u^{(0)} + 0.u^{(1)}X + \ldots + 0.u^{(N-1)}X^{N-1}$
$\in \mathbb{T}^{(N)}[X]^{1+k}$
GLWE sample
IN →
$0.b$, $0.a_1$, $0.a_2$, …, $0.a_n$
$\in \mathbb{T}^{1+n}$
LWE sample
modulus-switching
$\lfloor 2N \cdot 0.a_i \rceil$
$\log(N)+1$ bits
$\tilde{b}$, $\tilde{a}_1$, $\tilde{a}_2$, …, $\tilde{a}_n$
$\in \mathbb{Z}_{2N}^{1+n}$
negacyclic rotation
$\cdot X^{-\tilde{b}}$
$0.v^{(\tilde{b})} + 0.v^{(\tilde{b}+1)}X + \ldots - 0.v^{(0)}X^{N-\tilde{b}} - \ldots - 0.v^{(\tilde{b}-1)}X^{N-1}$
$0.u^{(\tilde{b})} + \ldots$
(ACC)
blind-rotate
$\mathsf{BK}_i \boxdot (X^{\tilde{a}_i} \cdot \square - \square) + \square$
for $i = 1 \ldots n$
$0.s^{(0)} + 0.s^{(1)}X + \ldots + 0.s^{(N-1)}X^{N-1}$
$0.r^{(0)} + \ldots$
sample-extract
$0.s^{(0)}$, $0.r^{(0)}$, $0.r^{(1)}$, …, $0.r^{(N-1)}$
→ OUT
$\in \mathbb{T}^{1+kN}$
LWE sample
External Product $\boxdot$: GGSW × GLWE → GLWE
IN$_2$ →
$0.c_1^{(0)} + 0.c_1^{(1)}X + \ldots + 0.c_1^{(N-1)}X^{N-1}$
$0.c_2^{(0)} + 0.c_2^{(1)}X + \ldots + 0.c_2^{(N-1)}X^{N-1}$
decomp. coeff's
$\lfloor B^d \cdot 0.c_i^{(j)} \rceil$
$(B = 2^\gamma)$
$\underbrace{\tilde{c}_i^{(j)}}_{\gamma d \text{ bits}} = \tilde{c}_{i,1}^{(j)} \mid \tilde{c}_{i,2}^{(j)} \mid \ldots \mid \underbrace{\tilde{c}_{i,d}^{(j)}}_{\gamma \text{ bits}}$
reorder to integer polynomials
$\gamma$ bits
$\tilde{c}_{1,1}^{(0)} + \tilde{c}_{1,1}^{(1)}X + \ldots + \tilde{c}_{1,1}^{(N-1)}X^{N-1}$
$\tilde{c}_{1,2}^{(0)} + \tilde{c}_{1,2}^{(1)}X + \ldots + \tilde{c}_{1,2}^{(N-1)}X^{N-1}$
$\tilde{c}_{1,d}^{(0)} + \tilde{c}_{1,d}^{(1)}X + \ldots + \tilde{c}_{1,d}^{(N-1)}X^{N-1}$
$\tilde{c}_{2,1}^{(0)} + \tilde{c}_{2,1}^{(1)}X + \ldots + \tilde{c}_{2,1}^{(N-1)}X^{N-1}$
$\tilde{c}_{2,d}^{(0)} + \tilde{c}_{2,d}^{(1)}X + \ldots + \tilde{c}_{2,d}^{(N-1)}X^{N-1}$
$T$
$\in \mathbb{Z}^{(N)}[X]^{(1+k)d}$
IN$_1$ →
$0.A_{1,1}^{(0)} + 0.A_{1,1}^{(1)}X + \ldots + 0.A_{1,1}^{(N-1)}X^{N-1}$
$0.A_{1,2}^{(0)} + \ldots$
$0.A_{2d,1}^{(0)} + 0.A_{2d,1}^{(1)}X + \ldots + 0.A_{2d,1}^{(N-1)}X^{N-1}$
$0.A_{2d,2}^{(0)} + \ldots$
$\in \mathbb{T}^{(N)}[X]^{(1+k)d,1+k}$
GGSW sample
vector- (→) -matrix (←) multiplication
$0.c_1'^{(0)} + 0.c_1'^{(1)}X + \ldots + 0.c_1'^{(N-1)}X^{N-1}$
$0.c_2'^{(0)} + \ldots$
OUT

Figure 9: Technical overview of TFHE Blind-Rotate, preceded by Modulus-Switching and followed by Sample-Extract, with a detail on External Product.

# Chapter 2

# Negacyclic Integer Convolution using Extended Fourier Transform

With the rise of lattice cryptography, (negacyclic) convolution has received increased attention. E.g., the NTRU scheme internally employs cyclic polynomial multiplication, which is equivalent to the standard convolution, on the other hand, many Ring-LWE-based cryptosystems perform negacyclic polynomial multiplication. A method by Crandall implements an efficient negacyclic convolution over a finite field of prime order using an extended Discrete Galois Transform (DGT) – a finite field analogy to Discrete Fourier Transform (DFT). Compared to DGT, the classical DFT runs faster by an order of magnitude, however, it suffers from inevitable rounding errors due to finite floating-point number representation. In a recent Fully Homomorphic Encryption (FHE) scheme by Chillotti et al. named TFHE, small errors are acceptable (although not welcome), therefore we decided to investigate the application of DFT for negacyclic convolution.

The primary goal of this chapter is to suggest a method for fast negacyclic convolution over integer coefficients using an extended DFT. The key contribution is a thorough analysis of error propagation, as a result of which we derive parameter bounds that can guarantee even error-free results. We also suggest a setup that admits rare errors, which allows to increase the degree of the polynomials and/or their maximum norm at a fixed floating-point precision. Finally, we run benchmarks with parameters derived from a practical TFHE setup. We achieve around 24$\times$ better times than the generic NTL library (comparable to Crandall's method) and around 4$\times$ better times than a naïve approach with DFT, with no errors.

## 2.1 Introduction

In 1994, Peter Shor discovered efficient quantum algorithms for discrete logarithm and factoring [119], which started the quest to design novel quantum-proof algorithms, aka. *Post-Quantum Cryptography.* Since then, there have emerged many new schemes, which are based on various problems that are believed to be quantum hard. E.g., supersingular elliptic curve isogeny [74], multivariate cryptography [48], or lattice cryptography [3], in particular Learning With Errors (LWE) and its variants [113, 98]. In addition, many *Fully Homomorphic Encryption* (FHE) schemes (e.g. [23, 35]) belong to lattice-based ones, including Gentry's first-ever FHE scheme [58]. Most notably, the NIST's Post-Quantum Cryptography Standardization Program entered the third “Selection Round” in July 2020 [104], while lattice-based cryptosystems occur among the selected algorithms.

With the popularity of lattice-based cryptography, the need for its fast implementation has risen. Besides linear algebra, many schemes require a fast algorithm for cyclic (i.e., mod $X^N - 1$) or negacyclic (i.e., mod $X^N + 1$) polynomial multiplication. Some schemes work with polynomial coefficients modulo an integer (e.g., NTRU [70]), however, our main interest is in the TFHE scheme [35], where negacyclic multiplication of integer-torus polynomials is performed. Here the *torus* refers to reals modulo 1, i.e., the fractional part of a real number. In practice, torus elements are represented as unsigned integers, which represent the fraction of 1 uniformly in the interval $[0, 1)$. It follows that integer-torus polynomial multiplication can be performed with their integer representation. Also note that TFHE accepts small errors – we prefer to avoid them, but their impact is not fatal for decryption.

Recently, there have emerged efforts to make TFHE work with multivalued plaintexts [25], also applications of TFHE for homomorphic evaluation of neural networks show promising results [20]. In particular, for neural networks, it holds that they are quite error-tolerant (also verified in [20]), which supports the acceptability of errors.

## Problem Statement

Our goal is to develop a method for fast negacyclic multiplication of univariate integer polynomials. For this method, we aim to estimate and tune its parameters in order to provide certain guarantees of its correctness. As outlined above, we will not focus solely on an error-free case and we will also accept the scenario, where errors may rarely occur. Last but not least—as we intend our method also for an FPGA implementation—we derive all results in a generic manner, i.e., without sticking to a concrete platform, although we run our tests on an ordinary 64-bit machine.

## Related Work

There is a long and rich history of methods for fast multiplication over various rings, ranging from Karatsuba’s algorithm [78], through Fast Fourier Transform (FFT; [43]) to Schönhage-Strassen algorithm [118]. Most of these methods are based on a similar principle as Bernstein pointed out in his survey [14].

It was the classical cyclic convolution, which was accelerated by FFT and Convolution Theorem, and which can be employed for polynomial multiplication modulo $X^N - 1$, too. On the contrary, polynomial multiplication modulo $X^N + 1$ (negacyclic convolution) cannot be directly calculated via FFT. One possible approach was implemented as a part of the TFHE Library [124], although not discussed in the paper [35]. However, this method suffers from a four-tuple redundancy in its intermediate results. An effective (non-redundant) method for negacyclic convolution has been proposed by Crandall [46] and recently improved by Al Badawi et al. [5]. In these methods, polynomials are considered over a finite ring and both authors employed a number-theoretic variant of FFT, named DGT, which operates on the field $\mathsf{GF}(p^2)$. On the one hand, DGT calculates exact results (as opposed to FFT, where rounding errors occur and propagate), on the other hand, it runs significantly slower as it uses modular arithmetics.

## Our Contributions

We propose an efficient algorithm for negacyclic convolution over the reals, for which we derive estimates of bounds on the maximum error and its variance. Based on our estimates, we show that our method can be used for an error-free negacyclic convolution over integers. Or—in case we admit errors—we suggest to relax the estimates in order to achieve higher performance: either in terms of shorter number representation (useful in particular for FPGA), longer polynomials, or larger polynomial coefficients that can be processed. Finally, we provide experimental benchmarking results of our implementation as well as we evaluate its rounding error magnitudes and result correctness, even with remarkably underestimated parameters.

### Chapter Outline

In Section 2.2, we provide a brief overview of the required mathematical background, i.e., cyclic and negacyclic convolutions, their relation to modular polynomial multiplication, as well as the Discrete Fourier Transform and Convolution Theorem. Next, in Section 2.3, we revisit a straightforward FFT-based approach for negacyclic polynomial multiplication, and we propose a method that avoids the calculation of redundant intermediates. We analyze error propagation thoroughly in Section 2.4, where we suggest lower bounds on floating point type bit-precision in order to guarantee certain levels of correctness. In Section 2.5, we discuss the implementation details and we propose a set of testing parameters with respect to TFHE. Using these parameters, we benchmark our implementation and we also examine the error magnitude and result correctness. Finally, we conclude this chapter in Section 2.6.

## 2.2 Preliminaries

In this section, we briefly recall some basic mathematical concepts related to convolution and Discrete Fourier Transform.

### Cyclic & Negacyclic Convolution

Let $\mathbf{f}, \mathbf{g} \in \mathbb{C}^N$ for some $N \in \mathbb{N}$. As opposed to the classical cyclic convolution defined as

$$(\mathbf{f} * \mathbf{g})_k := \sum_{j=0}^{N-1} f_j g_{(k-j) \bmod N}, \tag{2.1}$$

*negacyclic convolution* adds a factor of $-1$ with each wrap of the cyclic index at $\mathbf{g}$, i.e.,

$$(\mathbf{f} \mathbin{\bar{*}} \mathbf{g})_k := \sum_{j=0}^{N-1} (-1)^{\lfloor \frac{k-j}{N} \rfloor} f_j g_{(k-j) \bmod N}. \tag{2.2}$$

With respect to polynomials, it is easy to verify that the cyclic convolution calculates the coefficients of a product of two polynomials modulo $X^N - 1$. Indeed, their coefficients can be considered cyclic since $X^N = 1$. On the other hand, the *negacyclic* convolution calculates the coefficients of a product of two polynomials modulo $X^N + 1$, since $X^N = -1$ adds a factor of $-1$ with each wrap.

### Convolution Theorem

A relation known as the *Convolution Theorem* (CT) states an equality between the Fourier image of convoluted vectors and an element-wise (dyadic) product of their respective Fourier images (in the discrete variant). CT writes as follows:

$$\mathcal{F}(\mathbf{f} * \mathbf{g}) = \mathcal{F}(\mathbf{f}) \odot \mathcal{F}(\mathbf{g}), \tag{2.3}$$

where $\mathcal{F}(\cdot)$ stands for the *Discrete Fourier Transform* (DFT) and $\odot$ denotes the dyadic multiplication of two vectors. In fact, DFT is a change of basis, defined as

$$\mathcal{F}(\mathbf{f})_k := \sum_{j=0}^{N-1} f_j \exp\Big(-\frac{2\pi \mathrm{i} jk}{N}\Big) = F_k, \tag{2.4}$$

$$\mathcal{F}^{-1}(\mathbf{F})_j = \frac{1}{N}\sum_{k=0}^{N-1} F_k \exp\Big(\frac{2\pi \mathrm{i} jk}{N}\Big) = f_j. \tag{2.5}$$

Convolution theorem has gained its practical significance after *Fast Fourier Transform* (FFT) was (re)invented[1] in 1965 by Cooley & Tukey [43]. As opposed to a direct calculation of DFT coefficients, which requires $O(N^2)$ time, FFT runs in $O(N \log N)$. Next, by the convolution theorem, one can calculate the convolution of two vectors as $\mathbf{f} * \mathbf{g} = \mathcal{F}^{-1}\big(\mathcal{F}(\mathbf{f}) \odot \mathcal{F}(\mathbf{g})\big)$, which spends $O(N \log N)$ time, compared to $O(N^2)$ needed for a direct calculation.

## 2.3 Efficient Negacyclic Convolution

First, we describe a method for negacyclic convolution that uses the standard cyclic convolution and FFT. We identify its redundancy and briefly comment on possible workarounds. Next, we outline an approach that yields no redundancy and achieves a $4\times$ better performance than the previous method.

### 2.3.1 Redundant Approach

Since (negacyclic) convolution is equivalent to (negacyclic) polynomial modular multiplication, we switch to the polynomial point of view for now. Interested in polynomial multiplication modulo $X^N + 1$, we note that $X^{2N} - 1 = (X^N - 1) \cdot (X^N + 1)$. Hence, we can calculate the product first modulo $X^{2N} - 1$ (via cyclic convolution of $2N$ elements) and then only reduce the result modulo $X^N + 1$. This method can be optimized based on the following observations.

**Observation 2.1** (Redundancy of negacyclic extension)**.** *Let $p \in \mathbb{R}[X]$ be a real-valued polynomial of degree $N-1$, $N \in \mathbb{N}$, and let $\bar{p}(X) := p(X) - X^N \cdot p(X)$ be a negacyclic extension of $p(X)$. Then the Fourier image of* $\mathsf{coeffs}(\bar{p})$ *contains zeros at eventh positions (indexed from* 0*). In addition, the remaining coefficients (at oddth positions) are mirrored and conjugated. I.e.,*

$$\mathcal{F}\big(\mathsf{coeffs}(\bar{p})\big) = (0, P_1, 0, P_3, \ldots, 0, P_{N-1}, 0, \overline{P_{N-1}}, \ldots, 0, \overline{P_3}, 0, \overline{P_1}). \tag{2.6}$$

*Note* 2.1. Given $N$ input (real-valued) polynomial coefficients, $\mathcal{F}\big(\mathsf{coeffs}(\bar{p})\big)$ needs to calculate $2N$ complex values, i.e., $4N$ real values. The redundancy is clearly in the $N$ complex zeros and in the $N/2$ complex conjugates.

**Observation 2.2** (Convolution of negacyclic extensions)**.** *Let $p, q \in \mathbb{R}[X]$ be real-valued polynomials of degree $N-1$ for some $N \in \mathbb{N}$ and let $\bar{p}, \bar{q}$ be their respective negacyclic extensions. Then it holds*

$$\mathsf{coeffs}\big(p \cdot q \bmod (X^N + 1)\big) = \frac{1}{2}\,\mathcal{F}^{-1}\Big(\mathcal{F}\big(\mathsf{coeffs}(\bar{p})\big) \odot \mathcal{F}\big(\mathsf{coeffs}(\bar{q})\big)\Big)[0 \ldots N-1]. \tag{2.7}$$

By Observation 2.1, it follows that the dyadic multiplication in (2.7) can only be performed at odd positions of the first half, the rest can be copied (with appropriate sign). Also note that after $\mathcal{F}^{-1}$, the coefficients are negacyclic, hence we can only take the first half of the vector. This method is implemented in the original TFHE Library [124].

[1] Goldstine [63] attributes an FFT-like algorithm to C. F. Gauss dating to around 1805.

**Possible Improvements**

The clear goal is to omit all calculations leading to redundant values as outlined in Note 2.1. Digging deeper into FFT, we deduced the same initial step as proposed by Crandall [46] in his method for negacyclic convolution (namely, the folding step). However, without the additional twisting step, we ended up with a bunch of numbers, from which we were not able to recover the original values efficiently. Therefore, we decided to adapt the concept of the method by Crandall.

### 2.3.2 Non-Redundant Approach

The method for negacyclic polynomial multiplication by Crandall [46] is intended for polynomials over $\mathbb{Z}_p$ and it employs internally the *Discrete Galois Transform* (DGT). DGT is an analogy to DFT, which operates over the field $\mathsf{GF}(p^2)$ for a Gaussian prime number $p$, whereas DFT operates over $\mathbb{C}$. Note that recently Al Badawi et al. [5] extended the Crandall's method for non-Gaussian primes, too. The Crandall's method prepends DGT with two steps: folding and twisting. In the following definition we propose an analogous transformation using DFT.

**Definition 2.1.** Let $\mathbf{f} \in \mathbb{R}^N$ for some $N \in \mathbb{N}$, $N$ even. We define the *Discrete Fourier Negacyclic Transform* (DFNT, denoted $\bar{\mathcal{F}}$) as follows:

$$\bar{\mathcal{F}}(\mathbf{f}) := \mathcal{F}\Big(\underbrace{\big(\mathbf{f}[0\ldots {}^{N}\!/\!_2-1] + \mathrm{i}\cdot\mathbf{f}[{}^{N}\!/\!_2\ldots N-1]\big)}_{\text{folding}} \underbrace{\odot\big(\omega_{2N}^{j}\big)_{j=0}^{N/2-1}}_{\text{twisting}}\Big), \tag{2.8}$$

where $\omega_{2N}^{j} = \exp\big(\frac{2\pi\mathrm{i}j}{2N}\big)$ and $\mathcal{F}$ stands for the ordinary DFT. For the inverse DFNT, we have

$$\mathbf{t} := \mathcal{F}^{-1}(\mathbf{F}) \odot (\omega_{2N}^{-j})_{j=0}^{N/2-1}, \tag{2.9}$$

$$\bar{\mathcal{F}}^{-1}(\mathbf{F}) = \big[\Re(\mathbf{t}), \Im(\mathbf{t})\big]. \tag{2.10}$$

*Note* 2.2. We will refer to DFNT, where DFT is internally calculated via FFT, as the *Fast Fourier Negacyclic Transform* (FFNT).

With respect to negacyclic convolution, DFNT has two important properties:

1. given $N$ reals at input, it outputs ${}^{N}\!/\!_2$ complex numbers, i.e., there is *no redundancy*, unlike in the previous approach, and
2. it can be used for negacyclic convolution in the same manner as DFT for cyclic convolution, a theorem follows.

**Theorem 2.1** (Negacyclic Convolution Theorem; NCT). *Let* $\mathbf{f}, \mathbf{g} \in \mathbb{R}^N$ *for some* $N \in \mathbb{N}$, $N$ *even. It holds*

$$\bar{\mathcal{F}}(\mathbf{f} \mathbin{\bar{*}} \mathbf{g}) = \bar{\mathcal{F}}(\mathbf{f}) \odot \bar{\mathcal{F}}(\mathbf{g}). \tag{2.11}$$

For a full description of negacyclic convolution over the reals via NCT see Algorithm 3. Next, we analyze this algorithm from the error propagation point of view, which allows us to apply this method for negacyclic convolution over integers, too.

## 2.4 Analysis of Error Propagation

Since Algorithm 3 operates implicitly with real numbers (starting $N = 4$, $\omega_{2N}$'s are irrational), there emerge rounding errors provided that we use a standard finite floating-point representation.

**Algorithm 3** Efficient Negacyclic Convolution over $\mathbb{R}$.

```
Input: $\mathbf{f}, \mathbf{g} \in \mathbb{R}^N$ for some $N \in \mathbb{N}$, $N$ even.
Precompute: $\omega_{2N}^{j} \coloneqq \exp\left(\frac{2\pi i j}{2N}\right)$ for $j = -N/2 + 1 \dots N/2 - 1$.
Output: $\mathbf{h} \in \mathbb{R}^N$, $\mathbf{h} = \mathbf{f} \mathbin{\bar{*}} \mathbf{g}$.
 1: for $j = 0 \dots N/2 - 1$ do
 2:     $f'_j = f_j + i f_{j+N/2}$        // fold
 3:     $g'_j = g_j + i g_{j+N/2}$
 4: end for
 5: for $j = 0 \dots N/2 - 1$ do
 6:     $f''_j = f'_j \cdot \omega_{2N}^{j}$        // twist
 7:     $g''_j = g'_j \cdot \omega_{2N}^{j}$
 8: end for
 9: $\mathbf{F} = \mathcal{F}_{N/2}(\mathbf{f}'')$, $\mathbf{G} = \mathcal{F}_{N/2}(\mathbf{g}'')$
10: for $j = 0 \dots N/2 - 1$ do
11:     $H_j = F_j \cdot G_j$
12: end for
13: $\mathbf{h}'' = \mathcal{F}^{-1}{}_{N/2}(\mathbf{H})$
14: for $j = 0 \dots N/2 - 1$ do
15:     $h'_j = h''_j \cdot \omega_{2N}^{-j}$        // untwist
16: end for
17: for $j = 0 \dots N/2 - 1$ do
18:     $h_j = \Re(h'_j)$        // unfold
19:     $h_{j+N/2} = \Im(h'_j)$
20: end for
21: return $\mathbf{h}$
```

In this section, we analyze Algorithm 3 from the error propagation point of view and we derive estimates of the bounds of errors as well as their variance. Based on our estimates, we derive a bound for sufficient bit-precision of the employed floating point representation, which guarantees error-free convolution over the ring of integers. We also provide an estimate of the bit-precision based on error variance and the $3\sigma$-rule. In addition and as a byproduct, we derive all bounds for cyclic convolution, too. First of all, we revisit the FFT algorithm, as we will refer to it later.

## FFT in Brief

FFT [43] is a recursive algorithm, which builds upon the following observation: for $N = n_1 \cdot n_2$ and $k = k_1 + k_2 n_1$, we can write the $k$-th Fourier coefficient of an $\mathbf{f} \in \mathbb{C}^N$ as

$$\mathcal{F}(\mathbf{f})_{k_1+k_2n_1} = \sum_{j_2=0}^{n_2-1} \left( \underbrace{\left( \sum_{j_1=0}^{n_1-1} f_{j_2+j_1n_2} \omega_{n_1}^{j_1k_1} \right)}_{\mathcal{F}\left((f_{j_2+j_1n_2})_{j_1=0}^{n_1-1}\right)_{k_1}} \omega_N^{-j_2k_1} \right) \omega_{n_2}^{-j_2k_2}, \tag{2.12}$$

where

$$\omega_N^j = \exp\left(\frac{2\pi \mathrm{i} j}{N}\right), \tag{2.13}$$

while $\omega$'s can be precomputed.

*Note* 2.3. There exist two major FFT data paths for $N$ a power of two: the Cooley-Tukey data path [43] (aka. decimation-in-time), and the Gentleman-Sande data path [57] (aka. decimation-in-frequency). At this point, let us describe the decimation-in-time data path, we will discuss their implementation consequences later in Section 2.5.

For $N$ a power of two, FFT splits its input into two halves and proceeds recursively. Next, it multiplies the results with $\omega$'s, and finally it proceeds adequate pairs; see (2.14) and (2.15).

At the end of the recursion we have for $N = 2$:

$$\mathsf{FFT}_2 \left| f_0 \quad f_1 \right| = \left| f_0 + f_1 \quad f_0 - f_1 \right|. \tag{2.14}$$

Next, for $N \geq 4$ we have

$$\mathsf{FFT}_N(\mathbf{f})\colon \left|\begin{matrix} f_0 & f_1 \\ f_2 & f_3 \\ \vdots & \vdots \\ f_{N-2} & f_{N-1} \end{matrix}\right|_{n_1 \times n_2 \,=\, N/2 \times 2} \xrightarrow[\text{(recursively)}]{\mathsf{FFT}_{N/2}\ \text{columns}} \left|\begin{matrix} f'_0 & f'_1 \\ f'_2 & f'_3 \\ \vdots & \vdots \\ f'_{N-2} & f'_{N-1} \end{matrix}\right| \odot \left|\begin{matrix} 1 & 1 \\ 1 & \omega_N^{-1\cdot 1} \\ \vdots & \vdots \\ 1 & \omega_N^{-1\cdot(N/2-1)} \end{matrix}\right|_{\omega_N^{-j_2k_1}} \longrightarrow$$

$$\rightarrow \left|\begin{matrix} f''_0 & f''_1 \\ f''_2 & f''_3 \\ \vdots & \vdots \\ f''_{N-2} & f''_{N-1} \end{matrix}\right| \xrightarrow{\mathsf{FFT}_2\ \text{rows}} \left|\begin{matrix} f''_0 + f''_1 & f''_0 - f''_1 \\ f''_2 + f''_3 & f''_2 - f''_3 \\ \vdots & \vdots \\ f''_{N-2} + f''_{N-1} & f''_{N-2} - f''_{N-1} \end{matrix}\right| = \left|\begin{matrix} F_0 & F_{N/2} \\ F_1 & F_{N/2+1} \\ \vdots & \vdots \\ F_{N/2-1} & F_{N-1} \end{matrix}\right|. \tag{2.15}$$

$\mathsf{FFT}^{-1}$ proceeds similarly to the direct transformation with the following exceptions:

1. in the second step, it multiplies by $\omega_N^{j_2k_1}$ (i.e., with a positive exponent), and
2. the final result is multiplied by $1/N$ (only once at the top level).

### 2.4.1 Error Propagation through FFT and FFNT

Let us begin with two lemmas, which provide bounds on the error and variance of complex multiplication and FFT, respectively. Note that we will assume for our estimates of variance bounds that the rounding errors are uniformly random and independent.

*Note* 2.4. We will distinguish two types of the maximum norm $\|\cdot\|_\infty$ over $\mathbb{C}^N$. For 1. error vectors, and for 2. other complex vectors, we consider:

1. the maximum of real and imaginary parts (i.e., rectangular), and
2. the maximum of absolute values (i.e., circular), respectively.

**Lemma 2.2.** *Let $a, b \in \mathbb{C}$, $|a| \leq A_0$ and $|b| \leq B_0$ for some $A_0, B_0 \in \mathbb{R}^+$. Then*

$$|a \cdot b| \leq A_0 \cdot B_0, \tag{2.16}$$

$$\|\mathsf{Err}(a \cdot b)\|_\infty \lessapprox \sqrt{2} \cdot \big(A_0 \cdot \|\mathsf{Err}(b)\|_\infty + B_0 \cdot \|\mathsf{Err}(a)\|_\infty\big), \quad \textit{and} \tag{2.17}$$

$$\mathsf{Var}\big(\mathsf{Err}(a \cdot b)\big) \lessapprox 2 \cdot \big(A_0^2 \cdot \mathsf{Var}\big(\mathsf{Err}(b)\big) + B_0^2 \cdot \mathsf{Var}\big(\mathsf{Err}(a)\big)\big), \tag{2.18}$$

*where we neglected second-order error terms and for* (2.18)*, we further assumed that the errors of $a$ and $b$ are independent.*

*Proof.* Let $a = (p + E_p) + \mathrm{i}(q + E_q)$ and $b = (r + E_r) + \mathrm{i}(s + E_s)$, where we denote the parts' bounds as $|p| \leq P_0$ etc. According to Note 2.4, we split the complex error into parts – we write for the real part (similarly for the complex part)

$$\mathsf{Err}\big(\Re(a \cdot b)\big) = pE_r + rE_p - (qE_s + sE_q) + negl., \tag{2.19}$$

which can be bounded as

$$\begin{aligned}\big|\mathsf{Err}\big(\Re(a \cdot b)\big)\big| &\lessapprox P_0\|\mathsf{Err}(b)\|_\infty + R_0\|\mathsf{Err}(a)\|_\infty + Q_0\|\mathsf{Err}(b)\|_\infty + S_0\|\mathsf{Err}(a)\|_\infty \lessapprox \\ &\lessapprox (P_0 + Q_0)\|\mathsf{Err}(b)\|_\infty + (S_0 + R_0)\|\mathsf{Err}(a)\|_\infty.\end{aligned} \tag{2.20}$$

Since $|p + \mathrm{i}q| \lessapprox A_0$, we can bound $P_0 + Q_0 \lessapprox \sqrt{2}A_0$ and the result (2.17) follows, similarly for (2.18). □

**Lemma 2.3.** *Let $\mathbf{f} \in \mathbb{C}^N$, where $N = 2^\nu$ for some $\nu \in \mathbb{N}$, $\|\mathbf{f}\|_\infty \leq 2^{\varphi_0}$ for some $\varphi_0 \in \mathbb{N}$, and let $\chi$ denote the bit-precision of $\omega$'s as well as all intermediate values during the calculation of $\mathsf{FFT}_N(\mathbf{f}) =: \mathbf{F}$, represented as a floating point type. Then*

$$\|\mathbf{F}\|_\infty \leq 2^{\varphi_0+\nu}, \tag{2.21}$$

$$\|\mathsf{Err}(\mathbf{F})\|_\infty \lessapprox c_H \cdot \big(\sqrt{2} + 1\big)^\nu + c_N \cdot 2^\nu \quad (\textit{for } \nu \geq 2), \quad \textit{and} \tag{2.22}$$

$$\mathsf{Var}\big(\mathsf{Err}(\mathbf{F})\big) \lessapprox d_H \cdot 3^\nu + d_N \cdot 4^\nu \quad (\textit{for } \nu \geq 2), \tag{2.23}$$

*where*

$$\begin{aligned} c_H &= 2(\sqrt{2} - 1) \cdot \|\mathsf{Err}(\mathbf{f})\|_\infty + (2 - \sqrt{2}) \cdot 2^{\varphi_0-\chi+1}, & c_N &= -(2 + \sqrt{2}) \cdot 2^{\varphi_0-\chi-1}, \\ d_H &= {}^2\!/_3 \, \mathsf{Var}\big(\mathsf{Err}(\mathbf{f})\big) - {}^8\!/_{27} \, 2^{2\varphi_0-2\chi}, & d_N &= {}^1\!/_6 \, 2^{2\varphi_0-2\chi}. \end{aligned} \tag{2.24}$$

*Proof.* We write

$$\mathsf{FFT}_N\colon\ \mathsf{FFT}_2 \circ (\odot\, \omega_N) \circ \mathsf{FFT}_{N/2}, \tag{2.25}$$

from where we derive recurrence relations for the bounds on absolute value, error and variance.

In each recursion level, the values propagate to a lower level, then they are multiplied by a complex unit and two such values are added, or subtracted. Firstly, note that in every level the initial bound on the absolute value is doubled, hence (2.21) follows.

Regarding the errors, it is important to note that the final $\mathsf{FFT}_2$ acts on two values, each of which has been previously multiplied by $\omega_N^{j_2 k_1}$, where $j_2$ ranges in $\{0, 1\}$. I.e., one value is multiplied by 1 and only the other is multiplied by a (mostly) non-trivial complex unit, which is rounded to $\chi$ bits of precision, i.e., $\|\mathsf{Err}(\omega)\|_\infty \leq 2^{-\chi-1}$. Putting things together, we get the following recurrence relations for the bounds on the error and its variance after $\nu$ levels, respectively:

$$\begin{aligned} E_\nu &= \sqrt{2} \cdot \left(1 \cdot E_{\nu-1} + 2^{\varphi_0+\nu-1} \cdot 2^{-\chi-1}\right) + E_{\nu-1} = \\ &= \left(\sqrt{2}+1\right) \cdot E_{\nu-1} + \sqrt{2} \cdot 2^{\varphi_0+\nu-\chi-2}, && (2.26) \\ E_2 &= (E_1 + 2^{\varphi_0+1} \cdot \underbrace{E_{\omega_4}}_{=0}) \cdot \sqrt{2} + E_1 = (\sqrt{2}+1)E_1 = 2(\sqrt{2}+1)E_0, \quad \text{and} && (2.27) \\ V_\nu &= 2 \cdot \left(1^2 \cdot V_{\nu-1} + (2^{\varphi_0+\nu-1})^2 \cdot {}^1\!/_{12}\,(2^{-\chi})^2\right) + V_{\nu-1} = \\ &= 3V_{\nu-1} + {}^1\!/_3\, 2^{2\varphi_0+2\nu-2\chi-3}, && (2.28) \\ V_2 &= 3V_1 = 6V_0, && (2.29) \end{aligned}$$

where in (2.27), we applied the fact that $\omega_4$ is error-free; cf. (2.13). Also note that the error more than doubles in each step (while the bound only doubles), therefore the $\chi$ bits of precision are sufficient and rounding errors can be neglected. The results follow by solving (2.26) and (2.27), and (2.28) and (2.29), respectively. □

In the following proposition, we bound the error and variance of the result of cyclic and negacyclic convolution via FFT / FFNT, respectively. For a quick reference, we provide an overview of these methods in (2.30) and (2.31), respectively:

$$\begin{tikzcd}[row sep=tiny] \mathbf{f} \arrow[r, "\mathsf{FFT}_N"] & \mathbf{F} \arrow[dr, phantom, ""] & & \\ & & \overset{\odot}{\longrightarrow} \mathbf{H} \arrow[r, "\mathsf{FFT}_N^{-1}"] & \mathbf{h} = \mathbf{f} * \mathbf{g}, \\ \mathbf{g} \arrow[r, "\mathsf{FFT}_N"] & \mathbf{G} & & \end{tikzcd} \qquad (2.30)$$

$$\begin{tikzcd}[row sep=tiny] \mathbf{f} \arrow[r, "\text{fold}"] & \mathbf{f}' \arrow[r, "\text{twist}"] & \mathbf{f}'' \arrow[r, "\mathsf{FFT}_{N/2}"] & \bar{\mathbf{F}} & & & & \\ & & & & \overset{\odot}{\longrightarrow} \bar{\mathbf{H}} \arrow[r, "\mathsf{FFT}_{N/2}^{-1}"] & \mathbf{h}'' \arrow[r, "\text{untwist}"] & \mathbf{h}' \arrow[r, "\text{unfold}"] & \bar{\mathbf{h}} = \mathbf{f} \bar{*} \mathbf{g}. \\ \mathbf{g} \arrow[r, "\text{fold}"] & \mathbf{g}' \arrow[r, "\text{twist}"] & \mathbf{g}'' \arrow[r, "\mathsf{FFT}_{N/2}"] & \bar{\mathbf{G}} & & & & \end{tikzcd} \qquad (2.31)$$

**Proposition 2.4.** *Let* $\mathbf{f}, \mathbf{g} \in \mathbb{R}^N$*, where* $N = 2^\nu$ *for some* $\nu \in \mathbb{N}$*,* $\|\mathbf{f}\|_\infty \leq 2^{\varphi_0}$ *and* $\|\mathbf{g}\|_\infty \leq 2^{\gamma_0}$ *for some* $\varphi_0, \gamma_0 \in \mathbb{N}$*, and let* $\chi$ *denote the bit-precision of* $\omega$*'s as well as all intermediate values during the calculation of* $\mathsf{FFT}_N(\cdot)$ *and its inverse, represented as a floating point type. We denote* $\mathbf{h} := \mathsf{FFT}_N^{-1}\big(\mathsf{FFT}_N(\mathbf{f}) \odot \mathsf{FFT}_N(\mathbf{g})\big)$ *and* $\bar{\mathbf{h}} := \mathsf{FFNT}_N^{-1}\big(\mathsf{FFNT}_N(\mathbf{f}) \odot \mathsf{FFNT}_N(\mathbf{g})\big)$*, while we consider the errors as* $\|\mathsf{Err}(\mathbf{h})\|_\infty = \|\mathbf{h} - \mathbf{f} * \mathbf{g}\|_\infty$ *and* $\|\mathsf{Err}(\bar{\mathbf{h}})\|_\infty = \|\bar{\mathbf{h}} - \mathbf{f} \bar{*} \mathbf{g}\|_\infty$*, respectively. Then*

$$\begin{aligned} \log\|\mathsf{Err}(\mathbf{h})\|_\infty &\lessapprox (2\nu-2) \cdot \log\big(\sqrt{2}+1\big) + \varphi_0 + \gamma_0 - \chi + 4, && (2.32) \\ \log \mathsf{Var}\big(\mathsf{Err}(\mathbf{h})\big) &\lessapprox 4\nu + 2\varphi_0 + 2\gamma_0 - 2\chi - 1 - \log(3), \quad \textit{and} && (2.33) \\ \log\|\mathsf{Err}(\bar{\mathbf{h}})\|_\infty &\lessapprox (2\nu-4) \cdot \log\big(\sqrt{2}+1\big) + \varphi_0 + \gamma_0 - \chi + 4 + \log(3) + {}^1\!/_2, && (2.34) \\ \log \mathsf{Var}\big(\mathsf{Err}(\bar{\mathbf{h}})\big) &\lessapprox 4\nu + 2\varphi_0 + 2\gamma_0 - 2\chi - 3. && (2.35) \end{aligned}$$

*Proof.* Find the proof in Appendix A. □

We apply our estimates of the error and variance bounds in order to derive two basic parameter setups for convolution over integers: an error-free setup and a setup with rare errors based on the $3\sigma$-rule; see the following corollary.

**Corollary 2.5.** *Provided that*

$$\chi_0^{(c.)} \gtrsim \underbrace{2\log\bigl(\sqrt{2}+1\bigr)}_{\approx 2.54} \cdot \nu + \varphi_0 + \gamma_0 + \underbrace{5 - 2\log\bigl(\sqrt{2}+1\bigr)}_{\approx 2.46}, \quad \text{or} \tag{2.36}$$

$$\chi_0^{(nc.)} \gtrsim \underbrace{2\log\bigl(\sqrt{2}+1\bigr)}_{\approx 2.54} \cdot \nu + \varphi_0 + \gamma_0 + \underbrace{5 + \log(3) + {}^1\!/_2 - 4\log\bigl(\sqrt{2}+1\bigr)}_{\approx 2.00}, \tag{2.37}$$

*we have $\|\mathsf{Err}(\mathbf{h})\|_\infty \lesssim 1/2$, or $\|\mathsf{Err}(\bar{\mathbf{h}})\|_\infty \lesssim 1/2$, which means an error-free cyclic, or negacyclic convolution on integers via $\mathsf{FFT}_N$, or $\mathsf{FFNT}_N$, respectively. I.e., for $\mathbf{f}, \mathbf{g} \in \mathbb{Z}^N$, we have*

$$\Bigl\lfloor \mathsf{FFT}_N^{-1}\bigl(\mathsf{FFT}_N(\mathbf{f}) \odot \mathsf{FFT}_N(\mathbf{g})\bigr) \Bigr\rceil = \mathbf{f} * \mathbf{g}, \quad \text{or} \tag{2.38}$$

$$\Bigl\lfloor \mathsf{FFNT}_N^{-1}\bigl(\mathsf{FFNT}_N(\mathbf{f}) \odot \mathsf{FFNT}_N(\mathbf{g})\bigr) \Bigr\rceil = \mathbf{f} \,\bar{*}\, \mathbf{g}, \tag{2.39}$$

*respectively, up to negligible probability.*

*Next, if*

$$\chi_{3\sigma}^{(c.)} \gtrsim 2\nu + \varphi_0 + \gamma_0 + \underbrace{{}^1\!/_2 \log(6)}_{\approx 1.29}, \quad \text{or} \tag{2.40}$$

$$\chi_{3\sigma}^{(nc.)} \gtrsim 2\nu + \varphi_0 + \gamma_0 + \underbrace{\log(3) - {}^1\!/_2}_{\approx 1.08}, \tag{2.41}$$

*we have $3\sqrt{\mathsf{Var}\bigl(\mathsf{Err}(\mathbf{h})\bigr)} \lesssim 1/2$, or $3\sqrt{\mathsf{Var}\bigl(\mathsf{Err}(\bar{\mathbf{h}})\bigr)} \lesssim 1/2$, which estimates the required floating point type precision for the respective convolution variant based on the $3\sigma$-rule.*

*Note* 2.5. In the most common practical setting with the `binary64` type as per IEEE 754 standard [1] (aka. `double`), we have $\chi = 53$ bits of precision. For the 80-bit variant of the extended precision format (aka. `long double`), we have $\chi = 64$ bits of precision.

## 2.5 Implementation & Experimental Results

In this section, we briefly comment on how we use the data paths in our implementation (as outlined in Note 2.3), we discuss the choice of parameters with respect to TFHE, and then we focus on the following:

1. benchmarking with other implementations using chosen parameters,
2. performance on long polynomials using both 64-bit `double` and 80-bit `long double` floating point number representations, and
3. error magnitude and correctness of the results.

### Implementation Remarks

In our implementation of the Cooley-Tukey data path [43], we adapted the 4-vector approach from the Nayuki Project [103], which optimizes the RAM access for the most common 64-bit architectures. In a similar manner, we implemented the Gentleman-Sande data path [57]. To calculate FFT properly, both data paths require a specific reordering of their input or output, respectively. The reordering is based on bit-reversal of position indexes, counting from 0. E.g., for 16 elements (4 bits), we exchange the elements at positions $5 \leftrightarrow 10$, since $5 =$ `0b0101` and $10 =$ `0b1010`.

Since our goal is solely convolution, i.e., we do not care about the exact order of the FFT coefficients, the bit-reverse reordering can be omitted, as pointed out by Crandall and Pomerance [45]. By construction, it follows that the Gentleman-Sande data path must be used for the direct transformation and the Cooley-Tukey data path for the inverse.

For benchmarking purposes, we also adopted some code from the TFHE Library [124] to compare the redundant and non-redundant approaches; cf. Sections 2.3.1 and 2.3.2, respectively.

**Relation to the TFHE Parameters**

The main (cryptographic) motivation of our algorithm for negacyclic convolution over integers is the negacyclic polynomial multiplication in the TFHE scheme [35]. Below we outline a relation of the TFHE parameters to the parameters of negacyclic convolution via FFNT. As a result, we suggest a reasonable parameter setup for benchmarking.

In TFHE, negacyclic polynomial multiplication occurs in the bootstrapping procedure (namely, in the calculation of the external product), where an integer polynomial is multiplied by a torus polynomial. The coefficients of the right-hand side (torus) polynomial can be represented as integers scaled to $[0, 1)$ and bounded by 2 to the power of their bit-precision, denoted by $\tau$. In the left-hand side (integer) polynomial, the coefficients are bounded by $2^\gamma$, where $\gamma$ is one of the fundamental TFHE parameters. By construction, the parameter $\gamma$ is smaller than $\tau$, namely, $\gamma \leq \tau/l$, where $l$ is another TFHE parameter. In a corner case, it can be $\gamma = 1$ and the bound can be hence as low as $2^0$.

Based on our preliminary calculations for multivalue TFHE, we need the degree of TFHE polynomials to be at least $N = 2^{14}$ for 8-bit plaintexts with 128-bit security, and the torus precision to be at least $\tau = 34$ (both can be smaller for shorter plaintexts). Finally, we suggest to run the tests using polynomials with $\varphi_0 = \gamma_0 = \tau/2 = 17$ and $N = 2^{10}, \ldots, 2^{14}$.

### 2.5.1 Benchmarking Results

As a reference for benchmarking of our implementation [80] of negacyclic convolution, we have chosen the NTL Library [120] and the redundant method (as used in the original TFHE Library [124]; cf. Section 2.3.1), for which we used the same implementation of FFT as for our non-redundant method. Note that the implementation by Al Badawi et al. [5] shows similar results to the popular NTL (only about 1.01–1.2× faster) and they also show that NTL is faster than the concurrent FLINT Library [69]. For NTL, we tested both `ZZ_pX` and `ZZ_pE` classes, while the latter shows slightly better performance, hence we used that for benchmarking. Find the results of our benchmarks in Table 2.1.

*Note* 2.6. During the parameter setup, we silently passed over the fact that $\chi = 53$ (bit-precision of `double`) is lower than our $3\sigma$-rule estimates for all tested $\nu$'s, as per (2.41) in Corollary 2.5. Indeed, they dictate $\chi_{3\sigma}^{(nc.)} \gtrsim 2\nu + \varphi_0 + \gamma_0 + 1.08 = 55.08 \ldots 63.08$. For this reason, we reran the scenario with $\nu = 14$ for 1 000-times, we checked the results for correctness, and we did not detect *any* error across all tested polynomials.

### 2.5.2 Performance on Long Polynomials

As a reference for other prospective applications of our method, we tested our code on longer polynomials, too. We provide the performance results using both 64-bit `double` and 80-bit `long double` in Figure 2.1.

| Degree ($N$) | $2^{10}$ | $2^{11}$ | $2^{12}$ | $2^{13}$ | $2^{14}$ |
|---|---|---|---|---|---|
| NTL [ms] | 0.617 | 1.258 | 2.643 | 6.132 | 12.771 |
| $\mathsf{FFT}_{2N}$ [ms] | 0.122 | 0.230 | 0.458 | 0.982 | 2.277 |
| $\mathsf{FFNT}_N$ [ms] | 0.036 | 0.069 | 0.120 | 0.243 | 0.541 |
| $\mathsf{FFNT}_N$ over $\mathsf{FFT}_{2N}$ | 3.35× | 3.33× | 3.82× | 4.04× | 4.21× |
| $\mathsf{FFNT}_N$ avg. error [‰] | 0.06 | 0.08 | 0.12 | 0.18 | 0.27 |
| $\mathsf{FFNT}_N$ max. error [‰] | 0.37 | 0.55 | 0.98 | 1.47 | 1.95 |

Table 2.1: Mean time per negacyclic multiplication of uniformly random polynomials with $\|p\|_\infty \leq 2^{17}$ using NTL (similar times as FLINT), $\mathsf{FFT}_{2N}$ on negacyclic extension (implemented in [124]), and $\mathsf{FFNT}_N$, both using 64-bit `double`. Speedup of $\mathsf{FFNT}_N$ over $\mathsf{FFT}_{2N}$. Average and maximum rounding errors of $\mathsf{FFNT}_N$. 1 000 runs per degree and method on an Intel Core i7-8550U CPU @ 1.80GHz.

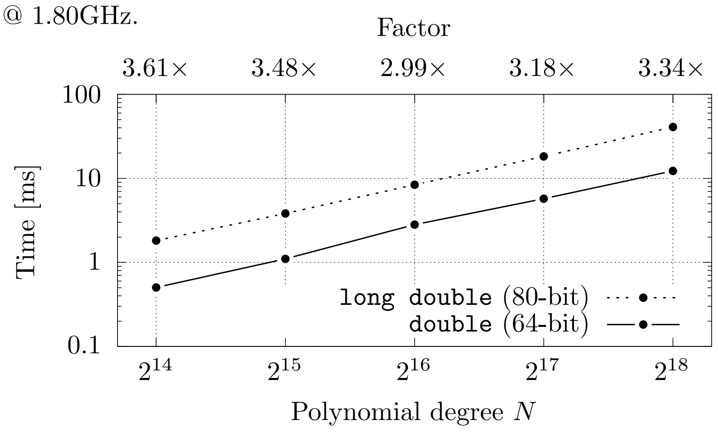


Figure 2.1: Mean time per polynomial multiplication $\bmod X^N + 1$ and speedup factor of `double` over `long double`. Uniformly random polynomials with $\|p\|_\infty \leq 2^{17}$, 1 000 measurements.

### 2.5.3 Error Magnitude & Correctness on Long Polynomials

As outlined in Note 2.6, our experimental setup exceeds the derived theoretical bounds, even for lower-degree polynomials. Hence, our next goal is to evaluate the error magnitude as well as to check the correctness of the results. We tested the following input polynomial scenarios:

1. uniformly random coefficients (bounded by $\|p\|_\infty \leq 2^{\varphi_0}$), and
2. all coefficients equal to the bound minus one, i.e., $2^{\varphi_0} - 1$.

Find the results of the random polynomial setup in Figure 2.2, where we tested both 64-bit `double` and 80-bit `long double` implementations.

Regarding the setup with all coefficients equal to the bound, we ran the same scenarios as for random polynomials (cf. Figure 2.2). With 64-bit `double`, the only correct results were obtained for the setup with $\|p\|_\infty \leq 2^{17}$ and $N = 2^{14}$, or $N = 2^{15}$, respectively. With 80-bit `long double`, all scenarios were calculated correctly, with maximum rounding error $\lesssim 0.109$ for $\|p\|_\infty \leq 2^{20}$ and $N = 2^{18}$.

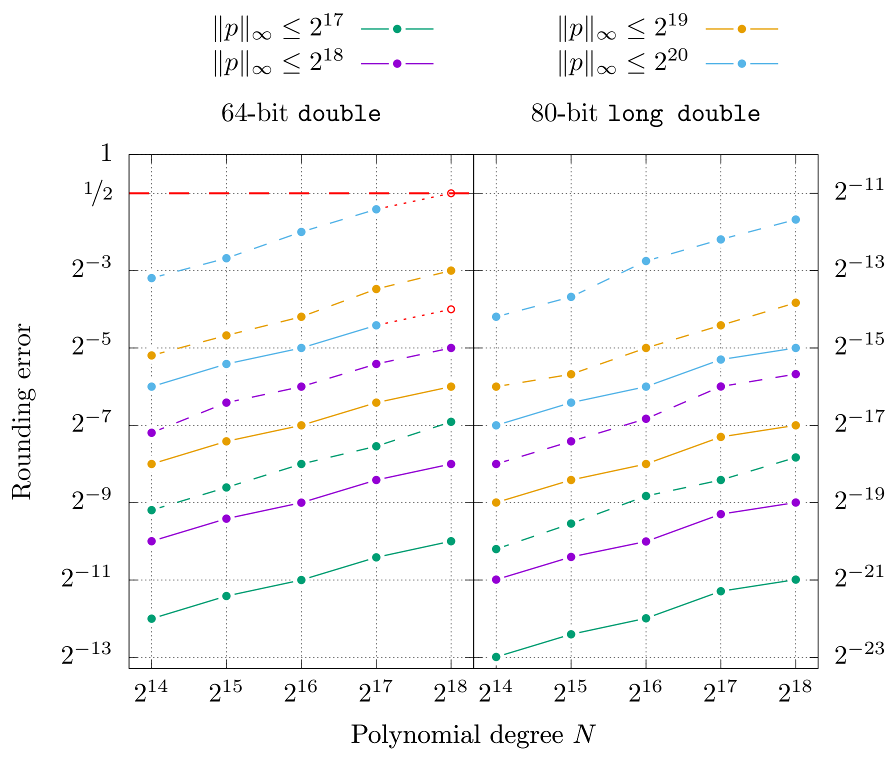


Figure 2.2: Median (solid) and Maximum (dashed) rounding errors for uniformly random polynomials. Erroneous results emphasized by empty red circles. 10 measurements per degree, bound and floating point type.

### Discussion

We observed a factor $\sim 4\times$ speedup of $\mathsf{FFNT}_N$ (i.e., the non-redundant approach) over $\mathsf{FFT}_{2N}$ (i.e., the redundant approach). Compared to NTL, which calculates the coefficients precisely using a number-theoretic transform, our FFT-based method shows by more than an order of magnitude better results. Even though we ran our tests with underestimated precision, we obtained correct results for much larger polynomials with uniformly random coefficients. Note that random-like polynomials occur in TFHE, hence our benchmarking scenario with random polynomials is representative for the usage with TFHE.

In addition, we tested our code with the 80-bit `long double` floating point type. It enabled error-free calculations with polynomials of higher degree and/or with greater coefficient bound, yet it was only about 3–4 times slower than the variant with the 64-bit `double`.

## 2.6 Conclusion

We showed that FFT-based convolution algorithms can significantly outperform similar algorithms based on number-theoretic transforms, and they can still guarantee error-free results in the integer domain. We derived estimates of the lower bound of the employed floating point type for error-free cyclic and negacyclic convolutions, as well as we suggested the bounds based on the $3\sigma$-rule.

We suggested a set of testing parameters for negacyclic convolution with particular respect to the usage with the TFHE Scheme on a multivalue plaintext space. We ran a benchmark that compares the popular NTL Library, the approach that is used in the TFHE Library, and our approach. Compared to the generic NTL Library, which employs a number-theoretic transform, and to the TFHE Library approach, which calculates redundant intermediate values, we achieved a speedup of around $24\times$ and $4\times$, respectively.

Finally, our experiments have shown approximate bounds for practical error-free results. Namely, using `double`, we could multiply polynomials without errors up to degree $N = 2^{17}$ and norm $\|p\|_\infty \leq 2^{20}$ with uniformly random coefficients, and up to degree $N = 2^{15}$ with coefficients equal to $2^{17}$. To conclude, we find our approach particularly useful for negacyclic integer polynomial multiplication, not only in TFHE.

### Future Directions

Our aim is to implement a version based on the 64-bit signed integer type instead of `double`, where we would keep the exponent at one place for the entire array. Such an approach requires less demanding arithmetics and it would serve as a proof-of-concept for a propective FPGA implementation.

### Acknowledgments

We would like to thank Ahmad Al Badawi and Sergiu Carpov for useful comments and remarks.

## Appendix

## A Proof of Proposition 2.4

*Proof.* Let us begin with the cyclic convolution. By (2.30) and Lemma 2.2 and 2.3, we have

$$\begin{aligned}
\|\mathsf{Err}(\mathbf{F} \odot \mathbf{G})\|_\infty &\lessapprox \Big(\underbrace{c_H^{(\mathbf{f})} \cdot \big(\sqrt{2}+1\big)^\nu}_{\gtrapprox \|\mathsf{Err}(\mathbf{F})\|_\infty} \cdot \underbrace{2^{\gamma_0+\nu}}_{\geq \|\mathbf{G}\|_\infty} + c_H^{(\mathbf{g})} \cdot \big(\sqrt{2}+1\big)^\nu \cdot 2^{\varphi_0+\nu}\Big) \cdot \sqrt{2} = \\
&= \big(\sqrt{2}+1\big)^\nu \cdot 2^{\nu+\varphi_0+\gamma_0-\chi+2} \cdot \big(2-\sqrt{2}\big) \cdot \sqrt{2} =: E_\mathbf{H}, \quad \text{and} \qquad (42) \\
\mathsf{Var}\big(\mathsf{Err}(\mathbf{F} \odot \mathbf{G})\big) &\lessapprox \Big(\underbrace{d_N^{(\mathbf{f})} \cdot 2^{2\nu}}_{\gtrapprox \mathsf{Var}(\mathsf{Err}(\mathbf{F}))} \cdot \underbrace{2^{2\gamma_0+2\nu}}_{\geq \|\mathbf{G}\|_\infty^2} + d_N^{(\mathbf{g})} \cdot 2^{2\nu} \cdot 2^{2\varphi_0+2\nu}\Big) \cdot 2 = \\
&= {}^2\!/_3 \cdot 2^{4\nu+2\varphi_0+2\gamma_0-2\chi} =: V_\mathbf{H}, \qquad (43)
\end{aligned}$$

which we apply as the initial error and variance bound to (2.22) and (2.23), respectively, together with multiplication by ${}^1\!/_N = 2^{-\nu}$, which poses the only difference between $\mathsf{FFT}^{-1}$ and $\mathsf{FFT}$ from the error point of view. We neglect other than leading terms and we get

$$\begin{aligned} \|\mathsf{Err}(\mathbf{h})\|_\infty &\lessapprox 2^{-\nu} \cdot \underbrace{2(\sqrt{2}-1) \cdot E_{\mathbf{H}}}_{\approx c_H^{(\mathbf{H})}} \cdot \big(\sqrt{2}+1\big)^{\nu} \lessapprox \\ &\lessapprox \big(\sqrt{2}+1\big)^{2\nu-2} \cdot 2^{\varphi_0+\gamma_0-\chi+4}, \quad \text{and} & (44) \\ \mathsf{Var}\big(\mathsf{Err}(\mathbf{h})\big) &\lessapprox 2^{-2\nu} \cdot \underbrace{1/6 \cdot 2^{2(\varphi_0+\gamma_0+2\nu)-2\chi}}_{=d_N^{(\mathbf{H})}} \cdot 4^{\nu} = 1/6 \cdot 2^{4\nu+2\varphi_0+2\gamma_0-2\chi}, & (45) \end{aligned}$$

and the cyclic results follow.

For the negacyclic convolution, we feed DFT with a folded and twisted input vector; cf. (2.31). It enters DFT with error bounded as

$$\|\mathsf{Err}(\mathbf{f}'')\|_\infty \lessapprox (1 \cdot 0 + 2^{\varphi_0+1/2} \cdot 2^{-\chi-1}) \cdot \sqrt{2} = 2^{\varphi_0-\chi}. \tag{46}$$

Regarding variance, it shows that the term with $\mathsf{Var}\big(\mathsf{Err}(\mathbf{f}'')\big)$ will be neglected. Next, we precompute

$$\begin{aligned} c_H^{(\mathbf{f}'')} &= 2(\sqrt{2}-1) \cdot \|\mathsf{Err}(\mathbf{f}'')\|_\infty + (2-\sqrt{2}) \cdot 2^{\varphi_0+1/2-\chi+1} \lessapprox \\ &\lessapprox 6(\sqrt{2}-1) \cdot 2^{\varphi_0-\chi}, \quad \text{and} & (47) \\ d_N^{(\mathbf{f}'')} &= 1/6\, 2^{2(\varphi_0+1/2)-2\chi}, & (48) \end{aligned}$$

and apply into

$$\begin{aligned} \|\mathsf{Err}(\bar{\mathbf{F}} \odot \bar{\mathbf{G}})\|_\infty &\lessapprox \Big(\underbrace{c_H^{(\mathbf{f}'')} \cdot \big(\sqrt{2}+1\big)^{\nu-1}}_{\gtrapprox \|\mathsf{Err}(\bar{\mathbf{F}})\|_\infty} \cdot \underbrace{2^{\gamma_0+1/2+\nu-1}}_{\geq \|\bar{\mathbf{G}}\|_\infty} + \\ &\quad + c_H^{(\mathbf{g}'')} \cdot \big(\sqrt{2}+1\big)^{\nu-1} \cdot 2^{\varphi_0+1/2+\nu-1}\Big) \cdot \sqrt{2} = \\ &= 3\big(\sqrt{2}+1\big)^{\nu-2} \cdot 2^{\nu+\varphi_0+\gamma_0-\chi+2} =: E_{\bar{\mathbf{H}}}, \quad \text{and} & (49) \\ \mathsf{Var}\big(\mathsf{Err}(\bar{\mathbf{F}} \odot \bar{\mathbf{G}})\big) &\lessapprox \Big(\underbrace{d_N^{(\mathbf{f}'')} \cdot 4^{\nu-1}}_{\gtrapprox \mathsf{Var}(\mathsf{Err}(\bar{\mathbf{F}}))} \cdot \underbrace{2^{2\gamma_0+1+2\nu-2}}_{\geq \|\bar{\mathbf{G}}\|_\infty^2} + \\ &\quad + d_N^{(\mathbf{g}'')} \cdot 4^{\nu-1} \cdot 2^{2\varphi_0+1+2\nu-2}\Big) \cdot 2 = \\ &= 1/3 \cdot 2^{4\nu+2\varphi_0+2\gamma_0-2\chi-1} =: V_{\bar{\mathbf{H}}}. & (50) \end{aligned}$$

Next, we apply these estimates as the initial error and variance bound into (2.22) and (2.23), respectively, together with multiplication by $2/N = 2^{-\nu+1}$. We have

$$\begin{aligned} \|\mathsf{Err}(\mathbf{h}'')\|_\infty &\lessapprox 2^{-\nu+1} \cdot \underbrace{2(\sqrt{2}-1) \cdot E_{\bar{\mathbf{H}}}}_{\approx c_H^{(\bar{\mathbf{H}})}} \cdot \big(\sqrt{2}+1\big)^{\nu-1} \approx \\ &\approx 3\big(\sqrt{2}+1\big)^{2\nu-4} \cdot 2^{\varphi_0+\gamma_0-\chi+4}, \quad \text{and} & (51) \\ \mathsf{Var}\big(\mathsf{Err}(\mathbf{h}'')\big) &\lessapprox 2^{-2\nu+2} \cdot \underbrace{1/6 \cdot 2^{(2\varphi_0+2\gamma_0+2+4\nu-4)-2\chi}}_{=d_N^{(\bar{\mathbf{H}})}} \cdot 4^{\nu-1} = \\ &= 1/3 \cdot 2^{4\nu+2\varphi_0+2\gamma_0-2\chi-3}, & (52) \end{aligned}$$

while in (52), it has shown that the term with $V_{\bar{\mathbf{H}}}$ was not the leading term, hence it was neglected. By (2.31) it remains to untwist and unfold, we have

$$\begin{aligned} \|\mathsf{Err}(\mathbf{h}')\|_\infty &\lessapprox \big(1 \cdot \underbrace{3(\sqrt{2}+1)^{2\nu-4} \cdot 2^{\varphi_0+\gamma_0-\chi+4}}_{\gtrapprox \|\mathsf{Err}(\mathbf{h}'')\|_\infty} + \underbrace{2^{2\nu+\varphi_0+\gamma_0-1}}_{\geq \|\mathbf{h}''\|_\infty} \cdot 2^{-\chi-1}\big) \cdot \sqrt{2} \approx \\ &\approx 3\sqrt{2} \cdot (\sqrt{2}+1)^{2\nu-4} \cdot 2^{\varphi_0+\gamma_0-\chi+4}, \quad \text{and} \qquad (53) \\ \mathsf{Var}\big(\mathsf{Err}(\mathbf{h}')\big) &\lessapprox (1^2 \cdot \underbrace{1/3 \cdot 2^{4\nu+2\varphi_0+2\gamma_0-2\chi-3}}_{\gtrapprox \mathsf{Var}\left(\mathsf{Err}(\mathbf{h}'')\right)} + \underbrace{2^{4\nu+2\varphi_0+2\gamma_0-2}}_{\geq \|\mathbf{h}''\|_\infty^2} \cdot 1/12 \cdot 2^{-2\chi}) \cdot 2 = \\ &= 2^{4\nu+2\varphi_0+2\gamma_0-2\chi-3}. \qquad (54) \end{aligned}$$

Since the unfolding operation does not change the error, the negacyclic results follow. □

# Chapter 3

# PARMESAN: Parallel ARithMEticS over ENcrypted data

*Fully Homomorphic Encryption* enables the evaluation of an arbitrary computable function over encrypted data. Among all such functions, particular interest goes for integer arithmetics. In this chapter, we present a bundle of methods for fast arithmetic operations over encrypted data: addition/subtraction, multiplication, and some of their special cases. On top of that, we propose techniques for signum, maximum, and rounding. All methods are specifically tailored for computations with data encrypted with the TFHE scheme (Chillotti *et al.*, Asiacrypt '16) and we mainly focus on parallelization of non-linear homomorphic operations, which are the most expensive ones. This way, evaluation times can be reduced significantly, provided that sufficient parallel resources are available. We implement all presented methods in the *Parmesan Library* and we provide an experimental evaluation. Compared to integer arithmetics of the *Concrete Library*, we achieve considerable speedups for all comparable operations. Major speedups are achieved for the multiplication of an encrypted integer by a cleartext one, where we employ special addition-subtraction chains, which save a vast amount of homomorphic operations.

## 3.1 Introduction

The idea of *Fully Homomorphic Encryption* (FHE), which allows for arbitrary computations over encrypted data, was first proposed by Rivest et al. [115] back in 1978. However, the question of whether such a scheme exists remained open for more than 30 years until 2009 when Gentry [58] gave a positive answer. Although resolved from the mathematical point of view, initial FHE schemes suffered from fairly low efficiency. Since then, the performance of FHE is being constantly improved, either through theoretical advances [60, 23, 33, 61, 50, 35] or with emerging attempts to develop a dedicated hardware [56, 130].

FHE schemes typically allow the evaluation of *addition* and a *non-linear operation* over encrypted data. For addition, this means that there exists operation $\oplus$ over ciphertexts, while for any pair of plaintexts $x$, $y$, it holds

$$\mathrm{FHE}.\mathsf{Encr}(x) \oplus \mathrm{FHE}.\mathsf{Encr}(y) \approx \mathrm{FHE}.\mathsf{Encr}(x+y), \tag{3.1}$$

where $\approx$ means "with high probability, encrypts the same". I.e., FHE.Encr is a plaintext $\rightarrow$ ciphertext space additive group homomorphism, up to a randomization of FHE.Encr and up to a certain (small) probability of error. The other, non-linear operation can be, e.g., multiplication or *Look-Up Table* (LUT) evaluation.

In principle, FHE enables evaluation of any computable function over encrypted data, e.g., by its decomposition to boolean gates, which may not be very efficient though. For a smooth practical deployment of FHE, we believe that it is important to develop *optimized* homomorphic variants of most common operations, with *basic integer arithmetic* at the first place. Indeed, arithmetic is a fundamental part of most CPUs' instruction sets and integers are one of the primary data types. Since current FHE schemes have a fairly limited plaintext space size, which can only be increased at an unfavorable cost, we build operations upon smaller blocks of data.

In this chapter, we put forward tailored and optimized methods for the homomorphic evaluation of basic arithmetic. In addition, we propose homomorphic variants of some other common operations. Our methods are built on top of a particular digit-based integer representation, encrypted with the TFHE Scheme by Chillotti et al. [35]. The TFHE scheme enables a limited number of (very fast) linear operations – these need to be interlaced with another operation referred to as *bootstrapping*, which:

- takes much more time to evaluate (currently tens of milliseconds; depends on parameters),
- is inherently capable of evaluating a custom LUT homomorphically, and
- enables evaluation of circuits of arbitrary depth.

Compared to other FHE schemes, bootstrapping of TFHE is among the fastest, which is the main reason why we choose TFHE. Nevertheless, in our algorithms, the underlying FHE scheme can be easily replaced with another LUT-based FHE scheme, if that scheme shows to be more efficient.

### Related Work

Many current use-cases of FHE[1] focus on a single-purpose application, where FHE operations are specifically optimized for this purpose. In particular, there is a lot of interest in cloud-assisted neural network (NN) inference [62, 20, 36], which typically requires expert-level knowledge of both FHE and NN's.

On the other hand, there exists a line of research on *homomorphic compilers*, summarized in [129], which aims at simplifying the homomorphization effort for ordinary developers. Contributions range from a general-purpose transpiler [64] (translates arithmetic operations into many boolean gates, "making them quite slow"), through an approximate-arithmetic-based compiler EVA [47, 39] (whereas we aim at precise arithmetic), to higher-level code optimizations [128].

A scheme known as CKKS [33] (employed in the EVA compiler) enables approximate arithmetic, which is particularly useful in machine learning tasks. However, due to its approximate nature, only a limited precision can be considered correct, therefore, it does not compare directly to our approach. Although there exists a bootstrapped variant of CKKS [30, 26], we are not aware of any implementation of multi-precision arithmetic based on CKKS.

An approach that covers precise integer arithmetics with arbitrary bit-lengths is proposed in [37], further developed in [13], and implemented as part of the *Concrete Library* [42]. Based on a multitude of previous works on homomorphic integers, authors of [13] provide a thorough comparison of state-of-the-art techniques, though mainly focusing on low-level optimizations

[1] An updated list of FHE applications can be found at https://fhe.org/fhe-use-cases.

of bootstrapping and also on finding the best TFHE parameters for selected approaches. For homomorphic arithmetics, they suggest extending the message space by a couple of bits to accommodate the (additive) carry, which allows to evaluate a limited number of additions without the need for bootstrapping. As soon as the carry bits need to propagate, they employ a standard (schoolbook) sequential approach. Authors also propose a parallelizable approach based on the *Chinese Remainder Theorem* (CRT); however, due to its specificities, we do not compare with it.

In our recent study [87], we compare selected sequential and parallel addition algorithms over TFHE-encrypted data: among three sequential and six parallel approaches, we identify the fastest parallel approach, which outperforms the fastest sequential approach starting from 5-bit addends. Since the parallel approach requires a non-standard integer representation, we also demonstrate that other operations like signum and maximum are possible.

Besides integer arithmetics, another important operation is indexing an (encrypted) array with an encrypted index, i.e., evaluation of a big LUT. Two approaches are proposed by Guimarães et al. [66], also studied in [13].

## Our Contributions

We propose, implement and evaluate a digit-based integer arithmetic over TFHE-encrypted data, with a particular focus on parallelization, so that the evaluation time is reduced as much as possible. Our methods are based on an algorithm for parallel addition, which we select based on a thorough comparison given in [87]. The list of arithmetic operations includes:

- *Addition/Subtraction:* a basic operation, upon which other operations are built (the underlying algorithm is determined based on the results of [87]). We further identify bootstrap operations that can be saved.
- *Scalar multiplication:* a special case of multiplication, where one integer is unencrypted (demonstrated in [87]). We define a new, presumably hard, computational problem, which is tied with optimization of the number of additions that are called within scalar multiplication (a special type of addition-subtraction chain). Inspired by an approach used in Elliptic Curve Cryptography, we propose a heuristic solution, within which we evaluate small instances of the computational problem, achieving an average improvement of about 20% compared to [87].
- *Multiplication:* the most demanding operation, for which we suggest employing the Karatsuba algorithm to optimize the number of digit-by-digit multiplications, and where we also call the parallel addition algorithm. We discuss and evaluate several aspects of this approach so that the best performance is achieved.
- *Squaring:* a special case of multiplication, where the input is duplicated. We show that a dedicated algorithm for squaring achieves about 30% improvements over multiplication. In addition, we propose a very efficient squaring method for (up to) 3-bit inputs.

We also investigate and optimize other useful operations:

- *Signum:* a fundamental operation for number comparison and other operations (demonstrated in [87]). Compared to [87], we reduce the circuit depth by one, which reduces the number of bootstraps and threads significantly.
- *Maximum:* gives the greater of two encrypted integers (demonstrated in [87]). Not only maximum is improved by the faster signum, but we also propose a new way of evaluation, which only needs half of the threads.

- *Rounding:* rounds an encrypted integer at a given bit-position. The rounding algorithm is non-trivial in the integer representation used by parallel addition.

We accompany each operation with a brief analysis, where we list its requirements (message space size, the ideal number of threads, etc.).

In the experimental part, we present our implementation (in a form of a library) and we compare it with the Concrete Library [42]. Our benchmarks show that for 32-bit encrypted integers, our library achieves speed-ups over Concrete ranging from 1.9× for multiplication on an ordinary 12-threaded server processor, through 7.0× for squaring on a 128-threaded supercomputer's node, to tens of times (and more) for scalar multiplication with selected inputs.

### Chapter Outline

In Section 4.2, we recall the TFHE scheme and its supported homomorphic operations: addition and LUT evaluation. We also recall a particular algorithm for parallel addition and its specifics. Next, in Section 3.3, we revisit and/or suggest new algorithms for basic arithmetic operations, which are suitable for homomorphic evaluation with TFHE, with a particular focus on their parallelization. In Section 3.4, we revisit and/or suggest other algorithms for comparison-based integer operations: signum, maximum and, rounding. We introduce our implementation and we provide and discuss the results of our benchmarks in Section 3.5. We conclude this chapter in Section 4.6.

## 3.2 Preliminaries

For reference, we first provide a summary of symbols & notation that we use throughout this chapter. Then, we recall the TFHE scheme and its homomorphic operations, in particular, we focus on LUT evaluation. Finally, we discuss integer representations and we recall a selected algorithm for parallel addition.

### Symbols & Notation

$\mathbb{N}, \mathbb{N}_0$ **. . .** positive and non-negative integers, i.e., $\{1, 2, 3, \dots\}$ and $\{0, 1, 2, \dots\}$,

$\mathbb{Z}$ **. . .** the ring of integers,

$\mathbb{R}$, $\mathbb{R}_0^+$ **. . .** real numbers and non-negative real numbers,

$\mathbb{T}$ **. . .** the real torus: $\mathbb{R}/\mathbb{Z}$, i.e., reals modulo 1,

$[a, b)$ **. . .** interval of reals or integers, which contains $a$ and does not contain $b$,

$x \gtreqless \pm b$ **. . .** comparison of $x$ with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (3.3),

$x \equiv \pm b$ **. . .** comparison of $x$ with $\pm b$, $b \in \mathbb{N}$, it outputs $\{-1, 0, +1\}$ as per (3.4),

LUT **. . .** Look-Up Table,

$(a, b, \circ, c \,\|\, d)$ **. . .** notation for a custom negacyclic LUT (cf. Section 3.2.1),

$\pi$ **. . .** bit-length of the TFHE message space,

$2^{2\Delta}$ **. . .** the sum of squared weights (aka. *quadratic weight*; cf. Section 3.2.1),

$\mathbf{x} = (x_{n-1} \dots x_1 x_0 \bullet)_\beta$ **. . .** base-$\beta$ representation of $X = \sum_{i=0}^{n-1} \beta^i x_i$, where $x_i \in \mathbb{Z}$ and $\beta \in \mathbb{N}$, $\beta > 1$ (cf. Section 3.2.2),

$X = \mathsf{eval}_\beta(\mathbf{x})$ **. . .** evaluation of representation $\mathbf{x}$ in base $\beta$ as $X = \sum_{i=0}^{n-1} \beta^i x_i$,

$\bar{x}$ **. . .** negative digit, $\bar{x} = -x$, $x \in \mathbb{N}$; used in redundant number representations, e.g., $(1\bar{1}\bullet)_2 \sim 2 - 1 = 1$,

$\mathcal{A}_\beta$ **. . .** alphabet of the standard base-$\beta$ representation, $\mathcal{A}_\beta = \{0, 1, \dots, \beta - 1\}$,

$\bar{\mathcal{A}}_2$ **. . .** signed binary alphabet, $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$,

**MSB/LSB . . .** Most/Least Significant Bit,

$\mathsf{eval}(\mathsf{AC}_k, X)$ **. . .** evaluation of addition chain $\mathsf{AC}_k$ for integer $k$ and additive group element $X$ into $k \cdot X$ (cf. Section 3.3.2),

$\mathsf{ASC}^*$ **. . .** free-doubling addition-subtraction chain (cf. Section 3.3.2).

### 3.2.1 The TFHE Scheme & its Multi-Value Operations, Revisited

The TFHE scheme, proposed by Chillotti et al. [35], is thoroughly described in Chapter 1. For the purposes of this chapter, we recall and further comment on its operations, in particular for the multi-value variant.

Let $\mathbb{Z}_{2^\pi}$ be the desired message space – each message $m \in \mathbb{Z}_{2^\pi}$ can be represented with $\pi$ bits. Then, multi-value TFHE encodes message $m \in \mathbb{Z}_{2^\pi}$ into the TLWE plaintext space as $\mu = {}^{m}/_{2^\pi} \in \mathbb{T}$. The other way around, decoding handles the error by rounding, i.e., $m' = \lfloor ({}^{m}/_{2^\pi} + e) \cdot 2^\pi \rceil \in \mathbb{Z}_{2^\pi}$. Note that if $|e| < {}^{1}/_{2^{\pi+1}}$, then $m' = m$.

Combining the encoding of multi-value TFHE and the two homomorphic operations of plain TFHE (i.e., addition and negacyclic LUT evaluation), we obtain a set of homomorphic operations for the $\mathbb{Z}_{2^\pi}$ message space, denoted by $\mathcal{M}$:

- *Addition/Subtraction:* $\mathcal{M} + \mathcal{M} \to \mathcal{M}$ via vector addition/subtraction of TLWE samples;
- *Scalar multiplication:* $\mathbb{Z} \cdot \mathcal{M} \to \mathcal{M}$ via scalar-vector multiplication of a TLWE sample by an integer (equivalent to repeated additions/subtractions; sometimes we refer to both operations simply as *addition*); and
- *Negacyclic* LUT *evaluation:* $\mathsf{LUT}(\mathcal{M}) \to \mathcal{M}$ via TFHE bootstrapping.

#### Noise Growth during Addition

As outlined in Section 3.2.1, in the TLWE scheme, a certain amount of noise (error) must be added to the message, and the error term is additive with respect to homomorphic addition. Let us assume a set of fresh(ly bootstrapped) independent samples $\{\mathbf{c}_i\}$, with equal error variance $V_0$. Then, since error variance is additive with squares of weights, we quantify the error growth after additions using the sum of squared weights:

$$\mathsf{Var}\Big(\sum w_i \cdot \mathbf{c}_i\Big) = \underbrace{\sum w_i^2}_{2^{2\Delta}} \cdot \underbrace{\mathsf{Var}(\mathbf{c}_i)}_{V_0}, \tag{3.2}$$

where $w_i$'s are integer weights. We refer to $\sum w_i^2$ as the *quadratic weights* and we denote it by $2^{2\Delta}$. E.g., for independent samples $x, y, z$ with equal error variance, we have the quadratic weights of the sum $1 \cdot x - 3 \cdot y + 2 \cdot z$ equal to $2^{2\Delta} = 1^2 + 3^2 + 2^2 = 14$. Note that $\Delta$ itself is intended to express the additional bit-length of the noise's standard deviation.

### (Negacyclic) LUT Evaluation

First, we define a class of functions to make further notation concise and we propose an encoding of these functions into negacyclic LUTs. Then, we outline how additions can be used to evaluate some other, non-negacyclic LUTs. Finally, we introduce a notation for negacyclic LUTs, without explicitly stating them in full.

**Threshold Functions & Their Encoding into LUTs** Let $b \in \mathbb{N}$. We introduce the following functions:

$$f_b(x) = \begin{cases} -1 & \dots\ x \leq -b, \\ 0 & \dots\ -b < x < +b, \\ +1 & \dots\ +b \leq x, \end{cases} \tag{3.3}$$

$$g_b(x) = \begin{cases} -1 & \dots\ x = -b, \\ 0 & \dots\ x \neq \pm b, \\ +1 & \dots\ x = +b. \end{cases} \tag{3.4}$$

We use the notations $x \gtreqless \pm b$ and $x \equiv \pm b$ for $f_b(x)$ and $g_b(x)$, respectively.

Recall that LUTs in TFHE are inherently negacyclic, therefore, we need to deal with this limitation. As a usual workaround, an additional bit of padding is added. However, this effectively bloats the message space twice, which in turn induces less efficient TFHE parameters, hence we prefer to avoid that. Instead—as outlined in [86]—we exploit any possible overlap as much as possible, which may lead to message space savings, hence better bootstrapping times. Note that this kind of "overlap optimization" is specific to TFHE – it is also reflected in many of our algorithms, which are—in certain sense—tailored for TFHE.

To encode $x \gtreqless \pm b$ or $x \equiv \pm b$ on a desired domain $[-a, +a]$ (with $a \geq b > 0$) into a negacyclic LUT, the range $[-a-b, a+b)$ shows to be the minimal range for $x \gtreqless \pm b$ and a sufficient range for $x \equiv \pm b$ [87]. For $x \gtreqless \pm b$, we define negacyclic function $f$, $f\colon [-a-b, a+b) \to \{-1, 0, 1\}$, on the non-negative part of the domain as

$$f(x) = \begin{cases} 0 & x \in [0, b-1], \\ 1 & x \in [b, a], \\ 0 & x \in [a+1, a+b-1]. \end{cases} \tag{3.5}$$

Such function $f$ contains the function $x \gtreqless \pm b$ on the domain $[-a, +a]$ and the non-negative part of $f$ also serves as a prescription for respective LUT. An analogous approach applies to $x \equiv \pm b$. For more details, we refer to [87].

**Evaluating Some Other LUTs** Thanks to the cheap additive homomorphism, one may also shift the function by a constant, if this helps to find a negacyclic extension in a smaller domain; an example follows.

**Example 3.1.** *The function $f\colon \mathbb{Z}_4 \to \mathbb{Z}_4$, $f(0) = 1$, $f(1) = 2$, $f(2) = 1$, $f(3) = 0$, is not negacyclic, but $f(x) - 1$ is. Therefore, it is not needed to extend the domain to $\mathbb{Z}_8$ – it is sufficient to evaluate the negacyclic $f(x) - 1$ over $\mathbb{Z}_4$ and add $+1$ to the result.*

Moreover, in the plaintext domain of TFHE, i.e., in the torus $\mathbb{T}$, we are not limited to encoding integers – we may also encode fractions. This allows us to evaluate some other non-negacyclic functions; an example follows.

**Example 3.2.** *$f\colon \mathbb{Z}_4 \to \mathbb{Z}_4$, $f(0) = 0$, $f(1) = 0$, $f(2) = 1$, $f(3) = 1$ can be evaluated using a shift by $-1/2$, as outlined in Example 3.1.*

In case we consider the first half of $\mathbb{Z}_{2^\pi}$ as positive and the rest as negative (i.e., the standard signed integer representation in computer arithmetics), we may perceive the function from Example 3.2 as a *non-negativity function* over $\mathbb{Z}_4$. I.e., a function that outputs 1 or 0 if the input is non-negative or negative, respectively.

**Notation for Negacyclic LUTs** Let $f\colon (\mathbb{Z}_{2^\pi}) \to \mathbb{Z}_{2^\pi}$ be a negacyclic LUT and let $\mathbf{f} \in \mathbb{Z}_{2^\pi}^{2^{\pi-1}}$ be the list of its values in $[0, 2^{\pi-1})$; the rest of $f$ is given by its negacyclicity. For $x \in \mathbb{Z}_{2^\pi}$, referred to as the *selector*, we denote $f(x)$ by $\mathbf{f}[x]$, meaning that $x$ may exceed the index set of $\mathbf{f}$, i.e., the negacyclic extension *is considered*. In case there are some unused function values (i.e., outside of the domain of $f$), we use the symbol $\circ$, which can be set to, e.g., zeros in $\mathbf{f}$. Finally, in case the value of $\pi$ is not explicitly given, we use (at most once) the symbol $\|$ to denote the place to be filled with an appropriate number of $\circ$'s. In this case, we *do consider* the negacyclic extension in the suffix; an example follows.

**Example 3.3.** *Let $\pi = 3$, i.e., we have the message space $\mathcal{M} = \mathbb{Z}_8$. The list $(1, 2, \circ, -3)$ represents the negacyclic LUT given by $(1, 2, \circ, -3, -1, -2, \circ, 3)$, whereas the list $(1, 2 \,\|{-3})$ represents $(1, 2, \circ, 3, -1, -2, \circ, -3)$. With a selector $-1 = 7$ in $\mathbb{Z}_8$, they evaluate respectively as $(1, 2, \circ, -3)[-1] = 3$ and $(1, 2 \,\|{-3})[-1] = -3$.*

### 3.2.2 Parallel Arithmetics

The main focus of this chapter is the parallelization of arithmetic (and other) operations over encrypted integers. Many of these operations are based on an algorithm for parallel integer addition, which requires a non-standard integer representation. We recall one particular parallel addition algorithm, which we choose based on the results of [87].

#### Integer Representations

For *base* $\beta \in \mathbb{N}$, $\beta \geq 2$, and *alphabet* $\mathcal{A}_\beta = \{0, 1, \ldots, \beta-1\}$, we call $\mathbf{x} \in \mathcal{A}_\beta^n$, $\mathbf{x} = (x_{n-1} \ldots x_1 x_0 \bullet)_\beta$, the *standard base-$\beta$ representation of $X \in \mathbb{N}$* iff

$$X = \sum_{i=0}^{n-1} \beta^i x_i =: \mathsf{eval}_\beta(\mathbf{x}). \tag{3.6}$$

For $i$ out of the range $[0, n)$, we assume $x_i = 0$.

For (finite) alphabet $\mathcal{A}$, other than the standard one, with $\mathcal{A} \subset \mathbb{Z}$, we talk about the *$(\beta, \mathcal{A})$-representation*. In particular, parallel addition algorithms typically employ an alphabet that:

1. contains negative digits, represented with bars (i.e., for $d \in \mathbb{N}$, $\bar{d} := -d$),
2. is symmetric around zero (e.g., $\bar{\mathcal{A}} = \{\bar{2}, \bar{1}, 0, 1, 2\}$), and
3. yields a redundant representation.

Point 3 actually states a necessary condition for a parallel addition algorithm to exist, as shown by Kornerup [91].

**Example 3.4.** *Let us illustrate redundancy on two different representations of 2, using base $\beta = 4$ and the aforementioned alphabet $\bar{\mathcal{A}}$, as follows: $(1\bar{2}\bullet)_4 = 1 \cdot 4^1 + (-2) \cdot 4^0 = 0 \cdot 4^1 + 2 \cdot 4^0 = (02\bullet)_4$.*

We refer to the $(2, \bar{\mathcal{A}}_2)$-representation, where $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$, as the *signed binary* representation. For any alphabet, this kind of representation is also referred to as the *radix-based representation.*

### Parallel Addition Algorithm(s)

A family of parameterizable algorithms for parallel addition of multi-digit integers was introduced by Avizienis [11] in 1961. Later in 1978, Chow et al. [38] further improved the meta-algorithm so that it can work with smaller alphabets and even with the minimum integer base $\beta = 2$.

In [87], we compare sequential and parallel algorithms for the addition of TFHE-encrypted integers. We implement three sequential approaches, and two algorithms for parallel addition, namely those using:

- $\beta = 2$, $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$ (i.e., signed binary), and
- $\beta = 4$, $\bar{\mathcal{A}}_4 = \{\bar{2}, \bar{1}, 0, 1, 2\}$, respectively.

For both parallel algorithms, we develop three strategies, how each algorithm can be turned into the TFHE-encrypted domain; hence altogether, we compare three + six variants. Based on our experiments, we observe that although parallel approaches introduce a certain computational overhead, the fastest parallel approach outperforms the fastest sequential approach starting from as short as 5-bit integers; for 31-bit integers, it is already more than $6\times$ faster, provided that a sufficient number of threads is available.

For the development of other arithmetic operations to be presented in this chapter, we choose the fastest parallel strategy, which uses the signed binary representation; in [87] referred to as *Strategy IIa-F*. We recall this parallel addition method in Algorithm 4. Note that in this chapter, we do not further develop nor compare with any sequential approach.

**Algorithm 4** Parallel addition with $\beta = 2$ and $\bar{\mathcal{A}}_2 = \{\bar{1}, 0, 1\}$.

```
Input: $(2, \bar{\mathcal{A}}_2)$-representations $\mathbf{x}, \mathbf{y} \in \bar{\mathcal{A}}_2^{\,n}$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Output: $(2, \bar{\mathcal{A}}_2)$-representation $\mathbf{z} \in \bar{\mathcal{A}}_2^{\,n+1}$ of $Z = X + Y$.
1: for $i \in \{0, 1, \dots, n\}$ in parallel do
2:     $w_i \leftarrow x_i + y_i$
3:     $q_i \leftarrow w_i \gtreqless \pm 2 \vee (w_i \equiv \pm 1 \wedge w_{i-1} \gtreqless \pm 1)$
4:     $z_i \leftarrow w_i - 2q_i + q_{i-1}$                                ▷ (refresh)
5: end for
6: return z
```

*Note* 3.1. In Algorithm 4 on line 3, we abuse notation and we combine the functions $x \gtreqless \pm b$ and/or $x \equiv \pm b'$ with logical operations. Unless $+1$ meets $-1$ in such an expression, we treat $+1$'s or $-1$'s as logical 1's and we keep their positive or negative signs, respectively. In case $-1$ *does* meet $+1$ (e.g., $-1 \wedge +1$), we evaluate the expression as 0. E.g., for $w_i = +1$ and $w_{i-1} = -2$ in Algorithm 4, we have $0 \vee (+1 \wedge -1) = 0 \vee 0 = 0$.

### Conversions & Other Operations in Signed Binary

The conversion from the standard to the signed binary representation is trivial, since $\mathcal{A}_2 \subset \bar{\mathcal{A}}_2$. Note that this does not hold in general for other signed representations that are used for parallel addition (e.g., $\mathcal{A}_4 \not\subset \bar{\mathcal{A}}_4$), where however parallel addition might be employed. For the opposite direction, a conversion is needed; in addition, from the impossibility result of Kornerup [91], it follows that this conversion *cannot* be parallelized; find more details in Appendix A.

In the signed binary representation, some operations can be implemented fairly straightforwardly: e.g., multiplication, which will be discussed in Section 3.3.3. Other operations require a more careful approach: e.g., rounding, which will be discussed in Section 3.4.3. Yet other operations—in particular bit-wise operations—require a conversion to the standard binary, which we leave as future work.

## 3.3 Parallel Arithmetics over TFHE-Encrypted Data

In this section, we propose approaches and algorithms for the evaluation of basic arithmetic operations over TFHE-encrypted multi-digit integers, with particular respect to parallelization. In contrast to other works, e.g., [66, 21], we provide our algorithms in the *cleartext* domain, which simplifies their reading and understanding. To turn an algorithm into the encrypted domain, operations are simply replaced with their homomorphic counterparts. Indeed, in our algorithms, either we use basic homomorphic operations of multi-value TFHE (i.e., /weighted/ summation and LUT evaluation), or we rely on algorithms defined previously. Note that this allows us/others to replace the underlying TFHE scheme with another compatible scheme if needed.

**Bootstrapping Strategy** First, let us commit to a *bootstrapping strategy*: we demand to *always return freshly bootstrapped samples* from all arithmetic operations. I.e., these samples are required to be a direct output of the bootstrapping algorithm, without any further homomorphic additions. Although this is not always needed—in particular in the last step before decryption—we make this guarantee so that the results' correctness is ensured, independent of the operation flow.

**Complexity Measure** Let us assume:

1. bootstrapping is the *dominant* operation and others are negligible,

2. we have an *unlimited* number of bootstrapping threads,

3. parallelization is *ideal*, i.e., there is no additional orchestration cost.

As the primary complexity measure, we consider the *total running time*, expressed in terms of bootstraps, i.e., the minimal number of consecutive bootstraps in case of ideal parallelization.

*Remark* 3.2. In some extreme cases, we may resort to a different measure, e.g., the total number of bootstraps. Note that the total number of bootstraps is equal to the total running time in the sequential setting (expressed as the number of bootstraps). In practice, it is proportional to the evaluation costs in terms of processor time or electricity consumption.

### 3.3.1 Parallel Addition

First, we focus on the cornerstone arithmetic operation, which is multi-digit integer addition. We recall how the parallel addition algorithm (Algorithm 4) can be turned into the TFHE-encrypted domain. Based on this algorithm, we build other arithmetic operations in the following (sub)sections. We also outline how some non-necessary operations can be avoided in parallel addition.

### Parallel Addition in the TFHE-Encrypted Domain

Let us revisit how the selected parallel addition algorithm (Algorithm 4) can be turned into the TFHE-encrypted domain (firstly proposed in [87]). To evaluate $w_i$ and $z_i$ (lines 2 and 4, respectively), additive homomorphism does the job. The value of $q_i$ (line 3) is non-linear in the inputs $w_{i-1}$ and $w_i$; we illustrate $q_i = q_i(w_{i-1}, w_i)$ in a table in the left-hand side of Figure 3.1. We suggest to "linearize" the table into a one-dimensional LUT, using $w_{i-1} + 3w_i$ as a selector; we provide an illustration in the right-hand side of Figure 3.1 (there is indeed a single value in each column). It follows that $q_i(w_{i-1}, w_i)$ can be rewritten as

$$q_i = \big(w_{i-1} + 3w_i \gtreqless \pm 4\big), \tag{3.7}$$

which allows to construct respective negacyclic LUT, associated to a threshold function (cf. Section 3.2.1). Note that in accordance with our bootstrapping strategy, we apply an additional identity bootstrap on line 4 so that the output consists of freshly bootstrapped samples.

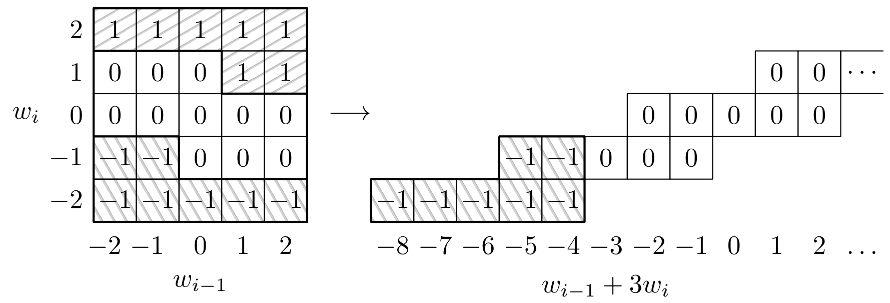


Figure 3.1: Left-hand side: values of $q_i = q_i(w_{i-1}, w_i)$ as per Algorithm 4. Right-hand side: "linearization" of the table into a one-dimensional LUT, using the selector $w_{i-1} + 3w_i$.

**Analysis** As shown in [87], we need a message space with $\pi \geq 5$ bits, and we demand quadratic weights $2^{2\Delta} \geq 20$. The algorithm further requires 1 bootstrapping thread per instance (usually per bit of input; a discussion on its optimization follows). It runs in 2 bootstrapping steps, totaling 2 bootstraps per instance.

### Avoiding Non-Necessary Operations

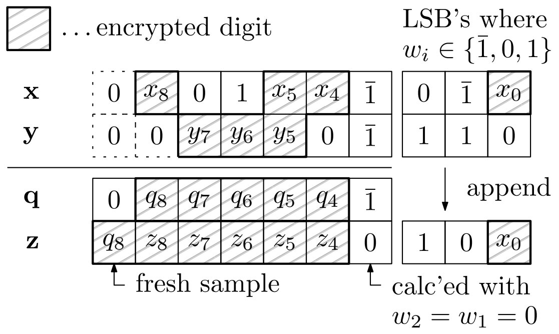
In case the encrypted digits of two addends $\mathbf{x}$ and $\mathbf{y}$ are not aligned and/or some are unencrypted[2], there may occur non-necessary operations, including bootstraps. Let us discuss these situations with respect to their position, either at the *Least Significant Bit* (LSB) or at the *Most Significant Bit* (MSB).

- *LSB Part:* Suffixes of LSBs of $\mathbf{x}$ and $\mathbf{y}$, where it is guaranteed that all $w_i = x_i + y_i \in \bar{\mathcal{A}}_2$ (e.g., $0 + x$ with $x$ encrypted, or $1 + \bar{1}$, ...), can be separated from both addends before the calculation and then simply appended back. Note that the "missing" separated digits must be considered to be zero for the rest of the calculation.
- *MSB Part:* At the MSB side, the parallel addition algorithm must be performed until the very end due to the left-propagating local carry. On top of that, an additional bit

[2]E.g., multiplication of encrypted 3-bit number $x = (x_2x_1x_0\bullet)_2$ by unencrypted $17 = (10001\bullet)_2$ may result in $(x_2x_1x_00x_2x_1x_0\bullet)$, which holds unencrypted zero at the position of $2^3$.

(say of index $n$) must be prepended: we have $q_n = 0$ and $z_n = x_n + y_n - 2q_n + q_{n-1} = 0+0-2\cdot 0+q_{n-1}$, which is a fresh sample. I.e., there is no need for the refreshal bootstrap of $z_n$ (unlike other $z_i$'s; cf. comment on line 4 of Algorithm 4).

We provide an example in Figure 3.2. We refer to bits that need to be bootstrapped as *active* bits.



. . . encrypted digit
LSB's where $w_i \in \{\bar{1}, 0, 1\}$
x
0 $x_8$ 0 1 $x_5$ $x_4$ $\bar{1}$ 0 $\bar{1}$ $x_0$
y
0 0 $y_7$ $y_6$ $y_5$ 0 $\bar{1}$ 1 1 0
q
0 $q_8$ $q_7$ $q_6$ $q_5$ $q_4$ $\bar{1}$
append
z
$q_8$ $z_8$ $z_7$ $z_6$ $z_5$ $z_4$ 0 1 0 $x_0$
fresh sample
calc'ed with $w_2 = w_1 = 0$

Figure 3.2: Example of avoiding non-necessary operations (bootstraps) during the addition of integers that are not aligned and/or contain unencrypted digits.

### 3.3.2 Scalar Multiplication

By *scalar multiplication* we mean (homomorphic) multiplication of *encrypted* integer $X$ by *known* integer $k$:

$$k \odot \mathsf{Encr}(X) \approx \mathsf{Encr}(k \cdot X). \tag{3.8}$$

By definition, scalar multiplication can be evaluated as $(k{-}1)\times$ repeated additions of $\mathsf{Encr}(X)$ to itself (later simplified to $X$). However, we can do better and decrease the number of additions. Let us give a simple example: $4 \cdot X$ can be calculated either as $((X + X) + X) + X$ in three additions, or as $(X + X) + (X + X)$ in just two additions, since $X + X$ can be reused. Hence, our goal is to minimize the number of additions needed to evaluate scalar multiplication – in our case, in the radix-based, TFHE-encrypted domain.

**Towards Our Method** First, we recall an approach adopted in *Elliptic Curve Cryptography* (ECC), namely the so-called *addition(-subtraction) chains*, which aim at minimizing the number of additions (and subtractions) during scalar multiplication over an elliptic curve. Next, we extend the definition of these chains by an assumption that doubling goes for free: unlike ECC, our setup employs a radix-based representation, where the cost of doubling is negligible compared to addition/subtraction. Finally, due to the anticipated intractability of finding the optimal chain of our type, we suggest applying the so-called *window method.* This method splits a particular (signed binary) representation of the actual scalar into a minimum number of sub-scalars of a short, fixed length. For those short scalars, it is feasible to pre-compute the (nearly) optimal chains of our type. Then, we evaluate them and combine the intermediate results with our parallel addition, obtaining the final result of scalar multiplication.

#### Addition (Subtraction) Chains

As outlined, the number of additions needed for scalar multiplication may differ from approach to approach. Hence, given $k$, our goal is to find a prescription that evaluates scalar multiplication, while calling the lowest number of additions. In ECC, this problem is formulated in terms of *Addition Chains*, which represent the decomposition of scalar multiplication into additions; let us recall a simplified definition.

**Definition 3.1** (Addition Chain (simplified))**.** Let $k \in \mathbb{N}$, $k > 1$. We call the tuple $(1, k_1 \ldots, k_{l-1}, k_l = k)$, $l, k_i \in \mathbb{N}$, an *Addition Chain* for $k$ if $\forall i \in [1, l]$ there $\exists r, s \in [0, i-1]$ such that $k_i = k_r + k_s$. I.e., every element $k_i$ is a sum of some two preceding elements.

To obtain $k \cdot X$ using an addition chain for $k$, denoted $\mathsf{AC}_k$, we evaluate $\mathsf{AC}_k$ using the same series of additions, but starting from $X$, instead of 1. We denote the result by $\mathsf{eval}(\mathsf{AC}_k, X) = k \cdot X$.

Many variants of this problem have been proposed and many approaches have been suggested – for a comprehensive overview of these methods, we recommend Chapter 9 of [41]. Here we point out two of them:

- if subtractions are allowed (i.e., $k_i = k_r \pm k_s$ as per Definition 3.1), we refer to *Addition-Subtraction Chains* (ASC),
- if multiple integers $k^{(0)}, \ldots, k^{(t-1)}$ are to be present in the chain (i.e., there is not only one final $k$), we refer to *Addition Sequences.*

Downey et al. [49] show that the set of all tuples of the form $(k^{(0)}, \ldots, k^{(t-1)}; l)$, such that there exists an addition sequence of length $l$ for $\{k^{(0)}, \ldots, k^{(t-1)}\}$, is $\mathsf{NP}$-complete (as a decision problem). It is hence widely believed that also finding the optimal/shortest addition(-subtraction) chain is an intractable task.

### Chains with Free Doubling

Between our problem of scalar multiplication and that of ECC, there is a substantial difference: in our case, *doubling goes at a negligible cost*, unlike ECC, where doubling is considered as expensive as addition. Indeed, in our base-2 representation with encrypted digits, doubling melts down to appending an unencrypted zero to the LSB (equivalent to left bit-shift). For this reason, we define another class of ASCs.

**Definition 3.2** (Free-Doubling Addition-Subtraction Chain ($\mathsf{ASC}^*$))**.** Let $k \in \mathbb{N}$, $k$ is not a power of 2. We call the tuple $([1], [k_1], \ldots, [k_l])$, with $l, k_i \in \mathbb{N}$, $k_i$ odd, a *Free-Doubling Addition-Subtraction Chain* for $k$ if the following holds:

- $\exists t \in \mathbb{N}$ such that $k = 2^t \cdot k_l$,
- $\forall i \in [1, l]$ there $\exists r, s \in [0, i-1]$, $t \in \mathbb{N}_0$, such that $k_i = \pm k_r \pm 2^t \cdot k_s$.

We consider $[k_i]$ as a class of numbers of the form $2^t \cdot k_i$.

**Example 3.5.** *An interesting example of* $\mathsf{ASC}^*$ *goes for* $805 =$ `0b1100100101` – *we encourage the reader to try herself before checking the solution*[3].

Hence, the problem of finding the order of additions/subtractions and shifts that lead to $k \odot \mathsf{Encr}(X)$ – with the lowest number of additions/subtractions – melts down to finding the shortest $\mathsf{ASC}^*$, which we assume to be intractable (more research is needed). For this reason, we resort to a heuristic approach.

[3]$([1], [5 = 1 + 1 \cdot 2^2], [25 = 5 + 5 \cdot 2^2], [805 = 5 + 25 \cdot 2^5])$.
Other similar examples are 1 173, 1 209, 1 305, 1 353, 1 377, 1 595, 1 605, 1 695, 1 743, 2 585, 3 129, 3 143, 3 195, 3 205, 3 633, 3 717 and 3 813; some include subtraction, which makes them even more tricky to discover by a pen and paper.

$$950048719935 \xrightarrow{\textsc{KoyamaTsuruokaRecoding}}$$

$$\underbrace{100\bar{1}000\bar{1}0\bar{1}\bar{1}}_{885}\,00\,\underbrace{\bar{1}\bar{1}00\bar{1}0\bar{1}\bar{1}0001}_{-3\,247}\,00000\,\underbrace{1000100000\bar{1}}_{1\,087} \xrightarrow{\textsc{WindowValues\&Shifts}_{12}}$$

$$(885, 30), (-3\,247, 16), (1\,087, 0), \text{ for which it holds}$$

$$885 \cdot 2^{30} - 3\,247 \cdot 2^{16} + 1\,087 \cdot 2^{0} = 950048719935.$$

Figure 3.3: Illustration of the window method on top of the Koyama-Tsuruoka recoding.

## Rewriting Scalars & Window Method

Due to the anticipated hardness of finding the optimal ASC* for scalar $k$, we suggest applying the following approach, inspired by methods of ECC:

1. pre-compute (ideally optimal) ASC*s for all odd integers of small, fixed bit-length $w_l$,
2. rewrite the binary representation of $k$ into a signed binary representation, such that there are as long sequences of zeros as possible,
3. apply the sliding window method.

Let us explain each step in detail.

**Step 1: Pre-computation of Short ASC*s** We pre-compute ASC*s for all odd 12-bit integers[4]. The description of our approach is out of the scope of this work: we leave this for future work and at this moment, we provide the pre-computed ASC*s "as is". Although we use a brute-force approach, we do not guarantee the optimality, which is rather tricky to show, mainly due to the unlimited power of two within $[k_i]$'s.

**Step 2: Rewriting the Scalar** To decrease the number of windows in the subsequent window method, it is worth using a signed binary representation for $k$ that not only minimizes the Hamming weight but also maximizes the length of sequences of zeros. For this purpose, we employ the Koyama-Tsuruoka recoding [92]. This recoding minimizes the resulting Hamming weight and on average, it achieves 1.42-bit long sequences of zeros, compared to 1.29-bit for the "traditional" *Non-Adjacent Form* (NAF; [18]).

**Step 3: Sliding Window Method** Finally, we apply the sliding window method of length 12, which we illustrate in Figure 3.3 together with the Koyama-Tsuruoka recoding; find a rigorous description of the sliding window method in [41], Chapter 9.1.3.

**The Overall Algorithm** We provide the overall scalar multiplication method in Algorithm 5. As outlined in the algorithm, any repeated window value can be re-used and also the ASC*s can be evaluated in parallel. We comment on the final aggregation on line 8 later in Section 3.3.3.

[4]Find our ASC*s for all odd 12-bit integers in our library [109] in the `assets/asc-12.yaml` file.

**Algorithm 5** Scalar Multiplication.

```
Input: $k, X \in \mathbb{Z}$ ($k$ to be cleartext, $X$ to be encrypted)
Input: $\mathsf{ASC}^*$s of length $l$
Output: $Z = k \cdot X$.
 1: $\mathbf{k} \leftarrow$ KoyamaTsuruokaRecoding($|k|$)
 2: $(w_i, s_i)_{i=1}^{n_w} \leftarrow$ WindowValues&Shifts$_l(\mathbf{k})$
                                   ▷ i.e., $|k| = \sum_{i=1}^{n_w} w_i \cdot 2^{s_i}$, $|w_i| < 2^l$
 3: for $i \in \{1, \ldots, n_w\}$ in parallel do
 4:     $W_i^{(X)} \leftarrow \mathsf{eval}(\mathsf{ASC}^*_{|w_i|}, X)$    ▷ do not calc. twice for the same $|w_i|$
 5: end for
 6: $Z \leftarrow 0$
 7: for $i = 1 \ldots n_w$ do
 8:     $Z \leftarrow Z + \mathrm{sgn}(w_i) \cdot W_i^{(X)} \cdot 2^{s_i}$    ▷ $W_i^{(X)}$ shifted, (negated)
 9: end for
10: return $\mathrm{sgn}(k) \cdot Z$
```

**Average Numbers of Additions** For 12-bit windows in the Koyama-Tsuruoka recoding, we observe that the average number of additions is 3.10 for $\mathsf{ASC}^*$s, as opposed to 3.88 for the standard double-and-add/sub method, which is suggested in [87] (i.e., about 20% fewer additions). More details are given in Appendix B.

### 3.3.3 Multiplication

There exist several (cleartext) algorithms for integer multiplication, most of them extend to algebraic rings, too; for a comprehensive overview, we refer to a thorough survey by Bernstein [14]. Sorted by their asymptotic complexity, below we provide the most famous ones:

- the *schoolbook algorithm*, where every pair of digits gets multiplied, followed by a summation, and which runs in $O(n^2)$,
- the *Karatsuba algorithm* [78], which is based on the *Divide-and-Conquer* strategy and which runs in $O(n^{\log 3})$, and
- the *Schönhage-Strassen algorithm* [118], which is based on a number-theoretic transform and which runs in $O(n \cdot \log n \cdot \log\log n)$.

Although the last one achieves the best asymptotic complexity, it is only worth for huge numbers: e.g., in the GMP Library [65], the threshold `MUL_FFT_THRESHOLD`[5] switches multiplication to the Schönhage-Strassen algorithm for integers longer than high thousands of bits.

Therefore, for the encrypted domain, we do not consider Schönhage-Strassen – instead, we find threshold $t_M$, starting from which Karatsuba outperforms the schoolbook algorithm.

In the following subsections, we recall the Karatsuba algorithm in the clear, we propose a method for the multiplication of individual encrypted signed bits, and we comment on the final summation in both the schoolbook and Karatsuba algorithm, which also applies to scalar multiplication.

[5]https://gmplib.org/manual/Multiplication-Algorithms, accessed Sep 2022.

### Karatsuba Algorithm

First, let us recall the cleartext version of the Karatsuba algorithm for balanced inputs as Algorithm 6. It follows the *Divide and Conquer* strategy (cf. line 5) and it switches to the schoolbook algorithm if the input length is lower than the threshold $t_M$ (cf. line 2). Indeed, with short inputs, the schoolbook algorithm outperforms Karatsuba; find more details in Appendix C. Note that we do not explicitly recall the schoolbook multiplication algorithm $\textsc{MulSchoolbook}_\beta$, which melts down to pairwise multiplication of individual (signed) digits, followed by a summation.

**Algorithm 6** Karatsuba Multiplication.

```
Input: $(\beta, \mathcal{A})$-representations $\mathbf{x}, \mathbf{y} \in \mathcal{A}^n$ of $X, Y \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: threshold $t_M \geq 4$,
Output: $X \cdot Y$.
 1: function $\textsc{MulKaratsuba}_\beta(\mathbf{r}, \mathbf{s})$
 2:     if $\mathsf{len}(\mathbf{r}) = \mathsf{len}(\mathbf{s}) < t_M$ then
 3:         return $\textsc{MulSchoolbook}_\beta(\mathbf{r}, \mathbf{s})$
 4:     end if
 5:     split $\mathbf{r}, \mathbf{s}$ equally into two parts, s.t. $(\mathbf{r}_1, \mathbf{r}_0) = \mathbf{r}$ and $(\mathbf{s}_1, \mathbf{s}_0) = \mathbf{s}$
                                                        ▷ little-endian representation
 6:     $n_0 \leftarrow \mathsf{len}(\mathbf{r}_0) = \mathsf{len}(\mathbf{s}_0)$
 7:     in parallel do
 8:         $A \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_1, \mathbf{s}_1)$
 9:         $B \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_0, \mathbf{s}_0)$
10:         $C \leftarrow \textsc{MulKaratsuba}_\beta(\mathbf{r}_1 + \mathbf{r}_0, \mathbf{s}_1 + \mathbf{s}_0)$
11:     end parallel
12:     return $A \cdot \beta^{2n_0} + (C - A - B) \cdot \beta^{n_0} + B$
                                              ▷ calc. using additions & base-$\beta$ shifts
13: end function
14: return $\textsc{MulKaratsuba}_\beta(\mathbf{x}, \mathbf{y})$
```

### Multiplication of Individual Encrypted Signed Bits

In our integer representation, digits hold signed bits. Let us outline an algorithm that calculates a product of two encrypted signed bits, using a single LUT evaluation; see Algorithm 7. An illustration of the LUT together with its selector is given in Figure 3.4. Due to the size of the LUT, we need $\pi \geq 5$.

**Algorithm 7** $1 \times 1$-bit Multiplication in one LUT.

```
Input: $x, y \in \bar{\mathcal{A}}_2$,
Output: $x \cdot y$.
 1: return $(0, 0, -1, 0, 1 \| 1, 0, -1, 0)[3x + y]$
```

### Summation of Intermediate Results

Both schoolbook and Karatsuba algorithms are followed by a summation of their intermediate results.

$$\begin{array}{cc}
 & x \cdot y \\
x & \begin{array}{c|c|c|c|}
\hline
1 & -1 & 0 & 1 \\ \hline
0 & 0 & 0 & 0 \\ \hline
-1 & 1 & 0 & -1 \\ \hline
\multicolumn{1}{c}{} & \multicolumn{1}{c}{-1} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1}
\end{array}
\end{array}
\qquad
\begin{array}{c}
3x + y \\
\begin{array}{|c|c|c|}
\hline
2 & 3 & 4 \\ \hline
-1 & 0 & 1 \\ \hline
-4 & -3 & -2 \\ \hline
\multicolumn{1}{c}{-1} & \multicolumn{1}{c}{0} & \multicolumn{1}{c}{1}
\end{array}
\end{array}$$

$$y$$

Figure 3.4: Values of $x \cdot y$ and those of selector $3x + y$. Find respective LUT in Algorithm 7.

For the schoolbook, we illustrate the summation in Figure 3.5. With each addition, the intermediate value grows one bit to the left (MSB; cf. Figure 3.2). Therefore, this way, the final result is *exactly* twice as long as the inputs. However, if we decide for some parallelization of the summation, it is worth leaving the last line for the very last step, otherwise, the result gets longer.

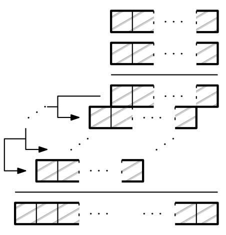


Figure 3.5: Summation within schoolbook multiplication.

For Karatsuba over $n$-bit inputs, there are several aspects of the final summation step (cf. line 12 of Algorithm 6) to comment:

1. The result of Karatsuba is *longer than* $2n$ – indeed, the last addition step extends the final result by at least one bit. Note that it also depends on the length of the nested products – these might be already longer than twice their input due to a nested Karatsuba.

2. The value of $-A - B$ can be pre-computed in parallel to the calculation of $C$, which is more demanding due to the additions $\mathbf{r}_1 + \mathbf{r}_0$ and $\mathbf{s}_1 + \mathbf{s}_0$. Then the value of $C + (-A - B)$ is calculated in the first place.

3. Depending on the length of $B$, different approaches may apply to minimize the result's length as well as the number of steps:

   - if $|B| = 2n_0$, $A$ and $B$ can be simply concatenated to obtain $A \cdot \beta^{2n_0} + B$, then shifted $C - A - B$ is added;
   - otherwise, $B$ is first added to shifted $C - A - B$, only then shifted $A$ is added.

4. For the first-level Karatsuba (i.e., with all nested schoolbooks), it is worth splitting inputs of odd length such that the LSB part is one bit longer – this approach leads to a shorter addition in the final step. However, for a nested Karatsuba, different approaches may achieve lower bootstrapping complexity; cf. Example 3.6. Finding the optimal approach for every scenario is out of the scope of this work.

**Example 3.6.** *For a nested Karatsuba, there might be worth another way of splitting odd numbers than the one described in point 4: Let us say we have* $t_M = 16$*. Then, splitting* $31 \to (16|15)$*—which is not proposed by point 4 and which calls schoolbook at the LSB part—leads to the concatenation (cf. point 3), and in total, multiplication this way requires* 2 497 *bootstraps. Whereas the proposed way of splitting, i.e.,* $31 \to (15|16)$*, which calls Karatsuba at the LSB part, requires* 2 531 *bootstraps – mainly due to the additional cost of the* $A \cdot 2^{2n_0} + B$ *addition, instead of their concatenation as in the previous case. This gives a counter-example to the odd-number splitting argument, which might not hold in case recursive calls of Karatsuba occur.*

We provide more details on other complexity measures of multiplication (and squaring) later in Section 3.5.4 and (in particular) in Appendix D.

## 3.3.4 Squaring

For integer squaring, which is a special case of multiplication, we implement a dedicated algorithm. Similarly to Karatsuba multiplication, we employ the divide and conquer strategy; find our method for squaring in Algorithm 8. For the threshold $t_S$ (line 2), we obtain the value $t_S = 4$ for our setup, using an approach analogical to multiplication (cf. Appendix C). We comment on the schoolbook squaring algorithm (line 3) later.

**Algorithm 8** Squaring via Divide-and-Conquer.

```
Input: $(\beta, \mathcal{A})$-representation $\mathbf{x} \in \mathcal{A}^n$ of $X \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: threshold $t_S \geq 4$,
Output: $X^2$.
1: function SquDivNConq_β(r)
2:     if $\mathsf{len}(\mathbf{r}) < t_S$ then
3:         return SquSchoolbook_β(r)
4:     end if
5:     split r equally into two parts, s.t. $(\mathbf{r}_1, \mathbf{r}_0) = \mathbf{r}$
                                                        ▷ little-endian representation
6:     $n_0 \leftarrow \mathsf{len}(\mathbf{r}_0)$
7:     in parallel do
8:         $A \leftarrow$ SquDivNConq_β($\mathbf{r}_1$)
9:         $B \leftarrow$ SquDivNConq_β($\mathbf{r}_0$)
10:        $C \leftarrow$ MulKaratsuba_β($\mathbf{r}_1, \mathbf{r}_0$)
11:    end parallel
12:    return $A \cdot \beta^{2n_0} + C \cdot \beta^{n_0+1} + B$        ▷ additions & base-β shifts
13: end function
14: return SquDivNConq_β(x)
```

Compared to multiplication on a duplicated input, our squaring algorithm evaluates fewer bootstraps in fewer steps, which is mainly achieved thanks to:

- fewer terms to be evaluated:
  - two additions on line 10 of Algorithm 6 are not evaluated on line 10 of Algorithm 8,
  - instead of $C-A-B$ on line 12 of Algorithm 6, there is only $C$ on line 12 of Algorithm 8; and
- more efficient squaring of short inputs in case of signed binary representation.

**Squaring of Short, Signed Binary Inputs** For short, signed binary input $\mathbf{x}$ (namely 2- or 3-bit with $\pi = 5$), we suggest to calculate individual bits of the resulting square directly and in parallel. The idea is to evaluate homomorphically $X = \mathsf{eval}_2(\mathbf{x})$ as per (3.6), which melts down to scalar multiplications and additions of TFHE samples (i.e., no bootstrap is needed). To evaluate a bit of $Y = X^2$, we use $X$ as a selector into a dedicated LUT; find an illustration in Table 3.1, where columns represent these LUTs. For the full algorithm, we refer to Appendix E, Algorithm 13.

Table 3.1: Bits of $Y = X^2$ and respective selector $X$. Columns $y_i$ are intended to be encoded into LUTs.

| $X$ | bits of $Y = X^2$ | | | | | | $X^2$ |
|---|---|---|---|---|---|---|---|
| | $y_5$ | $y_4$ | $y_3$ | $y_2$ | $y_1$ | $y_0$ | |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| $\pm1$ | 0 | 0 | 0 | 0 | 0 | 1 | 1 |
| $\pm2$ | 0 | 0 | 0 | 1 | 0 | 0 | 4 |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ |
| $\pm7$ | 1 | 1 | 0 | 0 | 0 | 1 | 49 |

Note that in the signed binary, 3 bits may encode $X \in [-7, 7]$, and the output is up to 6-bit. For 2-bit inputs, we only calculate the output bits up to $y_3$. Also note that the bit at $2^1$ position (i.e., $y_1$) is always zero, which stems from the fact $a^2 \bmod 4 \in \{0, 1\}$. Hence, for 2- and 3-bit inputs, we evaluate 3 and 5 LUTs, respectively, and we obtain the result in a single bootstrapping step. We provide more details on other complexity measures of squaring (and multiplication) later in Section 3.5.4 and (in particular) in Appendix D.

Recall that we have $t_S = 4$ and we use the signed binary, i.e., SquSchoolbook$_\beta$ (on line 3 of Algorithm 8) is fully implemented via this method for squaring of short inputs.

## 3.4 Signum-Based Operations over TFHE-Encrypted Data

In this section, we put forward some other, frequently used, signum-based operations in the signed binary representation, while bearing in mind the limited set of available homomorphic operations over the encrypted digits. Namely, we present parallel algorithms for *signum* and *maximum* (improved versions of [87]), and we introduce a new algorithm for *rounding* at a selected digit position.

In principle, these algorithms are based on number comparison. However, in the signed binary representation, the lexicographic comparison may fail[6]. Therefore, we suggest reducing the problem of number comparison to signum: we subtract the numbers (in parallel) and we compare the result with zero. This works in the signed binary as expected, i.e., the sign of the leading bit determines the sign of the result.

### 3.4.1 Signum

A method for comparison of two integers, given as a series of encrypted digits, was firstly proposed by Bourse et al. [21]. Later, we adjusted this method to signed integer representations in [87]; we recall it in Algorithm 9.

[6] As an example, $(\mathbf{0}11\bullet)_2 = 3 > 2 = (\mathbf{1}\bar{1}0\bullet)_2$, although $0 < 1$ at the leading position of each number.

**Algorithm 9** Signum over $(\beta, \mathcal{A}_\beta - \mathcal{A}_\beta)$-representation ([21]; modified).

```
Input: $(\beta, \mathcal{A}_\beta - \mathcal{A}_\beta)$-representation $\mathbf{z} \in (\mathcal{A}_\beta - \mathcal{A}_\beta)^n$ of $Z \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: message space with bit-length $\pi \geq 3$, such that $2^{\pi-1} \geq \beta$,
Output: $\operatorname{sgn}(Z)$.
1: $\gamma \leftarrow \pi - 1$
2: function SgnParalReduce$_\gamma(\mathbf{a})$
3:     $k \leftarrow \mathsf{len}(\mathbf{a})$
4:     if $k = 1$ then
5:         return $a_0 \gtreqless \pm 1$
6:     end if
7:     for $j \in \{0, 1, \ldots, \lceil k/\gamma \rceil - 1\}$ in parallel do
8:         for $i \in \{0, 1, \ldots, \gamma - 1\}$ in parallel do
9:             $s_{\gamma j+i} \leftarrow 2^i \cdot \left(a_{\gamma j+i} \gtreqless \pm 1\right)$          ▷ scale $f_1$ by $2^i$
10:        end for
11:        $b_j \leftarrow \sum_{i=0}^{\gamma-1} s_{\gamma j+i}$
12:    end for
13:    return SgnParalReduce$_\gamma(\mathbf{b})$
14: end function
15: return SgnParalReduce$_\gamma(\mathbf{z})$
```

We propose an improvement, which only works in the signed binary representation: we suggest skipping the bootstrapped (and scaled) comparison $a_{\gamma j+i} \gtreqless \pm 1$ on line 9 of Algorithm 9 in the first level of recursion since we have already $a_{\gamma j+i} \in \{\bar{1}, 0, 1\}$. Instead, $a_{\gamma j+i}$'s get directly scalar-multiplied by $2^i$'s and aggregated into $b_j$, which—compared to [87]—saves one level of bootstrapping and reduces the number of threads by a factor of about four.

Next, note that the comparison function on line 5 can be replaced with another function if needed – n.b., the value of $a_0$ is only guaranteed to have the same sign as the top-level input. E.g., one may compute the non-negativity function as outlined in Example 3.2, which is useful for number comparison.

**Analysis** The evaluation of $a \gtreqless \pm 1$ on lines 5 and 9 (i.e., signum of $a$) requires no extra plaintext space (over what is needed for the representation of $a$'s) since signum is already negacyclic. Indeed, we need the range $[-2^\gamma + 1, 2^\gamma - 1]$, which perfectly fits within $\pi = \gamma + 1$ bits.

The aforementioned optimization (line 9 in the first level of recursion) mandates $2^{2\Delta} \geq (2^0)^2 + \ldots + (2^{\gamma-1})^2$, which equals 85 for $\pi = 5$. Note that this is the largest value of $2^{2\Delta}$ within Parmesan.

For the full parallelization, we need $\lceil n/\gamma \rceil$ threads (bootstrapping starts from the second level of recursion; as opposed to [87], which therefore requires $n$ threads), and the algorithm runs in $\lceil \log_\gamma(n) \rceil$ bootstrapping steps. The total number of bootstraps can be expressed as

$$S_\gamma(n) \coloneqq \left\lceil \frac{n}{\gamma} \right\rceil + \left\lceil \frac{n}{\gamma^2} \right\rceil + \ldots + \left\lceil \frac{n}{\gamma^{\lceil \log_\gamma(n) \rceil}} \right\rceil. \tag{3.9}$$

## 3.4.2 Maximum

We present an improved version of the method [87] for maximum of two integers $X$ and $Y$, represented in the signed binary representation and encrypted with $\pi \geq 5$; find it in Algorithm 10. The function SgnParalReduce$^+_\gamma$ on line 3 customizes line 5 of Algorithm 9, so that it evaluates the non-negativity function (cf. Example 3.2). Recall the notation introduced in Section 3.2.1, which is now used on line 5 of the algorithm. We illustrate the LUT creation in Figure 3.6.

| | | $\max\{x,y\}_i$ | | | | | | $s{+}2x_i{+}6y_i$ | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | $x<y$ | | | $x\geq y$ | | | | | | | | |
| $x_i$ | 1 | $\bar{1}$ | 0 | 1 | 1 | 1 | 1 | $-4$ | 2 | 8 | $-3$ | 3 | 9 |
| | 0 | $\bar{1}$ | 0 | 1 | 0 | 0 | 0 | $-6$ | 0 | 6 | $-5$ | 1 | 7 |
| | $\bar{1}$ | $\bar{1}$ | 0 | 1 | $\bar{1}$ | $\bar{1}$ | $\bar{1}$ | $-8$ | $-2$ | 4 | $-7$ | $-1$ | 5 |
| | | $\bar{1}$ | 0 | 1 | $\bar{1}$ | 0 | 1 | $\bar{1}$ | 0 | 1 | $\bar{1}$ | 0 | 1 |
| | | | $y_i$ | | | | | | | $y_i$ | | | |

Figure 3.6: Values of $i$-th bit of $\max\{x,y\}$ for both cases $x<y$ and $x\geq y$, and those of respective selector $s+2x_i+6y_i$, where $s=(x\geq y)$, i.e., the non-negativity function.

**Algorithm 10** Maximum over $(2,\bar{\mathcal{A}}_2)$-representation with $\pi\geq 5$ bits of plaintext space.

**Input:** $(2,\bar{\mathcal{A}}_2)$-representations $\mathbf{x},\mathbf{y}\in\bar{\mathcal{A}_2}^n$ of $X,Y\in\mathbb{Z}$, for some $n\in\mathbb{N}$,
**Output:** $\max\{X,Y\}$.

```
1: $\mathbf{r} \leftarrow \mathbf{x}-\mathbf{y}$                ▷ use favorite parallel alg.
2: $\gamma \leftarrow \pi - 1$
3: $s \leftarrow \textsc{SgnParalReduce}^+_\gamma(\mathbf{r})$
4: for $i \in \{0,1,\dots,n-1\}$ in parallel do
5:     $m_i \leftarrow \big(0,0,0,1,1,\bar{1},1,0,1,1 \| (\bar{1},\bar{1}),\bar{1},0,\bar{1},1,0,\bar{1}\big)[s+2x_i+6y_i]$
6: end for
7: return m
```

**Implementation Remark** Note that for $\pi=5$, there is a negacyclic overlap of two values within the LUT on line 5: $1,1$ before $\|$ is directly followed by its own negacyclic image $\bar{1},\bar{1}$, which is therefore given in parentheses.

**Analysis** In addition to the requirements of subtraction and those of $\textsc{SgnParalReduce}^+_\gamma$ (n.b., $\mathbf{r}$ is one bit longer than $\mathbf{x}$ and $\mathbf{y}$), we have: one bootstrap per bit (as opposed to three bootstraps in [87]), $2^{2\Delta}\geq 1^2+2^2+6^2=41$ due to the selector on line 5, and we need $n$ threads for the full parallelization (vs. $2n$ threads in [87]). In total, maximum runs in $2+\lceil\log_\gamma(n+1)\rceil+1$ bootstrapping steps with the total number of $2n+S_\gamma(n+1)+n$ bootstraps; cf. (3.9) for the definition of $S_\gamma(\cdot)$.

## 3.4.3 Rounding

Integer rounding operation at a given position (within its binary representation) can be expressed as function $R$ of two inputs: integer $X\in\mathbb{Z}$ to be rounded, and position $i\in\mathbb{N}$ to hold the last non-zero bit. The function can be written as

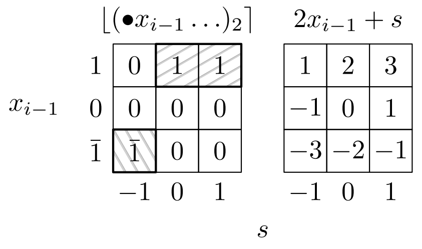
$$R(X,i) \coloneqq \left\lfloor \frac{X}{2^i} \right\rceil \cdot 2^i = \dots, \tag{3.10}$$

for $X = (x_{n-1} \ldots x_i x_{i-1} \ldots x_0 \bullet)_2$ also as

$$\ldots = (x_{n-1} \ldots x_i 0 \ldots 0 \bullet)_2 + \lfloor \underbrace{(\bullet x_{i-1} \ldots x_0)_2}_{r} \rceil \cdot 2^i. \tag{3.11}$$

With the standard binary alphabet, the value of $\lfloor r \rceil = \lfloor (\bullet x_{i-1} \ldots x_0)_2 \rceil$ equals to $x_{i-1}$. Indeed, if $x_{i-1} = 0$, then the remainder $r = (\bullet 0 x_{i-2} \ldots x_0)_2$ is always lower than $1/2$, conversely for $x_{i-1} = 1$, $r = (\bullet 1 x_{i-2} \ldots x_0)_2 \geq 1/2$.

However, with the signed binary alphabet, the leading bit $x_{i-1}$ does not determine how $r$ compares to $1/2$. In addition, such $r$ ranges in the interval $(-1, 1)$, unlike $[0, 1)$ for the standard binary alphabet, therefore, we need to compare with both $\pm 1/2$. Altogether nine combinations occur for $x_{i-1} \in \bar{\mathcal{A}}_2$ and for signum of the rest of $r$, denoted $s := \mathrm{sgn}\big((\bullet x_{i-2} \ldots x_0)_2\big)$. The resulting value of $\lfloor r \rceil$ as well as that of respective selector $2x_{i-1} + s$ are depicted in Figure 3.7. Find our method in Algorithm 11.



$\lfloor (\bullet x_{i-1} \ldots)_2 \rceil$
$2x_{i-1} + s$
$x_{i-1}$
$s$

| $x_{i-1}$ \ $s$ | $-1$ | $0$ | $1$ |
|---|---|---|---|
| $1$ | $0$ | $1$ ▨ | $1$ ▨ |
| $0$ | $0$ | $0$ | $0$ |
| $\bar{1}$ | $\bar{1}$ ▨ | $0$ | $0$ |

| $x_{i-1}$ \ $s$ | $-1$ | $0$ | $1$ |
|---|---|---|---|
| $1$ | $1$ | $2$ | $3$ |
| $0$ | $-1$ | $0$ | $1$ |
| $\bar{1}$ | $-3$ | $-2$ | $-1$ |

Figure 3.7: Values of $\lfloor r \rceil = \lfloor (\bullet x_{i-1} \ldots x_0)_2 \rceil$ and those of respective selector $2x_{i-1} + s$, where $s = \mathrm{sgn}\big((\bullet x_{i-2} \ldots x_0)_2\big)$.

**Algorithm 11** Rounding over $(2, \bar{\mathcal{A}}_2)$-representation.

```
Input: $(2, \bar{\mathcal{A}}_2)$-representation $\mathbf{x} \in \bar{\mathcal{A}}_2{}^n$ of $X \in \mathbb{Z}$, for some $n \in \mathbb{N}$,
Input: rounding position $i \in \mathbb{N}$,
Output: $R(X, i)$ as per (3.10).
 1: if $i > n$ then
 2:     return 0
 3: end if
 4: if $i = 1$ then
 5:     $t \leftarrow x_0$
 6:     go to line 11
 7: end if
 8: $\gamma \leftarrow \pi - 1$
 9: $s \leftarrow$ SgnParalReduce$_\gamma((x_{i-2}, \ldots, x_0))$          ▷ $(x_{i-2}, \ldots)$ might be empty
10: $t \leftarrow (0, 0, 1, 1 \| \bar{1}, 0, 0)[2x_{i-1} + s]$
11: $\mathbf{u} \leftarrow (x_{n-1} \ldots x_i \bullet)_2 + (t \bullet)_2$          ▷ use favorite parallel alg.
12: return $\mathbf{u} \ll i$
```

**Analysis** Rounding calls signum, evaluates a LUT, and runs parallel addition, while the LUT evaluation requires neither larger $\pi$ nor $2^{2\Delta}$ than any of those operations. Altogether, for $i > 1$, rounding requires $\max\{\lceil i-1/\gamma \rceil, n-i\}$ threads, it runs in $\lceil \log_\gamma (i-1) \rceil + 1 + 2$ bootstrapping steps, and in total $S_\gamma(i-1) + 1 + 2 \cdot (n-i)$ bootstraps are called; cf. (3.9).

## 3.5 Implementation & Experimental Results

In this section, we introduce our library for parallel arithmetics over TFHE-encrypted data. Then, we comment on its dependency on the Concrete Library and we also compare the abilities of these libraries in their current versions. Next, we outline an experiment design, covering the choice of parameters, inputs, and hardware. Finally, we put forward the results of our benchmarks and we conclude with a brief discussion.

### 3.5.1 The PARMESAN Library

We implement all operations, presented in the previous sections, in the PARMESAN Library [109]. Parmesan is an experimental library based on an existing implementation of TFHE – the Concrete Library [42], which we discuss later. Parmesan, as well as Concrete, are written in Rust[7] and they are compatible with the Rust's ecosystem, i.e., they can be easily added to a custom project via a standard Rust dependency.

To make the starting point smooth, our library goes with a simple demo (in the `README` file), which includes:

- TFHE parameter initialization, which loads a hard-coded parameter set;
- creation of User's and Cloud's scopes, which also generates respective keys;
- digit-by-digit encryption of integers $a$ and $b$, given in a (signed) base-2 representation;
- homomorphic addition $\mathsf{Encr}(a) \oplus \mathsf{Encr}(b)$;
- decryption of the result; and
- final check if $\mathsf{Decr}\big(\mathsf{Encr}(a) \oplus \mathsf{Encr}(b)\big) = a + b$.

All supported operations can be found in the `ParmArithmetics` trait, which is implemented for both Parmesan ciphertexts *and* for signed 64-bit integers (Rust type `i64`).

### 3.5.2 The Concrete Library

Among existing implementations of TFHE, we choose the Concrete Library [42] by Zama[8]: Concrete is open-source[9], is actively developed, implements state-of-the-art techniques, and also long-lasting support can be expected. At the time of writing, the latest release of Concrete is the `beta-2` version of `v0.2.0`, which we later denote as `v0.2`$_\beta$. In the beta version, many features are not stabilized yet and further improvements are expected to come with the full version (including a certain level of parallelization).

The Concrete library is written in Rust, where bundles of code are referred to as "crates". The `concrete` crate covers more than the implementation of TFHE (in the `concrete-core` sub-crate) – it consists of other three sub-crates: `concrete-boolean`, `concrete-shortint` and `concrete-int`, which respectively implements booleans, 2- to 7-bit unsigned integers and multi-precision unsigned integers, including various homomorphic operations for each type.

[7]https://rust-lang.org
[8]https://zama.ai
[9]Under the BSD 3-Clause Clear License.

### Relation to Parmesan

Internally, Parmesan's `TFHE` ciphertexts and respective homomorphic operations employ structures and functions of `concrete-core`. On top of these `TFHE`-level operations, integer arithmetic is built from the scratch: starting from the signed, radix-based representation, until the implementation of various arithmetic operations and their parallelization.

### Concrete's Arithmetics

In `concrete-int`, various arithmetic as well as bit-level operations are implemented over radix-based integer representations with selected power-of-two bases. Currently, a limited set of operations is implemented also for the CRT-based[10] representation (addition and multiplication; appears to be under development). In our experiments, we focus solely on the radix-based representation, which is by its nature closer to Parmesan. In addition, CRT-based representation cannot be used to mimic standard computer arithmetics, which operate mod $2^n$, unlike CRT, which operates modulo a product of coprime integers; for more details on CRT-based arithmetics, we refer to [13].

*Remark* 3.3. Given base $\beta = 2^k$ and digit-length $l$, the radix-based arithmetic of Concrete is equivalent to the standard unsigned $kl$-bit integer arithmetic. For each of the $l$ digits, encrypted with `TFHE`, the cleartext space actually consists of two parts: a $k$-bit *message* part, which covers the standard base-$\beta$ alphabet, and a *carry* part, which effectively extends the standard alphabet by a couple of additional bits. This allows performing a certain number of additions without the need for bootstrapping. In our experiments with Concrete, we run multiple additions until we reach the first bootstrap and in the results, we amortize the cost.

Multiplication is implemented using the standard schoolbook algorithm without any parallelization and for squaring, there is no special function. For scalar multiplication, there is no optimization in terms of free doubling (we verify this by calling multiplication by $4\,096 = 2^{12}$), nor in terms of addition-subtraction chains.

### Parmesan's vs. Concrete's Arithmetics

We provide a comparison of Parmesan's and Concrete's operations in Table 3.2. Usage-wise, the biggest difference is that Concrete mimics the behavior of native unsigned integer types (i.e., that of the ring $\mathbb{Z}_{2^{kl}}$), whereas Parmesan natively supports negative integers. Regarding the precision bound, this is rather a matter of implementation and both libraries could be easily extended.

## 3.5.3 Experiment Setup

Let us discuss particular choices of parameters, inputs, and hardware.

### Choice of Parameters

In Parmesan, we need 5 bits of message space and we do not need any padding. For this purpose, we choose Concrete's parameters named `PARAM_MESSAGE_2_CARRY_3`: there are 2 bits for the message and 3 extra bits for the carry, altogether 5 bits. Although there is no parameter corresponding to $2^{2\Delta}$ in Concrete, we did not encounter any error during any of our experiments or tests of Parmesan with this parameter choice, therefore, we consider our choice adequate. All

[10]Chinese Remainder Theorem.

Table 3.2: The current state of implementation of arithmetics in Parmesan (experimental library) and in Concrete `v0.2`$_\beta$ (some features not yet fully implemented in beta).

| Feature | **Parmesan** | **Concrete** `v0.2`$_\beta$ |
|---|---|---|
| Radix-based representation | ✓ (signed, unlimited) | ✓ (unsigned, mod $2^{kl}$) |
| CRT-based representation | ✗ | (✓) |
| Addition/subtraction, multiplication | ✓(parallel) | ✓(sequential) |
| ASC$^*$s for scalar multiplication | ✓ | ✗ |
| Karatsuba multiplication | ✓ | ✗ |
| Dedicated squaring | ✓ | ✗ |
| Bit operators | ✗ | ✓ |
| Signum, maximum, rounding | ✓ | ✗ |

parameters in Concrete are claimed to be chosen with the (expected) level of 128-bit security, which we verify with the `lattice-estimator`[11] [9].

In Concrete, we use the default builder for 2-bit unsigned integers, upon which we build longer integers. Regarding the digit's bit-length, there is no clear recommendation, however, our experiments with parallel addition algorithms [87] as well as an example implementation of the Game of Life[12] tend to prefer shorter message space.

### Choice of Inputs

We choose to benchmark 4-, 8-, 16-, and 32-bit values with Addition/Subtraction, Multiplication, and Squaring. For Signum, Maximum, and Rounding, we only benchmark 32-bit numbers to spot the effect of recursion.

For Scalar Multiplication, we choose to verify the following scalars: 4 095, 4 096, 4 097, 805, and 3 195. For 4 096, there shall be no operation needed in any of the bases: 2 (Parmesan), 4 (our setup of Concrete), 8, or 16. For 4 095 and 4 097, we aim at observing, whether one operation is used in both cases, i.e., whether both ways $4\,095 = 2^{12} - 1$ and $4\,097 = 2^{12} + 1$ are used. If this is not the case, we would observe a big gap for 4 095, since its Hamming weight is 12, as opposed to 2 for 4 097. As outlined in Example 3.5, 805 has a very efficient addition chain of just 3 additions, a similar property goes also for 3 195.

### Choice of Hardware

For our experiments, we choose two machines:

- an experimental server with a 12-threaded Intel Core i7-7800X processor (EURECOM's internal machine), and
- a supercomputer's node with two 64-threaded AMD EPYC 7543 processors (operated by e-INFRA CZ[13]).

Note that the 128-threaded machine has a sufficient number of threads to achieve the full parallelization for most of the operations (unlike, in particular, the multiplication of long integers).

[11] https://github.com/malb/lattice-estimator
[12] https://www.zama.ai/post/the-game-of-life-rebooted-with-concrete-v0-2, accessed Sep 2022.
[13] https://e-infra.cz/en

### 3.5.4 Results & Discussion

With chosen parameters, inputs, and hardware, we benchmark Parmesan using our dedicated experimental tool[14]. Since our main aim is to benchmark Parmesan on a highly multi-threaded processor, we accompany the code with scripts tailored for the *Portable Batch System* (PBS), which is a queuing system of many super-computing infrastructures, including e-INFRA CZ.

#### Observed Quantities

In our experiments, we primarily focus on the *total running time* (as per our Complexity Measure; cf. Section 3.3), and we also approximately measure the *processor load*. Besides that, we analytically evaluate other quantities:

- *(bootstrapping) circuit depth*: the minimal number of consecutive bootstraps in case of ideal parallelization,
- *total number of bootstraps (aka. #PBS)*,
- *ideal number of threads*: the minimum number of threads required for ideal parallelization, and
- *efficiency of CPU/thread usage*: the total number of available bootstrapping slots (i.e., the ideal number of threads multiplied by the circuit depth) divided by the total number of bootstraps called.

**Example 3.7.** *Let us evaluate these analytical quantities on an example of* 16-bit multiplication *(cf. Algorithm 6). The calculation spreads into three pools of threads to calculate the values of $A$, $B$, and $C$ (cf. lines 8–10), using two instances of 8-bit schoolbook multiplication for $A$ and $B$, and two 8-bit additions followed by a 9-bit schoolbook multiplication for $C$. For each 8-bit multiplication, we need 64 threads for pairwise multiplications, then we call 7× addition (cf. Figure 3.5), each with 8 active bits (i.e., 8 threads in 14 steps). For 9-bit multiplication, we need 81 threads (multiplication) followed by 16×9 threads (summation).*

*The calculation of $A$ and $B$ is followed by their addition (2×16 threads), then $A+B$ is subtracted from $C$ (2×18 threads) and finally $C-(A+B)$ is added to concatenated $A\|B$ (n.b., their length allows that; 2×24 threads). In total, we have 725 bootstraps in 23 steps and it shows that 81 threads are necessary & sufficient for the full parallelization; see Table 3.3, where columns $A$, $B$ and $C$ are later re-used for other calculations. Efficiency evaluates to* $^{725}/_{81\cdot 23} \approx 38.9\%$. *Find other bit-lengths, also for squaring, in Appendix D.*

*Remark* 3.4. In terms of circuit depth, Karatsuba shows to be worth starting from less than 16 bits, provided that a sufficient number of threads is available (cf. Remark .5 in Appendix C) and a careful thread scheduling is applied (similar to that of Table 3.3). Recall that the threshold for Karatsuba (given in Table 6 in Appendix C) is calculated with respect to the total number of bootstraps, not to the circuit depth. Let us provide an example for 10-bit inputs: we compare the parameters of Karatsuba and schoolbook in Table 3.4 (more details on 10-bit Karatsuba in Table 7 in Appendix D).

Although Karatsuba has a lower bootstrapping depth as well as requires a lower number of threads, such an approach requires careful parallelization (as per Table 7), which is currently out of the scope. Therefore, we implement the multiplication threshold $t_M$ as per Table 6. Also, a little experiment on a 12-threaded machine shows that 20-bit multiplication is faster with $t_M$ as proposed (i.e., 10- and 11-bit multiplications via schoolbook), compared to calling Karatsuba starting from 10-bit inputs, achieving 18.0 s and 20.8 s, respectively.

[14]https://github.com/fakub/bench-parmesan

Table 3.3: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 16-bit Karatsuba multiplication, which splits each input into two 8-bit parts. Using 81 threads in 23 steps, totalling 725 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| 64 | – | 8\|8 | 80 | $C$: $\mathbf{r}_1 + \mathbf{r}_0 \mid \mathbf{s}_1 + \mathbf{s}_0$ |
| – | 64 | 8\|8 | 80 | |
| – | – | 81 | **81** | $C$: 9-bit pairwise mul. |
| 8 | 8 | 9 | 25 | $A, B$: 8-bit schoolbook summation (14 rows); |
| 8 | 8 | 9 | 25 | |
| ⋮ | ⋮ | ⋮ | ⋮ | $C$: 9-bit scb. $\Sigma$ (+2 rows) |
| 8 | 8 | 9 | 25 | |
| – | 16 | 9 | 25 | $B$: $A + B$ |
| – | 16 | 9 | 25 | |
| – | – | 18 | 18 | $C$: $C - (A+B)$ |
| – | – | 18 | 18 | |
| – | – | 24 | 24 | $C$: $A \| B + (C-A-B) \| 0$ |
| – | – | 24 | 24 | |
| Total #PBS | | | 725 | |

Table 3.4: Parameters of 10-bit multiplication. For 8-bit, the depths become equal to 15.

| Algorithm | Depth | #PBS | #thr's |
|---|---|---|---|
| Karatsuba | **17** | 320 | **36** |
| Schoolbook | 19 | **280** | 100 |

### Experimental Results

We summarize the results of our benchmarks in Table 3.5, where one can find a performance comparison of Parmesan and Concrete `v0.2`$_\beta$ as well as the analytical quantities for Parmesan.

In Figure 3.8, we display approximate, per-thread processor load measured during a calculation of the maximum of 32-bit (encrypted) inputs. In that figure, one may spot the expected behavior of operations called within maximum (cf. Algorithm 10); one may observe high loads of:

1. 32 threads, 2 steps $\sim$ 32-bit subtraction (line 1 of that algorithm),
2. 9 threads, 1 step $\sim$ first step of signum's recursion (line 3; calls Algorithm 9 with a 33-bit input),
3. 3 threads, 1 step $\sim$ second step of signum's recursion,
4. 1 thread, 1 step $\sim$ the non-negativity function at the end of signum,
5. 32 threads, 1 step $\sim$ maximum selector (line 5).

Table 3.5: Benchmarking results of Parmesan vs. Concrete. Experimental machines with a 12-threaded Intel Core i7-7800X processor and with two 64-threaded AMD EPYC 7543 processors were used, in captions as "12-thr." and "128-thr.", respectively. Observed quantities are described in Section 3.5.4, "Speed-Up" gives the speed-up factor of Parmesan over Concrete on respective machine. For Concrete's addition/subtraction, the timings are amortized over three calls, since bootstrapping is only done during the third call; cf. Remark 3.3.

| | | **Parmesan** | | | | | | **Concrete** `v0.2`$_\beta$ | | **Speed-Up** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Operation | $n =$ #bits | Circ. depth | #PBS | Ideal #thr's | Eff. [%] | 12-thr. [ms] | 128-thr. [ms] | 12-thr. [ms] | 128-thr. [ms] | 12-thr. | 128-thr. |
| PBS | – | – | – | – | – | 110 | 140 | – | – | – | – |
| Add/Sub (Alg. 4, 4; amort. for Concrete) | 4 | 2 | $2n$ | $n$ | 100 | 220 | 420 | 270 | 480 | 1.2 | 1.1 |
| | 8 | | | | | 330 | 400 | 550 | 820 | 1.7 | 2.1 |
| | 16 | | | | | 570 | 390 | 1 150 | 1 310 | 2.0 | 3.4 |
| | 32 | | | | | 990 | 450 | 2 270 | 2 260 | 2.3 | 4.9 |
| Scalar Mul (Alg. 5) #bits = 16, val's of $k \to$ | 4 095 | 2 | $2n$ | 16 | 100 | 580 | 460 | 16 340 | 16 750 | 28 | 36 |
| | 4 096 | 0 | 0 | – | – | $\approx 0$ | $\approx 0$ | 1 720 | 1 630 | $\approx \infty$ | $\approx \infty$ |
| | 4 097 | 2 | $2n$ | 16 | 100 | 580 | 450 | 1 720 | 1 600 | 3.0 | 3.6 |
| | 805 | 6 | 114 | 22 | 86 | 1 900 | 1 350 | 7 560 | 7 380 | 4.0 | 5.2 |
| | 3 195 | 6 | 114 | 22 | 86 | 1 890 | 1 370 | 10 660 | 10 390 | 5.6 | 7.6 |
| Mul (Alg. 6; mod $2^n$ in Concrete) | 4 | 7 | 40 | 16 | 36 | 950 | 1 490 | 1 230 | 2 080 | 1.3 | 1.4 |
| | 8 | 15 | 176 | 64 | 18 | 3 330 | 3 290 | 4 600 | 5 390 | 1.4 | 1.6 |
| | 16 | 23 | 725 | 81 | 39 | 10 990 | 6 770 | 18 580 | 18 620 | 1.7 | 2.8 |
| | 32 | 41 | 2 617 | 289 | 22 | 38 440 | 15 260 | 72 160 | 72 780 | 1.9 | 4.8 |
| Squ (Alg. 8; mod $2^n$ in Concrete) | 4 | 5 | 24 | 5 | 96 | 650 | 1 020 | 1 230 | 2 020 | 1.9 | 2.0 |
| | 8 | 11 | 122 | 16 | 69 | 2 200 | 2 430 | 4 620 | 5 370 | 2.1 | 2.2 |
| | 16 | 19 | 488 | 64 | 40 | 7 920 | 5 090 | 18 560 | 18 490 | 2.3 | 3.5 |
| | 32 | 27 | 1 837 | $\leq 129$ | $\geq 53$ | 27 770 | 10 100 | 72 200 | 71 060 | 2.6 | 7.0 |
| Signum (Alg. 9) | 32 | 3 | 11 | 8 | 46 | 390 | 470 | (not implemented) | | – | – |
| Maximum (Alg. 10) | 32 | 6 | 109 | 32 | 57 | 1 880 | 1 370 | | | | |
| Rounding (at 5th bit, Alg. 11) | 32 | 4 | 56 | 27 | 52 | 1 140 | 1 030 | | | | |

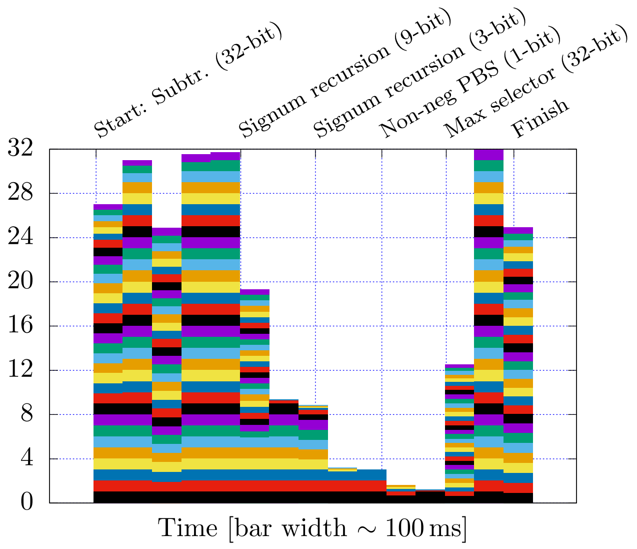


Figure 3.8: Approximate per-thread processor load measured during a calculation of 32-bit maximum on a 128-threaded supercomputer node. Other than the first 32 threads are omitted due to their negligible load.

## Discussion

In our benchmarks, we compare two different approaches for basic integer arithmetic over TFHE-encrypted data, implemented in our Parmesan Library and in the Concrete Library, respectively.

Parmesan's arithmetic is based on an algorithm for parallel addition, which uses a redundant integer representation, and in the encrypted domain, it employs a 5-bit message space to hold one bit of an encrypted integer. Parallel addition takes constant time (independent of input length), provided that a sufficient number of threads is available. On top of parallel addition, other arithmetic algorithms are implemented with particular respect to parallelization, including additional optimizations: ASC$^*$s and the window method for scalar multiplication, Karatsuba algorithm for multiplication, divide-and-conquer strategy for squaring, etc.

On the other hand, Concrete's arithmetics employ the standard sequential algorithm for addition. In certain sense, its integer representations are also redundant, as it allows buffering the carry for a limited number of additions, without bootstrapping. However, in the latest beta version of Concrete, there is neither parallelization nor any other optimization of basic arithmetic implemented yet. E.g., for scalar multiplication, this we can easily spot if we compare the results with our particular choices of inputs (cf. Section 3.5.3): namely for $k = 4\,096 = 4^6$, no bootstrapping would be needed in the base-4 representation (our setup of Concrete), however, the timing reveals that many bootstraps occur.

Although there are substantial differences between these two libraries, they are both built upon the same TFHE implementation (that of `concrete-core` crate), and they both use TFHE parameters provided in the `concrete-int` crate (i.e., we may expect the same "quality"). Hence, in certain sense, our benchmarks provide an insight into the direct comparison of these two approaches. E.g., for 16-bit squaring on the 12-threaded processor, Parmesan on the one hand achieves a speed-up by a factor of 2.3, on the other hand, it employs all of the 12 threads, as opposed to Concrete, which runs on a single thread only. However, as stated in Sections 3.5.1 and 3.5.2, neither of these libraries is in a production version, therefore, our results shall not be perceived as definitive.

Generally speaking, the redundant representation, required by parallel addition, introduces a certain computational overhead, which can be mitigated by a sufficient number of threads. Therefore, the choice of approach depends on the optimization goal: if minimizing the actual execution time is the priority, we recommend a parallel approach, if minimizing the computational cost (in terms of total CPU time) is the priority, we recommend a sequential approach.

## 3.6 Conclusion

We propose and implement parallel algorithms for a fast integer arithmetic over TFHE-encrypted data. We compare our library – the Parmesan Library – with the Concrete Library: for 32-bit inputs on an ordinary 12-threaded server processor, we observe speed-ups by a factor of 2.3 for addition, 1.9 for multiplication, and 2.6 for squaring. On a supercomputer's node with 128 threads, the speed-ups are even higher.

Particular speed-ups are achieved for scalar multiplication, for which we propose a new technique based on the window method and a special kind of addition chains denoted $\mathsf{ASC}^*$. E.g., the calculation of 4 095 times an encrypted 16-bit integer achieves a 28× speed-up over Concrete on the 12-threaded machine. The advantage of our technique is a combination of multiple factors that our method employs:

- *subtraction*, which goes at the same cost as addition,
- *free doubling* in the radix-based representation, and
- *pre-computed* $\mathsf{ASC}^*$*s* for up to 12-bit windows.

Besides integer arithmetic, we implement three signum-based operations: signum (also used for number comparison) and maximum, which go with significant optimizations compared to previous work [87], and rounding. These operations aim at completing the most common integer operations.

Thanks to the generic form of our algorithms, the underlying TFHE scheme might be easily replaced with another, LUT-based FHE scheme in the future.

#### Future Directions & Open Problems

For the Parmesan Library, there are several aspects to be considered:

- finalize the standard computer arithmetic by implementing:
  - bit operations (`&`, `|`, `!`, ...) and integer division,
  - conversions between the standard and the signed binary representation (if needed, e.g., for bit operations),
  - support for `(u)intXY`-like types, etc.
- algorithms for other common non-atomic operations (like maximum),
- other optimizations taking TFHE batching [35] into account,
- proper thread scheduling (cf. Remark 3.4) and dedicated optimizations with respect to a fixed number of threads,

- check how the (very recent) *FINAL Scheme* [17] compares to TFHE as the underlying FHE scheme.

From the theoretical point of view, the most interesting open problem is the possible NP-completeness of ASC$^*$s. For the moment, we also do not provide any argument of optimality for our ASC$^*$s, which we generated by a brute-force method for only up to 12-bit integers.

#### Comments on a Preliminary Comparison with `tfhe-rs`

Preliminary experiments show a (not so clear) dominance[15] of PARMESAN even over the most recent version of `tfhe-rs v0.2` [132] which is the state-of-the-art implementation of TFHE and TFHE-based integer arithmetic (as of Q2/2023). Compared to its predecessor Concrete `v0.2`$_\beta$, the new library `tfhe-rs` vastly improves the performance of not only bootstrapping (about 4×), but also it employs parallelism whenever possible[16] which makes it far more competitive than Concrete. However, the implicit non-parallelizability of its addition algorithm does not allow `tfhe-rs` for pushing the latency further down as soon as sufficient parallel resources are available, as opposed to PARMESAN.

## Appendix

## A Conversions Between Binary Representations

Recall that since $\mathcal{A}_2 \subset \bar{\mathcal{A}}_2$, no conversion is needed from the standard to the signed binary representation. Let us discuss the opposite conversion in more detail. First, note that in the signed binary, the leading bit determines the sign of the represented integer. Therefore, we outline this conversion only for positive integers; find it in Algorithm 12.

**Algorithm 12** Conversion from the signed to the standard binary representation (assuming a positive input).

**Input:** $(2, \bar{\mathcal{A}}_2)$-representation $\bar{\mathbf{x}} \in \bar{\mathcal{A}}_2{}^n$ of $X \in \mathbb{N}$, for some $n \in \mathbb{N}$ ($\bar{\mathbf{x}}$ has a positive leading bit),
**Output:** $(2, \mathcal{A}_2)$-representation $\mathbf{x} \in \mathcal{A}_2{}^n$ of $X$.

```
1: $\mathbf{x} \leftarrow \bar{\mathbf{x}}$
2: for $i = 0 \dots n-1$ do
3:     if $x_i < 0$ then
4:         $x_i \leftarrow x_i + 2$
5:         $x_{i+1} \leftarrow x_{i+1} - 1$
6:     end if
7: end for
8: return $\mathbf{x}$
```

For negative integers, we suggest two options:

1. We remember the negative sign, we flip all signs and we proceed with Algorithm 12 normally.

[15] Due to different operation bounds, results were compared based on estimates because they could not be compared directly.
[16] E.g., digit-by-digit multiplication can be parallelized, addition remains sequential with carry.

2. We extend the (negative) input by zeros to a pre-determined length $\bar{n} \geq n$ and we run Algorithm 12. This gives us a representation with leading $\bar{1}$ at the $\bar{n}$-th position, which we trim and we obtain $\mathbf{x}' \in \mathcal{A}_2{}^{\bar{n}}$. Such an $\mathbf{x}'$ is a standard complement code of length $\bar{n}$ for $X < 0$. E.g., for an $\bar{n} = 8$-bit representation, we have $(\bar{1}0\bar{1}0\bar{1}0\bullet)_2 \xrightarrow{Alg.\ 12} (\bar{1}11010110\bullet)_2 \sim$ `(int8_t)(0b11'01'01'10)` $= -42$.

Also, recall that for the "opposite" conversion, there does not exist a parallel algorithm. Indeed, the existence of such a parallel algorithm would allow the following scenario: one may convert (trivially) from the standard to the signed binary, perform addition, and convert back, all in parallel. This contradicts the impossibility of parallel addition over a non-redundant integer representation (in this example the standard binary), as shown by Kornerup [91].

## B Average Number of Additions in Scalar Multiplication

We evaluate the average number of additions over 12-bit windows in the Koyama-Tsuruoka recoding (recall that the Koyama-Tsuruoka recoding achieves minimal Hamming weight) using:

1. our 12-bit $\mathsf{ASC}^*$s, and
2. the standard double-and-add/sub method.

E.g., for $k = 885$, Koyama-Tsuruoka recoded as $\mathtt{0b100\bar{1}000\bar{1}0\bar{1}\bar{1}}$: the $\mathsf{ASC}^*$ is $(1, 7 = -1 + 1 \cdot 2^3, 223 = -1 + 7 \cdot 2^5, 885 = -7 + 223 \cdot 2^2)$, which requires 3 additions, whereas the standard approach gives $(1, 7 = -1 + 1 \cdot 2^3, 111 = -1 + 7 \cdot 2^4, 443 = -1 + 111 \cdot 2^2, 885 = -1 + 443 \cdot 2^1)$, which requires 4 additions.

First, we evaluate the Koyama-Tsuruoka recoding for all odd integers in the interval $[1, 4\,095]$, we trim them to 12 LSBs and we take the absolute value, obtaining 1 792 unique values. Then, we evaluate both $\mathsf{ASC}^*$s and double-and-add/sub, totalling 5 558 and 6 956 additions, respectively. For simplicity, we assume that those 1 792 values are uniformly distributed. This gives us an average of 3.10 additions for $\mathsf{ASC}^*$s and 3.88 additions for double-and-add/sub.

## C Threshold for Karatsuba Algorithm

Karatsuba algorithm has a lower asymptotic complexity than the schoolbook algorithm ($O(n^{\log 3})$ vs. $O(n^2)$), however, for short inputs, schoolbook is better. Hence, the aim is to derive threshold $t_M$, under which the schoolbook algorithm outperforms Karatsuba. In the clear as well as in the encrypted domain, the threshold $t_M$ needs to be evaluated with respect to the complexity of all involved operations in that domain.

*Remark* .5. As outlined in Remark 3.2, it is important to choose a complexity measure. For algorithms like parallel addition/subtraction, we aim at achieving the *lowest bootstrapping depth* – they require as many threads as the number of bits, which makes this choice reasonable with existing (highly) multi-threaded CPUs.

However, for multiplication, parallelization is enormous in the first couple of steps and then it drops down rapidly. With a limited number of threads, we are closer to the "single-threaded" scenario, where we aim at minimizing the *total number of bootstraps*, which we suggest to apply for multiplication.

Both schoolbook and Karatsuba multiplication can be constructed using just *addition* (cf. Algorithm 4) and *single-bit multiplication* (cf. Algorithm 7), while *addition* requires $A = 2$

Table 6: Bootstrapping complexity of the schoolbook (Scb.) and the Karatsuba (Kar.) multiplication algorithms with different input bit-lengths (Bits). In Karatsuba, we consider splitting odd numbers such that the longer part is at LSB (cf. Section 3.3.3, item 4).

| Bits | ... | 8 | ... | 14 | 15 | 16 | 17 | 18 | 19 | ... | 32 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Scb. | **...** | **176** | **...** | **560** | **645** | 736 | **833** | 936 | 1 045 | ... | 3 008 | ... |
| Kar. | ... | 221 | ... | 572 | 678 | **725** | 843 | **896** | **1 026** | **...** | **2 617** | **...** |

bootstraps (per bit), whereas *single-bit multiplication* requires $M = 1$ bootstrap. We evaluate the bootstrapping complexity of the schoolbook algorithm for $n$-bit inputs as

$$B_{\times}^{(s)}(n) = M \cdot n^2 + A \cdot n \cdot (n-1) = 3n^2 - 2n. \tag{12}$$

We use this result to evaluate the complexity of a first-level Karatsuba, which is sufficient to find the threshold $t_M$ – indeed, recursion emerges for longer inputs, where also the halved input is longer than $t_M$. We observe that Karatsuba outperforms the schoolbook algorithm starting from around $n = 16$-bit inputs; find the concrete bootstrapping complexities in Table 6. It shows that there is no single threshold $t_M$, instead, there is a slight overlap.

## D Thread Scheduling

We suggest thread scheduling for a potential 10-bit Karatsuba multiplication in Table 7. Next, we suggest thread scheduling for 32-bit multiplication and for 4-, 8-, 16- and 32-bit squarings in Tables 8, 9, 10, 11 and 12, respectively. Note that 16-bit multiplication is covered in Table 3.3.

Table 7: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in potential 10-bit Karatsuba multiplication (intentionally under the threshold $t_M$ to compare with schoolbook), splitting the input into two 5-bit parts. Using 36 threads in 17 steps, totalling 320 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| 25 | – | 5\|5 | 35 | $C$: $\mathbf{r}_1 + \mathbf{r}_0 \mid \mathbf{s}_1 + \mathbf{s}_0$ |
| – | 25 | 5\|5 | 35 | |
| – | – | 36 | **36** | $C$: 6-bit pairwise mul. |
| 5 | 5 | 6 | 16 | $A, B$: 5-bit schoolbook summation (8 rows); $C$: 6-bit scb. $\Sigma$ (+2 rows) |
| 5 | 5 | 6 | 16 | |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| 5 | 5 | 6 | 16 | |
| – | 10 | 6 | 16 | $B$: $A + B$ |
| – | 10 | 6 | 16 | |
| – | – | 12 | 12 | $C$: $C - (A{+}B)$ |
| – | – | 12 | 12 | |
| – | – | 15 | 15 | $C$: $A \Vert B + (C{-}A{-}B) \Vert 0$ |
| – | – | 15 | 15 | |
| Total #PBS | | | 320 | |

Table 8: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 32-bit Karatsuba multiplication, splitting the input into two 16-bit parts. Note that $C$ is calculated via 17-bit schoolbook algorithm; cf. Table 6. Using 289 threads in 41 steps, totalling 2 617 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| – | – | 16\|16 | 32 | $C$: $\mathbf{r}_1 + \mathbf{r}_0 \mid \mathbf{s}_1 + \mathbf{s}_0$ |
| – | – | 16\|16 | 32 | |
| – | – | 289 | **289** | $C$: 17-bit pairwise mul. |
| 80 | 80 | 17 | 177 | $A, B$: 16-bit Karatsuba (23 rows); |
| 80 | 80 | 17 | 177 | |
| ⋮ | ⋮ | ⋮ | ⋮ | $C$: 17-bit scb. $\Sigma$ (+9 rows) |
| 24 | 24 | 17 | 65 | |
| – | 33 | 17 | 50 | $B$: $A + B$ |
| – | 33 | 17 | 50 | |
| – | – | 17 | 17 | |
| ⋮ | ⋮ | ⋮ | ⋮ | |
| – | – | 17 | 17 | |
| – | – | 34 | 34 | $C$: $C - (A{+}B)$ |
| – | – | 34 | 34 | |
| – | – | 35 | 35 | $C$: $(C{-}A{-}B)\|0 + B$ |
| – | – | 35 | 35 | |
| – | – | 33 | 33 | $C$: $\ldots + A\|0$ |
| – | – | 33 | 33 | |
| Total #PBS | | | 2 617 | |

## E Algorithm for Squaring of Short Inputs

We provide the full algorithm for squaring of short inputs as Algorithm 13. Note that columns of LUTs in that algorithm hold squares of respective selector in binary: e.g., the 5th column contains 100110, which is a (reversed) binary representation of $5^2 = 25$. The bit at $2^1$ position (i.e., $y_1$) is always zero thanks to the fact $a^2 \bmod 4 \in \{0, 1\}$. Due to line 1, we need $2^{2\Delta} \geq (2^2)^2 + (2^1)^2 + (2^0)^2 = 21$.

Table 9: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 4-bit Divide & Conquer squaring. Using 5 threads in 5 steps, totalling 24 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| – | – | 4 | 4 | $C$: 2-bit pairwise mul. |
| 3 | – | 2 | **5** | $A, B$: 2-bit squ. (direct); |
| – | 3 | 2 | **5** | $C$: 2-bit scb. $\Sigma$ |
| – | – | 5 | **5** | $C$: $A\|B + C\|0$ |
| – | – | 5 | **5** | |
| Total #PBS | | | 24 | |

Table 10: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 8-bit Divide & Conquer squaring. Using 16 threads in 11 steps, totalling 122 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| – | – | 16 | **16** | $C$: 4-bit pairwise mul. |
| 4 | 4 | 4 | 12 | |
| 5 | 5 | 4 | 14 | $A, B$: 4-bit D'n'Q |
| $\vdots$ | $\vdots$ | $\vdots$ | $\vdots$ | (5 rows); $C$: 4-bit scb. $\Sigma$ (+1 row) |
| 5 | 5 | 4 | 14 | |
| – | – | 4 | 4 | |
| – | – | 8 | 8 | $C$: $C\|0 + B$ |
| – | – | 8 | 8 | |
| – | – | 9 | 9 | $C$: $\ldots + A\|0$ |
| – | – | 9 | 9 | |
| Total #PBS | | | 122 | |

**Algorithm 13** Short-Input Squaring.

**Input:** 3-bit representation $(x_2x_1x_0\bullet)_2 \in (\bar{\mathcal{A}}_2)^3$ of $X \in \mathbb{Z}$,
**Output:** 6-bit representation $(y_5 \ldots y_0\bullet)_2 \in (\bar{\mathcal{A}}_2)^6$ of $X^2$.

```
1: $X \leftarrow \mathsf{eval}_2(x_2x_1x_0\bullet)$        ▷ no bootstrap needed; cf. (3.6)
2: in parallel do
3:     $y_0 \leftarrow (0,1,0,1,0,1,0,1 \| 1,0,1,0,1,0,1)[X]$
4:     $y_1 \leftarrow 0$        ▷ always 0
5:     $y_2 \leftarrow (0,0,1,0,0,0,1,0 \| 0,1,0,0,0,1,0)[X]$
6:     $y_3 \leftarrow (0,0,0,1,0,1,0,0 \| 0,0,1,0,1,0,0)[X]$
7:     $y_4 \leftarrow (0,0,0,0,1,1,0,1 \| 1,0,1,1,0,0,0)[X]$
8:     $y_5 \leftarrow (0,0,0,0,0,0,1,1 \| 1,1,0,0,0,0,0)[X]$
9: end parallel
10: return $(y_5 \ldots y_0\bullet)_2$
```

Table 11: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 16-bit Divide & Conquer squaring. Using 64 threads in 19 steps, totalling 488 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| – | – | 64 | **64** | $C$: 8-bit pairwise mul. |
| 16 | 16 | 8 | 40 | |
| ⋮ | ⋮ | ⋮ | ⋮ | $A, B$: 8-bit D'n'Q |
| 9 | 9 | 8 | 26 | (11 rows); |
| – | – | 8 | 8 | $C$: 8-bit scb. $\Sigma$ |
| – | – | 8 | 8 | |
| – | – | 8 | 8 | |
| – | – | 16 | 16 | $C$: $C\|0 + B$ |
| – | – | 16 | 16 | |
| – | – | 18 | 18 | $C$: $\ldots + A\|0$ |
| – | – | 18 | 18 | |
| Total #PBS | | | 488 | |

Table 12: A suggestion of thread scheduling for the calculation of intermediate values $A$, $B$, and $C$, followed by their aggregation, in 32-bit Divide & Conquer squaring. Using 129 threads in 27 steps, totalling 1 837 bootstraps.

| $A$ | $B$ | $C$ | Total #thr's | *Comment* |
|---|---|---|---|---|
| – | – | 80 | 80 | |
| – | – | 80 | 80 | |
| – | – | 81 | 81 | $A, B$: 16-bit D'n'Q |
| – | 64 | 25 | 89 | (19 rows); |
| 64 | 40 | 25 | **129** | $C$: 16-bit Karatsuba |
| ⋮ | ⋮ | ⋮ | ⋮ | (23 rows) |
| 18 | 18 | 24 | 60 | |
| 18 | – | 24 | 42 | |
| – | – | 33 | 33 | $C$: $C\|0 + B$ |
| – | – | 33 | 33 | |
| – | – | 35 | 35 | $C$: $\ldots + A\|0$ |
| – | – | 35 | 35 | |
| Total #PBS | | | 1 837 | |

Chapter 4

# A Practical TFHE-Based Multi-Key Homomorphic Encryption

Fully Homomorphic Encryption enables arbitrary computations over encrypted data and it has a multitude of applications, e.g., secure cloud computing in healthcare or finance. Multi-Key Homomorphic Encryption (MKHE) further allows to process encrypted data from multiple sources: the data can be encrypted with keys owned by different parties. In this chapter, we propose a new variant of MKHE instantiated with the TFHE scheme. Compared to previous attempts by Chen et al. and by Kwak et al., our scheme achieves computation runtime that is linear in the number of involved parties and it outperforms the faster scheme by a factor of 4.5-6.9$\times$, at the cost of a slightly extended pre-computation. In addition, for our scheme, we propose and practically evaluate parameters for up to 128 parties, which enjoy the same estimated security as parameters suggested for the previous schemes (100 bits). It is also worth noting that our scheme—unlike the previous schemes—did not experience *any* error in any of our nine setups, each running 1 000 trials.

## 4.1 Introduction to MKHE

For the first time publicly discovered in 2009 by Gentry [58], *Fully Homomorphic Encryption* (FHE) refers to a cryptosystem that allows for an evaluation of an arbitrary computable function over encrypted data[1]. With FHE, a secure cloud-aided computation, between a user (**U**) and a semi-trusted cloud (**C**), may proceed as follows:

- **U** generates secret keys sk, and evaluation keys ek, which she sends to **C**;
- **U** encrypts her sensitive data $d$ with sk, and sends the encrypted data to **C**;
- **C** employs ek to evaluate function $f$, homomorphically, over the encrypted user data (without ever decrypting it), yielding an encryption of $f(d)$, which it sends back to **U**;
- **U** decrypts the message from **C** with sk, obtaining the desired result: $f(d)$ in plain.

[1]For a basic overview of the evolution of FHE schemes, we refer to a survey by Acar et al. [2] (from 2018; note that for implementations, much progress has been made since then).

In such a setup, there is one party that holds all the secret keying material. In case the data originate from multiple sources, *Multi-Key (Fully) Homomorphic Encryption* (MKHE) comes into play. First proposed by López-Alt et al. [97], MKHE is a primitive that enables the homomorphic evaluation over data encrypted with multiple different, unrelated keys. This allows to relax the intrinsic restriction of a standard FHE, which demands a single data owner.

### Previous Work

Following the seminal work of López-Alt et al. [97], different approaches to design an MKHE scheme have emerged: first attempts require a fixed list of parties at the beginning of the protocol [40, 102], others allow parties to join dynamically [24, 110], Chen et al. [29] extend the plaintext space from a single bit to a ring. Later, Chen et al. [27] propose an MKHE scheme based on the TFHE scheme [35], and they claim to be the first to practically implement an MKHE scheme; in this chapter, we refer to their scheme as CCS. The evaluation complexity of their scheme is quadratic in the number of parties and authors only run experiments with up to 8 parties. The CCS scheme is improved in recent work by Kwak et al. [93], who achieve quasi-linear complexity (actually quadratic, but with a very low coefficient at the quadratic term); in this chapter, we refer to their scheme as KMS. Parallel to CCS and KMS, which are both based on TFHE, there exist other promising schemes: e.g., [28], defined for BFV [22, 53] and CKKS [33], improved in [79] to achieve linear complexity, or [100], implemented in the Lattigo Library [101], which requires to first construct a common public key; also referred to as the *Multi-Party HE* (MPHE). The capabilities/use-cases of TFHE and other schemes are fairly different, therefore we solely focus on the comparison of TFHE-based MKHE.

### Our Contributions

We propose a new TFHE-based MKHE scheme with a linear evaluation complexity and with a sufficiently low error rate, which allows for a practical instantiation with an order of hundreds of parties while achieving evaluation times proportional to those of plain TFHE. More concretely, our scheme builds upon the following technical ideas ($k$ is the number of parties):

**Summation of RLWE keys:** Instead of *concatenation* of RLWE keys (in certain sense proposed in both CCS and KMS), our scheme works with RLWE encryptions under the *sum* of RLWE keys of individual parties, i.e., $Z = \sum_{q=1}^{k} z^{(q)}$. As a result, this particular improvement decreases the evaluation complexity from quadratic to linear.

**Ternary distribution for RLWE keys:** Widely adopted by existing FHE implementations [67, 101, 99, 121], zero-centered ternary distribution $\zeta\colon (-1, 0, 1) \to (p, 1 - 2p, p)$ works well as a distribution of the coefficients of RLWE keys; we suggest $p \approx 0.1135$. It helps reduce the growth of a certain noise term by a factor of $k$, which in turn helps find more efficient TFHE parameters.

**Avoid FFT in pre-computations:** In our experiments, we notice an unexpected error growth for higher numbers of parties and we verify that the source of these errors is Fast Fourier Transform (FFT), which is used for fast polynomial multiplication. To keep the evaluation times low and to decrease the number of errors at the same time, we suggest replacing FFT with an exact method just in the pre-computation phase. We also show that FFT causes a considerable amount of errors in KMS, however, replacing FFT in its pre-computations is unfortunately not sufficient.

We provide two variants of our scheme:

**Static variant:** the list of parties is fixed – the evaluation cost is independent of the number of parties who provide their inputs and the result is encrypted with all keys; and

**Dynamic variant:** the computation cost is proportional to the number of participating parties, and the result is only encrypted with their keys (i.e., any subset of parties can go offline).

The variants only differ in pre-computation algorithms which in turn affect security assumptions. Performance-wise, given a fixed number of parties, the variants are equivalent (it only depends on the parameters of TFHE) and the evaluation complexity is linear in the number of involved parties. The construction of our scheme remains similar to that of plain TFHE, making it possible to adopt prospective advances of TFHE (or its implementation) to our scheme.

Next, we analyze and practically evaluate our scheme, and we compare it with previous attempts:

- We support our scheme by a theoretical noise-growth & security analysis. Thanks to the low noise growth, we instantiate our scheme with as many as 128 parties. We also show that our scheme is secure in the semi-honest model. In addition, we informally outline possible countermeasures in case there is a malicious party;
- We design and evaluate a deep experimental study, which may help evaluate future schemes. In particular, we suggest simulating the NAND gate to measure errors more realistically. Compared to the KMS scheme, we achieve 4.5-6.9× better bootstrapping times, while using the same implementation of TFHE and parameters with the same estimated security (100 bits). The bootstrapping times are around 140 ms per party (with an experimental implementation);
- We extend previous work by providing an experimental evaluation of the probability of errors. For our scheme, the measured noises fall within the expected bounds, which are designed to satisfy the rule of $4\sigma$ (probability of 1 in 15 787) – we indeed do not encounter *any* error in any of our 9 000 trials in total.

### Chapter Outline

We recall the TFHE scheme in a form of a detailed technical description in Section 4.2 and we present our scheme in Section 4.3. We analyze security, correctness & noise growth, and performance of our scheme in Section 4.4, which is followed by a thorough experimental evaluation in Section 4.5. We conclude this chapter in Section 4.6.

## 4.2 Preliminaries

In this section, we recall the basic variant of the TFHE scheme [35]. Later in this chapter, we refer to some of the algorithms and/or definitions.

### Symbols & Notation

Throughout this chapter, we use the following symbols & notation:

- $\mathbb{B}$: the set of binary coefficients $\{0,1\} \subset \mathbb{Z}$,
- $\mathbb{T}$: the additive group $\mathbb{R}/\mathbb{Z}$ referred to as the *torus* (i.e., real numbers modulo 1),
- $\mathbb{Z}_n$: the quotient ring $\mathbb{Z}/n\mathbb{Z}$ (or its additive group),

- $M^{(N)}[X]$: the set of polynomials modulo $X^N+1$, with coefficients from $M$ and with $N \in \mathbb{N}$,
- $\$$: the uniform distribution,
- $a \overset{\alpha}{\leftarrow} M$: the draw of random variable $a$ from $M$ with distribution $\alpha$ (for $\alpha \in \mathbb{R}$, we consider the zero-centered /discrete/ Gaussian draw with standard deviation $\alpha$),
- $\mathrm{E}[X]$, $\mathsf{Var}[X]$: the expectation and the variance of random variable $X$, respectively.

We use logarithm base 2 throughout this chapter.

### 4.2.1 Description of TFHE, Revisited

For convenience, we revisit the TFHE scheme in this chapter in a more technical way than in Chapter 1, so that our newly proposed algorithms can directly refer to this technical description of the base-line TFHE. In the rest of this chapter, we focus solely on the basic variant of TFHE with a Boolean message space: true and false are encoded into $\mathbb{T} \sim [-1/2, 1/2)$ as $-1/8$ and $1/8$, respectively. To homomorphically evaluate the NAND gate over input samples $\mathbf{c}_{1,2}$, the sum $(1/8, \mathbf{0}) - \mathbf{c}_1 - \mathbf{c}_2$ is bootstrapped with a LUT, which holds $1/8$ and $-1/8$ for the positive and for the negative half of $\mathbb{T}$, respectively.

Below, we provide a technical description of the TFHE scheme in a form of self-descriptive algorithms. Parameters and secret keys are considered implicit inputs.

∘ `TFHE.Setup`$(1^\lambda)$: Given security parameter $\lambda$, generate parameters for:

- LWE encryption: dimension $n$, standard deviation $\alpha > 0$ (of the noise);
- LWE decomposition: base $B'$, depth $d'$;
- set up LWE gadget vector: $\mathbf{g}' \leftarrow (1/B', 1/B'^2, \ldots, 1/B'^{d'})$;
- RLWE encryption: polynomial degree $N$ (a power of two), standard deviation $\beta > 0$;
- RLWE decomposition: base $B$, depth $d$;
- set up RLWE gadget vector: $\mathbf{g} \leftarrow (1/B, 1/B^2, \ldots, 1/B^d)$.

Other input parameters of the `Setup` algorithm may include the maximal allowed probability of error, or the plaintext space size (for other than Boolean circuits).

∘ `TFHE.SecKeyGen`$()$: Generate secret keys for:

- LWE encryption: $\mathbf{s} \overset{\$}{\leftarrow} \mathbb{B}^n$;
- RLWE encryption: $z \overset{\$}{\leftarrow} \mathbb{B}^{(N)}[X]$, (alternatively $z_i \overset{\zeta}{\leftarrow} \{-1, 0, 1\}$ for some distribution $\zeta$).

For LWE key $\mathbf{s} \in \mathbb{B}^n$, we denote $\bar{\mathbf{s}} := (1, \mathbf{s}) \in \mathbb{B}^{1+n}$ the extended secret key, similarly for an RLWE key $z \in \mathbb{Z}^{(N)}[X]$, we denote $\bar{\mathbf{z}} := (1, z) \in \mathbb{Z}^{(N)}[X]^2$.

∘ `TFHE.LweSymEncr`$(\mu)$: Given message $\mu \in \mathbb{T}$, sample fresh mask $\mathbf{a} \overset{\$}{\leftarrow} \mathbb{T}^n$ and noise $e \overset{\alpha}{\leftarrow} \mathbb{T}$. Evaluate $b \leftarrow -\langle \mathbf{s}, \mathbf{a} \rangle + \mu + e$ and output $\bar{\mathbf{c}} = (b, \mathbf{a}) \in \mathbb{T}^{1+n}$, an LWE encryption of $\mu$. This algorithm is used as the main encryption algorithm of the scheme. We generalize this as well as subsequent algorithms to input vectors and proceed element-by-element.

∘ $\underline{\mathsf{TFHE}.\mathtt{RLweSymEncr}(m, a = \emptyset, z_{in} = z)}$: Given message $m \in \mathbb{T}^{(N)}[X]$, sample fresh mask $a \xleftarrow{\$} \mathbb{T}^{(N)}[X]$, unless explicitly given. If the pair $(a, z_{in})$ has been used before, output $\perp$. Otherwise, sample fresh noise $e \in \mathbb{T}^{(N)}[X]$, $e_i \xleftarrow{\beta} \mathbb{T}$, and evaluate $b \leftarrow -z_{in} \cdot a + m + e$. Output $\bar{\mathbf{c}} = (b, a) \in \mathbb{T}^{(N)}[X]^2$, an RLWE encryption of $m$. In case $a$ is given, we may limit the output to only $b$.

∘ $\underline{\mathsf{TFHE}.\mathtt{(R)LwePhase}(\bar{\mathbf{c}})}$: Given (R)LWE sample $\bar{\mathbf{c}}$, evaluate and output $\varphi \leftarrow \langle \bar{\mathbf{s}}, \bar{\mathbf{c}} \rangle$, where $\bar{\mathbf{s}}$ is respective (R)LWE extended secret key.

∘ $\underline{\mathsf{TFHE}.\mathtt{EncrBool}(b)}$: Set $\mu = \pm 1/8$ for $b$ true or false, respectively. Output $\mathtt{LweSymEncr}(\mu)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{DecrBool}(\bar{\mathbf{c}})}$: Output $\mathtt{LwePhase}(\bar{\mathbf{c}}) > 0$, assuming $\mathbb{T} \sim [-1/2, 1/2)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{RgswEncr}(m)}$: Given $m \in \mathbb{Z}^{(N)}[X]$, evaluate $\mathbf{Z} \leftarrow \mathtt{RLweSymEncr}(\mathbf{0})$, where $\mathbf{0}$ is a vector of $2d$ zero polynomials (i.e., $\mathbf{Z} \in (\mathbb{T}^{(N)}[X])^{2d \times 2}$). Output $\mathbf{Z} + m \cdot \mathbf{G}$, an RGSW sample of $m$.

∘ $\underline{\mathsf{TFHE}.\mathtt{Prod}\big(\mathsf{BK}, (b, a)\big)}$: Given RGSW sample $\mathsf{BK}$ of $s \in \mathbb{Z}^{(N)}[X]$, and RLWE sample $(b, a)$ of $m \in \mathbb{T}^{(N)}[X]$, evaluate and output:

$$(b', a') \leftarrow \begin{pmatrix} \mathbf{g}^{-1}(b) \\ \mathbf{g}^{-1}(a) \end{pmatrix}^T \cdot \mathsf{BK} =: \mathsf{BK} \boxdot (b, a), \tag{4.1}$$

which is an RLWE sample of $s \cdot m \in \mathbb{T}^{(N)}[X]$; in TFHE also referred to as the *external product*.

∘ $\underline{\mathsf{TFHE}.\mathtt{BlindRotate}\big(\bar{\mathbf{c}}, \{\mathsf{BK}_i\}_{i=1}^n, tv\big)}$: Given $\bar{\mathbf{c}} = (b, a_1, \ldots, a_n) \in \mathbb{T}^{1+n}$, an LWE sample of $\mu \in \mathbb{T}$ under key $\mathbf{s} \in \mathbb{B}^n$; $(\mathsf{BK}_i)_{i=1}^n$, RGSW samples of $\mathbf{s}_i$ under RLWE key $z$ (aka. *blind-rotate keys*); and $\mathsf{RLWE}_z(tv) \in \mathbb{T}^{(N)}[X]^2$, (usually trivial) RLWE sample of $tv \in \mathbb{T}^{(N)}[X]$ (aka. *test vector*), evaluate:

```
1: $\tilde{b} \leftarrow \lfloor 2Nb \rceil, \quad \tilde{a}_i \leftarrow \lfloor 2Na_i \rceil$ for $1 \le i \le n$
2: $\mathsf{ACC} \leftarrow X^{\tilde{b}} \cdot \mathsf{RLWE}(tv)$
3: for $i = 1, \ldots, n$ do
4:     $\mathsf{ACC} \leftarrow \mathsf{ACC} + \mathtt{Prod}\big(\mathsf{BK}_i, X^{\tilde{a}_i} \cdot \mathsf{ACC} - \mathsf{ACC}\big)$   ▷ $\mathsf{ACC}$ or $X^{\tilde{a}_i} \cdot \mathsf{ACC}$ if $\mathbf{s}_i = 0$ or $\mathbf{s}_i = 1$, resp.
5: end for
```

Output $\mathsf{ACC} = \mathsf{RLWE}_z(X^{\tilde{\varphi}} \cdot tv)$, an RLWE encryption of test vector "rotated" by $\tilde{\varphi}$, where $\tilde{\varphi} = \lfloor 2Nb \rceil + s_1 \lfloor 2Na_1 \rceil + \ldots + s_n \lfloor 2Na_n \rceil \approx 2N(\bar{\mathbf{s}} \cdot \bar{\mathbf{c}}) \approx 2N\mu$.

∘ $\underline{\mathsf{TFHE}.\mathtt{KeyExtract}(z)}$: Given RLWE key $z \in \mathbb{Z}^{(N)}[X]$, output $\mathbf{z}^* \leftarrow (z_0, -z_{N-1}, \ldots, -z_1)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{SampleExtract}(b, a)}$: Given RLWE sample $(b, a) \in \mathbb{T}^{(N)}[X]^2$ of $m \in \mathbb{T}^{(N)}[X]$ under RLWE key $z \in \mathbb{Z}^{(N)}[X]$, output LWE sample $(b', \mathbf{a}') \leftarrow (b_0, a_1, \ldots, a_N) \in \mathbb{T}^{1+N}$ of $m_0 \in \mathbb{T}$ (the constant term of $m$) under the extracted LWE key $\mathbf{z}^* = \mathtt{KeyExtract}(z)$.

∘ $\underline{\mathsf{TFHE}.\mathtt{KeySwitchKeyGen}()}$: For $j \in [1, N]$, evaluate and output a key-switching key for $z_j$ and $\mathbf{s}$: $\mathsf{KS}_j \leftarrow \mathtt{LweSymEncr}\big(\mathbf{z}_j^* \cdot \mathbf{g}'\big)$, where $\mathbf{z}^* \leftarrow \mathtt{KeyExtract}(z)$. $\mathsf{KS}_j$ is a $d'$-tuple of LWE samples of $\mathbf{g}'$-respective fractions of $\mathbf{z}_j^*$ under the key $\mathbf{s}$.

○ $\underline{\texttt{TFHE.KeySwitch}\big(\bar{\mathbf{c}}', \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given LWE sample $\bar{\mathbf{c}}' = (b', a'_1, \dots, a'_N) \in \mathbb{T}^{1+N}$ (extraction of an RLWE sample), which encrypts $\mu \in \mathbb{T}$ under the extraction of an RLWE key $\mathbf{z}^* = \texttt{KeyExtract}(z)$, and a set of key-switching keys for $z$ and $\mathbf{s}$, evaluate and output

$$\bar{\mathbf{c}}'' \leftarrow (b', \mathbf{0}) + \sum_{j=1}^{N} \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j, \tag{4.2}$$

which is an LWE sample of the same $\mu \in \mathbb{T}$ under the LWE key $\mathbf{s}$.

○ $\underline{\texttt{TFHE.Bootstrap}\big(\bar{\mathbf{c}}, tv, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given LWE sample $\bar{\mathbf{c}}$ of $\mu \in \mathbb{T}$ under LWE key $\mathbf{s}$, test vector $tv \in \mathbb{T}^{(N)}[X]$ that encodes a LUT, and two sets of keys for blind-rotate and for key-switching (aka. *bootstrapping keys* – the evaluation keys of TFHE), evaluate:

```
1: $\bar{\mathbf{c}}' \leftarrow \texttt{BlindRotate}\big(\bar{\mathbf{c}}, \{\mathsf{BK}_i\}_{i=1}^n, tv\big)$;
2: $\bar{\mathbf{c}}'' \leftarrow \texttt{KeySwitch}\big(\texttt{SampleExtract}(\bar{\mathbf{c}}'), \{\mathsf{KS}_j\}_{j=1}^N\big)$.
```

Output $\bar{\mathbf{c}}''$, which is an LWE sample of—vaguely speaking—"evaluation of the LUT at $\mu$", under the key $\mathbf{s}$, with a refreshed noise. Details on the encoding of the LUT are given in Chapter 1.

○ $\underline{\texttt{TFHE.Add}(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2)}$: Output $\bar{\mathbf{c}}_1 + \bar{\mathbf{c}}_2$, which encrypts the sum of input plaintexts. Using just "+".

○ $\underline{\texttt{TFHE.NAND}\big(\bar{\mathbf{c}}_1, \bar{\mathbf{c}}_2, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)}$: Given encryptions of bools $b_1$ and $b_2$ under LWE key $\mathbf{s}$, and bootstrapping keys for $\mathbf{s}$ and $z$, set the test vector as $tv \leftarrow 1/8 \cdot (1 + X + X^2 + \ldots + X^{N-1})$. Output $\bar{\mathbf{c}}'' \leftarrow \texttt{Bootstrap}\big(1/8 - \bar{\mathbf{c}}_1 - \bar{\mathbf{c}}_2, tv, \{\mathsf{BK}_i\}_{i=1}^n, \{\mathsf{KS}_j\}_{j=1}^N\big)$, which is an encryption of $\neg(b_1 \wedge b_2)$ under the key $\mathbf{s}$.

## 4.3 Our TFHE-Based Multi-Key Scheme

In this section, we first recall the notion of Multi-Key Homomorphic Encryption (MKHE) and we propose two variants of MKHE. Then, we summarize ideas and changes that lead from the standard TFHE scheme [35] towards our proposal of MKHE – we outline the format of multi-key bootstrapping keys, and we comment on a ternary distribution for RLWE keys. Finally, we provide a technical description of our scheme, which we denote AKÖ (by authors' initials).

### 4.3.1 MKHE and Our Variants

In addition to the capabilities of a standard FHE scheme, given in the introduction, an MKHE scheme:

(i) runs a homomorphic evaluation over ciphertexts encrypted with unrelated keys of multiple parties (accompanied by corresponding evaluation keys); and

(ii) requires the collaboration of all involved parties, holding the individual keys, to decrypt the result.

Note that there exist multiple approaches to reveal the result: e.g., one outlined in [27], referred to as *Distributed Decryption*, or one described in [100], referred to as *Collective Public-Key Switching*.

We propose our scheme in two variants:

**Static variant:** the list of parties is fixed at the beginning of the protocol, then evaluation keys are jointly calculated – no matter how many parties join a computation, the evaluation time is also fixed and the result is encrypted with all the keys; and

**Dynamic variant:** after a "global" list of parties is fixed, evaluation keys are jointly calculated, however, only a subset of parties may join a computation – the evaluation cost is proportional to the size of the subset and the result is only encrypted with respective keys (i.e., the remaining parties can go offline). If a party joins later, a part of the joint pre-calculation of evaluation keys needs to be executed in addition, as opposed to CCS [27] and KMS [93].

Note that in many practical use cases—in particular, if we require semi-honest parties—the (global) list of parties is fixed, e.g., hospitals may constitute the parties. In addition, the pre-calculation protocol is indeed lightweight.

### 4.3.2 Towards the AKÖ Scheme

As outlined in the introduction, our scheme is based on the three following ideas:

(i) create RLWE samples encrypted under the sum of RLWE keys of individual parties,

(ii) use a ternary (zero-centered) distribution for individual RLWE keys, and

(iii) avoid Fast Fourier Transform (FFT) in pre-computations.

Below, we discuss (i) and (ii), leaving (iii) for the experimental part (Section 4.5).

#### (R)LWE Keys & Bootstrapping Keys

First, we outline the structure of the secret (R)LWE keys, which are unrelated and owned by multiple parties, based on which we propose a structure of respective bootstrapping keys. Note that secret keys of individual parties are *never* revealed to any other party, however, the description of AKÖ involves all of them.

The underlying (and never reconstructed) LWE key is the *concatenation* of individual keys, i.e., $\mathbf{s} := \big(\mathbf{s}^{(1)}, \mathbf{s}^{(2)}, \dots, \mathbf{s}^{(k)}\big) \in \mathbb{B}^{kn}$, where $\mathbf{s}^{(p)} \in \mathbb{B}^{n}$ are secret LWE keys of individual parties. We refer to $\mathbf{s}$ as the *common* LWE *key*. For RLWE keys, we consider their *summation*, i.e., $Z := \sum_p z^{(p)}$, which we refer to as the *common* RLWE *key*. Note that this particular improvement decreases the computational complexity (as well as the blind-rotate key sizes) from $O(k^2)$ to $O(k)$.

For bootstrapping keys, we follow the original construction of TFHE, where we use the common (R)LWE keys. For *blind-rotate keys*, we generate an RGSW sample of each bit of the common LWE key $\mathbf{s} = \big(\mathbf{s}^{(1)}, \dots, \mathbf{s}^{(k)}\big)$, under the common RLWE key $Z = \sum_p z^{(p)}$. In addition, any party shall neither leak its own secrets nor require the secrets of others. For this purpose, we employ RLWE public key encryption [98]. Let us outline the desired form of a blind-rotate key for bit $s$:

$$\mathsf{BK}_s = \begin{pmatrix} \mathbf{b}^{\triangle} + s \cdot \mathbf{g} & \mathbf{a}^{\triangle} \\ \mathbf{b}^{\square} & \mathbf{a}^{\square} + s \cdot \mathbf{g} \end{pmatrix}, \quad \mathsf{BK}_s \in \big(\mathbb{T}^{(N)}[X]\big)^{2d \times 2}, \tag{4.3}$$

where $(\mathbf{b}^{\triangle}, \mathbf{a}^{\triangle})$ and $(\mathbf{b}^{\square}, \mathbf{a}^{\square})$ hold $d{+}d$ RLWE encryptions of zero under the key $Z$; cf. TFHE.Rgsw-Encr. For *key-switching keys*, we need to generate an LWE sample of the sum of $j$-th coefficients of individual RLWE secret keys $z^{(p)}$, under the common LWE key $\mathbf{s}$, for $j \in [0, N-1]$. Here a simple concatenation of masks (values $\mathbf{a}$) and a summation of masked values (values $b$) do the job. With such keys, bootstrapping itself is identical to that of the original TFHE.

### Ternary Distribution for RLWE Keys

For individual RLWE keys, we suggest to use zero-centered ternary distribution $\zeta_p \colon (-1, 0, 1) \to (p, 1-2p, p)$, parameterized by $p \in (0, 1/2)$, which is widely adopted by the main FHE libraries like HElib [67], Lattigo [101], SEAL [99], or HEAAN [121]. Although not adopted in CCS nor in KMS, in our scheme, a zero-centered distribution for RLWE keys is particularly useful, since we sum the keys into a common key, which is then also zero-centered. This helps reduce the blind-rotate noise from $O(k^3)$ to $O(k^2)$, which in turn helps find more efficient TFHE parameters.

It is worth noting that for "small" values of $p$, such keys are also referred to as *sparse keys* (in particular with a fixed/limited Hamming weight), and there exist specially tailored attacks [31, 123]. At this point, we motivate the choice of $p$ solely by keeping the information entropy of $\zeta_p$ equal to 1 bit, however, there is no intuition—let alone a proof—that the estimated security would be at least similar (more on concrete security estimates in Section 4.5.1 and Appendix C.2). For the information entropy of $\zeta_p$, we have

$$H(\zeta_p) = -2p\log(p) - (1-2p)\log(1-2p) \overset{!}{=} 1, \tag{4.4}$$

which gives a numerical solution of $p \approx 0.1135$. For $z_i \sim \zeta_p$, we have $\mathsf{Var}[z_i] = 2p \approx 0.227$.

## 4.3.3 Technical Description of AKÖ

We provide a technical description of AKÖ in the same form as for the TFHE scheme in Section 4.2.1. We mark algorithms that differ fundamentally from their TFHE counterparts with •, existing algorithms (possibly slightly modified) are marked with ∘. Algorithms with index $q$ are executed locally at the respective party. We remind that encryption algorithms naturally generalize to vector inputs.

### Static Variant of AKÖ

Below, we provide algorithms for the static variant of AKÖ:

• AKÖ.Setup($1^\lambda, k$): Given security parameter $\lambda$ and the number of parties $k$, generate and distribute to all $k$ parties the same parameters as generated by the TFHE.Setup($1^\lambda$) algorithm (n.b., $k$ is taken into account, hence the parameters differ from those given by TFHE.Setup($1^\lambda$)), and a *common random polynomial* (CRP) $\underline{a} \xleftarrow{\$} \mathbb{T}^{(N)}[X]$.

∘ AKÖ.SecKeyGen$_q$(): Generate secret keys $\mathbf{s}^{(q)} \xleftarrow{\$} \mathbb{B}^n$ and $z^{(q)} \in \mathbb{Z}^{(N)}[X]$, s.t. $z_i^{(q)} \xleftarrow{\zeta_p} \{-1, 0, 1\}$.

∘ AKÖ...: (R)LweSymEncr$_q$, (R)LwePhase$_q$, DecrBool$_q$, KeyExtract, Prod, BlindRotate, SampleExtract, KeySwitch, Add, Bootstrap, and NAND are the same as in TFHE.

∘ AKÖ.RLwePubEncr$\big(m, (b, a)\big)$: Given message $m \in \mathbb{T}^{(N)}[X]$ and public key $(b, a) \in \mathbb{T}^{(N)}[X]^2$ (an RLWE sample of $0 \in \mathbb{T}^{(N)}[X]$ under key $z \in \mathbb{Z}^{(N)}[X]$), generate temporary RLWE key $r^{(q)}$, s.t. $r_i^{(q)} \xleftarrow{\zeta} \{-1, 0, 1\}$. Evaluate $b' \leftarrow$ RLweSymEncr$_q(m, b, r^{(q)})$ and $a' \leftarrow$ RLweSymEncr$_q(0, a, r^{(q)})$. Output $(b', a')$, which is an RLWE sample of $m$ under the key $z$.

∘ $\underline{\texttt{AKÖ.RLweRevPubEncr}\big(m,(b,a)\big)}$: Proceed as RLwePubEncr, with a difference in the evaluation of $b' \leftarrow \texttt{RLweSymEncr}_q(0, b, r^{(q)})$ and $a' \leftarrow \texttt{RLweSymEncr}_q(m, a, r^{(q)})$, where only $m$ and 0 are swapped, i.e., $m$ is added to the right-hand side instead of the left-hand side.

• $\underline{\texttt{AKÖ.BlindRotKeyGen}_q()}$: Calculate and broadcast public key $b^{(q)} \leftarrow \texttt{RLweSymEncr}_q(0, \underline{a})$, using the CRP $\underline{a}$ as the mask. Evaluate $B = \sum_{p=1}^{k} b^{(p)}$ (n.b., $(B, \underline{a})$ is an RLWE sample of zero under the common RLWE key $Z = \sum_{p=1}^{k} z^{(p)}$, hence it may serve as a common public key). Finally, for $j \in [1, n]$, output the *blind-rotate key* (related to $s_j^{(q)}$ and $Z$):

$$\mathsf{BK}_j^{(q)} \leftarrow \begin{pmatrix} \texttt{RLwePubEncr}_q\big(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, (B, \underline{a})\big) \\ \texttt{RLweRevPubEncr}_q\big(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, (B, \underline{a})\big) \end{pmatrix}, \tag{4.5}$$

which is an RGSW sample of the $j$-th bit of $\mathbf{s}^{(q)}$, under the common RLWE key $Z$.

• $\underline{\texttt{AKÖ.KeySwitchKeyGen}_q()}$: For $i \in [1, N]$, broadcast $[\mathbf{b}_i^{(q)} | \mathbf{A}_i^{(q)}] \leftarrow \texttt{LweSymEncr}_q\big(\mathbf{z}_i^{(q)*} \cdot \mathbf{g}'\big)$, where $\mathbf{z}^{(q)*} \leftarrow \texttt{KeyExtract}(z^{(q)})$. Aggregate and for $i \in [1, N]$, output the *key-switching key* (for $Z_i = \sum_p z_i^{(p)}$ and $\mathbf{s} = (\mathbf{s}^{(1)}, \ldots, \mathbf{s}^{(k)})$):

$$\mathsf{KS}_i = \Big[\underbrace{\sum_{p=1}^{k} \mathbf{b}_i^{(p)}}_{\mathbf{b}_i} \,\Big|\, \underbrace{\mathbf{A}_i^{(1)}, \mathbf{A}_i^{(2)}, \ldots, \mathbf{A}_i^{(k)}}_{\mathbf{A}_i}\Big], \tag{4.6}$$

which is a $d'$-tuple of LWE samples of $\mathbf{g}'$-respective fractions of $\mathbf{Z}_i^*$ under the common LWE key $\mathbf{s}$. Here, $\mathbf{Z}_i^*$ is the $i$-th element of the extraction of the common RLWE key $Z = \sum_p z^{(p)}$, i.e., $\mathbf{Z}^* = \texttt{KeyExtract}(Z)$.

### Changes to AKÖ towards the Dynamic Variant

For the dynamic variant, we provide modified versions of BlindRotKeyGen and KeySwitchKeyGen; other algorithms are the same as in the static variant. Note that, in case we allow a party to join later, all temporary keys need to be stored permanently and both algorithms need to be (partially) repeated. This causes a slight pre-computation overhead over CCS and KMS.

• $\underline{\texttt{AKÖ.BlindRotKeyGen\_dyn}_q()}$: Calculate and broadcast public key $b^{(q)}$ as described in the AKÖ .BlindRotKeyGen$_q$() algorithm. Then, for $j \in [1, n]$:

```
1: generate two vectors of $d$ temporary RLWE keys $\mathbf{r}_j^{(q)}$ and $\mathbf{r}'^{(q)}_j$ (with coefficients distributed $\sim \zeta_p$)
2: for $p \in [1, k]$, $p \neq q$, output $\mathbf{b}_{q,j}^{\triangle(p)} \leftarrow \texttt{RLweSymEncr}_q(0, b^{(p)}, \mathbf{r}_j^{(q)})$
3: output $\mathbf{b}_{q,j}^{\triangle(q)} \leftarrow \texttt{RLweSymEncr}_q(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, b^{(q)}, \mathbf{r}_j^{(q)})$
4: output $\mathbf{a}_{q,j}^{\triangle} \leftarrow \texttt{RLweSymEncr}_q(0, \underline{a}, \mathbf{r}_j^{(q)})$
5: for $p \in [1, k]$, output $\mathbf{b}_{q,j}^{\square(p)} \leftarrow \texttt{RLweSymEncr}_q(0, b^{(p)}, \mathbf{r}'^{(q)}_j)$
6: output $\mathbf{a}_{q,j}^{\square} \leftarrow \texttt{RLweSymEncr}_q(\mathbf{s}_j^{(q)} \cdot \mathbf{g}, \underline{a}, \mathbf{r}'^{(q)}_j)$
```

To construct the $j$-th blind-rotate key of party $q$, related to subset of parties $\mathcal{S} \ni q$, evaluate

$$\mathsf{BK}^{(q)}_{j,\mathcal{S}} \leftarrow \begin{pmatrix} \sum_{p\in\mathcal{S}} \mathbf{b}^{\Delta(p)}_{q,j} & \mathbf{a}^{\Delta}_{q,j} \\ \sum_{p\in\mathcal{S}} \mathbf{b}^{\square(p)}_{q,j} & \mathbf{a}^{\square}_{q,j} \end{pmatrix}, \tag{4.7}$$

which is an RGSW sample of $\mathbf{s}^{(q)}_j$ under the subset RLWE key $Z_{\mathcal{S}} = \sum_{p\in\mathcal{S}} z^{(p)}$. N.b., $\mathsf{BK}^{(q)}_{j,\mathcal{S}}$ is only calculated at runtime, once $\mathcal{S}$ is known.

- `AKÖ.KeySwitchKeyGen_dyn`$_q$(): Proceed as `AKÖ.KeySwitchKeyGen`$_q$(), while instead of outputting aggregated $\mathsf{KS}_i$'s, aggregate relevant parts at runtime, once $\mathcal{S}$ is known. I.e.,

$$\mathsf{KS}_{i,\mathcal{S}} = \Big[\sum_{p\in\mathcal{S}} \mathbf{b}^{(p)}_i \;\Big|\; \big(\mathbf{A}^{(p)}_i\big)_{p\in\mathcal{S}}\Big]. \tag{4.8}$$

**Possible Improvements**

In [27], authors suggest an improvement that decreases the noise growth of key-switching, which can also be applied in our scheme; we provide more details in Appendix A.

## 4.4 Theoretical Analysis of Our Scheme

In this section, we provide a theoretical analysis of our AKÖ scheme with respect to *security*, *correctness* (noise growth), and *performance*.

### 4.4.1 Security

We assume that all parties follow the protocol *honestly-but-curiously* (i.e., we assume the semi-honest model). Before we comment on each algorithm that may leak secrets, let us recall what *is* secure and what *is not* in LWE (selected methods; also holds for RLWE):

✓ re-use secret key $\mathbf{s}$ with fresh mask $\mathbf{a}$ and fresh noise $e$;

✓ re-use common random mask $\underline{\mathbf{a}}$ with multiple distinct secret keys $\mathbf{s}^{(p)}$ and fresh noises $e^{(p)}$;

✗ publish $\langle \mathbf{s}, \mathbf{a}\rangle$ in any form (e.g., release the phase $\varphi$ or the noise $e$);

✗ re-use the pair $(\mathbf{s}, \mathbf{a})$ with fresh noises $e_i$.

Below, we show that if all parties act semi-honestly, our scheme is secure in both of its variants. Note that rather than formal security proofs, we provide informal sketches. In selected cases, we also briefly discuss what issues may rise with a malicious party and we outline possible countermeasures.

**Public Key Encryption**

In AKÖ, there are two algorithms for public key encryption: `RLwe(Rev)PubEncr`$\big(m, (b, a)\big)$. Basically, they re-use a common random mask (the public key pair $(b, a)$) with fresh temporary key $r^{(q)}$. Provided that $b$ and $a$ are indistinguishable from random (random-like), it does not play a role to which part the message $m$ is added/encrypted, i.e., both variants are secure.

### Blind-Rotate Key Generation (static variant)

Provided that CRP $\underline{a}$ is random-like, which is trivial to achieve in the random oracle model, we can assume that (our) $b^{(q)}$ is random-like. Assuming that other parties act honestly, also their $b^{(p)}$'s are random-like, hence the sum $B$ is random-like, too. With $(B, \underline{a})$ random-like, public key encryption algorithms are secure, hence $\texttt{AKÖ.BlindRotKeyGen}_q$ is secure, too.

### Blind-Rotate Key Generation (dynamic variant)

In this variant, party $q$ re-uses temporary secret key $r^{(q)}$ for encryption of zeros using public keys $b^{(p)}$ of other parties, and for encryption of own secret key $\mathbf{s}^{(q)}$. This is secure provided that $b^{(p)}$'s are random-like, which is true if generated honestly.

### Key-Switching Key Generation (both variants)

The $\texttt{AKÖ.KeySwitchKeyGen(\_dyn)}_q$ algorithms employ the standard $\mathsf{LWE}$ encryption, hence they are both secure.

### Summary

We have shown that if all parties act semi-honestly, our scheme is secure in both of its variants. We also outline possible countermeasures if there is a malicious party. However, we leave a rigorous discussion on threat models that involve malicious actors for the future work.

### On the Presence of a Malicious Party

Although we assume that *all* parties are semi-honest, we comment briefly and informally on the possible presence of a malicious party. First, note that there is another *insecure* thing in LWE:

✗ use malicious common mask $\underline{\mathbf{a}}$ (in particular in RLWE).

For this issue, let us outline an RLWE key recovery attack, given an encryption oracle:

1. the attacker provides malicious common mask (public key) $a' = 1/4 + 0 \cdot X + \ldots + 0 \cdot X^{N-1}$;
2. the victim encrypts 0 with her secret key $z$ as $(b = -z \cdot a' + e, a') = (-1/4\, z + e, 1/4)$;
3. the attacker rounds the coefficients of $4b \in [-2, 2)^{(N)}[X]$ to integers, yielding the secret key $z$.

**Blind-Rotate Key Generation (static variant)** In case there is malicious party $p'$, it may wait for others and collect their $b^{(p)}$'s, then it may publish malicious $b^{(p')} = 1/4 - \sum_{p \neq p'} b^{(p)}$, i.e., $B = 1/4$ (cf. the attack outlined before). However, such an attack can be mitigated easily: each party $p$ first commits on $b^{(p)}$ before publishing it, i.e., before learning $b$'s of others. Then, even if some $b$'s are malicious, the aggregate $B$ can be considered random-like: indeed, it is sufficient that one party (us) provides an *unpredictable* random-like $b^{(q)}$.

**Blind-Rotate Key Generation (dynamic variant)** In case there is malicious party $p'$, an attack with $b^{(p')} = 1/4$ (or similar) could be mounted; let us outline a possible mitigation:

- parties generate and distribute all keys normally;
- a series of bootstraps with some dummy data is performed;

- the results are checked for correctness: the protocol halts unless everything is correct.

Recall that this is just a *proposal* of a possible countermeasure and we only provide a brief reasoning: To generate malicious *and* functional $b^{(p')}$, i.e., $b^{(p')}$ of a specific form (e.g., $1/4$) *and* $b^{(p')} = -z^{(p')} \cdot \underline{a} + e$, the attacker $p'$ would need to find short vectors/polynomials $z^{(p')}$ and $e$ that solve the equation, which is considered intractable. If the attacker finds *some* solution to $z^{(p')}$ and $e$, which is not short, the noise growth is expected be enormous, hence it is very likely to destroy the correctness and the protocol halts.

## 4.4.2 Correctness & Noise Growth

The most challenging part of all LWE-based schemes is to estimate the noise growth across various operations. First, we provide estimates of the noise growth of blind-rotate and key-switching, next, we combine them into an estimate of the noise of a freshly bootstrapped sample. Finally, we identify the maximum of error, which may cause incorrect bootstrapping. By default, we evaluate all noises for the static variant, while for the dynamic variant, we provide more comments in the proofs.

### Noise Growth of Blind-Rotate

In the following lemma and theorem, we provide an estimate of the noise growth during blind-rotate, without considering any implementation aspects.

**Lemma 4.1** (Correctness & Noise Growth of `AKÖ.Prod`). *Given* RGSW *sample* BK *generated by the* `AKÖ.BlindRotKeyGen` *algorithm, which encrypts constant polynomial $s \in \mathbb{Z}^{(N)}[X]$ under the common* RLWE *key $Z = \sum_p z^{(p)}$, and* RLWE *sample $\bar{\mathbf{c}} = (b, a)$ that encrypts $m \in \mathbb{T}^{(N)}[X]$ under the same key, the* `AKÖ.Prod` *algorithm returns* RLWE *sample $\bar{\mathbf{c}}' = (b', a')$ that encrypts $s \cdot m$ under $Z$ with additional noise $e_{\texttt{Prod}}$, given by $\langle \bar{\mathbf{Z}}, \bar{\mathbf{c}}' \rangle = s \cdot \langle \bar{\mathbf{Z}}, \bar{\mathbf{c}} \rangle + e_{\texttt{Prod}}$, for which*

$$\mathsf{Var}[e_{\texttt{Prod}}] \approx \underbrace{NdV_B\beta^2\big(3 + 6pkN\big)}_{\mathsf{BK}\ \textit{error}} + \underbrace{s^2\varepsilon^2\big(1 + 2pkN\big)}_{\textit{decomp. error}}, \tag{4.9}$$

*where*

- *$\varepsilon^2 := 1/12B^{2d}$ is the variance of (real-valued) uniform distribution on $[-1/2B^d, 1/2B^d)$,*
- *$V_B := (B^2+2)/12$ is the mean of squares of (integer valued) uniform distribution on $[-B/2, B/2)$ (assuming $B$ is even),*
- *other notation and parameters are as per the* `AKÖ.Setup` *algorithm, and*
- *we refer to the two terms as the* blind-rotate key error *and the* decomposition error*, respectively.*

*If this error is sufficiently small, it holds $\langle \bar{\mathbf{Z}}, \bar{\mathbf{c}}' \rangle \approx s \cdot \langle \bar{\mathbf{Z}}, \bar{\mathbf{c}} \rangle$, i.e., the* `AKÖ.Prod` *algorithm is indeed multiplicatively homomorphic.*

*Proof.* Find the proof in Appendix B.1. For the dynamic variant, we have $(3 + k \cdot 6pN) \to \big(1 + k(2 + 6pN)\big)$ in the BK error term, which we consider practically negligible as $6pN \approx 700$. □

**Theorem 4.2** (Noise Growth of Blind-Rotate). *The* `AKÖ.BlindRotate` *algorithm returns a sample with noise variance given by*

$$\mathsf{Var}[\langle \bar{\mathbf{Z}}, \mathsf{ACC} \rangle] \approx \underbrace{knNdV_B\beta^2(3 + 6pkN)}_{\mathsf{BK}\ \textit{error}} + \underbrace{1/2 \cdot kn\varepsilon^2(1 + 2pkN)}_{\textit{decomp. error}} + \underbrace{\mathsf{Var}[tv]}_{\textit{usually}\ 0}. \tag{4.10}$$

*The resulting* ACC *encrypts* $X^{\langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}}) \rangle} \cdot tv$.

*Proof.* Find the proof in Appendix B.2. For the dynamic variant, $(3 + 6pkN) \to (1 + 2k + 6pkN)$. □

**Noise Growth of Key-Switching**

In the following theorem, we provide an estimate of the noise growth during key-switching, which holds for both variants.

**Theorem 4.3** (Noise Growth of Key-Switching)**.** *The* `AKÖ.KeySwitch` *algorithm returns a sample that encrypts the same message as the input sample, while changing the key from* $\mathbf{Z}^*$ *to* $\mathbf{s}$*, with additional noise* $e_{\mathsf{KS}}$*, given by* $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}'' \rangle = \langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}' \rangle + e_{\mathsf{KS}}$*, for which*

$$\mathsf{Var}[e_{\mathsf{KS}}] \approx \underbrace{Nkd'V_{B'}\beta'^2}_{\mathsf{KS}\ error} + \underbrace{2pkN\varepsilon'^2}_{decomp.\ error} . \tag{4.11}$$

*If the error is sufficiently small, it holds* $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}'' \rangle \approx \langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}' \rangle$.

*Proof.* Find the proof in Appendix B.3. For the dynamic variant, key-switching keys are structurally equivalent, hence this estimate holds in the same form. □

**Noise of a Freshly Bootstrapped Sample**

In the following corollary, we combine noise estimates of blind-rotate and key-switching, yielding a noise estimate of a freshly bootstrapped sample. For the dynamic variant, the BK error term is changed according to Theorem 4.2.

**Corollary 4.4** (Noise of a Freshly Bootstrapped Sample)**.** *The* `AKÖ.Bootstrap` *algorithm returns a sample with noise variance given by*

$$V_0 \approx \underbrace{3knNdV_B\beta^2(1 + 2pkN)}_{\mathsf{BK}\ error} + \underbrace{{}^1\!/_2 kn\varepsilon^2(1 + 2pkN)}_{b.\text{-}r.\ decomp.} + \underbrace{Nkd'V_{B'}\beta'^2}_{\mathsf{KS}\ error} + \underbrace{2pkN\varepsilon'^2}_{k.\text{-}s.\ decomp.} . \tag{4.12}$$

*For the dynamic variant, the* BK *error term is changed according to Theorem 4.2.*

**Maximum of Error**

During homomorphic evaluations, freshly bootstrapped samples get homomorphically added/subtracted, before being possibly bootstrapped again. Before a noisy sample gets blindly rotated, it gets scaled and rounded to $Z_{2N}$; cf. line 1 of `BlindRotate`. In the following lemma, we evaluate the variance of such a rounding error.

**Lemma 4.5** (Rounding Error of Blind-Rotate)**.** *The rounding step on line 1 of the* `AKÖ.BlindRotate` *algorithm induces an additional error with variance (in the torus scale) given by*

$$\mathsf{Var}\big[\big\langle \bar{\mathbf{s}}, {}^1\!/_{2N} \cdot (\tilde{b}, \tilde{\mathbf{a}}) - (b, \mathbf{a}) \big\rangle\big] = \frac{1 + {}^{kn}\!/_2}{48N^2} =: V_{round}(N, n, k). \tag{4.13}$$

*Proof.* Each of the $1 + kn$ values of $(b, \mathbf{a})$ gets rounded to the closest multiple of ${}^1\!/_{2N}$, i.e., the error is uniform on the interval $(-{}^1\!/_{4N}, {}^1\!/_{4N}]$. The result follows. □

After rounding, the noise gets refreshed inside the `BlindRotate` algorithm. It follows that the maximum of error across the whole computation appears right after rounding of the sample to-be-bootstrapped. We focus on this error in the experimental part, since it may cause incorrect blind-rotation, in turn, incorrect LUT evaluation. In the following corollary, we evaluate the variance of the maximal error throughout the calculation and we define quantity $\kappa$, which is a scaling factor of normal distribution $N(0,1)$.

**Corollary 4.6** (Maximum of Error)**.** *The maximum average error throughout homomorphic computation is achieved inside the* `AKÖ.Bootstrap` *algorithm by the rounded sample* $1/2N \cdot (\widetilde{b}, \tilde{\mathbf{a}})$*. Its variance is given by*

$$V_{\max} \approx \max\Big\{\sum k_i^2\Big\} \cdot V_0 + V_{round}, \tag{4.14}$$

*where $k_i$ are coefficients of linear combinations of independent, freshly bootstrapped samples, which are evaluated during homomorphic calculations, before being bootstrapped (e.g., $\sum k_i^2 = 2$ for the NAND gate evaluation). We denote*

$$\kappa := \frac{\delta/2}{\sqrt{V_{\max}}} = \frac{\delta}{2\sigma_{\max}}, \tag{4.15}$$

*where $\delta$ is the distance of encodings that are to be distinguished (e.g., $1/4$ for encoding of bools).*

We use $\kappa$ to estimate the probability of *correct blind rotation* (CBRot). E.g., for $\kappa = 3$, we have Pr[CBRot] $\approx 99.73\% \approx 1/370$ (aka. rule of $3\sigma$), however, we rather lean to $\kappa = 4$ with Pr[CBRot] $\approx 1/15\,787$. Since the maximum of error is achieved within blind-rotate, it dominates the overall probability of *correct bootstrapping* (CBStrap), i.e., we assume Pr[CBStrap] $\approx$ Pr[CBRot].

### 4.4.3 Performance

Since the structure of all components in both variants of AKÖ is equivalent to that of plain TFHE with only $n \to kn$ (due to LWE key concatenation), we evaluate the performance characteristics very briefly: `AKÖ.BlindRotate` is dominated by $4d \cdot kn$ degree-$N$ polynomial multiplications, whereas `AKÖ.KeySwitch` is dominated by $Nd' \cdot (1+kn)$ torus multiplications, followed by $1+kn$ summations of $Nd'$ elements. Using FFT for polynomial multiplication, for bootstrapping, we have the complexity of $O(N \log N \cdot 4dkn) + O(Nd' \cdot (1+kn))$.

For key sizes, we have $|\mathsf{BK}| = 4dNkn \cdot |\mathbb{T}_{\mathsf{RLWE}}|$ and $|\mathsf{KS}| = d'N(1+kn) \cdot |\mathbb{T}_{\mathsf{LWE}}|$, where $|\mathbb{T}_{\mathsf{(R)LWE}}|$ denotes the size of respective torus representation.

## 4.5 Experimental Evaluation

For a fair comparison, we implement our AKÖ scheme[2] side by side with previous schemes CCS [27] and KMS [93]. These are implemented in a fork [122] of a library[3] [106] that implements TFHE in Julia. For the sake of simplicity, we implement only the static variant on AKÖ – recall that performance-wise, the two variants are equivalent, for noise growth, the differences are negligible.

In this section, we first comment on errors induced by existing TFHE implementations. Then, we introduce type-1 and type-2 decryption errors that one may encounter during TFHE-based homomorphic evaluations. Finally, we provide three kinds of results of our experiments:

[2]Available at `https://gitlab.eurecom.fr/fakub/3-gen-mk-tfhe` as the `3gen` variant, code mostly written by Yavuz Akın.

[3]As noted by the authors, the code serves solely as a proof-of-concept.

1. for all the three schemes (CCS, KMS, and $\mathtt{AK\ddot{O}}$) and selected parameter sets, we measure the *performance*, the *noise variances*, and the *amount of decryption errors* of the two types,

2. we demonstrate the *effect of FFT* during the pre-computation phase of $\mathtt{AK\ddot{O}}$ with 32 parties,

3. we compare the performance of all the three schemes with a *fixed parameter set* tailored for 16 parties, with different numbers of actually participating parties (i.e., the dynamic variant).

We run our experiments on a machine with an Intel Core i7-7800X processor and 128 GB of RAM.

### Implementation Errors

The major source of errors that stem from a particular implementation of the $\mathsf{TFHE}$ scheme is Fast Fourier Transform (FFT), which is used for fast modular polynomial multiplication in $\mathsf{RLWE}$; find a study on FFT errors in [81]. Also, the finite representation of the torus (e.g., 64-bit integers) changes the errors slightly, however, we neglect this contribution as long as the precision (e.g., $2^{-64}$) is smaller than the standard deviation of the $(\mathsf{R})\mathsf{LWE}$ noise. Note that these kinds of errors are not taken into account in Section 4.4.2, which solely focuses on the theoretical noise growth of the scheme itself.

The magnitude of the FFT error depends on (i) the finite torus representation (i.e., the precision of coefficients of multiplied polynomials), and on (ii) particular FFT implementation (e.g., what float representation is chosen); find a study on FFT errors in [81].

Due to the excessive noise that we observe for higher numbers of parties with our scheme, we suggest replacing FFT in pre-computations (i.e., in blind-rotate key generation) with an exact method. This leads to an increase of the pre-computation costs (n.b., it has no effect on the bootstrapping time), however, in Section 4.5.2, we show that the benefit is worth it – the pre-computation time indeed shows to be slower, yet it is not prohibitive.

### Types of Decryption Errors

The ultimate goal of noise analysis is to keep the probability of obtaining an incorrect result reasonably low. Below, we describe two types of decryption errors, which originate from bootstrapping, and which we measure in our experiments. N.b., the principle of `BlindRotate` is the same across the three schemes, hence it is well-defined for all of them.

*Note* 4.1. For the notion of *correct decryption*, we always assume symmetric intervals around encodings. E.g., for the Boolean variant of $\mathsf{TFHE}$, which encodes true and false as $\pm 1/8$, we only consider the "correct" interval for true as $(0, 1/4)$, although $(0, 1/2)$ would work, too. Hence in the Boolean variant, actual incorrect decryption & decoding would be half less likely than what we actually measure.

**Fresh Bootstrap Error** We bootstrap (ideally) noiseless sample $\mathbf{c}$ of $\mu$, i.e., `BlindRotate` rotates the test vector "correctly", meaning that $\tilde{\varphi}/2N \approx \mu$ selects the correct position from the test vector. Then, we evaluate the probability of the resulting phase $\varphi' = \langle \bar{\mathbf{s}}, \bar{\mathbf{c}}' \rangle$ falling outside the correct interval. We refer to this error as the *type-1 error*, denoted $\mathsf{Err}_1$. Note that this probability relates to the noise of a correctly blind-rotated, freshly bootstrapped sample. It can be estimated from $V_0$; see (4.12).

**Blind Rotate Error** Let us consider a homomorphic sum of two independent, freshly bootstrapped samples. We evaluate the probability that the sum, after the rounding step inside `BlindRotate`, selects a value at an *incorrect* position from the test vector, which encodes the LUT (as discussed in Section 4.4.2). We refer to this error as the *type-2 error*, denoted $\mathsf{Err}_2$. It can be estimated from $V_{\max}$; see (4.14). We evaluate $\mathsf{Err}_2$ by simulating the NAND gate:

$$\left.\begin{array}{l}\text{fresh } \mathbf{c}_1 \xrightarrow{\texttt{Bootstrap}} \mathbf{c}_1' \\ \text{fresh } \mathbf{c}_2 \xrightarrow{\texttt{Bootstrap}} \mathbf{c}_2'\end{array}\right\} \; ({}^1\!/\!_8 - \mathbf{c}_1' - \mathbf{c}_2') \to \text{eval. } \tilde{\varphi} \text{ of } \texttt{BlindRotate} \to \text{check } {}^{\tilde{\varphi}}\!/\!_{2N} \overset{?}{\in} (0, {}^1\!/\!_4). \tag{4.16}$$

## 4.5.1 Experiment #1: Thorough Comparison of Performance & Errors

For the three schemes—CCS, KMS, and AKÖ—we measure the main quantities: the bootstrapping time (median), the variance $V_0$ of a freshly bootstrapped sample (defined in (4.12)), the scaling factor $\kappa$ (defined in (4.15)), and the number of errors of both types. We extend the previous work – there is no experimental evaluation of noises/errors in CCS nor in KMS. In all experiments, we replace FFT in pre-computations with an exact method. For CCS and KMS, we employ the parameters suggested by the original authors, and we estimate their security with the `lattice-estimator` by Albrecht et al. [8, 9]. We obtain an estimate of about 100 bits, therefore for our scheme, we also suggest parameters with estimated 100-bit security. We provide more details on concrete security estimates of the parameters of CCS and KMS, and those of AKÖ in Appendix C.1 and C.2, respectively. The parameters and the results for CCS, KMS, and AKÖ can be found in Table 4.1, 4.2 and 4.3, respectively.

In the results for CCS, we may notice that for 2 to 8 parties, the measured value of $\kappa$, denoted $\kappa^{(m)}$, agrees with the calculated value $\kappa^{(c)}$, whereas for 16 parties (n.b., parameters added in KMS [93]), the measured value $\kappa^{(m)}$ drops significantly, which indicates an unexpected error growth.

In the results for KMS, we may notice a similar drop of $\kappa$ – here it occurs for all numbers of parties – we suppose that this is caused by FFT in bootstrapping (more on FFT later in Section 4.5.2). For both experiments, we further use $\kappa^{(m)}$ and $Z$-values of the normal distribution to evaluate the expected rate of $\mathsf{Err}_2$, which is in perfect accordance with the measured one.

For our AKÖ scheme, the results do not show *any* error of any type. Regarding the values of $\kappa$ (also $V_0$), we measure lower noise than expected – this we suppose to be caused by a certain statistical dependency of variables – indeed, our estimates of noise variances are based on an assumption that variables are independent, which is not always fully satisfied. We are able to run AKÖ with up to 128 parties, while the only limitation for 256 parties appears to be the size of RAM. We believe that with more RAM ($>128$ GB) or with a more optimized implementation, it would be possible to practically instantiate the scheme with even more parties. For this purpose, we provide parameter sets for 256 and even for 512 parties, where other technical limits are reached: in particular the speed of the `lattice-estimator` and the size of a machine word, which efficiently implements the torus.

## 4.5.2 Experiment #2: The Effect of FFT in Pre-computations

As outlined previously, polynomial multiplication in RLWE—when implemented using FFT—introduces additional error, on top of the standard RLWE noise. In this experiment, we compare noises of freshly bootstrapped samples: once *with* FFT in blind-rotate key generation (induces additional errors), once *without* FFT (we use an exact method instead). For this comparison, we choose our AKÖ scheme with 32 parties; find the results in Figure 4.1. Note that within bootstrapping, we still employ FFT, i.e., the performance of evaluation is not affected.

Table 4.1: Parameters, bootstrapping times ($t_B$; median), noises and errors of the CCS scheme [27], with original parameters and *without* FFT in pre-computations (i.e., using precise calculations); parameters for 16 parties and key sizes taken from [93]. Labels $^{(c)}$ and $^{(m)}$ refer to calculated and measured values, respectively. Running 1 000 trials, i.e., evaluating 2 000 bootstraps; cf. (4.16). N.b., the actual error rate of a NAND gate would be approximately half of $\mathsf{Err}_2$; cf. Note 4.1.

| $k$ | LWE | | | | RLWE | | UniEnc | | $\lvert\mathsf{keys}\rvert$ | $t_B$ | $V_0^{(c)}$ | $V_0^{(m)}$ | $\kappa^{(c)}$ | $\kappa^{(m)}$ | $\mathsf{Err}_{1,2}$ | | Exp. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $\alpha$ | $B'$ | $d'$ | $N$ | $\beta$ | $B$ | $d$ | [MB] | [s] | $[10^{-4}]$ | $[10^{-4}]$ | | | [‰] | | $\mathsf{Err}_2$ |
| 2 | 560 | $3.05 \cdot 10^{-5}$ | $2^2$ | 8 | 1 024 | $3.72 \cdot 10^{-9}$ | $2^9$ | 3 | 95 | .58 | 16.2 | 14.6 | 2.19 | 2.30 | 1 | 24 | 21 |
| 4 | | | | | | | $2^8$ | 4 | 108 | 2.4 | 19.1 | 18.6 | 2.01 | 2.04 | 3 | 41 | 41 |
| 8 | | | | | | | $2^6$ | 5 | 121 | 10 | 6.36 | 6.27 | 3.39 | 3.41 | 0 | 0 | .65 |
| 16 | | | | | | | $2^2$ | 12 | 214 | 86 | 2.15 | 34.5 | 5.07 | 1.49 | 29 | 128 | 136 |

Table 4.2: Parameters, bootstrapping times ($t_B$; median), noises and errors of the KMS scheme [93], with original parameters and without FFT in pre-computations (key sizes taken from [93]). Running 1 000 trials.

| $k$ | LWE | | | | RLWE | | RGSW | | RLEV | | UniEnc | | \|keys\| | $t_B$ | $V_0^{(c)}$ | $V_0^{(m)}$ | $\kappa^{(c)}$ | $\kappa^{(m)}$ | $\mathsf{Err}_{1,2}$ | | Exp. |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $\alpha$ | $B'$ | $d'$ | $N$ | $\beta$ | $B$ | $d$ | $B$ | $d$ | $B$ | $d$ | [MB] | [s] | $[10^{-4}]$ | $[10^{-4}]$ | | | [‰] | | $\mathsf{Err}_2$ |
| 2 | | | | | | | $2^{13}$ | 3 | $2^7$ | 2 | $2^{10}$ | 3 | 215 | .61 | .458 | 11.5 | 12.7 | 2.60 | 1.5 | 12 | 9.3 |
| 4 | | | | | | | $2^8$ | 5 | $2^8$ | 2 | $2^6$ | 7 | 286 | 2.1 | .915 | 15.3 | 8.97 | 2.26 | 4 | 29 | 24 |
| 8 | 560 | $3.05 \cdot 10^{-5}$ | $2^2$ | 8 | 2 048 | $4.63 \cdot 10^{-18}$ | $2^{11}$ | 4 | $2^6$ | 3 | $2^4$ | 8 | 251 | 5.4 | 1.83 | 17.1 | 6.34 | 2.13 | 3 | 35 | 33 |
| 16 | | | | | | | $2^9$ | 5 | $2^6$ | 3 | $2^4$ | 9 | 286 | 15 | 3.66 | 32.0 | 4.49 | 1.56 | 22.5 | 122 | 119 |
| 32 | | | | | | | $2^8$ | 6 | $2^7$ | 3 | $2^2$ | 16 | 322 | 35 | 7.32 | 30.1 | 3.17 | 1.60 | 23 | 109 | 110 |

Table 4.3: Parameters, key sizes (calculated), bootstrapping times ($t_B$; median), noises and errors of the static variant of AKÖ, without FFT in pre-computations. Running 1 000 trials, no errors of type $\mathsf{Err}_2$ (let alone $\mathsf{Err}_1$) experienced. *For 256 and 512 parties, we exceed the limit of RAM (128 GB). **For 512 parties, better parameters could be found – the practical size of the torus representation (64-bit) poses the limit.

| $k$ | LWE | | | | RLWE | | | | \|keys\| | $t_B$ | $V_0^{(c)}$ | $V_0^{(m)}$ | $\kappa^{(c)}$ | $\kappa^{(m)}$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $n$ | $\log_2(\alpha)$ | $B'$ | $d'$ | $N$ | $\log_2(\beta)$ | $B$ | $d$ | [GB] | [s] | $[10^{-4}]$ | $[10^{-4}]$ | | |
| 2 | 520 | $-13.52$ | $2^3$ | 3 | | | $2^7$ | 2 | .08 | .19 | 4.69 | 4.18 | 4.04 | 4.27 |
| 3 | 510 | $-13.26$ | $2^2$ | 5 | | | $2^7$ | 2 | .13 | .31 | 4.64 | 4.40 | 4.04 | 4.14 |
| 4 | 510 | $-13.26$ | $2^2$ | 5 | 1 024 | $-30.70$ | $2^6$ | 3 | .24 | .56 | 3.96 | 2.02 | 4.33 | 5.93 |
| 5 | 520 | $-13.52$ | $2^2$ | 5 | | | $2^6$ | 3 | .31 | .73 | 3.76 | 1.91 | 4.41 | 6.00 |
| 8 | 540 | $-14.04$ | $2^2$ | 5 | | | $2^4$ | 4 | .66 | 1.2 | 4.43 | 4.20 | 4.01 | 4.11 |
| 16 | 590 | $-15.34$ | $2^3$ | 4 | | | $2^{26}$ | 1 | .93 | 1.8 | 4.56 | 1.02 | 4.04 | 7.90 |
| 32 | 620 | $-16.12$ | $2^3$ | 4 | | | $2^{26}$ | 1 | 2.0 | 4.3 | 3.58 | 1.21 | 4.38 | 6.78 |
| 64 | 650 | $-16.90$ | $2^3$ | 4 | 2 048 | $-62.00$ | $2^{25}$ | 1 | 4.1 | 8.6 | 3.41 | 1.80 | 4.20 | 5.25 |
| 128 | 670 | $-17.42$ | $2^3$ | 5 | | | $2^{24}$ | 1 | 9.1 | 18 | 2.40 | .486 | 4.15 | 5.47 |
| 256* | 740 | $-19.24$ | $2^2$ | 8 | | | $2^{18}$ | 2 | 37 | – | .187 | – | 4.00 | – |
| 512** | 730 | $-18.98$ | $2^3$ | 5 | 4 096 | $-62.00$ | $2^{27}$ | 1 | 80 | – | 2.53 | – | 4.01 | – |

In the plot, we may notice a tremendous growth of the noise of a freshly bootstrapped sample in case FFT is employed for blind-rotate key generation: in almost 4% of such cases, even a freshly bootstrapped sample gets decrypted incorrectly (i.e., $\mathsf{Err}_1 \approx 4\%$), which corresponds to violet bars outside the interval delimited by the red dashed lines. On the other hand, such a growth does not occur for lower numbers of parties, hence we suggest verifying whether in the particular case, the effect of FFT is remarkable, or negligible, and then decide accordingly. Recall that pre-computations with FFT are much faster (e.g., for 64 parties, we have 33 s vs. 212 s of the total pre-computation time).

### Unexpected Error Growth in KMS

For the KMS scheme, we observe an unexpected error growth (cf. Table 4.2), which we suppose to be caused by FFT in bootstrapping (i.e., evaluation). We replace *all* FFTs in the entire computation of KMS—including bootstrapping—with an exact method, and we re-run Experiment #1 with the KMS scheme, using the same setup. Due to a prohibitively slow evaluation ($\sim 40\times$ slower), we only re-run the experiment for 2 parties. We obtain $V_0^{(m)} \approx 5.58 \cdot 10^{-4}$, which is still much more than the expected value $V_0^{(c)} \approx 0.458 \cdot 10^{-4}$, but it already makes the standard deviation about 30% smaller, compared to the "with FFT in bootstrapping" case. Also, it increases the value of $\kappa^{(m)}$ from 2.60 to 3.73 and it results in no type-2 errors. At least partially, this confirms our hypothesis that the unexpected error growth in KMS is caused by FFT in evaluation.

A possible theoretical explanation can be found in the design of KMS: in the blind-rotate of KMS, we may observe that there are (up to) four nested FFTs: one in the circled $\star$ product,

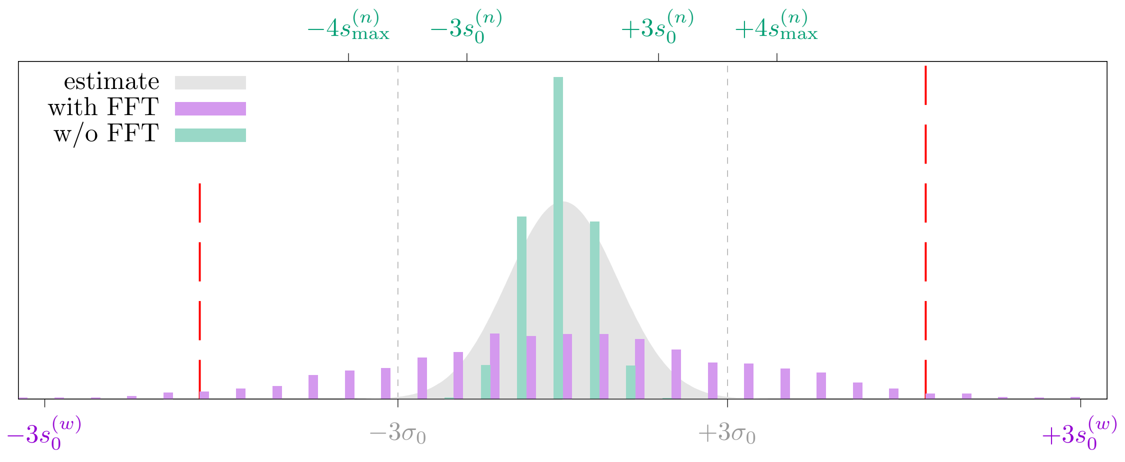


Figure 4.1: Noises of freshly bootstrapped samples of the static variant of `AKÖ` with 32 parties (parameters as per Table 4.3), comparing blind-rotate keys generated *with* and *without* the use of FFT, running 2 000 bootstraps. Red dashed lines mark the boundaries of the correct interval; cf. Note 4.1. The values $s_0^{(\cdot)}$ and $s_{\max}^{(\cdot)}$ refer to the sample standard deviation of a freshly bootstrapped sample and that of a rounded sample within blind-rotate (cf. (4.14); calculated from respective $s_0$), respectively. Labels $^{(w)}$ and $^{(n)}$ refer to *with FFT* and *no FFT*, respectively. N.b., the values $\pm 4s_{\max}^{(w)}$ are far outside the graph.

followed by three inside `ExtProd`: one in the $\odot$ product and two in `NewHbProd`. Compared with `AKÖ`, where there is just one level of FFT inside blind-rotate in `Prod`, this is likely the most significant practical improvement over KMS.

### 4.5.3 Experiment #3: Performance Comparison

We extend the performance comparison of CCS and KMS, presented in Figure 2 of KMS [93] (which we re-run on our machine), by the performances of our `AKÖ` scheme. Note that the setup of that experiment corresponds to the dynamic variant – recall that performance-wise, the dynamic variant is equivalent to the static variant, which is implemented in our experimental library. For each scheme, we employ its own parameter set tailored for 16 parties (cf. Table 4.1, 4.2 and 4.3), while we instantiate it with different numbers of actually participating parties; find the results in Figure 4.2.

### 4.5.4 Discussion

The goal of our experiments is to show the practical usability of our `AKÖ` scheme: we compare its performance as well as the probability of errors with previous schemes – CCS [27] and KMS [93].

In terms of bootstrapping time, `AKÖ` runs faster than both previous attempts (cf. Figure 4.2). Also, the theoretical complexity of `AKÖ` is linear in the number of parties (cf. Section 4.4.3), as opposed to quadratic and quasi-linear for CCS and KMS, respectively.

To evaluate the number of errors that may occur during bootstrapping, we propose a new method that simulates the rounding step of `BlindRotate` (cf. (4.16)), which is the same across all the three schemes. Our experiments show that both CCS and KMS suffer from a considerably high error rate (cf. Table 4.1 and 4.2, respectively): for CCS, the original parameters are rather

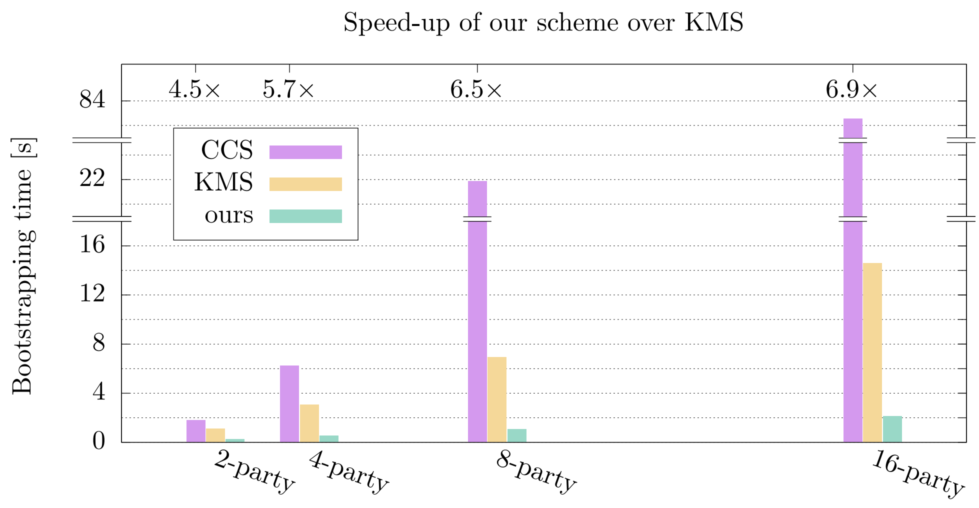


Figure 4.2: Comparison of median bootstrapping times of the CCS scheme [27], the KMS scheme [93], and our `AKÖ` scheme. 100 runs with respective parameters for 16 parties were executed. N.b., FFT in pre-computations does not affect performance.

poor; for KMS, it seems that there are too many nested FFT's in bootstrapping – we show that FFT in evaluation—at least partially—causes the unexpected error growth.

To sum up, `AKÖ` significantly outperforms both CCS & KMS in terms of bootstrapping time and/or error rate. The major practical limitation of the CCS scheme is the quadratic growth of the bootstrapping time, whereas the KMS scheme suffers from the additional error growth in implementation. A disadvantage of `AKÖ` is that it requires (a small amount of) additional pre-computations if a new party decides to join the computation in the dynamic variant. Also `AKÖ` does not enable parallelization, as opposed to KMS.

## 4.6 Conclusion

We propose a new `TFHE`-based MKHE scheme named `AKÖ` in two variants, depending on whether only a subset of parties is desired to take part in a homomorphic computation. We implement `AKÖ` side-by-side with other similar schemes CCS and KMS, and we show its practical usability in thorough experimentation, where we also suggest secure & reliable parameters. Thanks to its low noise growth, `AKÖ` can be instantiated with hundreds of parties; namely, we tested up to 128 parties in our experiments. Compared to previous schemes, `AKÖ` achieves much faster bootstrapping times, however, a slight overhead of pre-computations is induced. For KMS, we show that FFT errors are prohibitive for its practical deployment – unfortunately, replacing FFT in pre-computations is not enough.

Besides benchmarking, we suggest emulating (a part of) the NAND gate to achieve a more realistic error analysis: the measured amount of errors shows to be in perfect accordance with the expected amount. This method may help future schemes to evaluate their practical reliability.

### Future Work

We plan to extend the threat model to assume malicious parties, formally. For implementation, we would like to experimentally verify the improvement of key-switching proposed by [27] (discussed in Appendix A). Another interesting topic might be to extend the message space to more than Boolean.

## Appendix

## A Possible Improvement of Key-Switching

In [27], authors suggest to pre-compute multiples of key-switching keys: the aim is to decrease the contribution of noise from the key-switching keys. On the one hand, the performance may improve by choosing more efficient parameters, on the other hand, the size of key-switching keys may grow significantly.

Instead of encrypting $\mathbf{z}_i^{(q)*}\mathbf{g}'$, authors suggest to encrypt its multiples by integers in $[1, {}^{B'}\!/2]$. Then, in the `KeySwitch` algorithm, instead of multiplication of a key-switching key $\mathsf{KS}_i$ by decomposition digits of $\mathbf{g}'^{-1}(a_i')$, cf. (4.2), an appropriate pre-computed multiple of $\mathsf{KS}_i$ is used (with appropriate sign). In case $B'$ is "too big" for practical considerations, we rather suggest to encrypt multiples of $\mathbf{z}_i^{(q)*}\mathbf{g}'$ only by powers of two in $[1, {}^{B'}\!/2]$, and then combine these multiples to reach the digits of $\mathbf{g}'^{-1}(a_i')$. For this purpose, we suggest to employ a signed binary representation with the lowest Hamming weight, also referred to as the *Non-Adjacent Form* (NAF; [18]).

## B Proofs of Noise Analysis

### B.1 Noise Growth of Homomorphic Product

*Lemma* 4.1. Given $\mathsf{RGSW}$ sample $\mathsf{BK}$ generated by the `AKÖ.BlindRotKeyGen` algorithm, which encrypts constant polynomial $s \in \mathbb{Z}^{(N)}[X]$ under the common $\mathsf{RLWE}$ key $Z = \sum_p z^{(p)}$, and $\mathsf{RLWE}$ sample $\bar{\mathbf{c}} = (b, a)$ that encrypts $m \in \mathbb{T}^{(N)}[X]$ under the same key, the `AKÖ.Prod` algorithm returns $\mathsf{RLWE}$ sample $\bar{\mathbf{c}}' = (b', a')$ that encrypts $s \cdot m$ under $Z$ with additional noise $e_{\mathtt{Prod}}$, given by $\langle \bar{\mathbf{Z}}, \bar{\mathbf{c}}' \rangle = s \cdot \langle \bar{\mathbf{Z}}, \bar{\mathbf{c}} \rangle + e_{\mathtt{Prod}}$, for which

$$\mathsf{Var}[e_{\mathtt{Prod}}] \approx \underbrace{NdV_B\beta^2(3 + 6pkN)}_{\text{BK error}} + \underbrace{s^2\varepsilon^2(1 + 2pkN)}_{\text{decomp. error}}, \tag{17}$$

where

- $\varepsilon^2 := {}^{1}\!/_{12B^{2d}}$ is the variance of (real-valued) uniform distribution on $[-{}^{1}\!/_{2B^d}, {}^{1}\!/_{2B^d})$,
- $V_B := {}^{(B^2+2)}\!/_{12}$ is the mean of squares of (integer valued) uniform distribution on $[-{}^{B}\!/2, {}^{B}\!/2)$ (assuming $B$ is even),
- other notation and parameters are as per the `AKÖ.Setup` algorithm, and
- we refer to the two terms as the *blind-rotate key error* and the *decomposition error*, respectively.

If this error is sufficiently small, it holds $\langle \bar{\mathbf{Z}}, \bar{\mathbf{c}}' \rangle \approx s \cdot \langle \bar{\mathbf{Z}}, \bar{\mathbf{c}} \rangle$, i.e., the `AKÖ.Prod` algorithm is indeed multiplicatively homomorphic.

*Proof.* We unfold the construction of BK and multiplication within the `AKÖ.Prod` algorithm:

$$\begin{aligned}\bar{\mathbf{c}}' = \Big(&\Big\langle \mathbf{g}^{-1}(b), -r\cdot\mathbf{B} + s\cdot\mathbf{g} + \mathbf{e}_1\Big\rangle + \Big\langle \mathbf{g}^{-1}(a), -r\cdot\mathbf{B}' + \mathbf{e}_1'\Big\rangle,\\ &\Big\langle \mathbf{g}^{-1}(b), -r\cdot\mathbf{a} + \mathbf{e}_2\Big\rangle + \Big\langle \mathbf{g}^{-1}(a), -r\cdot\mathbf{a}' + s\cdot\mathbf{g} + \mathbf{e}_2'\Big\rangle\Big). \end{aligned} \tag{18}$$

Then we write

$$\begin{aligned}\big\langle\bar{\mathbf{Z}},\bar{\mathbf{c}}'\big\rangle &= \big\langle \mathbf{g}^{-1}(b), -r\mathbf{B} + s\,\mathbf{g} + \mathbf{e}_1\big\rangle + \big\langle \mathbf{g}^{-1}(a), -r\mathbf{B}' + \mathbf{e}_1'\big\rangle +\\ &\quad + Z\big\langle \mathbf{g}^{-1}(b), -r\mathbf{a} + \mathbf{e}_2\big\rangle + Z\big\langle \mathbf{g}^{-1}(a), -r\mathbf{a}' + s\,\mathbf{g} + \mathbf{e}_2'\big\rangle =\\ &= \Big\langle \mathbf{g}^{-1}(b), r\sum_{p=1}^{k} z^{(p)}\mathbf{a} - r\sum_{p=1}^{k}\mathbf{e}^{(p)} + s\,\mathbf{g} + \mathbf{e}_1\Big\rangle + \Big\langle \mathbf{g}^{-1}(a), r\sum_{p=1}^{k} z^{(p)}\mathbf{a}' - r\sum_{p=1}^{k}\mathbf{e}'^{(p)} + \mathbf{e}_1'\Big\rangle +\\ &\quad + \big\langle \mathbf{g}^{-1}(b), -Zr\mathbf{a} + \mathbf{e}_2\big\rangle + \big\langle \mathbf{g}^{-1}(a), -Zr\mathbf{a}' + Zs\,\mathbf{g} + Z\mathbf{e}_2'\big\rangle =\\ &= \Big\langle \mathbf{g}^{-1}(b), s\,\mathbf{g} - r\sum_{p=1}^{k}\mathbf{e}^{(p)} + \mathbf{e}_1 + \mathbf{e}_2\Big\rangle + \Big\langle \mathbf{g}^{-1}(a), Zs\,\mathbf{g} - r\sum_{p=1}^{k}\mathbf{e}'^{(p)} + \mathbf{e}_1' + Z\mathbf{e}_2'\Big\rangle =\\ &= s\Big(\underbrace{\big\langle \mathbf{g}^{-1}(b), \mathbf{g}\big\rangle}_{\approx b} \pm b\Big) + Zs\Big(\underbrace{\big\langle \mathbf{g}^{-1}(a), \mathbf{g}\big\rangle}_{\approx a} \pm a\Big) +\\ &\quad + \Big\langle \mathbf{g}^{-1}(b), -r\sum_{p=1}^{k}\mathbf{e}^{(p)} + \mathbf{e}_1 + \mathbf{e}_2\Big\rangle + \Big\langle \mathbf{g}^{-1}(a), -r\sum_{p=1}^{k}\mathbf{e}'^{(p)} + \mathbf{e}_1' + Z\mathbf{e}_2'\Big\rangle =\\ &= s\cdot\underbrace{(b + Za)}_{\langle\bar{\mathbf{Z}},\bar{\mathbf{c}}\rangle} +\end{aligned}$$

$$+\, s\cdot\underbrace{\Big(\overbrace{\langle \mathbf{g}^{-1}(b), \mathbf{g}\rangle_d - b}^{\mathsf{Var}[\cdot]=\varepsilon^2} + Z\big(\overbrace{\langle \mathbf{g}^{-1}(a), \mathbf{g}\rangle_d - a}^{\mathsf{Var}[\cdot]=\varepsilon^2}\big)\Big)}_{\text{decomp. errors}} - \tag{19}$$

$$- r\sum_{p=1}^{k}\langle \mathbf{g}^{-1}(b), \mathbf{e}^{(p)}\rangle_d - r\sum_{p=1}^{k}\langle \mathbf{g}^{-1}(a), \mathbf{e}'^{(p)}\rangle_d + \tag{20}$$

$$+ \langle \mathbf{g}^{-1}(b), \mathbf{e}_1 + \mathbf{e}_2\rangle_d + \langle \mathbf{g}^{-1}(a), \mathbf{e}_1' + Z\mathbf{e}_2'\rangle_d. \tag{21}$$

Note that

- the decomposition error has a uniform distribution on $[-1/2B^d, 1/2B^d)$, hence variance of $\varepsilon^2$;
- decomposition digits have a uniform distribution on $[-B/2, B/2)$, hence mean of squares of $V_B$.

We evaluate the variance for each term and the result follows:

**(19):** $\mathsf{Var}[\cdot] = s^2\varepsilon^2\big(1+2pkN\big)$, since we multiply the second term by $Z = \sum_p z^{(p)}$, which is a sum of $k$ polynomials, each with $N$ coefficients with variance of $2p$ and zero mean;

**(20):** $\mathsf{Var}[\cdot] = 2\cdot 2pkN^2 dV_B\beta^2$, since there are two identical independent terms, where we multiply a polynomial $r$, which has $N$ coefficients with variance of $2p$, with a sum of $k$ independent terms, each of which is an inner product of size $d$, where we multiply two polynomials of $N$ coefficients: one of them has $V_B$ mean of squares and the other has variance of $\beta^2$ and zero mean[4];

[4]It holds $\mathsf{Var}[X\cdot Y] = \mathrm{E}[X^2]\cdot\mathsf{Var}[Y]$ for independent variables with $\mathrm{E}[Y] = 0$.

(21): $\mathsf{Var}[\cdot] = (3 + 2pkN)NdV_B\beta^2$, since we sum four independent error terms with variance of $\beta^2$, one of which is multiplied by $Z$ (a sum of $k$ polynomials of $N$ coefficients with variance of $2p$).

For the dynamic variant, we exchange $\mathbf{e}_1 \to \sum_p \mathbf{e}_1^{(p)}$, respectively for $\mathbf{e}'_1$, in the proof. Here, $\sum_p \mathbf{e}_1^{(p)}$ emerges from the sum of $\mathbf{b}_{q,j}^{\Delta(p)}$; cf. (4.7). This changes $(3 + 2pkN) \to (1 + 2k + 2pkN)$ in (21). □

## B.2 Noise Growth of Blind-Rotate

*Theorem* 4.2. The `AKÖ.BlindRotate` algorithm returns a sample with noise variance given by

$$\mathsf{Var}[\langle \bar{\mathbf{Z}}, \mathsf{ACC}\rangle] \approx \underbrace{knNdV_B\beta^2(3 + 6pkN)}_{\text{BK error}} + \underbrace{1/2 \cdot kn\varepsilon^2(1 + 2pkN)}_{\text{decomp. error}} + \underbrace{\mathsf{Var}[tv]}_{\text{usually } 0} . \tag{22}$$

The resulting ACC encrypts $X^{\langle \bar{\mathbf{s}}, (\tilde{b}, \tilde{\mathbf{a}})\rangle} \cdot tv$.

*Proof.* In the `AKÖ.BlindRotate` algorithm, the (usually noiseless) sample $tv$ gets gradually multiplied by BK's, we write:

$$\begin{aligned} &\langle \bar{\mathbf{Z}}, \mathsf{ACC} + \mathtt{AKÖ.Prod}(\mathsf{BK}, X^a \cdot \mathsf{ACC} - \mathsf{ACC})\rangle = \\ &= \langle \bar{\mathbf{Z}}, \mathsf{ACC}\rangle + s \cdot \langle \bar{\mathbf{Z}}, X^a \cdot \mathsf{ACC} - \mathsf{ACC}\rangle + e_{\mathtt{Prod}}(s) = \\ &= \langle \bar{\mathbf{Z}}, X^{s \cdot a} \cdot \mathsf{ACC}\rangle + e_{\mathtt{Prod}}(s), \end{aligned} \tag{23}$$

i.e., with each step, the noise grows by the additive term $e_{\mathtt{Prod}}(s)$. The length of the common LWE key $\mathbf{s}$ is $kn$, the mean of squares of $\mathbf{s}_i$ is $1/2$, hence the result follows. □

## B.3 Noise Growth of Key-Switching

*Theorem* 4.3. The `AKÖ.KeySwitch` algorithm returns a sample that encrypts the same message as the input sample, while changing the key from $\mathbf{Z}^*$ to $\mathbf{s}$, with additional noise $e_{\mathsf{KS}}$, given by $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}''\rangle = \langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}'\rangle + e_{\mathsf{KS}}$, for which

$$\mathsf{Var}[e_{\mathsf{KS}}] \approx \underbrace{Nkd'V_{B'}\beta'^2}_{\text{KS error}} + \underbrace{2pkN\varepsilon'^2}_{\text{decomp. error}} . \tag{24}$$

If the error is sufficiently small, it holds $\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}''\rangle \approx \langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}'\rangle$.

*Proof.* We write (for clarity with indexes that indicate the length of inner products):

$$\begin{aligned}
\langle \bar{\mathbf{s}}, \bar{\mathbf{c}}'' \rangle &= \Big\langle (1,\mathbf{s}), (b', \mathbf{0}) + \sum_{j=1}^{N} \mathbf{g}'^{-1}(a'_j)^T \cdot \mathsf{KS}_j \Big\rangle_{1+kn} = \\
&= b' + \sum_{j=1}^{N} \Big\langle (1,\mathbf{s}), \mathbf{g}'^{-1}(a'_j)^T \cdot \Big[ -\mathbf{A}_j\mathbf{s} + \mathbf{Z}^*_j\mathbf{g}' + \underbrace{\sum_{p=1}^{k} \mathbf{e}^{(p)}_j}_{\mathbf{e}_j} \Big| \mathbf{A}_j \Big] \Big\rangle_{1+kn} = \\
&= b' - \sum_{j=1}^{N} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{A}_j\mathbf{s} \big\rangle_{d'} + \sum_{j=1}^{N} \mathbf{Z}^*_j \Big( \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{g}' \big\rangle_{d'} \pm a'_j \Big) + \sum_{j=1}^{N} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{e}_j \big\rangle_{d'} + \\
&\quad + \sum_{j=1}^{N} \big\langle \mathbf{s}, \mathbf{g}'^{-1}(a'_j)^T \cdot \mathbf{A}_j \big\rangle_{kn} = \\
&= \underbrace{b' + \langle \mathbf{Z}^*, \mathbf{a}' \rangle_N}_{\langle \bar{\mathbf{Z}}^*, \bar{\mathbf{c}}' \rangle} + \sum_{j=1}^{N} \mathbf{Z}^*_j \Big( \underbrace{\big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{g}' \big\rangle_{d'} - a'_j}_{\text{decomp. error}} \Big) + \sum_{j=1}^{N} \big\langle \mathbf{g}'^{-1}(a'_j), \mathbf{e}_j \big\rangle_{d'}, \qquad (25)
\end{aligned}$$

while the decomposition error term has variance of $2pkN\varepsilon'^2$ and the other term has variance of $Nkd'V_{B'}\beta'^2$ (n.b., $\mathbf{e}_j$ is a sum of $k$ error terms), where $\varepsilon'^2 := 1/12B'^{2d'}$ and $V_{B'} := (B'^2+2)/12$ are respectively analogical to $\varepsilon^2$ and $V_B$, introduced in Lemma 4.1. The result follows. □

## C Security Estimates of (R)LWE Parameters

### C.1 Parameters of CCS [27] and KMS [93]

We outline the usage of the `lattice-estimator` [8] for the purpose of security estimation of parameters of (R)LWE over the torus, considering a finite representation of the torus (the baseline Julia implementation [106] employs 32 bits for LWE and 64 bits for RLWE). As the final security estimate, we take the largest `rop` value – the documentation of the estimator (e.g., in `/estimator/lwe_guess.py`) states:

```
rop: Total number of word operations (approx. CPU cycles)
```

Below, we present a sample output of the estimator on the LWE parameters that are shared by CCS and KMS:

```
sage: from estimator import *
      from estimator.lwe_parameters import LWEParameters
      from estimator.nd import NoiseDistribution as ND
sage: LWE.estimate(LWEParameters(n=560, q=2^32, Xs=ND.Uniform(0,1),
      Xe=ND.DiscreteGaussianAlpha(3.05*10^(-5), 2^32), m=sage.all.oo))
# bkw                   :: rop: approx. 2^144.3, m:   ...
# usvp                  :: rop: approx. 2^103.2, red: ...
# --- other lines with rop higher than 100 ---
# dual_hybrid           :: rop: approx. 2^99.4, mem:  ...
```

from which we read 99.4 ≈ 100 bits of security. For the RLWE parameters of CCS and KMS, we obtain 106.7 and 110.6 bits, respectively, both for the `dual_hybrid` attack.

## C.2 Parameters of Our Scheme

We generate our parameters using our highly experimental tool[5], therefore, we provide the parameters "as-is", with no guarantees on their optimality. In our tool, we set the target security at 100 bits, which we verify with the `lattice-estimator` [8]. Recall that for RLWE keys, we suggest to use a ternary distribution $\zeta_p\colon (-1,0,1) \to (p, 1-2p, p)$ with $p \approx 0.1135$; see Section 4.3.2, where we also mention some recent attacks on sparse keys [31, 123], both of which are considered by the estimator.

In case a new attack on sparse keys occurs, we suggest to increase the value of $p$, until the key is not "sparse" – e.g., in the `lattice-estimator` (e.g., in `estimator/nd.py`), authors consider "sparse" keys as follows:

```
We consider a~distribution "sparse" if its density is < 1/2.
```

Then, new parameters would need to be generated to reflect this change, however, we believe that this would pose no obstacle. Indeed, the most important property of the ternary keys with respect to the noise growth is that they are zero-centered, which remains untouched.

Below, we provide an output of the estimator for our RLWE parameters, taking into account the ternary distribution with $p = 0.113546$, which corresponds to 116.27 out of 1 024:

```
# -- N = 1024 -------------------------------------------------------------------
sage: LWE.estimate(LWEParameters(n=1024, q=2^64, Xs=ND.SparseTernary(1024,
      p=116.27), Xe=ND.DiscreteGaussianAlpha(2^(-30.7), 2^64), m=sage.all.oo))
# usvp          :: rop: approx. 2^99.9    (other /higher/ values omitted)
```

We also check the estimate for uniform binary key distribution, replacing the ternary distribution, and we obtain `rop: approx. 2^98.3`, i.e., the estimate is actually higher for the ternary distribution. Next, for $N = 2\,048$, we obtain:

```
# -- N = 2048    n.b., runs around 4 hours!    --------------------------------
sage: LWE.estimate(LWEParameters(n=2048, q=2^64, Xs=ND.SparseTernary(2048,
      p=2*116.27), Xe=ND.DiscreteGaussianAlpha(2^(-63.0), 2^64), m=sage.all.oo))
# bdd           :: rop: approx. 2^101.2
# bdd_hybrid    :: rop: approx. 2^101.2
```

Then, for selected LWE parameters, we obtain:

```
# -- n = 520 --------------------------------------------------------------------
sage: LWE.estimate(LWEParameters(n=520, q=2^64, Xs=ND.Uniform(0,1),
      Xe=ND.DiscreteGaussianAlpha(2^(-13.52), 2^64), m=sage.all.oo))
# dual_hybrid   :: rop: approx. 2^100.2    (other /higher/ values omitted)
```

We verify with `q=2^32`, so that we may represent torus in LWE with a 32-bit type:

```
sage: LWE.estimate(LWEParameters(n=520, q=2^32, Xs=ND.Uniform(0,1),
      Xe=ND.DiscreteGaussianAlpha(2^(-13.52), 2^32), m=sage.all.oo))
# dual_hybrid   :: rop: approx. 2^100.2
```

Other selected parameters give:

```
# -- n = 510 --------------------------------------------------------------------
sage: LWE.estimate(LWEParameters(n=510, q=2^64, Xs=ND.Uniform(0,1),
```

[5]Available at `https://gitlab.eurecom.fr/fakub/tfhe-param-testing` in the `mk-tfhe` branch.

```
      Xe=ND.DiscreteGaussianAlpha(2^(-13.26), 2^64), m=sage.all.oo))
# dual_hybrid   :: rop: approx. 2^99.8
# with q=2^32 also rop: approx. 2^99.8

# ...

# -- n = 670 -------------------------------------------------------------------
sage: LWE.estimate(LWEParameters(n=670, q=2^64, Xs=ND.Uniform(0,1),
      Xe=ND.DiscreteGaussianAlpha(2^(-17.42), 2^64), m=sage.all.oo))
# dual_hybrid   :: rop: approx. 2^104.9
# with q=2^32 also rop: approx. 2^104.9

# -- n = 740 -------------------------------------------------------------------
sage: LWE.estimate(LWEParameters(n=740, q=2^64, Xs=ND.Uniform(0,1),
      Xe=ND.DiscreteGaussianAlpha(2^(-19.24), 2^64), m=sage.all.oo))
# dual_hybrid   :: rop: approx. 2^106.7
```

For all parameter combinations, the `lattice-estimator` gives around or more than 100 bits of security, even with `q=2^32`.

# Conclusion

Since the discovery of *fully homomorphic encryption* in 2009, the performance has gradually improved over the years. However, since the introduction of TFHE in 2016 and CKKS in 2017, no fundamentally novel approach has been proposed. Hence, recent research efforts and advances in FHE are leaning towards improvements of these concepts which is no different in this work. For instance, there is much focus on *parallelization* (to push down the latency), *efficient implementation* (to prevent wasting resources on unnecessary operations/precision), or *batching methods* (to process more data at the same cost). Making all of that work together is a challenging task and there is still a long way until a massive deployment of FHE in the wild, in particular, due to the specificities of each and every real-world application.

In parallel to the improvements of base-line FHE schemes, we may observe attempts towards their extensions and variants of, or their usage in particular applications. Interestingly, for instance, the TFHE scheme originates as an effort to employ its predecessor—the FHEW scheme—in an *electronic voting system*. Other topics include but are not limited to: *verifiability* [55] (to prevent the cloud from cheating), *private information retrieval* [131] (PIR; to allow database access without disclosing the query), or a setup with *multiple parties* [97] (to allow processing data from multiple sources or to distribute trust).

## Summary of Contributions

The two main contributions of this thesis are the following: (i) *fast(est) homomorphic arithmetic* which employs parallelism, and (ii) *practical (hybrid) multi-key FHE* which enables applications of FHE for multiple parties.

### Fast(est) Homomorphic Arithmetic

Chapter 3 presents the *PARMESAN Library* (Parallel ARithMEticS over ENcrypted data) which implements integer arithmetic and other operations over TFHE-encrypted integers of arbitrary bit-length (aka. *multi-precision arithmetic*). Using a combination of signed binary integer representation, for which there exist parallel algorithms for integer addition, and a state-of-the-art implementation of the TFHE scheme, PARMESAN outperforms other algorithms that employ addition algorithm with carry – provided that a sufficient number of threads and long-enough inputs are given.

Particular speed-ups are achieved for scalar multiplication (i.e., multiplication of an encrypted number by an unencrypted one) thanks to the use of newly defined and practically evaluated *free-doubling addition-subtraction chains* (ASC$^*$s). The use of ASC$^*$s saves about 20 % of additions on average, compared to the standard double-and-add/sub method, provided that both employ the same Koyama-Tsuruoka recoding of the input scalar. The main advantage of the ASC$^*$-based method for scalar multiplication is its versatility – it is agnostic of the underlying implementation of the addition algorithm, i.e., it can be ported to other implementations of homomorphic arithmetic.

### Practical (Hybrid) Multi-Key FHE

Chapter 4 introduces a hybrid multi-key fully homomophic scheme named AKÖ. Compared to its predecessors – CCS and KMS schemes – AKÖ not only vastly outperforms them in terms of *latency* (in particular the CCS scheme), it can also be practically instantiated with an order of *hundreds of parties* and, at the same time, it introduces only a very *low rate of evaluation errors.* The drawback of AKÖ is that it imposes a pre-computation overhead which is not present in CCS nor in KMS.

AKÖ goes with a set of practical parameters for different numbers of parties, which are evaluated with respect to *reliability* in thorough experiments. The evaluation not only captures predicted errors that stem from the inherent LWE noise, but it also deals with errors imposed by concrete implementation choices – in particular, rounding errors of Fast Fourier Transform (FFT) are discussed. Interestingly, these rounding errors are shown to trigger a prohibitively high probability of an evaluation error in the case of the KMS scheme while the AKÖ scheme remains untouched by FFT errors (to a certain extent – countermeasures are proposed for more than 16 parties).

## Future Work

Related to the contents of this thesis, the following topics are left for further investigation:

**Parallelization.** To achieve practical usability of FHE, *low latency* needs to be achieved. Due to the still very high computational overhead of homomorphic operations, which does not seem to be pushed down easily, it is worth considering the distribution of the computational load among multiple workers (threads). However, not all operations allow for parallelization (e.g., blind-rotate of TFHE), hence, one may need to resort to parallelization of operations at a lower level (e.g., that of FFT in polynomial multiplication). On top of that, additional overhead might be induced (e.g., by switching from binary to signed binary integer representation as per Chapter 3), therefore, a careful comparison of various approaches must always be made.

**Multi-key FHE.** As pointed out, our scheme is not multi-key in the true sense of the word: indeed, there is a pre-computation overhead which is not supposed to be present in a truly multi-key scheme. On the other hand, experiments show that previous attempts are impractical for different reasons: CCS is prohibitively slow and KMS suffers from an enormous error growth, (at least) due to FFT errors. Therefore, the question of whether a practical truly multi-key fully homomorphic scheme exists is raised.

**Other topics.** The prospective NP-completeness of $\mathsf{ASC}^*$ decision problem (i.e., the existence of an $\mathsf{ASC}^*$ of length $l$ for integer $k$; as discussed in Chapter 3) is left as an open problem.

## Other Relevant Research Directions

We believe that FHE is capable of making its way through toward a mass deployment in the wild in a near future. Until then, we expect that the following research directions would attract attention and therefore might contribute to the overall improvement of the applicability of FHE.

**Specialized hardware.** Very recently, a promising *hardware accelerator* named FPT [126] has been presented. Authors employ an FPGA[6] and currently, they achieve the lowest latency of TFHE bootstrapping of about 500 µs which is by almost two orders of magnitude faster than with a CPU. Their approach is based on a careful analysis of error growth in FFTs which is also the topic of Chapter 2. Another promising approach, which employs optoelectronics, is developed by Optalysys[7].

**Verifiability & zero-knowledge proofs.** There emerge attempts to provide FHE schemes with a certain form of *verifiability*, either based on algebraic properties [54, 15], or based on zero-knowledge proofs [55]. However, the computational overhead needs to be pushed down significantly.

[6]In simple words, *Field Programmable Gate Array* (FPGA) is a configurable hardware.
[7]White-papers are available at `https://optalysys.com/white-papers`. Accessed 2023-05-11.

# Bibliography

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pages 1–84, 2019.

[2] Abbas Acar, Hidayet Aksu, A Selcuk Uluagac, and Mauro Conti. A survey on homomorphic encryption schemes: Theory and implementation. *ACM Computing Surveys (Csur)*, 51(4):1–35, 2018.

[3] Miklós Ajtai. Generating hard instances of lattice problems. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 99–108, 1996.

[4] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. OpenFHE: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

[5] Ahmad Al Badawi, Bharadwaj Veeravalli, and Khin Mi Mi Aung. Efficient polynomial multiplication via modified discrete galois transform and negacyclic convolution. In *Future of Information and Communication Conference*, pages 666–682. Springer, 2018.

[6] Martin Albrecht, Shi Bai, and Léo Ducas. A subfield lattice attack on overstretched ntru assumptions: Cryptanalysis of some fhe and graded encoding schemes. In *Advances in Cryptology–CRYPTO 2016: 36th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2016, Proceedings, Part I*, pages 153–178. Springer, 2016.

[7] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, et al. Homomorphic encryption standard. *Protecting privacy through homomorphic encryption*, pages 31–62, 2021.

[8] Martin R Albrecht and contributors. Security Estimates for Lattice Problems. `https://github.com/malb/lattice-estimator`, 2022.

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

[10] Frederik Armknecht, Stefan Katzenbeisser, and Andreas Peter. Group homomorphic encryption: characterizations, impossibility results, and applications. *Designs, codes and cryptography*, 67(2):209–232, 2013.

[11] Algirdas Avizienis. Signed-digit numbe representations for fast parallel arithmetic. *IRE Transactions on electronic computers*, (3):389–400, 1961.

[12] David A Barrington. Bounded-width polynomial-size branching programs recognize exactly those languages in nc. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 1–5, 1986.

[13] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Parameter optimization & larger precision for (t)fhe. 2022.

[14] Daniel J Bernstein. Multidigit multiplication for mathematicians. 2001.

[15] Alexandre Bois, Ignacio Cascudo, Dario Fiore, and Dongwoo Kim. Flexible and efficient verifiable computation on encrypted data. In *Public-Key Cryptography–PKC 2021: 24th IACR International Conference on Practice and Theory of Public Key Cryptography, Virtual Event, May 10–13, 2021, Proceedings, Part II*, pages 528–558. Springer, 2021.

[16] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *TCC*, volume 3378, pages 325–341. Springer, 2005.

[17] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder VL Pereira, and Nigel P Smart. FINAL: Faster FHE instantiated with NTRU and LWE. *Cryptology ePrint Archive, Report 2022/074*, 2022.

[18] Andrew D Booth. A signed binary multiplication technique. *The Quarterly Journal of Mechanics and Applied Mathematics*, 4(2):236–240, 1951.

[19] Joppe W Bos, Kristin Lauter, Jake Loftus, and Michael Naehrig. Improved security for a ring-based fully homomorphic encryption scheme. In *Cryptography and Coding: 14th IMA International Conference, IMACC 2013, Oxford, UK, December 17-19, 2013. Proceedings 14*, pages 45–64. Springer, 2013.

[20] Florian Bourse, Michele Minelli, Matthias Minihold, and Pascal Paillier. Fast homomorphic evaluation of deep discretized neural networks. In *Annual International Cryptology Conference*, pages 483–512. Springer, 2018.

[21] Florian Bourse, Olivier Sanders, and Jacques Traoré. Improved secure integer comparison via homomorphic encryption. In *Cryptographers' Track at the RSA Conference*, pages 391–416. Springer, 2020.

[22] Zvika Brakerski. Fully homomorphic encryption without modulus switching from classical gapsvp. In *Annual Cryptology Conference*, pages 868–886. Springer, 2012.

[23] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. *ACM Transactions on Computation Theory (TOCT)*, 6(3):13, 2014.

[24] Zvika Brakerski and Renen Perlman. Lattice-based fully dynamic multi-key fhe with short ciphertexts. In *Annual International Cryptology Conference*, pages 190–213. Springer, 2016.

[25] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. New techniques for multi-value input homomorphic evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 106–126. Springer, 2019.

[26] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Improved bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 34–54. Springer, 2019.

[27] Hao Chen, Ilaria Chillotti, and Yongsoo Song. Multi-key homomorphic encryption from tfhe. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 446–472. Springer, 2019.

[28] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 395–412, 2019.

[29] Long Chen, Zhenfeng Zhang, and Xueqing Wang. Batched multi-hop multi-key fhe from ring-lwe with compact ciphertext extension. In *Theory of Cryptography Conference*, pages 597–627. Springer, 2017.

[30] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. Bootstrapping for approximate homomorphic encryption. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 360–384. Springer, 2018.

[31] Jung Hee Cheon, Minki Hhan, Seungwan Hong, and Yongha Son. A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret lwe. *IEEE Access*, 7:89497–89506, 2019.

[32] Jung Hee Cheon, Jinhyuck Jeong, and Changmin Lee. An algorithm for ntru problems and cryptanalysis of the ggh multilinear map without a low-level encoding of zero. *LMS Journal of Computation and Mathematics*, 19(A):255–266, 2016.

[33] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 409–437. Springer, 2017.

[34] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *international conference on the theory and application of cryptology and information security*, pages 3–33. Springer, 2016.

[35] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. TFHE: fast fully homomorphic encryption over the torus. *Journal of Cryptology*, 33(1):34–91, 2020.

[36] Ilaria Chillotti, Marc Joye, and Pascal Paillier. Programmable bootstrapping enables efficient homomorphic inference of deep neural networks. In *International Symposium on Cyber Security Cryptography and Machine Learning*, pages 1–19. Springer, 2021.

[37] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 670–699. Springer, 2021.

[38] Catherine Y Chow and James E Robertson. Logical design of a redundant binary adder. In *1978 IEEE 4th Symposium onomputer Arithmetic (ARITH)*, pages 109–115. IEEE, 1978.

[39] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. Eva improved: Compiler and extension library for ckks. In *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 43–55, 2021.

[40] Michael Clear and Ciaran McGoldrick. Multi-identity and multi-key leveled fhe from learning with errors. In *Annual Cryptology Conference*, pages 630–656. Springer, 2015.

[41] Henri Cohen, Gerhard Frey, Roberto Avanzi, Christophe Doche, Tanja Lange, Kim Nguyen, and Frederik Vercauteren. *Handbook of Elliptic and Hyperelliptic Curve Cryptography, Second Edition.* Chapman & Hall/CRC, 2nd edition, 2012.

[42] Concrete: State-of-the-art TFHE library for boolean and integer arithmetics (v0.2). `https://docs.zama.ai/concrete/`, 2022.

[43] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965.

[44] Anamaria Costache, Benjamin R Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. On the precision loss in approximate homomorphic encryption. *Cryptology ePrint Archive*, 2022.

[45] Richard Crandall and Carl B Pomerance. *Prime numbers: a computational perspective*, volume 182. Springer Science & Business Media, 2006.

[46] Richard E Crandall. Integer convolution via split-radix fast galois transform. *Center for Advanced Computation Reed College*, 1999.

[47] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. Eva: An encrypted vector arithmetic language and compiler for efficient homomorphic computation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 546–561, 2020.

[48] Jintai Ding and Dieter Schmidt. Rainbow, a new multivariable polynomial signature scheme. In *International Conference on Applied Cryptography and Network Security*, pages 164–175. Springer, 2005.

[49] Peter Downey, Benton Leong, and Ravi Sethi. Computing sequences with addition chains. *SIAM Journal on Computing*, 10(3):638–646, 1981.

[50] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[51] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

[52] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[53] Junfeng Fan and Frederik Vercauteren. Somewhat Practical Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2012/144, 2012.

[54] Dario Fiore, Rosario Gennaro, and Valerio Pastro. Efficiently verifiable computation on encrypted data. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 844–855. ACM, 2014.

[55] Dario Fiore, Anca Nitulescu, and David Pointcheval. Boosting verifiable computation on encrypted data. In *Public-Key Cryptography–PKC 2020: 23rd IACR International Conference on Practice and Theory of Public-Key Cryptography, Edinburgh, UK, May 4–7, 2020, Proceedings, Part II 23*, pages 124–154. Springer, 2020.

[56] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. Basalisc: Flexible asynchronous hardware accelerator for fully homomorphic encryption. *arXiv preprint arXiv:2205.14017*, 2022.

[57] W Morven Gentleman and Gordon Sande. Fast fourier transforms: for fun and profit. In *Proceedings of the November 7-10, 1966, fall joint computer conference*, pages 563–578, 1966.

[58] Craig Gentry. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pages 169–178, 2009.

[59] Craig Gentry and Shai Halevi. Implementing gentry's fully-homomorphic encryption scheme. In *Advances in Cryptology–EUROCRYPT 2011: 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings 30*, pages 129–148. Springer, 2011.

[60] Craig Gentry, Shai Halevi, and Nigel P Smart. Better bootstrapping in fully homomorphic encryption. In *International Workshop on Public Key Cryptography*, pages 1–16. Springer, 2012.

[61] Craig Gentry, Amit Sahai, and Brent Waters. Homomorphic encryption from learning with errors: Conceptually-simpler, asymptotically-faster, attribute-based. In *Annual Cryptology Conference*, pages 75–92. Springer, 2013.

[62] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International Conference on Machine Learning*, pages 201–210. PMLR, 2016.

[63] Herman H Goldstine. A history of numerical analysis from the 16th through the 19th century. *Bull. Amer. Math. Soc*, 1:388–390, 1979.

[64] Shruthi Gorantala, Rob Springer, Sean Purser-Haskell, William Lam, Royce Wilson, Asra Ali, Eric P Astor, Itai Zukerman, Sam Ruth, Christoph Dibak, et al. A general purpose transpiler for fully homomorphic encryption. *arXiv preprint arXiv:2106.07893*, 2021.

[65] Torbjörn Granlund and the GMP development team. GNU MP: The GNU Multiple Precision Arithmetic Library, 2022.

[66] Antonio Guimarães, Edson Borin, and Diego F Aranha. Revisiting the functional bootstrap in TFHE. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 229–253, 2021.

[67] Shai Halevi and Victor Shoup. Design and implementation of a homomorphic-encryption library. *IBM Research (Manuscript)*, 6(12-15):8–36, 2013.

[68] Shai Halevi and Victor Shoup. Bootstrapping for helib. *Journal of Cryptology*, 34(1):7, 2021.

[69] William Hart, Fredrik Johansson, and Sebastian Pancratz. FLINT: Fast Library for Number Theory. `https://www.flintlib.org`, 2011.

[70] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *International Algorithmic Number Theory Symposium*, pages 267–288. Springer, 1998.

[71] Jeffrey Hoffstein, Jill Pipher, and Joseph H Silverman. Ntru: A ring-based public key cryptosystem. In *Algorithmic Number Theory: Third International Symposiun, ANTS-III Portland, Oregon, USA, June 21–25, 1998 Proceedings*, pages 267–288. Springer, 2006.

[72] homenc. HElib. `https://github.com/homenc/HElib`, 2023.

[73] Yuval Ishai and Anat Paskin. Evaluating branching programs on encrypted data. In *Theory of Cryptography: 4th Theory of Cryptography Conference, TCC 2007, Amsterdam, The Netherlands, February 21-24, 2007. Proceedings 4*, pages 575–594. Springer, 2007.

[74] David Jao and Luca De Feo. Towards quantum-resistant cryptosystems from supersingular elliptic curve isogenies. In *International Workshop on Post-Quantum Cryptography*, pages 19–34. Springer, 2011.

[75] Marc Joye. Sok: Fully homomorphic encryption over the [discretized] torus. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, pages 661–692, 2022.

[76] Marc Joye and Pascal Paillier. Blind rotation in fully homomorphic encryption with extended keys. In *Cyber Security, Cryptology, and Machine Learning: 6th International Symposium, CSCML 2022, Be'er Sheva, Israel, June 30–July 1, 2022, Proceedings*, pages 1–18. Springer, 2022.

[77] Charanjit S Jutla and Nathan Manohar. Sine series approximation of the mod function for bootstrapping of approximate he. In *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*, pages 491–520. Springer, 2022.

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

[79] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. Asymptotically Faster Multi-Key Homomorphic Encryption from Homomorphic Gadget Decomposition. Cryptology ePrint Archive, Paper 2022/347, 2022.

[80] Jakub Klemsa. Benchmarking FFNT. `https://gitlab.fit.cvut.cz/klemsjak/ffnt-benchmark`, 2021.

[81] Jakub Klemsa. Fast and error-free negacyclic integer convolution using extended fourier transform. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings*, pages 282–300. Springer, 2021.

[82] Jakub Klemsa. TFHE Parameter Setup for Effective and Error-Free Neural Network Prediction on Encrypted Data. In *Intelligent Computing*, pages 702–721. Springer, 2021.

[83] Jakub Klemsa. Hitchhiker's guide to the tfhe scheme. 2023.

[84] Jakub Klemsa, Lukáš Kencl, and Tomáš Vaněk. VeraGreg: A Framework for Verifiable Privacy-Preserving Data Aggregation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1820–1825. IEEE, 2018.

[85] Jakub Klemsa and Martin Novotný. Exploiting Linearity in White-Box AES with Differential Computation Analysis. In *Science and Information Conference*, pages 404–419. Springer, 2020.

[86] Jakub Klemsa and Martin Novotný. WTFHE: neural-netWork-ready Torus Fully Homomorphic Encryption. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2020.

[87] Jakub Klemsa and Melek Önen. Parallel Operations over TFHE-Encrypted Multi-Digit Integers. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, CODASPY '22, page 288–299, New York, NY, USA, 2022. Association for Computing Machinery.

[88] Jakub Klemsa and Melek Önen. PARMESAN: Parallel ARithMEticS over ENcrypted data. 2023.

[89] Jakub Klemsa and Ivana Trummová. Security Notions for the VeraGreg Framework and Their Reductions. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pages 8–20. IEEE, 2020.

[90] Jakub Klemsa, Melek Önen, and Yavuz Akın. A Practical TFHE-Based Multi-Key Homomorphic Encryption with Linear Complexity and Low Noise Growth, 2023.

[91] Peter Kornerup. Necessary and sufficient conditions for parallel, constant time conversion and addition. In *Proceedings 14th IEEE Symposium on Computer Arithmetic (Cat. No. 99CB36336)*, pages 152–156. IEEE, 1999.

[92] Kenji Koyama and Yukio Tsuruoka. Speeding up elliptic cryptosystems by using a signed binary window method. In *Annual International Cryptology Conference*, pages 345–357. Springer, 1992.

[93] Hyesun Kwak, Seonhong Min, and Yongsoo Song. Towards Practical Multi-key TFHE: Parallelizable, Key-Compatible, Quasi-linear Complexity. Cryptology ePrint Archive, Paper 2022/1460, 2022.

[94] Vernam Lab. CUDA-accelerated Fully Homomorphic Encryption Library. `https://github.com/vernamlab/cuFHE`, 2018.

[95] 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. In *Advances in Cryptology–EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part III*, pages 227–256. Springer, 2023.

[96] Helger Lipmaa. On the cca1-security of elgamal and damgård's elgamal. In *Information Security and Cryptology: 6th International Conference, Inscrypt 2010, Shanghai, China, October 20-24, 2010, Revised Selected Papers 6*, pages 18–35. Springer, 2011.

[97] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*, pages 1219–1234, 2012.

[98] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 1–23. Springer, 2010.

[99] Microsoft. SEAL (release 4.1). `https://github.com/Microsoft/SEAL`, January 2023.

[100] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. Multiparty homomorphic encryption from ring-learning-with-errors. *Proceedings on Privacy Enhancing Technologies*, pages 291–311, 2021.

[101] Christian Vincent Mouchet, Jean-Philippe Bossuat, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. Lattigo: A multiparty homomorphic encryption library in go. In *Proceedings of the 8th Workshop on Encrypted Computing and Applied Homomorphic Cryptography*, number CONF, pages 64–70, 2020.

[102] Pratyay Mukherjee and Daniel Wichs. Two round multiparty computation via multi-key fhe. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 735–763. Springer, 2016.

[103] Fast Fourier transform in x86 assembly. `https://www.nayuki.io/page/fast-fourier-transform-in-x86-assembly`, 2021. Accessed: 2021-01-30.

[104] NIST. NIST's Post-Quantum Cryptography Program Enters "Selection Round". `https://www.nist.gov/news-events/news/2020/07/nists-post-quantum-cryptography-program-enters-selection-round`, 2020.

[105] NuCypher. A GPU implementation of fully homomorphic encryption on torus. `https://github.com/nucypher/nufhe`, 2022.

[106] NuCypher. TFHE.jl. `https://github.com/nucypher/TFHE.jl`, 2022.

[107] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *Advances in Cryptology–EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2–6, 1999 Proceedings 18*, pages 223–238. Springer, 1999.

[108] Palisade. PALISADE Lattice Cryptography Library. `https://gitlab.com/palisade/palisade-release`, 2022.

[109] PARMESAN: Parallel ARithMEticS on tfhe ENcrypted data. `https://github.com/fakub/parmesan`, 2021.

[110] Chris Peikert and Sina Shiehian. Multi-key fhe from lwe, revisited. In *Theory of cryptography conference*, pages 217–238. Springer, 2016.

[111] Hilder VL Pereira and Nigel P Smart. Final: Faster fhe instantiated with ntru and lwe. In *Advances in Cryptology–ASIACRYPT 2022: 28th International Conference on the Theory and Application of Cryptology and Information Security, Taipei, Taiwan, December 5–9, 2022, Proceedings, Part II*, volume 13792, page 188. Springer Nature, 2023.

[112] John M Pollard. The fast fourier transform in a finite field. *Mathematics of computation*, 25(114):365–374, 1971.

[113] Oded Regev. On lattices, learning with errors, random linear codes, and cryptography. In *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*, pages 84–93, 2005.

[114] Jan Říha, Jakub Klemsa, and Martin Novotný. Multiprecision ANSI C Library for Implementation of Cryptographic Algorithms on Microcontrollers. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4. IEEE, 2019.

[115] Ronald L Rivest, Len Adleman, Michael L Dertouzos, et al. On data banks and privacy homomorphisms. *Foundations of secure computation*, 4(11):169–180, 1978.

[116] Ronald L Rivest, Adi Shamir, and Leonard Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[117] Tomas Sander, Adam Young, and Moti Yung. Non-interactive cryptocomputing for nc/sup 1. In *40th Annual Symposium on Foundations of Computer Science (Cat. No. 99CB37039)*, pages 554–566. IEEE, 1999.

[118] Arnold Schönhage and Volker Strassen. Schnelle multiplikation grosser zahlen. *Computing*, 7(3):281–292, 1971.

[119] Peter W Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, 1994.

[120] Victor Shoup et al. NTL: A library for doing number theory. `https://libntl.org`, 2001.

[121] SNUCrypto. HEAAN (release 1.1). `https://github.com/snucrypto/HEAAN`, 2018.

[122] SNUPrivacy. MK-TFHE. `https://github.com/SNUPrivacy/MKTFHE`, 2022.

[123] Yongha Son and Jung Hee Cheon. Revisiting the hybrid attack on sparse secret lwe and application to he parameters. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 11–20, 2019.

[124] TFHE: Fast Fully Homomorphic Encryption Library over the Torus. `https://github.com/tfhe/tfhe`, 2016.

[125] TrustworthyComputing. Multi-GPU acceleration for Fully Homomorphic Encryption. `https://github.com/TrustworthyComputing/REDcuFHE/`, 2022.

[126] Michiel Van Beirendonck, Jan-Pieter D'Anvers, and Ingrid Verbauwhede. FPT: a Fixed-Point Accelerator for Torus Fully Homomorphic Encryption. Cryptology ePrint Archive, Paper 2022/1635, 2022.

[127] Marten Van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully homomorphic encryption over the integers. In *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30–June 3, 2010. Proceedings 29*, pages 24–43. Springer, 2010.

[128] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. Heco: Automatic code optimizations for efficient fully homomorphic encryption. *arXiv preprint arXiv:2202.01649*, 2022.

[129] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. Sok: Fully homomorphic encryption compilers. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1092–1108. IEEE, 2021.

[130] J Wilson. The Multiply and Fourier Transform Unit: A Micro-Scale Optical Processor, 2022. Accessed: 2022-09-24.

[131] Xun Yi, Mohammed Golam Kaosar, Russell Paulet, and Elisa Bertino. Single-database private information retrieval from fully homomorphic encryption. *IEEE Transactions on Knowledge and Data Engineering*, 25(5):1125–1134, 2012.

[132] Zama. TFHE-rs: Pure Rust implementation of TFHE for boolean and integer arithmetics over encrypted data (v0.2). `https://github.com/zama-ai/tfhe-rs`, 2023.

# List of Publications

## Chapter 1: A Guide to the TFHE Scheme

- Chapter submitted to *Journal of Cryptographic Engineering* as:
  Jakub Klemsa. Hitchhiker's guide to the tfhe scheme. 2023 (in pre-print: doi:10.21203/rs.3.rs-2841900/v1).

- Preceding work:
  Jakub Klemsa. TFHE Parameter Setup for Effective and Error-Free Neural Network Prediction on Encrypted Data. In *Intelligent Computing*, pages 702–721. Springer, 2021.

- Preceding work:
  Jakub Klemsa and Martin Novotný. WTFHE: neural-netWork-ready Torus Fully Homomorphic Encryption. In *2020 9th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 2020.

## Chapter 2: Negacyclic Integer Convolution using Extended FFT

- Chapter published as:
  Jakub Klemsa. Fast and error-free negacyclic integer convolution using extended fourier transform. In *Cyber Security Cryptography and Machine Learning: 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8–9, 2021, Proceedings*, pages 282–300. Springer, 2021.

## Chapter 3: PARMESAN: Parallel ARithMEticS over ENcrypted data

- Chapter serves as a report for the PARMESAN Library [109]:
  Jakub Klemsa and Melek Önen. PARMESAN: Parallel ARithMEticS over ENcrypted data. 2023 (in pre-print: `https://ia.cr/2023/544`).

- Preceding work:
  Jakub Klemsa and Melek Önen. Parallel Operations over TFHE-Encrypted Multi-Digit Integers. In *Proceedings of the Twelveth ACM Conference on Data and Application Security and Privacy*, CODASPY '22, page 288–299, New York, NY, USA, 2022. Association for Computing Machinery.

## Chapter 4: TFHE-Based Multi-Key Homomorphic Encryption

- Chapter accepted for publication at *28th European Symposium on Research in Computer Security, The Hague, The Netherlands* (ESORICS 2023) as:
  Jakub Klemsa, Melek Önen, and Yavuz Akın. A Practical TFHE-Based Multi-Key Homomorphic Encryption with Linear Complexity and Low Noise Growth, 2023 (in pre-print: `https://ia.cr/2023/065`).

## Other Publications

- Revisited results of my Master's thesis:
  Jakub Klemsa and Martin Novotný. Exploiting Linearity in White-Box AES with Differential Computation Analysis. In *Science and Information Conference*, pages 404–419. Springer, 2020.

- Verifiable additive homomorphic encryption scheme (VeraGreg):
  Jakub Klemsa, Lukáš Kencl, and Tomáš Vaněk. VeraGreg: A Framework for Verifiable Privacy-Preserving Data Aggregation. In *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 1820–1825. IEEE, 2018

- Formal security for VeraGreg:
  Jakub Klemsa and Ivana Trummová. Security Notions for the VeraGreg Framework and Their Reductions. In *2020 Third ISEA Conference on Security and Privacy (ISEA-ISAP)*, pages 8–20. IEEE, 2020.

- Contributed as a supervisor of Jan Říha's Master's thesis:
  Jan Říha, Jakub Klemsa, and Martin Novotný. Multiprecision ANSI C Library for Implementation of Cryptographic Algorithms on Microcontrollers. In *2019 8th Mediterranean Conference on Embedded Computing (MECO)*, pages 1–4. IEEE, 2019.