PSL UNIVERSITÉ PARIS

THÈSE DE DOCTORAT
DE L'UNIVERSITÉ PSL

Préparée à l'École Normale Supérieure de Paris

Development of Optimized Operations for Homomorphic Cryptography

Soutenue par
Nicolas Bon
Le 14 Novembre 2025

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité
Informatique

Composition du jury :

Caroline FONTAINE Université Paris-Saclay	*Rapportrice*
Pierre-Alain FOUQUE Université Rennes 1	*Rapporteur*
Ilaria CHILLOTTI Desilo	*Examinatrice*
Adeline ROUX-LANGLOIS Normandie Université	*Examinatrice*
Renaud SIRDEY Commissariat à l'Energie Atomique	*Examinateur*
David POINTCHEVAL Ecole Normale Supérieure	*Directeur de thèse*
Sonia BELAÏD CryptoExperts	*Coencadrante de thèse*
Matthieu RIVAIN CryptoExperts	*Codirecteur de thèse*

1794 ENS | PSL

CRYPTOEXPERTS

Résumé

Dans cette thèse, nous étudions le chiffrement homomorphe, une technique cryptographique qui permet d'effectuer des calculs directement sur des données chiffrées, sans nécessiter de déchiffrement préalable. Ce domaine a connu un essor spectaculaire au cours des quinze dernières années, avec l'émergence de nombreux schémas de chiffrement de plus en plus performants. Néanmoins, les calculs homomorphes restent encore nettement plus coûteux que leurs équivalents classiques, ce qui freine leur adoption dans des applications concrètes.

Nous nous concentrons dans ce travail sur l'un des schémas les plus prometteurs : TFHE. Nous proposons de nouvelles techniques destinées à accélérer les calculs homomorphes pour différents cas d'usage. En exploitant un encodage innovant des messages, nous commençons par convevoir des algorithmes plus efficaces pour l'évaluation homomorphe de fonctions booléennes.

Dans un second temps, nous abordons le problème du transchiffrement, une approche visant à réduire la consommation de bande passante lors de la transmission de données chiffrées de manière homomorphe. Cela nécessite l'évaluation d'un algorithme de chiffrement symétrique dans le domaine homomorphe. Pour cela, et toujours en nous appuyant sur notre technique d'encodage, nous développons une implémentation homomorphe du chiffrement standard AES, plus rapide que celles de l'état de l'art, et contribuons à la conception d'un chiffrement par flot spécifiquement optimisé pour le transchiffrement.

Nous poursuivons avec une contribution qui étend les capacités de TFHE, en lui permettant de fonctionner sur des espaces de messages plus larges. Cette amélioration est possible grâce à un nouvel algorithme d'évaluation de table de correspondances dans ces espaces étendus.

Enfin, nous proposons une méthode conceptuellement simple et pratique pour générer des jeux de paramètres assurant sécurité, exactitude des calculs et efficacité, facilitant ainsi l'usage de TFHE dans les applications concrètes.

Mots-Clés ⋆ Cryptographie, Chiffrement complètement homomorphe, Calcul sécurisé

Abstract

In this thesis, we study fully homomorphic encryption, a cryptographic technique that allows computations to be performed directly on encrypted data, without requiring prior decryption. This field has experienced remarkable growth over the past fifteen years, with the emergence of increasingly efficient encryption schemes. Nevertheless, homomorphic computations remain significantly more costly than their classical counterparts, which still hinders their adoption in practical applications.

In this work, we focus on one of the most promising schemes: TFHE. We propose new techniques aimed at accelerating homomorphic computations for various use cases. By leveraging an innovative message encoding strategy, we begin by designing more efficient algorithms for the homomorphic evaluation of Boolean functions.

Next, we address the problem of transciphering, an approach that seeks to reduce bandwidth consumption during the transmission of homomorphically encrypted data. This requires the evaluation of a symmetric encryption algorithm within the homomorphic domain. Still relying on our encoding technique, we develop a homomorphic implementation of the standard AES encryption scheme that outperforms state-of-the-art implementations, and present our contribution to the design of a stream cipher specifically optimized for transciphering.

We continue with a contribution that extends the capabilities of TFHE by enabling it to operate over larger message spaces. This improvement is made possible by a new algorithm for evaluating look-up tables in these extended spaces.

Finally, we propose a conceptually simple and practical method for generating parameter sets that ensure security, correctness, and efficiency, thereby facilitating the use of TFHE in real-world applications.

Keywords ⋆ Cryptography, Fully Homomorphic Encryption, Secure computation

Acknowledgments

Je remercie particulièrement Caroline Fontaine et Pierre-Alain Fouque pour avoir accepté de rapporter ce manuscrit. Je remercie également Ilaria Chillotti, Adeline Roux-Langlois et Renaud Sirdey pour leur participation au jury de cette thèse.

J'ai beaucoup entendu que l'ingrédient le plus important pour une thèse réussie est la qualité de l'encadrement. Après ces trois dernières années, je peux confirmer que c'est absolument vrai. J'adresse donc un grand merci à mes trois encadrants : Sonia Belaïd, Matthieu Rivain et David Pointcheval. Merci Sonia, pour m'avoir fait confiance et accompagné depuis le premier jour de stage, quand je ne connaissais pas grand chose à la crypto, ni à la recherche. Merci pour tes encouragements constants et ton attention au quotidien. Merci Matthieu, pour ton inébranlable optimisme, ton inépuisable capacité à trouver des idées et tout simplement pour ta gentillesse. Enfin, merci David pour m'avoir accueilli à l'ENS. J'ai pu constater que ta réputation d'efficacité n'est pas usurpée, et je te remercie pour ta disponibilité pour mes questions malgré tout ce que tu gères. Merci à tous les trois. C'était très enrichissant et très agréable de travailler avec vous.

J'ai eu la chance de trouver un excellent environnement à CryptoExperts. Je remercie donc tous mes collègues anciens et actuels : Ryad Benadjila, Ghozlen Boukacem, Gaëtan Cassiers, Thibauld Feneuil, Louis Goubin, Viet-Sang Nguyen, Victor Normand, Pascal Paillier, Mélissa Rossi, Abdul Rahman Taleb, Muaad Tamtam, Ronan Thoraval et Auguste Warmé-Janville.

Faire une thèse Cifre signifie avoir un deuxième bureau. J'ai beaucoup apprécié faire partie de "l'open space" de l'ENS et de l'équipe qui y vit : donc merci à Léonard Assouline, Henry Bambury, Hugo Beguinet, Céline Chevalier, Cédric Geissert, Wissam Ghantous, Lenaïck Gouriou, Paul Hermouet, Laurent Holin, Antoine Houssais, Guirec Lebrun, Jules Maire, Brice Minaud, Ky Nguyen, Phong Nguyen, Paola de Perthuis, Eric Sageoli, Robert Schädlich, Erkan Tairi, Florian Tousnakhoff et Quoc-Huy Vu. Je remercie aussi Lise-Marie Bivard pour sa précieuse aide dans les différents méandres administratifs qu'il a fallu traverser.

Je tiens aussi à remercier mes co-auteurs : l'équipe du CEA pour nos réflexions sur l'AES: Aymen Boudguiga, Daphné Trama et Renaud Sirdey. Egalement, merci aux experts du symétrique de l'INRIA et de Versailles pour notre collaboration sur Transistor : Jules Baudrin, Christina Boura, Anne Canteaut, Gaëtan Leurent, Léo Perrin et Yann Rotella.

Merci à Christina Boura et Louis Goubin pour m'avoir donné l'opportunité d'enseigner, et aux étudiantes et aux étudiants de leur sympathie. Merci à Samuel Tap pour avoir pris le temps de m'expliquer les arcanes de TFHE et du paramétrage. Merci à Pierrick Méaux pour l'invitation à Luxembourg et pour les discussions sur le transchiffrement. Merci à Guirec Lebrun pour m'avoir fait découvrir MLS (et pour sa solidarité dans les différentes galères que cela a impliqué). Merci à Ryad Benadjila pour avoir été un très bon coach de C. Merci à l'équipe de développement de `tfhe-rs` pour ce fantastique outil dont j'ai usé et abusé.

L'obsessivité qui vient avec l'activité de recherche peut très vite dévorer les autres aspects de la vie, mais j'ai toujours pu compter sur mes proches pour me sortir la tête du guidon. Je remercie donc chaleureusement mes amis, des anciens du Nord-Isère aux Ensimagiens en passant par mes colocataires successifs, et ma famille, en particulier mes parents et ma petite soeur. Je termine avec une pensée pour Dalia, et la remercie pour tout ce qu'elle m'apporte en partageant ma vie.

Contents

How to read this thesis ?

The thesis begins by a brief section written in French, that introduces the topic and provides a summary of the contributions of this thesis.

Chapter 1 is an introduction presenting Fully Homomorphic Encryption (FHE), including some historical background and a review of the state of the art. Then, Chapter 2 introduces the TFHE cryptosystem in detail, presenting all its internal components and their functioning. In particular, its bootstrapping operation, which lies at the heart of its homomorphic capabilities, is described in depth.

In Chapter 3, we address one of TFHE's fundamental issues: the negacyclicity problem. This is one of the main hurdles when using TFHE in practice, as it limits the performance of homomorphic operations and greatly complicates the design of homomorphic programs. We provide a formal presentation of the problem as well as an overview of the existing solutions found in the literature. We then introduce a new approach using a plaintext space of odd order, which resolves the negacyclicity issue while also enabling new functionalities for TFHE. This construction forms the foundation on which the rest of this thesis builds.

Chapter 4 presents a method to accelerate the evaluation of arbitrary Boolean functions in TFHE. The core technique of TFHE's original paper performs one bootstrapping per logic gate in the function's circuit. The problem is that this approach does not scale well when working with more complex functions or more inputs. To overcome this, we developed a new type of encoding, called p-encodings, which embed bits into a larger space. This allows multiple bits to be compressed into the same ciphertext by summing them, enabling evaluation of the entire function through a single bootstrapping. We develop algorithms to find suitable p-encodings for a given function. If the circuit is still too large, we also present an algorithm to decompose it into sub-blocks that can be processed using our method. To test this construction, we apply it to several cryptographic primitives and demonstrate significant performance improvements compared to the state of the art.

This work resulted in the publication:

Nicolas Bon, David Pointcheval, and Matthieu Rivain. "Optimized Homomorphic Evaluation of Boolean Functions". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.3 (2024), pp. 302–341. DOI: 10.46586/tches.v2024.i3.302-341

Chapter 5 aims to improve the homomorphic implementation of the AES standard from the previous chapter. To do so, we exploit both Boolean and arithmetic representations and develop a general framework for efficiently switching between the two. This requires generalizing the encoding method from the previous chapter beyond the Boolean case to the arithmetic case, and adapting advanced homomorphic operators from the literature to this encoding strategy. This leads to the fastest AES implementation in the literature.

This work resulted in the publication:

Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. "Further Improvements in AES Execution over TFHE". in: *IACR Commun. Cryptol.* 2.1 (2025), p. 39. DOI: 10.62056/AHMP-4TW9. URL: https://doi.org/10.62056/ahmp-4tw9

The main target use case of Chapter 5 is transciphering, a cryptographic technique that solves the problem of ciphertext expansion. When data is encrypted homomorphically, it occupies much more memory space and thus consumes more bandwidth when sent to a server. Transciphering addresses this issue: instead, the client encrypts the data using a conventional symmetric cipher, and the server homomorphically decrypts the data to bring it into the homomorphic domain. Experimental results show that using a standard cipher like AES is not very efficient; instead, one would prefer a cipher specifically designed to be efficiently evaluable under homomorphic encryption. This is exactly what we construct in Chapter 6: we present our contribution to the design of `Transistor`, a stream cipher that is highly efficient under TFHE. We provide its specification and explain the rationale behind its design choices, including the use of an odd modulus. Most of the chapter is dedicated to analyzing the strong performance of `Transistor` in the homomorphic domain.

This work was published in:

Jules Baudrin, Sonia Belaïd, Nicolas Bon, Christina Boura, Anne Canteaut, Gaëtan Leurent, Pascal Paillier, Léo Perrin, Matthieu Rivain, Yann Rotella, and Samuel Tap. "Transistor: a TFHE-friendly Stream Cipher". In: *Advances in Cryptology - CRYPTO* (2025). URL: https://eprint.iacr.org/2025/282

The published version above includes significantly more content, including an in-depth security analysis of the scheme.

One of TFHE's main limitations is that programmable bootstrapping becomes very slow as the size of the message increases. As a result, evaluating Look-Up Tables (LUTs) larger than 8 bits is impractical. In Chapter 7, we tackle this issue by extending TFHE's bootstrapping capabilities beyond 8 bits through a method that accelerates homomorphic evaluation of large LUTs. Once again leveraging our encoding technique, we design a Look-Up Table (LUT) decomposition algorithm that enables bootstrapping to be applied on smaller messages.

Finally, Chapter 8 goes beyond the scope of our encoding method in odd spaces and introduces ORPHEUS, a tool designed to help homomorphic program designers choose parameter sets that ensure the three central properties: security, correctness of computation, and efficiency. The strength of ORPHEUS lies in its flexibility, allowing easy extension to new homomorphic operators. Furthermore, its optimization algorithm estimates runtime using a cost model that depends on the machine executing the program, allowing parameter sets to be tailored to various usage contexts.

Notations

Throughout this manuscript, we adopt the following conventions:

- **Scalars** are denoted by lowercase letters (e.g., a), and **polynomials** by uppercase letters (e.g., A).
- **Vectors** are written in bold (e.g., $\mathbf{a}$ or $\mathbf{A}$), and **matrices** in bold uppercase roman type (e.g., $\mathcal{A}$).

We use the following common symbols for standard mathematical sets:

- $\mathbb{N}$ for the set of natural numbers,
- $\mathbb{Z}$ for the set of integers,
- $\mathbb{R}$ for the set of real numbers,
- $\mathbb{B} = \{0, 1\}$ for the set of bits,
- $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ for the torus, i.e., real numbers modulo 1.

The ring of integers modulo q, denoted classically by $\mathbb{Z}/q\mathbb{Z}$, is abbreviated as $\mathbb{Z}_q$. Similarly, finite fields are denoted by $\mathbb{F}$, with $\mathbb{F}_p$ representing the finite field of prime order p.

Polynomial rings are written as, for example, $\mathbb{Z}_q[X]$. In particular, we frequently use cyclotomic polynomial rings of the form $\mathbb{Z}_q[X]/(X^N + 1)$, where N is a power of two. These are abbreviated as $\mathbb{Z}_{q,N}[X]$.

We adopt the following mathematical operator notations:

- $\lfloor x \rceil$ denotes the rounding of a real value x to the nearest integer,
- $[x]_q$ denotes the reduction of x modulo q,
- $\langle \mathbf{x}, \mathbf{y} \rangle$ denotes the inner product of vectors $\mathbf{x}$ and $\mathbf{y}$.

Finally, for randomness and distributions, if $\mathcal{D}$ is a distribution, the notation $x \xleftarrow{\$} \mathcal{D}$ means that x is sampled according to $\mathcal{D}$. The same notation is used for uniform sampling from a set; for example, $x \xleftarrow{\$} \mathbb{Z}_q$ denotes that x is sampled uniformly at random from $\mathbb{Z}_q$.

Acronyms

AES	Advanced Encryption Standard. i, iii, ix, x, xix, 31, 39, 48, 56–62, 64, 65, 68–71, 73–77, 79, 82, 83, 94, 95, 97, 133
BSK	Bootstrapping Key. xvii, 4, 19–21, 93
FFT	Fast Fourier Transform. 5, 118
FHE	Fully Homomorphic Encryption. ix, xvi, xvii, 1, 3–7, 20, 49, 56, 61–65, 71, 73, 76, 79, 94, 101, 115–117, 121, 124, 133
GGSW	Generalized GSW [GSW13], see Definition 2.6.2 and 2.6.3. 11, 15, 16, 19, 20, 26, 94, 111
GLWE	General Learning With Errors, see Definition 2.1.2. 7, 8, 10, 11, 14–16, 19–21, 58, 66, 73, 118, 119, 121
KSK	Keyswitching Key. 12–14, 21, 93
LFSR	Linear-Feedback Shift Register. 81–87, 89–91, 95
LSB	Least Significant Bit. 32
LUT	Look-Up Table. x, xix, 23, 26, 32, 33, 60, 61, 65–68, 75, 95, 97, 98, 111, 113, 115, 133
LWE	Learning With Errors, see Definition 2.1.1. 2, 7–14, 17, 19–21, 33, 58, 68, 73, 106, 111, 118, 119, 121
MSB	Most Significant Bit. 24, 25, 33, 49
MVB	Multi-Value Bootstrapping. See Section 5.3.2. 65–67, 69, 71
NTT	Number Theoretic Transform. 5
PBS	Programmable Bootstrapping, see Section 2.7.2. 7, 18, 20–24, 26, 32, 34, 36–39, 46, 51–59, 65, 69–73, 80, 81, 84, 85, 88–94, 97–101, 103, 106, 108, 110, 111, 113–125, 135, 136, 138

RLWE Ring Learning With Errors. 8

S-box Substitution Box. A substitution table in a symmetric-key algorithm. 31, 46, 48, 55–61, 64–67, 73, 74, 81, 82, 86, 95, 97

TBM Tree-Based Method (or Tree-Based Bootstrapping). See Section 5.3.2. 62, 65, 67, 69, 71, 74, 76

TFHE Fully Homomorphic Encryption over the Torus. i, iii, ix, x, xvii–xix, 4, 5, 7–24, 27, 29, 31–34, 36, 39, 49, 53, 57, 58, 60–62, 64, 65, 68, 69, 71, 73, 74, 76, 77, 79–81, 84, 85, 87–91, 93–95, 97–100, 106, 115–121, 123–126, 133

WoP-PBS Without-Padding Programmable Bootstrapping. See Section 3.3 and 7.4. 22, 26, 97, 111, 113, 114

Introduction en Français

Mise en Contexte

Cryptographie. La *cryptographie* est un domaine technique à l'interface entre l'informatique et les mathématiques appliquées. Elle étudie les méthodes permettant de protéger l'information. Les techniques cryptographiques les plus classiques sont:

- Le *chiffrement*, qui transforme un message, un fichier ou plus généralement n'importe quel type de donnée en la "brouillant". Le seul moyen pour déchiffrer est de posséder la *clé* du chiffrement. La donnée est donc illisible par les personnes non autorisées, . Cette technique est notamment utilisée pour protéger la confidentialité des messages sur les applications de messageries instantanées telles que WhatsApp ou Signal.

- L'*authentification*, qui permet de vérifier l'identité de l'émetteur d'un message. Par exemple, elle assure qu'un utilisateur se connecte bien au serveur de sa banque et non à un serveur frauduleux contrôlé par un pirate.

- Le *contrôle d'intégrité*, qui permet de s'assurer qu'un message n'a pas été modifié ou corrompu pendant sa transmission. C'est notamment utile pour s'assurer qu'un logiciel téléchargé n'a pas été altéré afin d'y introduire une faille de sécurité.

A l'origine exclusivement réservée au domaine militaire, la cryptographie a été transformée en un enjeu de société majeur au XXIe siècle. Une grande partie des échanges se fait désormais en ligne, qu'il s'agisse de transactions bancaires, d'échanges commerciaux ou de simples messages à ses proches. De fait, rendre les systèmes de communications résistants face aux attaques d'acteurs malveillants est devenu un enjeu stratégique central pour garantir la sécurité et les libertés individuelles des citoyens. Des exemples de tels attaquants sont les cybercriminels qui pratiquent l'usurpation d'identité pour monter des escroqueries, ou bien rançonnent des entreprises ou des services publics en bloquant leur infrastructure ou en retenant leurs données. Il s'agit aussi de gouvernements autoritaires pratiquant la surveillance de masse sur leur population, afin de neutraliser des opposants politiques ou opprimer des groupes minoritaires.

Les travaux fondateurs de Claude Shannon sur la théorie de l'information montrent qu'*il ne peut exister de chiffrement parfait.* Autrement dit, un système cryptographique *théoriquement incassable* serait inutilisable dans le monde réel. Ainsi, la pratique de la cryptographie consiste à garantir un niveau de sécurité suffisant à un système, sans altérer sa fonctionnalité ni ses performances.

Pour ce faire, les cryptographes cherchent à déterminer la puissance de calcul nécessaire à un attaquant pour casser un système de sécurité, par exemple en déchiffrant un message secret dont il n'a pas la clé. L'exemple le plus basique d'attaque est l'attaque par force brute (*brute force*), qui consiste à essayer toutes les clés possibles jusqu'à trouver la bonne. Il convient donc de choisir des clés suffisamment grandes pour que cette stratégie soit trop lente, ou trop coûteuse à mettre en oeuvre. Evidemment, les attaques contre les systèmes cryptographiques se sophistiquent d'années en années, donc les techniques cryptographiques doivent évoluer pour s'y adapter et toujours avoir un temps d'avance.

Calculer sur des données chiffrées. La cryptographie a connu un essor fulgurant au cours des dernières décennies. Notamment, le trafic Internet, qui était en clair jusqu'alors, a été sécurisé par l'introduction du protocole HTTPS, qui permet de chiffrer et d'authentifier les échanges entre le client et le serveur.

Cependant, il reste un cas d'usage où la cryptographie demeure impuissante, et dans lequel les données sont encore mal protégées: le *calcul délégué*. Cette (vague) dénomination englobe tout les cas d'usages dans lesquels un utilisateur envoie une donnée à un serveur, non pas pour qu'il en assure le transit à travers Internet, mais pour qu'il la traite et lui renvoie un résultat. On peut penser par exemple aux applications telle que Google Maps, dans lesquelles l'utilisateur envoie sa position actuelle et sa destination au serveur, qui calcule alors un itinéraire qu'il renvoie sur le téléphone de l'utilisateur. On peut également penser aux applications d'Intelligences Artificielles génératives dans lesquelles l'entrée de l'utilisateur est traitée par un algorithme pour générer du texte, de la musique ou des images. Enfin, cela concerne tous les cas où des entreprises louent des serveurs externes pour effectuer des calculs lourds ou héberger des services.

Le problème est qu'effectuer des calculs sur des données chiffrées est un immense défi technologique, qu'on a longtemps pensé impossible. Par conséquent, le serveur doit nécessairement déchiffrer les données pour pouvoir les traiter, ce qui rend ces dernières vulnérable à la moindre compromission du serveur par une attaque informatique.

Cryptographie Homomorphe. La cryptographie homomorphe (*Fully Homomorphic Encryption* en anglais, souvent abrégé en FHE) est la branche de la cryptographie qui s'attaque à ce problème. Son but est de développer des algorithmes de chiffrement permettant à un serveur d'effectuer des calculs directement sur les données chiffrées, sans nécessiter de déchiffrement préalable. Il devient alors inutile pour un attaquant d'essayer de s'y introduire, car l'intégralité des données de valeur qu'il contient sont chiffrées et donc inutilisables. De plus, le fournisseur du service n'a lui-même pas accès aux données, ce qui assure une confidentialité totale vis-à-vis de l'utilisateur.

Cette idée apparaît dans un article de recherche pour la première fois en 1978, mais il faut attendre 2009 pour que la première construction viable d'un algorithme de chiffrement homomorphe apparaisse dans un article de recherche intitulé *Fully Homomorphic Encryption from Ideal Lattices* par Craig Gentry [Gen09]. Ce travail a surtout une valeur théorique, car l'algorithme de Gentry demande tellement de ressources pour être calculé qu'il est impossible de l'utiliser dans le monde réel.

Cet article a initié un veritable essor du domaine dans la communauté scientifique, et les progrès ont été extrêmement rapides. Les algorithmes actuels commencent à être utilisable en pratique, et certains projets concrets d'aapplications homomorphes commencent à voir le jour.

Le challenge actuel est donc d'améliorer les performances du chiffrement homomorphe. Pour cela, les cryptographes doivent s'attaquer à deux problématiques principales:

- *La quantité de calcul nécessaire:* Effectuer un calcul dans le domaine homomorphe (c'est-à-dire directement sur les chiffrés) nécessite beaucoup plus d'opérations que lorsqu'on l'effectue en clair. Par conséquent, une application fonctionnant homomorphiquement est beaucoup plus lente et plus coûteuse en énergie que si on l'exécute de façon classique (de 3 à 5 ordres de grandeur en fonction de la nature des calculs).

- *Le bruit:* La sécurité des schémas de chiffrement homomorphes reposent sur la théorie des réseaux euclidiens. Concrètement, cela signifie que lorsqu'on chiffre les données on leur ajoute une petite perturbation aléatoire qu'on appelle le *bruit*. Comme ce bruit est très faible, il ne pose pas de problème lors du déchiffrement car il est facile de se débarasser de cette imprécision en arrondissant simplement les valeurs. Par contre, lorsqu'on effectue des calculs homomorphes entre plusieurs valeurs chiffrées, leurs bruits s'additionnent ce qui fait croître l'imprécision. Par conséquent, le nombre d'opérations homomorphes qu'il est

possible d'effectuer est limité, car un bruit trop élevé deviendrait prépondérant par rapport à l'information contenue dans les messages, rendant le déchiffrement impossible. On a donc un "quota" de quantité de calculs disponible au terme duquel il faut nécessairement s'arrêter.

Bootstrapping. Dans son article fondateur de 2009, Craig Gentry introduit une notion fondatrice appelée *bootstrapping*[1], qui résout complètement le second problème. Il s'agit d'une opération qui permet au serveur de réduire le bruit d'un chiffré de manière homomorphe, sans violer la confidentialité des données ! Donc si un schéma de chiffrement possède une opération de bootstrapping (on dit qu'il est *bootstrappable*), cela signifie qu'il n'a pas de limitations sur la quantité de calcul qu'il est possible de faire sur les données chiffrées. En effet, il suffit d'appliquer un bootstrapping à chaque fois que la quantité de bruit dans les chiffrés devient trop grande, et de continuer les calculs avec le chiffré bootstrappé !

Pour comprendre comment cela fonctionne, rappelez vous que lorsque l'utilisateur déchiffre le résultat, il élimine le bruit avec une opération d'arrondi. Donc, s'il est possible de calculer homomorphiquement la fonction de déchiffrement du schéma, alors il est possible de retirer le bruit sans que le serveur aie besoin de déchiffrer !

Concrètement, disons que le serveur possède un message chiffré $\mathbf{c}_1$ qui correspond au message clair (bruité) $m + e_1$, chiffré avec la clé secrète $\mathbf{s}_1$. Ici m et e_1 représentent respectivement le message clair et le bruit. Supposons que $\mathbf{c}_1$ soit le résultat d'opérations homomorphes, donc le bruit e_1 est élevé. Si nous voulons continuer à calculer, il faut réduire le bruit par un processus de bootstrapping. Pour cela, le client doit fournir au serveur une clé de bootstrapping BSK. Pour la créer, le client peut considérer la clé secrète $\mathbf{s}_1$ comme un message et la chiffrer sous une autre clé secrète $\mathbf{s}_2$ pour produire $\mathsf{BSK} = \mathtt{Enc}_{\mathbf{s}_2}(\mathbf{s}_1)$, où $\mathtt{Enc}$ désigne la fonction de chiffrement.

La propriété fondamentale découverte par Gentry est que si le serveur calcule homomorphiquement le déchiffrement de $\mathbf{c}_1$ en utilisant la clé BSK, il obtient un chiffré $\mathbf{c}_2$ du même message sous la clé secrète $\mathbf{s}_2$. Mais comme le déchiffrement élimine le bruit, e_1 disparaît et le message chiffré est à nouveau « frais » ! En réalité, tout le bruit n'est pas supprimé (ce qui ne serait pas souhaitable, car toute la sécurité du chiffrement repose sur la présence de bruit dans les chiffrés). Comme il y a du bruit dans BSK, le résultat $\mathbf{c}_2$ contient un nouveau bruit e_2. Mais si l'algorithme est bien conçu, il est possible d'avoir $e_2 \ll e_1$, ce qui permet de gagner de la marge pour d'autres calculs.

Le bootstrapping n'est cependant pas une solution miracle. En effet, c'est une opération extrêmement coûteuse pour la totalité des schémas de chiffrement homomorphe. Donc en résolvant le second problème (celui du bruit qui limite la quantité de calcul), nous avons en fait aggravé le premier (la lenteur du FHE) ! L'un des axes de recherches principaux dans la communauté scientifique est donc de créer des schémas de chiffrement homomorphe avec un bootstrapping le plus efficace possible.

Le schéma TFHE. L'un des schémas homomorphes les plus prometteurs se nomme TFHE [Chi+20] (pour *Fully Homomorphic Encryption over the Torus*). Comme tous les autres schémas, il permet d'effectuer des opérations linéaires, c'est-à-dire l'addition de deux chiffrés et la multiplication d'un chiffré par une constante. Ces opérations linéaires sont quasiment gratuites en terme de quantité de calcul, donc sont extrèmement rapides. Par contre, elles augmentent le bruit dans les chiffrés, et le serveur ne peut donc pas en effectuer un nombre illimité. Fort heureusement, TFHE possède une opération de bootstrapping relativement efficace par rapport aux standards du FHE, et donc les chiffrés peuvent être régulièrement rafraîchi sans tuer les performances. La particularité du bootstrapping de TFHE est qu'il est *programmable*. Cela signifie qu'en plus de réduire le niveau de bruit, le bootstrapping permet d'évaluer n'importe quelle

[1] on dirait *réamorçage* en français

fonction sur le chiffré, sans aucun surcoût ni en terme de calculs ni en terme de bruit ! Cette fonctionnalité représente une avancée majeure dans le domaine, car cela permet de rentabiliser le temps passé dans les opérations de bootstrapping.

Cependant, TFHE a deux défauts de taille par rapport aux autres schémas de l'état de l'art. D'abord, il ne permet de manipuler seulement des données de faible précision, de l'ordre de quelques bits. Par conséquent, si on veut travailler avec des entiers, cela nécessite de découper les valeurs en petites "tranches" de quelques bits et de manipuler ces sous-blocs. Cela rend la conception de programmes homomorphes bien plus complexe que dans le domaine des clairs. C'est la raison pour laquelle le développement de systèmes de compilation automatique de programmes homomorphes est devenu un domaine de recherche actif ces dernières années.

Le second problème est que TFHE n'est pas intrinsèquement parallélisable, contrairement à d'autres chiffrements prometteurs de l'état de l'art tels que CKKS [Che+17] qui permettent d'encoder plusieurs valeurs dans un même chiffré et d'effectuer des calculs en parallèle sur chacun d'eux. Si ces schémas montrent des meilleurs timings amortis, leur latence est beaucoup plus grande.

Malgré ces faiblesses, TFHE reste très étudié dans la littérature scientifique et constitue l'un des principaux espoirs pour l'adoption massive du chiffrement homomorphe. Pendant cette thèse, nous nous sommes concentrés sur ce schéma et avons développé des algorithmes permettant d'accélérer les calculs homomorphes dans certains cas d'usages. Nous présentons un résumé de nos travaux dans la prochaine section.

Résumé de la thèse et des contributions scientifiques

La thèse commence par une introduction (chapitre 1) présentant le chiffrement totalement homomorphe, incluant quelques considérations historiques et présentant un état de l'art de l'écosystème. Puis, le chapitre 2 introduit en détail le cryptosystème TFHE, en présentant tous ses composants internes ainsi que leur fonctionnement. En particulier, son opération de bootstrapping, qui constitue le coeur de ses capacités homomorphes, est présenté en détail.

Dans le chapitre 3, nous présentons l'une des problématiques fondamentales de TFHE: *le problème de négacyclicité*. Il s'agit de l'un des principaux écueils lors de l'utilisation de TFHE en pratique, car il limite les performances des opérations homomorphes de TFHE et complexifie énormément la conception de programmes homomorphes. Nous donnons une présentation formelle du problème ainsi qu'un aperçu de l'état de l'art des solutions existantes dans la littérature pour le résoudre. Puis, nous introduisons une nouvelle méthode consistant à utiliser un espace de plaintext d'ordre *impair*, ce qui résout ce problème de négacyclicité tout en activant de nouvelles fonctionnalités pour TFHE. Cette construction constitue la base sur laquelle reposent les contributions du reste de cette thèse.

Le chapitre 4 présente une méthode pour accélérer l'évaluation de fonctions booléennes arbitraires en TFHE. La technique de base de l'article fondateur de TFHE est d'effectuer un bootstrapping par porte logique dans le circuit de la fonction. Le problème est que cette stratégie passe très mal à l'échelle quand on veut travailler avec des fonctions plus complexes ou avec plus d'entrées. Pour résoudre ce problème, nous avons développé un nouveau type d'encodage, appelé *p-encodages*, qui plongent les bits dans un espace plus grand. Grâce à cela, il devient possible de compresser plusieurs bits dans le même chiffré en les sommant, puis d'évaluer l'intégralité de la fonction en ne calculant qu'un seul bootstrapping. Nous développons des algorithmes permettant de trouver les p-encodages adaptés à une fonction donnée. Dans le cas où le circuit serait tout de même trop grand, nous présentons également un algorithme pour découper le circuit en sous-blocs évaluables avec notre méthode. Pour tester notre construction, nous l'appliquons à quelques primitives cryptographiques pour les implémenter en homomorphe, et démontrons un gain de performance significatif par rapport à l'état de l'art. Ce travail a donné lieu à la publication:

Nicolas Bon, David Pointcheval, and Matthieu Rivain. "Optimized Homomorphic Evaluation of Boolean Functions". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.3 (2024), pp. 302–341. DOI: 10.46586/tches.v2024.i3.302-341

Le chapitre 5 vie à améliorer l'implémentation homomorphe du standard AES du chapitre précédent. Pour ce faire, nous exploitons à la fois les représentations booléennes et arithmétiques et développons un cadre générique pour passer efficacement de l'une à l'autre. Cela nécessite de généraliser la méthode d'encodage du chapitre précédent au-delà du cas booléen vers le cas arithmétique, ainsi que d'adapter des opérateurs homomorphes avancés de l'état de l'art à cette logique d'encodage. Nous produisons ainsi l'implémentation d'AES la plus rapide de la littérature. Ce travail a donné lieu à la publication:

Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. "Further Improvements in AES Execution over TFHE". in: *IACR Commun. Cryptol.* 2.1 (2025), p. 39. DOI: 10.62056/AHMP-4TW9. URL: https://doi.org/10.62056/ahmp-4tw9

Le principal cas d'usage visé par le chapitre 5 est le transchiffrement, une technique cryptographique permettant de résoudre le problème d'expansion de chiffré. Concrètement, lorsque des données sont chiffrées homomorphiquement, elles prennent beaucoup plus d'espace en mémoire, et donc consomment plus de bande passante quand elles sont envoyées au serveur. Le transchiffrement résout ce problème: le client va plutôt envoyer les données chiffrées avec un algorithme de chiffrement symétrique classique, et le serveur va déchiffrer homomorphiquement ces données pour les récupérer dans le domaine homomorphe. Les résultats expérimentaux montrent qu'utiliser un chiffrement standard tel que l'AES n'est pas très efficace, une meilleure option serait d'utiliser un algorithme de chiffrement spécialement conçu pour s'évaluer rapidement dans le domaine homomorphe. C'est ce que nous construisons dans le chapitre 6: nous y présentons notre constribution à la conception de `Transistor`, un chiffrement à flot s'évaluant très efficacement avec TFHE. Nous en donnons la spécification et expliquons le raisonnement motivant ses choix de conception, notamment l'utilisation du modulo impair. La majeure partie de ce chapitre est consacrée à l'analyse des bonnes performances de `Transistor` dans le domaine homomorphe. Ce travail a donné lieu à la publication:

Jules Baudrin, Sonia Belaïd, Nicolas Bon, Christina Boura, Anne Canteaut, Gaëtan Leurent, Pascal Paillier, Léo Perrin, Matthieu Rivain, Yann Rotella, and Samuel Tap. "Transistor: a TFHE-friendly Stream Cipher". In: *Advances in Cryptology - CRYPTO* (2025). URL: https://eprint.iacr.org/2025/282

La version publiée ci-dessus a beaucoup plus de contenu, notamment une analyse approfondie de la sécurité du schéma.

L'une des limitations principales de TFHE est que l'opération de bootstrapping programmable devient très lente à mesure qu'on traite des messages de plus en plus grands. Ainsi, évaluer des tables de correspondances (*Look-Up Tables*) de taille supérieure à 8 bits est impossible en pratique. Dans le chapitre 7, nous nous attaquons à ce problème et étendons les capacités du bootstrapping de TFHE au-delà de 8 bits grâce à une méthode accélérant l'évaluation homomorphe de ces grandes *Look-Up Tables*. En nous appuyant à nouveau sur notre technique d'encodage, nous avons conçu un algorithme de décomposition des LUT permettant d'utiliser le bootstrapping sur des plus petits messages.

Enfin, le chapitre 8 va au-delà du cadre de notre méthode d'encodage dans des espaces impairs et introduit ORPHEUS, un outil destiné à aider les concepteurs de programmes homomorphes à dimensionner des jeux de paramètres garantissant les trois propriétés centrales : la sécurité, la correction des calculs et l'efficacité. La force de ORPHEUS réside dans sa flexibilité permettant de l'étendre facilement à des nouveaux opérateurs homomorphes. De plus, son algorithme

d'optimisation estime le temps de calcul grâce à un modèle de coût dépendant de la machine sur laquelle le programme tourne, permettant d'adapter les jeux de paramètres à différents contextes d'utilisation.

Chapter

1 Introduction to Fully Homomorphic Encryption

1.1 Motivation

Cryptography has experienced a tremendous boom over the past decades. In particular, Internet traffic, which used to be transmitted in plaintext, was secured by the introduction of the HTTPS protocol, which encrypts and authenticates communications between client and server.

However, one use case remains where cryptography is still ineffective, and where data is still poorly protected: delegated computation. This (vague) term encompasses all scenarios where a user sends data to a server not merely to transmit it across the Internet, but to have it processed and to receive a result in return. One can think, for example, of applications like Google Maps, where the user sends their current location and destination to a server, which then computes a route and sends it back to the user's phone. Another example is the new generative Artificial Intelligence applications, where the user's input is processed by an algorithm to generate text, music, or images. Finally, this also applies to cases where companies rent external servers to perform heavy computations or to host services.

The problem is that performing computations on encrypted data is a massive technological challenge, long thought to be impossible. As a result, the server must necessarily decrypt the data before processing it, which leaves the data vulnerable to any potential compromise of the server by a cyberattack.

Homomorphic encryption (Fully Homomorphic Encryption, or FHE) is the branch of cryptography that tackles this issue. Its goal is to develop encryption algorithms that allow a server to perform computations directly on encrypted data, without the need to decrypt it first. This way, an attacker has no incentive to breach the server, as all valuable data it contains remains encrypted and therefore useless. Additionally, the service provider itself does not have access to the data, ensuring full confidentiality for the user.

Figure 1.1 demonstrates a procedure of private outsourcing of computation using homomorphic encryption. We consider a use-case where a client owns some sensitive data, while a server provides some AI-based service (represented by a neural network).

1. The client generates a *secret key* and an *evaluation key*. The former allows them to encrypt and decrypt their data, while the latter allows only to perform homomorphic computations on encrypted data, but not to acquire any information on what is actually encrypted.

2. Using the secret key, the client encrypts its sensitive data.

3. The client uploads the encrypted data on the server, as well as the evaluation key. Note that this evaluation key could have been sent beforehand during some user enrollment procedure.

4. The server runs a *homomorphized* version of its neural network. Using the evaluation key, it can run the computation directly on the encrypted data to get the result the client

wished. Because of the encryption layer, the server cannot gain any information on neither the input data nor the result of the computation.

5. The server then sends back the encrypted result to the client.

6. Using the secret key, the client can decrypt the result.

Historical Background. The initial appearance of the notion of homomorphic encryption in the scientific literature can be traced back to 1978, in a paper by Rivest, Adleman and Dertouzos [RAD78]. They theorize the existence of *privacy homomorphism*, which are *encryption functions which permit encrypted data to be operated on without preliminary decryption.*

Over the following decades, research focused primarily on the development of partially homomorphic schemes, which support only a single homomorphic operation, typically either addition or multiplication. Additive homomorphisms were particularly favored in applications like electronic voting. One of the most well-known examples is the Paillier cryptosystem [Pai99].

In parallel, a different class of schemes emerged: leveled (or somewhat) homomorphic encryption schemes. These allow for both addition and multiplication, but only a limited number of operations can be performed sequentially. Notable examples include the DGHV scheme [Dij+10] and the original BGV scheme [BGV14][1].

Interestingly, the bounded depth of computation in these schemes arises from distinct causes. In DGHV, the size of the integers in the ciphertexts grows with each operation, eventually becoming computationally prohibitive. In BGV (like in the others lattice-based schemes), a random error called *noise* is introduced during encryption, and this noise increases until it overwhelms the signal, rendering the ciphertext undecipherable. In both cases, a critical resource (either magnitude or noise) accumulates irreversibly, ultimately preventing further computation.

This bound was suddenly removed in 2009 in a groundbreaking construction by Gentry, wich opened the door to a very fast development of the field in the following years. We present this transformative result in the next section.

1.2 The Breakthrough of Bootstrapping

In 2009, Craig Gentry publishes the breakthrough paper *Fully Homomorphic Encryption using Ideal Lattices* [Gen09]. In this work, he introduces the first ever fully homomorphic encryption scheme. Its main idea is the *bootstrappability* of a scheme.

Bootstrappability is the ability of a scheme to evaluate homomorphically its own decryption circuit. Gentry shows that if a scheme achieves bootstrappability, then it achieves fully homomorphic encryption. Building on this theoretical result, he constructs a lattice-based bootstrappable scheme and demonstrates the first ever example of fully homomorphic encryption scheme. Since this foundational work, lattice-based constructions have remained the only serious candidates for fully homomorphic encryption.

Such homomorphic schemes work by injecting some small random noise in the plaintext data before encryption. Its role is to ensure security, relying on computational hardness assumptions such as Learning With Errors (LWE) (we explain it further in Section 2.1). Because the noise is small, it can be easily removed during decryption by a simple rounding operation.

Gentry's original scheme offered homomorphic additions and multiplications. However, because the plaintext messages are noisy, these operations increase the noise in the ciphertexts. So, if too many operations are performed, the noise gets so large that it becomes preponderant and blows the meaningful information contained in the ciphertexts.

[1] Although these schemes were actually shown to be bootstrappable (see Section 1.2), the bootstrapping process was too inefficient to be practical.

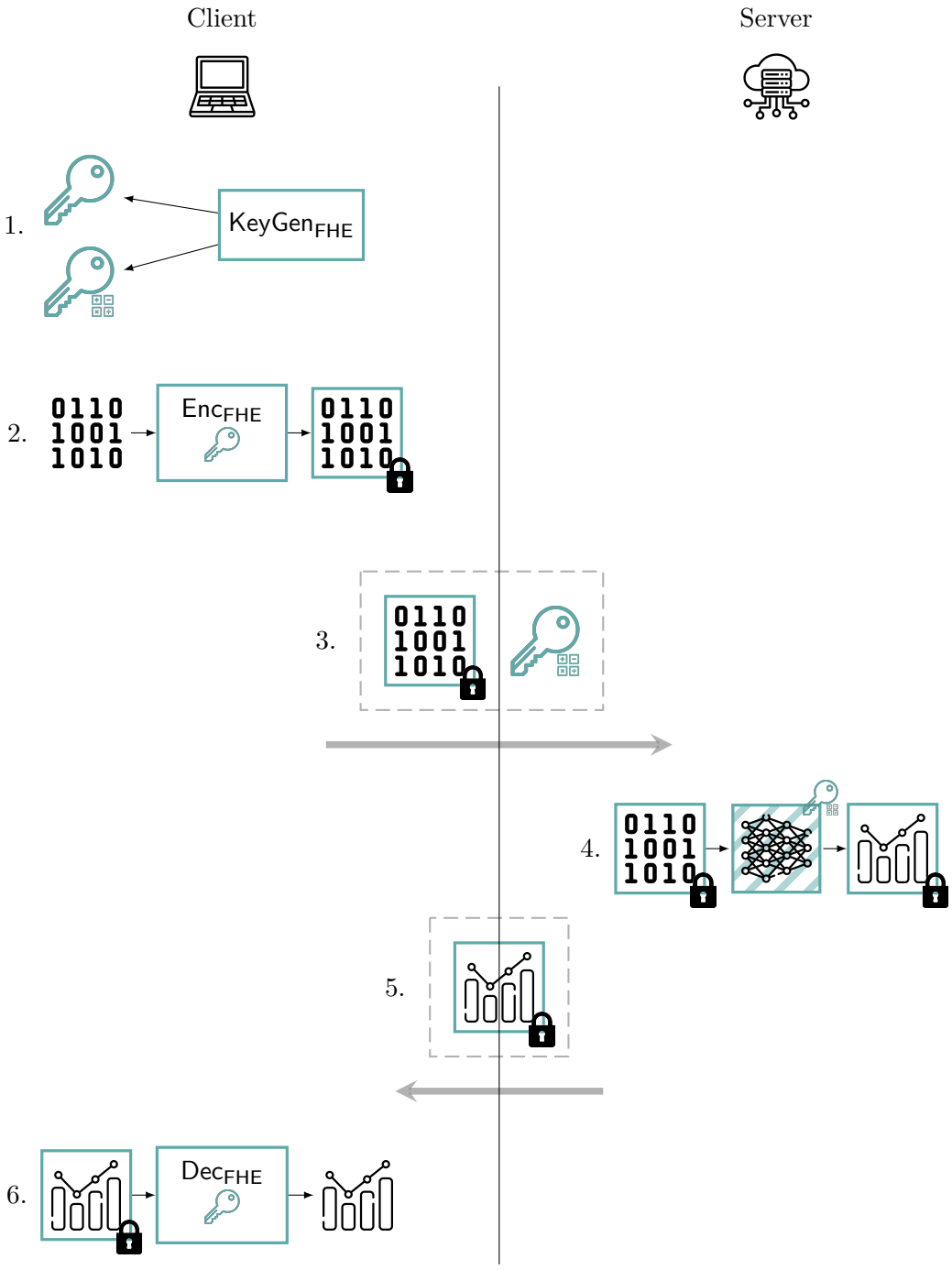


Figure 1.1: An illustration of a protocol of outsourced computation using FHE. Hatching denotes that the homomorphized version is run.

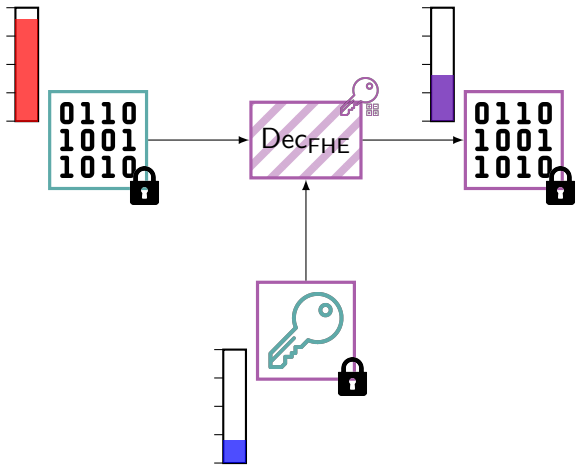


Figure 1.2: High-level overview of the bootstrapping principle. Hatching denotes the fact that the operation is ran homomorphically. The gauges denotes indicative noise levels for the ciphertexts.

This is where bootstrappability comes in. It gives access to an operation (the *bootstrapping*) that decrypts homomorphically the noisy ciphertexts. But how can this help to achieve FHE ?

Imagine the server has a ciphertext $\mathbf{c}_1$ encrypting the (noisy) message $m + e_1$ under the secret key $\mathbf{s}_1$, where m is the plain message and e_1 denotes the noise. Let us say that $\mathbf{c}_1$ is the output of a circuit of homomorphic operations, so the noise e_1 is rather large. If we want to keep computing without losing information, we need to reduce the noise by a bootstrapping.

To do so, the client needs to give to the server a *bootstrapping key* BSK. In order to create it, the client can treat the secret key $\mathbf{s}_1$ as a message and encrypt it under another secret key $\mathbf{s}_2$ to produce $\mathsf{BSK} = \mathtt{Enc}_{\mathbf{s}_2}(\mathbf{s}_1)$. Now, if the server computes *homomorphically* the decryption of $\mathbf{c}_1$ using BSK, it retrieves an encryption $\mathbf{c}_2$ of the same message under the secret key $\mathbf{s}_2$. But because, decryption removes noise, e_1 disappears and the ciphertext is fresh again!

Actually, *all* the noise does not get removed (this would not be desirable, because all the security of the encryption relies on the presence of noise in the ciphertexts). As there is some noise in BSK, the result $\mathbf{c}_2$ carries some noise e_2. But if things are well dimensioned, it is possible to get $e_2 \ll e_1$ and so retrieve some room for further computations. We illustrate the bootstrapping principle in Figure 1.2.

Gentry's original scheme was merely theoretical, because the homomorphic operations, and in particular the bootstrapping, were extremely slow. But since then, significant improvements have been achieved in FHE efficiency. Modern schemes are on the verge of being usable in practice for some use-cases. In the next section, we give a tour of the modern schemes and libraries that makes the FHE landcape as of today.

1.3 Current Landscape of the FHE Schemes and Libraries

After a decade of research, homomorphic schemes have evolved to stabilize around two main paradigms. We present both by their main representatives: TFHE [Chi+20] and CKKS [Che+17]. They have quite different (and complementary) philosophies of computing, and lead to radically different design choices when used:

TFHE: This scheme relies on a very low-latency bootstrapping operation[2]. It manipulates small limbs of data of a few bits, and operates *exact* computations on them.

CKKS: CKKS has quite opposite features of TFHE. It relies on a *approximate* paradigm for the computations, so is quite appropriate for floating-point computations. Moreover, it supports SIMD (Same Instruction Multiple Data) within its ciphertexts, so it allows for heavy parallelization. While its bootstrapping is quite heavy and appeared later in the literature [Che+18], a lot of advancements happened in the last few years making it closer to practical [CCS19; HK20; Kim+22].

We should also mention BFV/BGV [Bra12; BGV14; FV12], that share most of the properties of CKKS. However these constructions do not work with approximate floating-point computation, but rather with exact computations over large integer rings.

Libraries. Several libraries implement these schemes, facilitating their adoption. For TFHE, we can mention tfhe-rs [Zam22c] and TFHEpp [Mat20]. For CKKS, we can mention HEAAN [Inc20] and Lattigo [24]. Finally, OpenFHE [Bad+22] is an attempt at constructing a universal library that supports all the schemes and allows conversions.

These libraries target users familiar with FHE, and offer quite low-level API. One of the issue slowing down the massive adoption of FHE is the difficulty of developing concrete applications from the cryptographic primitives. A layer of compilation is thus required from "plaintext" programming to homomorphic code. Such projects of homomorphic compiler include Concrete [Zam22b] and HEIR [Con23].

Another active line of research is hardware acceleration. Quite promising theoretical results have been achieved [Gee+23; Ber+23b; Cou+23; Kri+24] and the first "FHE-tailored" chips have started to be produced. Such hardware include NTT/FFT-dedicated accelerators, as these operations account for the largest part of the computations.

Some standardization projects are arising [Alb+18; ST25], which would be an important step for the adoption of FHE in practice.

1.4 Security Properties

In cryptography, the security of a scheme is modeled by a *security game*. In an idealized world, a challenger running the scheme is opposed to a polynomial-time attacker. This attacker has access to *oracles* that they can query. For encryption scheme, one of these games is the *indisguishability game*: the attacker picks two messages and sends them to the challenger. The challenger flips a coin and encrypts one of the messages at random. They send this ciphertext (named the *challenge ciphertext*) to the attacker, who has to guess which message has been encrypted. If the attacker can do better that random guessing (so guess right with probability larger than 0.5), we say that they have a *non-negligible advantage*, indicating a vulnerability against the type of attack.

Usually, the "Holy Grail" for encryption schemes is CCA2 security (or *adaptative chosen-ciphertext security*). In the corresponding game, the attacker has access to an encryption and a decryption oracle, that they can query both before and after having received the challenge ciphertext, with the exception that they are forbid to query the decryption oracle directly on the challenge ciphertext.

By design, homomorphic schemes cannot achieve such security property because they are intrinsically *malleable*. So a trivial attack would be to homomorphically add an encryption of zero to the challenge ciphertext and query the decryption oracle on the result. The oracle would accept the decryption and output the message, completely breaking the security of the scheme.

[2] by FHE standards

Actually, things are even worse: if the scheme is bootstrappable, then even CCA1 security is unachievable. This corresponds to a *non-adaptative chosen-ciphertext security*, the difference with CCA2 being that the attacker is no longer allowed to query the decryption oracle after having seen the challenge ciphertext (so it modelizes a weaker security property). To understand the attack, recall that in Gentry's blueprint, the bootstrapping key is an encryption of the secret key. So, by querying the decryption oracle with the bootstrapping key, the attacker may recover the secret key and decrypt any ciphertext sent by the challenger. Some attacks are even possible with more constrained decryption oracles: for example [Lof+12] demonstrates attacks with only an access to an oracle that tells if a ciphertext is valid or not, which seems a very practical concern.

Because these advanced security properties are so hard to achieve, the *de facto* standard for FHE schemes has become CPA security (where the attacker has only access to an encryption oracle and no decryption oracle). This is the level of security targeted by most applied libraries. However, such security level is not sufficient in practice for most of the use-cases of FHE.

The problem has first been revealed in the work of [LM21] for approximate homomorphic schemes. In their work, they leverage the fact that the decryption function leaks the noise in ciphertexts to mount a secret-key recovery attack. They named this framework $\mathsf{CPA}^{\mathsf{D}}$ ($\cdot^{\mathsf{D}}$ denoting the use of the output of the Decryption algorithm). To be protected against such attacks, users should be prohibited to disclose the results of decrypted messages. But in some use-cases (such as Multi-Party Computations), it is necessary to share the decrypted message with other users. So the outputs of the decryption algorithm should be protected by injecting random noise to flood the leaked information [CHK20]. Actually, further works [Che+24a; Che+24b] have shown that non-approximate schemes are also vulnerable to $\mathsf{CPA}^{\mathsf{D}}$ attacks: instead of exploiting the noise leakage, the attacker can actively tamper with the noise in ciphertexts to trigger decryption errors and harvest information to mount a key-recovery attack.

What these attacks show is that FHE by itself is not sufficient to build secure applications in the real world. To prevent against tampering of ciphertexts, CPA-secure FHE schemes need to be augmented with some machinery to prove the well-formedness of ciphertexts. Some literature on the topic has been developed in the recent years, notably the works of [MN24; Brz+25]. They develop the new security notion of vCCA (for *verified chosen ciphertext security*) and implement it in practice with SNARKS [Bit+12] (succing non-interactive argument of knowledge).

Chapter

2 Presentation of the TFHE Scheme

The previous chapter presented insights on Fully Homomorphic Encryption in general, but this thesis will exclusively cover the TFHE scheme.

Introduced in [Chi+16; Chi+17] as an evolution of the FHEW scheme [DM15], TFHE quickly gained traction to become one of the most promising scheme to attain performances good enough to be used in real-world scenarios. It has been enriched with more features later: and more complete versions of this work have been published in [Chi18; Chi+20].

At the core of TFHE lies a powerful *Programmable Bootstrapping* operation (Programmable Bootstrapping, see Section 2.7.2 (PBS)). As we presented in Section 1.2, it allows to manage the noise in the ciphertexts during the computation, allowing to achieve Fully Homomorphic Encryption. In TFHE, this bootstrapping operation goes a step further: it enables the evaluation of arbitrary functions directly on the refreshed ciphertexts, with no computational or noise overhead.

In this chapter we provide an in-depth presentation of the TFHE scheme. We introduce the hardness assumptions it relies on, its encryption procedure as well as its homomorphic capabilities. We conclude with insights into its practical performance.

2.1 Hardness Assumptions: LWE and GLWE Problems

Original LWE problem. In 2005, Regev laid the foundations for an important part of modern lattice-based cryptography by defining the Learning With Errors (LWE) problem in [Reg05]. The version usually used in FHE literature is presented in Definition 2.1.1:

Definition 2.1.1. (Learning With Errors). Let q and n two integers, respectively called *modulus* and *dimension*. Let χ_s and χ_e be distributions over *small* values of $\mathbb{Z}_q$. We consider a secret vector, sampled as: $\mathbf{s} = (s_0, \ldots, s_{n-1}) \xleftarrow{\$} \chi_s^n$. The LWE distribution $\mathcal{D}^{\mathsf{LWE}}_{q,n,\chi_s,\chi_e}(\mathbf{s})$ is defined as:

$$\mathcal{D}^{\mathsf{LWE}}_{q,n,\chi_s,\chi_e}(\mathbf{s}) = \left\{ (\mathbf{a}, b) \;\middle|\; \mathbf{a} = (a_0, \ldots, a_{n-1}) \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_q\right)^n, e \xleftarrow{\$} \chi_e, b = \langle \mathbf{a}, \mathbf{s} \rangle + e \right\}$$

The *decisional* version of the problem is to distinguish this distribution from a uniformly random one, namely:

$$\mathcal{D}^{(\mathsf{random})} = \left\{ (\mathbf{a}, r) \;\middle|\; \mathbf{a} \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_q\right)^n, r \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_q\right) \right\}$$

The *search* version of the problem is to recover $\mathbf{s}$ from samples of $\mathcal{D}^{\mathsf{LWE}}_{q,n,\chi_s,\chi_e}(\mathbf{s})$.

Regev proved that the search and decisional problems are reducible to each other and their average case is as hard as worst-case lattice problems.

The hardness of this problem depends on the parameters q, n, χ_s and χ_e, and so does the security of the schemes built upon it. Common practice in the field is to derive an approximate concrete security level λ for a given parameter set with a tool named `lattice-estimator`

[APS15]. Users can input concrete values and distributions for the parameters, and the tool evaluates the security of the underlying LWE instance by running simulations of attacks of the literature.

In Definition 2.1.1, we did not specify the shapes of the distributions χ_s and χ_e (beyond the fact they yields small values). Several distributions are possible: a discrete Gaussian with a small variance, a uniform distribution restricted on a small interval, or a binomial. Some versions with sparse secrets also exist. The choice of the right distribution depends of the considered use-case, each possibility offering a different trade-off in terms of security and efficiency [Buc+16; CP19; Bha+19; Sha+24].

Most implementations of TFHE select a uniform distribution on $\{0, 1\}$ for the secret, and a Gaussian with a small variance σ^2_{LWE} for the noise. We will use these distributions in this thesis, and will use the notation $\mathsf{LWE}_{(q,n,\sigma)}$ for these instances.

Extension to the Polynomials. Looking for more efficient solutions, the LWE problem has been extended in a *ring variant* named RLWE in [LPR10; Ste+09]. A generalized version over rings named GLWE, first formalized in [BGV11] and used by TFHE, is presented below. It is very similar to the LWE one, but deals with polynomial values instead of integers:

Definition 2.1.2. (Generalized Learning with Errors) Let q and k two integers, and N a power of two, respectively called *modulus*, *dimension* and *degree*. We consider the polynomial ring $\mathbb{Z}_q[X]/(X^N + 1)$, that we denote in short $\mathbb{Z}_{N,q}[X]$. Let χ_S and χ_E be distributions over the small values of $\mathbb{Z}_{N,q}[X]$, (that is to say, polynomials with small coefficients). We consider a secret vector $\mathbf{S}$, sampled as $\mathbf{S} = (S_0, \ldots, S_{k-1}) \xleftarrow{\$} \chi_S^k$. The GLWE distribution $\mathcal{D}^{\mathsf{GLWE}}_{q,k,N,\chi_S,\chi_E}(\mathbf{S})$ is defined as:

$$\mathcal{D}^{\mathsf{GLWE}}_{q,k,N,\chi_S,\chi_E}(\mathbf{S}) = \left\{ (\mathbf{A}, B) \;\middle|\; \mathbf{A} = (A_0, \ldots, A_{k-1}) \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_{N,q}[X]\right)^k, E \xleftarrow{\$} \chi_E, B = \langle \mathbf{A}, \mathbf{S} \rangle + E \right\}$$

The *decisional* version of the problem is to distinguish this distribution from a uniformly random one, namely:

$$\mathcal{D}^{(\mathsf{random})} = \left\{ (\mathbf{A}, R) \;\middle|\; \mathbf{A} \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_{N,q}[X]\right)^k, R \xleftarrow{\$} \mathcal{U}\left(\mathbb{Z}_{N,q}[X]\right) \right\}$$

The *search* version of the problem is to recover $\mathbf{S}$ from samples of $\mathcal{D}^{\mathsf{GLWE}}_{q,k,N,\chi_S,\chi_E}(\mathbf{S})$.

Note that if we fix $k = 1$, we fall back on the classical RLWE problem, notably used in BGV [BGV14]. Also, taking $N = 1$ produces an LWE instance with $n = k$.

Concretely, using polynomial rings allows to encode more information in a single sample, yielding more compact ciphertexts and public keys. The schemes can also benefit from high-speed polynomial arithmetic techniques such as Fast Fourier Transforms (FFT).

The general consensus is that the hardness of $\mathsf{GLWE}_{(q,k,N,\sigma)}$ is similar to the hardness of $\mathsf{LWE}_{(q,k\cdot N,\sigma)}$, which makes it possible to use the `lattice-estimator` as well.

2.2 Torus Equivalence and Discretization

The T in TFHE stands for *Torus*, because in the seminal paper of TFHE [Chi+20], the authors worked with torus-based variants of LWE and GLWE.

The torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ corresponds to the reals modulo 1. Algebraically, this space does not have a ring structure, but actually is a $\mathbb{Z}$-*module* one, which means that:

- The sum of two torus elements is well-defined, and yields another torus element.

- The multiplication between an element of $\mathbb{T}$ and an element of $\mathbb{Z}$ is also well-defined, and produces an element of $\mathbb{T}$.
- On the other hand, multiplying two elements of $\mathbb{T}$ does not make sense. To be convinced of it, we can remark that, for any non-zero torus element x, $0 \times x = 0$ while $1 \times x = x$. But since 0 and 1 are equivalent over the torus, these results should not be different.

Recall the LWE assumption (Definition 2.1.1). If we rescale the elements of $\mathbb{Z}_q$ by dividing them by q, we get elements of the discretized torus. We can then redefine seamlessly the LWE problem over the torus. Extensive details about this transformation can be found in [Chi18].

This brings two advantages:

- LWE over the torus is *scale-invariant*, which makes the analysis of the security and of the noise much simpler.
- The $\mathbb{Z}$-module structure propagates in the ring versions, as well as in matrices spaces. Thus, it allows for very powerful generalizations of homomorphic schemes on a wide variety of spaces, like in [Bou+20; Bel+24].

When implementing the scheme in practice, torus elements are represented by integers in machine. The torus is thus seen as *discretized*, which we denote by

$$\mathbb{T}_q = \left\{ \frac{a}{q} \;\middle|\; a \in \mathbb{Z}_q \right\}$$

with $q = 2^\Omega$ (Ω denotes the number of bits of precision of the type, so 32 or 64 bits in most implementations). The properties of the torus structure are preserved.

This thesis is mainly about practical instantiations of the scheme. So, for the sake of clarity we will adopt a notation closer to the reality of the objects manipulated in machine. So the torus elements will be seen as elements of $\mathbb{Z}_q$ (but keeping the algebraic rules imposed by the structure of $\mathbb{T}$), and the same will be applied for ring extensions.

2.3 Encryption and Decryption in TFHE

Spaces. To understand the encryption procedure of TFHE, we must first introduce its plaintext space and its ciphertext space, and how the former can be embedded into the latter.

The plaintext space of TFHE is the *discretized torus* $\mathbb{T}_p$. As explained in Section 2.2, we trivially identify it to the ring $\mathbb{Z}_p$ with p an integer. Conversely, the ciphertext space is the discretized torus $\mathbb{T}_q$ introduced in the previous section identified to a ring $\mathbb{Z}_q$, with $q = 2^\Omega$. In practice, $p \ll q$.

We need a way to encode plaintext values into the ciphertext space. To do so, let us consider a mapping $\rho : \mathbb{Z}_p \to \mathbb{Z}_q$, defined as

$$\rho : m \mapsto \left\lfloor \frac{m \cdot q}{p} \right\rceil .$$

The image of this mapping only reaches p elements in $\mathbb{Z}_q$, forming the set $\left\{ \left\lfloor \frac{kq}{p} \right\rceil \mid k \in \mathbb{Z}_p \right\}$. These elements are distributed around $\mathbb{Z}_q$ and form what we refer to as *sectors of* $\mathbb{Z}_q$, defined as:

$$\left\{ \left(\frac{(2k-1)q}{2p}, \frac{(2k+1)q}{2p} \right) \;\middle|\; k \in \mathbb{Z} \right\} .$$

A representation of such a mapping is shown on Figure 2.1.

To encode a given plaintext element m into the ciphertext space, we use the corresponding center of sector $\left\lfloor \frac{mq}{p} \right\rceil$. However, decoding is more permissive: every elements of a sector are

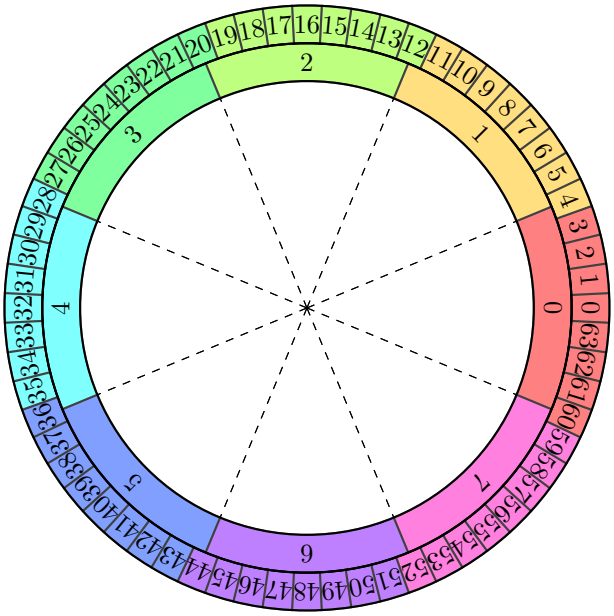


Figure 2.1: An example of embedding of $\mathbb{Z}_8$ into $\mathbb{Z}_{64}$

decoded by the corresponding plaintext element. This will be useful to remove the noise in ciphertexts.

We can now move to the actual encryption and decryption algorithms. TFHE features two main types of encryption: LWE encryption and GLWE encryption. Both share similar structural patterns but operate within different mathematical spaces.

LWE Encryption. LWE encryption deals with scalar values. The plaintext space is $\mathbb{Z}_p$ and the secret key is sampled uniformly at random from $\mathbb{B}^n$. We denote by q the ciphertext modulus. We also need $\chi_{\sigma_{\mathsf{LWE}}}$, a centered Gaussian distribution of standard deviation σ_{LWE} in $\mathbb{Z}_q$. The encryption algorithm produces a ciphertext $\mathbf{c}$ of the form:

Definition 2.3.1. (LWE ciphertext) A LWE ciphertext encrypting a message $m \in \mathbb{Z}_p$ under a secret key $\mathbf{s} = (s_0, \ldots, s_{n-1}) \in \mathbb{B}^n$ has the form:

$$\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(m) = \left(a_0, \ldots, a_{n-1}, b = \sum_{i=0}^{n-1} a_i \cdot s_i + \tilde{m} + e \right) \in \mathbb{Z}_q^{n+1} \tag{2.1}$$

where:

- the elements $\mathbf{a} = (a_0, \ldots, a_{n-1})$ are sampled uniformly at random in $\mathbb{Z}_q$.
- $\tilde{m}$ is the message encoded in the ciphertext space: $\tilde{m} = \rho(m) \in \mathbb{Z}_q$.
- e is a small random Gaussian noise sampled from $\chi_{\sigma_{\mathsf{LWE}}}$.

Decryption is performed in two steps: first, we compute the *phase* of the ciphertext as $\phi(\mathbf{c}) = b - \langle \mathbf{a}, \mathbf{s} \rangle$. The phase corresponds to the noisy message $\tilde{m} + e$. To recover the actual message, we simply decode the phase by looking up the plaintext element corresponding to the sector. This can be interpreted as a rounding: $m = \left\lfloor \frac{p}{q} \phi(c) \right\rceil$.

As long as $|e| < \frac{q}{2p}$, this rounding produces the right sector center, and thus we recover the correct plaintext value. Otherwise the phase lies in a different sector and we recover a wrong value. It is thus very important to keep the noise level low enough to ensure correct decryption.

Security-wise, this encryption relies on the hardness of the assumption $\mathsf{LWE}_{(q,n,\sigma_{\mathsf{LWE}})}$. The dimensioning of these parameters should be handled properly to ensure security, correctness and efficiency. Chapter 8 of this thesis will be dedicated to this question.

GLWE Encryption. This encryption mode mirrors the structure of LWE encryption but operates within polynomial rings. This time, the plaintext space is $\mathbb{T}_p[X]/(X^N+1)$, identified with the ring $\mathbb{Z}_{N,p}[X]$.

The secret key $\mathbf{S}$ is represented as a vector $(S_0, \ldots, S_{k-1})$, sampled uniformly at random from $\Big(\mathbb{B}[X]/(X^N+1)\Big)^k$. The message is encoded in a polynomial $\tilde{M} \in \mathbb{Z}_{N,q}[X]$, with the same encoding process than LWE (but applied coefficient-wise). The noise is also a polynomial from the same ring, whose coefficients are drawn from the distribution $\chi_{\sigma_{\mathsf{GLWE}}}$.

The encryption procedure outputs a ciphertext $\mathbf{C}$ of form:

Definition 2.3.2. (GLWE ciphertext) A GLWE ciphertext encrypting a message $M \in \mathbb{Z}_{N,p}[X]$ under a secret key $\mathbf{S} = (S_0, \ldots, S_{k-1}) \in \Big(\mathbb{B}[X]/(X^N+1)\Big)^k$ has the form:

$$\mathbf{C} = \mathsf{GLWE}_{\mathbf{S}}(M) = \left(A_0, \ldots, A_{k-1}, B = \sum_{i=0}^{k-1} A_i \cdot S_i + \tilde{M} + E\right) \in \mathbb{Z}_{N,q}[X]$$

where:

- the elements $\mathbf{A} = (A_0, \ldots, A_{k-1})$ are sampled uniformly at random in $\mathbb{Z}_{N,q}[X]$.
- $\tilde{M}$ is the message encoded in the ciphertext space: $\tilde{M} = \rho(M)$.
- E is a polynomial with small Gaussian coefficients, that are sampled from $\chi_{\sigma_{\mathsf{GLWE}}}$.

Decryption follows the same steps as the LWE case: the phase is computed as $\phi(\mathbf{C}) = B - \langle \mathbf{A}, \mathbf{S} \rangle$ and rounded to the closest plaintext value.

The security of this encryption relies on the hardness of the assumption $\mathsf{GLWE}_{(q,k,N,\sigma_{\mathsf{GLWE}})}$. Because the plaintext is a polynomial of degree N, it is possible to trivially batch up to N plaintexts from $\mathbb{Z}_p$ by encoding them in the coefficients. With this method, they can be processed in parallel by the linear homomorphism presented below (but not by the bootstrapping!).

To give an idea of the size of the objects at play here, n is usually chosen between 500 and 1000, q is usually 2^{64} and N is a power of two between 2^8 and 2^{12}. Chapter 8 is dedicated to the problem of parametrization of the scheme, and gives further explanations on the role and the range of each parameter.

A third encryption flavour, named GGSW also exists. We introduce it in Section 2.6 where it is necessary.

2.4 Linear Homomorphisms

On a torus, two operations are well-defined: the sum of two torus elements and the external product between a torus element and a *scalar*.

It is not hard to see that both TFHE encryption modes are linearly homomorphic. We define the two operations `sumTFHE` and `clearMultTFHE` that we present in the following. We present only the LWE version, but they can be trivially transposed to GLWE as well.

`SumTFHE`$(\mathbf{c}, \mathbf{c}')$: Let $\mathbf{c} = (a_0, \ldots, a_{n-1}, b)$ and $\mathbf{c}' = (a'_0, \ldots, a'_{n-1}, b')$ be two LWE ciphertexts encrypting the messages m and m' from $\mathbb{Z}_p$, with respective noise variance σ and σ'. Summing coefficient-wise both ciphertexts yields a new ciphertext $\mathbf{c}'' = (a_0 + a'_0, \ldots, a_{n-1} + a'_{n-1}, b + b')$ encrypting $m + m'$ with a larger noise $e + e'$.

`ClearMultTFHE`(c, λ): Let $\mathbf{c} = (a_0, \ldots, a_{n-1}, b)$ an LWE ciphertext encrypting a message $m \in \mathbb{Z}_p$ and $\lambda \in \mathbb{Z}_p$ a constant. Multiplying each coefficient by the constant yields a new ciphertext $\mathbf{c}' = (\lambda \cdot a_0, \ldots, \lambda \cdot a_{n-1}, \lambda \cdot b)$ encrypting the message $\lambda \cdot m$ with a larger noise $\lambda \cdot e$.

These notations will be used throughout this manuscript. However, sometimes when it is clear from the context that we are referring to homomorphic operations, we may use the classical $+$ and $\cdot$ symbols to lighten the formulas.

2.5 Keyswitching

We now introduce a more advanced operation: the `KeySwitch`. Let $\mathbf{c} = \mathsf{LWE}_{\mathbf{s}}(m) = (a_0, \ldots, a_{n-1}, b)$ be an LWE ciphertext encrypting a message m under the secret key $\mathbf{s}$. `KeySwitch` allows the server to homomorphically transform c into a new ciphertext c' encrypting the same message m under another secret key $\mathbf{s}'$. This operation is very useful in practical settings: for example it allows to switch between different ciphertext types and shapes to speed up some computations, particularly in the bootstrapping algorithm. It is also central in some multi-client use-cases.

We start by explaining the LWE-to-LWE version of keyswitching and then generalize to the ring case.

Some intuition on keyswitching. The rationale behind the keyswitching algorithm is to homomorphically evaluate the linear part of the decryption function (namely $b - \langle \mathbf{a}, \mathbf{s} \rangle$), given an encryption of $\mathbf{s}$ under the key $\mathbf{s}'$. This "encrypted secret key" is called the *keyswitching key* and is denoted by KSK. Even if it looks a lot like a bootstrapping operation, keyswitching is actually quite different. Notably, it *increases* the noise in the ciphertext (see [Chi+20; Tap23] for concrete noise analysis).

More formally, let KSK be the vector of encryption of every bit of the key $\mathbf{s}$ under the key $\mathbf{s}'$:

$$\mathsf{KSK} = \{\mathsf{LWE}_{\mathbf{s}'}(s_i)\}_{0 \leq i < n}$$

Notice how, if we treat the coefficients a_i (the mask of $\mathbf{c}$) as scalar constants, we can homomorphically evaluate the product $-\langle \mathbf{a}, \mathbf{s} \rangle$ with the basic linear operations of TFHE. Moreover, adding the constant b is easy if we remark that the trivial ciphertext $\mathsf{TrivialLWE}(b) = (0, \ldots, 0, b)$ is a valid instance of $\mathsf{LWE}_{\mathbf{s}'}(b)$. So the operation $b - \langle \mathbf{a}, \mathbf{s} \rangle$ is equivalent in the encrypted world to:

$$\begin{aligned} b - \langle \mathbf{a}, \mathbf{s} \rangle &\sim \mathsf{TrivialLWE}(b) - \langle \mathbf{a}, \mathsf{KSK} \rangle \\ &= \mathsf{LWE}_{\mathbf{s}'}(b - \langle \mathbf{a}, \mathbf{s} \rangle) \\ &= \mathsf{LWE}_{\mathbf{s}'}(m + e) \end{aligned}$$

and we effectively get a new encryption of m under $\mathbf{s}'$ (at the cost of some extra noise).

However, there is a problem with this approach: recall that the a_i's are uniformly distributed in the ring $\mathbb{Z}_q$. So they have in average a very large magnitude (about $\frac{q}{4}$). Also recall that the scalar multiplication of TFHE increases the noise by the same factor as the multiplicative constant. So if one uses this first version of keyswitching, the noise would completely skyrocket and the result would be unusable.

Thankfully, there is a well-known way to improve the noise growth in the scalar multiplication: it is called *gadget decomposition*. We introduce it below:

Gadget Decomposition. To begin with, we introduce an *exact* version of the gadget decomposition for clarity. The variant used in TFHE is an *approximate* one, which is conceptually only slightly more complex.

Recall that we want to compute the scalar product $a_i \cdot \mathsf{LWE}_{\mathbf{s}}(m)$, with a_i a potentially large value in $\mathbb{Z}_q$, without noise explosion.

The core idea is to work with a *decomposition* of the constant a_i in some basis. Let $(\mathfrak{B}, \ell)$ be two integers such that $\mathfrak{B}^\ell = q$. We denote the decomposition of a_i in this basis by:

$$\mathbf{dec}^{(\ell,\mathfrak{B})}(a_i) = (a_{i,0}, \ldots, a_{i,\ell-1}) \text{ such that: } a_i = \sum_{j=0}^{\ell-1} a_{ij} \cdot \mathfrak{B}^j$$

In parallel, instead of working with a single ciphertext $\mathsf{LWE}_{\mathbf{s}}(m)$, we use a collection of ℓ ciphertexts, each encrypting a scaled version of m. These ciphertexts are fresh encryption, so they all have the same fresh noise level σ_{LWE}.:

$$\left\{\mathsf{LWE}_{\mathbf{s}}(m \cdot \mathfrak{B}^j)\right\}_{0 \leq j < \ell}$$

Now, observe that instead of directly performing the product $a_i \cdot \mathsf{LWE}_{\mathbf{s}}(m)$, we can instead performs the sum

$$\sum_{j=0}^{\ell-1} a_{i,j} \cdot \mathsf{LWE}_{\mathbf{s}}(m \cdot \mathfrak{B}^j) \text{ with: } \mathbf{dec}^{(\mathfrak{B},\ell)}(a_i) = (a_{i,0}, \ldots, a_{i,\ell-1})$$

which yields the correct result. Computing the product this way makes the noise grow only by a factor $\mathfrak{B} \cdot \ell$ instead of a_i (at the cost of storing ℓ ciphertexts instead of one).

Note that what we just presented here was a very simple case. A rigorous formalism of gadget decomposition is developed in [GMP19], and some more analysis can be found in [Joy21].

One of the innovations of TFHE is the realization that approximating the decomposition yields a significant performance improvement, at the cost of only a slight degradation of the noise. More formally, a gadget decomposition can be described with two metrics: the *quality* that quantify the maximal magnitude attained by the coefficients a_i's (the lower the better), and the *precision* that quantify how close the output of the gadget decomposition is from the ones of an exact decomposition. Seminal works on TFHE have shown that in practice it is possible to construct gadgets ensuring simultaneously good enough quality and precision, while significantly accelerating the computations and reducing the memory requirements.

So instead of taking exactly $\mathfrak{B}^\ell = q$, TFHE uses smaller values such that $\mathfrak{B}^\ell < q$. In Chapter 8, we introduce a tool for parameter selection allowing (among other features) to construct good gadgets.

Back to a better keyswitching. Coming back to the keyswitching algorithm, we pick decomposition parameters ($\mathfrak{B}$, ℓ) and add a dimension to the keyswitching key to store each scaled version. So KSK becomes:

$$\mathsf{KSK} = \left\{\left(\mathsf{LWE}_{\mathbf{s}'}(s_i \cdot \mathfrak{B}^0), \ldots \mathsf{LWE}_{\mathbf{s}'}(s_i \cdot \mathfrak{B}^{\ell-1})\right)\right\}_{0 \leq i < n}$$

and we replace the simple scalar multiplications in the keyswitching algorithm by inner products between the decompositions of each a_i and each member of the keyswitching key. This gives us the full LWE-to-LWE algorithm, that we detail in Algorithm 1

Algorithm 1 LWE-to-LWE `KeySwitching`

Context:
- $\mathbf{s}$: the input LWE secret key
- $\mathbf{s}'$: the output LWE secret key
- ℓ: the level of the decomposition
- $\mathfrak{B}$: the base of the decomposition

Input:
- $\mathbf{c} = (a_0, \ldots, a_{n-1}, b)$: a ciphertext $\mathsf{LWE}_{\mathbf{s}}(m)$
- $\mathsf{KSK} = \left\{\left(\mathsf{LWE}_{\mathbf{s}'}(s_i \cdot \mathfrak{B}^0), \ldots, \mathsf{LWE}_{\mathbf{s}'}(s_i \cdot \mathfrak{B}^{\ell-1})\right)\right\}_{0 \leq i < n}$: the keyswitching key

Result: $\mathbf{c}_{\mathsf{out}}$: a ciphertext $\mathsf{LWE}_{\mathbf{s}'}(m)$

$\mathbf{c}_{\mathsf{out}} \leftarrow (0, \ldots, 0, b)$
for $i \in \{0, \ldots, n-1\}$ **do**
  $\mathbf{c}_{\mathsf{out}} \leftarrow \mathbf{c}_{\mathsf{out}} - \left\langle \mathbf{dec}^{(\ell,\mathfrak{B})}(a_i), \mathsf{KSK}_i \right\rangle$
end
return $\mathbf{c}_{out}$

Generalization to GLWE. We introduced keyswitching in its LWE-to-LWE form, but everything generalizes nicely to construct a LWE-to-GLWE flavour. Here, KSK is a collection of GLWE ciphertexts, and the decomposition is applied coefficient-wise on the polynomials. The resulting GLWE ciphertext encrypts a polynomial whose degree-zero coefficient encodes the original message.

It is then possible to pack several LWE ciphertexts into a single GLWE one, by multiplying the results by different monomials to move the encoded coefficient in a higher degree. They can then be summed. This is known in the literature as the `PackingKeySwitch`. As this will be useful particularly in Chapter 5, we detail it in Algorithm 2.

Algorithm 2 `PackingKeySwitch`

Context:
- $\mathbf{s}$: the input LWE secret key
- $\mathbf{S}'$: the output GLWE secret key
- ℓ: the level of the decomposition
- $\mathfrak{B}$: the base of the decomposition

Input:
- $\{\mathbf{c}_i = \mathsf{LWE}_{\mathbf{s}}(m_i)\}_{0 \leq i < \alpha}$: a list of α LWE ciphertexts to be packed, with $\alpha \leq N$
- $\mathsf{KSK} = \left\{\left(\mathsf{GLWE}_{\mathbf{S}'}(s_i \cdot \mathfrak{B}^0), \ldots, \mathsf{GLWE}_{\mathbf{S}'}(s_i \cdot \mathfrak{B}^{\ell-1})\right)\right\}_{0 \leq i < n}$: the keyswitching key

Result: $\mathbf{C}_{\mathsf{out}}$: a GLWE ciphertext encrypting the list of messages $\{m_i\}_{0 \leq i < \alpha}$

$\mathbf{C}_{\mathsf{out}} \leftarrow 0$
for $k \in \{0, \ldots, \alpha - 1\}$ **do**
  $\mathbf{C}'_k \leftarrow \mathtt{KeySwitch}(\mathbf{c}_k, \mathsf{KSK})$
  $\mathbf{C}_{\mathsf{out}} \leftarrow \mathbf{C}_{\mathsf{out}} + X^k \cdot \mathbf{C}'_k$
end
return $\mathbf{C}_{out}$

More possibilities exist in the literature: notably generalizing Algorithm 1 to define a GLWE-to-GLWE flavour. It is also possible to evaluate functions while keyswitching by applying it on the decomposed scalars (making it a *public* functional keyswitch) or on the encryption of the bits of the original secret key (making it a *private* one). An in-depth tour of keyswitches can be found in [Tap23].

2.6 External Products

In the previous section, we showed how the use of gadget decomposition allowed for practical multiplications by constant. Actually, we can push it further: by using decompositions of ciphertexts themselves, it becomes possible to multiply two ciphertexts together! By reference to the original GSW scheme [GSW13], these decomposed ciphertexts are called GGSW ciphertexts (for *Generalized GSW*).

We start by formalizing a bit more the notion of gadget decomposition: while several decomposition algorithms exist on the literature, we focus in this thesis on the most classical one (called *canonical* in the seminal paper of TFHE). We recall its definition:

Definition 2.6.1. (Gadget Matrix)
Let $\mathbb{Z}_{q,N}[X]^{k+1}$ be the domain of GLWE. Let the two positive integers ℓ and $\mathfrak{B}$ be the base of the gadget. The gadget matrix $\mathcal{H} \in \mathbb{Z}_{N,q}[X]^{(k+1)\ell \times k+1}$ is defined by:

$$\mathcal{H} = \left(\begin{array}{c|c|c} q/\mathfrak{B} & \dots & 0 \\ \vdots & \ddots & \vdots \\ q/\mathfrak{B}^\ell & \dots & 0 \\ \hline \vdots & \ddots & \vdots \\ \hline 0 & \dots & q/\mathfrak{B} \\ \vdots & \ddots & \vdots \\ 0 & \dots & q/\mathfrak{B}^\ell \end{array}\right) \in \mathbb{Z}_{N,q}[X]^{(k+1)\ell \times k+1}$$

Using this matrix, it is possible to define a new type of ciphertexts, called GGSW ciphertexts. We give their classical definition below:

Definition 2.6.2. (GGSW ciphertexts) Let $(\mathfrak{B}, \ell)$ an approximate decomposition basis. A GGSW ciphertext $\mathcal{C}$ encrypting a message $M \in \mathbb{Z}_{N,p}[X]$ under a GLWE secret key $\mathbf{S} = (S_0, \dots, S_{k-1}) \in \mathbb{B}_{N,q}[X]^k$ has the form:

$$\mathcal{C} = \mathcal{Z} + M \cdot \mathcal{H}$$

where each row of $\mathcal{Z}$ is a valid GLWE ciphertext of 0 for the key $\mathbf{S}$.

In more recent works (for example [Chi+21] and its follow-ups), an alternative (but equivalent) definition is used. We reproduce it here as well:

Definition 2.6.3. (GGSW ciphertext, second definition) Let $(\mathfrak{B}, \ell)$ an approximate decomposition base. A GGSW ciphertext encrypting a message $M \in \mathbb{Z}_{N,p}[X]$ under a GLWE secret key $\mathbf{S} = (S_0, \dots, S_{k-1}) \in \mathbb{B}_{N,q}[X]^k$ has the form:

$$\mathsf{GGSW}_{\mathbf{S}}(M) = \left\{ \left(\mathsf{GLWE}_{\mathbf{S}} \left(-S_\alpha \cdot M \cdot \frac{q}{\mathfrak{B}^j} \right) \right)_{0 \le j < \ell} \right\}_{0 \le \alpha \le k}$$

with S_k fixed by convention to -1.

These ciphertexts allow the definition of an *external product*. It is possible to multiply a GGSW ciphertext (encrypting a message M_1) with a GLWE one (encrypting a message M_2) to obtain another GLWE ciphertext that encrypts the product $M_1 \cdot M_2$. We denote this operation by:

$$\boxdot : \mathsf{GGSW} \times \mathsf{GLWE} \to \mathsf{GLWE}$$

Algorithm 3 details this procedure. As for `KeySwitch`, the External Product increases the noise in the ciphertexts. Again, exact noise formulas can be found in [Chi+20; Tap23]. The line of work of [Bou+20; Bel+24] gives a nice mathematical overview of how these decompositions techniques can be interpreted as lifts from the $\mathbb{Z}$-module structure to the underlying ring.

Algorithm 3 `External Product`

Context: $\begin{cases} \mathbf{S}\text{: a GLWE secret key} \\ \ell\text{: the level of the decomposition} \\ \mathfrak{B}\text{: the base of the decomposition} \end{cases}$

Input: $\begin{cases} \mathcal{C}_1 = \left\{ \mathbf{K}_\alpha = \left(\mathsf{GLWE}_{\mathbf{S}} \left(-S_\alpha \cdot M \cdot \frac{q}{B^j} \right) \right)_{0 \leq j < \ell} \right\}_{0 \leq \alpha \leq k+1} \text{: a ciphertext } \mathsf{GGSW}_{\mathbf{S}}(M_1) \\ \mathbf{C}_2 = (A_0, \ldots, A_{k-1}, B)\text{: a ciphertext } \mathsf{GLWE}_{\mathbf{S}}(M_2) \end{cases}$

Result: $\mathbf{C}_{\mathsf{out}}$: a ciphertext $\mathsf{GLWE}_{\mathbf{S}}(M_1 \cdot M_2)$

$\mathbf{C}_{\mathsf{out}} \leftarrow \left\langle \mathbf{K}_k, \mathbf{dec}^{(\ell,\mathfrak{B})}(B) \right\rangle$
for $\alpha \in \{0, \ldots, k-1\}$ **do**
  $\mathbf{C}_{\mathsf{out}} \leftarrow \mathbf{C}_{\mathsf{out}} - \left\langle \mathbf{K}_\alpha, \mathbf{dec}^{(\ell,\mathfrak{B})}(A_\alpha) \right\rangle$
end
return $\mathbf{C}_{out}$

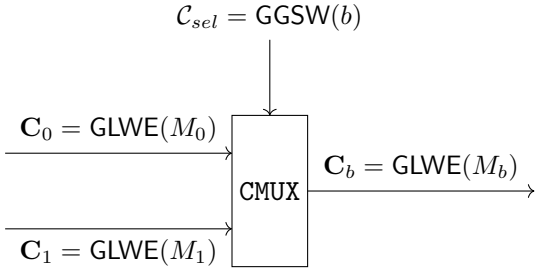


Figure 2.2: Representation of a `CMUX` operator

CMUX. Beyond adding a new homomorphic capabilities in TFHE's arsenal, external products are more importantly the workhorse of the whole bootstrapping algorithm that will be introduced in Section 2.7. An external product can be used to construct a CMUX operation, defined as follows:

Definition 2.6.4. (`CMUX`) Let $\mathcal{C}_{sel}$ be a GGSW ciphertext encrypting a bit $b \in \mathbb{B}$ (the *selector*). Let $\mathbf{C}_0$ and $\mathbf{C}_1$ be two GLWE ciphertexts. The CMUX operation allows to homomorphically select one of these two messages according to the value of the "selector" bit b by computing:

$$\mathbf{C}_b = \mathcal{C}_{sel} \boxdot (\mathbf{C}_1 - \mathbf{C}_0) + \mathbf{C}_0$$

where $\boxdot$ denotes the external product, and $+$ the homomorphic sum of TFHE. It produces a new ciphertext $\mathbf{C}_b$ encrypting the message M_b.

We can now move to the most important feature of TFHE: the *programmable bootstrapping*.

2.7 Programmable Bootstrapping

Programmable Bootstrapping is a quite complex construction. To present it, we start by an informal presentation (Section 2.7.1) and then move to the actual algorithm (Section 2.7.2). In the first informal part, we make some oversimplifications to make the process easier to understand. For a rigorous presentation of the algorithm, one shall refer to the second section.

2.7.1 An Informal Overview of Blind Rotation

Recall Gentry's blueprint introduced in Section 1.2. To be bootstrappable, a scheme requires to be able to evaluate its own decryption circuit in the encrypted space using its homomorphic capabilities.

In TFHE, for an LWE ciphertext $\mathbf{c}$ of form $(a_0, \ldots, a_{n-1}, b)$, encrypted under a key $\mathbf{s}$, the decryption algorithm has two steps:

- Computing the phase $\phi(\mathbf{c}) = b - \langle \mathbf{a}, \mathbf{s} \rangle$ ("the linear step"),
- Rounding the phase to the closest plaintext value ("the rounding step").

Performing the first step homomorphically is quite simple to do (in fact, this is exactly what we do in the keyswitching algorithm). But performing the rounding is trickier. Actually, TFHE do both operations at once using an operation called *blind rotation*, which is the core of the bootstrapping algorithm.

Introduced in [DM15], the blind rotation takes advantage of the particular structure of the ring $\mathbb{Z}_{N,q}[X]$. To explain this algorithm, we start by taking a closer look to this ring.

Fun with Rings. Let $v(X) = \sum_{i=0}^{N-1} v_i X^i$ be an element of the ring $\mathbb{Z}_{N,q}[X] = \mathbb{Z}_q[X]/(X^N+1)$, and let μ be an integer. Observe what happens when we multiply this polynomial with the monomial $X^{-\mu}$:

$$X^{-\mu} \cdot v(X) = v_\mu + v_{\mu+1} \cdot X + \cdots + v_{n-1} X^{n-1-\mu} - v_0 X^{n-\mu} - \ldots - v_{\mu-1} X^{n-1}.$$

What we can take away from this is that the monomial multiplication simply performs a *rotation* of the polynomial's coefficients (overlooking the red minus signs). In the bootstrapping algorithm, the blind rotation is used to bring the wanted coefficient in first position (so, in the degree-zero slot). If we want to rotate the polynomial such that the μ-th coefficient is brought into the degree-zero one, we just have to multiply the polynomial by the monomial $X^{-\mu}$.

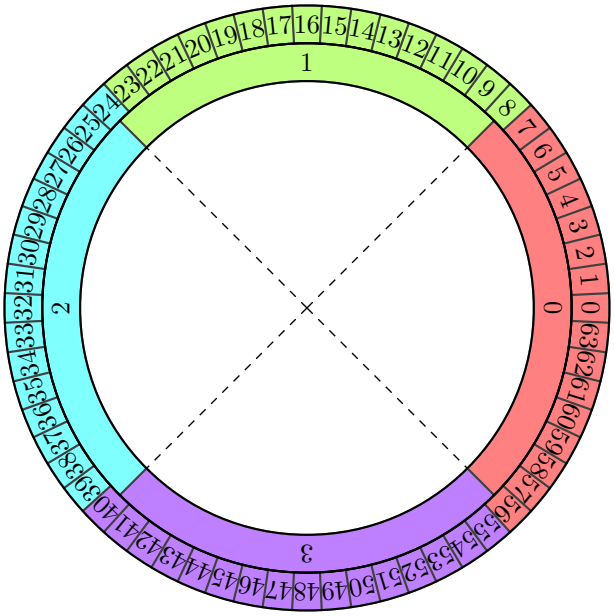
But there's a problem here. When the coefficients are "sent to the other side" of the polynomial, they get an extra minus sign. So actually, a multiplication by X^N does not yield the initial polynomial, but rather the "opposite" one. To make a complete round trip, it requires to multiply instead by X^{2N}. This is natural because X has order $2N$ in $\mathbb{Z}_{N,q}[X]$. In the literature, this problem is called *the negacyclicity problem* and we dive deeper into it in Chapter 3.

The goal here is to introduce blind rotation without the complexity induced by the negacyclicity problem. So for now, we assume that the exponent of the monomial lives in $\{0, \ldots, N-1\}$. As we are only interested in the value of the degre-zero coefficient, we can safely overlook these minus signs as they can not reach it.

Constructing a rounding function. How can we use the above property to construct a rounding function? Suppose we have a plaintext space $\mathbb{Z}_p$ and a ciphertext space $\mathbb{Z}_q$. Let $\mathbf{c}$ be an LWE ciphertext, and $\mu = \phi(\mathbf{c}) = m + e$ its noisy phase produced by the linear step of the decryption algorithm. We want to round μ to the closest plaintext in $\mathbb{Z}_p$ to retrieve the message m.

To do so, we are going to use a specifically tailored polynomial, called the *accumulator polynomial* $\mathsf{acc}(X)$. This polynomial is made of contiguous "windows" of size $\frac{N}{p}$ in which every coefficient is equal to each others. Coefficients of the window of index 0 have value 0, for index 1 the value is 1, and so on. These windows are centered on the p coefficients corresponding to the noiseless encodings of the plaintexts values in $\mathbb{Z}_N$. We illustrate such an accumulator on Figure 2.3 (in the "Rounding version").

Our rounding procedure takes three steps:



Rounding version:

$\cdots + 0 \cdot X^7$	$1 \cdot X^8 + \cdots + 1 \cdot X^{23}$	$2 \cdot X^{24} + \cdots + 2 \cdot X^{39}$	$3 \cdot X^{40} + \cdots + 3 \cdot X^{55}$	$0 \cdot X^{56} + \ldots$

Programmable version:

$\cdots + f(0)X^7$	$f(1)X^8 + \cdots + f(1)X^{23}$	$f(2)X^{24} + \cdots + f(2)X^{39}$	$f(3)X^{40} + \cdots + f(3)X^{55}$	$f(0)X^{56} + \ldots$

Figure 2.3: Example of an accumulator polynomial with $p = 4$ and $N = 64$, used to evaluate a simple rounding operation.

1. Switching the modulus of μ to send it into $\mathbb{Z}_N$ to produce $\tilde{\mu} = \left\lfloor \frac{\mu \cdot N}{q} \right\rceil$.
2. Computing the product $X^{-\tilde{\mu}} \cdot acc(X)$.
3. Extracting the degree-zero coefficient to retrieve $v_{\tilde{\mu}}$ (the $\tilde{\mu}$-th coefficient of the initial accumulator).

If the noise $|\tilde{e}|$ in $\tilde{\mu}$ is smaller than half the width of a window (here $\frac{N}{2p}$), we properly get $v_{\tilde{\mu}} = m$.

Making it programmable. This rounding using polynomials is the core idea of the bootstrapping algorithm. But actually, the killer feature of TFHE is that you can transform this rounding operation into a look-up table evaluation! This is why TFHE's bootstrapping is called *programmable*.

It is simply done by replacing the coefficients of the accumulator by the evaluations of a function $f : \mathbb{Z}_p \mapsto \mathbb{Z}_p$. So, the algorithm outputs $f(i)$ instead of i! This is illustrated on Figure 2.3 (in the "Programmable version").

In the next section, we introduce the actual instantiation of the PBS algorithm, without the oversimplifications we made in this section.

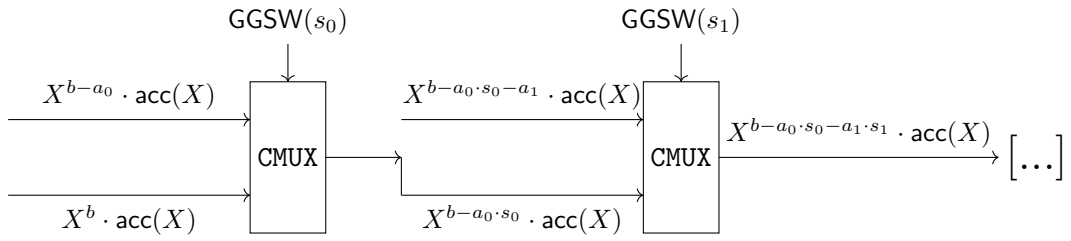


Figure 2.4: Illustration of the `BlindRotate` algorithm, implemented as a chain of `CMUX`.

2.7.2 The Full Algorithm

We consider an LWE ciphertext $\mathbf{c} = (a_0, \ldots, a_{n-1}, b)$ with large noise, that we want to bootstrap. It is encrypted under the LWE secret key $\mathbf{s}$.

The server requires a *bootstrapping key*, that we denote by BSK, defined by:

$$\mathsf{BSK} = \{\mathsf{GGSW}_{\mathbf{S}}(s_i)\}_{0 \leq i < n}$$

where $\mathbf{S}$ is a GLWE secret key, of dimension k and degree N.

TFHE's bootstrapping algorithm can be broken down into three steps: `ModSwitch`, `BlindRotate` and `SampleExtract`.

`ModSwitch`. The coefficients of $\mathbf{c}$ live in $\mathbb{Z}_q$, but the polynomials of the GLWE ring have degree N. Recall from last section that we would like the exponent of the monomial used in `BlindRotate` to live in $\mathbb{Z}_{2N}$ (because X has order $2N$ in $\mathbb{Z}_{N,q}[X]$).

The simplest case would be to choose directly $q = 2N$. However it does not work in practice: by enforcing such a constraint, it is impossible to construct an LWE instance which is secure and that enables fast arithmetic. So we need to switch the modulus of $\mathbf{c}$ to produce a new vector $\tilde{\mathbf{c}}$ living in the right space. This is simply done by computing:

$$\forall\, i \in \{0, \cdots, n-1\}, \tilde{a}_i = \left\lfloor \frac{a_i \cdot 2N}{q} \right\rceil_{2N} \text{ and: } \tilde{b} = \left\lfloor \frac{b \cdot 2N}{q} \right\rceil_{2N}$$

This operation adds some extra noise in the ciphertext, that is called *drift* in the literature. [Ber+25] provides an in-depth study of the behaviour of this noise, as well as alternative strategies to mitigate it.

`BlindRotate`. In the previous section, we introduced the rationale behind blind rotation. We use a polynomial called accumulator that we rotate using a multiplication by $X^{-\tilde{\mu}}$, where $\tilde{\mu}$ is the (noisy) phase resized in $\mathbb{Z}_{2N}$.

For now, we do not specify the actual formula of the accumulator. Several possibilities exist and depend of which countermeasure against the negacyclicity problem is chosen. We elaborate further on the construction of this polynomial in Chapter 3.

After `ModSwitch`, the coefficients of $\tilde{\mathbf{c}}$ live in the right space $\mathbb{Z}_{2N}$. It remains to compute the product $X^{-\tilde{\mu}} \cdot \mathsf{acc}(X)$ homomorphically. This operation can be carried out by *a chain of `CMUX`*.

This idea comes naturally if we rewrite the expression of the monomial $X^{-\tilde{\mu}}$ as:

$$X^{-\tilde{\mu}} = X^{-\left(\tilde{b} - \langle \tilde{\mathbf{a}}, \mathbf{s} \rangle\right)} = X^{-\tilde{b}} \cdot \prod_{i=0}^{n-1} X^{\tilde{a}_i \cdot s_i} = X^{-\tilde{b}} \cdot \prod_{i=0}^{n-1} \begin{cases} X^{\tilde{a}_i} \text{ if } s_i = 1 \\ 1 \text{ if } s_i = 0 \end{cases}$$

This leads to the natural algorithm depicted in Algorithm 4.

`BlindRotate` outputs a GLWE ciphertext encrypting the rotated accumulator under the GLWE key $\mathbf{S}$. The degree-zero coefficient of this polynomial encodes the rounded message. The

Algorithm 4 `BlindRotate`

Context: $\begin{cases} \mathbf{s}\text{: a } \textit{LWE secret key} \\ \mathbf{S}\text{: a } \textit{GLWE secret key} \\ f : \mathbb{Z}_p \mapsto \mathbb{Z}_p\text{: a function on the plaintext space.} \\ \mathsf{acc}(X) \in \mathbb{Z}_{N,q}[X]\text{: an accumulator polynomial encoding the function } f. \end{cases}$

Input: $\begin{cases} \tilde{\mathbf{c}} = (\tilde{a}_0, \dots, \tilde{a}_{n-1}, b) \in \mathbb{Z}_{2N}^{n+1}\text{: a modswitched LWE ciphertext, with } \tilde{\mu} = \tilde{b} - \langle \tilde{\mathbf{a}}, \mathbf{s} \rangle \\ \mathsf{BSK} = \{\mathsf{GGSW}_{\mathbf{S}'}(s_i)\}_{0 \leq i < n} \end{cases}$

Result: $\mathbf{C}_{\mathsf{out}} = \mathsf{GLWE}_{\mathbf{S}'}(X^{\tilde{\mu}} \cdot \mathsf{acc}(X))$

$\mathbf{C}_{\mathsf{out}} \leftarrow \mathsf{TrivialGLWE}(\mathsf{acc}(X) \cdot X^{-\tilde{b}})$
for $i \in \{0, \dots, n-1\}$ **do**
  $\mathbf{C}_{\mathsf{out}} \leftarrow \mathtt{CMUX}(\mathbf{C}_{\mathsf{out}}, X^{\tilde{a}_i} \cdot \mathbf{C}_{\mathsf{out}}, \mathsf{BSK}_i)$
end
return $\mathbf{C}_{out}$

problem is that we started with an LWE ciphertext, so if we want to keep computing (and so evaluate other PBS) we need to switch back to LWE encryption.

This is done in two steps: the `SampleExtract` that we introduce in the following, and then a regular LWE-to-LWE `KeySwitch` (that we introduced in Section 2.5).

`SampleExtract`. Let $\mathbf{C} = \mathsf{GLWE}_{\mathbf{S}'}(M)$ be a ciphertext encrypting the polynomial $M = \sum_{i=0}^{N-1} m_i X^i$. The `SampleExtract` takes as input a GLWE ciphertext and an index α, and output a ciphertext $\bar{\mathbf{c}} = \mathsf{LWE}_{\bar{\mathbf{s}}'}(m_\alpha)$ where $\bar{\mathbf{s}}'$ is a *flattened* version of the GLWE key.

Definition 2.7.1. (Flattened GLWE key) Let $\mathbf{S} = \left(S_0 = \sum_{j=0}^{N-1} s_{0,j} X^j, \dots, S_{k-1} = \sum_{j=0}^{N-1} s_{k-1,j} X^j\right) \in \mathbb{Z}_{N,q}[X]^k$. The *flattened* version of this key is a LWE secret key:

$$\bar{\mathbf{s}} = (\bar{s}_0, \dots, \bar{s}_{kN-1}) \in \mathbb{Z}_q^{kN}$$

such that $\bar{s}_{iN+j} = s_{i,j}$ for $0 \leq i < k$ and $0 \leq j < N$.

`SampleExtract` is actually a simple rearrangement of the coefficients of the ciphertext: its computational cost is negligible and it does not add any noise in the ciphertexts.

In the PBS, `SampleExtract` is run with $\alpha = 0$. After that, we have an LWE ciphertext encrypting the right value but with noise smaller than in the input. However, the key of this ciphertext is the flattened version of $\mathbf{S}'$. To close the loop, we need to come back to the original key $\mathbf{s}$. This is done with a simple LWE-to-LWE keyswitching.

Putting it all together. This serie of operations allows to evaluate a bootstrapping. The final keyswitching produces a ciphertext of same morphology and same secret key than the input one, so it is possible to loop and evaluate successively several functions. Linear operations can be interleaved in this loop, at any stage. On Figure 2.5, we show such a loop, where linear operations take place before the final keyswitching.

2.8 Performances of the PBS

In the FHE space, bootstrapping operations are notoriously heavy and slow operations. In the case of TFHE, it is relatively low-latency.

One key aspect of TFHE is *the degradation of the speed of the bootstrapping when the plaintext modulus p grows*. We ran some quick experiments on a laptop to show this degradation. The results are displayed on Figure 2.6.

Algorithm 5 `SampleExtract`

Context:
- $\mathbf{S}' = (S'_0, \ldots, S'_{k-1})$: the input GLWE secret key
- $\bar{\mathbf{s}}' = (\bar{s}_0, \ldots, \bar{s}_{kN-1})$: the output LWE secret key
- $\forall\ 0 \leq i < k,\ S_i = \sum_{j=0}^{N-1} \bar{s}_{iN+j} X^j$
- $M = \sum_{i=0}^{N-1} m_i \cdot X^i$: a polynomial message
- $\forall 0 \leq i < k, A_i = \sum_{j=0}^{N-1} a_{i,j} X^j$
- $B = \sum_{j=0}^{N-1} b_j X^j$

Input:
- $\mathbf{C} = (A_0, \ldots, A_{k-1}, B)$: a ciphertext $\mathsf{GLWE}_{\mathbf{S}}(M)$
- $\alpha \in \{0, \ldots, N-1\}$: the index of the coefficient to be extracted

Result: $\mathbf{c}_{\mathsf{out}}$: a ciphertext $\mathsf{LWE}_{\bar{\mathbf{s}}'}(m_\alpha)$

$b' \leftarrow b_\alpha$
for $0 \leq i < k$ **do**
  for $0 \leq j \leq \alpha$ **do**
    $a'_{i \cdot N + j} = a_{i, \alpha - j}$
  end
  for $\alpha + 1 \leq j < N$ **do**
    $a'_{i \cdot N + j} \leftarrow -a_{i, N + \alpha - j}$
  end
end
$\mathbf{c}_{\mathsf{out}} \leftarrow (a'_0, \ldots, a'_{k \cdot N - 1}, b')$

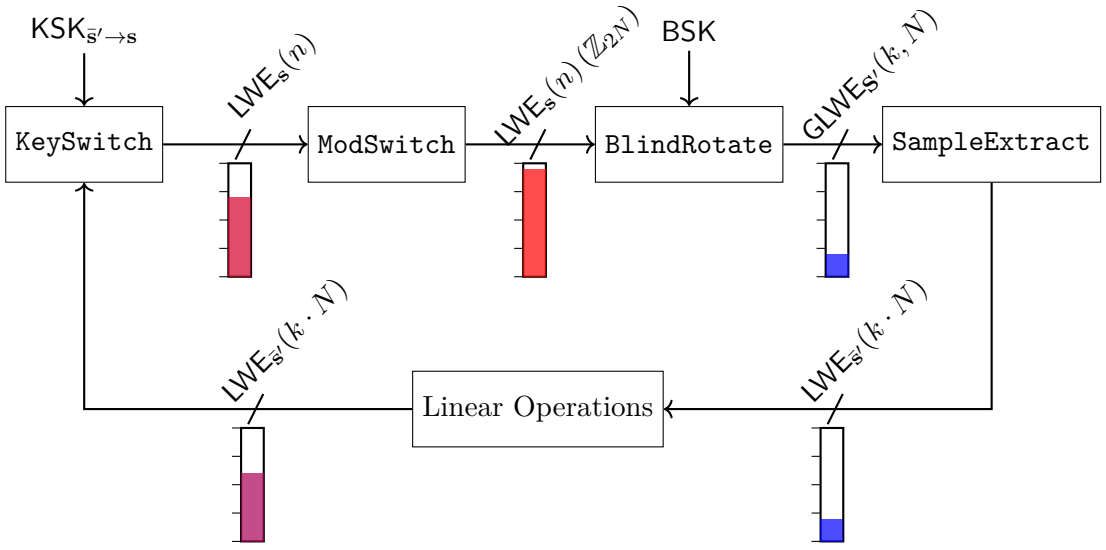


Figure 2.5: The PBS circuit, with the inner operations and the shapes of the ciphertexts at every step. Some arbitrary linear operations are interleaved in the middle. We also illustrate indicative noise levels in the gauges under the wires.

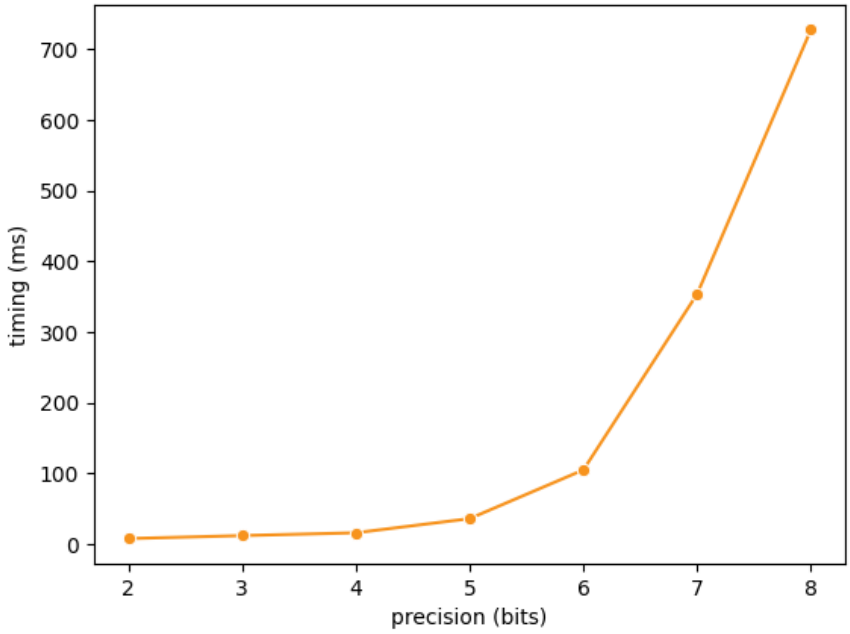


Figure 2.6: Timings of the PBS with respect to $\log_2(p)$*, where* p *is the plaintext modulus*

Intuitively, the reason of this degradation is simple. When one wants to use a greater p, the torus must be sliced into more parts, that are thus smaller. When we switch the modulus to $2N$ in the bootstrapping, the "windows" of the accumulator polynomial gets smaller, so the bound on the noise to ensure correct bootstrapping must be smaller. To keep some room to enable the homomorphic capabilities of TFHE, one has to pick a greater value for the degree N. As this must be a power of two, it quickly reach values where the polynomial operations gets very slow.

Usually, PBS is not used for plaintext modulus greater than 8 bits. Some constructions exist in the literature to extend the capabilities of this algorithm to larger plaintext spaces, notably the tree bootstrapping of [GBA21] and the Without-Padding Programmable Bootstrapping. See Section 3.3 and 7.4 (WoP-PBS) of [Chi+21]. In this thesis, we propose our own method in Chapter 7.

Chapter

3 Overcoming the Negacyclicity Problem with an Odd Plaintext Modulus

In Section 2.7, we presented the algorithm of the PBS. But we left a question unanswered: what is the actual composition of the accumulator polynomial $\mathsf{acc}(X)$? This is a tricky question, because this definition depends on the countermeasure picked against the negacyclicity problem (that we briefly introduced in Section 2.7.1).

In this chapter, we formalize the negacyclicity problem. We then present the classical countermeasure (the strategy of the padding bit), and our technique of odd plaintext modulus, which is a foundational block for the following contributions presented in this thesis. We finish by going through several others works that propose different countermeasures.

3.1 Basics on Negacyclicity

What is negacyclicity ? Let $v(X)$ be a polynomial of the ring $\mathbb{Z}_{N,q}[X]$, such that $v(X) = \sum_{k=0}^{N-1} v_k X^k$. Recall how a multiplication by X in this ring "rotates" the coefficients of the polynomial:

$$X \cdot v(X) = -v_{N-1} + v_0 \cdot X \cdots + v_{N-2} X^{N-1} \ .$$

In TFHE's blind rotation, the polynomial multiplication is done by the monomial $X^{-\tilde{\mu}}$, with $\tilde{\mu} \in \{0, \ldots, 2N-1\}$. This leads to two problems:

- A coefficient v_j can be brought in first place by two different rotations: the one induced by the polynomial multiplication by $X^{[-j]_{2N}}$ and the one by $X^{[-j+N]_{2N}}$. It means that two different messages produce the same output.
- Each time a coefficient goes last to first, it gets negated (because $X^N = -1$ in the ring). So actually, the multiplication by $X^{[-j]_{2N}}$ yields correctly v_j, but the one by $X^{[-j+N]_{2N}}$ yields $-v_j$.

As the actual value of $\tilde{\mu}$ is encrypted, this is not possible to predict beforehand whether this undesirable minus sign will appear or not. So a countermeasure needs to be implemented into the scheme to neutralize the negacyclicity problem. The most frequent strategy is to develop encodings for the LUT f in the accumulator specifically tailored to handle this issue.

The "natural" case. Some functions interact naturally very well with the blind rotation algorithm: these are called the *negacyclic functions* and they are presented in Definition 3.1.1.

Definition 3.1.1. (Negacyclic Function) Let p and p' be two positive integers, with p even. A function $f : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$ is negacyclic if and only if it satisfies the following property:

$$\forall x \in \mathbb{Z}_p, f\left(\left[x + \frac{p}{2}\right]_p\right) = [-f(x)]_{p'}$$

With such functions, the accumulator is quite simple to build: intuitively we only fill it with the values of the first half of the torus (so the $f(x)$ with $x \in [0, \frac{p}{2} - 1]$). So, if we rotate by the value corresponding to $i \geq \frac{p}{2}$, a minus sign appears and we retrieve correctly $-f\left(i - \frac{p}{2}\right)$.

We give an explicit formula for this accumulator in Definition 3.1.2

Definition 3.1.2. (Negacyclic Accumulator) Let p and p' be two positive integers, with p **even**. Let $f : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$ be a negacyclic function. Let N be a power of two. Then, the accumulator $\mathsf{acc}(X)$, defined by:

$$\mathsf{acc}(X)^{\mathsf{negacyclic}} = X^{\frac{-2N}{2p}} \cdot \sum_{j=0}^{2N/p-1} X^j \cdot \sum_{i=0}^{p/2-1} f(i) X^{i\frac{2N}{p}} \mod (X^N + 1)$$

is a valid accumulator for the blind rotation. It means that running the `BlindRotate` algorithm with this accumulator yields the right value.

Remark on the encoding: $f(i)$ is an element of the plaintext space $\mathbb{Z}_p$. In the context of this definition, we refer to its encoding into the ciphertext space $\mathbb{Z}_q$ (according to the procedure of Section 2.3). That is to say, we actually put $\left\lfloor \frac{f(i)\cdot q}{p} \right\rceil$ in the coefficients of the polynomial.

Unfortunately, this construction is merely theoretical. Indeed, in practical setting, restricting the functions to be evaluated by PBS only to negacyclic function greatly limits the capability of the scheme. In order to be able to evaluate *any* function in the PBS, more sophisticated techniques are required. We introduce the first example of such technique in the next section.

3.2 The Classical Countermeasure: the Bit of Padding

The first countermeasure, that already appeared in the original TFHE paper, is called the *bit of padding*. As in the ideal case presented in the previous section, it works in an **even** plaintext space.

The idea is to ensure that the message stays in the first half of the torus all along the computations. An equivalent way of presenting it is to force the Most Significant Bit (MSB) of the message to zero. By doing this, the modswitched phase used as the exponent in the blind rotation $\tilde{\mu}$ lives in $\{0, \ldots, N-1\}$, so the arising minus signs never reach the degree-zero coefficient.

The advantage of this method is that it relaxes the negacyclicity constraint and makes the bootstrapping able to evaluate any function (not only the negacyclic one). However, this construction is rather fragile: any linear function can break it by propagating a carry into the MSB.

To avoid this, when evaluating an homomorphic circuit, the program needs to keep track of the maximal value that each homomorphic operation can yield and make sure that it will never propagate a carry into the MSB. This often leads to extra bootstrapping to control the growth of the message and eliminate the overflows.

In the following, we give the definition of the accumulator $\mathsf{acc}(X)$ in this "bit-of-padding" case:

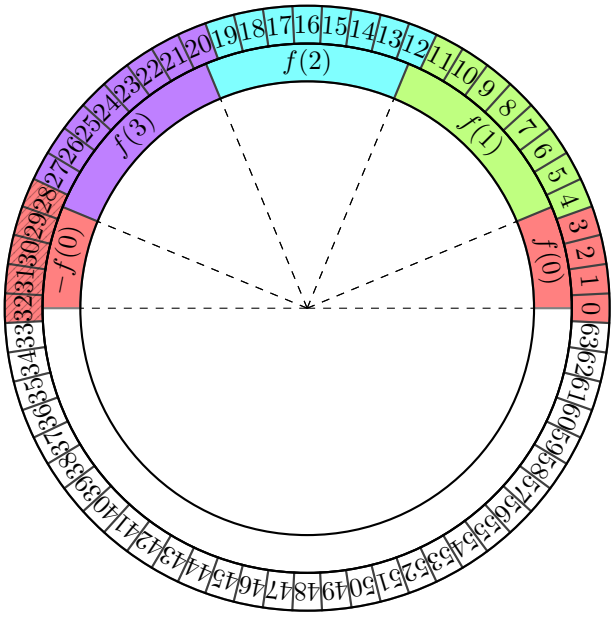
Definition 3.2.1. (Bit-of-Padding accumulator) Let p and p' be two positive integers, with p **even**. Let $f : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$ be a function. Let N be a power of two. Then, the accumulator $acc(X)$:

$$\mathsf{acc}(X)^{\text{bit-of-padding}} = X^{\frac{-N}{2p}} \cdot \sum_{j=0}^{N/p-1} X^j \cdot \sum_{i=0}^{p-1} f(i) X^{i\frac{N}{p}} \mod (X^N + 1)$$

is a valid accumulator for the blind rotation.

Remark on the encoding: Contrary to the purely negacyclic case of Definition 3.1.2, $f(i)$ is not encoded in the Most Significant Bits of the polynomial's coefficient. Instead, the MSB is reserved and fixed to zero to maintain the encoded value in the first half of the torus. That is to say, we actually put $\left\lfloor \frac{f(i)\cdot q}{2\cdot p} \right\rceil$ in the coefficients of the polynomial.

An example of such an accumulator is shown on Figure 3.1.



Bit-of-Padding Accumulator:

$\cdots + f(0)\cdot X^3$ | $f(1)X^4 + \cdots + f(1)X^{11}$ | $f(2)X^{12} + \cdots + f(2)X^{19}$ | $f(3)X^{20} + \cdots + f(3)X^{27}$ | $-f(0)X^{28} + \ldots$

Figure 3.1: An example of bit-of-padding accumulator, with $p = 4$ and $N = 32$. Hatching shows the parts which have an additional minus signs.

The drawback of the bit-of-padding padding makes very challenging the development of homomorphic applications, because it prevents to use the linear operations freely. One should keep track of the maximal value contained in a ciphertext to avoid an overflow that would fill the padding bit. In Chapter 4, we will demonstrate how this method makes the evaluation of Boolean circuit very inefficient, and we propose a new method that improves it.

Other constructions have been developed to avoid these drawbacks, we present them in Section 3.3.

3.3 Other Countermeasures Avoiding the Bit of Padding

Because the classical bit-of-padding solution brings too many constraints on the evaluation of computational circuits, several alternative constructions have been proposed to homomorphiclaly

evaluate LUT. To do so, many of these constructions adopt a similar strategy: they decompose the target function into a sequence of several PBS steps.

A notable line of work, including [Yan+21], [LMP22], and [KS23], employs a two-step PBS strategy. In these approaches, the first PBS evaluates a quantity encrypting the bit indicating whether the message lies in the positive or negative half of the torus. This effectively extracts the "sign" of the message. The second PBS then evaluates the actual function of interest regardless of the effects of negacyclicity, producing a result that may be flipped in sign. The ambiguity introduced by this flip is corrected using the result of the first PBS. While the specific techniques used to perform this correction vary across works, the overall architecture remains the same: use the first PBS to identify the torus half, and use this information to rectify the sign of the second PBS output.

Another interesting approach is presented in [Cle+23], where the target function is expressed as a sum of functions sharing a particular structure: so-called *pseudo-odd* and a *pseudo-even* function. These two properties are special cases of negacyclic functions, so they allow an evaluation with a PBS without suffering from the negacyclicity issue. Although this decomposition may require more than two PBS calls, an important advantage of the method is that the evaluations can be carried out in parallel. This contrasts with the previously mentioned approaches, which are intrinsically sequential due to the dependency between the two PBS stages.

A completely different line of work is explored in [Chi+21], which introduces the "Without-Padding PBS" (WoP-PBS) construction. A more refined version of this technique is presented in [Ber+23a], and we describe it here. The method begins by evaluating a scaled sign function (that is inherently negacyclic) using a standard PBS. Following this, the individual bits of the message are extracted and converted into GGSW ciphertexts (see Definitions 2.6.2 and 2.6.3) using a procedure known as *circuit bootstrapping*. These ciphertexts are then used as selectors in a graph of `CMUX` operations (Definition 2.6.4) to select an appropriate accumulator based on the high-order bits of the message. Finally, a traditional PBS is used to rotate the selected accumulator according to the remaining lower-order bits.

Although this technique is slower for small plaintext sizes, it scales particularly well with an increasing plaintext sizes. Notably, it enables the homomorphic evaluation of LUTs with larger plaintext sizes beyond what is feasible with classical PBS techniques, up to approximately 20 input bits.

In this thesis, we focus on a method we introduced in [BPR24]: *the odd plaintext modulus*.

3.4 Our Contribution: the Odd Plaintext Modulus

Negacyclicity has a quite different effect depending of the parity of the plaintext modulus p. Recall that $\mu = m + e \in \mathbb{Z}_q$, with e sampled from a small centered Gaussian. Because the error is small, μ does not take all the values of $\mathbb{Z}_q$ with the same probability: in particular, the densest parts in terms of probability over $\mathbb{Z}_q$ are the one close to the "noiseless" encodings of m, namely $\left\{ \left\lfloor \frac{kq}{p} \right\rceil \;\middle|\; k \in \mathbb{Z}_p \right\}$. We illustrate this distribution on Figure 3.2. We call these sections of the torus the *dense spots*.

When we transpose these dense spots into $\mathbb{Z}_{2N}$, they become the sectors close to $\left\{ \left\lfloor \frac{k \cdot 2N}{p} \right\rceil \mid k \in \mathbb{Z}_p \right\}$.

Let $k \in \mathbb{Z}_p$, the multiplication $X^{-\frac{k \cdot 2N}{p}} \cdot v(X)$ in the ring $\mathbb{Z}_{N,q}[X]$ yields the same degree-zero coefficient as the multiplication $X^{\left[-\frac{k \cdot 2N}{p} + N\right]_{2N}} \cdot v(X)$, up to the minus sign. To make the rest of this section clearer, we change a bit the writing of the exponent as such:

$$\left[\frac{-k \cdot 2N}{p} + N \right]_{2N} = \left[\frac{(-k + \frac{p}{2}) \cdot 2N}{p} \right]_{2N}$$

This is where the parity of p plays a part: if p is even, then $\left[\frac{(-k+\frac{p}{2}) \cdot 2N}{p} \right]_{2N}$ is a dense spot

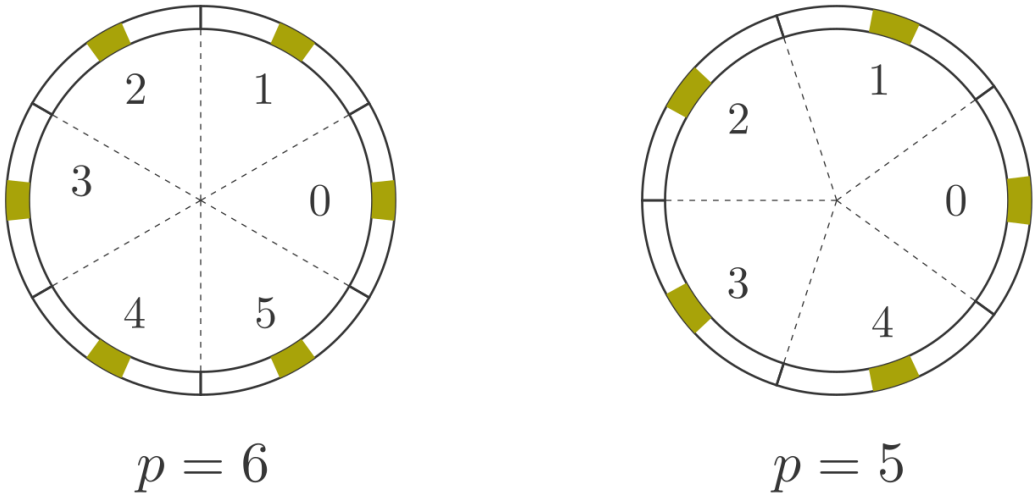


Figure 3.2: Distribution of the values of μ across $\mathbb{Z}_q$ for $p = 6$ and $p = 5$: the colored parts show the dense spots where the value has a high probability to lie in. The width of these sectors depends on σ (the standard deviation of the error distribution χ of TFHE). Note that this repartition looks similar for $\tilde{\mu}$ in $\mathbb{Z}_{2N}$.

(a) With p even, the dense spots of each half of the torus are aligned.

(b) With p odd, the dense spots face empty spots, close to the bounds of the p-sectors.

Figure 3.3

as well. So collisions happen with high probability. On the other hand, let us consider an odd value for p. Then, $\left\lceil \frac{(-k+\frac{p}{2})\cdot 2N}{p} \right\rfloor_{2N}$ is no longer a dense spot, as it lies exactly halfway between the two dense spots $\left\lceil \frac{(-k+\frac{p-1}{2})\cdot 2N}{p} \right\rfloor_{2N}$ and $\left\lceil \frac{(-k+\frac{p+1}{2})\cdot 2N}{p} \right\rfloor_{2N}$. As a consequence, collisions never occur. Figure 3.3 illustrates this phenomenon.

So, by selecting odd values for the plaintext modulus, the negacyclicity problem is naturally neutralized.

We present in Definition 3.4.1 the formula for the accumulator in this case. Note that because N is a power of two and p an odd value, some rounding is required in the quotient $\frac{N}{p}$, but we omit it to keep notations lighter, (or equivalently, the division operator is assumed to include rounding).

Definition 3.4.1. (Odd-case Accumulator) Let p and p' be two integers, with p **odd**. Let $f : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$ a function, and N a power of two. Then, the accumulator $\mathsf{acc}(X) \in \mathbb{Z}_{N,q}[X]$ has

the form:

$$\mathsf{acc}(X)^{\text{odd-modulus}} = X^{-\frac{N}{2p}} \cdot \sum_{j=0}^{N/p-1} X^j \cdot \left(\sum_{i=0}^{\frac{p-1}{2}} f(i) X^{i\frac{2N}{p}} + \sum_{i=0}^{\frac{p-1}{2}-1} -f\left(i + \frac{p+1}{2}\right) X^{i\frac{2N}{p}+\frac{N}{p}} \right)$$

Remark on the encoding: With this construction, we do not need a bit of padding anymore. So, the values $f(i)$ are encoded using the simple encoding of Section 2.3, that is to say $\left\lfloor \frac{f(i)\cdot q}{p} \right\rceil$.

Let us explain the structure of this accumulator. The polynomial has degree N and is made of p distinct windows of width $\frac{N}{p}$. Each of these windows has constant coefficient value $f(k)$, for $k \in \{0, \ldots, p-1\}$. For $0 \leq \alpha \leq \frac{p-1}{2}$, the range of degrees whose coefficients are $f(\alpha)$ is $\left[\alpha\frac{2N}{p} - \frac{N}{2p} \; ; \; \alpha\frac{2N}{p} + \frac{N}{2p}\right]$. Now, for $\frac{p+1}{2} \leq \beta \leq p-1$, we can write $\beta = \alpha + \frac{p+1}{2}$, with $0 \leq \alpha < \frac{p-1}{2}$. This time, the range of spanned degrees is $\left[\alpha\frac{2N}{p} + \frac{N}{2p} \; ; \; (\alpha+1)\frac{2N}{p} - \frac{N}{2p}\right]$. Thus, the values $k \in \{0, \ldots, p-1\}$ spans the entire space $[0; N)$ without overlap. The values over $\frac{p+1}{2}$ gets negated by the negacyclicity, so the underlying coefficient is also negated to compensate this effect. We illustrate this construction on Figure 3.4.

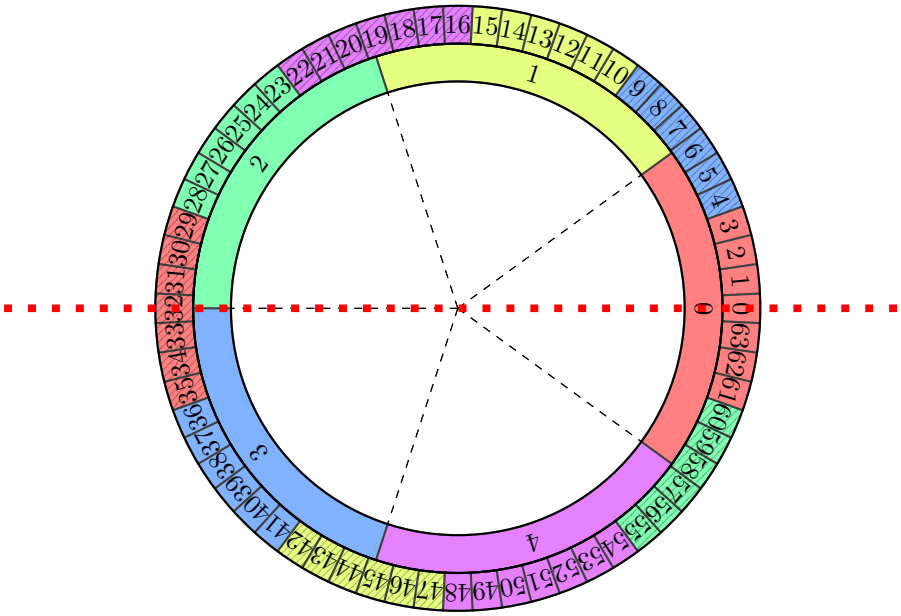


Figure 3.4: Illustration of the construction of the Odd-Modulus accumulator. On top is the ring $\mathbb{Z}_{2N}$ splitted in windows. Below is a representation of the polynomial v, with its version once rotated by a multiplication by X^N. On the figure, $p = 5$ and $N = 32$. Hatching shows the parts which have an additional minus signs

If we compare this approach to the bit-of-padding one, the windows have half the size. So we may think that the bound on the maximal noise to ensure correctness must be twice tighter. However, because the bit-of-padding accumulator makes use of only one half of the torus, the windows actually have the same width if we consider the same size of plaintext space. In Section 4.6.1, we elaborate further on how the parametrization of the scheme is handled in this case.

With this technique, no bit of padding is required. Consequently, it allows to use the linear homomorphisms without worrying about carry propagation. This is a key improvement that is at the root of the works presented in the rest of this thesis.

3.5 Conclusion

Our idea of employing odd plaintext moduli that we developed in Section 3.4 do not have any computational overhead with respect to the classical bit-of-padding technique of Section 3.2, while removing all the constraints the latter put on the homomorphic compilation. This is not the case of the other methods of the state of the art that we summed up in Section 3.3, which all increase the amount of computation

However, working with an odd modulus may seem unhandy: data types in programming are usually defined on a power-of-two modulus for example. But we show in the following chapters (Chapter 4, 5, 6 and 7) that this introduces new applications and homomorphic capabilities to TFHE.

Chapter 4

Accelerating the Evaluation of Boolean Circuits with p-encodings

TFHE being the most efficient scheme to handle data of small precision, it is a natural choice when it comes to evaluate homomorphically Boolean circuits. However, performances of the existing frameworks are still limited.

The most straightforward method, already introduced in the original TFHE paper [Chi+20] under the name *gate bootstrapping*, consists in running one bootstrapping for each bivariate Boolean gate of the circuit. As a consequence, the conversion of the original Boolean circuit in a homomorphic circuit handling encrypted bits is straightforward, moreover the noise growth is contained thanks to the systematic use of bootstrapping. However, this approach is very expensive due to the high numbers of bootstrappings and makes it quite suboptimal for large circuits.

In this chapter, we introduce a new framework to homomorphically evaluate Boolean functions on encrypted data efficiently, i.e. by reducing the amount of necessary bootstrappings. Our approach introduces an *intermediate homomorphic layer* which encodes the bits (elements of $\mathbb{Z}_2$) on a small ring $\mathbb{Z}_p$ before encrypting them. This allows us to evaluate Boolean functions with one cheap homomorphic sum followed by one bootstrapping. After formalizing the underlying concept of *p-encoding* and explaining our evaluation strategy, we investigate the issue of finding valid sets of encodings for a Boolean function. We characterize this problem and provide an exact constructive algorithm to solve it. We further provide a sieving heuristic finding solutions more efficiently but at the cost of loosing optimally. Since our method is only efficient for Boolean functions with limited number of inputs, we also propose a heuristic to decompose any Boolean circuit into Boolean functions which are efficiently evaluable using our approach. Finally, we apply our technique to various cryptographic primitives, namely the SIMON block cipher, the Trivium stream cipher, the Keccak permutation, the Ascon Substitution Box. A substitution table in a symmetric-key algorithm (S-box) and the AES S-box. Compared to previous works implementing the same primitives (for SIMON, Trivium and AES) our implementations achieve significant speedups.

After some technical preliminaries on Boolean circuits (Section 4.1), we introduce a new concept of *intermediate homomorphic layer* and explain how bits are encoded in Section 4.3 and the algorithms to construct it in Sections 4.4 and 4.5. Finally, we describe some implementation works in Section 4.6 and our experimental results in Section 4.7.

4.1 Preliminaries on Boolean Functions and Boolean Circuits

A Boolean function has the form $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$, with ℓ being called the *arity* of the function.

Definition 4.1.1. The Algebraic Normal Form (ANF) of a Boolean function $f : \{0,1\}^\ell \mapsto \{0,1\}$ is a polynomial expression in which each term corresponds to a specific input combination of n

variables. The ANF is defined as follows:

$$f(x_1, x_2, \ldots, x_l) = a_0 \oplus a_1 x_1 \oplus a_2 x_2 \oplus \ldots \oplus a_{2^n-1} x_1 x_2 \ldots x_l$$

$$\text{where: } a_0, a_1, a_2, \ldots, a_{2^\ell-1} \in \{0,1\} \quad \text{are the Boolean coefficients and}$$
$$x_1, x_2, \ldots, x_\ell \quad \text{are called the Boolean variables}$$

This result means that any Boolean function can be evaluated by the means of `AND` and `XOR` operations. In the following, we will focus on the implementation of Boolean circuits composed of these operations exclusively.

A Boolean function can be represented by its *truth table*, which is simply a table gathering all the possible inputs and the corresponding result of the application by the function. It can also be represented with a Boolean formula. A third representation is the *Boolean circuit*:

Definition 4.1.2. A Boolean circuit associated to the Boolean function f is a finite Directed Acyclic Graph whose edges are *wires* and vertices are *Boolean gates* representing Boolean operations. We consider `AND` gates and `XOR` gates, of fan-in 2 and fan-out 1. We also consider copy gates, of fan-in 1 and fan-out > 1, that outputs several copies of its input. A circuit is further formally composed of input gates of fan-in 0 and fan-out 1, and output gates of fan-in 1 and fan-out 0.

Evaluating a ℓ-input m-output circuit consists in writing an input $\mathbf{x} \in \mathbb{B}^\ell$ in the input gates, processing the gates from input gates to output gates, then reading the outputs from the output gates.

This notion of Boolean circuit will be particularly useful in Section 4.5.

4.2 State of the Art on Homomorphic Boolean Computations

Let $f : \mathbb{B}^\ell \mapsto \mathbb{B}^{\ell'}$ be a Boolean function. There are two ways of handling its computations, and thus designing its homomorphic version

- **As a multivariate table:** The programmable bootstrapping of TFHE can be seen as a homomorphic lookup table evaluation. So we can imagine evaluating the function all at once with a unique (and potentially large) bootstrapping..
- **As a Boolean circuit:** If we look at f as a Boolean circuit (see Definition 4.1.2), then it is sufficient to design a homomorphic version of each Boolean gate. Then, it is quite easy to concatenate the blocks to evaluate any function we want, not only f

While both techniques seems straightforward, there are not a viable solution in practice. This section attemps to explain why, and presents the state of the art for the two strategy: evaluating f as a LUT or as a Boolean circuit:

As a LUT: The algorithm of TFHE's programmable bootstrapping takes only one ciphertext in input, so one may think that it can be programmed only with univariate functions. However, there is an easy workaround allowing to evaluate multivariate functions. To evaluate a ℓ-bit LUT, you pick $p = 2^\ell$ for the plaintext modulus. Each input bit is encoded in the Least Significant Bit (LSB) of one value in $\mathbb{Z}_{2^\ell}$, and encrypted in a separate ciphertext. Now, to evaluate the function f, the i-th input is shifted into the i-th least significant bit by a multiplication by 2^i. They are then all summed together to encode the value $b_{\ell-1}b_{\ell-2}\ldots b_1 b_0$ and the LUT can be processed by a bootstrapping on ℓ bits.

This approach does not scale well when the number of inputs ℓ increases. Indeed, working in a larger plaintext space slows terribly the PBS algorithm. On the other hand, using a circuit representation ensures a constant time for the bootstrapping algorithm no matter the number of inputs.

As a circuit: As we introduced in Section 4.1, any Boolean function can be easily compiled into a Boolean circuit made of `XOR` and `AND` gates.

If we naturally pick the plaintext modulus $p = 2$, `XOR` operations are evaluable with the homomorphic sum of TFHE, which are computationally free (at the cost of a slight noise growth). However, evaluating the non-linear gate `AND` requires bootstrapping. As we explained in the previous paragraph, bootstrapping is a univariate operation but it is possible to turn it into a bivariate one by using two bits of plaintext instead of one (so $p = 4$). But this is not the end of it ! Because 4 is even, the negacyclicity problem applies and we need a third bit to work as a padding bit, so working with $p = 8$.

The problem now is that XOR can no longer be evaluated with a simple sum, because carries would propagate into the MSB and ruin the correctness. So one would need *one bootstrapping per Boolean gate* to evaluate the circuit while keeping the MSB clean. This is the solution implemented in `tfhe-rs` library [Zam22c][1].

[Chi+21] proposes a different approach: by leveraging a newer version of the TFHE scheme supporting multiplications of LWE ciphertexts, Boolean circuits are evaluated with homomorphic sums for `XOR` gates and this new multiplication operation for `AND` gates. While this approach is clearly a progress from the vanilla framework, we note that a few bootstrappings are still required to control the noise growth and that this new operation of LWE multiplications remains costly both in terms of performances and in terms of noise. Thus, we choose to stick to the first version of the TFHE scheme to keep the framework lighter and we tackle the performance issues of [Chi+20] with a different approach than the one of [Chi+21].

We have just seen that both approaches do not scale well for different reasons: LUT because of bootstrapping complexity and circuit because of the number of bootstrapping occurences. Our approach attempts to gain the best of both worlds: we construct a constant-time bootstrapping for any number of inputs (solving the LUT problem), while using only one bootstrapping to evaluate the entire function (solving the circuit problem).

4.3 Boolean Encoding over $\mathbb{Z}_p$ and Homomorphic Evaluation Strategy Between $\mathbb{B}$ and $\mathbb{Z}_p$

We propose a construction in which we embed Boolean values in $\mathbb{Z}_p$ for well-chosen values of p, forming an "intermediate homomorphic layer" between $\mathbb{B}$ (the plaintext space of bits) and $\mathbb{Z}_q$ (the ciphertext space). In the following, we explain how we define such a layer, and we describe our new strategy to evaluate Boolean functions in a more efficient way without considering the circuit representation of the function. Note that we generalize this construction to arithmetic spaces in Chapter 5.

4.3.1 Encoding of $\mathbb{B}$ over $\mathbb{Z}_p$

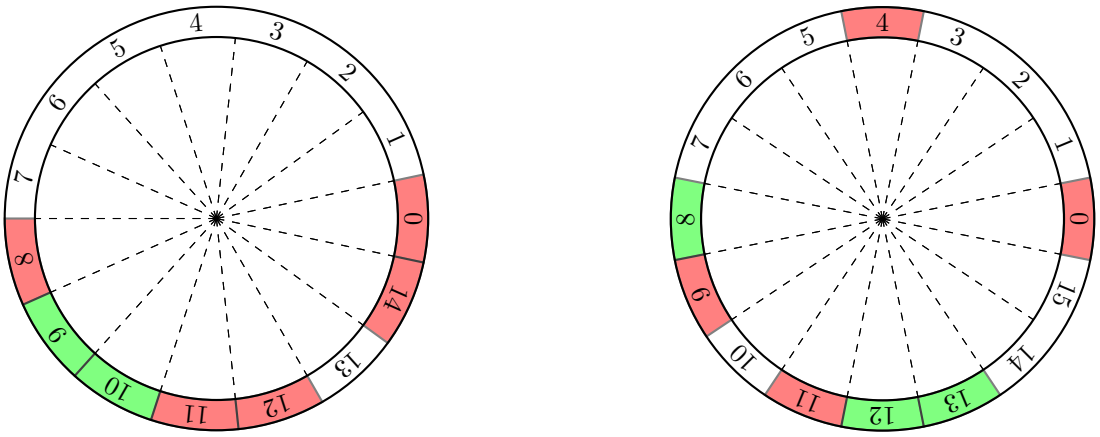
To represent Boolean values over $\mathbb{Z}_p$, we use a mapping function that we call a *p-encoding*:

Definition 4.3.1 (p-encoding)**.** A *p-encoding* is a function $\mathcal{E} : \mathbb{B} \mapsto 2^{\mathbb{Z}_p}$ that maps the Boolean space to a subset of the discretized torus. A p-encoding is *valid* if and only if:

$$\begin{cases} \mathcal{E}(0) \cap \mathcal{E}(1) = \emptyset \quad \text{and} \\ \text{if } p \text{ is even: } \forall\, x \in \mathbb{Z}_p, \forall\, b \in \mathbb{B} : x \in \mathcal{E}(b) \iff \left[x + \frac{p}{2}\right]_p \notin \mathcal{E}(b) \end{cases} \tag{4.1}$$

We call this last property *relaxed negacyclicity*.

[1] more precisely, in the `boolean` crate.



Example with $p = 15$ *Example with $p = 16$*

Figure 4.1: Representation of two valid p-encodings. The green part represents $\mathcal{E}(1)$, and the red part $\mathcal{E}(0)$. Note that even if p is even on the right-hand figure, the relaxed negacyclity is still respected.

In our approach when we need to encrypt a bit, we apply a p-encoding to embed it in $\mathbb{Z}_p$, then we encrypt the result using the classical setup of TFHE. When new values are freshly encrypted or produced by a PBS, only one element of $\mathbb{Z}_p$ is chosen for each bit. We call such an encoding a canonical p-encoding:

Definition 4.3.2 (Canonical Encoding)**.** A p-encoding $\mathcal{E}$ is said *canonical* if and only if it is valid and $|\mathcal{E}(0)| = |\mathcal{E}(1)| = 1$

Let c be a ciphertext encoding a bit b under a p-encoding $\mathcal{E}$, where $\mathcal{E}$ is an arbitrary valid encoding: its associated subsets can be any subset of $\mathbb{Z}_p$ as long as the validity requirements of (4.1) are fulfilled. One can transform the ciphertext $\mathbf{c}$ into another ciphertext $\mathbf{c}'$ encoded under any *canonical* p-encoding, possibly under a different p, by simply performing a PBS.

Property 4.3.1 (Reduction to a canonical encoding)**.** *Let $\mathcal{E}$ be a valid p-encoding and $\mathcal{E}'$ a canonical p'-encoding. We denote $\alpha' = \mathcal{E}'(0)$ and $\beta' = \mathcal{E}'(1)$. Let $\mathbf{c}$ be a ciphertext encrypting a bit b under $\mathcal{E}$. Then, one can produce a ciphertext $\mathbf{c}'$ encrypting the same bit b under $\mathcal{E}'$ by applying a PBS on $\mathbf{c}$. This PBS performs the function :*

$$\texttt{Cast}_{\mathcal{E}\mapsto\mathcal{E}'} : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'}$$
$$x \mapsto \begin{cases} \alpha' & \text{if } x \in \mathcal{E}(0) \\ \beta' & \text{if } x \in \mathcal{E}(1) \\ \perp & \text{otherwise.} \end{cases}$$

Here, $\perp$ simply denotes a placeholder value for a state that cannot be reached.

Our goal is to represent the Boolean function we want to evaluate with a sum of p-encodings (we define what we mean by "sum of p-encoding" in Section 4.3.2). When sums are carried out on ciphertexts (and so homomorphically on the underlying p-encodings), the sets $\mathcal{E}(0)$ and $\mathcal{E}(1)$ of the p-encodings may move, grow, shrink, but they should never overlap as it would result in a loss of information. As we removed the need for a bit of padding, we do not need to track a potential overflow of data (informally we say that ciphertexts are free to "loop around the torus"). After the sum, the encoding of the result can be reset to a canonical one with a PBS to allow further computation. Or, if the homomorphic computation is over, the result can be recovered by decrypting the ciphertext and checking in which partition the decrypted value lies.

The next subsection explains in further details the process of evaluating Boolean functions on with p-encodings.

4.3.2 A New Strategy for Homomorphic Boolean Evaluation

In the following, we consider two Boolean variables x and y and their two respective encodings over $\mathbb{Z}_p$:

$$\mathcal{E}_x = \begin{cases} 0 \mapsto \{\alpha_i\}_{0\leq i\leq l_0} \\ 1 \mapsto \{\beta_i\}_{0\leq i\leq l_1} \end{cases} \quad \text{and } \mathcal{E}_y = \begin{cases} 0 \mapsto \{\alpha'_i\}_{0\leq i\leq l'_0} \\ 1 \mapsto \{\beta'_i\}_{0\leq i\leq l'_1} \end{cases} \tag{4.2}$$

Let f be a bivariate Boolean function and let us construct two sets $\mathcal{P}_0$ and $\mathcal{P}_1$ such that:

$$\mathcal{P}_b = \{[\gamma+\delta]_p \mid (\gamma,\delta) \in \mathcal{E}_x(b_x) \times \mathcal{E}_y(b_y) \text{ and } f(b_x,b_y) = b \text{ with } (b_x,b_y) \in \mathbb{B}^2\} \ \forall\ b \in \mathbb{B}. \tag{4.3}$$

We say that the sum of p-encodings $\mathcal{E}_x + \mathcal{E}_y$ is *suitable for the evaluation of* f if and only if $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$. The definition can be generalized to any number of arguments ℓ for f. For a given f, finding such correct encodings is not trivial. We elaborate further on this point in Section 4.4.

If $\mathcal{E}_x$ and $\mathcal{E}_y$ are suitable for f, then one can use the computed sets $\mathcal{P}_0$ and $\mathcal{P}_1$ to construct a new p-encoding

$$\mathcal{E}_z = \begin{cases} 0 \mapsto \mathcal{P}_0 \\ 1 \mapsto \mathcal{P}_1 \end{cases}$$

that encodes the bit $f(x,y)$. As $\mathcal{E}_z$ is valid, then the clear value of the bit can be recovered by decryption, or further computations can be performed without the need of a bootstrapping.

Definition 4.3.3 (Application of a function to a vector of encodings)**.** Let $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ be a Boolean function and let $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$ be a vector of p-encodings. We define $f(\mathcal{E})$ by:

$$f(\mathcal{E}) = \begin{cases} 0 \mapsto \mathcal{P}_0 \\ 1 \mapsto \mathcal{P}_1 \end{cases}$$

with:

$$\mathcal{P}_b = \left\{ \left[\sum_{i=1}^{l} \gamma_i\right]_p \mid (\gamma_1,\ldots,\gamma_l) \in \prod_{i=1}^{\ell} \mathcal{E}_i(b_i) \text{ and } f(b_1,\ldots,b_l) = b \right\} \forall b \in \mathbb{B}$$

$f(\mathcal{E})$ always exists, but is valid only if it respects the constraints of Definition 4.1.

Let us illustrate the latter definition on two toy example. We consider the two Boolean operators & and $\oplus$. The p-encoding resulting of the function $f : (x,y) \mapsto x \ \& \ y$ is:

$$\mathcal{E}_{\&} = \begin{cases} 0 \mapsto \{\alpha_i + \alpha'_j\}_{\substack{0\leq i\leq l_0 \\ 0\leq j\leq l'_0}} \cup \{\alpha_i + \beta'_j\}_{\substack{0\leq i\leq l_0 \\ 0\leq j\leq l'_1}} \cup \{\alpha'_i + \beta_j\}_{\substack{0\leq i\leq l'_0 \\ 0\leq j\leq l_1}} \\ 1 \mapsto \{\beta_i + \beta'_j\}_{\substack{0\leq i\leq l_1 \\ 0\leq j\leq l'_1}} \end{cases} \tag{4.4}$$

and the p-encoding resulting of the operation $f : (x,y) \mapsto x \oplus y$ is:

$$\mathcal{E}_{\oplus} = \begin{cases} 0 \mapsto \{\alpha_i + \alpha'_j\}_{\substack{0\leq i\leq l_0 \\ 0\leq j\leq l'_0}} \cup \{\beta_i + \beta'_j\}_{\substack{0\leq i\leq l'_0 \\ 0\leq j\leq l'_1}} \\ 1 \mapsto \{\alpha_i + \beta'_j\}_{\substack{0\leq i\leq l_0 \\ 0\leq j\leq l'_1}} \cup \{\alpha'_i + \beta_j\}_{\substack{0\leq i\leq l'_0 \\ 0\leq j\leq l_1}} \end{cases} \tag{4.5}$$

Figure 4.2 further illustrates this construction for these two operations.

To wrap up, here is our proposed framework to evaluate a Boolean function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ given a vector of suitable p-encodings $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$:

1. Encrypt each input b_i with some canonical p-encoding $\mathcal{E}_i$ into a ciphertext c_i such that $\mathcal{E}_{sum} = f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ is a valid encoding.

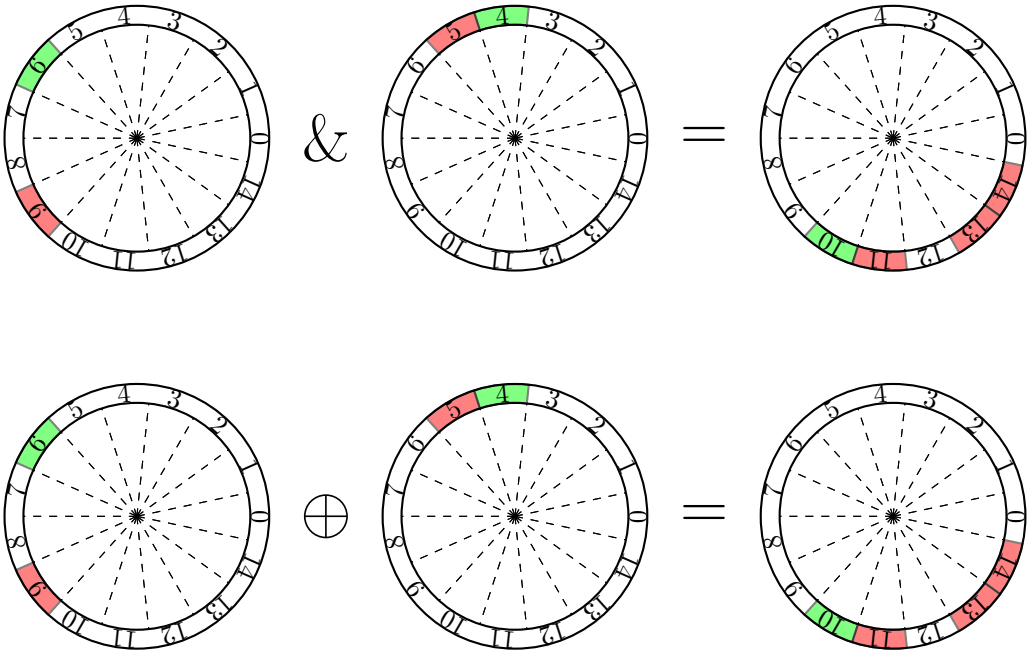


Figure 4.2: Starting from two canonical encodings, we produce two new p-encodings corresponding to the results of the `AND` and the `XOR` operations.

2. For a Boolean function f to be evaluated on $b_1, \ldots, b_\ell$, compute homomorphically the sum of the ciphertexts $\mathbf{c} = \mathbf{c}_1 + \cdots + \mathbf{c}_\ell$. This yields an encryption of $b = f(b_1, \ldots, b_\ell)$, encoded with a valid p-encoding $\mathcal{E}_{sum} = f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$.

3. (a) If the result is directly required by the client, $\mathbf{c}$ is returned as ciphertext which can be decrypted to get the result in $\mathbb{Z}_p$ and associated to the right Boolean value.

   (b) If the result is reused in further homomorphic computations, a PBS calculating $\texttt{Cast}_{\mathcal{E}_{sum} \mapsto \mathcal{E}_{new}}$ on the result is computed (like introduced in Property 4.3.1), with $\mathcal{E}_{new}$ a new canonical p-encoding. The resulting value can then be used as an input for a next Boolean function.

Let us formalize this process by defining the notion of *gadget associated to a Boolean function* f :

Definition 4.3.4 (Gadget)**.** Let f be a Boolean function of arity ℓ. A gadget associated to f is an homomorphic operator defined by a tuple $\Gamma = \left(\mathcal{E}_{\mathbf{in}} = (\mathcal{E}_{in}^{(1)}, \ldots, \mathcal{E}_{in}^{(\ell)}), \mathcal{E}_{out}, p_{in}, p_{out}\right)$ such that:

- All the elements of $\mathcal{E}_{\mathbf{in}}$ are p_{in}-encodings, and $\mathcal{E}_{out}$ is a canonical p_{out}-encoding.
- The encoding $\mathcal{E}_{sum} = f(\mathcal{E}_{in}^{(1)}, \ldots, \mathcal{E}_{in}^{(\ell)})$ is a valid encoding.

Applying a gadget to ciphertexts $\mathbf{c}_1, \ldots, \mathbf{c}_\ell$, that encrypt the bits $b_1, \ldots, b_\ell$, produces a new ciphertext c' encrypting the bit $f(b_1, \ldots, b_\ell)$ under the p_{out}-encoding $\mathcal{E}_{out}$. To do so, we perform the following algorithm:

- Constructing an intermediate ciphertext $\mathbf{c}_{inter} = \sum_{i=1}^{\ell} \mathbf{c}_i$ using the homomorphic sum of TFHE. This ciphertext encrypts $f(b_1, \ldots, b_\ell)$ under the p_{in}-encoding $f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$.
- Reducing the encoding of $\mathbf{c}_{inter}$ from $\mathcal{E}_{inter}$ to $\mathcal{E}_{out}$ by applying a PBS on $\mathbf{c}_{inter}$ performing the function $\texttt{Cast}_{\mathcal{E}_{inter} \mapsto \mathcal{E}_{out}}$. This produces the expected result $\mathbf{c}'$.

The advantage of this construction is that only one PBS is performed to apply the function. Moreover, depending on the function, the input size of the PBS lookup table might be much smaller than the arity of the function. Gadgets can be seen as a way to compress several Boolean operators into a single evaluation of univariate look-up table. Of course, for a given p_{in} and a given f, such a gadget may not exist. In such a case, two solutions can be considered:

- Increasing the value of p_{in} (e.g. taking $p_{in} \geq 2^{\ell}$ always works, but is very inefficient).
- Splitting the function into a graph of subfunctions, and evaluating each one with a gadget.

The question of constructing valid gadgets for a given f is treated in Section 4.4. The question of efficiently splitting a function is treated in Section 4.5.

Example: We illustrate our approach with a simple working example: let f be a basic multiplexing function, such that

$$f(a,b,c) = \begin{cases} a \text{ if } c = 1 \\ b \text{ if } c = 0 \end{cases}$$

Instead of leveraging its Boolean representation $f(a,b,c) = a\&c \oplus b\&\bar{c}$, which would cost 3 PBS with the approach of [Chi+20], our strategy consists in constructing a gadget and apply it to the inputs a, b and c, which takes only one PBS. Here is the step-by-step procedure:

1. Encrypting the bits with the 7-encodings:

$$\mathcal{E}_a = \mathcal{E}_b = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{and } \mathcal{E}_c = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{2\} \end{cases}$$

.

2. Applying the function f on the 7-encodings by summing the ciphertexts, producing a valid 7-encoding:

$$\mathcal{E}_{sum} = \begin{cases} 0 \mapsto \{0,1,2,5\} \\ 1 \mapsto \{3,4,6\} \end{cases}$$

3. With one PBS, resetting the result to a target canonical p-encoding (with any p), for example

$$\mathcal{E}_{new} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{with } p = 7$$

A visualization of this procedure can be found in Figure 4.3. We just defined the gadget $\Gamma = ((\mathcal{E}_a, \mathcal{E}_b, \mathcal{E}_c), \mathcal{E}_{new}, 7, 7)$.

4.3.3 Encoding Switching

To apply a gadget to a given ciphertext, it has to be encrypted under the right encoding. Thus, we need a method to homomorphically switch the encoding of a ciphertext. This allows as well to plug the output of any gadget on the input of any other one, and so to evaluate a chain of gadgets as long as we want. In the following, we explore different possibilities of encoding switching. Let us begin with some trivial cases:

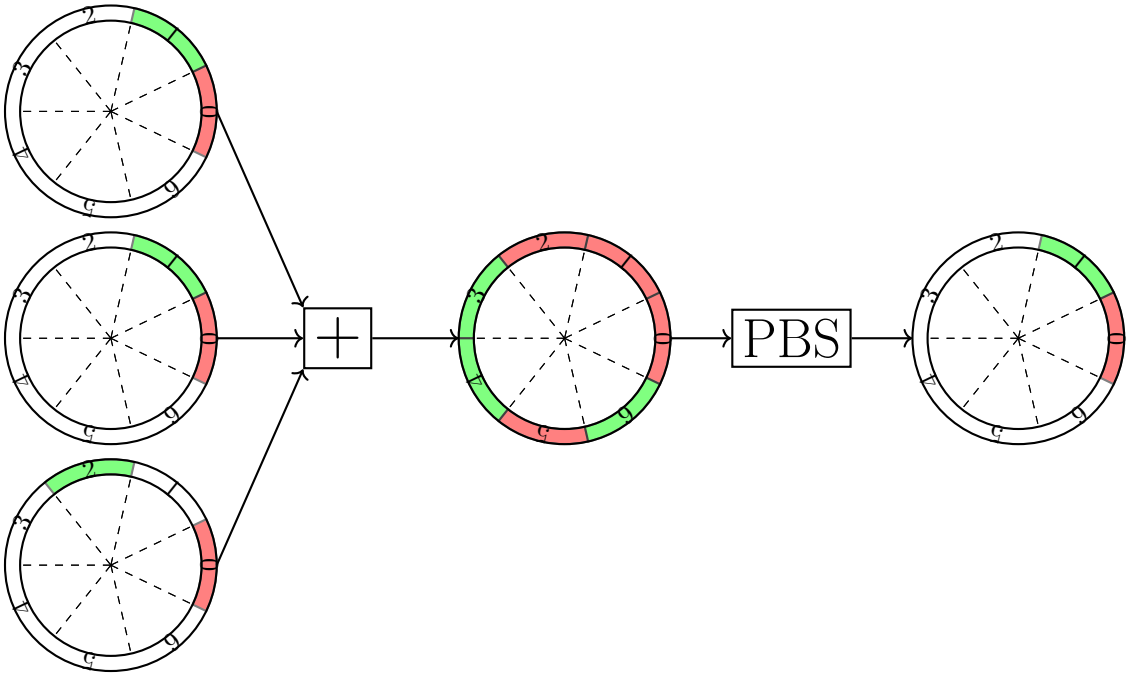


Figure 4.3: Illustration of an execution of the framework for the multiplexing function.

Property 4.3.2 (Encoding switching with a sum by a constant)**.** *Let* $\mathbf{c}$ *be a ciphertext encoded under* $\mathcal{E} = \begin{cases} 0 \mapsto \{\alpha_i\}_{0 \leq i \leq l_0} \\ 1 \mapsto \{\beta_i\}_{0 \leq i \leq l_1} \end{cases}$ *and* $a \in \mathbb{Z}_p$ *a constant. The encoding of* $\mathbf{c}$ *can be switched to:*

$$\mathcal{E}' = \begin{cases} 0 \mapsto \{[\alpha_i + a]_p\}_{0 \leq i \leq l_0} \\ 1 \mapsto \{[\beta_i + a]_p\}_{0 \leq i \leq l_1} \end{cases}$$

by an homomorphic addition of the ciphertext x *and the clear value* a*.*

Proof. All the elements of $\mathcal{E}'(0)$ (resp. $\mathcal{E}'(1)$) are offset by exactly a from their counterparts in $\mathcal{E}(0)$ (resp. $\mathcal{E}(1)$). Thus, if the original encoding $\mathcal{E}$ was valid, then $\mathcal{E}(0) \cap \mathcal{E}(1) = \emptyset$. So we trivially get $\mathcal{E}'(0) \cap \mathcal{E}'(1) = \emptyset$ and thus the validity of $\mathcal{E}'$. □

Property 4.3.3 (Encoding switching with multiplication by a constant)**.** *Let* $\mathbf{c}$ *be a ciphertext encoded under* $\mathcal{E} = \begin{cases} 0 \mapsto \{\alpha_i\}_{0 \leq i \leq l_0} \\ 1 \mapsto \{\beta_i\}_{0 \leq i \leq l_1} \end{cases}$ *and* $a \in \mathbb{Z}_p$ *a constant value coprime with* p*. The encoding of* $\mathbf{c}$ *can be switched to:*

$$\mathcal{E}' = \begin{cases} 0 \mapsto \{[a \cdot \alpha_i]_p\}_{0 \leq i \leq l_0} \\ 1 \mapsto \{[a \cdot \beta_i]_p\}_{0 \leq i \leq l_1} \end{cases}$$

by an homomorphic multiplication of the ciphertext $\mathbf{c}$ *by the clear value* a*.*

Proof. As a is coprime with p, the multiplication by a is a bijection from $\mathbb{Z}_p$ to $\mathbb{Z}_p$. By definition, all the α_i's are different of the β_i's. If we apply a bijection on them, the inequalities are conserved. □

Note that the condition of coprimality between a and p is a sufficient condition for the multiplication to be a valid encoding switching, but is not necessary. In particular, one other case is particularly useful in practice:

Property 4.3.4 (Encoding switching for a canonical encoding containing a zero)**.** *Let* $\mathbf{c}$ *be a ciphertext encoded under the p-encoding:* $\mathcal{E} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ *and let* $a \in \mathbb{Z}_p \setminus \{0\}$*. Then, it can be switched to:* $\mathcal{E}' = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{a\} \end{cases}$ *by a simple homomorphic multiplication between the ciphertext* $\mathbf{c}$ *and the clear value* a*. This holds as well if* $\mathcal{E}(0)$ *and* $\mathcal{E}(1)$ *swapped.*

Proof. The property is trivial by the linear homomorphism of the TFHE scheme. □

These techniques are powerful because they do not require any bootstrapping, so they can be considered as free in terms of performances. However, any valid p-encoding can be turned into any other one with a programmable bootstrapping, even with a different modulus p. A reduced version of this is given by Property 4.3.1, but it can be extended to any valid output p-encoding.

Property 4.3.5 (Arbitrary encoding switching with a PBS)**.** *Let* $\mathbf{c}$ *be a ciphertext encoded under* $\mathcal{E}$*. Its encoding can be switched to* $\mathcal{E}'$ *(even with a different modulus* p'*) by applying a PBS on* $\mathbf{c}$ *evaluating the function*

$$\mathtt{Cast}_{\mathcal{E} \mapsto \mathcal{E}'} : \mathbb{Z}_p \mapsto \mathbb{Z}_{p'} \tag{4.6}$$

$$x \mapsto \begin{cases} \alpha' \in \mathcal{E}'(0) & \text{if } x \in \mathcal{E}(0) \\ \beta' \in \mathcal{E}'(1) & \text{if } x \in \mathcal{E}(1) \\ \bot & \text{otherwise.} \end{cases} \tag{4.7}$$

$\bot$ *simply denotes an arbitrary placeholder value, as it will never be reached.*

Note for $p = 2$: The case $p = 2$ is particular: we can observe that valid 2-encodings are automatically negacyclic. Moreover, they allow to evaluate the `XOR` operation by simply performing an homomorphic sum (so without bootstrapping). So it might be efficient to switch between 2-encodings for `XOR` operations and p-encodings (with odd p) for non-linear Boolean functions. We make use of this strategy in our implementation of the Keccak permutation in Section 4.7.3 and for the AES in Section 4.7.5.

4.4 Algorithms of Construction of Gadgets

Let $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ a Boolean function with ℓ entries. This section addresses the problem of constructing a gadget for f. To do so, we pick a value for p and we search a vector of ℓ p-encodings $\mathcal{E}_{\mathbf{in}}$ suitable for f.

4.4.1 Reduction of the Search Space

While exhaustive search is a first option, it quickly becomes impractical due to the explosion of the number of possibilities as p grows. As a consequence, a reduction of the search space is needed without leaving out a potential solution.

We introduce two lemmas that will be used to reduce the search space:

Lemma 4.4.1 (Reducibility to singletons)**.** *Let* $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ *and let* $(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ *be a vector of p-encodings suitable for* f *and having the form:* $\forall\, i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x_j^{(i)}\}_{1 \le j \le l_0^{(i)}} \\ 1 \mapsto \{y_j^{(i)}\}_{1 \le j \le l_1^{(i)}} \end{cases}$*. Then*

any vector of canonical p-encodings $(\mathcal{E}'_1, \ldots, \mathcal{E}'_\ell)$ *of the form:* $\forall\, i \in \{1, \ldots, \ell\}, \mathcal{E}'_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$ *with* $x^{(i)} \in \mathcal{E}_i(0)$ *and* $y^{(i)} \in \mathcal{E}_i(1)$ *is suitable for the function* f *as well.*

Proof. Let us assume that the vector $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ of Lemma 4.4.1 is suitable for the function f. Then the sets $\mathcal{P}_0$ and $\mathcal{P}_1$ constructed like in Equation 4.3 are disjoint. Now, let us consider the vector of canonical p-encodings $\mathcal{E}' = (\mathcal{E}'_1, \ldots, \mathcal{E}'_\ell)$ respecting the property:

$$\forall\ b\ \in \mathbb{B}, \forall i \in \{1, \ldots, \ell\}, \mathcal{E}'_i(b) \subset \mathcal{E}_i(b).$$

As a consequence, if we build the sets $\mathcal{P}'_0$ and $\mathcal{P}'_1$ relative to the encodings $\mathcal{E}'$, then we naturally get $\mathcal{P}'_0 \subset \mathcal{P}_0$ and $\mathcal{P}'_1 \subset \mathcal{P}_1$. So we get $\mathcal{P}'_0 \cap \mathcal{P}'_1 = \emptyset$, proving Lemma 4.4.1. □

Lemma 4.4.2 (Reducibility to the singleton zero)**.** *Let* $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ *and let* $(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ *be a vector of p-encodings suitable for* f *and of the form:* $\forall i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$ *Then any vector of canonical p-encodings* $(\mathcal{E}'_1, \ldots, \mathcal{E}'_\ell)$ *of the form:* $\forall i \in \{1, \ldots, \ell\}, \mathcal{E}'_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{y^{(i)} - x^{(i)}\} \end{cases}$ *is suitable for the function* f *as well.*

Proof. Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ be a function and $\mathcal{E}$ be a vector of canonical p-encodings $(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ suitable for f with:

$$\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases} .$$

Let us build the sets $\mathcal{P}_0$ and $\mathcal{P}_1$ according to Equation 4.3. Each element of these sets is the sum of exactly one element of each p-encoding, that is to say an element $\mathcal{E}_i(0) \cup \mathcal{E}_i(1)$.

Let us pick an indice $k \in \{1, \ldots, \ell\}$, a value $a \in \mathbb{Z}_p$ and replace $\mathcal{E}_k$ in the vector $\mathcal{E}$ by:

$$\mathcal{E}'_k = \begin{cases} 0 \mapsto \{x^{(i)} - a\} \\ 1 \mapsto \{y^{(i)} - a\} \end{cases}$$

By using the Property 4.3.2, we directly have $\mathcal{P}'_0 \cap \mathcal{P}'_1 = \emptyset$ from $\mathcal{P}_0 \cap \mathcal{P}_1 = \emptyset$ (by suitability of the encodings for f).

By iterating this procedure on each of the ℓ elements of $\mathcal{E}$, and by picking each time $a = -x^{(i)}$, we prove Lemma 4.4.2. □

Using both Lemmas 4.4.1 and 4.4.2, we can restrict the search to the encodings of the form

$$\mathcal{E}_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$$

with $d_i \neq 0$ without any loss of generality.

Moreover, we restrict the solution further: we only consider p-encodings with p *odd* and *prime*. The choice of an odd p allows to free ourselves from the negacyclicity constraint. To explain the constraint of primality, we introduce the following lemma, that allows to drastically improve the performances of the search:

Lemma 4.4.3. *Let* p *be a prime and* $f : \mathbb{B} \longrightarrow \mathbb{B}$ *be a Boolean function and let* $\mathcal{E} = (\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ *be p-encodings suitable for* f *with:* $\forall\ i \in \{1, \ldots, \ell\}, \mathcal{E}_i = \begin{cases} 0 \mapsto \{x^{(i)}\} \\ 1 \mapsto \{y^{(i)}\} \end{cases}$ *. For every* $a \in \mathbb{Z}_p \setminus \{0\}$*, the vector of p-encodings* $\mathcal{E}' = (\mathcal{E}'_1, \ldots, \mathcal{E}'_\ell)$ *with:* $\mathcal{E}'_i = \begin{cases} 0 \mapsto \{[a \cdot x^{(i)}]_p\} \\ 1 \mapsto \{[a \cdot y^{(i)}]_p\} \end{cases}$ *is suitable for* f *as well.*

Proof. This is an immediate consequence of Property 4.3.3. □

As a consequence, if p is prime (which we shall always choose in practice), any solution can be turned into a solution with $d_1 = 1$ by simply multiplying all the p-encodings of the solution by $[d_1^{-1}]_p$. So we can fix $d_1 = 1$ without any loss of generality, reducing drastically the size of the search space.

4.4.2 Formalization of the Search Problem

According to the lemmas from Section 4.4.1, we can reduce the problem of finding a vector of p-encodings $(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ such that $f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ is valid to the problem of finding a vector $\mathbf{d} = (d_1, \ldots, d_\ell)$ such that $d_1 = 1$, $\mathcal{E}_i = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$ and $f(\mathcal{E}_1, \ldots, \mathcal{E}_\ell)$ is valid. In the following, we describe an algorithm to find such a vector $\mathbf{d}$.

We denote $\mathcal{V}$ the matrix of elements of $\mathbb{B}$ of shape $2^\ell \times \ell$ gathering all the possible sequences of entries for the function f:

$$\mathcal{V} = \begin{pmatrix} 0 & \ldots & 0 & 0 \\ 0 & \ldots & 0 & 1 \\ 0 & \ldots & 1 & 0 \\ \vdots & \ddots & \vdots & \vdots \\ 1 & \ldots & 1 & 1 \end{pmatrix}$$

Also, we denote by $\mathbf{b}$ the vector of all the outputs of the function f, sorted in same order as the rows of $\mathcal{V}$. Thus, we have: $\forall i \in \{1, \ldots, 2^\ell\}, b_i = f(\mathcal{V}_i)$ for $\mathcal{V}_i$ the i-th row of $\mathcal{V}$. Let us define the vector $\mathbf{r}$ as: $\mathbf{r} = \mathcal{V}\mathbf{d}$. To make $\mathbf{d}$ a solution of the problem, $\mathbf{r}$ has to verify the following property:

$$\forall (i, j) \in \{1, \ldots, 2^\ell\}, f(\mathcal{V}_i) \neq f(\mathcal{V}_j) \implies r_i \neq r_j$$

An alternative formulation is: we look for two disjoint subsets $\mathcal{P}_0$ and $\mathcal{P}_1$ of $\mathbb{Z}_p$, such that: $f(\mathcal{V}_i) = b \iff r_i \in \mathcal{P}_b$.

The following section describes an algorithm finding a solution to this problem.

4.4.3 Algorithm

We start by constructing two sets $\mathcal{F}$ and $\mathcal{T}$ such that:

$$\mathcal{F} = \{\mathcal{V}_i \mid b_i = 0\} \text{ and } \mathcal{T} = \{\mathcal{V}_i \mid b_i = 1\}.$$

Each line $\mathcal{V}_i$ represents a linear combination of the d_j's, that verifies:

$$r_i = \sum_{j=0}^{\ell} \mathcal{V}_{ij} \cdot d_j \mod p.$$

The values r_i produced by the elements of $\mathcal{F}$ must be different from the ones produced by $\mathcal{T}$. As a consequence, we can write:

$$\forall\ (\mathcal{V}_i, \mathcal{V}_j) \in \mathcal{F} \times \mathcal{T}, \sum_{k=0}^{\ell} \mathcal{V}_{ik} \cdot d_k \neq \sum_{k=0}^{\ell} \mathcal{V}_{jk} \cdot d_k,$$

which is equivalent to writing:

$$\forall\ (\mathcal{V}_i, \mathcal{V}_j) \in \mathcal{F} \times \mathcal{T}, \sum_{k=0}^{\ell} (\mathcal{V}_{ik} - \mathcal{V}_{jk}) \cdot d_k \neq 0.$$

So we can rewrite our constraints in the set $\mathcal{C} = \{\mathcal{V}_i - \mathcal{V}_j \mid (\mathcal{V}_i, \mathcal{V}_j) \in \mathcal{F} \times \mathcal{T}\}$. $\mathcal{C}$ contains vectors with coordinates in $\{0, 1, -1\}$ representing linear combinations that have to be non-zero. Note that if an element of the set $\mathcal{C}$ is the opposite of an other, it does not bring further constraint and can thus be safely discarded from the set.

The use of a set in the implementation at this point of the algorithm allows to remove a lot of duplicate constraints and to simplify the next step. Then, the problem reduces to solving a "linear system of inequalities" in the ring $\mathbb{Z}_p$:

$$\begin{cases} c_1^{(1)} \cdot d_1 + \cdots + c_l^{(1)} \cdot d_\ell \neq 0 \mod p \\ c_1^{(2)} \cdot d_1 + \cdots + c_l^{(2)} \cdot d_\ell \neq 0 \mod p \qquad \text{with } c_i^{(j)} \in \{0, \pm 1\} \\ \vdots \end{cases}$$

After filtering the duplicates, we pack all the elements of $\mathcal{C}$ in ℓ matrices $\{\mathcal{C}_i\}_{1 \leq i \leq \ell}$ (each row being a linear combination), where the matrix $\mathcal{C}_i$ packs all the constraints involving only the i first inputs (i.e. all the coefficients of column index greater than i are zeros).

We then perform a recursive search (Algorithm 6), affecting at each step of depth i a possible value d_i for the i-th input. To do so, we call Algorithm 7 to construct the set of all possible values complying with the constraints of the matrix $\mathcal{C}_i$ and the previously set values for the preceding inputs. If we reach a dead-end, we backtrack by deleting the previous input and assigning it the next possible value. Algorithms 6 and 7 formalize this idea: Algorithm 6 is a basic recursive backtracking algorithm using calls to the set construction function (Algorithm 7) to get the possibilities for the next value of $\mathbf{d}$. The latter, when called at depth $j+1$, takes as input the j values already computed at higher depth for $\mathbf{d}$ and the matrix of constraints $\mathcal{C}_{j+1}$. Each line of $\mathcal{C}_{j+1}$ creates a (potentially duplicate) forbidden value for d_{j+1}, these values are all computed and the complement of this set in $\mathbb{Z}_p$ is returned by the algorithm (i.e. the set for possible values for d_{j+1} at this point of the search).

Theorem 4.4.1. *Running Algorithm 6 with increasing values of p ensures that the first solution $\mathbf{d}$ found is optimal for the function f, i.e. the solution works and its associated p is the smallest as possible.*

Optimizations: Several optimizations are possible to improve the performances of the search. First, in Algorithm 7, one can check the size of the set $\bar{S}$ at each iteration and stop as soon as the size of the set is p. Such a set means that a dead-end has been reached and that no value will be returned by the function. Then, one can leverage symmetries existing in the table but also in the function. For example, if we consider the function $f : (x, y) \longrightarrow x \oplus y$, the two variables x and y have symmetric roles. Thus, if the pair of encodings $(\mathcal{E}_x, \mathcal{E}_y)$ is valid, then the pair $(\mathcal{E}_y, \mathcal{E}_x)$ is valid as well. As a consequence, one can arbitrarily set $d_x \leq d_y$ and removing half the possibilities for (x, y).

Development of an heuristic: This algorithm of the previous section is deterministic and finds any existing set of encodings compliant with the function f *for a given value of* p. However, the right value for p is not known *a priori*, so we have to run the full algorithm for each possible value of p until we find one that works. For these reasons, we might prefer an efficient heuristic over the previous algorithm in some contexts. In Section 4.4.5, we define such a heuristic which allows to drastically improve the performance by executing directly the algorithm with realistic values for p.

Algorithm 6 Recursive function `search` that adds an element to the vector $\mathbf{d}$

Input: $\begin{cases} \mathbf{d} = (d_i)_{1 \le i \le j}$: vector of values already computed \\ $\mathcal{C} = \{\mathcal{C}_i \mid i \in \{0, \dots, \ell - 1\}\}$: pre-computed constraint matrices \\ $p \in \mathbb{N}^*$: modulus of the input encodings \\ $\ell \in \mathbb{N}^*$: target number of encodings required \end{cases}$

Result: $\mathbf{d}$: a list of encodings such that f is evaluable

if $j = \ell$ **then**
  return $\mathbf{d}$; `/* Base case: full solution found */`
else
  $\mathcal{P} \leftarrow$ `get_possible_values`$(\mathbf{d}, \mathcal{C}_{j+1}, p)$; `/* Compute possible values for` d_k `*/`
  for $x \in \mathcal{P}$ **do**
    $\mathbf{d} \leftarrow (\mathbf{d} \,\|\, x)$; `/* Append` x `to current vector */`
    $\mathbf{d}_{\text{sol}} \leftarrow$ `search`$(\mathbf{d}, \mathcal{C}, p, \ell)$; `/* Recursive call */`
    if $\mathbf{d}_{sol} \neq \perp$ **then**
      return $\mathbf{d}_{sol}$; `/* Propagate valid solution */`
    else
      $\mathbf{d} \leftarrow \mathbf{d}[: j + 1]$; `/* Backtrack by removing` x `from the solution vector */`
    end
  end
  return $\perp$; `/* All possibilities failed */`
end

Algorithm 7 Function `get_possible_values` that builds the set of possible values for the next slot of $\mathbf{d}$ given the slots already filled in.

Input: $\begin{cases} \mathbf{d} := (d_i)_{\{1 \le i \le j\}}$:the set of values for the inputs already computed \\ $\mathcal{C}_{j+1}$:The matrix of constraints of this step, pre-computed \\ $p \in \mathbb{N}^*$: the modulus of input encoding \end{cases}$

Result: S : contains only values suitable for the $j + 1$-th slot of $\mathbf{d}$.

$\bar{S} \leftarrow \{\}$; `/*` $\bar{S}$ `is the set of forbidden values for` d_{j+1} `*/`
for $c \in C_{j+1}$ **do**
  $\bar{c} \leftarrow (-c[-1] \cdot c[0], -c[-1] \cdot c[1], \dots)$; `/* */`
  $\bar{C} \leftarrow \bar{C} \cup \{\bar{c}\}$ $\bar{c} \leftarrow c[j + 1]$; `/* We retrieve the` $(j + 1)$`th coefficient of the inequation` c `*/`
  $\bar{S} \leftarrow \bar{S} \cup \left\{ \left[-\bar{c} \cdot \sum_{k=1}^{j} c_k \cdot d_k \right]_p \right\}$; `/* We compute the value forbidden by` c `*/`
end
$S \leftarrow \mathbb{Z}_p \setminus \bar{S}$
return S

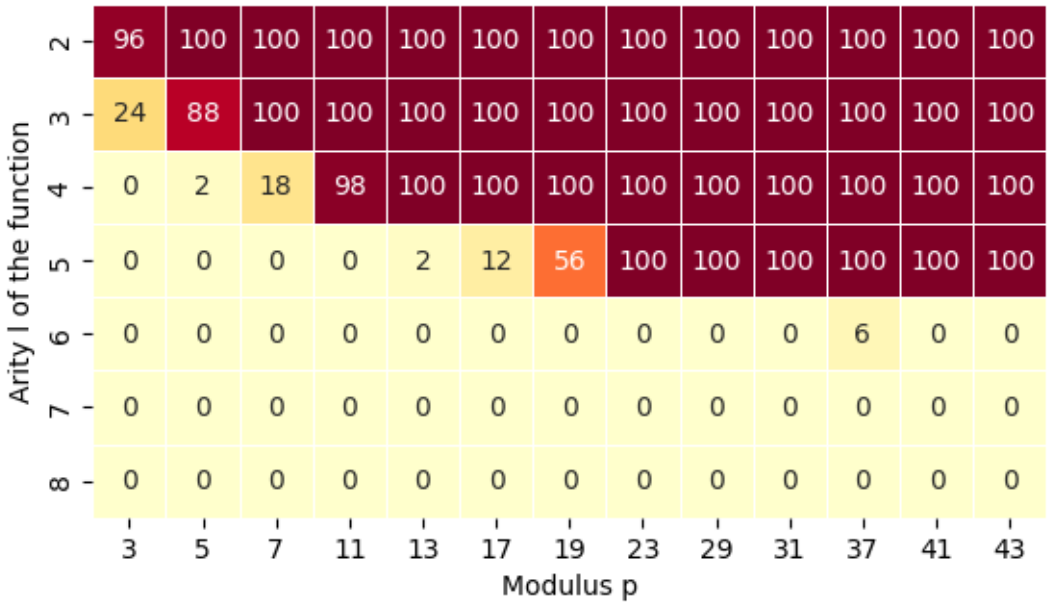


Figure 4.4: Rate of success of the algorithm for 100 random Boolean functions for different values of ℓ and p.

4.4.4 Performances Measurements

In this section, we present some experimental results to demonstrate the performances of the algorithm. We ran Algorithm 6 for a lot of random Boolean functions of arity ℓ. Two metrics are particularly interesting for us:

- The running time of the algorithm, especially in the cases where there is no solution.
- The probability of success, for a random function

Figure 4.4 shows the rate of success for random Boolean functions of arity $\ell \in \{2, 9\}$ and for prime values of $p \in [3, 39]$. It illustrates the intuitive idea that one has to increase p to evaluate functions of bigger arity ℓ. It also give a rough idea of the value of p required for a given function of arity ℓ.

Figure 4.5 shows the evolution of the time of execution of the algorithm for random Boolean functions *for which no solution exists*. It shows the explosion of the complexity for high values of p, and justifies the need of a more efficient algorithm for those function (we introduce one in Section 4.5).

Lastly, Figure 4.6 shows how long it takes to find a solution when one exists, relatively to the running time when no solution exist at all. It illustrates a form of "speed of convergence" and shows that it is located around $\frac{1}{3}$.

4.4.5 An Efficient Sieving Heuristic to Find Suitable Encodings

Let us consider a function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ of matrix of constraints $C = (C_j^{(i)})_{\substack{1 \le i \le n_j \\ 1 \le j \le \ell}}$ and its associated system of linear inequalities:

$$\begin{cases} c_1^{(1)} \times d_1 + c_2^{(1)} \times d_2 + \cdots + c_\ell^{(1)} \times d_\ell \neq 0 \mod p \\ c_1^{(2)} \times d_1 + c_2^{(2)} \times d_2 + \cdots + c_\ell^{(2)} \times d_\ell \neq 0 \mod p \\ \cdots \end{cases}$$

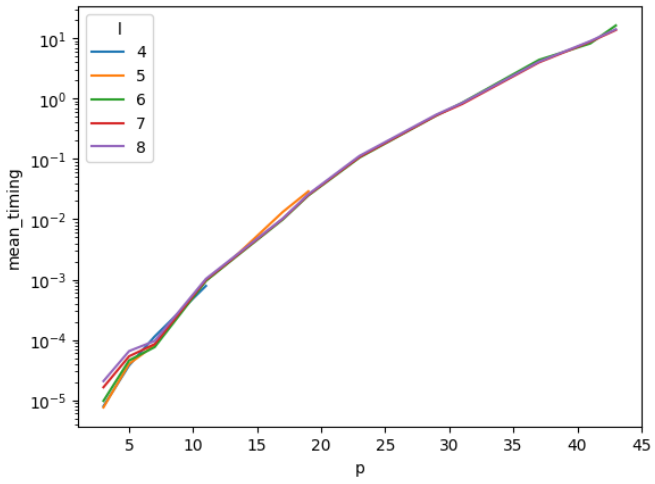


Figure 4.5: Running time of the algorithm for different values of ℓ and p for random functions. Note that the scale is logarithmic.

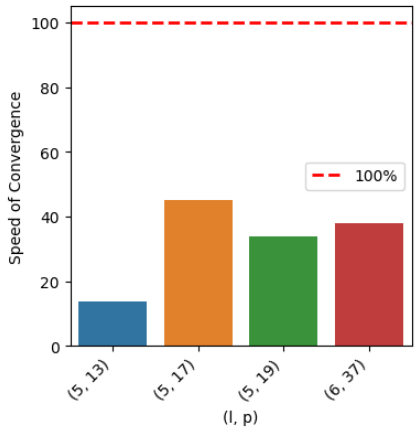


Figure 4.6: Ratio between the time to find a solution when it exists with the time to run the full algorithm when no solution exists.

Figure 4.7: Some metrics about running time.

The principle is to sample random values in $\mathbb{Z}$ (with some large bound) and affect them to the d_j's. If all the corresponding values for all the $C_i = \sum_{j=1}^{\ell} c_j^{(i)} \times d_j$ are not divisible by a value p, then the vector $(d_j \mod p \mid j \in \{1, \ldots, \ell\})$ is a solution of the system of inequalities generated by C.

To reduce the amount of samples required to find a solution, we want to avoid sampling trivially wrong sets of d_j's. For example, if all the d_j's are themselves divisible by p, then the C_i's will all be divisible as well. To tackle this problem, we perform the sampling across *prime numbers in* $\mathbb{Z}$.

Algorithm 8 Sample a solution $\mathbf{d}$ in $\mathbb{Z}$ for a function f and returns a possible value for p.

Input:
- $\{\mathcal{C}_i\}_{1 \leq i \leq n}$: The lines of the matrix of constraints $\mathcal{C}$ of the function f
- P: The set of possible values for p to be tested
- D: The set of possible values in $\mathbb{Z}$ to assign to the d_i's. (large primes)

Result: p such that it is possible to evaluate f using a modulus smaller or equal than p.

```
d ←$ D ;                                   /* Sample random prime values in ℤ */
r = C × d ;                                /* r is the right member of the system */
for p ∈ P do
    if 0 ∈ [r]_p then
        P ← P \ {p} ;    /* If this value of p divides one of the coordinates of r,
         then it will not work */
end
if | P |> 0 then
    return min(P) ;      /* Returns the smallest possible value for p, if any.  */
return ⊥
```

Running this algorithm several times and keeping the smallest returned value for p, one gets an upper bound on the minimum p required to evaluate a function with our framework. Note that, on the contrary of the deterministic search algorithm, this heuristic does not require a prime p.

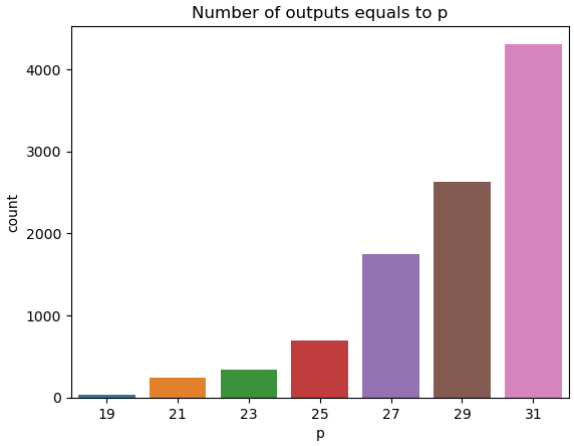


Figure 4.8: The outputs of 10000 *runs of the Algorithm 8 for the first subfunction of the Ascon S-box*

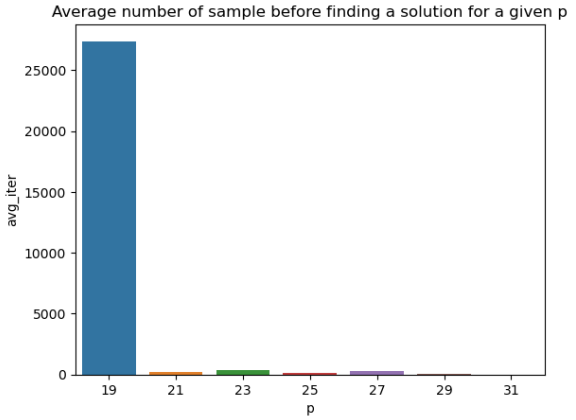


Figure 4.9: Number of iterations required to get a solution for a given value of p

Example: Let us consider the S-box of the block cipher ASCON. We study this S-box in more details and provide an exact optimized solution for its homomorphic evaluation in Section 4.7.4. Here, we apply Algorithm 8 on the five functions generating the five output bits and monitor the results until we gather $N = 10000$ non-zero possible values for p.

The figure 4.8 shows the repartition of the returned values of p by the algorithm during these N runs on the first subfunction. The optimal value of p found by the deterministic approach of Section 4.4.3 is 17 so the upper bound 19 is pretty close, despite being rarely found by the algorithm. Also, the figure 4.9 shows 21 (the second best solution found by the sieving) is almost instantly found by the algorithm.

In the process of finding the smallest p possible and a correct vector of p-encoding to evaluate a function f, this heuristic is really efficient to get a tight upper bound on the value of p.

4.5 Scaling our Approach to any Boolean Circuit

Our framework optimizes the homomorphic evaluation of single Boolean functions but suffers the following limitations:

1. For a Boolean function with a high number of inputs, the search algorithm may be very time-consuming.

2. Some functions simply do not have any solution for acceptable values for p ($p < 32$ for example) and thus are not efficiently evaluable in a single PBS.[2]

As a consequence, we need a solution to extend our framework to these cases. In this section, we propose a strategy to leverage the circuit representation of a "tough" function f to find a strategy of homomorphic evaluation with as few bootstrappings as possible.

4.5.1 Graph of Subcircuits

Let $f : \mathbb{B}^\ell \longrightarrow \mathbb{B}$ be a Boolean function, and let $\mathcal{F}$ be a Boolean circuit representing f (some preliminaries about Boolean circuits can be found in Section 4.1). Let us describe the layout of the circuit $\mathcal{F}$. It has ℓ input wires, denoted by $\{y_j\}_{1\leq j\leq \ell}$, and the output wire is denoted by z. The intermediary wires are denoted by $\{t_j\}_{1\leq j\leq \theta}$. The Boolean operation gates are of fan-out 1.

[2]The PBS can be evaluated for larger values of p but it quickly becomes inefficient as p grows.

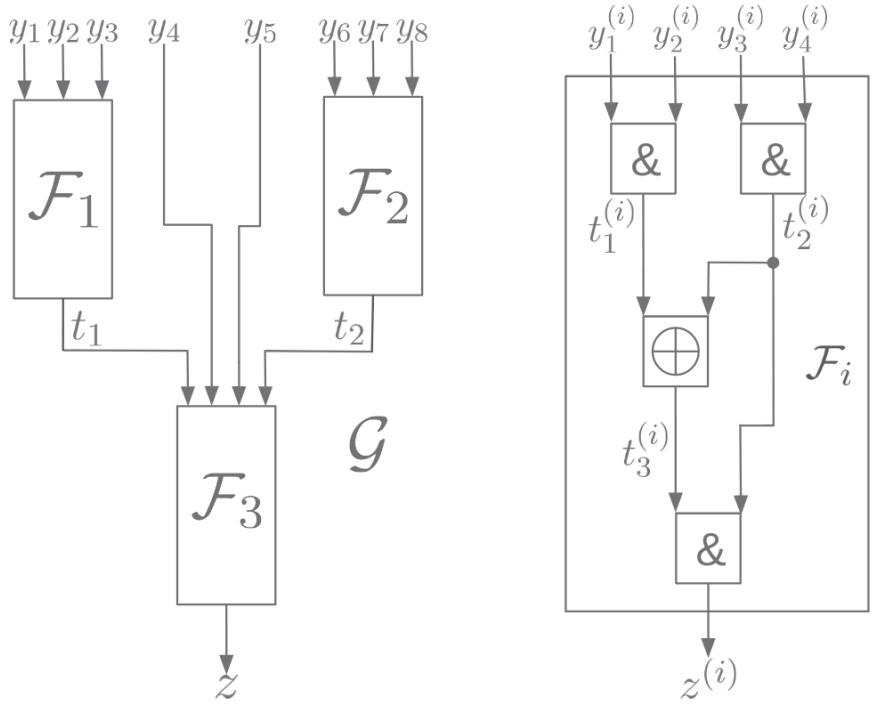


Figure 4.10: Example of graph of subcircuits (on left) and of a valid subcircuit (on right). Each subcircuit $\mathcal{F}_i$ is evaluated homomorphically with a gadget Γ_i.

Our goal is to split the circuit into a directed acyclic graph $\mathcal{G}$, whose vertexes are subcircuits $\{\mathcal{F}_1, \ldots, \mathcal{F}_k\}$ and whose edges connect the outputs of a subcircuit with the input of another. Each subcircuit $\mathcal{F}_i$ represents a subfunction $f_i : \mathbb{B}^{l_i} \mapsto \mathbb{B}$ that is evaluable with a gadget with our framework.

We use the same notations to refer to the elements of a subcircuit $\mathcal{F}_i$ and we index them with i. The output of $\mathcal{F}_i$ is denoted by $z^{(i)}$ and its inputs by $\{y_j^{(i)}\}_{1 \leq j \leq \ell}$ and so on.

The graph is valid for f with respect to modulus p if the following properties are satisfied:

- Each subcircuit $\mathcal{F}_i$ has only one output $z^{(i)}$.
- For a subcircuit $\mathcal{F}_i$, all its inputs are either inputs of the whole circuit or outputs of other subcircuits of the graph. We can write this property as:

  $$\{y_j^{(i)}\}_{1 \leq j \leq l_i} \subset \left(\{y_j\}_{1 \leq j \leq \ell} \cup \{z^{(j)}\}_{1 \leq j < i}\right)$$

  Thus, the indexing of the $\mathcal{F}_i$'s respects the topological order of the graph, i.e. no gates of $\mathcal{F}_i$ has a child in any of the $\mathcal{F}_j$, with $j < i$.
- All the Boolean functions f_i represented by the subcircuits $\mathcal{F}_i$ are evaluable in a single bootstrapping with modulus p with our proposed method.
- The last subcircuit $\mathcal{F}_c$ of the graph has z (the output of the main circuit) for output: $z^{(c)} = z$.

To homomorphically evaluate the function f, we evaluate each subcircuits with one bootstrapping for each of them and get the final result. In order to reduce the cost of evaluation for a given p, the goal is hence to find the *smallest* valid graph possible in terms of number of subcircuits. Taking a greater value of p produces a different graph that may be smaller (as subcircuits might be larger), but the timings of bootstrapping in this graph might on the other hand be greater. One can therefore run the search for different values of p and keep the most efficient setup among the possible graphs.

4.5.2 Heuristics to Find a Small Graph

Finding such a graph can be done by exhaustively evaluating all the possible subcircuits with our method introduced in Section 4.4, and then find the more efficient one. However it is not really practical to evaluate *all* the possible subcircuits, so we develop some heuristics to reduce the search space. Let us start by defining a few bounds on the considered subcircuits, we will leave the other ones apart in our algorithm:

- The subcircuits have at most B inputs ($\forall i, l^{(i)} < B$). The purpose of this bound is to limit the running time of Algorithm 6. In practice, for our experiments, we took $B = 10$.
- The subcircuits are evaluable with one single bootstrapping with a maximum value p_{max}. This value ensures a bootstrapping with a reasonable timing. If the search algorithm fails for p_{max}, the subcircuit is dropped without trying to extend p. In our experiment, we took $p_{max} = 31$.

In order to decompose our Boolean circuit into a graph satisfying the above property for a modulus p, we would want to exhaustively search all the subcircuits of $\mathcal{F}$ compliant with the bounds we introduced earlier. However, all subcircuits are not equally worth to evaluate. In particular a wire incoming a copy gate is particularly worth evaluating because is costs one bootstrapping but produce several inputs for the next subcircuits.

We gather wires that precede a copy gate in the set $\mathcal{Z}$. We add to this set the global output z. We also gather the input wires of the global circuit $\mathcal{F}$ in the set $\mathcal{Y}$. We define the notion of *atomic subcircuit* that is a valid subcircuit whose all inputs belong to $\mathcal{Y} \cup \mathcal{Z}$ and whose output belongs to $\mathcal{Z}$. Note that the merge of two atomic subcircuits that respect the global circuit wiring is also an atomic subcircuit.

Our heuristic works as follows:

1. For each of these outputs $z_i \in \mathcal{Z}$, we exhaustively construct a set $\widehat{\mathcal{F}_{z_i}}$ that gathers all the atomic subcircuits whose output is z_i. We then filter out the subcircuits of $\widehat{\mathcal{F}_{z_i}}$ that do not comply with the bounds introduced at the beginning of the section or that are not evaluable with a gadget with the input modulus p (we use Algorithm 6 to decide that).

2. Now we want to construct the smallest valid graph evaluating $\mathcal{F}$ using subcircuits from the $\widehat{\mathcal{F}_{z_i}}$'s. While finding the smallest graph is hard, constructing any valid graph is easy. As a consequence, our strategy to find a small graph is to randomly create a lot of valid graphs and to take the smallest one. The procedure to create a valid graph is the following: we start from the output z and we randomly draw a subcircuit $\mathcal{F}_z$ from $\widehat{\mathcal{F}_z}$. The inputs of $\mathcal{F}_z$ can be sorted into two categories: the ones belonging to $\mathcal{Y}$ and the ones belonging to $\mathcal{Z}$. For each one of these latter wires $w \in \mathcal{Z}$, we repeat the procedure, i.e. we draw a subcircuit $\mathcal{F}_w$ from $\widehat{\mathcal{F}_w}$, and so on. When we have reached all the input wires of $\mathcal{F}$, we get a valid graph $\mathcal{G}$. This second step is run a large amount of times (the number of trials is a parameter of the method), and the smallest graph, i.e. the one with the fewest subcircuits, is returned.

We carried on this method on the S-box of AES in Section 4.7.5.

4.5.3 Parallelization of the Execution of the Graph

Once we have our graph $\mathcal{G}$, we can identify its $n_{\mathcal{L}}$ *layers*. Formally, they are defined as:

Definition 4.5.1. A layer $\mathcal{L}$ of a graph $\mathcal{G}$ is a set of subcircuit $\{\mathcal{F}_\alpha, \ldots, \mathcal{F}_\omega\}$ of $\mathcal{G}$ that verifies: $\forall \mathcal{F}_i, \mathcal{F}_j \in \mathcal{L}, \mathcal{F}_i$ is not an ancestor node of $\mathcal{F}_j$.

By construction, all the subcircuits belonging to the same layer can be evaluated in parallel. This reduces the number of bootstrapping steps from k (the number of subcircuits in the graph $\mathcal{G}$) to $n_{\mathcal{L}}$ (the number of layers). Our graph-finding heuristic can be tweaked to select the graph with minimum number of layers instead of minimum number of subcircuits to optimize parallelization.

4.6 Implementation Considerations: Adaptation of the Parameters Selection and of the `tfhe-rs` Library

From a high level point of view, our technique can be seen as adding an additional layer of abstraction on top of TFHE. It remains to explain how to select the parameters for the TFHE scheme. Moreover, to implemented our framework, we had to fork the `tfhe-rs` library [Zam22c]. The following section covers these two issues.

4.6.1 Crafting of Parameters

In practical setting, we need a set of parameters for TFHE. Finding an optimal set of parameters for a given application is a hard problem, that we study in depth in Chapter 8. This work being anterior to the one of Chapter 8, we used the framework of [Chi+21] and implemented in [Zam22a].

In this paper, the authors elaborate a strategy where they define an *atomic pattern* of FHE operators, that is to say a subgraph of FHE operators in which the noise of the output is independent from the one in the inputs. Then, they develop an optimization framework to derive the best set of parameters for a given atomic pattern. In particular, the first atomic pattern they study, that they denote by $\mathcal{A}^{(CJP21)}$, is a subgraph composed of a linear combination of ciphertexts with clear constants, then a `Keyswitch` and then a `BlindRotate` followed by a `SampleExtract` (`ModulusSwitch` is seen as a part of `BlindRotate`). To dimension the parameters of TFHE to evaluate such an atomic pattern, their framework takes as input the 2-norm of the vector of constants of the linear combination (denoted by ν) and a noise bound t on the standard deviation of the distribution of error in a ciphertext that ensures a correct decryption with a good probability $(1 - \epsilon)$. We elaborate further on how this bound is constructed below in this section. If we look closely, the evaluation of a gadget we introduced in Definition 4.3.4 can be seen as a $\mathcal{A}^{(CJP21)}$ with a few differences. Thus, we slightly modified the tool `concrete-optimizer` [Zam22a], that allows to generate parameters for different types of atomic patterns, to support our gadget as a new atomic pattern. Let us dive into the differences between a gadget and a $\mathcal{A}^{(CJP21)}$:

Support of odd values for p: Using an odd value for p changes the bootstrapping procedure, and in particular the definition of the accumulator for the `BlindRotate` (as explained in Chapter 3). With our construction, the windows in the polynomial are half the size of the ones for an even p, which impacts the noise bound t. As this bound depends of the failure probability α that the user is ready to tolerate, we shall denote it t_α hereafter, which satisfies: $t_\alpha = \frac{\Delta}{2z^*(1-\sqrt[N]{1-\alpha})}$ where z^* is the *standard score* and Δ is the scaling factor (see [Chi+21] for more explanations). The impact of our adaptation on this formula is solely with respect to the scaling factor. In the context of an $\mathcal{A}^{(CJP21)}$, we have $\Delta = \frac{q}{2^\pi p}$ with π the number of MSB for padding. As explained in Chapter 3, we do not need any padding mechanism anymore, so the 2^π vanishes. However, the length of a window is divided by 2, and p does not divide q anymore so we need to add a rounding. We finally get $\Delta = \left\lfloor \frac{q}{2p} \right\rceil$.

Link between input encodings and ν: In a scenario where only one gadget has to be evaluated, its inputs are freshly encrypted ciphertexts. Then, there is no need to perform any encoding switching before evaluating the gadget, and so we are in the context of a $\mathcal{A}^{(CJP21)}$ with $\nu = 1$. However, if we are in a context of a *graph* of gadgets like in Section 4.5, the output of a gadget can be used as input of subsequent gadgets under different encodings. In this case, some encoding switchings are necessary. If these encoding switching are made using a mutiplication by a constant (Property 4.3.3), we are still in the context of a $\mathcal{A}^{(CJP21)}$ but with $\nu \neq 1$. To formalize that, we first recall that Algorithm 6 produces gadgets of the form $\Gamma = (\mathcal{E}_{\mathbf{in}}, \mathcal{E}_{out}, p_{in}, p_{out}, f)$, with $\mathcal{E}_{in}^{(i)} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_i\} \end{cases}$. Thus, if we fix that all gadget output ciphertexts are encoded under $\mathcal{E}_{out} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$, then the encoding switchings needed before an evaluation of Γ corresponds to a linear combination of the inputs with the vector $\mathbf{d} = (d_i \mid i \in [1, \ell])$, so we fall back on a $\mathcal{A}^{(CJP21)}$ with $\nu = \|\mathbf{d}\|$.

We implemented these changes in `concrete-optimizer` and uses it to generate sets of parameters for our implementations detailed in Section 4.7.

4.6.2 Concrete Implementations of p-Encodings and Homomorphic Functions in `tfhe-rs`

To implement our framework, we relied on the `tfhe-rs` library [Zam22c]. Here is a list of the major changes we applied to the code:

Addition of the notion of p-encoding: An encoding $\mathcal{E}$ is simply implemented with a structure `Encoding` storing two `HashSets` and the modulus p. The `HashSets` represent both sets $\mathcal{E}(0)$ and $\mathcal{E}(1)$. When creating an `Encoding`, the code checks whether the two underlying sets are disjoint or not. Moreover, the operation of encryption and decryption are modified as well. The signatures change from:

```
encrypt(Boolean, ClientKey) -> Ciphertext
```

to:

```
encrypt(Boolean, ClientKey, Encoding) -> Ciphertext
```

(same for `decrypt`). The functions also perform the mapping $\mathbb{B} \mapsto \mathbb{Z}_p$ before encryption and the other way around after decryption.

Support of odd moduli: The native `tfhe-rs` only support power-of-two-moduli p. We extended the library to handle odd values for p. This required modifying the encryption and decryption algorithm, and to compute the sets of parameters with the method of Section 4.6.1.

Definition of the new structure `Gadget`: According to the evaluation strategy we introduced in Section 4.3.2, we wrote a new structure `Gadget`, associated to a Boolean function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$, carrying:

- A list of the `Encoding` objects for the inputs: $\mathcal{E}_{in} = (\mathcal{E}_1, \ldots, \mathcal{E}_l)$, with the input modulus p_{in} they encoded on.
- The output `Encoding` object $\mathcal{E}_{out}$, with the output modulus p_{out} it is encoded on.
- The clear function f.

When such a structure is constructed, it self-checks whether $f(\mathcal{E}_{in})$ is valid. Then, when provided ℓ `Ciphertexts` objects encoded under their respective p-encoding, it executes the homomorphic sum and the PBS and outputs the results encoded under $\mathcal{E}_{out}$. Some utilitary functions performing encoding-switching are also available, allowing the chaining of several `Gadget`.

Implementation of the accumulator: The procedure of bootstrapping of `tfhe-rs` is slightly modified to support the new version of the accumulator we introduced in Chapter 3.

Parsing of graphs: We implemented a Python script that produces graphs to represent more complex functions that requires several PBS, as described in Section 4.5. These graphs are stored with a comprehensive file format and our Rust implementation has a module of parsing allowing to load these graphs and automatically generate the corresponding graph of `Gadget`.

4.7 Application to Cryptographic Primitives

In this section, we apply our approach on some cryptographic primitives. For each primitive, we first explain the construction of the gadgets required and report the concrete performances of our implementation. We detailed all the timings of our experimentations along with the sets of parameters we used in Section 4.7.6.

For performance measurement, we implemented our framework in our fork of the library `tfhe-rs` [Zam22c] adapted as discussed in Section 4.6 and we generated the sets of parameters thank to our version of `concrete-optimizer` [Zam22a]. By default, we tailored the sets of parameters to limit the probability of failure ϵ of a bootstrapping to 2^{-40}, and a security level of $\lambda = 128$ bits. All experiments have been carried out on a laptop with a 12th Gen Intel(R) Core(TM) i5-1245U CPU with 10 cores and a frequency of 4.4 GHz, and 16 GB of RAM.

4.7.1 SIMON Block Cipher

SIMON is a hardware-oriented block cipher developed in [Bea+15], which relies only on the following operations: `AND`, rotation, `XOR`. It is a classical Feistel network for which the Feistel function consists in applying basic operations on the branch, xoring the subkey and then xoring the result with the other branch as depicted in the Figure 4.11 (on this figure, S^i denotes the left circular shift by i bits.). We use one ciphertext per bit so the rotation operation is essentially free. Note that the key is considered as a plaintext, which does not change anything in the framework. In our implementation, we considered a (128-128) instance of SIMON (i.e. the whole state and the key are of size 128).

The Boolean function to evaluate can be defined as

$$f(b_0, b_1, b_2, b_3, b_4) = b_0 \cdot b_1 \oplus b_2 \oplus b_3 \oplus b_4 \ .$$

Using Algorithm 6, we found the smallest possible p ($p = 9$) and the following 9-encodings to evaluate each bit of the Feistel function with one single bootstrapping (i.e. totalling 64 PBS per round).

$$\mathcal{E}_0 = \mathcal{E}_1 = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases} \quad \text{and } \mathcal{E}_2 = \mathcal{E}_3 = \mathcal{E}_4 = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{2\} \end{cases} \quad \text{with } p = 9.$$

The sum of these p-encodings yields the output encoding:

$$\mathcal{E}_{out} = \begin{cases} 0 \mapsto \{0, 1, 4, 5, 8\} \\ 1 \mapsto \{2, 3, 6, 7\} \end{cases} \quad \text{with } p = 9$$

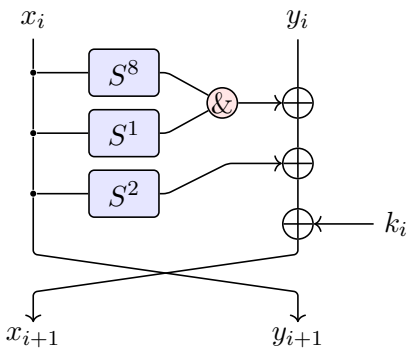


Figure 4.11: One Feistel round of SIMON.

which is valid for f. After the PBS, all the bits of the state are encrypted under the encoding $\mathcal{E}_0$. We formalize that with the gadget $\Gamma = ((\mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, \mathcal{E}_3, \mathcal{E}_4), \mathcal{E}_0, 9, 9)$

To perform a Feistel round on a state of size k, the gadget Γ is applied in parallel $k/2$ times. Note that one bit may be used in several evaluation as b_0, b_1 and b_2. So we sometimes have to switch from $\mathcal{E}_0$ to $\mathcal{E}_1$ by a simple external multiplication by 2, which is negligible in terms of performances.

Using our version of `concrete-optimizer` [Zam22a], we crafted a set of parameters suitable for this modulus and these encodings. On our machine, one PBS with such parameters takes about 9.5 ms. The theoretical timings achieved on one full block without any parallelization is 41 seconds (68 rounds $\times$ 64 bits $\times$ 9.5 ms) which we confirmed experimentally.

Nonetheless, this setting is intrinsically parallelizable: the 64 gadgets of each round can be performed in parallel. We implemented parallelization using the module `Rayon` of Rust, which made the total timings drop to 13 seconds on our machine.

Compared to [Ben+22] that implemented the same block cipher on an equivalent hardware with parallelism, our implementation is about 10 times faster. Table 4.6 shows the comparison. Note that in this paper, the probability of failure is not specified. As ours is pretty conservative, this is a good argument in favor of our framework.

4.7.2 The Trivium Stream Cipher

Trivium [De 06] is a stream cipher that uses a circular state. At each round, the bits are rotated within the state, except for three of them that are refreshed using the Boolean function of Section 4.7.1. The outer stream is generated by xoring three bits of the state each round once a "warming-up" phase is achieved.

For each generated key bit, it requires performing this function three times and aggregating five `XOR` operations in the center. Our strategy is to evaluate the refreshing function three times per round with one PBS for each of them, then get the result in $\mathbb{Z}_2$ and chain the five `XOR` operations to get the output. Figure 4.12 illustrates the layout of the cipher.

In [BOS23], the authors implement Trivium using the original `tfhe-rs` library, with 2 bits of message and 2 bits of carry for a total of 4 significative bits out of the 32 of a ciphertext component. They call this mode the `shortint` mode. The use-case they target is transciphering.

To compare our implementation with the one of [BOS23], timings are not a good metric as in their work they are provided on a massive AWS instance with a significant amount of parallelism. A better metric is to count the number of PBS and compare the parameter sets.

We reproduced the PBS operation with their parameter set on our machine and then simply estimated the timings of one round of Trivium with their approach with no parallelism. The

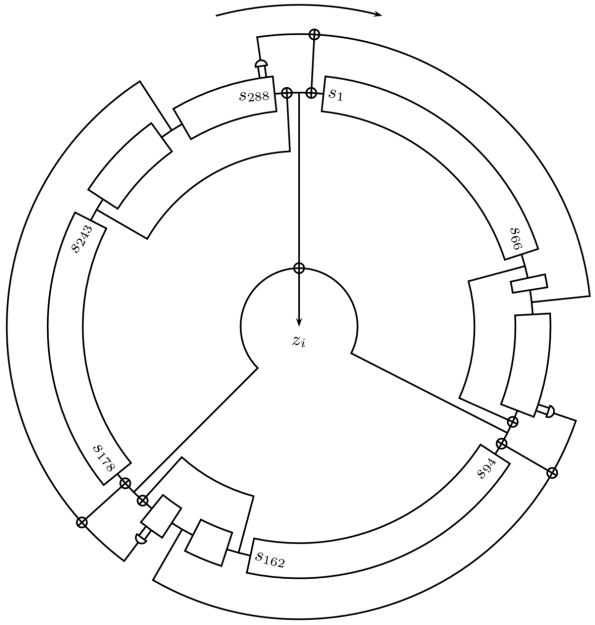


Figure 4.12: The trivium stream cipher. Figure extracted from [De 06]

.

results are summed up in Table 4.1. Note that in our implementation we do not refresh the output bits with a PBS after the chain of `XOR`, because in the use-case of transciphering one more `XOR` has to be performed with the message. We take advantage of this and move the last PBS into the transciphering phase.

Table 4.1: Comparison of timings of one round of Trivium between our work and [BOS23], with $\epsilon = 2^{-40}$.

Instance	Timing PBS	Number of PBS per round	Estimated timings
[BOS23]	6.6	7	46.2 ms
Our work	9.5	3	28.5 ms

4.7.3 Keccak Permutation

Keccak is a hash function standardized by NIST under the name *SHA-3* [NIS15]. It is a sponge function, whose transformation is called the *Keccak permutation*. It consists of five sub-functions: θ, ρ, π, χ, and ι.

Let us recall that our approach encrypts each bit in one TFHE ciphertext. Let us explain the stategies of homomorphization of these sub-functions:

- ρ and π simply reorder the bits within the state, so they are not impacted by the homomorphization.
- θ is just a serie of `XOR` operations, so it can be performed with a serie of homomorphic additions and without any PBS provided that the input ciphertexts are defined over $\mathbb{Z}_p$ with $p = 2$.
- χ is the only non-linear function of the permutation, and has to be performed with a PBS. It is the transformation that applies the function defined by

$$f_\chi(a, b, c) = a \oplus c \oplus b\&c$$

to get each bit of the output state.

- Finally, ι performs a simple `xor` with a constant, so it can be handled in a similar manner that θ. The difference is that the constant is in clear this time.

The p-encodings we use are:

- $\mathcal{E}_{\&} = \begin{cases} 0 \mapsto \{1\} \\ 1 \mapsto \{2\} \end{cases}$ with $p_{\&} = 3$ to evaluate the & operator in the alternative formula of χ.
- $\mathcal{E}_{\oplus} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ with $p_{\oplus} = 2$ for the other operations of $\oplus$.

Our strategy of homomorphic evaluation of the Keccak permutation is as follows:

1. Encrypt the input state under the encoding $\mathcal{E}_{\oplus}$.
2. Evaluate the subfuctions θ, ρ, and π. Theses functions being purely linear, they can be performed only with sums under $\mathcal{E}_{\oplus}$.
3. Change the encoding from $\mathcal{E}_{\oplus}$ to $\mathcal{E}_{\&}$ with one PBS per bit of the state (Property 4.3.5).
4. Evaluate the `AND` operator of the subfunction χ with the gadget
$$\Gamma_{\&} = ((\mathcal{E}_{\&}, \mathcal{E}_{\&}), \mathcal{E}_{\oplus}, 3, 2)$$
associated to function $f_{\&} : (x, y) \mapsto x\&y$. This gadget is applied once per bit of the state.
5. Evaluate the remaining $\oplus$ operators of χ and the ι subfunctions, then jump back Step 2. for the next loop iteration.

Casting a ciphertext from $\mathcal{E}_{\oplus}$ to $\mathcal{E}_{\&}$ (Step 3) is a bit tricky because $p_{\oplus} = 2$ is even. Because of the negacyclicity problem, one needs $\mathcal{E}_{\&}(0) = [-\mathcal{E}_{\&}(1)]_{p_{\&}}$. With $p_{\&} = 3$, the only candidate is the encoding $\mathcal{E}_{\&}$ defined above.

As a result, each round takes two programmable bootstrappings per bit. An implementation with our tweaked version of `tfhe-rs` takes 16.5 seconds (without any parallelism) on our hardware to perform one Keccak round on a state of 1600 bits in spite of the two PBS required per round and per bit. Those timings are possible because of the small values of p allowing the use of a set of small parameters, which speeds up the computation. A full run of Keccak counting 24 rounds, we can then estimate the timings without parallelism to 6.6 minutes. For the sake of simplicity, we use the same set of parameters for both types of PBS, avoiding the hassle of using two different server keys.

This strategy of implementation complies with the more generic one that we introduce in Section 4.7.4 and that is illustrated on Figure 4.14. It suits very well the use-cases where linear and non-linear operations are alternating.

4.7.4 Ascon

Ascon [Dob+21; Dob+19] is a lightweight block cipher algorithm that was designed to provide efficient and secure encryption and authentication for a wide range of applications, particularly in resource-constrained environments such as embedded systems and IoT devices. The name "Ascon" stands for "Authenticated encryption for Small Constrained Devices". We implemented its S-box, whose circuit is represented on Figure 4.13.

This layout is a bit different from the others: the S-box takes five bits as input and outputs five bits. We denote $f_0, \ldots, f_4$ the five functions of $\mathbb{B}^5 \mapsto \mathbb{B}$ that generate the 5 output bits $x_0, \ldots, x_4$. Thus, we need to define five gadgets (one per function).

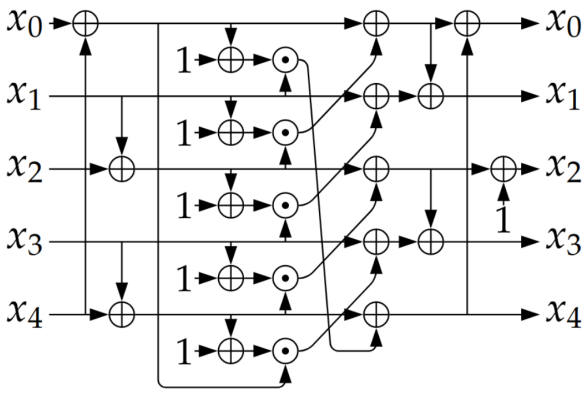


Figure 4.13: The 5-bits look-up table of ASCON. Figure extracted from [Dob+21]

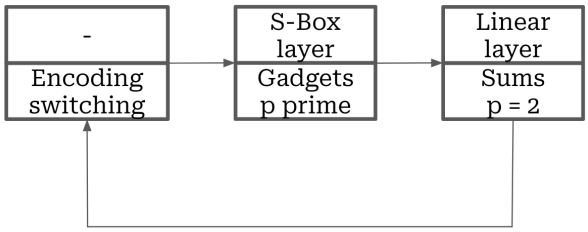


Figure 4.14: A common layout to evaluate cryptographic primitives. The upper part of the boxes represents what happens in the clear, while the lower part shows the encrypted operations.

These functions, once analyzed by the algorithm, can be computed in one single bootstrapping each, but for different values of p (respectively $p = 17, 7, 7, 15, 11$ that are the smallest possible values). We could implement the gadgets $\Gamma_0, \ldots, \Gamma_4$ (associated to $f_0, \ldots, f_4$) with different values for p_{in}, but this would imply to introduce some encoding switchings before each round of hashing. To keep things simpler we generated only encodings with $p = 17$, making the implementation more straightforward as no encoding switching is required. For each subfunction f_i, five canonical 17-encodings $(\mathcal{E}_{i,0}, \ldots, \mathcal{E}_{i,4})$ of form

$$\mathcal{E}_{i,j} = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{d_{i,j}\} \end{cases}$$

are computed. The results are displayed in the Table 4.2. Note the zero values in some cases, they show that the variable is not used in the subfunction.

The S-box layer is followed by a linear layer, where the bits of the states are shifted and combined with `XOR` operations. This can be trivially done with $p = 2$. Finally, to prepare the next round, an encoding switching is performed to send back the ciphertexts on 17-encodings. This is summed up in Figure 4.14. Note that there is no encoding switching from non-linear layer to linear layer because the gadgets can directly outputs ciphertexts under $\mathcal{E}_\oplus = \begin{cases} 0 \mapsto \{0\} \\ 1 \mapsto \{1\} \end{cases}$ with $p = 2$.

To wrap up, we construct the five gadgets $\Gamma_i = ((\mathcal{E}_{i,0}, \ldots, \mathcal{E}_{i,4}), \mathcal{E}_\oplus, 17, 2, f_i)$. They will carry the evaluation of the S-boxes and output ciphertexts encrypted under $\mathcal{E}_\oplus$. Then, the linear layer is trivially evaluated with homomorphic sums. An encoding switching from $\mathcal{E}_\oplus$ to $\mathcal{E}_{i,j}$ allows to come back to non-linear operations.

Using this solution, the S-box is evaluated in 92 ms. Note that the 5 different PBS described in Table 4.2 have different norms of vector $\mathbf{d}$ so they may have a different set of parameters for each. We use the more restrictive one (i.e. the one with greater $\|\nu\|$) for the 5. Estimating the

subfunction	$d_{i,0}$	$d_{i,1}$	$d_{i,2}$	$d_{i,3}$	$d_{i,4}$
f_0	1	2	3	7	14
f_1	1	2	2	2	4
f_2	1	2	2	4	0
f_3	1	1	5	5	3
f_4	1	2	0	4	3

Table 4.2: Parameters $d_{i,j}$ for Ascon, with $p = 17$ for every subfunction.

timings of a full run of Ascon is not trivial because it depends a lot of the parameters. To give a rough idea, in hashing mode, 64 S-boxes are required per round, with 12 rounds recommended. The outputs of the S-boxes are in $\mathbb{Z}_2$ to allow the evaluation of the linear layer of Ascon. At the end of this linear layer, the encoding of each of the 320 bits of the state must be switched back to $\mathbb{Z}_{17}$ with a PBS. To do so, we use the same set of parameters as for the encoding switching in Step 3 of the Keccak evaluation in Section 4.7.3.

This gives an estimation of 89 seconds for one Ascon hash.

4.7.5 AES

AES [DR00], or Advanced Encryption Standard, stands as one of the most widely used and trusted encryption algorithms in the world of computer security. Its standardization occured in 2001 when it was adopted by NIST to replace the obsolete DES (Data Encryption Standard). Implementing this primitive in FHE is known as particularly tricky and only few attempts have been made [GHS12], [CLT14], [Tra+23].

A round of AES can be decomposed into 4 steps:

1. `SubBytes`: a non-linear substitution step where each byte is replaced by another according to a lookup table. This step concentrates all the challenge for homomorphization, the other one being trivial with our framework.

2. `ShiftRows`: a transposition step where the last three rows of the state are shifted cyclically a certain number of times. As our framework encrypts each bit in a distinct ciphertext, this step is for free.

3. `MixColumns`: a linear mixing operation which operates on the columns of the state, combining the four bytes in each column. This step can be implemented using only `XOR` operations and bit-shiftings. The former are trivial with our framework using $p = 2$ and the latter are for free as the ones in the previous step.

4. `AddRoundKey`: each byte of the state is combined with a byte of the key from the key schedule using a `XOR`. Still using $p = 2$, this can be carried out easily.

We refer to Section 5.2 for a more in-depth presentation of the algorithm.

The S-box of `SubBytes` takes 8 bits in input and yields 8 bits of output. It is defined by two substeps: an inversion in $GF(2^8)$ followed by an affine transformation. While the latter is trivial to compute with TFHE, the former is much trickier and thus we did not take advantage of this representation. Using our framework, the obvious-looking solution is to split the full S-box $\mathbb{B}^8 \mapsto \mathbb{B}^8$ into 8 subfunctions $f_0, \ldots, f_7 : \mathbb{B}^8 \mapsto \mathbb{B}$. We could then give them to the search algorithm of Section 4.4. If this would work, we could evaluate the Rjindael S-box in 8 PBS. Unfortunately, the algorithm does not converge for values of p "reasonable", that is to say less than 7 bits.

We thus need to leverage an alternative representation of the S-box. A well known efficient Boolean representation of the AES S-box is given in [BP10]. In this work, authors applied logic

minimization techniques to produce an optimized Boolean circuit (in terms of number of gates) of the S-box splitted in 3 phases:

1. A purely linear layer mapping the 8 input bits onto 22 bits.
2. A middle non-linear layer, represented by a circuit with exclusively `AND` and `XOR` logic gates, mapping the previous 22 bits onto 18 bits.
3. A final purely linear layer mapping the 18 bits on the 8 output bits of the S-box.

To design our implementation of AES, we will use the strategy we introduced for Keccak (Section 4.7.3) and ASCON (Section 4.7.4) and that is illustrated on Figure 4.14. The steps `ShiftRows`, `MixColumns`, `AddRoundKeys` only involves `XOR` operators, so we will carry them out with $p = 2$. Same things with the steps 1. and 3. of the circuit of `SubBytes` of [BP10]. The only part remaining is the Step 2. of the `SubBytes`, that is a non-linear circuit. We evaluate this circuit using gadgets and the approach introduced in Section 4.5. A layer of encoding switching allows to link both parts.

In particular, `MixColumns` can be reduced to a serie of `XOR` (in our implementation, we use the circuit designed in [Max19]).

In the following, we focus on the implementation of the non-linear layer using the approach by graphs of Section 4.5.

Homomorphization of the S-box

We start from the circuit representation given in the work of [BP10]. This set of instructions is compiled into a circuit $\mathcal{A}$, compliant with the definitions introduced in Section 4.5.1.

Each of the 18 outputs $(z_0, \ldots, z_{17})$ are isolated from each other and the circuits $(\mathcal{A}_0, \ldots, \mathcal{A}_{17})$ generating them are separated. Of course, some intermediary values are used in several circuits, but for now we ignore this and we considerate the 18 problems as independent from each other.

Then, for each circuit $\mathcal{A}_i$, we run the algorithm explained in Section 4.5 to produce an efficient graph. We merge all those graphs and run everything for a total of 36 PBS to evaluate the full circuit $\mathcal{A}$, with a global $p = 11$. This allows a relatively quick bootstrapping.

Recall that the `SubBytes` step is made of 16 S-boxes. So, we can derive that one execution of the `SubBytes` step takes $16 \times 36 = 576$ PBS.

The outputs of this step would be encoded with $p = 2$, allowing the `XOR` operations of the following steps to be performed efficiently. We also need to take into account the encoding switching to come back to $p = 11$ before each `SubBytes`. It costs one PBS per bit, so 128 PBS. Finally, this gives a total of 704 PBS per round. For `AES-128`, which takes 10 rounds, we estimate a full run to 7040 PBS.

Performances

In terms of performances, with a set of parameters ensuring a security level of $\lambda = 128$ bits and an error probability $\epsilon = 2^{-40}$, a PBS takes 17 ms on our hardware. The total runtime of the whole implementation on one thread is 135 s. We note that the 16 evaluations of S-boxes in `SubBytes` can be parallelized, as well as each of the 128 encoding switchings before `SubBytes`. Moreover, within each S-box, we can locally apply our strategy of parallelization introduced in Section 4.5.3.

We compare favorably to previous works of [GHS12] and [CLT14], who report timings of respectively 18 minutes and 5 minutes for a full AES, Once again, authors do not mention the value of ϵ. The more recent work of [Tra+23], also proposes an implementation of `AES-128` using a completely different technique called the *tree-bootstrapping*. On a similar experimental setup, but with a failure probability $\epsilon = 2^{-23}$, they claim an execution in 270 s on one thread.

Identification		TFHE parameters									Timings
Ref.	Sections	n	k	N	σ_{LWE}	σ_{GLWE}	B_g	ℓ_g	B_{KS}	ℓ_{KS}	PBS
PBS_{gate}	Table 4.5	722	2	512	$2^{16.2}$	$2^{7.8}$	2^6	3	2^3	4	10 ms
$PBS_{(9,2)}$	4.7.1, 4.7.2	684	3	512	2^{16}	2^2	2^{10}	2	2^3	4	9.5 ms
$PBS_{(3,2)}$	4.7.3	676	5	256	2^{22}	2^7	2^{18}	1	2^4	3	5.25 ms
$PBS_{(2,1)}$	4.7.3, 4.7.4	676	5	256	2^{22}	2^7	2^{18}	1	2^4	3	5.25 ms
$PBS_{(17,5)}$	4.7.4	740	2	1024	2^{13}	2^2	2^7	3	2^5	3	18 ms
$PBS_{(11,4)}$	4.7.5	708	3	512	2^{15}	2^2	2^6	4	2^2	7	17 ms

Table 4.3: Sets of TFHE parameters for each PBS used in our implementations, with the constraints used to generate the sets, and the performances. Each setting is referenced as $PBS_{(p,\mathcal{N}_d)}$ with $\mathcal{N}_d = \lceil \log_2(\|d\|) \rceil$. All this parameters ensure a level of security $\lambda = 128$ bits and a failure probability of bootstrapping of $\epsilon = 2^{-40}$. q is always fixed to 2^{32}. PBS_{gate} refers to the naive case of the gate bootstrapping implemented in [Zam22c] and is used to estimate the timings of the naive strategy in Table 4.6.

Section	Primitive	Complexity in PBS
4.7.1	One round of SIMON-128	64 $PBS_{(9,2)}$
	One full run of SIMON-128	4352 $PBS_{(9,2)}$
4.7.2	One round of Trivium	3 $PBS_{(9,2)}$
	One warm-up phase of Trivium $(*)$	3456 $PBS_{(9,2)}$
4.7.3	One round of Keccak	1600 $PBS_{(3,2)}$ + 1600 $PBS_{(2,1)}$
	A full Keccak permutation $(*)$	38400 $PBS_{(3,2)}$ + 38400 $PBS_{(2,1)}$
4.7.4	One evaluation of Ascon's S-box	5 $PBS_{(17,5)}$
	One full Ascon hashing run $(*)$	3840 $PBS_{(17,5)}$ + 3840$PBS_{(2,1)}$
4.7.5	One evaluation of the AES S-box	36 $PBS_{(11,4)}$
	A full run of `AES-128`	5760 $PBS_{(11,4)}$ + 1280 $PBS_{(2,1)}$

Table 4.4: Complexity of the different primitives we implemented with respect to the PBS of Table 4.3. The primitives indicated by a $(*)$ are estimations while the others have been fully implemented.

We ran again our code with another set of parameters tailored for the same ϵ and obtained a full run in 103 s. Note that in our implementation, we used the mode restrictive set of parameters $\mathtt{PBS}_{(11,4)}$ for every bootstrapping (even the ones that should be performed with $\mathtt{PBS}_{(2,1}$. We also derived the theoretical timing that could have been achieved if we had implemented this with two server keys (one for each set of parameters). This theoretical timing should be of 105 s with $\epsilon = 2^{-40}$, we added it in Table 4.6.

4.7.6 Summary of Applications

We summarize hereafter the parameters and performances of our implementations of cryptographic primitives. Table 4.3 gives an overview of the TFHE parameters used for each value of p in these examples. They all meet the required level of security of 2^{128} and the error probability $\epsilon = 2^{-40}$. It also shows the associated p and the norm of $\mathbf{d}$, denoted by $\mathcal{N}_{\mathbf{d}}$ (that corresponds to $\mathcal{N}_{\mathbf{d}} = \lceil \log_2(\|\mathbf{d}\|) \rceil$) that are the input of the parameter selection algorithm. To allow the comparison with the strategy of gate bootstrapping, we also included the set of parameters hardcoded in `tfhe-rs` to evaluate boolean operators. Table 4.4 shows the complexity of the cryptographic primitives expressed in PBS with our framework. It can be compared with Table 4.5, that illustrates the number of PBS required with the naive strategy of gate bootstrapping. Finally, Table 4.6 shows the concrete performance achieved by our implementations on our machine, as well as the comparison with other works and with the gate bootstrapping. For more information about an implementation or a comparison, the reader is referred to the related section.

Section	Primitive	Number of logic gates
4.7.1	One round of SIMON-128	256
	One full run of SIMON-128	17408
4.7.2	One round of Trivium	13
	One warm-up phase of Trivium (∗)	14976
4.7.3	One round of Keccak	7687
	A full Keccak permutation (∗)	184488
4.7.4	One evaluation of Ascon's S-box	16
	One full Ascon hashing run (∗)	19968
4.7.5	One evaluation of the AES S-box	115 ([BP10])
	A full run of `AES-128`	23360 ([BP10], [Max19])

Table 4.5: Number of logic gates in the circuit of each primitive. This shows the heavy cost of the naive method of performing one bootstrapping per gate (except the `NOT` ones).

Primitive	Section or Other work	Performances
One full run of SIMON	Gate Bootstrapping	174 s
	[Ben+22] †	128 s
	Our work (Section 4.7.1)	10 s
One warm-up phase of Trivium (*)	Gate Bootstrapping	1498 s
	[BOS23] (estimation on our machine)	53 s
	Our work (Section 4.7.2)	32.8 s
One Full Keccak permutation (∗)	Gate Bootstrapping	30.7 min
	Our work (Section 4.7.3)	8.8 min
One Ascon hashing (∗)	Gate Bootstrapping	200s
	Our work (Section 4.7.4)	92 s
One full evaluation of AES-128 ($\epsilon = 2^{-23}$) on one thread	[GHS12] †	18 min
	[CLT14] †	5 min
	[Tra+23]	270 s
	Our work (Section 4.7.5)	103 s
One full evaluation of AES-128 ($\epsilon = 2^{-40}$) on one thread	Gate Bootstrapping	234 s
	Our work (Real implementation)	135 s
	Our work (Theoretical timing with two keys)	105 s

Table 4.6: Timings of evaluation of full primitives, and comparison with previous works when they exist. Like on Table 4.4, a star (∗) is added in the cells if our timing is not obtained from a full implementation but estimated from an implemented building block. Also, the security level of each implementation is $\lambda = 128$ and the default error probability is $\epsilon = 2^{-40}$. The concurrent works that do not indicates their ϵ are marked with †.

4.8 Conclusion

In this chapter, we presented a first application of our technique of using an odd plaintext modulus: by embedding ℓ bits into a prime field, it becomes possible to unlock the full potential of homomorphic addition and "pack" several bits into a ciphertext. We can then retrieve the result of any Boolean function $f : \mathbb{B}^\ell \mapsto \mathbb{B}$ by a unique bootstrapping. This approach scales much better than the conventional technique of using a LUT of size 2^ℓ.

We ended this chapter by presenting an homomorphic implementation of the AES scheme. Implementing the circuit of the S-box using our technique was by far the most challenging part of this design. This is not very surprising: this component has been designed to prevent cryptanalysis, which indirectly makes the circuit representation of this function unsuitable for this kind of Boolean-oriented approaches.

An interesting axis of improvement would be to use an *arithmetic* representation of the values, so the S-box would simply be evaluated as a LUT. But by fully committing to arithmetic representation, the linear part of the scheme (so the `ShiftRows`, `MixColumns`and `AddRoundKey`steps) becomes the new bottleneck, as we could not leverage the additive homomorphism of TFHE to evaluate XOR operations anymore. In the next chapter, we elaborate further on these ideas and construct a more efficient version of homomorphic AES by combining both Boolean and arithmetic representations.

Chapter

5 Accelerating Homomorphic AES Evaluation

The last example of application of the previous chapter was to evaluate homomorphically the AES cipher. But why would one want to perform such a computation in the first place?

There are two answers to this question. First, AES is a particularly interesting benchmark, as an example of a nontrivial algorithm which has eluded "practical" FHE execution performances for years. This is part of the reason why it will most likely be selected by NIST as a flagship reference in its upcoming call on threshold (homomorphic) cryptography [ST25]. Since 2023, the algorithm has thus been the subject of a renewed attention from the FHE community and has served as a playground to test advanced operators[Tra+23; Wei+23; BPR24; Wei+24]. AES is particularly interesting as a benchmark notably because of the tension between boolean- and byte-oriented operations within the algorithm.

There is also a more down-to-earth reason: evaluating symmetric ciphers using FHE is the core of a cryptographic technique called *transciphering*, which is a promising solution for solving the *ciphertext expansion* problem of FHE. In this protocol, the client first encrypts its data using a symmetric encryption scheme and sends both the encrypted data and (once and for all) the FHE-encrypted symmetric key to the server. Leveraging its encrypted-domain computing capabilities, the server can then decrypt the encrypted data *within the homomorphic domain*, ultimately producing homomorphic ciphertexts on which it can perform the requested calculations. With this trick, the amount of data uploaded by the client is drastically reduces, as symmetric ciphertexts are much lighter than homomorphic ones.

In this chapter, we introduce `Hippogryph`, the fastest homomorphic implementation of AES using TFHE at the time of writing. To construct it, we leveraged three main ideas:

- We generalized the p-encoding construction introduced in Chapter 4 to the arithmetic case.
- The LUT-oriented implementation of [Tra+23] is very efficient to evaluate the large S-box of AES, so we borrowed this idea as it was.
- We associated both previous techniques by developing a framework of conversion between Boolean and arithmetic representations.

The result of this work is doubly interesting: first, we manage to outperform the rest of the literature on homomorphic AES evaluation. Second, we develop a generic framework useful to resolve the recurring tension between Boolean and arithmetic representations within homomorphic circuits, which we believe is of independent interest.

We start this chapter by introducing the notion of transciphering in Section 5.1, which will be a recurring theme in the rest of this manuscript. Then, after some preliminaries on the AES cipher (Section 5.2), we introduce in Section 5.3 some advanced homomorphic operators from [Tra+23] that will be useful in this work. We then generalize the p-encoding construction of Chapter 4 to the arithmetic case (Section 5.4). Finally, Section 5.5 introduces our new design and Section 5.6 presents a detailed comparison with existing approaches, supported by relevant benchmarks.

Table 5.1: State-of-the-art single-core *homomorphic evaluation of AES. The table indicates both the original timings, in seconds, provided in the papers and, in brackets, the timings obtained on our single machine test bench (a 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop).*

Year	Reference	Method	Timings
2023	[Tra+23]	Tree-Based Method (TBM)	270 (270) s
	[BPR24] (Chapter 4)	p-encoding method	135 (90) s
	[Wei+23]	TFHE in "LHE" mode	86 (87) s
2024	[Wei+24]	TFHE in "LHE" mode	46 (60) s
2025	This work	Combined Tree-Based Method (or Tree-Based Bootstrapping). See Section 5.3.2 (TBM)/p-encodings	32 s

5.1 Introduction to Transciphering

One common challenge with all FHE schemes is that the ciphertexts are much larger than the corresponding plaintexts. For example, a plaintext message of a few kilobytes can require tens or even hundreds of megabytes of data, making the processing of large data sets impractical. While compression techniques can help reduce the expansion factor in TFHE ciphertexts, the encrypted data still remains an order or two of magnitude larger than the original plaintext.

It is possible to mitigate this issue using *transciphering* [NLV11]. The idea is to off-load the task of actually encrypting the data to a symmetric cipher, and to simply encrypt homomorphically the key that is used. The user then sends both the homomorphically encrypted key and the ciphertext to the server, which can then homomorphically decrypt the received ciphertexts. This is done by running a fully homomorphic evaluation of the decryption function of the symmetric cipher, producing valid homomorphic ciphertexts representing the data. The server can then proceed to the homomorphic operations, like in the traditional FHE setting. This principle is illustrated on Figure 5.1

Implementing FHE encryption through transciphering solves the bandwidth issue: the data sent by the client to the server is encrypted using a symmetric cipher, thus avoiding the significant ciphertext expansion implied by direct FHE encryption. The only exception is the symmetric key, which does experience expansion, but this overhead is amortized across the entire data set.

But still, a question remains: which symmetric cipher should we use ? The straightforward solution is to simply pick something very standard, such as AES, and turn it into a homomorphic version. This is what the first works on transciphering tried to do: the first attempt to transcipher AES ciphertexts into FHE data was made in 2012 by Gentry, Halevi, and Smart [GHS12]. They used the BGV scheme [BGV12], a fully homomorphic encryption method based on the Ring-LWE problem, as implemented in HElib [HS20], an open-source library for FHE. However, their implementation resulted in an execution latency of 17.5 minutes, with now obsolete parameters (despite an amortized cost of 5.8 seconds per block).

Since then, some progress have been made. We give a tour of the current litterature on the topic in Section 5.6. But still, it seems that fast data transmission with AES will remain impractical, because its design is not adapted at all to homomorphic evaluation.

That is why many researchers have since developed new "FHE-friendly" symmetric cryptosystems to improve efficiency. Several proposal exists, including block ciphers such as LowMC [Alb+15], PRINCE [Bor+12], and CHAGHRI [AMT22], as well as stream ciphers like Elisabeth [Cos+22], PASTA [Dob+23], Kreyvium [Can+16] and Transistor [Bau+25]. These new schemes, sometimes referred to as hybrid encryption schemes, offer faster and more efficient homomorphic execution, though none have yet been standardized.

Still, homomorphic AES remains an active line of research in the FHE community. In 2022, the National Institute of Standards and Technology (NIST) announced a future call for threshold

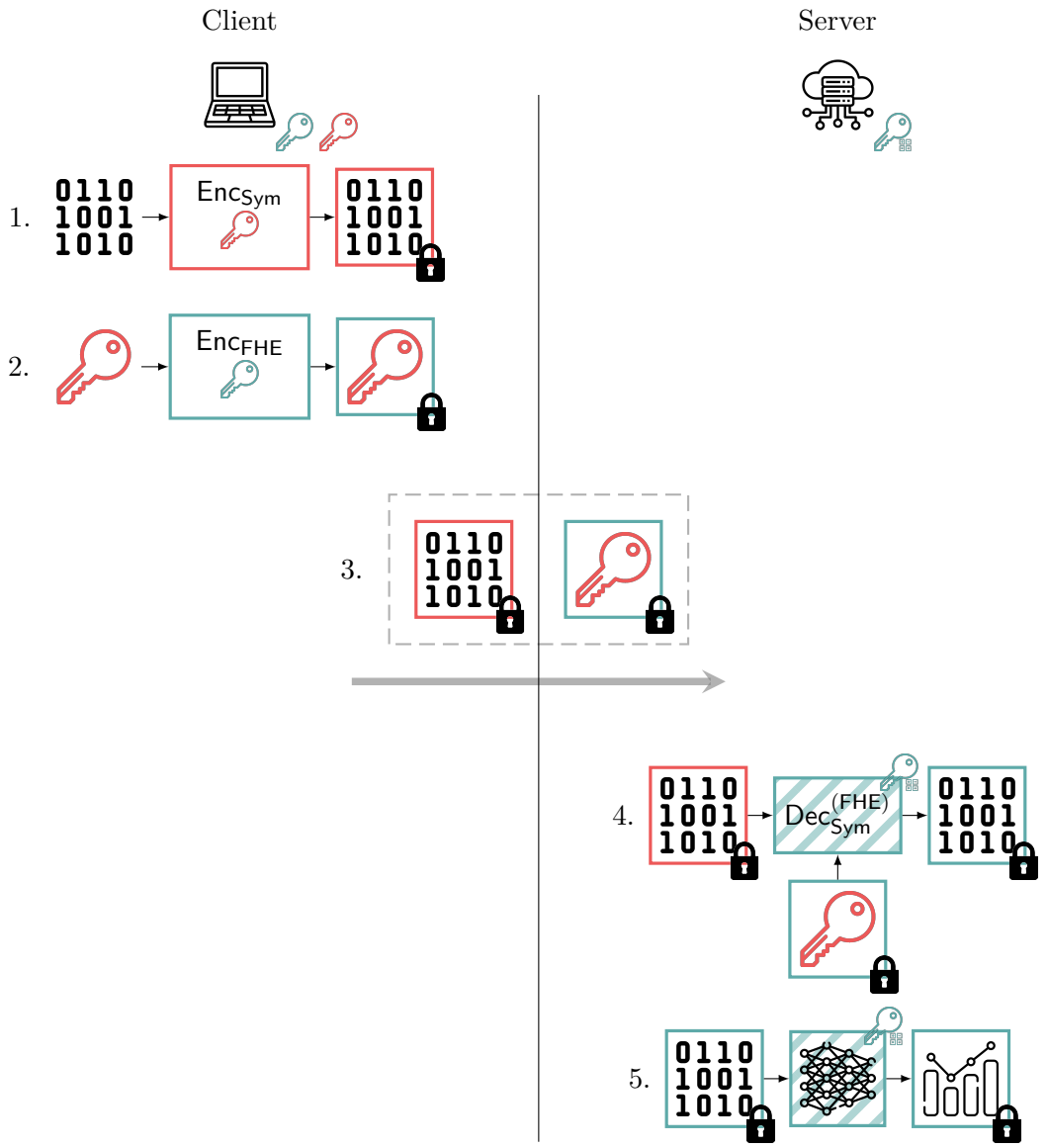


Figure 5.1: An illustration of the transciphering process:

1. *Data is encrypted with the symmetric cipher.*
2. *The symmetric key is homomorphically encrypted.*
3. *The client uploads both the encrypted data, as well as the encrypted key.*
4. *The server homomorphically evaluates the decryption algorithm of the symmetric cipher, using the symmetric key homomorphically encrypted. It retrieves the data, as valid homomorphic ciphertexts.*
5. *The server can then evaluate the homomorphic application!*

The red color is for symmetric algorithms, green for FHE, and hatched algorithms means that they are evaluated homomorphically.

encryption with a specific focus on FHE, indicating that AES would serve as the benchmark for evaluating proposals [ST25]. It particularly exemplifies the challenge of switching between boolean- and byte-oriented operations, a recurrent issue in TFHE-based implementations.

5.2 Preliminaries on AES

The Advanced Encryption Standard (AES), based on the Rijndael algorithm winner of the NIST competition in 2000 [DR00], is a symmetric block cipher supporting key sizes of 128, 192, and 256 bits. Depending on the key size, AES uses 10, 12, or 14 rounds of processing, each applying a fixed sequence of substitution, permutation, and mixing steps to transform plaintext into ciphertext (or ciphertext into plaintext for decryption). A key schedule generates round keys for each encryption round, plus an initial key.

`Hippogryph` focuses on AES with 128-bit keys, which uses 10 rounds. The encryption begins with an `AddRoundKey`step, followed by 10 rounds. Each round includes four steps: `SubBytes`, `ShiftRows`, `MixColumns`, and `AddRoundKey`, except the final round, which omits `MixColumns`. Below, we recall the key expansion and the subroutines:

- `Key Expansion`: The `Key Expansion` operation is performed once for a given secret key. Starting from the 128-bit key (in our context), it generates eleven 128-bit round keys, which are then used in the `AddRoundKey`operation throughout the AES encryption or decryption process, without needing access to the original key. The key expansion involves XORs and $\mathbb{F}_{256}$ multiplications.

- `SubBytes`: The `SubBytes`operation is the only non-linear transformation in the cipher. It involves a substitution step, where each byte in the state matrix is replaced according to a fixed S-box. Since it operates independently on each byte of the state, `SubBytes`can be easily parallelized, allowing for more efficient execution.

- `AddRoundKey`: During this transformation, the state is updated by combining it with the current round key using a bitwise XOR operation. Specifically, the 128-bit round key is organized into a matrix format to align with the structure of the state matrix, and the two matrices are XORed element-wise to produce the new state.

- `ShiftRows`: The `ShiftRows`step is a byte transposition that cyclically shifts the rows of the state by different offsets. For AES with 128-bit keys, the first row remains unchanged, the second row is shifted by one byte, the third by two bytes, and the fourth row by three bytes.

- `MixColumns`: The `MixColumns`step processes the state column by column through matrix multiplication. To compute each byte of the state matrix, they combine scalar multiplication in `GF`(256) with XOR operations. This approach facilitates parallelization of the operation.

5.3 Some Building Blocks for LUT-based Evaluation

In this section, we present the approach from [Tra+23], some components of which we use for our own work. We formally present some advanced homomorphic primitives used in this work that we reuse as well.

[Tra+23] is a "Full-LUT" approach, that is to say AES is evaluated entirely with TFHE's programmable bootstrapping, encoding exclusively all operations within LUTs. To meet the performance constraints of the bootstrapping algorithm, this method operates on elements in $\mathbb{Z}_{16}$, ensuring efficient computation.

5.3.1 AES Subroutines as LUTs

The `SubBytes`step, which involves the evaluation of an S-box, is inherently a LUT operation and is therefore naturally implemented in FHE using a PBS. However, PBS is too slow in $\mathbb{F}_{256}$ (as we have seen in Section 2.8). So, they rely on a construction evaluating PBS over $\mathbb{Z}_{16}$ rather than $\mathbb{F}_{256}$. Moreover, converting the other AES steps into LUT evaluations also requires additional effort.

In particular, in the original AES design [DR00], the `MixColumns`step is computed using a series of XOR operations and multiplications in $\mathbb{F}_{256}$. Unfortunately, TFHE's native multiplication `ClearMultTFHE` cannot directly handle these $\mathbb{F}_{256}$ multiplications because of the polynomial nature of the elements of this field. As a result, `MixColumns`must be reformulated as a LUT evaluation.

Additionally, the `AddRoundKey`step, which uses XOR as its key operation, presents its own challenges because XOR is a bivariate operation that requires two inputs. Classical bootstrapping, which operates on single inputs, is insufficient for this purpose. To address this, the authors utilize a specialized bootstrapping method that supports operations on multiple encrypted inputs.

5.3.2 LUTs Evaluation

Since the AES evaluation involves computing an 8-bit S-box, a straightforward solution would be to work with 8-bit messages. With such messages, the homomorphic S-box evaluation would require only one bootstrapping per byte. However, processing messages with more bits significantly slow down the bootstrapping process. To address this issue, [Tra+23] proposes a decomposition approach and demonstrates that the optimal representation of 8-bit inputs for their purpose is in $\mathbb{Z}_{16}$. Specifically, a message $m \in \{0, \cdots, 255\}$ is split into two 4-bit chunks (or *nibbles*) h and l such that $m = 16h + l$. The encryption of m is then represented as two ciphertexts encrypting h and l with the same key $\mathbf{s}$.

However, bootstrapping these decomposed inputs requires a method capable of handling multiple encrypted inputs. The authors explore several approaches for this, namely the chain-based method and the tree-based method [GBA21]. Their analysis concludes that the Tree-Based Method (TBM) is the most suitable for their needs. They also relies on the Multi-Value Bootstrapping. See Section 5.3.2 (MVB) to produce several outputs for the cost of one PBS. We provide details about TBM and MVB in the following:

Multi-Value Bootstrapping from [CIM19]. Multi-Value Bootstrapping (MVB) is a technique that enables the evaluation of k distinct Look-Up Tables $(f_i)_{1 \leq i \leq k}$ on a single encrypted input, using only one `BlindRotate`. This method is based on the factorization of the accumulator polynomials $\mathsf{acc}_i(X)$ associated with each function f_i. Specifically, each accumulator polynomial is expressed as:

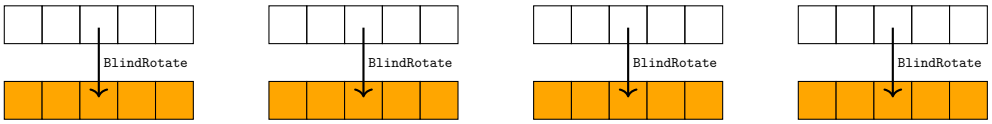
$$\mathsf{acc}_i(X) = \sum_{j=0}^{N-1} \alpha_{i,j} X^j, \quad \alpha_{i,j} \in \mathbb{Z}_q.$$

The factorization then splits it into two parts:

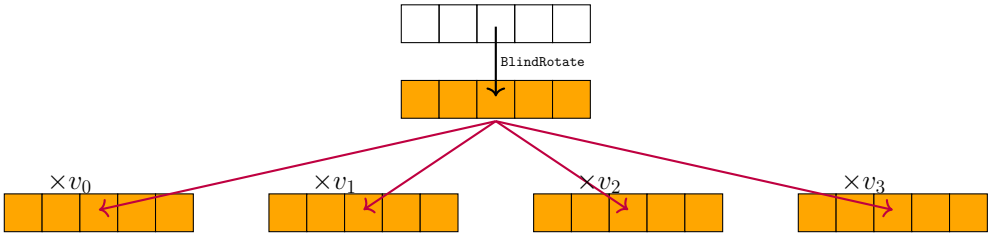
$$\mathsf{acc}_i(X) = v_0(X) \cdot v_i(X) \mod (X^N + 1),$$

where $v_0(X)$ is a common factor shared across all accumulators set as:

$$v_0(X) = \frac{1}{2} \cdot (1 + X + \cdots + X^{N-1}),$$



(a) Classic bootstrapping method to evaluate several LUTs on a single input



(b) MVBmethod

Figure 5.2: Difference between the classical approach and the MVB. Pink arrows represent `clearMultTFHE` *operations on GLWE ciphertexts (Figure inspired by [Tra+23]).*

and $v_i(X)$ is a distinct factor specific to each function f_i:

$$v_i(X) = \alpha_{i,0} + \alpha_{i,N-1} + (\alpha_{i,1} - \alpha_{i,0}) \cdot X + \cdots + (\alpha_{i,N-1} - \alpha_{i,N-2}) \cdot X^{N-1}.$$

This factorization is made possible thanks to the identity:

$$(1 + X + \cdots + X^{N-1}) \cdot (1 - X) \equiv 2 \mod (X^N + 1).$$

By leveraging this factorization and as illustrated on Figure 5.2, multiple LUTs can be evaluated on a single encrypted input by performing the following steps:

1. Computing a `BlindRotate` operation on an accumulator polynomial initialized with the value of v_0.
2. Then multiplying with `ClearMultTFHE` the obtained rotated polynomial by each $v_i(X)$ corresponding to the LUT of f_i to obtain the respective $\mathsf{acc}_i(X)$.

Finally, at the cost of a single `BlindRotate` and k `clearMultTFHE` operations (on GLWE), one can obtain the evaluation of k different LUTs on one single encrypted input. Moreover, this specific choice of factorization allows for a very-low norm for the vectors v_i's (which in practice are very sparse), and so a very-low noise expansion.

This MVBprimitive thus allows significant speed-ups in the implementation of [Tra+23], in particular in the evaluation of the S-box or in the multiplications in $\mathbb{F}_{256}$ that occur during the `MixColumns`step. Indeed, since each byte is decomposed into two nibbles h and l, the LUT corresponding, for instance, to the S-box must also be decomposed into two tables: one providing the most significant nibble and one providing the least significant nibble. That is to say:

$$\mathtt{tab}_{\mathrm{MSN}}[i] = \left\lfloor \frac{\mathtt{S\text{-}box}[i]}{16} \right\rfloor \quad \text{and} \quad \mathtt{tab}_{\mathrm{LSN}}[i] = [\mathtt{S\text{-}box}[i]]_{16}\,.$$

Each of these tables must be evaluated on an 8-bit payload ciphertext.

Tree-Based Method from [Tra+23]. Let $B, B', d \in \mathbb{N}^*$. The Tree-Based Method (TBM) allows to evaluate a LUT $f : \mathbb{Z}_{B^d} \mapsto \mathbb{Z}_{B'}$ with a large input size B^d, by processing d limbs of data in $\mathbb{Z}_B$. We consider input messages that are written as:

$$m = \sum_{i=0}^{d-1} m_i B^i, \quad \text{with } m_i \in \mathbb{Z}_B,$$

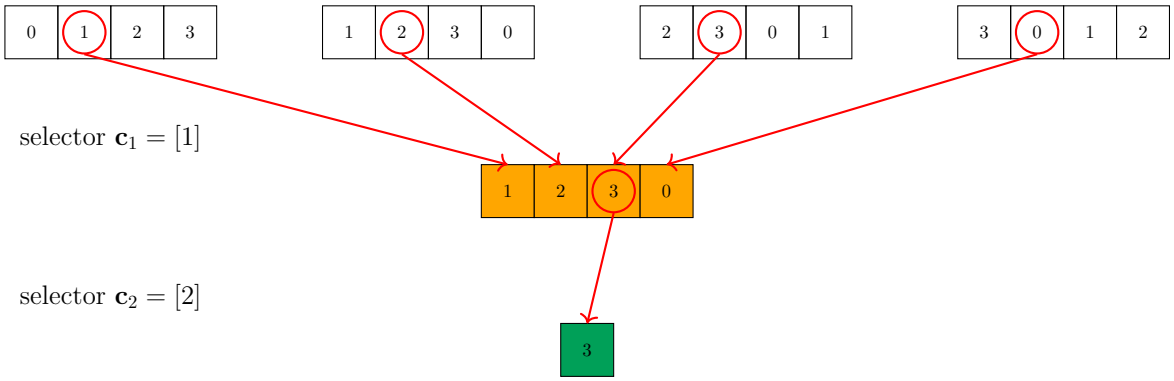


Figure 5.3: Illustration of the tree-based method on messages $m_1 = 1, m_2 = 2$ in the space $\mathbb{Z}_4$. The corresponding ciphertexts are $\mathbf{c}_1 \in \mathsf{LWE}(m_1)$ and $\mathbf{c}_2 \in \mathsf{LWE}(m_2)$. We apply the addition in $\mathbb{Z}_4$ via programmable bootstrapping. Red arrows indicate bootstrappings. (Figure inspired by [Tra+23].)

and that are represented by d ciphertexts $(\mathbf{c}_0, \mathbf{c}_1, \ldots, \mathbf{c}_{d-1})$ corresponding to the d message components $(m_0, m_1, \ldots, m_{d-1})$. To evaluate f, we encode a LUT for f using B^{d-1} accumulators, each represented by a polynomial $\mathsf{acc}_i(X)$. These accumulators encode the functions:

$$\begin{aligned} f_i : \mathbb{Z}_B &\to \mathbb{Z}_{B'} \\ x &\mapsto f(i + x \cdot B^{d-1}) \end{aligned}$$

Next, we apply a BlindRotate and a SampleExtract to each accumulator $\mathsf{acc}_i(X)$, using $\mathbf{c}_{d-1}$ as the selector. This operation produces B^{d-1} LWE ciphertexts, each encrypting $f(i + m_{d-1} \cdot B^{d-1})$ for $i \in \mathbb{Z}_{B^{d-1}}$. Finally, a Keyswitch operation from LWE to GLWE aggregates these ciphertexts into B^{d-2} GLWE encryptions, representing the LUT of h, defined as:

$$\begin{aligned} h : (\mathbb{Z}_B)^{d-1} &\mapsto \mathbb{Z}'_B \\ (u_0, \ldots, u_{d-1}) &\mapsto f \circ g(u_0, \ldots, u_{d-2}, m_{d-1}) \end{aligned}$$

using the bijection g, which reverses the decomposition:

$$\begin{aligned} g : (\mathbb{Z}_B)^d &\to \mathbb{Z}_{B^d} \\ (u_0, \ldots, u_{d-1}) &\mapsto \sum_{i=0}^{d-1} u_i \cdot B^i \end{aligned}$$

This process is repeated iteratively, using the next ciphertext at each step, until a single LWE ciphertext encrypting $f(m_0, \ldots, m_{d-1})$ is obtained.

In the implementation described in [Tra+23], this primitive is employed to evaluate an 8-bit LUT by dividing it into two limbs of 4 bits each, which they determined to be optimal for their specific setting. To further enhance the performance of the TBM, the blind rotations for the accumulators $acc_i(X)$ of *the first layer of the tree* can be performed simultaneously using the MVBtechnique (as discussed in [GBA21]).

Finally, the "full-LUT" approach facilitates efficient computation of the S-box through the Tree-Based Method, as opposed to directly evaluating the corresponding Boolean circuit. However, this approach also requires LUT-based computation of XOR operations and other intermediary steps, which is notably slower when operating in $\mathbb{Z}_{16}$ compared to binary messages. Consequently, our new method `Hippogryph` proposed in this chapter strategically applies LUT evaluation exclusively where it is most effective and yields the best performance, namely for the evaluation of the S-box.

5.4 Generalization of p-encodings to the Arithmetic Case

In Chapter 4 of this thesis, we introduced the notion of p-encodings and used it in Section 4.7.5 to evaluate AES homomorphically. In this method, data was encrypted bit per bit and only Boolean operations were performed. It leveraged the fact that, in the plaintext space $\mathbb{Z}_2$, the `SumTFHE` operation actually performs a XOR. Thus, the linear operations `MixColumns`and `AddRoundKey`could be efficiently performed with minimal cost, using only the homomorphic sum of TFHE. To be able to evaluate `SubBytes` under Boolean representation, we used p-encodings to evaluate the circuits of `SubBytes`with a minimal number of bootstrappings. While this has brought some improvements, evaluating the LUT of AES as a Boolean circuit is still suboptimal, and in this chapter we attempt at doing it using arithmetic representation.

To achieve that, we generalize p-encodings beyond the Boolean case by defining the (o, p)-encoding construction. Informally, instead of embedding the Boolean space in $\mathbb{Z}_p$, we embed any space $\mathbb{Z}_o$ in $\mathbb{Z}_p$ (with $o < p$). So, what was called p-encoding in Chapter 4 corresponds to a $(2, p)$-encoding in this one. Definition 5.4.1 formalizes this generalization.

Definition 5.4.1 ((o, p)-encoding)**.** Let $\mathbb{Z}_o$ be the message space. A (o, p)-*encoding* is a function $\mathcal{E} : \mathbb{Z}_o \mapsto 2^{\mathbb{Z}_p}$ that maps each element of $\mathbb{Z}_o$ to a subset of the discretized torus $\mathbb{Z}_p$. A (o, p)-encoding is *valid* if and only if:

$$\begin{cases} \forall (i, j) \in \mathbb{Z}_o^2, i \neq j, \mathcal{E}(i) \cap \mathcal{E}(j) = \emptyset \text{ and} \\ \text{if } p \text{ is even: } \forall\, x \in \mathbb{Z}_p, \forall i \in \mathbb{Z}_o : x \in \mathcal{E}(i) \iff \left[x + \frac{p}{2}\right]_p \in \mathcal{E}([-i]_o) \end{cases} \tag{5.1}$$

The latter property is a direct consequence of the negacyclicity problem, which we discussed extensively in Chapter 3.

In this work, we focus exclusively on cases where $p = 2$ or p is an odd prime. As a result, a lot of the subleties of negacyclicity can be overlooked. Furthermore, among the various types of (o, p)-encodings, one particular class proves especially useful for our purposes: the *canonical* (o, p)-encoding.

Definition 5.4.2 (canonical (o, p)-encoding)**.** A (o, p)-encoding $\mathcal{E}$ is said *canonical* if and only if it verifies:

$$\begin{aligned} \mathcal{E} : \mathbb{Z}_o &\to \mathbb{Z}_p \\ x &\mapsto x \end{aligned}$$

(with $o < p$). Informally, we simply embed a smaller space into a larger one, without altering the order of the elements.

In Chapter 4 the Boolean space is used (so $o = 2$). The `SubBytes`circuit is evaluated using $(2, 11)$-encoding, while the rest is evaluated with a $(2, 2)$-encoding (*i.e.,* the trivial encoding of TFHE with plaintext space $\mathbb{Z}_2$). Consequently, an *Encoding Switching* operation is required. This operation can be straightforwardly performed using a PBS.

Definition 5.4.3 (Encoding Switching)**.** Let $\mathbf{c}$ be a ciphertext encrypting a message $m \in \mathbb{Z}_o$ under the (o, p)-encoding $\mathcal{E}$. Its encoding can be switched to the (o, p')-encoding $\mathcal{E}'$ by applying a PBS on $\mathbf{c}$ evaluating the function:

$$\begin{aligned} \mathtt{Cast}_{\mathcal{E} \mapsto \mathcal{E}'} : \mathbb{Z}_p &\to \mathbb{Z}_{p'} \\ x &\mapsto x' \end{aligned}$$

where x' is defined as $\forall i \in \mathbb{Z}_o, x \in \mathcal{E}(i) \implies x' \in \mathcal{E}'(i)$.

5.5 Design of Hippogryph

Building on the primitives presented in Section 5.3 and 5.4, we develop a hybrid approach, `Hippogryph`, that not only combines their respective strengths but also introduces new contributions to enable their effective integration. The guiding principles of this design are outlined below:

- The `SubBytes`step, which was the weak point of our implementation in Chapter 4, is evaluated using the strategy of [Tra+23].
- Conversely, the linear steps (namely `ShiftRows`, `MixColumns`and `AddRoundKey`) are computed using a trivial $(2, 2)$-encoding, which makes them extremely fast.
- Since the two aforementioned points rely on different data representations (arithmetic for `SubBytes`and Boolean for the other steps), a decomposition layer and a recomposition layer are necessary to transition from one to another. The decomposition and recomposition steps are denoted by `Decomposer` and `Recomposer`, respectively.

Our design for one round of AES is summed up on 5.4. In the following we explain each of its components.

`SubBytes`. The `SubBytes`step is implemented following the design of [Tra+23]. Each 8-bit input is represented by two ciphertexts, each encrypting a 4-bit limb. Two instances of the TBM are then used to compute the limbs of the output. The only modification from the design of [Tra+23] is the adoption of the canonical $(16, 17)$-encoding, as specified in Definition 5.4.2:

$$\begin{aligned} \mathcal{E}_{17} : \mathbb{Z}_{16} &\to \mathbb{Z}_{17} \\ i &\mapsto i. \end{aligned}$$

This modification is introduced to ensure compatibility with the `Recomposer` operation, a point which will be explained in the dedicated paragraph. In Figure 5.4, ciphertexts encrypted under this $(16, 17)$-encoding are represented by blue rectangles. This process is repeated 16 times, once for each byte of the AES state. An additional improvement comes from the fact that the two TBM are using a MVBto evaluate the first step. So, the same common factor can be used for both evaluations, requiring only one `BlindRotate` per byte for this first step.

Linear Circuit. For this part, we follow the design of Chapter 4. The ciphertexts manipulated in this block are encoded under the trivial $(2, 2)$-encoding $\mathcal{E}_2$, and encrypt a single bit each. They are represented by yellow squares on 5.4. Consequently, this circuit takes 256 inputs (one for each of the 128 bits in an AES block, and one for each of the 128 bits in the current round key), and outputs a new state of 128 bits, by combining the three following steps:

- `ShiftRows`: This step is trivially implemented in FHE by permuting the input ciphertexts according to the AES specifications.
- `MixColumns`: Here, we use the XOR-only circuit representation of [Max19]. Evaluating a XOR on ciphertexts under $\mathcal{E}_2$ is simply done using the native addition of TFHE `SumTFHE`.
- `AddRoundKey`: This step is a simple XOR between the state and the round key.

Evaluating the sums within this circuit increases the noise in the ciphertexts. However, this problem can actually be overlooked: using $p = 2$, there is plenty of room for the noise to grow, so the bottleneck of the construction in terms of noise is actually the intermediate layer inside the TBM in $\mathbb{Z}_{17}$. In our experimentations, we made sure to select parameters ensuring correctness up to the target probability of success.

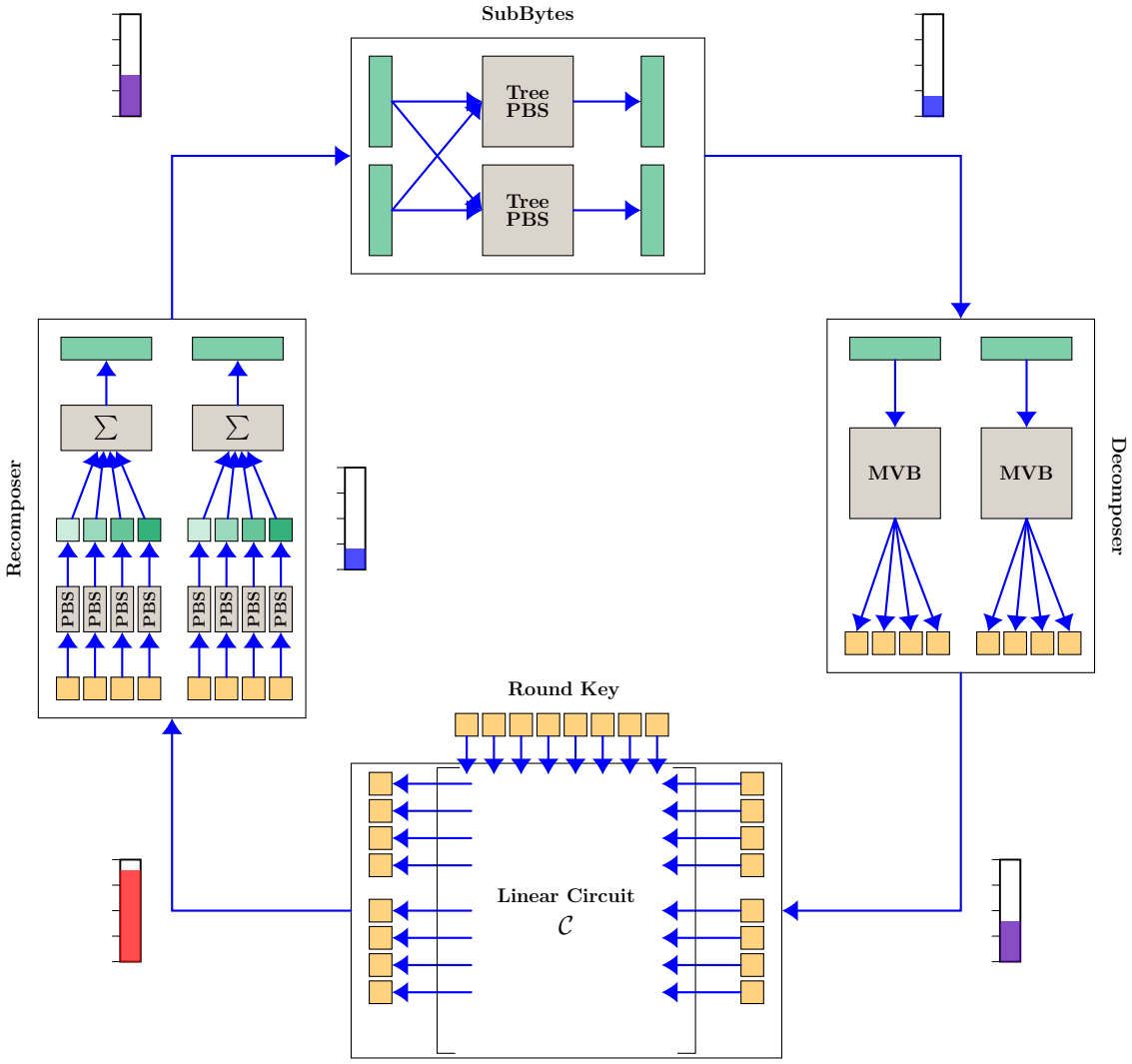


Figure 5.4: Structure of one round of AES with our method. Ciphertexts in blue live in $\mathbb{Z}_{17}$ while the ones in yellow are in $\mathbb{Z}_2$. Squares represent encryptions of one single bit while rectangles represent nibbles. Indicative noise levels at different spots of the circuit are indicated by the gauges.

Decomposer. From the `SubBytes`step to the linear circuit steps, a switch of representation is needed at two levels. First, we need to decompose each ciphertext of a 4-bit limb into 4 ciphertexts each encrypting a single bit. Secondly, we need to switch the encoding from $\mathcal{E}_{17}$ to $\mathcal{E}_2$. Fortunately, by combining the MVBprimitive and the encoding switching primitive (from Definition 5.4.3), it is possible to do both changes at once for each nibble with a single PBS. Formally, the MVBwill evaluate the four functions:

$$\begin{aligned}\forall i \in \{0,\dots,3\}, f_i : \mathbb{Z}_{17} &\to \mathbb{Z}_2 \\ x &\mapsto \mathcal{E}_2((\mathcal{E}_{17}^{-1}(x))_i)\end{aligned}$$

where $(y)_i$ refers to the extraction of the i-th bit of y.

Recomposer. Conversely, a transformation from the Boolean domain to the arithmetic domain is required. As in the `Decomposer` operation, this involves two key steps:

- Casting the ciphertexts from a plaintext modulus of 2 to 17.
- Recombining each group of 4 bits into a single ciphertext encrypting the whole nibble.

To achieve this efficiently, we introduce four intermediary $(2, 17)$-encodings, namely:

$$\begin{aligned}\forall i \in \{0,\dots,3\}, \mathcal{E}_{17}^{(i)} : \mathbb{Z}_2 &\to \mathbb{Z}_{17} \\ x &\mapsto \begin{cases} 0 \text{ if } x = 0 \\ 2^{i+1} \text{ if } x = 1. \end{cases}\end{aligned} .$$

Using little-endian representation, we perform an encoding switching (5.4.3) on the i-th bit of each nibble, transitioning from $\mathcal{E}_2$ to $\mathcal{E}_{17}^{(i)}$. In Figure 5.4, the resulting ciphertexts are representing by squares filled with different shades of blue. Once the bits are expressed in this intermediary representation, we simply sum them to reconstruct the result in $\mathcal{E}_{17}$.

Necessity of an odd modulo in `SubBytes`: The inputs to the `Recomposer` are encrypted modulo 2. Since no padding bits are used, the negacyclicity problem necessitates that the PBS in the `Recomposer` evaluates a negacyclic function. As stated in Property 5.5.1, the existence of a Boolean recomposition algorithm relying solely on PBS and linear operations depends on the parity of the output plaintext modulus.

Property 5.5.1. *A `Recomposer` using only linear operations and one PBS per bit exists only if the output modulo is **odd**.*

Proof. Let p be an integer. Let $(b_0, \dots, b_{d-1})$ be the bits to encrypt, and let $(\mathbf{c}_0, \dots, \mathbf{c}_{d-1})$ denote their corresponding ciphertexts, encoded with the trivial $(2, 2)$-encoding $\mathcal{E}_2$. We aim to construct a `Recomposer` that uses only one programmable bootstrapping (PBS) per bit and linear operations to homomorphically compute an encryption of the message $m = \sum_{i=0}^{d-1} b_i 2^i$ under the canonical $(2^d, p)$-encoding $\mathcal{E}_p$. The purpose of this proof is to demonstrate how the parity of p influences the existence of such an algorithm.

To do so, following the blueprint introduced earlier in the section, we want to bootstrap the ciphertext $\mathbf{c}_i$ into $\mathbb{Z}_p$ with the p-encoding $\mathcal{E}_p^{(i)} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{0\} \\ 1 \mapsto \{2^{i+1}\} \end{cases}$. Once we have those, a simple sum will reconstruct the message under the canonical $(2^d, p)$-encoding. Let us analyze if this bootstrapping is possible.

As the ciphertexts are encrypted modulo 2, there is no bit of padding. So, if we send them modulo p with a PBS, the result will necessarily be encoded under a negacyclic $(2, p)$-encoding, that is to say of the form: $\mathcal{E}^{(\text{neg})} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{\gamma\} \\ 1 \mapsto \{[-\gamma]_p\} \end{cases}$ with $\gamma \in \mathbb{Z}_p$.

Now, we need a linear transformation that casts a ciphertext from $\mathcal{E}^{(\text{neg})}$ to $\mathcal{E}_p^{(i)}$. Let us denote this hypothetical linear transformation by $\mathcal{L}$, and define it as:

$$\begin{aligned} \mathcal{L} : \mathbb{Z}_p &\mapsto \mathbb{Z}_p \\ x &\mapsto a \cdot x + b \end{aligned}$$

By simply considering the encoding switching from $\mathcal{E}^{(\text{neg})}$ to $\mathcal{E}_p^{(0)}$, it is clear that the constants a and b need to verify the property:

$$\begin{cases} a \cdot \gamma + b = 0 \mod p \\ a \cdot (-\gamma) + b = 1 \mod p \end{cases}$$

which can be rewritten as:

$$\begin{cases} b = 2^{-1} \mod p \\ \gamma = (b-1) \cdot a^{-1} \mod p \end{cases}$$

It is clear that such a b only exists if and only if 2 has an inverse modulo p. This latter argument forces p to be odd. In that case, fixing a to 1, the $(2, p)$-encoding

$$\mathcal{E}^{(neg)} = \begin{cases} \mathbb{Z}_2 \mapsto \mathbb{Z}_p \\ 0 \mapsto \{[2^{-1} - 1]_p\} \\ 1 \mapsto \{[1 - 2^{-1}]_p\} \end{cases}$$

is supposed to be what we are looking for.

Let us check if that is the case. As it is negacyclic, the PBS is evaluable. Then, the linear transformation $x \mapsto x + 2^{-1} \mod p$ produces a ciphertext under the right p-encoding. Trivially, adding a constant to a TFHE ciphertext do not increase its noise. The same reasoning can be followed for the others bits.

Finally, summing the produced ciphertexts gives an encryption of m under $\mathcal{E}_p$. The whole procedure is only possible if p is odd. □

In [Tra+23], the authors determined that the most time-efficient way to slice the 8-bit inputs of the S-box for the tree-based method is into two 4-bit chunks. Given that our Recomposer block requires an odd plaintext modulus, as established in Property 5.5.1, we selected the smallest odd modulus capable of representing 4-bit values: $p = 17$.

Key Expansion. To the best of our knowledge, no previous work on AES transciphering has performed the key expansion phase in the homomorphic domain. Similarly, we work under the assumption that FHE encryptions of the eleven AES round keys are directly available. Since the round keys need to be computed only once for a given secret key, this makes sense in a client-server setting as the client is then expected to compute the key expansion and to send encryptions of the resulting round keys (rather than sending an encryption of the secret key under the homomorphic scheme).

5.6 Experimental Results

In this section, we compare our new framework to several state-of-the-art homomorphic AES executions, including the ones performed with the two building blocks of our new design. We particularly emphasize that *all implementations have been tested on the same machine*, a 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop with 64 Gib total system memory with an Ubuntu 22.04.2 LTS operating system. All execution timings can be found in Table 5.3. Parameter sets used to obtain these results are presented in Table 5.2. Depending on the framework, we had to use different implementations of TFHE as available in the `TFHElib`[1], `tfhe-rs`[2] or `TFHEpp`[3] libraries.

Table 5.2: Parameters sets used for our homomorphic AES evaluation, with $\lambda \approx 128$ bits as the security parameter. p_{err} denotes the probability of bootstrapping failure. $\mathfrak{B}_{KS}$ and ℓ_{KS} denote the basis and levels associated with the gadget decomposition in `KeySwitch`, $\mathfrak{B}_{PBS}$ and ℓ_{PBS} denote the decomposition basis and the precision of the decomposition of `BlindRotate`. σ_{LWE} and σ_{GLWE} are the standard deviations of noises used in LWE and GLWE ciphertexts, respectively.

p_{err}	n	N	k	ℓ_{PBS}	$\mathfrak{B}_{\text{PBS}}$	$\mathfrak{B}_{\text{KS}}$	ℓ_{KS}	σ_{LWE}	σ_{GLWE}
2^{-40}	754	1024	1	2	2^{23}	2^{4}	3	$2^{46.4}$	$2^{16.7}$
2^{-128}	900	4096	1	2	2^{15}	2^{3}	6	$2^{44.5}$	2^{2}

Table 5.3: Comparison of our method with different state-of-the art approaches on a single core. *The only execution timing that was not obtained on our machine is marked with a *, i.e. for Thunderbird, making the comparison more in favor of that method. See the discussion at the end of Section 5.6.1.*

Year	Reference	Framework	Library	Timings (s)
2023	[Tra+23]	Tree-Based Method (TBM)	`TFHElib` [4]	270
	[BPR24], Chap. 4	p-encoding method	`tfhe-rs`[5]	90
	[Wei+23]	Fregata	`TFHEpp` [6]	87
2024	[Wei+24]	Thunderbird	`TFHEpp` [7]	46*
2025	this work	`Hippogryph`	`tfhe-rs`[8]	32

The implementation of `Hippogryph` and the unified software test bench for AES execution over TFHE, which we used to obtain the consistent same-machine experimental results presented in this paper, are available as open-source Git repositories[9].

5.6.1 State-Of-The-Art Homomorphic AES Executions

The approaches introduced in [Tra+23] and [BPR24], which form the foundation of our proposal, are discussed in 5.3. Additionally, we briefly describe the two other main state-of-the-art methods for homomorphic AES executions: Fregata [Wei+23] and Thunderbird [Wei+24].

Fregata [Wei+23]

In this work, the authors present a novel evaluation framework especially designed for faster AES homomorphic evaluation. Instead of relying on functional bootstrapping, they decided to use

[1] https://tfhe.github.io/tfhe/
[2] https://github.com/zama-ai/tfhe-rs
[3] https://github.com/virtualsecureplatform/TFHEpp
[9] https://github.com/CryptoExperts/Hippogryph and https://github.com/daphnetrm/Benchmark-of-\gls{AES}-Evaluation-with-\gls{TFHE}.

CMUX gate as the building block of their framework. They also propose a new technique for an efficient S-box evaluation relying on mixed packing (which combines different ways of organizing encrypted data within polynomials to balance parallelism and flexibility). But one of the major contributions of this work is the optimization of TFHE's circuit bootstrapping. Indeed, they propose to use PBSManyLUT [Chi+21] in the first step of circuit bootstrapping. As their framework relies on the use of TFHE in LHE mode, this optimization of circuit bootstrapping is the key to an efficient homomorphic AES evaluation. Fregata being designed to perform one round of AES without any bootstrapping and to use circuit bootstrapping on each bit of the state matrix after a full round evaluation, running these circuit bootstrappings then becomes the most time consuming part. Finally, they also leverage on encoding messages in $\{0, 1\}$ as $\{0, \frac{1}{2}\}$ over the torus to transform XOR operations into simple LWE sums (which is the same thing as using our $(2, 2)$-encoding in the linear parts).

Their results, obtained with the TFHEpp library [Mat20], reached an AES homomorphic evaluation latency of 86 seconds on a 12th Gen Intel(R) Core(TM) i5-12500× 12 with 15.3 GB RAM machine. When running the Fregata implementation[10] on our machine, we also obtained a latency of about 87 seconds.

Thunderbird [Wei+24]

The work presented in the Thunderbird paper leverages on the Fregata framework to produce an even faster AES homomorphic evaluation, still using TFHEpp. Specifically, Thunderbird combines the gate bootstrapping and leveled evaluation modes of TFHE to cater to various function types within symmetric encryption algorithms. More specifically, their approach builds upon the Fregata framework with additional optimizations:

- The circuit bootstrapping proposed in Fregata is optimized by replacing the second step (namely a private keyswitch) by a public keyswitch followed by a new operation called `EvalSquareMult`.

- Instead of following a standard AES implementation, the authors introduce a LUT-based AES implementation that merges SubBytes, ShiftRows and MixColumns operations into 8-to-32-bit tables (which results in a smaller number of XOR operations when running the overall AES).

Moreover, as in [Wei+23], they rely on encoding the messages in $\{0, 1\}$ as $\{0, \frac{1}{2}\}$ over the Torus. With such encoding, XOR operation can be performed for free. They call this optimization `FreeXOR`. They also propose another technique to evaluate XOR, namely `HomoXOR` relying on gate bootstrapping with messages encoded in $\{\frac{-1}{8}, \frac{1}{8}\}$ over the Torus. The evaluation of AES with this technique is less efficient than with `FreeXOR`. For this work, the tests were run on an Intel(R) Core(TM) i5-11500 CPU @ 2.70GHz machine with 32 GB of RAM and they obtained an average execution latency of 46 seconds.

It is important to note that the implementation of the Thunderbird framework is not publicly available. To obtain a fair comparison with our work, we tried to reproduce their results by implementing the framework ourselves, starting from Fregata on which Thunderbird is based. We successfully implemented all of Thunderbird building blocks but one–the improved circuit-bootstrapping–which was producing decryption errors despite our best efforts to faithfully follow [Wei+24]. As a consequence, for this specific building block, we relied on the theoretical speedup reported in the Thunderbird paper, which results in a slightly unfair comparison to our approach. In summary, *we measured an AES execution time of 60 secs with our implementation, but used instead the 46 secs reported in [Wei+24] for comparison.*

[10] https://github.com/WeiBenqiang/Fregata

5.6.2 Results

To measure the performances of our method, we implemented it using our fork of `tfhe-rs` [Zam22c], modified to support odd moduli (see Section 4.6.2). The results were then compared against the current state-of-the-art frameworks.

For a fair comparison, all implementations were tested on the same machine, *using a single core*. As shown in Table 5.3, our novel framework achieves the lowest latency when evaluating the AES as the evaluation of the algorithm only takes about 30 seconds. Hence, `Hippogryph` is between 1.44 and 1.87 times faster than the best-in-class Thunderbird approach (depending, as discussed above, whether we respectively consider the 46 secs timing given in the Thunderbird paper or a timing of 60 secs as measured with our implementation). Moreover, when enabling several cores on our 12th Gen Intel(R) Core(TM) i7-12700H CPU laptop, we can reach an execution time that is smaller than 5 seconds, using only 6 cores, and further reduce this timing to 1.1 seconds by using 32 cores on a more powerful machine as discussed below.

A Few Words About Parallelisation. The purpose of transciphering is to minimize the bandwidth overhead when transferring large amounts of data. Given that servers typically have more computational resources than clients, they can effectively leverage multiple cores to parallelize computations and enhance execution times. In this context, AES offers inherent parallelizability, as most operations within each encryption round can be executed concurrently on each byte of the state matrix—except for the ShiftRows step.

It is worth noting that in practical transciphering applications, one could process multiple AES blocks in parallel to achieve better amortized performance. However, the parallelization we apply here focuses on accelerating computations within a single AES block, rather than processing multiple blocks independently.

To implement this approach, we used Rust's `rayon` crate. Our tests were conducted tests on two distinct machines to assess performance across different setups:

- Laptop (12th Gen Intel(R) Core(TM) i7-12700H CPU, 6 cores): We parallelized all round functions except ShiftRows, which mainly involves reordering ciphertexts within the state matrix. This setup already provided a significant speedup compared to single-core execution: 4.6 seconds for a failure probability of 2^{-40}.

- Server (AMD Ryzen Threadripper PRO 7995WX, 96 cores): We leveraged 32 cores to process the 16 bytes of the AES internal state in parallel, each using one thread for each of the 2 independents TBM in the `SubBytes`step. This setup brought us remarkably close to breaking the 1-second barrier, with an execution time of just 1.1 seconds.

Detailed execution timings illustrating these improvements can be found in Table 5.4.

Memory-wise, our implementation manipulates at most 128 TFHE ciphertexts which account for the internal state and 128×11 other ciphertexts for the round keys. Each TFHE ciphertext accounts for around 48 kbits (755*64 bits). Hence, the total memory consumption of our implementation is approximately 74 Mbits when encrypting a single block. When encrypting multiple blocks in parallel, the ciphertexts for the round keys are shared across all blocks, reducing the per-block overhead.

Key Expansion: To the best of our knowledge, all previous works on AES transciphering do not perform the key expansion phase in the homomorphic domain. To ensure fair comparisons with related works, we made the same assumption. However, we do have a TFHE implementation of key expansion that has roughly the same structure as `Hippogryph`. Our measurements show that a run of `KeyExpansion`is approximately 20 % faster than the encryption/decryption.

What About Recent CPA$^{\mathbf{D}}$ Attacks? To obtain a fair comparison, we use parameters equivalent to those used in the state-of-the-art, that typically achieve an error probability of about 2^{-40}. But to take into account recent attacks in the CPA^D model [LM21] on several FHEs (including TFHE) [Che+24a; Che+24b], we also give execution times of our approach with a example parameters set achieving an error probability of 2^{-128} (Table 5.2). When running with such parameters, an AES evaluation takes about 463 seconds on our machine, still using a single core (see also Table 5.4). Although more optimal parameters may be found, this timing also illustrates that achieving CPA^D security may have a significant cost on FHE performances. At this point, we leave that cost mitigation as a future work.

Table 5.4: Different evaluation timings of `Hippogryph` *for different setups.*

Machine	# cores	p_{err}	Timings (s)
laptop	1	2^{-40}	32
laptop	1	2^{-128}	463
laptop	6	2^{-40}	4.6
server	32	2^{-40}	1.1

5.7 Conclusion

In this chapter, we combined the strengths of both Boolean and arithmetic representations to develop a framework enabling homomorphic conversion between the two. This approach proved effective: we managed to construct the fastest homomorphic implementation of AES currently available in the literature.

Although the focus here is primarily on AES, this work represents a first step toward resolving the common tension between Boolean and byte-level representations when executing algorithms over TFHE. Beyond achieving "the fastest AES-over-TFHE," our method could also be used for implementing other block ciphers of similar SPN (for *Substitution-Permutation Network*) structures in transciphering applications.

Transciphering is the central theme of the next chapter. While block ciphers were the focus here, the following chapter shifts attention to stream ciphers as alternative solutions.

Chapter

6 Better Transciphering with `Transistor`

In the previous chapter (Section 5.1), we have presented a technique called transciphering to adress the challenge of ciphertext expansion, common with all the FHE scheme. For example, a plaintext message of a few kilobytes can require tens or even hundreds of megabytes of data, making the processing of large data sets impractical. While compression techniques can help reduce the expansion factor in TFHE ciphertexts, the encrypted data still remains an order or two of magnitude larger than the original plaintext.

Transciphering consists in encrypting the data using a symmetric encryption cipher, and only encrypting the key of this cipher using the FHE scheme. Then, the server can evaluate the decryption algorithm of the symmetric cipher homomorphically to produce usable FHE ciphertexts. More information on transciphering is given in Section 5.1.

While `Hippogryph`, our implementation of homomorphic AES we introduced in previous chapter, could in theory be used for such transciphering task, the performances would not be acceptable for large volume of data (which is the use-case targeted by transciphering). We would rather have a symmetric cipher designed specifically to interact well with the FHE scheme. An abundant literature on the topic has appeared during the last few years, leading to the development of new families of ciphers tailored for the different homomorphic scheme on the market.

In this chapter, we present `Transistor`, a stream cipher optimized for transciphering with TFHE. This design is the outcome of a careful study of the constraints and advantages specific to achieving efficient homomorphic evaluations with TFHE. In particular, we argue that operating on elements of $\mathbb{F}_p$, where p is a small prime (4–5 bits), is a good choice for leveraging the full potential of TFHE's programmable bootstrapping: we chose $p = 17$. This choice is independent of the data format supported by the application running on the server, as changes of representations are easily feasible through bootstrapping [Ber+23a].

The design of `Transistor` has combined two very different challenges: ensuring security (in the sense of "traditional" cryptographic security), while being evaluable efficiently in the homomorphic domain. This thesis being about the development of efficient homomorphic operations, we mainly present the latter aspect in this chapter. A full version of this work is available in [Bau+25], giving a more complete vision of the other aspect of the design. In particular, we present a careful analysis of the noise evolution throughout the homomorphic evaluation of `Transistor`, to fine-tune the TFHE parameters for optimal performance. Our homomorphic implementation of `Transistor` significantly outperforms the state of the art, achieving a throughput of over 60 bits/s on a standard CPU. This represents a factor 3 speedup compared to `FRAST` [Cho+24], the previous fastest method, while also achieving a considerably lower error probability and eliminating the need for an expensive initialization phase.

The chapter is structured as follows. Section 6.1 discusses the design constraints for a stream cipher intended for use with TFHE, along with the design choices we made. The specification of `Transistor` and the reasoning behind its design is detailed in Section 6.2. Section 6.3 provides a brief high-level summary of the security implications. Finally, Section 6.4 details the

homomorphic implementation of our scheme, providing performance metrics and benchmarks.

6.1 Constraints for a TFHE-friendly Stream Cipher

TFHE Operations. TFHE enables the evaluation of both linear functions and look-up tables on encrypted data, each offering complementary properties.

Linear operations in TFHE are highly efficient but contribute to an increase in ciphertext noise. Specifically, when performing a linear combination of ciphertexts $c_1, \ldots, c_n$ with constant coefficients $\alpha_1, \ldots, \alpha_n$, the noise variance increases in proportion to the squared ℓ_2-norm of the coefficient vector, i.e., $\sum_{i=1}^{n} \alpha_i^2$. Therefore, to optimize efficiency and control the noise growth, a TFHE-friendly cipher can make greedy use of linear operations while minimizing the norm of the coefficient vectors to limit the resulting noise.

Conversely to linear operations, the programmable Bootstrapping (PBS) is a slow operation, but it allows the computation of any (small-precision) function chosen by the designer while reducing the noise in the ciphertext to a nominal level at the same time. Therefore, while we should minimize the number of these operations for the sake of efficiency, they are essential for introducing non-linearity into the cipher and limiting the noise growth throughout the execution. In practice, within our context, the use of PBS introduces further constraints which we address below.

The arrangement of operations. The PBS produces ciphertexts with a nominal noise level, which is typically lower than that of the input ciphertexts but still significantly higher than the noise in a fresh ciphertext. This implies that if the input bits are fresh encrypted data, they can undergo complex linear operations (specifically with potentially high ℓ_2-norms). In contrast, the linear functions applied to the outputs of each PBS should involve somewhat limited linear operations in their resulting ℓ_2-norms in order to limit the noise growth.

The size of the plaintext space. The choice of the plaintext space $\mathbb{Z}_p$ has a significant impact on the PBS. Indeed, execution time of the PBS grows exponentially with the number of bits of p, which is therefore usually limited to a few bits. Although some recent works ([GBA21; Chi+21; Cle+22; KS23]) introduce more sophisticated techniques for efficiently evaluating larger LUTs, their performance in terms of bits per second remains less favorable compared to using lower precision.

The parity of the plaintext space. Following the general line of work of this thesis, we chose an odd plaintext modulus p to get rid of the negacyclicity problem. But using an odd modulus may seem unsuitable for manipulating bits or groups of bits. Indeed, it may lead to data expansion, as an element of $\mathbb{Z}_p$ cannot perfectly encode a group of bits. To mitigate this issue, one can select an odd p that is slightly larger but close to 2^ℓ for some ℓ, allowing for the efficient embedding of ℓ-bit chunks into elements of $\mathbb{Z}_p$.

Our design choices.

We deduce the following guidelines for our design:

1. The plaintext space of the scheme will be reduced to a few bits to take advantage of the relative speed of the PBS at small precision. Specifically, we chose $p = 17$ which meets our constraints as being odd (no negacyclicity) and the closest to a low power of 2 (thus well suited to encode nibbles of data). Besides, letting p be a prime number eases the design and security analysis thanks to the field structure of $\mathbb{Z}_p = \mathbb{F}_p$.

Moreover, operating in $\mathbb{F}_{17}$ does not constrain the server-side application to this field. Once the server retrieves the homomorphic ciphertexts, they can be efficiently converted to any other space with a bootstrapping. We elaborate more on this point in Section 6.4.2.

2. The non-linearity comes from a layer of S-boxes, each computing a function $\mathbb{F}_p \to \mathbb{F}_p$ giving rise to one PBS evaluation. Given our fixed choice of p, the number of PBS per element of the output stream represents the main performance metric which we search to minimize.

3. The initial key material (stored as fresh TFHE ciphertexts) can go through complex linear combinations before hitting the S-box layer.

4. Each S-box output should only go through lightweight linear operations (i.e., with low ℓ_2-norms) before undergoing another PBS in order to make the noise in the input of the PBS sufficiently low to ensure correctness.

5. Each S-box output should only go through lightweight linear operations (i.e., with low ℓ_2-norms) before being released. This way, the TFHE ciphertexts obtained after the stream-cipher decryption keep a noise level as close to nominal as possible.

6.2 Description of `Transistor`

Bringing everything together, we designed the stream cipher `Transistor`. Its overall structure is presented in Section 6.2.1, its details are explained in Section 6.2.2, and the influence of noise is discussed in Section 6.2.3. A reference implementation can be found at https://github.com/CryptoExperts/Transistor/.

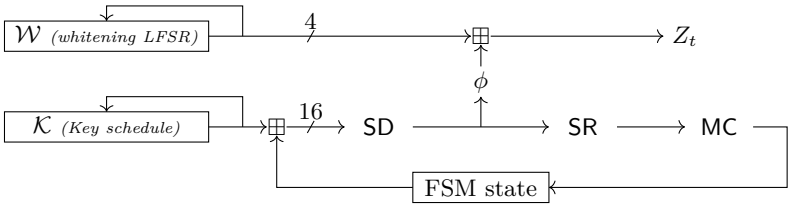
6.2.1 Overall Structure

Usage. `Transistor` is a stream cipher that generates a keystream consisting of elements from $\mathbb{F}_p = \mathbb{F}_{17}$, referred to as *digits*. It is intended for transciphering, i.e., for the type of protocol we summarized in Section 5.1. More precisely, a 128-bit master key and an IV are used to initialize the internal state of `Transistor` using a PRF (namely, `SHAKE` [NIS15]), as suggested in [BG07]. This initialization is only performed on the client side, and in particular is not evaluated homomorphically, meaning that its cost is negligible. On the other hand, it ensures for example that related IVs cannot be exploited. The entire resulting internal state is then encrypted using TFHE and sent alongside the ciphertext. This ciphertext is obtained by casting the plaintext message to a string of digits of $\mathbb{F}_p$, which is added digit by digit to the keystream produced by `Transistor` using the group law of $\mathbb{F}_p$.

Internal State. The overall structure of `Transistor` is outlined in Figure 6.1.

The idea is to generate two pseudo-random sequences with a very long period using two distinct LFSRs. One of them generates whitening subkeys, while the other acts as a sort of key schedule. The output of the latter is fed into a Finite State Machine (FSM) with its own state, and which operates on it using non-linear operations. We thus have the following components:

- a register of 16 elements of $\mathbb{F}_p$ (the *FSM state*),
- a Linear-Feedback Shift Register (LFSR) over $\mathbb{F}_p$ (the *key schedule* or *key-LFSR* $\mathcal{K}$) of length $|\mathcal{K}| = 64$,
- an LFSR over $\mathbb{F}_p$ (the *whitening LFSR* $\mathcal{W}$), of length $|\mathcal{W}| = 32$,
- a non-linear round function from $\mathbb{F}_p^{16}$ to itself (the *round function*), and



(a) General structure (rectangles correspond to registers).

(b) SD. *(c) SR.* *(d) MC.* *(e) ϕ.*

Figure 6.1: A high level view of `Transistor`.

- a filter $\phi : \mathbb{F}_p^{16} \to \mathbb{F}_p^4$ that extracts 4 digits from the FSM.

The FSM state is initialized to all zeros, and each LFSR is initialized using digits derived from the 128-bit master key and IV using SHAKE [NIS15].

Security Claim. `Transistor` is a stream cipher providing 128 bits of security, meaning any attack should require at least 2^{128} elementary operations, assuming no more than 2^{31} digits (about 1 GB) are generated with each IV. We allow up to 2^{128} digits in total per key, corresponding to the multi-initial-state setting.

6.2.2 Detailed Description

Obviously taking inspiration from the AES, the state of the FSM is organized into a two-dimensional array of size 4×4, where each entry corresponds to a digit in $\mathbb{F}_p$. With this representation, the successive operations applied to the state can be defined as follows.

SubDigits (**SD**) is an S-box layer: the permutation π is applied on each digit.

MixColumns (**MC**) applies to each column an MDS matrix M over $\mathbb{F}_p$.

ShiftRows (**SR**) rotates the i-th row by i positions to the left.

Filter (ϕ) takes 4 digits from the state and returns them.

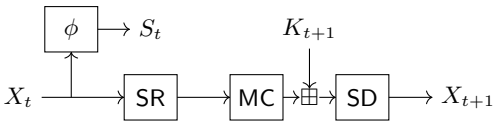
In what follows, we provide a more detailed description of each step, using the notation summarized in Figure 6.2a. The keystream output at clock $t \geq 0$ consists of a tuple $Z_t \in \mathbb{F}_p^4$, called a block. The internal state of the FSM, just before the filter is applied, is denoted by X_t (so that $S_t = \phi(X_t)$). As a consequence, $X_{t+1} = \mathsf{SD}\left(K_{t+1} + (\mathsf{MC} \circ \mathsf{SR}(X_t))\right)$, where K_t is obtained by concatenating 16 successive digits generated by the key-schedule LFSR $\mathcal{K}$. The FSM is initialized with the all-zero value and its initial state is denoted by $X_{-1} := 0$.

S-box Layer (SD). We let π be defined by its lookup table:

$$\pi = [1, 12, 6, 11, 14, 3, 15, 5, 10, 9, 13, 16, 7, 8, 0, 2, 4] \tag{6.1}$$

so that $\pi(0) = 1$, $\pi(1) = 12$, and so on. It has the following polynomial representation, and is thus of maximum degree:

$$\begin{aligned}\pi(x) \;&= 1 + 4x^1 + 13x^2 + 7x^3 + 16x^4 + 15x^5 + 5x^7 + 5x^8 \\ &\quad + 11x^9 + 13x^{10} + 12x^{11} + 13x^{12} + 15x^{14} + x^{15}\ .\end{aligned}$$



(a) Notation throughout clocks.

0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

(b) Numbering in the FSM.

Figure 6.2: Our notations. Note that the numbering of the digits differs from the one traditionally used for the AES.

It was chosen by enumerating all APN permutations of $\mathbb{F}_{17}$, i.e., all permutations A such that the equation $A(x+a) = A(x) + b$ has at most 2 solutions x for all $a \neq 0$ and all b. Then, we selected π among those that offer a good balance between minimizing the number of pairs (a, b) for which the previous equation has exactly two solutions, and minimizing the maximum modulus of the Walsh spectrum (see the full article [Bau+25]).

Linear Layer (MC). We opted for a 4×4 Maximum Distance Separable (MDS) matrix to ensure optimal diffusion. The matrix we chose is

$$M = \begin{bmatrix} 2 & 1 & 1 & 1 \\ 1 & -1 & 1 & -2 \\ 1 & 1 & -2 & -1 \\ 1 & -2 & -1 & 1 \end{bmatrix} . \tag{6.2}$$

We verified that there is no MDS matrix in $\mathbb{F}_{17}$ with coefficients in $\{-1, 1\}$ by exhaustively testing all such matrices. As we were interested in MDS matrices with minimal ℓ_2-norm and we were able to find during the initial experiments matrices with a squared ℓ_2-norm of 7, it was evident from the definition of the ℓ_2-norm that matrices with minimal ℓ_2-norm could not have coefficients x with $|x| > 2$. Thus, by testing all matrices with coefficients in $\{-2, -1, 1, 2\}$, we found a total of 30 720 MDS matrices with an ℓ_2-norm of 7. We selected M for its symmetries, particularly because it is its own transpose.

Filter. The filter function ϕ maps $\mathbb{F}_p^{16}$ (i.e., the full FSM state) to a tuple (a, b, c, d) in $\mathbb{F}_p^4$. As summarized in Figure 6.1e, we have that a, b, c and d correspond to the digits of the FSM state with indices 4, 6, 12, and 14 respectively (using the numbering from Figure 6.2b).

LFSRs. The whitening LFSR $\mathcal{W}$ and the key schedule LFSR $\mathcal{K}$ are simply LFSRs over $\mathbb{F}_p$ of maximum period, and have length 32 and 64 respectively. We obtain a maximum-period LFSR over $\mathbb{F}_p^w$ using the coefficients of a primitive polynomial as the taps. More precisely, we used the `SageMath` implementation of the finite field $\mathbb{F}_{p^w}$, which resulted in a pseudo-Conway polynomial. The output of the LFSR is taken from its last cell.

More precisely, an LFSRs of length ℓ at time t is a list of digits $x_0^t, ..., x_{\ell-1}^t$ that is clocked as follows:

1. $x_0^{t+1} \leftarrow -\sum_{i=0}^{\ell-1} x_i^t c_i$,
2. $x_i^{t+1} \leftarrow x_{i-1}^t$ for $0 < i < \ell$,
3. the output is $x_{\ell-1}^t$,

where $C = (c_i)_{0 \leq i < \ell}$ is the list of its taps, each being a digit of $\mathbb{F}_{17}$. We define clock_C to be the function applying the operations above to a list $x_0^t, ..., x_{\ell-1}^t$ to update it, and returning $x_{\ell-1}^t$.

For the key schedule $\mathcal{K}$, we use the following taps:

$$\begin{aligned} C(\mathcal{K}) = \{&9, 4, 6, 4, 8, 6, 6, 16, 3, 9, 15, 12, 8, 12, 11, 4, 4, 8, 1, 8, 8, 9, 4, 6, 6, 7, 6, 3, \\ &16, 14, 14, 6, 10, 15, 14, 13, 10, 1, 1, 10, 13, 11, 14, 10, 7, 4, 15, 8, 16, 3, 13, \\ &14, 15, 16, 3, 16, 9, 3, 6, 12, 15, 9, 12, 3\} , \end{aligned}$$

and for the whitening LFSR $\mathcal{W}$ we use

$$C(\mathcal{W}) = \{8, 14, 14, 14, 1, 6, 12, 10, 14, 14, 14, 5, 2, 5, 6, 13, 6, 15, 14, 3, \\ 13, 16, 1, 13, 9, 1, 7, 15, 13, 6, 14, 3\} .$$

Master Key Processing. We generate the digits in $\mathcal{K}$ first, and then those in $\mathcal{W}$. To generate them, we concatenate the 128-bit long master key with an IV and then a byte set to 1. The result is fed into SHAKE128, and the output byte stream of this primitive is used to generate digits of $\mathbb{F}_{17}$ using rejection sampling: if a byte x is equal to 255, we discard it; otherwise, we generate the digit $\lfloor x/15 \rfloor$. Since $15 \times 17 = 255$, this results in an unbiased transformation.

6.2.3 Controlling the Noise Evolution

We first detail the implementation of each building block of the scheme using TFHE, as this is essential to justify our design choices and to understand the evolution of the noise throughout the cipher. We then use this discussion to explain how the noise influences the overall security and efficiency of `Transistor`.

LFSR. A naive approach for implementing an LFSR homomorphically would be to maintain an encrypted state, and update it by computing a linear combination with the feedback coefficients. However, this method would cause the noise in the state to accumulate over time, necessitating periodic use of PBS operations to refresh and control the noise growth. For this reason we introduce the principle of the *silent LFSR*. Every output of an LFSR is a linear combination of the digits in its initial state. By computing on the fly the coefficients of these linear combinations in clear, we can evaluate the output of the LFSR at every clock cycle without updating an encrypted version of the internal state. This way, the noise variance in the output of the silent LFSR remains stable over time. This principle is comparable to the approach of `FLIP` [Méa+16] and follow-up works, whereby a key state is queried without being updated.

To bound the noise variance in the output of the silent LFSR, we consider the worst-case scenario in which all the coefficients in the linear combinations are of maximal absolute value, i.e., $\frac{p-1}{2}$.[1] The resulting noise variance is thus equal to the original noise variance multiplied by the worst-case squared ℓ_2-norm. Specifically, in the output of the key schedule LFSR $\mathcal{K}$ and of the whitening LFSR $\mathcal{W}$, the noise variances $\sigma^2_{\mathcal{K}}$ and $\sigma^2_{\mathcal{W}}$ satisfy

$$\sigma^2_{\mathcal{K}} \leq |\mathcal{K}| \cdot \left(\frac{p-1}{2}\right)^2 \cdot \sigma^2_{\text{fresh}} \quad \text{and} \quad \sigma^2_{\mathcal{W}} \leq |\mathcal{W}| \cdot \left(\frac{p-1}{2}\right)^2 \cdot \sigma^2_{\text{fresh}}, \tag{6.3}$$

where σ^2_{fresh} is the noise variance of the encrypted key material in the LFSRs.

SubDigits. Each digit of the state of the FSM goes through a PBS that evaluates the permutation π. All PBSs can be evaluated in parallel for higher speed. We denote by σ^2_{PBS} the noise variance at the output of SubDigits for encrypted digits.

ShiftRows. Since each digit in the state is encrypted in a separate ciphertext digit, this step involves simply rearranging the ciphertext digits within the state. Consequently, it incurs no additional noise growth and no performance impact.

[1] Constant coefficients of $\mathbb{F}_p$ are encoded as integers of the interval $[-\frac{p-1}{2}, \frac{p-1}{2}]$ to minimize their absolute value and hence their impact on the noise.

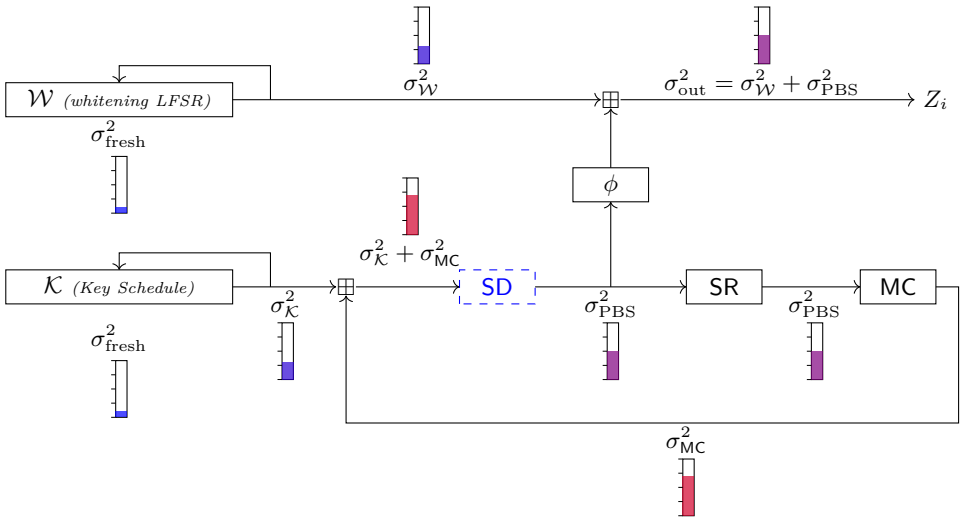


Figure 6.3: Evolution of the noise variance in a homomorphic evaluation of `Transistor`*. Operations involving PBSs are in blue and dashed. The gauges allow to visualize the evolution of the noise on a logarithmic scale).*

MixColumns. This operation involves a straightforward linear combination of the digits of the state. The matrix M has been specifically constructed to minimize the ℓ_2-norm, considering coefficients in $\mathbb{F}_{17}$. From the homomorphic perspective, this choice is crucial, as the variance of the noise increases proportionally with the square of this ℓ_2-norm, that we denote L_{MC}. Namely, the noise variance σ^2_{MC} after MixColumns satisfies $\sigma^2_{\mathsf{MC}} = L^2_{\mathsf{MC}} \cdot \sigma^2_{\text{PBS}}$.

Sums. The output of MixColumns is then added to the next output of the LFSR to be injected again into SubDigits. The noise variance after the addition step corresponds to the sum of both noise variances. Similarly, the noise variance at the output of the scheme, referred to as σ^2_{out}, is equal to the sum $\sigma^2_{\mathcal{W}} + \sigma^2_{\text{PBS}}$.

Figure 6.3 illustrates the evolution of the noise variance throughout the operations of `Transistor`. Building on the previous equations, the main constraints influencing the design of `Transistor` are related to the noise variance at both the input of the PBS and the output of the scheme. Specifically, the noise variance $\sigma^2_{\mathcal{K}} + \sigma^2_{\mathsf{MC}}$ at the input of the PBS (i.e., at input of SubDigits) must remain sufficiently low, otherwise it could lead to a high probability of PBS failure. Additionally, the noise variance $\sigma^2_{\text{out}} = \sigma^2_{\mathcal{W}} + \sigma^2_{\text{PBS}}$ at the output stream must remain low enough for subsequent applications, ideally as close as possible to the nominal noise variance at the PBS output σ^2_{PBS}. In practical settings, we have $\sigma^2_{\text{PBS}} \gg \sigma^2_{\text{fresh}}$, the noise variances from both LFSRs are negligible compared to σ^2_{PBS}. For example in our implementation, the noise magnitude of σ_{fresh} is around 2^{14}, while the noise magnitude of σ_{PBS} is around 2^{52}. Consequently, $\sigma^2_{\text{out}} \approx \sigma^2_{\text{PBS}}$ which validates the second constraint. Similarly, the noise variance at the input of the PBS is close to that at the output of MixColumns. The latter additionally remains low due to the minimized ℓ_2-norm of the coefficients of the MDS matrix M, thereby validating the first constraint.

To wrap up, the design of `Transistor` allows to control the evolution of the noise in the FSM while getting a very low number of PBS per element. To complete our noise analysis, we need to set the parameters of the TFHE scheme to ensure the correctness of the PBS. Concretely, the noise $\sigma^2_{\mathcal{K}} + \sigma^2_{\mathsf{MC}}$ at the input of SubDigits should be low enough to fail with a negligible probability. Of course, these parameters must ensure that the PBS operates as fast as possible while maintaining the security of the scheme. In Section 6.4.4, we detail our method for selecting the parameters.

6.3 A Brief Summary of the Security Analysis

`Transistor`'s full paper provides an extensive analysis of the security of the cipher. Even if these considerations are far from the topic of this manuscript, we provide in this section a brief overview of this analysis. We refer to the full paper [Bau+25] for the developments of the proofs.

Time-Memory-Data Trade-Offs To dimension the size of the LFSR, we computed a bound to ensure that exhaustive attacks are out of reach even when leveraging trade offs with pre-computation and storage.

Using $\mathcal{K} = 64$ and $\mathcal{W} = 32$, the length of the keystream generated from the same key is limited to 2^{31} digits. As a result, TMDTO attacks have a time complexity of 2^{296} in the single IV-setting, which drops to 2^{130} when keystreams generated from 2^{130} IVs are available to the attacker.

Guess and Determine In this kind of attacks, the attacker links the FSM state X_t to the filter output S_t and try to guess the key schedule K_t.

Based on an analysis of the filtering procedure of `Transistor`, we showed that in total the attacker has to guess $\frac{12}{16}|\mathcal{K}|$ digits, leading to a complexity $p^{\frac{3}{4}|\mathcal{K}|} \approx 2^{196}$ without taking into account the whitening LFSR. If we consider it, the attacker first has to guess its content, leading to an attack with complexity $p^{\frac{3}{4}|\mathcal{K}|+|\mathcal{W}|} \approx 2^{294}$.

Three consecutive outputs are statistically independent of the secret key The basic strategy in (fast) correlation attacks against stream ciphers consists in recovering some information about (a part of) the initial state of the cipher from the knowledge of the keystream. In this context, an important quantity is the smallest length of output sequence $(S_t)_{t\in\mathbb{N}}$ that can provide information on the sequence produced by the key-LFSR. In the paper we prove that this length is 4, that is to say 3 consecutive outputs are statistically independent of the secret key. This is a very good performance with respect to the state of the art: the only other cipher with this property is `Rocca` [Sak+21, Section 4.5]. However, in this case this property has been derived from an automatic search method, while the structure of `Transistor` enables us to derive this argument in a very simple way from the MDS property of MixColumns.

(Fast) Correlation Attacks Using Biased Linear Relations Our paper also provides an estimation of the minimal data complexity required to recover the internal state of the key-register from the knowledge of the output sequence $(S_t)_{t\in\mathbb{N}}$, given that at least four consecutive outputs $(S_t, S_{t+1}, S_{t+2}, S_{t+3})$ need to be considered together. Applying the so-called Xiao-Massey lemma [XM88; Bry89], it is possible to see that as soon as the key-LFSR and the considered segment of the output sequence are not statistically independent, there exists a biased linear relation between the digits of these two sequences.

In the paper, we exhibit an upper bound on the correlation of such linear relation. This bound depends on the minimal number of active S-boxes over n rounds, as well as the modulus of the Fourier coefficients of `Transistor`'s S-box. We show that the design of our S-box brings the correlation down to a value small enough so that the amount of keystream that the attacker has to observe exceed the limit of 2^{31} digits fixed by TMDTO trade-offs.

Linear Distinguishers Another type of attack studied in the paper is linear distinguishing attacks [MS88; CT00; CJS01; Tod+18]. These attacks do not recover the initial state of the key-register, but the counterpart is that they can use together several keystream segments produced from multiple initial state. They consist in exhibiting a biased linear relation among the keystream digits. Such a relation is typically derived from a *parity-check equation* for the key-LFSR, defined by the multiples of the LFSR feedback polynomial.

In `Transistor`'s design, this polynomial has been chosen so that it shows good properties regarding the utility of these parity-check equations. The conclusion of our analysis is that such an attack would be more expensive than an exhaustive search for the key .

The full security analysis also takes into account algebraic attacks; such that Gröbner basis or the use of annihilators of the filtering function.

6.4 Performances of Transciphering with `Transistor`

This section focuses on the performances of transciphering with `Transistor`. We first address the wrapping of a (`Transistor`) symmetric key as a compact set of TFHE ciphertexts for which we additionally introduce a trade-off between bandwidth and computation. Next, we explain how to manage different data representations to be able to fit with the input format of the server application. We then provide a detailed description of the homomorphic evaluation of `Transistor`. We finally give some implementation benchmarks and comparison to the state of the art.

6.4.1 Key Wrapping and Bandwidth in TFHE Transciphering

Assume one wants to generate a fresh TFHE ciphertext vector $(c_1, \ldots, c_t)$ for a plaintext vector $(m_1, \ldots, m_t) \in \mathbb{Z}_p^t$, where $c_i = (a_{i,1}, \ldots, a_{i,n}, b_i)$, for every $i \in [1, t]$. Since the $a_{i,j}$'s are uniformly sampled at random over $\mathbb{Z}_q$, a folklore trick is to generate them pseudorandomly from a seed. We get the following compressed encryption procedure (where λ denotes the security level in bits):

`CompressEncrypt`$(s, m_1, \ldots, m_t)$

1. Sample $\mathsf{seed} \leftarrow \{0,1\}^\lambda$
2. Expand $((a_{i,j})_{1 \le j \le n})_{1 \le i \le t} \leftarrow \mathsf{PRG}(\mathsf{seed})$
3. $\forall\, i \in [1, t]$: $b_i \leftarrow \sum_{j=1}^{n} a_{i,j} \cdot s_j + \tilde{m}_i + e_i$ with $e_i \leftarrow \chi_\sigma$
4. Return $(\mathsf{seed}, b_1, \ldots, b_t)$

Recovering standard TFHE ciphertexts $c_1, \ldots, c_t$ from the compressed form $(\mathsf{seed}, b_1, \ldots, b_t)$ is simply done by expanding the $a_{i,j}$'s from seed. The size of the obtained compressed ciphertext vector is $\lambda + t \cdot \log_2(q)$ against $t \cdot (n+1) \cdot \log_2(q)$ for a standard TFHE encryption, meaning a compression by a factor about $(n+1)$.

This compressed TFHE encryption method can be applied directly to transmit homomorphically encrypted data from the user to the server. Alternatively, it can be combined with transciphering to encrypt a symmetric key. The resulting bandwidth requirements and the corresponding plaintext-to-ciphertext expansion factor are summarized in Table 6.1, where they are further compared with the naive (uncompressed) TFHE encryption. In particular, for `Transistor`, a wrapped key is of size $\lambda + (|\mathcal{K}| + |\mathcal{W}|) \cdot \log_2(q)$, (which in our case gives 784 bytes) for a security of $\lambda = 128$ bits (target security of `Transistor`), the standard choice of $q = 2^{64}$ (which we use in our implementation) and $|\mathcal{K}| + |\mathcal{W}| = 96$ per the specification of `Transistor` (see Section 6.2). This fixed cost is hence very quickly amortized while the amount of data to encrypt grows. Moreover, this approach can be applied to the server keys as well, which are actually encryptions of the secret key's bits. We took this optimization into account in our estimations of the server key sizes in Table 6.4.

Table 6.1: Bandwidth of homomorphic ciphertexts (in bits).

Approach used	**Naive**	**Compressed**	**`Transistor`**
Fixed cost	0	λ	$\lambda + (\lvert\mathcal{K}\rvert + \lvert\mathcal{W}\rvert) \cdot \log_2(q)$
Per message in $\mathbb{Z}_p$	$(n+1) \cdot \log_2(q)$	$\log_2(q)$	$\log_2(p)$
Expansion factor	$(n+1) \cdot \log_2(q)/\log_2(p)$	$\log_2(q)/\log_2(p)$	1

Compressing further. We introduce hereafter a tweak to compress a TFHE encryption further than the folklore compression. By definition of the TFHE encryption process, the least significant bits of the body $b_i = \sum_{j=1}^{n} a_{i,j} \cdot s_j + \tilde{m}_i + e_i$ are randomized by the error e_i and can hence be discarded without loss of information. We can thus tweak the above compressed encryption process by returning $(\mathsf{seed}, \mathrm{Tr}_\ell(b_1), \ldots, \mathrm{Tr}_\ell(b_t))$ where $\mathrm{Tr}_\ell(\cdot)$ denotes the truncation of the ℓ least significant bits. To decompress such ciphertexts, besides pseudorandomly generating the masks from the seed, one just needs to pad the truncated bodies with ℓ bits to 0. By the randomness of the mask, the effect of this truncation plus 0-padding is to add a uniform random error of ℓ bits to the body, namely an error of standard deviation:

$$\sigma_0^2 = \frac{(2^\ell - 1)^2}{12} \approx \frac{2^{2\ell}}{12} \approx 0.08 \cdot 2^{2\ell} \ .$$

This optimization comes in two flavors:

1. *The "free" variant.* The number of truncated bits ℓ is selected to have a small impact on the noise distribution. For instance in `Transistor`, the noise of the fresh ciphertexts is summed with the noise coming from the FSM. Thus, we can compute ℓ to keep $\sigma_{\mathcal{W}}^2 < \sigma_{\mathrm{PBS}}^2$. Our experiments shows $\sigma_{\mathrm{PBS}}^2 = 2^{52}$, so running the numbers we find that we can truncate up to $\ell = 19$ bits, allowing to reduce the volume of the TFHE ciphertexts to send by a factor $1 - \frac{19}{64} \approx 0.7$.

2. *The communication-computation trade-off.* In this variant, one selects a high value of ℓ. The truncated body should at least contain $\log_2(p)$ bits to keep the plaintext information, plus a margin of a few bits in order to remain bootstrappable. Denoting this margin δ, the truncated body should be of at least $\log_2(p)+\delta$ bits and ℓ can be up to $\log_2(q)-(\log_2(p)+\delta)$. Taking the maximum level of truncation, inducing the maximum level of bootstrappable noise, implies some adaptation of the underlying homomorphic computation. Specifically, it should start with applying a noise-reduction bootstrapping to the decompressed ciphertexts before performing the original evaluation. We hence obtain a trade-off with reduced bandwidth against additional bootstrappings.

In the context of transciphering with `Transistor`, the trade-off provided by the second option gives rise to an initialization procedure which consists in decompressing and bootstrapping the wrapped key.

This kind of compression has been more extensively studied in the concurrent work [Bon+24].

6.4.2 Transciphering vs. Data Representation

Managing data representation is a common challenge when working with TFHE. Since this scheme is only efficient at very low precision, an abstraction layer is required to construct practical data types (e.g., 8-, 32-, or 64-bit integers) from smaller encrypted chunks. Common constructions include radix-based decompositions and Chinese Remainder Theorem (CRT) representations, leading to different efficiency trade-offs. Carry propagation in radix-based representations is notoriously slow due to the large number of required bootstrappings, while CRT representations impose constraints on feasible operations. These constructions have been studied in [Ber+23a].

As a result, there is no universal representation that is optimal for all homomorphic operations. Thankfully, the representation of data in the transciphering algorithm can be chosen independently of that of the homomorphic application running on the server. If the representation in $\mathbb{Z}_p$ does not suit the application, the server can convert the ciphertexts to the desired representation before running the application. We stress that this additional step of conversion would be necessary for any transciphering algorithm, as the data format desired in output of transciphering is completely application-dependent.

As a concrete example, assume that the data to be encrypted (i.e., the input to the homomorphic computation) consists of elements from $\mathbb{Z}_{16}$. The overall transciphering process unfolds as follows. On the client side, the plaintext is first embedded from $\mathbb{Z}_{16}$ into $\mathbb{Z}_{17}$ before being encrypted using `Transistor`. On the server side, the keystream is homomorphically generated and then used to homomorphically decrypt the ciphertext. This results in a TFHE encryption of the original plaintext, now embedded in $\mathbb{Z}_{17}$, meaning that the plaintext space for the TFHE encryption is $\mathbb{Z}_{17}$. A programmable bootstrapping (PBS) operation is then applied to switch the plaintext space from $\mathbb{Z}_{17}$ back to $\mathbb{Z}_{16}$.

In terms of computation, this process adds one PBS per $\mathbb{Z}_{16}$-element of the original plaintext, in addition to the four PBS per element required for keystream generation with `Transistor`. Moreover, embedding $\mathbb{Z}_{16}$ into $\mathbb{Z}_{17}$ increases the size of the encrypted data by a factor of $1 + 1/16 = 1.0625$.

This approach can be generalized to address other plaintext representations. In particular, for larger chunks of bits, the bootstrapping operation would allow to merge several elements of $\mathbb{Z}_{16}$ (embedded into $\mathbb{Z}_{17}$) into one element of $\mathbb{Z}_{2^\ell}$ with $\ell > 4$. On the other hand, one may split an element of $\mathbb{Z}_{16}$ (or its $\mathbb{Z}_{17}$ embedding) into 4 elements of $\mathbb{Z}_2$ using a PBS with multiple look-up tables ("PBSmanyLUT") as proposed in [Chi+21].

6.4.3 Detailed Homomorphic Implementations

In the following, we provide a more detailed way of how we implemented the homomorphic version of `Transistor`.

Homomorphic evaluation of LFSRs. The `Transistor` design involves two LFSRs operating on elements of $\mathbb{F}_{17}$. The standard way to implement an LFSR is to evaluate the linear feedback function on the state at each clock cycle, thus producing a new element that enters the state, while the state is shifted to output an element.

We suggest the *silent LFSR* approach for the homomorphic evaluation of LFSRs. In this approach, the encrypted LFSR state is immutable to avoid any noise growth in the underlying ciphertexts (hence keeping the LFSR "silent"). We use the fact that every output element of the LFSR can be expressed as a linear combination of the initial state. So, at each clock cycle, we compute *in the clear* the coefficients of this linear combination and homomorphically evaluate it on the immutable encrypted state. This process is depicted in Algorithm 9.

Homomorphic evaluation of `Transistor`. The complete homomorphic evaluation of a round of on clock cycle of `Transistor` is depicted in Algorithm 10, using $\mathcal{K}$.`clock` and $\mathcal{W}$.`clock` as subroutines (i.e., Algorithm 9 evaluated on the key schedule and whitening LFSRs). The most computation intensive part of the algorithm is by far the evaluation of the PBS in SubDigits which can be fully parallelized to reduce the latency.

6.4.4 TFHE Parameters

We discuss hereafter the selection of the different TFHE parameters involved in the homomorphic implementation of `Transistor`. An extensive presentation of the TFHE parameters and their role within the scheme can be found in Section 8.1.

Algorithm 9 `LFSR.clock` - Produce a pseudo random element of the state.

Input:
- ℓ : Size of the state of the LFSR.
- $(u_1, \ldots, u_\ell)$: Encrypted initial state of the LFSR.
- $(\lambda_1^{(0)}, \ldots, \lambda_\ell^{(0)})$: Coefficients of retroaction in the definition of the LFSR.
- $(\lambda_1^{(i)}, \ldots, \lambda_\ell^{(i)})$: Previous coefficients used in the linear combination.

Result:
- $o^{(i)}$: Encryption of the i-th pseudorandom element of $\mathbb{F}_{17}$.
- $(\lambda_1^{(i+1)}, \ldots, \lambda_\ell^{(i+1)})$: Updated coefficients of the linear combination.

$o^{(i)} \leftarrow 0$
`/* Evaluation of the linear combination */`
for $k \in \{1, \ldots, \ell\}$ **do**
  $o^{(i)} \leftarrow$ `SumTFHE`$(o^{(i)},$ `ClearMultTFHE`$(u_k, \lambda_k^{(i)}))$
end
`/* Update of the next coefficients */`
for $k \in \{2, \ldots, \ell\}$ **do**
  $\lambda_k^{(i+1)} \leftarrow \lambda_{k-1}^{(i)} + \lambda_\ell^{(i)} \cdot \lambda_k^{(0)}$
end
$\lambda_1^{(i+1)} \leftarrow \lambda_\ell^{(i)} \cdot \lambda_1^{(0)}$
return $o^{(i)}$

Table 6.2: TFHE Parameters used in our experiments

p_{err}	q	n_{short}	k	N	σ_{short}	σ_{long}	$\mathfrak{B}_{\mathsf{PBS}}$	ℓ_{PBS}	$\mathfrak{B}_{\mathsf{KS}}$	ℓ_{KS}	λ_{short}	λ_{long}
2^{-40}	2^{64}	788	2	1024	2^{47}	2^{14}	2^{23}	1	2^4	3	131.8	128.9
2^{-128}	2^{64}	774	1	2048	2^{47}	2^{14}	2^{23}	1	2^3	5	131.8	128.9

Figure 6.4 shows the ciphertext format at the different steps of the homomorphic evaluation of `Transistor`. It shows that the manipulated ciphertexts can be of three different types: LWE ciphertexts of dimension n_{short}, LWE ciphertexts of dimension n_{long}, or GLWE ciphertexts of dimension k and polynomial degree N.

Optimization of the TFHE parameters. To generate a set of parameters, we use the method developed in [Ber+23a]. Given the negligible noise contribution from the LFSRs, the FSM can be modeled using the *atomic pattern* introduced in [Ber+23a] (specifically the instance referred to as $\mathcal{A}^{(\text{CJP21})}$), which is a pattern of homomorphic operators taking a set of ciphertexts in input, computing linear combinations of those ciphertexts and applying a programmable bootstrapping to each of them. The FSM round in `Transistor` which composes a multiplication by a constant matrix (MixColumns), followed by a bootstrapping step (SubDigits) is precisely an instance of such an atomic pattern. The framework proposed in [Ber+23a] generates parameters that guarantee a specified security level λ for the LWE encryption and target error probability p_{err}, while optimizing the PBS to be as fast as possible.

Table 6.2 shows the parameters used for our experiments, all ensuring 128 bits of security. The obtained security levels λ_{short} et λ_{long} have been estimated using the `lattice estimator` [APS15].

Encryption security. The security level (in bits) of the LWE ciphertexts is a function of the modulus q, the dimension n and the noise standard deviation σ. While no explicit formula

Algorithm 10 `Transistor.clock` - Produce r encypted elements of the key stream

Input: $\begin{cases} \mathcal{K} : \text{the LFSR used for the pseudo-keyschedule and its state (cf Algorithm 9).} \\ \mathcal{W} : \text{the LFSR used for the whitening.} \\ X = \begin{pmatrix} x_{1,1} & \dots & x_{1,\sqrt{m}} \\ \dots & \dots & \dots \\ x_{\sqrt{m},1} & \dots & x_{\sqrt{m},\sqrt{m}} \end{pmatrix} : \text{Encrypted state of the FSM} \end{cases}$

Result: $\{Y = (y_1, \dots, y_r) : \text{Encryption of } r \text{ elements of the key stream}$

```
/* Compute the pseudo-key schedule and adds it to the FSM                    */
for i ∈ [1, √m] do
    for j ∈ [1, √m] do
        k_{i,j} ← K.clock()
        x_{i,j} ← SumTFHE(x_{ij}, k_{i,j})
    end
end
/* Compute SubDigits with a layer of PBS                                     */
for i ∈ [1, √m] do
    for j ∈ [1, √m] do
        x_{i,j} ← PBS_TFHE(x_{i,j}, S)
    end
end
/* Extract the output bits and whiten them                                   */
(y_1, ..., y_r) ← φ(X)
 for i ∈ [1, r] do
    w_i ← W.clock()
    y_i ← SumTFHE(y_i, w_i)
end
/* Compute ShiftRows, (same as in clear)                                     */
X ← SR(X)
/* Compute MixColumns                                                        */
for i ∈ [1, √m] do
    for j ∈ [1, √m] do
        z_{i,j} ← 0
        for k ∈ [1, √m] do
            z_{i,j} ← SumTFHE(z_{i,j}, ClearMultTFHE(x_{k,j}, MC_{i,k}))
        end
    end
end
return Y
```

exists for this function, the *lattice estimator* tool allows to produce an estimation of this function by simulating the main attacks of the literature [APS15], which we denote $\mathcal{O}$ (for *security oracle*). The selected TFHE parameters are constrained to satisfy $\mathcal{O}(q, n, \sigma) \geq \lambda$ for both $(q, n, \sigma) = (q, n_{\text{short}}, \sigma_{\text{short}})$ and $(q, n, \sigma) = (q, n_{\text{long}}, \sigma_{\text{long}})$.

Correctness of the PBS. To compute the error probability p_{err}, we have to evaluate the variances occurring inside the programmable bootstrapping of the SubDigits layer. This reason-

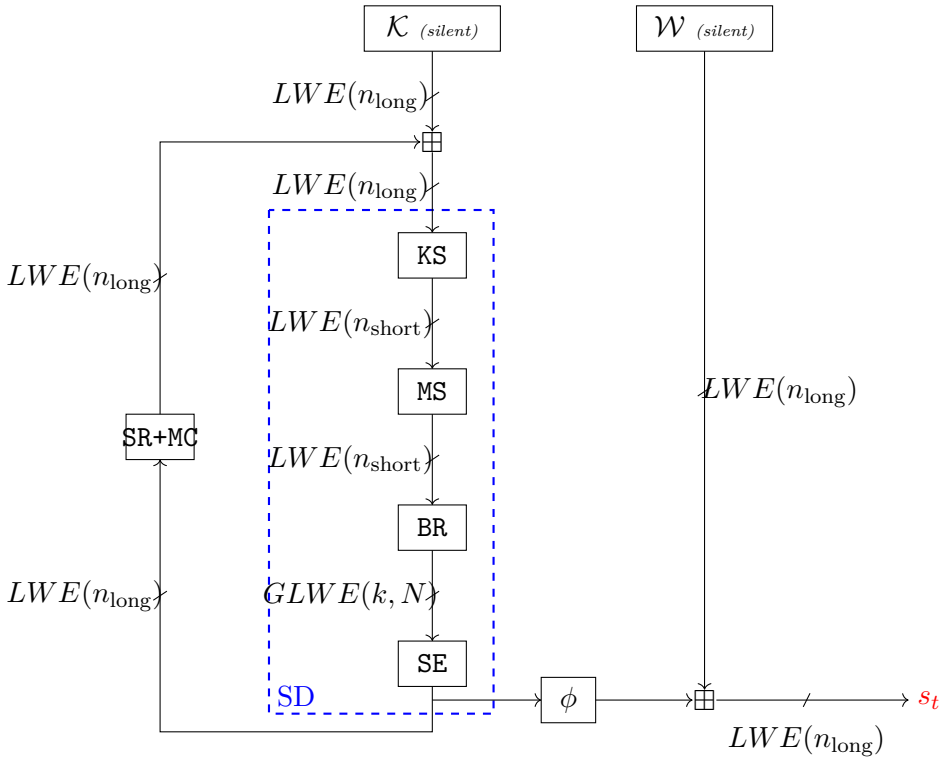


Figure 6.4: Types and shapes of ciphertexts in homomorphic `Transistor`*. The SubDigits is broken down into its elementary components*

ing will be more detailed in Chapter 8 (with additional bits in Appendix A.1). Here, we provide a short reasoning sufficient for our purpose.

In the following, we denote the maximum error probability of the bootstrapping by $p_{\text{err}} := 2^{-\kappa}$. We aim to choose parameters such that the PBS outputs a correct ciphertext with probability at least $1 - p_{\text{err}}$. This translates to the following constraint:

$$\sigma^2_{\text{in-BR}} \leq C(\kappa) \cdot \left(\frac{1}{4p}\right)^2 \quad \text{with} \quad C(\kappa) := \left(\frac{1}{\sqrt{2} \cdot \mathsf{erfc}^{-1}(2^{-\kappa})}\right)^2 .$$

Under the Gaussian assumption, the noise in input of the BlindRotate is lower than $\mathsf{erfc}^{-1}(2^{-\kappa}) \cdot \sqrt{\sigma^2_{\text{in-BR}}}$ with probability $1 - 2^{-\kappa}$. The above constraint thus implies that, with probability $1 - 2^{-\kappa}$, the noise is lower than $1/4p$ which ensures the correctness of the PBS.

As $|\mathcal{W}| \leq |\mathcal{K}|$, we can check that we always have $\sigma^2_{\text{out}} \leq \sigma^2_{\text{in-PBS}}$ which implies that the output is always bootstrappable with correctness probability at least $1 - p_{\text{err}}$ in the subsequent bootstrapping.

6.4.5 Performances

We provide hereafter some benchmarks of our implementation of `Transistor` for two sets of parameters tailored for two different error probabilities. We first consider $p_{\text{err}} = 2^{-40}$, which is a common choice in the literature to benchmark homomorphic implementations. Our results for this error probability allow a fair comparison with the state of the art. While such an error probability theoretically allows transciphering to be error-free with a large amount of data with good probability, some recent works have shown that non-negligible error probabilities could be exploited by an adversary in some contexts [Che+24a; Che+24b]. Thus, we also provide another set of parameters and associated benchmark for $p_{\text{err}} = 2^{-128}$.

Table 6.3: Performances of our two instances of `Transistor`.

p_{err}	Time for one PBS	Latency (one round)	Throughput
2^{-40}	11.9 ms	195 ms	83.84 bits/s
2^{-128}	15.28 ms	251 ms	65.10 bits/s

Table 6.4: Size of the server keys for the two considered sets of parameters.

	Theoretical sizes	Sizes for $p_{\text{err}} = 2^{-40}$	Sizes for $p_{\text{err}} = 2^{-128}$
KSK	$n_{\text{long}} \cdot l_{\text{KS}} \cdot \log_2 q$	49 KB	82 KB
BSK	$n_{\text{short}} \cdot l_{\text{BS}} \cdot \log_2 q \cdot N \cdot (k+1)$	6.5 MB	12.7 MB

Our implementation relies on our customized version of `tfhe-rs` [Zam22c] which has been adapted to support odd p (size of the plaintext space), that we described in Section 4.6.2. The experiments were carried on a processor 12th Gen Intel(R) Core(TM) i5-1245U with 4.4 GHz. Table 6.3 summarizes the obtained timings for the two sets of parameters. The throughput is computed assuming that $\log_2(17)$ bits are encoded on one element of $\mathbb{F}_{17}$. Encoding 4 bits on each element would scale the throughput by a factor $4/\log_2(17) \approx 0.98$.

Although our current implementation does not leverage the inherent parallelism of `Transistor`, it is important to note that it can be easily parallelized across 16 threads. Specifically, during the SubDigits steps, which dominate the overall runtime, the 16 PBS operations can be executed concurrently. This parallelization would result in nearly a 16-fold reduction in total execution time.

Without taking into account the server key(whose sizes are shown in Table 6.4), and using compressed encryption (Section 6.4.1), transciphering 1 KB of plain data requires 1.78 KB of data to be sent, instead of 64 KB. For larger amounts of message, the volume of the encrypted symmetric key becomes negligible with respect to the message: for 1 MB of plain data, we use 1.0008 MB, and for 1 GB, this goes down to 1.000001 GB. The two sets of parameters yield the same bandwidth consumption, but not the same running time as shown in Table 6.3.

By applying the “free truncation optimization" introduced in Section 6.4.1, we can reduce the volume of the encrypted symmetric key by a factor 0.7. This is particularly useful when transciphering a small volume of data (the volume of the encrypted key being preponderant). For example, to transcipher 1 KB of data, using this technique decreases the volume from 1.78 KB to 1.54 KB.

In Table 6.4 we provide the sizes for the server keys, namely the *key-switching key* (KSK) and the *bootstrapping key* (BSK) while using the ciphertext compression technique described in Section 6.4.1. Those keys are only generated and communicated to the server once (during some user enrollment step).

6.4.6 Comparisons to the State of the Art

Comparisons with other TFHE-friendly ciphers.

In what follows, we compare `Transistor` with several of the most competitive state-of-the-art schemes. Our results are summarized in Table 6.5. Although such comparisons must be interpreted with care — due to differences in libraries, hardware platforms, and bootstrapping error probabilities — they still offer valuable insights into the relative efficiency and trade-offs of these approaches.

In [BOS23], `Trivium` and `Kreyvium` were evaluated on a powerful AWS instance, which makes

Table 6.5: Performance of state-of-the-art TFHE-friendly ciphers (single-threaded when applicable). Communication cost accounts for both the encrypted symmetric key and the evaluation keys.

Cipher	Setup	Latency	Throughput	Communication Cost[a]	p_{err}
`Trivium` [BOS23] (128 thr.)	2259 ms	121 ms	529 bits/s	640 B + 35.6 MB †	2^{-40}
`Kreyvium` [BOS23] (128 thr.)	2883 ms	150 ms	427 bits/s	1024 B + 35.6 MB †	2^{-40}
`Margrethe` [Ara+24]	No	27.2 ms	147.06 bits/s	64 MB *	$< 2^{-1000}$
	No	54.2 ms	73.8 bits/s	128 MB *	$< 2^{-1000}$
PRF-based construction [Deo+24]	No	5.675 ms	881 bits/s	32.8 MB = 8.9 MB + 23.9 MB	2^{-64}
`FRAST` [Cho+24]	25 s (8 thr.)	6.2 s	20.66 bits/s	34.05 MB = 148 KB + 33.91 MB	2^{-80}
`Transistor`	No	251 ms	65.10 bits/s	13.54 MB = 780 B + 12.78 MB	2^{-128}

* In `Margrethe`, no keyswitching nor bootstrapping keys are required.
† Values recomputed from the data of the papers. For consistency's sake, we applied the compression of ciphertexts of Section 6.4.1 to estimate the communication cost.

direct comparison with our local experiments impractical. However, an important distinction is that `Transistor` requires no setup phase, unlike these ciphers. Also, the implementation optimizes the running time by switching between two sets of parameters, doubling the size of the evaluation keys.

`Margrethe` [Ara+24] has low noise ratios ($2^{-15.3}$ or $2^{-21.9}$), compared to around 2^{-12} for `Transistor`. Its authors also report low latencies of 27 or 54 ms, and high throughputs of 147 or 73 bits/s (depending on the configuration). However, these come at the cost of much larger sizes for the encryptions of the symmetric key. Indeed, the use of Vertical Packing mandates that symmetric keys be encrypted under GGSW form, resulting in hundreds of megabytes. Although an alternative (lifting from LWE to GGSW using circuit bootstrapping) could reduce the transmission size, it would significantly degrade performance.

Similarly, Deo et al. proposed [Deo+24] a pseudorandom function whose security relies on the hardness of the LWR problem [BPR12]. It enables a stream cipher–like transciphering scheme, where each pseudorandom element in $\mathbb{Z}_p$ (with $p = 2^5$) is produced using a single bootstrapping. Their implementation reaches up to 881 bits/s on their hardware, surpassing `Transistor` in throughput. However, as with `Margrethe`, this efficiency comes at the cost of a significant increase in key size: their PRF requires 500 to 1000 elements encrypted in GGSW form, which is much larger than the 96 elements in LWE form for `Transistor`.

The authors of `FRAST` [Cho+24] implemented it with `tfhe-rs`, like `Transistor`. It targets 128-bit security with an error probability of 2^{-80}. On the same platform, our instance of `Transistor` (with a tighter error bound of 2^{-128}) achieves three times higher throughput, significantly lower latency, and does not require any setup phase—unlike `FRAST`, which involves a 25-second setup. In addition, `FRAST` requires substantially larger evaluation key material, due to its use of multiple derivatives of the PBS algorithm. Finally, no information is provided regarding the output noise level of the scheme.

Comparisons with other homomorphic schemes.

TFHE yields very different trade-offs between the various performance metrics (latency, throughput, bandwith consumption, ...), compared to other FHE schemes. In scenarios where throughput is a priority, TFHE—and by extension, `Transistor`—is generally not the most suitable choice. However, TFHE shines in use cases that require low-latency homomorphic computations and efficient evaluation of look-up tables (LUTs), thanks to its native support for fast programmable bootstrapping. In the following, we provide a brief survey of some transciphering approaches built upon alternative homomorphic encryption schemes.

For CKKS-based transciphering, we look at the AES evaluation using the so-called `XBOOT` optimization proposed in [Niu+25]. While the reported amortized throughput significantly

outperforms that of `Transistor`, it comes at the cost of a much higher latency—153s and 236s using 64 threads, compared to 251ms for `Transistor` running on a single thread. We observe similar results for BGV-based transciphering. In this case we instead consider the optimized evaluation of `RASTA` [Dob+18] presented in [Niu+25]. Using the same `XBOOT` optimization as the CKKS variant, it also yields high latencies: 303s, again with 64-thread parallelism.

Lastly, `FINAL` [Bon+22] is another, more recent homomorphic encryption scheme based on NTRU. Its bootstrapping is approximately 30% faster than TFHE's, but it is not programmable, and therefore does not support LUT evaluation. The works [MPP24; Con+22] implement the stream cipher `Filip` using `FINAL`, relying on the Improved Filter Permutator paradigm, without LUTs' evaluation. The competitive performances of these implementations (resp. 159 bits/s and 381 bits/s) comes with a trade-off in key size: encrypting the symmetric key requires resp. 215 MB and 200 MB, versus 4.8 MB for `Transistor` (uncompressed). This is mainly due to the size of the key register in `Filip` (2^{14} bits), while `Transistor` only requires the upload of 96 elements in $\mathbb{Z}_{17}$ (around 400 bits).

6.5 Conclusion

`Transistor` is a new stream cipher design tailored to TFHE transciphering, that significantly outperforms the state-of-the-art of TFHE-friendly stream ciphers. After analyzing the constraints of the TFHE setting in the context of a symmetric cipher, we designed `Transistor` by combining an LFSR-based key schedule, an LFSR-based whitening, and a non-linear FSM with an AES-like structure. We report implementation results of `Transistor` using state-of-the-art TFHE, using a trick to implement *silent* LFSRs. This general structure can be easily adapted to other contexts, and we believe it will find applications beyond TFHE.

One of the constraint of this design was that the cipher should work in a small plaintext space. This was because the bottleneck in terms of running time was the evaluation of the S-box. But what if we could extend the capabilities of TFHE and evaluates larger S-boxes, like in more conventional designs ?

This is the point of the next chapter, where we propose a construction to accelerate the evaluation of larger LUT.

Chapter 7

Accelerating Large Look-Up Tables

In previous chapters, we have worked with relatively small plaintext spaces, with p taken no bigger than 8 bits. The reason was that TFHE's bootstrapping becomes very slow as p grows, and is considered prohibitive beyond 6 bits. This puts a bound on the size of the Look-Up Tables evaluable with the PBS operation.

One of the use-cases for evaluating large LUT is transciphering. As we have seen in Chapter 6, LUTs operations are the only way to evaluate non-linear operations homomorphically, that are the key to ensure the security of the symmetric schemes used for transciphering. For instance, the 8-bit S-box of AES have been quite challenging to evaluate in Chapter 4 and 5. Another use-case would be to homorphically evaluate the activation function in a neural network, that are intrisically non linear as well. A lot of literature about low-precision machine learning exists, notably for 16-bit precision [Kös+17; Das+18], as well as 8-bit [Sun+19; Cam+20]. In a more general way, a larger LUT operation would be a very useful operator in a generic framework of compilation of homomorphic programs. We took inspiration from the literature about masking, notably the line of work of [CRV14; GR16; Gou+17]. Their goal was to generate circuit representations for S-boxes to construct efficient masked implementations. We use the same kind of techniques for developing a new framework to evaluate large S-boxes using TFHE. It increases the size of evaluable LUTs, which extends the capabilities of TFHE. We give an overview of the performances of our method versus the classical PBS on Figure 7.1. A more advanced construction, called the WoP-PBS (for Without Padding Programmable Bootstrapping) also exists: we present it and compare our work to it in Section 7.4.

Our method outperforms the vanilla PBS algorithm from 6-bit LUT and we extend our experiments until 14-bit ones. In particular, we show better performances than the rest of the literature (including the WoP-PBS) for a bit-size of 8, that corresponds to the size of the S-box of AES.

After formalizing the problem, we give an overview of our method, that relies on a random decomposition of the LUT into smaller ones, and explain how we evaluate those decompositions homomorphically. Then, we explain in-depth the algorithms that produce these decompositions. We finish by presenting the results of our experimentations, by comparing our work to the vanilla PBS and to some constructions of the literature.

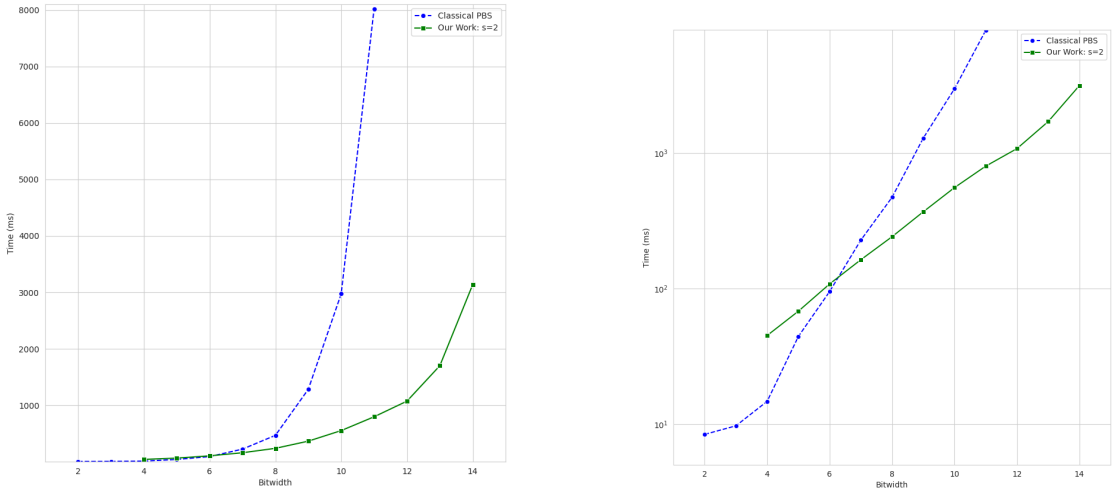
7.1 Context and Formalisation of the Problem

Formally, our goal in this chapter is to homomorphically evaluate a function denoted by

$$F : \mathbb{Z}_s^n \mapsto \mathbb{Z}_s^m \ ,$$

for an input space of size s^n possibly larger than what is efficiently achievable in the current state of the art (to fix ideas, let us say greater than 256).

The operation of look-up table that TFHE natively offers (that is to say the PBS) only takes one element of $\mathbb{Z}_s$ in input and outputs one element of $\mathbb{Z}_s$. To evaluate a function with two



(a) Linear scale *(b) Logarithmic scale*

Figure 7.1: Comparison of the performances of the classical PBS and our method. See Section 7.4 for details about the experimental setup.

inputs x_1 and x_2 from $\mathbb{Z}_s$, one can work in the ring $\mathbb{Z}_{s^2}$ and consider the value $x_1 \cdot s + x_2 \in \mathbb{Z}_{s^2}$, turning the input space of the function from $\mathbb{Z}_s^2$ into $\mathbb{Z}_{s^2}$. Doing this reduces the problem to the case of one LUT with a single input. The problem is that increasing the size of the input space has a catastrophic impact on the performances.

Consequently, the metric to take into account to predict the performances is the value of s^n. This shows that one should either put a limit on the size of the base space $\mathbb{Z}_s$ or on the arity n of the function.

From a high-level perspective, our technique consists in decomposing the function F into a circuit of subfunctions from $\mathbb{Z}_s$ to $\mathbb{Z}_s$, that are much faster than manipulating a massive input of $\mathbb{Z}_s^n$.

7.2 Overview of the Method

To simplify the explanation, we first consider single-output functions (*i.e.*, with $m = 1$) that we denote by

$$f : \mathbb{Z}_s^n \to \mathbb{Z}_s$$
$$\mathbf{x} = (x_0, \ldots, x_{n-1}) \mapsto y.$$

We generalize the technique to functions $F : \mathbb{Z}_s^n \to \mathbb{Z}_s^m$ with $m \geq 1$ in Section 7.3.2.

7.2.1 Building Blocks

Embedding in a prime field. Our method requires elements of $\mathbb{Z}_s$ to be embedded in a finite field. So, if s is not prime, we do not directly manipulate the values x_i's in the ring $\mathbb{Z}_s$.

Instead, we use our encoding method that we introduced in Chapter 4 and generalized to the arithmetic case in Chapter 5. To match with the notation of this chapter, we call it a (s, p)-*encoding*. Recall that this is simply an embedding of the ring $\mathbb{Z}_s$ into a greater space $\mathbb{F}_p$, with $p > s$. We choose p as the smallest prime number greater than s. As a consequence, we identify this new space as the prime field $\mathbb{F}_p$, and we define an injective map function $\mathcal{E} : \mathbb{Z}_s \mapsto \mathbb{F}_p$.

Apart from the choice of p that is crucial for our method to work, the mapping $\mathcal{E}$ itself can be arbitrarily chosen. For the sake of simplicity, we choose the natural "identity" embedding:

$$\begin{aligned} \mathcal{E} : \mathbb{Z}_s &\to \mathbb{F}_p \\ x &\mapsto x \end{aligned}$$

As $s \leq p$, this always defines a valid injective map. In the rest of the paper, we will use x_i to denote indifferently the original value living in $\mathbb{Z}_s$ and its embedding in $\mathbb{F}_p$. Moreover, we replace the function f we wish to evaluate by its embedding in $\mathbb{F}_p$:

$$\begin{aligned} \bar{f} : \mathrm{Im}(\mathcal{E})^n &\to \mathbb{F}_p \\ \mathbf{x} = (x_0, \ldots, x_{n-1}) &\mapsto \mathcal{E}(f(\mathcal{E}^{-1}(x_0), \ldots, \mathcal{E}^{-1}(x_{n-1}))) \end{aligned}$$

where $\mathcal{E}^{-1} : \mathrm{Im}(\mathcal{E}) \mapsto \mathbb{Z}_s$ refers to the function mapping the elements of $\mathbb{F}_p$ to the original value of $\mathbb{Z}_s$ they encode.

Construction of a pool of derived variables. Our method requires constructing a pool of variables derived from the input of the function f. For reason that will soon be clear, these variables should be *linearly independent* from the inputs. To achieve that, we define a notion of *atomic*[1] *function of arity* r as follows:

Definition 7.2.1. Let $\mathcal{F}_p$ be the set of all functions from $\mathbb{F}_p$ to $\mathbb{F}_p$, and r be an integer. An atomic function of arity r is a function ϕ of the form

$$\begin{aligned} \phi : \mathbb{F}_p^r &\to \mathbb{F}_p \\ (x_0, \ldots, x_{r-1}) &\mapsto \psi \left(\sum_{j=0}^{r-1} \alpha_j \cdot x_j \right) \end{aligned}$$

with $\psi \in \mathcal{F}_p$ and $\boldsymbol{\alpha} = (\alpha_j)_{0 \leq j < r} \in \mathbb{F}_p^r$.

If an atomic function is sampled at random such that ψ is non-linear, then its output is linearly independent from the inputs with great probability.

Such an atomic function can be evaluated homomorphically using the native operations of TFHE. The procedure is described in Algorithm 11.

A few remarks on this straightforward algorithm are in order:

- The algorithmic cost of `EvalAtom` can be boiled down to the cost of the final PBS, as the other operations are linear so essentially free.
- The noise in the result is reset to a nominal level by the final PBS.
- The noise growth in the linear combination is exactly quantified by the norm of $\boldsymbol{\alpha}$. Trivially, the standard deviation of the noise in the coefficients of the ciphertexts is multiplied by $\|\boldsymbol{\alpha}\|$.

Now, we define a notion of *chain of atomic functions*:

Definition 7.2.2. Let $n, \lambda \in \mathbb{N}$. A chain of atomic functions $\Phi_{n,\lambda}$ of basis n and length λ is a function defined as follows:

$$\begin{aligned} \Phi_{n,\lambda} : \mathbb{F}_p^n &\to \mathbb{F}_p^\lambda \\ (x_0, \ldots, x_{n-1}) &\mapsto (x_n, \ldots, x_{n+\lambda-1}) \end{aligned}$$

[1] This denomination is inspired by the notion of *atomic pattern* from [Ber+23a]

Algorithm 11 `EvalAtom` - Homomorphic evaluation of an atomic function

Context: $\begin{cases} \mathbb{F}_p\text{: the field of embedding.} \\ (x_0, \ldots, x_{r-1}) \in \mathbb{F}_p^r\text{: the inputs of the atomic function in the clear.} \end{cases}$

Input: $\begin{cases} \phi : \mathbb{F}_p^r \mapsto \mathbb{F}_p\text{: the atomic function of arity } r \text{ we want to evaluate.} \\ (\mathbf{c}_0, \ldots, \mathbf{c}_{r-1})\text{: the ciphertexts encrypting the inputs} \\ \boldsymbol{\alpha} = (\alpha_j)_{0 \leq j < r} \in \mathbb{F}_p^r\text{: the coefficients of the linear combination of } \phi \\ \psi\text{: the outer function of } \phi. \end{cases}$

Result: $\mathbf{c}_r$: an encryption of $x_r = \phi(x_0, \ldots, x_{r-1})$

$\mathbf{c}_r \leftarrow 0$
`/* Evaluation of the linear combination */`
for $k \in \{0, \ldots, r-1\}$ **do**
  $\mathbf{c}_r \leftarrow \texttt{SumTFHE}(\mathbf{c}_r, \texttt{ClearMultTFHE}(\mathbf{c}_k, \alpha_k))$
end
`/* Evaluation of the final function` ψ `with a bootstrapping */`
$\mathbf{c}_r \leftarrow \texttt{PBS\_TFHE}(\mathbf{c}_r, \psi)$
return $\mathbf{c}_r$

with

$$\forall k \in \{0, \ldots, \lambda - 1\}, \quad x_{n+k} = \phi_k(x_0, \ldots, x_{n+k-1}) := \psi_k \left(\sum_{j=0}^{n+k-1} \alpha_{k,j} \cdot x_j \right)$$

from a set of λ atomic functions $(\phi_k)_{0 \leq k < \lambda}$ of arity $(n+k)_{0 \leq k < \lambda}$ and with $(\alpha_{k,j})_{\substack{0 \leq k < \lambda \\ 0 \leq j < n+k}} \in \mathbb{F}_p$.

Concretely, the k-th *link* of the chain is an atomic function ϕ_k taking in input:

- the n inputs of the chain, and
- the $k-1$ outputs of the previous links of the chain

Homomorphically evaluating a chain is done by simply calling `EvalAtom` (Algorithm 11) on each link in the chain.

7.2.2 Core of the Method

Decomposition of the function. The goal of our method is to decompose the target function f, using a chain of atomic functions $\Phi_{n,\lambda}$, into a form:

$$\forall \mathbf{x} \in \mathbb{Z}_s^n, f(x_0, \ldots, x_{n-1}) = \sum_{i=0}^{t-1} \Big(\sum_{j=0}^{L-1} \beta_{i,j} \cdot x_j \Big) \cdot \Big(\sum_{k=0}^{L-1} d_{i,k} \cdot x_j \Big) \tag{7.1}$$

where $\lambda, t \in \mathbb{N}$, $L = n + \lambda$, $\beta_{i,j}$ (resp. $d_{i,k}$) $\in \mathbb{F}_p$ for $0 \leq i < t$ and $0 \leq j < L$ (resp. k), and $x_i \in \Phi_{n,\lambda}(x_0, \ldots, x_{n-1})$ for $n \leq i < L$.

We present a method to construct such decompositions in Section 7.3. Now, we simply explain how to evaluate it using the native operations of TFHE.

Homomorphic evaluation of the decomposition. To homomorphically evaluate this decomposition of f, we need a few FHE operations:

- All the x_i's are produced by homomorphically evaluating the chain $\Phi_{n,\lambda}$, that is to say by successive calls to `EvalAtom`.

- As every terms of the formula lives in $\mathbb{F}_p$, the products plaintext-ciphertext (i.e. the products $\beta_{ij} \cdot x_j$ and $d_{i,j} \cdot x_j$) are simply evaluated using the native `ClearMultTFHE` operation. The sums are very easy as well: they are done with `SumTFHE`.

- Evaluating the multiplication between two ciphertexts is trickier. One can use a PBS with two inputs, but this would lead to poor performances. Instead, a trick appearing in [Ber+23a] is to leverage the following property:

$$x \cdot y = \left[4^{-1}\right]_p \cdot ((x+y)^2 - (x-y)^2) \ . \tag{7.2}$$

  Both squaring operations can be evaluated using a simple PBS. Evaluating two PBS on entries of size p is significantly more efficient that one PBS of size p^2 (as illustrated by Figure 7.1, doubling the bit-size of the input is devastating for the performances). We could evaluate the multiplication by $\left[4^{-1}\right]_p$ using `ClearMultTFHE` but that would lead to a slight noise growth. Instead, we include it into the tables evaluated by the PBS. This procedure is formalized in Algorithm 12.

Algorithm 12 `EncryptedProduct` - Homomorphic evaluation of a ciphertext-ciphertext product

Context: $\begin{cases} \mathbb{F}_p \text{: the field of embedding.} \\ x_1, x_2 \in \mathbb{F}_p \text{ : the inputs of the product in the clear.} \end{cases}$
Input: $\mathbf{c}_1, \mathbf{c}_2$: the ciphertexts encrypting the inputs x_1 and x_2.
Result: $\mathbf{c}'$: an encryption of $x_1 \cdot x_2$

```
/* We compute the sum and the difference                                    */
```
$\boldsymbol{\sigma}_1 \leftarrow \texttt{SumTFHE}(\mathbf{c}_1, \mathbf{c}_2)$
$\boldsymbol{\sigma}_2 \leftarrow \texttt{SumTFHE}(\mathbf{c}_1, \texttt{MulTFHE}(\mathbf{c}_2, [-1]_p))$

```
/* Evaluation of the squares (we precompute the minus sign in front of the
   second one)                                                              */
```
$\boldsymbol{\sigma}'_1 \leftarrow \texttt{PBS\_TFHE}(\boldsymbol{\sigma}_1, x \mapsto x^2 \cdot \left[4^{-1}\right]_p)$
$\boldsymbol{\sigma}'_2 \leftarrow \texttt{PBS\_TFHE}(\boldsymbol{\sigma}_2, x \mapsto x^2 \cdot \left[-4^{-1}\right]_p)$

```
/* Summing the square                                                       */
```
$\mathbf{c}_r \leftarrow \texttt{SumTFHE}(\boldsymbol{\sigma}'_1, \boldsymbol{\sigma}'_2)$

return $\mathbf{c}_r$

Finally, putting everything together, we obtain Algorithm 13 for the homomorphic evaluation of the decomposition. We can estimate the cost of this algorithm by counting the number of PBS it requires:

- λ PBS to evaluate the chain,
- $2t$ PBS to evaluate the t ciphertext-ciphertext multiplications using Equation 7.2.

The total cost in PBS can thus be evaluated to $\lambda + 2t$.

7.3 Finding Efficient Decompositions

In Section 7.2, we have described the case where the function F has only one output ($m = 1$). If we want to evaluate functions with several outputs ($m > 1$), a trivial method would be to

Algorithm 13 Homomorphic evaluation of a decomposition

Context: $\begin{cases} f : \mathbb{F}_p^n \mapsto \mathbb{F}_p : \text{the function to evaluate.} \\ (x_0, \ldots, x_{n-1}) \in \mathbb{F}_p^n : \text{the clear inputs of the function.} \\ \Phi_{n,\lambda} = (\phi_0, \ldots, \phi_{\lambda-1}) : \text{the chain used to generate the decomposition} \\ L = n + \lambda. \end{cases}$

Input: $\begin{cases} (\mathbf{c}_0, \ldots, \mathbf{c}_{n-1}) : \text{the ciphertexts encrypting the inputs} \\ (\beta_{i,j})_{\substack{0 \le i < t \\ 0 \le j < L}} \in \mathbb{F}_p^{t \times L} : \text{coefficients of the left-hand linear combination of Equation 7.1.} \\ (d_{i,j})_{\substack{0 \le i < t \\ 0 \le j < L}} \in \mathbb{F}_p^{t \times L} : \text{coefficients of the right-hand linear combination of Equation 7.1.} \end{cases}$

Result: $\mathbf{c}_y$: an encryption of $y = f(x_0, \ldots, x_{n-1})$

```
/* Evaluation of the chain of atomic functions                              */
for k ∈ {n, ..., L} do
    c_k ← EvalAtom(φ_k, (c_0, ..., c_{k-1}))
end
c_y ← 0
for i ∈ {0, ..., t − 1} do
    /* We compute the linear combinations with the β's and d's              */
    for j ∈ {0, ..., L − 1} do
        γ_ij ← ClearMultTFHE(c_j, β_ij)
        γ'_ij ← ClearMultTFHE(c_j, d_ij)
    end
    σ_i ← SumTFHE(γ_i0, ..., γ_{i,L−1})
    σ'_i ← SumTFHE(γ'_i0, ..., γ'_{i,L−1})
    /* We compute the encrypted product using the trick of Algorithm 12     */
    c_y ← SumTFHE(c_y, EncryptedProduct(σ_i, σ'_i))
end
return c_y
```

consider the m functions $f_i : \mathbb{F}_p^n \mapsto \mathbb{F}_p$, finding a decomposition for each one, and evaluating them in parallel.

However, we found out that all the decompositions can be constructed using the same chain, which allows to save a lot of PBS. In this section, we describe first how we construct an efficient decomposition for the first output. Then, we show the intermediary results can be recycled to construct the next ones for much cheaper that with full fresh decompositions.

7.3.1 Construction of an Efficient Decomposition for the First Output

For now, and as in Section 7.2, we only consider a function with one output $f : \mathbb{F}_p^n \mapsto \mathbb{F}_p$. It can be seen as the first subfunction of the whole function $F : \mathbb{F}_p^n \mapsto \mathbb{F}_p^m$.

In the rest of the chapter, we denote by $\mathbf{x}^{(i)}$ the i-th[2] element of $\mathbb{Z}_s^n$. It gives:

$$\mathbb{Z}_s^n = \left\{ \mathbf{x}^{(i)} = (x_0^{(i)}, \dots, x_{n-1}^{(i)}) \mid 0 \leq i < s^n \right\} .$$

Construction of a Valid Decomposition

Before trying to find an efficient decomposition, let us try to construct a *valid* decomposition. We start by choosing appropriate λ and t values and construct a random chain of atomic functions $\Phi_{n,\lambda}$ (we address the choice of λ and t later in this section).

Recall Equation 7.1. It can be rewritten as a matrix-vector equation, where each row of the matrix corresponds to a different input of the chain:

$$\mathbf{y} = \mathcal{A} \cdot \boldsymbol{\beta} \tag{7.3}$$

where:

- $\mathbf{y} = (y_0, \dots, y_{s^n-1})^\top \in \mathbb{F}_p^{s^n}$ is a vector containing the s^n outputs of the function f:

$$\forall i \in \mathbb{Z}_{s^n}, y_i = f(x_0^{(i)}, \dots, x_{n-1}^{(i)})$$

- $\boldsymbol{\beta} = (\beta_0, \dots, \beta_{tL-1})^\top \in \mathbb{F}_p^{tL}$ is a flattened container containing the betas, such that:

$$\forall i, j \in \mathbb{Z}_t \times \mathbb{Z}_L, \beta_{iL+j} = \beta_{i,j} .$$

- $\mathcal{A} \in \mathbb{F}_p^{s^n \times tL}$ is the matrix defined by block as:

$$\mathcal{A} = (\mathcal{A}_0 \mid \mathcal{A}_1 \mid \dots \mid \mathcal{A}_{t-1}) \quad \text{with:}$$

$$\mathcal{A}_i = \begin{pmatrix} x_0^{(0)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(0)} \right\rangle & \dots & x_{L-1}^{(0)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(0)} \right\rangle \\ x_0^{(1)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(1)} \right\rangle & \dots & x_{L-1}^{(1)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(1)} \right\rangle \\ \vdots & \vdots & \vdots \\ x_0^{(s^n-1)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(s^n-1)} \right\rangle & \dots & x_{L-1}^{(s^n-1)} \cdot \left\langle \mathbf{d}_i, \mathbf{x}^{(s^n-1)} \right\rangle \end{pmatrix} \tag{7.4}$$

$$\text{and: } \mathbf{d}_i = (d_{i,0}, \dots, d_{i,L-1}).$$

Putting everything together, we can write the whole product of Equation 7.3, which gives:

[2] The sort order does not matter, so we can arbitrarily pick the lexicographical order.

$$s^n \left\{ \begin{pmatrix} y_0 \\ y_1 \\ \vdots \\ \vdots \\ y_{s^n-1} \end{pmatrix} \right. = \overbrace{\left(\begin{array}{c|c|c|c} & & & \\ \mathcal{A}_0 & \mathcal{A}_1 & \dots & \mathcal{A}_{t-1} \\ & & & \end{array}\right)}^{t \cdot L} \cdot \left(\begin{array}{c} \beta_{0,0} \\ \vdots \\ \beta_{0,L-1} \\ \hline \vdots \\ \vdots \\ \hline \beta_{t-1,0} \\ \vdots \\ \beta_{t-1,L-1} \end{array}\right) \tag{7.5}$$

To make the link with the notion of chain, we define the *matrix representation of a chain*:

Definition 7.3.1. Let $\Phi_{n,\lambda}$ a chain of atomic functions of basis n and length λ. The *matrix representation* of the chain $\Phi_{n,\lambda}$ is a matrix of size $s^n \times (n+\lambda)$ whose i-th row stores the concatenation of the inputs and the outputs of the evaluation of $\Phi_{n,\lambda}$ on the input $\mathbf{x}^{(i)}$. Thus, it can be written:

$$\texttt{Mat}(\Phi_{n,\lambda}) = \begin{pmatrix} x_0^{(0)} & \dots & x_{n-1}^{(0)} & x_n^{(0)} & \dots & x_{n+\lambda-1}^{(0)} \\ x_0^{(1)} & \dots & x_{n-1}^{(1)} & x_n^{(1)} & \dots & x_{n+\lambda-1}^{(1)} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_0^{(s^n-1)} & \dots & x_{n-1}^{(s^n-1)} & x_n^{(s^n-1)} & \dots & x_{n+\lambda-1}^{(s^n-1)} \end{pmatrix} \in \mathbb{F}_p^{s^n \times (n+\lambda)}$$

where $x_{n+k}^{(i)} = \phi_k(x_0^{(i)}, \dots, x_{n+k-1}^{(i)})$ for $0 \leq k < \lambda$ and $0 \leq i < s^n$.

This matrix can be seen as the concatenation of two blocks:

$$\texttt{Mat}(\Phi_{n,\lambda}) = \left(\begin{array}{c|c} \texttt{Triv} & \texttt{Eval} \end{array} \right)$$

where `Triv` is the trivial writing of all possible inputs of the chain (so all the elements of $\mathbb{Z}_{s^n}$) and each row of `Eval` corresponds to evaluation of the chain with the corresponding row of `Triv` as input.

Using this notation, the matrix $\mathcal{A}$ can alternatively be defined as $\mathcal{A} = \mathcal{V} \square \mathcal{U}$, where:

- $\mathcal{U}$ is the matrix representation of the chain $\Phi_{n,\lambda}$.
- $\mathcal{V}$ is the matrix defined by:
$$\mathcal{V} = \mathcal{U} \cdot \mathcal{D}^T \tag{7.6}$$
where $\mathcal{D}$ is the matrix containing all the d_{ij}'s:
$$\mathcal{D} = \begin{pmatrix} \mathbf{d}_0 \\ \mathbf{d}_1 \\ \vdots \\ \mathbf{d}_{t-1} \end{pmatrix} = \begin{pmatrix} d_{0,0} & \dots & d_{0,L-1} \\ d_{1,0} & \dots & d_{1,L-1} \\ \vdots & \ddots & \vdots \\ d_{t-1,0} & \dots & d_{t-1,L-1} \end{pmatrix} \in \mathbb{F}_p^{t \times L} \tag{7.7}$$
- $\square$ denotes the *face-splitting* product of Slyusar [Sly99], illustrated on Figure 7.2.

What we do in practice is sampling a random chain $\Phi_{n,\lambda}$ and a random matrix $\mathcal{D}$ and treat $\boldsymbol{\beta}$ as an unknown. To check whether the pair $(\Phi_{n,\lambda}, \mathcal{D})$ produces a valid decomposition of f, we need to find a vector $\boldsymbol{\beta}$ satisfying Equation 7.5. For this, it is sufficient that $\texttt{Rank}(\mathcal{A}) = s^n$ (so full row-rankness), which allows to perform a Gaussian elimination to obtain the value of the vector $\boldsymbol{\beta}$. If the matrix is rank-deficient, we sample a new chain and matrix and start the procedure

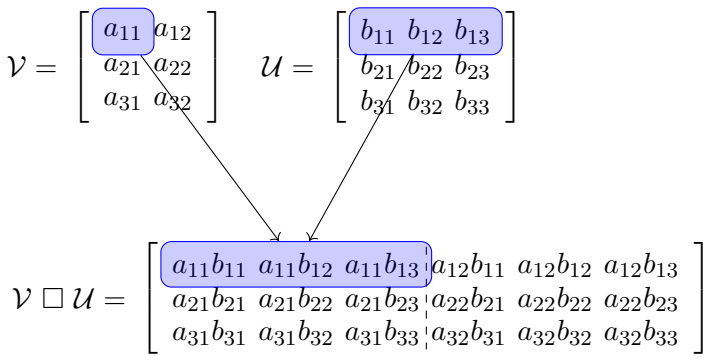


Figure 7.2: Illustation of the face-splitting product.

again. Note that for $\mathcal{A}$ to be full row-rank, it is necessary (but not sufficient) that $\mathcal{D}$ be full row-rank as well.

One of the strength of this method is that the pair $(\Phi_{n,\lambda}, \mathcal{D})$ and the underlying matrix $\mathcal{A}$ can be used for *any* function f. Indeed, the definition of f only impacts the target vector $\mathbf{y}$ in the system. So once a pair $(\Phi_{n,\lambda}, \mathcal{D})$ yielding a full row-rank matrix $\mathcal{A}$ has been found, it can be used for any function f of same arity n.

Bound on the dimensions of $\mathcal{A}$. We just showed that to produce a valid decomposition, the matrix $\mathcal{A}$ needs to be *full row-rank*. As long as $\mathcal{D}$ is full row-rank, then the columns of $\mathcal{A}$ are linearly independent with great probability so the only condition for full row-rankness is that it should have at least more columns than rows. This gives the simple inequality for the decomposition parameters:

$$s^n \leq t \cdot (n + \lambda). \tag{7.8}$$

To wrap up, the method to find a valid decomposition consists in drawing random chains $\Phi_{n,\lambda}$ and random matrices $\mathcal{D} \in \mathbb{F}_p^{t \times L}$ until the matrix $\mathcal{A}$ is full row-rank. As its columns are products of results of random functions, linear relationships between them are unlikely and in practice a solution is quickly found whenever the parameters satisfy Equation 7.8.

Performances of the search algorithm. Our experiments show very different time to find a solution with respect to the value of n. Two phenomenons explain this variability:

- The `GaussElimination` algorithm has cubic complexity with the number of rows of $\mathcal{A}$ (which is s^n). The combination of both factors makes the computational cost of testing one chain increase largely with the value of n. On Figure 7.3, we give the time of executions [3] of the algorithm for one chain.
- The probability that a random chain (associated with a random $\mathcal{D}$) yields a valid decomposition varies a lot, and seems intuitively inversely correlated with the tightness of the bound of Equation 7.8. Thus, a trade-off to speed-up the search algorithm would be to relax this bound by adding a multiplicative factor $\gamma > 1$ on the left-hand side of Equation 7.8 to allow for more margin, which gives

  $$\gamma \cdot s^n \leq t \cdot (n + \lambda)$$

  Our experimental results show that a value of $\gamma = 1.1$ is enough to find a solution in less than 200 iterations. The impact of γ can be visualized on Figure 7.4. Increasing the value of γ comes to the cost of picking slightly larger values for λ and t, which slows down a bit the homomorphic evaluation of the decomposition.

[3] The machine on which we run the tests is a server equipped with an AMD Ryzen Threadripper PRO 7995WX of 96 cores.

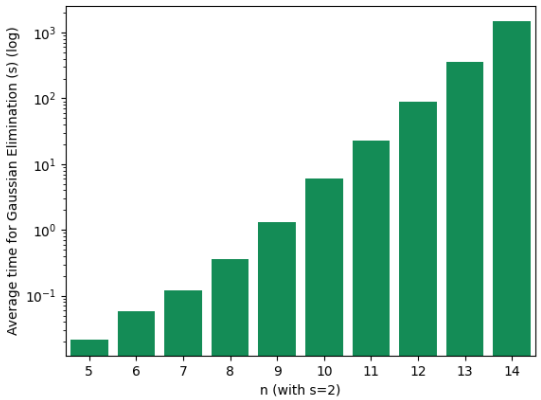


Figure 7.3: Average time to run `GaussElimination` *in function of* n*, with* $s = 2$ *and* $p = 3$*.*

Still, the cost of finding one valid decomposition stays quite reasonable for values of n up to 14, which we picked as an arbitrary upper bound for the arity of the functions in our experiments in Section 7.4.

However, in order to be able to optimize the speed of the homomorphic evaluation of the decomposition, we need a *cost model* to quantify the quality of a decomposition. This allows to generate several valid decompositions and select the best one. We define such a cost model hereafter.

Selection of the Best Decomposition

In TFHE, the PBS is by far the slowest operation by several orders of magnitude. Thus, it may seem that finding an efficient decomposition can be boiled down to reducing the number of PBS. However, the speed of the PBS itself is dependent the parameters picked for the TFHE scheme. Thankfully, extensive work has been done to try to optimize the parameters of TFHE in a given context to minimize the running time of the PBS operation while maintaining the correctness of the homomorphic computation. In the following, we explain how we select parameters in our own optimization framework.

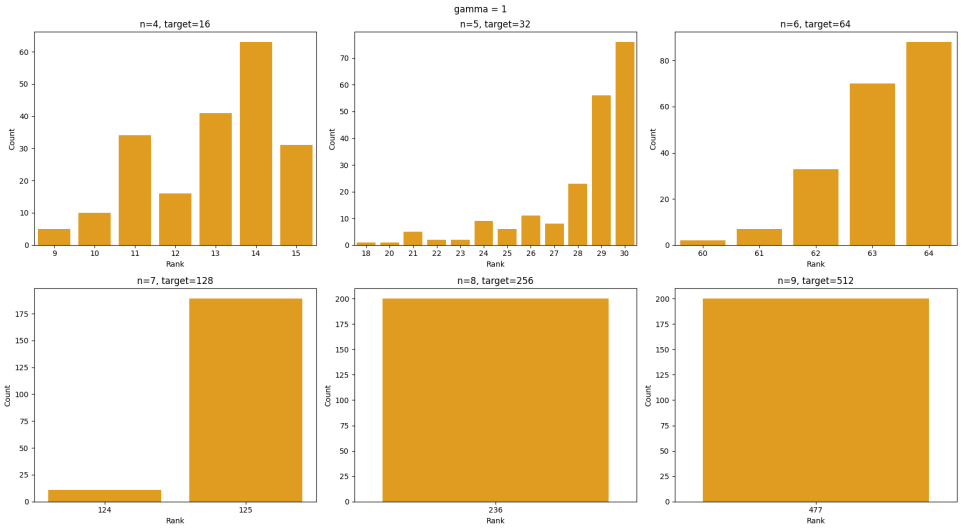
In Chapter 8, we present a tool to generate parameters sets for TFHE. Briefly, the elements to take into account to select the right parameters are:

- the target security level λ, that corresponds to the computational hardness of the underlying LWE instance,
- the error probability ϵ, which is the probability that the noise overflows over some bits of the message during the computation,
- the size of the embedding field p,
- the norms of the linear combinations the ciphertexts go through. Such linear combinations happen in the computations of the atomic functions ($\|\boldsymbol{\alpha}_i\|$), and in the evaluation of the decomposition itself ($\|\boldsymbol{\beta}_i\|$ and $\|\mathbf{d}_i\|$). In practice, we consider the worst one of all the circuit ν_{max}.

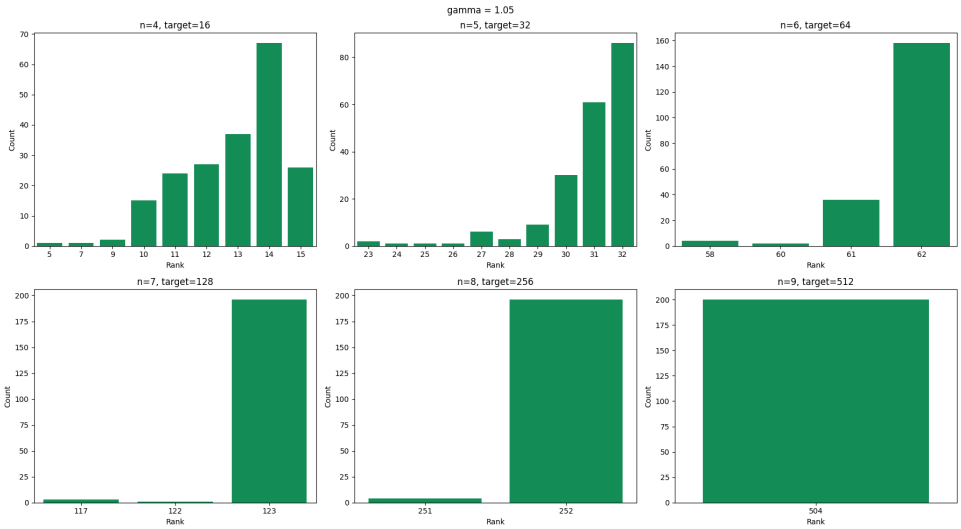
Using our tool, we dynamically crafted a set of parameters for every candidate decomposition we generated. For more details about the methods of selection, we refer to Chapter 8.

We can abstract the use of our tool by modelling the computational cost of each PBS in a given decomposition by an oracle (which in practice corresponds to a call to our tool):

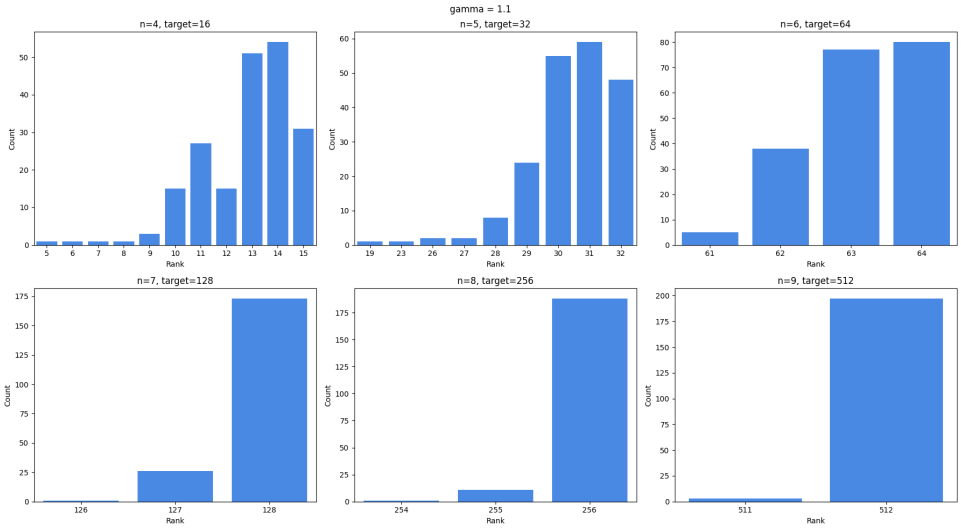
$$Cost(PBS) = \mathcal{O}(\lambda, \epsilon, p, \nu_{max}) \ .$$



(a) $\gamma = 1$



(b) $\gamma = 1.05$



(c) $\gamma = 1.1$

Figure 7.4: Distribution of the ranks of 200 matrices $\mathcal{A}$ generated using our algorithm, with different values of n and different tolerance values γ. Increasing γ to 1.1 almost guarantees to find a quickly find a full-rank matrice. The experiments have been carried out for $s = 2$, $p = 3$ and a function with one single output.

Now that we have a model for the cost of *one* PBS, we can simply estimate the cost of the whole decomposition by multiplying this cost with the number of required PBS. This gives:

$$Cost = Cost(PBS) \times \#PBS$$

with:

$$\#PBS = \lambda + 2t$$

For the sake of clarity, we can break down this last equation: the evaluation of the chain takes λ PBS, and the t products are evaluated in $2t$ PBS.

Wrapping up, we now have a way to construct valid decompositions for single-output function by sampling random chains, and a way to estimate their cost.

7.3.2 Generalization to Several Outputs

We just explained a way to compute a function

$$f : \mathbb{Z}_s^n \mapsto \mathbb{Z}_s.$$

Now, we generalize our technique to multi-outputs functions:

$$F : \mathbb{Z}_s^n \mapsto \mathbb{Z}_s^m.$$

The idea is to first evaluate the first output (that we denote f_0) using the technique introduced previously, and then reuse as much computation as possible to speed up the evaluation of the $m-1$ other outputs of F (namely, the f_k with $k \in \{1, \dots, m-1\}$). An obvious first optimization would be to re-use the same chain to evaluate the λ values $(x_n, \dots, x_{n+\lambda-1})$, which would save $(m-1) \times \lambda$ PBS. But, by taking inspiration from [GR16; Gou+17], we can push this further.

Indeed, after having completed the evaluation of the first output, we dispose of several variables that are linearly independent:

- the n fresh inputs of the function f_0,
- the λ random variables that we evaluated using the chain,
- the t_0 results of products $\{\rho_i\}_{0 \leq i < t_0}$ defined by :

$$\forall 0 \leq i < t_0, \rho_i = \left(\sum_{j=0}^{n_0-1} \beta_{ij} \cdot x_j \right) \cdot \left(\sum_{j=0}^{n_0-1} d_{ij} \cdot x_j \right) \tag{7.9}$$

  with $n_0 = n + \lambda$.

This accounts for a total of $n_1 = n_0 + t_0$ variables that we can use as a new base. Now we can pick a new value t_1, which is bounded by Equation 7.10:

$$s^n \leq n_1 \cdot t_1 \tag{7.10}$$

If we compare with Equation 7.8, t_1 can be smaller than the previous value t_0. Remember than t_1 corresponds to the number of ciphertext-ciphertext products in the decomposition (that each requires 2 PBS): so as we evaluate more outputs, the fewer PBS are required for each one.

We then follow the same procedure than for the first output: we sample a new random matrix $\mathcal{D}^{(1)} \in \mathbb{F}_p^{t_1 \times L}$ and we construct a new matrix $\mathcal{A}^{(1)} = \mathcal{V}^{(1)} \square \mathcal{U}^{(1)} \in \mathbb{F}_p^{s^n \times t_1 \cdot n_1}$ where $\mathcal{U}^{(1)}$ is the matrix representation of the chain $\Phi_{n,\lambda}$ augmented with t_0 columns containing the

corresponding results of products $(\rho_i)_{0\leq i<t}$ and $\mathcal{V}^{(1)} = \mathcal{U}^{(1)} \cdot (D^{(1)})^T$. The matrix $\mathcal{U}^{(1)}$ can be defined as:

$$\mathcal{U}^{(1)} = \begin{pmatrix} \mathcal{U}^{(0)} & \mathcal{P}^{(0)} \end{pmatrix} \in \mathbb{F}_p^{s^n \times n_1} \tag{7.11}$$

where $\mathcal{P}^{(0)} \in \mathbb{F}_p^{s^n \times t_0}$, as explained previously, compiles the results of each products.

If $\mathcal{A}^{(1)}$ is full-rank, it is then possible to solve the equation:

$$\mathbf{y}^{(1)} = \mathcal{A}^{(1)} \cdot \boldsymbol{\beta}^{(1)} \tag{7.12}$$

and construct a decomposition. The same procedure can then be carried out iteratively on the rest of the outputs of the function F.

Algorithm 14 `CreateDecompositions` - Generation of the decompositions for a function F

Context: $\mathbb{F}_p$: The finite field of embedding
Input: $F : \mathbb{Z}_s^n \mapsto \mathbb{Z}_s^m$: the function to evaluate.
Result: $\left(\boldsymbol{\beta}^{(0)}, \ldots, \boldsymbol{\beta}^{(m-1)}\right) \in \prod_{k=0}^{m-1} \mathbb{F}_p^{t_k \cdot}$: the coefficients of each decomposition

$\lambda \leftarrow$ `OptimalShape`(s, n)
$n_0 \leftarrow n + \lambda$
$\Phi^{(base)} \xleftarrow{\$} \Phi_{n,\lambda}$; `/* Sample a random chain */`
$\mathcal{U}^{(0)} \leftarrow$ `Mat`$(\Phi^{(base)})$
$t_0 = \left\lceil \frac{s^n}{n_0} \right\rceil$
for $k \in \{0, \ldots, m-1\}$ **do**
  `/* Sample random chains and matrices` $\mathcal{D}$`, until it produces a full-rank matrix */`
  do
    $\mathcal{D}^{(k)} \xleftarrow{\$} \mathbb{F}_p^{t_k \times n_k}$
    $\mathcal{V}^{(k)} \leftarrow \mathcal{U}^{(k)} \cdot (D^{(k)})^T$
    $\mathcal{A}^{(k)} \leftarrow \mathcal{V}^{(k)} \square \mathcal{U}^{(k)}$
  while *Rank*$(A^{(i)}) < s^n$;
  `/* Solve the linear system to get the coefficients of the decomposition */`
  $\boldsymbol{\beta}^{(k)} \leftarrow$ `GaussElimination`$(\mathcal{A}^{(k)}, \mathbf{y}^{(k)})$
    `/* Compute intermediary products */`
  $\mathcal{P}^{(k)} \leftarrow$ `ComputeProducts`$(\mathcal{A}^{(k)}, \boldsymbol{\beta}^{(k)})$
    `/* Append them to the pool of available variables */`
  $\mathcal{U}^{(k+1)} \leftarrow \left(\mathcal{U}^{(k)} | \mathcal{P}^{(k)}\right)$
    `/* Compute the shape of the decomposition of the next output */`
  $n_{k+1} \leftarrow n_k + t_k$
  $t_{k+1} \leftarrow \left\lceil \frac{s^n}{n_k} \right\rceil$
end
return $\left(\boldsymbol{\beta}^{(0)}, \ldots, \boldsymbol{\beta}^{(m-1)}\right)$

We wrap this up into Algorithm 14. We make use of a few subroutines that we explain in the following:

`OptimalShape` : The shape of the matrices for each decomposition can be defined recursively:

s	p	n	λ	t_0	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}	t_{11}	t_{12}	t_{13}	#PBS
2	3	4	2	3	2	2	2	-	-	-	-	-	-	-	-	-	-	20
2	3	5	4	4	3	2	2	2	-	-	-	-	-	-	-	-	-	30
2	3	6	13	4	3	3	3	2	2	-	-	-	-	-	-	-	-	47
2	3	7	20	5	4	4	4	3	3	3	-	-	-	-	-	-	-	72
2	3	8	25	8	7	6	5	5	4	4	4	-	-	-	-	-	-	111
2	3	9	48	9	8	7	7	6	6	6	5	5	-	-	-	-	-	166
2	3	10	69	13	12	10	9	9	8	8	7	7	7	-	-	-	-	249
2	3	11	103	18	16	14	13	12	11	11	10	10	9	9	-	-	-	369
2	3	12	165	24	21	19	17	16	15	15	14	13	13	12	12	-	-	547
2	3	13	244	32	29	26	24	23	21	20	19	19	18	17	17	16	-	806
2	3	14	390	41	37	34	32	30	29	27	26	25	24	24	23	22	22	1182

(a) $s = 2$

s	p	n	λ	t_0	t_1	t_2	t_3	t_4	t_5	t_6	#PBS
4	5	2	4	3	2	-	-	-	-	-	14
4	5	3	10	5	4	3	-	-	-	-	34
4	5	4	26	9	7	6	5	-	-	-	80
4	5	5	60	16	13	11	10	9	-	-	178
4	5	6	146	27	23	21	19	17	16	-	392
4	5	7	297	54	46	41	37	34	32	30	845

(b) $s = 4$

s	p	n	λ	t_0	t_1	t_2	t_3	#PBS
16	17	2	25	10	7	-	-	59
16	17	3	113	36	27	23	-	285
16	17	4	487	134	105	90	80	1305

(c) $s = 16$

Table 7.1: Optimal parameters for the shapes of the matrices for different values of s and n.

$$\begin{cases} n_0 = n + \lambda \\ t_k = \left\lceil \frac{s^n}{n_k} \right\rceil \\ n_{k+1} = n_k + t_k \end{cases}$$

The formula for t_k can be easily obtained using the bound of Equation 7.8. It is then clear that the only degree of freedom is the value of λ, which shall be chosen to minimize the total number of PBS. The initial chain needs λ PBS, and then each decomposition takes $2t_k$ PBS. So we run an exhaustive search on the value of λ to minimize the number of PBS, which is given by:

$$\min_{\lambda \in \{0,\dots,s^n-1\}} \lambda + 2 \sum_{k=0}^{m-1} t_k(\lambda).$$

Table 7.1 shows the optimal shapes for every configurations of s and n.

`Rank` computation and `GaussianElimination`. In concrete implementations, both procedures are actually run at the same time: a Gaussian elimination is performed on $\mathcal{A}^{(k)}$, and a solution $\boldsymbol{\beta}^{(k)}$ is found only if $\mathcal{A}^{(k)}$ is full rank.

`ComputeProducts`. This subroutine produces the matrix $\mathcal{P}^{(k)}$, already introduced above. This is a matrix of shape $s^n \times t_k$, defined by:

$$\mathcal{P}^{(k)} = \left(\mathcal{U}^{(k)} \cdot (\mathcal{B}^{(k)})^T\right) \odot \left(\mathcal{U}^{(k)} \cdot (D^{(k)})^T\right) \tag{7.13}$$

where $\mathcal{B}^{(k)}$ is a matrix containing the coefficients of $\boldsymbol{\beta}^{(k)}$ rearranged in two dimensions as:

$$\mathcal{B}^{(k)} = \begin{pmatrix} \beta^{(k)}_{0,0} & \beta^{(k)}_{0,1} & \cdots & \beta^{(k)}_{0,n_k-1} \\ \beta^{(k)}_{1,0} & \beta^{(k)}_{1,1} & \cdots & \beta^{(k)}_{1,n_k-1} \\ \vdots & \vdots & \ddots & \vdots \\ \beta^{(k)}_{t_i-1,0} & \beta^{(k)}_{t_i-1,1} & \cdots & \beta^{(k)}_{t_i-1,n_k-1} \end{pmatrix}$$

and $\odot$ is the coefficient-wise product.

It can be checked that $(\mathcal{P}^{(k)})_{jj'}$ corresponds to the j'-th term of the decomposition (Equation 7.1), when evaluated with the vector $\mathbf{x}^{(j)}$ as input.

Alternatively, we provide in Figure 7.5 an overview of the first steps of the algorithm, and we detail them below:

1. A chain $\Phi_{n,\lambda}$ is sampled and its matrix representation $\mathcal{U}^{(0)}$ is constructed.
2. A matrix $\mathcal{D}^{(0)}$ is sampled.
3. Using these two matrices, we compute the matrix $\mathcal{A}^{(0)} = \left(\mathcal{U}^{(0)} \cdot (\mathcal{D}^{(0)})^T\right) \square \mathcal{V}^{(0)}$.
4. Using a Gauss elimination, the vector $\boldsymbol{\beta}^{(0)}$ corresponding to the LUT $\mathbf{y}^{(0)}$ is computed. If $\mathcal{A}^{(0)}$ is not full rank, then we go back to Step 1. and sample a new chain and a new matrix.
5. We construct the t_0 variables products (that will be summed in the decomposition formula of Equation 7.1). Those products are re-used in the base of the next chain.
6. Then we process the next iteration: we construct the new matrice $\mathcal{U}^{(1)}$ by appending the products $\mathcal{P}^{(0)}$ and we sample a new matrix $D^{(1)}$. The algorithm then keep on iterating to process all the m outputs.

7.4 Experimental Results

State of the Art. In [Ber+23a], the authors introduce a construction called WoP-PBS (Without-Padding Programmable Bootstrapping), which leverages a technique known as circuit bootstrapping. This process transforms an LWE ciphertext into a GGSW ciphertext via ℓ_{GGSW} PBS[4]. The core idea involves extracting each bit of the message and encoding it into a GGSW ciphertext. These ciphertexts serve as selectors in a `CMUX` tree that selects an appropriate accumulator polynomial based on the extracted bits. A classical PBS is then used to rotate this polynomial using the remaining bits of the message.

As circuit bootstrapping dominates the computational cost, the overall complexity of WoP-PBS scales linearly with the number of bits. This leads to excellent asymptotic performance. However, the requirement of ℓ PBS operations per circuit bootstrapping becomes prohibitive at low precisions.

The authors demonstrate that WoP-PBS surpasses classical PBS for precisions of 8 bits and above. Moreover, they successfully evaluate functions over plaintext spaces up to 24 bits, showcasing impressive scalability. The scheme supports the treatment of several messages at the same time, which amortizes the accumulator selection part. We will compare our work to version with 1, 2 and 4 messages (called *blocks* in the WoP-PBS paper).

[4]Here, ℓ_{GGSW} refers to the level of the gadget decompositions used in GGSW format, see Definition 2.6.2.

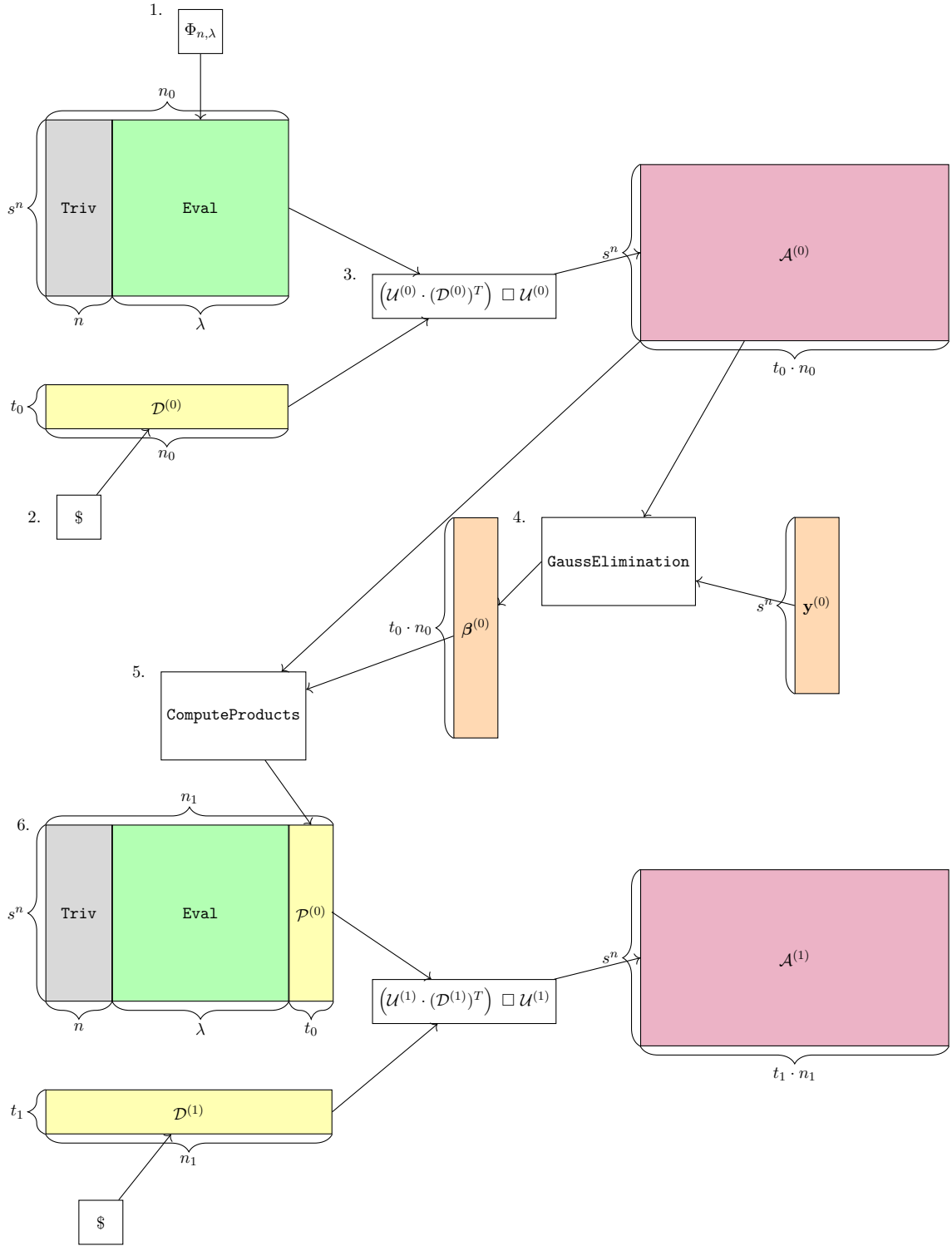


Figure 7.5: Overview of an iteration of the search algorithm for a decomposition. The outputs are the vectors $\mathcal{B}^{(i)}$, that gives the decomposition for the function F represented by the vectors $\mathbf{y}^{(i)}$. We also show the construction of the next matrices $\mathcal{U}^{(1)}$ and $\mathcal{A}^{(1)}$.

Experimental Results and Comparison with WoP-PBS. Direct comparison with [Ber+23a] is difficult, as the authors do not report runtime measurements. Instead, they rely on a cost model to quantify performance. Fortunately, they include the cost of the classical PBS in this model, which serves as a baseline to translate our empirical timings into their cost framework.

Using our approach, we generated decompositions for random LUTs of size 2^n, where $4 \leq n \leq 14$ and measure the time of execution of the LUT int the homomorphic domain. We evaluated multiple chunk sizes ($s = 2$, 4 and 16), all followed by a small prime (resp. 3, 5 and 17) that is used for the value of p. Parameters were selected to ensure an error probability of 2^{-40}, which is slightly stricter than the 2^{-35} used in [Ber+23a], placing our results at a slight disadvantage in the comparison.

Figure 7.6 summarizes our experimental results. It presents the runtime for homomorphic LUT evaluation using our technique, alongside classical PBS performance. To enable a fair comparison with WoP-PBS, we mapped their cost model to estimated runtimes on our hardware using classical PBS as a baseline.

Our method outperforms the state of the art for precisions between 6 and 10 bits and surpasses the single-block version (*i.e.,*the version with one single message in input) of WoP-PBS up to 14 bits. Its implementation is also very simple, as it simply requires an implementation of the standard PBS and not any advanced homomorphic operators. While [Ber+23a] report results up to 24 bits, our method does not realistically scale that far. Although there is no theoretical barrier, generating a decomposition at such precision would require solving a linear system of size 2^{24}, which is computationally impractical with our experimental setup.

7.5 Conclusion

With this work, we have shown that homomorphic evaluations in a large plaintext space can be accelerated by decomposing it into several smaller ones. We have provided algorithms to generate such decompositions for any function, and showed metrics to prove the significant improvement of performances achieved.

An interesting perspective would be to try to get a better theoretical understanding of the probability to find a decomposition with respect to the parameters picked. In this work we relied on a randomized trial-and-error technique, which demonstrated to be good enough in practice, but it would be interesting to get an idea of the time required to find decomposition for arbitrarily large input space.

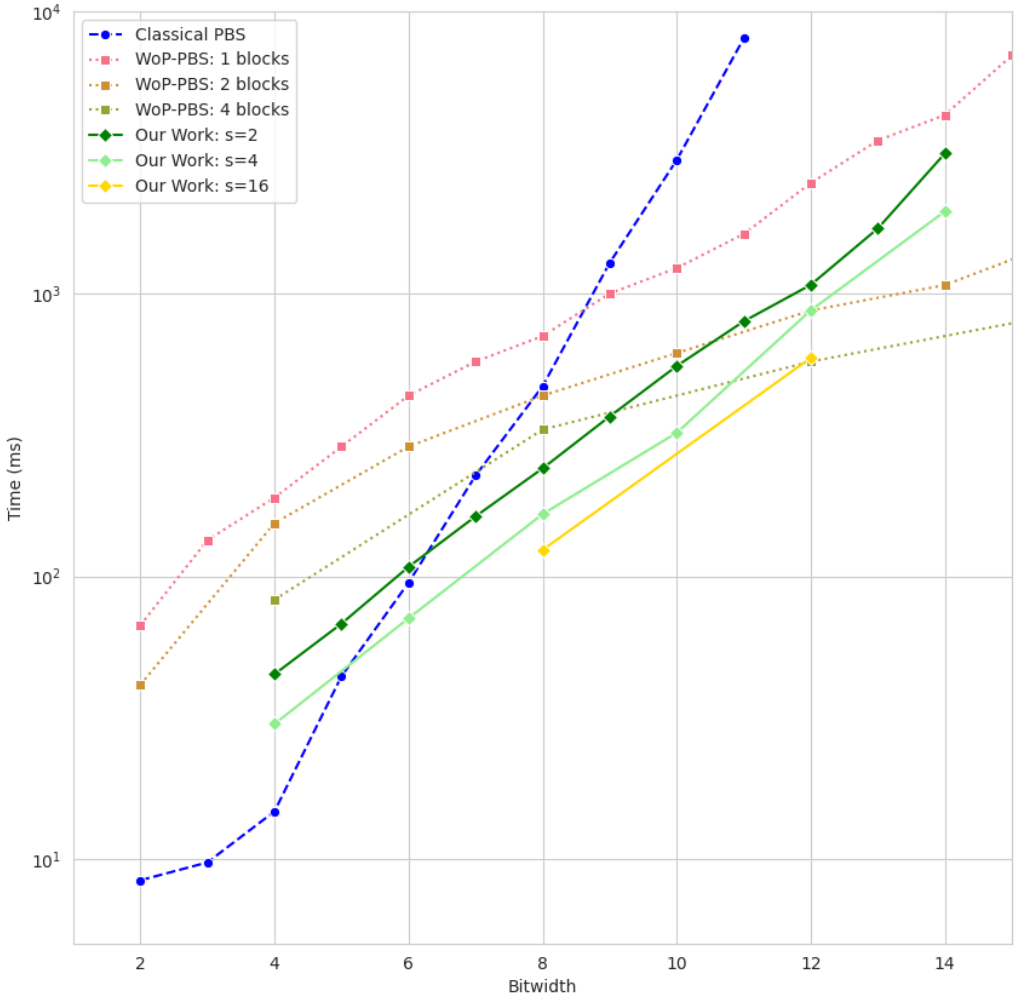


Figure 7.6: Comparison of timings of the classical PBS, the WoP-PBS and our work. The i-block version of WoP-PBS corresponds to slicing the input into i messages. Timings of WoP-PBS have been inferred from the cost model of [Ber+23a]. Experiments run on a server equipped with an AMD Ryzen Threadripper PRO 7995WX with 96 cores, with a maximal frequency of 5.4 GHz and 528 GB of RAM.

Chapter 8

A Practical Solution for Parameter Selection

Along this manuscript, a recurring problem we faced was the prohibitive runtime of the PBS algorithm. In the previous chapter, we presented different method to deal with its poor performances, notably by trying to limit the number of calls to this algorithm in practical settings. But would it be possible to reduce the running time of this operation itself?

Recall that neither the runtime of PBS nor the resulting noise level are affected by the specific LUT being evaluated. On the big picture, the plaintext precision is the factor that matters the most, and is its well-known that PBS becomes impractical for precisions exceeding 8 bits.

But if we dive deeper into the inner workings of this operation, we discover that the running time of the PBS (and of all the other sophisticated homomorphic constructions) mostly depends on the parametrization of the TFHE scheme. So, to improve the performances of homomorphic operations, what we would like is a procedure that would select parameters to optimize the performances of the algorithms.

However, selecting parameters for TFHE poses a significant (and relatively unexplored) engineering challenge. The scheme demands configuring a dozen distinct parameters, resulting in a vast space of potential configurations. This complexity is further compounded by additional considerations: the required precision of plaintexts and the homomorphic circuits to be evaluated need to be taken into account in the process. Moreover, implementing advanced homomorphic operators (such as different PBS implementations) makes the parameter selection even more intricate. Finally, one also needs to consider the specific implementation and the execution environment. Performances can vary significantly across different libraries and machine, necessitating optimization for each specific deployment scenario.

Surprisingly, the literature on parameter selection for TFHE is rather sparse. Some works have specifically addressed the selection of cryptographically secure parameters. Notably, the `lattice-estimator` [APS15] is a tool that simulates state-of-the-art attacks and estimates the concrete security of various LWE parameter sets. It is widely used by practitioners to assess the security of their implementations. Additionally, the recent work in [Bia+24] offers a comprehensive survey on the security of FHE schemes, covering lattice theory concepts and associated attacks. While selecting cryptographically secure parameters is certainly essential, another key consideration is ensuring the correctness of homomorphic computations. Due to the noise inherent in ciphertexts and the effects of homomorphic operations, errors can arise during computations. Therefore, it is crucial to maintain the underlying error probability below a specified target threshold. The core challenge in parameter selection lies in ensuring cryptographic security, computational correctness (with a concrete upper bound on error probability), and the optimization of performance in homomorphic operations.

The main step forward regarding this specific problem was made by Bergerat *et al.* in [Ber+23a]. Their work formalizes the parameter selection process for TFHE as an optimization problem. The authors model homomorphic computations as graphs of *atomic patterns* (APs), which are sub-graphs of FHE operators. They propose a generic methodology to optimize the entire graph by solving a minimization problem under constraints, where each AP is assigned

an efficient set of parameters. However, their approach to assigning parameters to APs does not consider the specificities of different FHE libraries or the machine on which they run. While this methodology is implemented in the `concrete-optimizer` [Zam22b], it is neither documented nor designed to be a user-friendly standalone tool. Additionally, it lacks machine-specific profiling and does not support extensibility to other libraries.

In contrast, mainstream libraries such as `tfhe-rs` [Zam22c] and `OpenFHE` [Bad+22] use hardcoded sets of parameters. However, to the best of our knowledge, these libraries provide no documentation on how these parameters were selected or any formal proof that they consistently achieve the intended levels of security and correctness.

Our contributions. In this chapter, we aim at filling the aforementioned gap by presenting an efficient method to identify efficient sets of parameters for individual atomic patterns. This is achieved through our new library and machine-dependent cost model, and efficient optimization algorithm. Our approach is implemented in a tool called ORPHEUS (for "Optimized Research of Parameters for Homomorphic Encryption made Universal and Simple"), which will be released as open-source software.

Specifically, our contribution is threefold:

- *Efficiency of parameter selection.* We introduce a concrete methodology to reduce the search space of TFHE parameters and select the optimal set for a given homomorphic circuit.
- *Library and Machine-tailored parameter selection.* We replace the naive cost model from [Ber+23a] (based on operation counts) by developing a profiling method able to adapt to different libraries and execution environments. After profiling phase, our model predicts the performance of a given circuit instantiated with any set of parameters without the need for costly empirical measurements.
- *Open-source tool:* Our techniques have been implemented and tested in ORPHEUS. This tool has been designed with modularity and user-friendliness in mind, making it simple for cryptographers to extend it to new homomorphic operators, machines or libraries.

Additionally:

- Taking as a core example the classic TFHE atomic pattern made of linear operations followed by a layer or PBS (referred to as CJP in [Ber+23a]), we go through the whole methodology step-by-step to provide a concrete example of how the optimization algorithm works.
- We also present benchmarks that validate and even improve the (non-documented) parameter selection of one of the mainstream libraries, namely `tfhe-rs` [Zam22c].

Alongside presenting ORPHEUS, this chapter attempts to provide a survey to help FHE practitioners to better understand noise behavior and the role of parameters in noise management.

8.1 TFHE Parameter Selection Problem

In this paper, we build upon the framework developed in [Ber+23a], which formalizes the parameter selection for TFHE as an optimization problem. The authors model a homomorphic computation as a graph consisting of two types of sub-graphs: atomic patterns and noise-accumulating patterns. Atomic patterns are subgraphs of FHE operators that produce one or

more ciphertexts with noise independent of the input noise, *i.e.,* by incorporating PBS in practice. In contrast, noise-accumulating patterns involve FHE operators that add noise to the input noise, such as `KeySwitch`or dot product operations.

[Ber+23a] proposes a high-level framework to formalize the problem of selecting optimal parameters for a TFHE circuit as an abstract optimization problem, based on a target security level and a target error probability. Furthermore, they introduce a method, implemented in a tool referred to as `concrete-optimizer` [Zam22b], to select parameters for an individual AP, and a strategy to optimize parameter selection for a graph of APs within a global circuit. However, the `concrete-optimizer`'s method for an individual AP is neither well-documented nor designed as a user-friendly, standalone tool. Additionally, while using a generic cost model, it does not support machine-dependent profiling or extension to other TFHE libraries. This chapter and the associated tool, ORPHEUS, aim to fill this gap.

Our methodology covers all possible FHE graphs of atomic patterns, including the trivial case of a leveled circuit without PBS. Specifically, we address three types of FHE graphs:

1. The entire circuit to evaluate boils down to a single noise-accumulating pattern or a single atomic pattern: in this scenario, we start with fresh ciphertexts (with noise variance corresponding to a fresh encryption), and the output ciphertexts must have noise small enough to allow for decryption without exceeding the error probability p_{err}.

2. The circuit is composed of several instances of the same atomic pattern, such as the classic TFHE atomic pattern made of linear operations followed by a layer of PBS (referred to as CJP in [Ber+23a]).[1] In this situation the output of the considered atomic pattern is the input of a similar atomic pattern, enforcing the constraint that the output noise must not exceed the input noise.

3. The circuit combines different atomic patterns: in this case, the parameters of the atomic patterns must be optimized together using a higher-level compilation method (such as the one in [Ber+23a]). This introduces additional constraints, such as the inequalities between the noise variances at the input/output of consecutive atomic patterns and the equality of ciphertext dimensions.

Our methodology applies to each of these three scenarios, taking as input either a noise-accumulating pattern or an atomic pattern, along with a set of constraints, including a security level λ and a error probability p_{err}. It is illustrated with a running example in Section 8.2 using a classical atomic pattern. For simplicity, we omit the description of the methodology for noise-accumulating patterns, as it is similar to, and more straightforward than, the one applied to atomic patterns.

Before delving into the details, we first formalize the parameter selection problem in the rest of this section. We begin by presenting the parameters of the TFHE scheme, followed by a discussion of the three fundamental challenges in FHE: security, correctness, and performance.

8.1.1 TFHE Parameters

Chapter 2 was dedicated to an in depth-presentation of the TFHE scheme. Particularly, Sections 2.5 and 2.7.2 present two main building blocks, respectively the `KeySwitch` and the Programmable Bootstrapping. In this section, we focus on the parameters that dimension the scheme.

We present all these parameters in Definition 8.1.1 and provide their explicit definition sets, along with reasonable and typical ranges that will serve as the foundation for our subsequent

[1] Different instances of an atomic pattern all have the same structure but the coefficients in their linear operations as well as the look-up tables in their PBS might differ from one instance to another.

selection process. In practice, these ranges can be adjusted without affecting the methodology itself, although they might influence the performance of the convergence process.

Definition 8.1.1. A *tuple of TFHE parameters* is defined as:

$$\mathcal{P} = (q, n_{\text{short}}, k, N, \sigma_{\text{short}}, \sigma_{\text{long}}, \mathfrak{B}_{\mathsf{PBS}}, \ell_{\mathsf{PBS}}, \mathfrak{B}_{\mathsf{KS}}, \ell_{\mathsf{KS}})$$

belonging to the space:

$$\mathcal{Q} \times \mathcal{N}_{\text{short}} \times \mathcal{K} \times \mathcal{N}_{\text{poly}} \times \mathcal{S}_{\text{short}} \times \mathcal{S}_{\text{long}} \times \mathcal{B}_{\text{PBS}} \times \mathcal{L}_{\text{PBS}} \times \mathcal{B}_{\text{KS}} \times \mathcal{L}_{\text{KS}},$$

with

- q: the modulus used for the ciphertext space, typically chosen as a standard integer precision, such as 2^{32} or 2^{64}. In our experiment, we restrict it to $\mathcal{Q} = \{2^{64}\}$ to match the main TFHE libraries (namely `tfhe-rs` [Zam22c] and `OpenFHE` [Bad+22]). However, it can be changed to 2^{32} or any other value without affecting the methodology.

- n_{short}: the dimension of the LWE ciphertexts used within the PBS algorithm[2]. As it will be explained in Section 8.1.2, this value is directly linked to the security level and the noise variance of these ciphertexts. A small dimension would require a higher noise variance to achieve the same security level, which is ultimately constrained by the modulus q and the expected correctness level. On the other hand, a large dimension would slow down the computations. A typical range for this parameter which we use in our experiments is $\mathcal{N}_{\text{short}} = [\![200, 1500]\!]$.

- N: the degree of the polynomials in GLWE ciphertexts. We restrict N to be a power of 2, as this choice is computationally efficient for Fast Fourier Transform (FFT)-based polynomial multiplication and offers favorable algebraic properties. This approach was originally adopted in the TFHE paper [Chi+20], and has been maintained in subsequent works. A typical range for this parameter which we use in our experiments is $\mathcal{N}_{\text{poly}} = \{2^8, \ldots, 2^{14}\}$. Indeed, larger powers are inefficient, while smaller ones make correctness hard to achieve (taking a small N makes the accumulator polynomial of the `BlindRotate` shorter, so it leaves less room for error).

- k: the dimension of GLWE ciphertexts. The product $k \cdot N$ must meet the same lower bound as n_{short} to ensure security. We set an upper bound for this product at 2^{14}, based on practical experiments indicating that the PBS becomes too slow when this value is exceeded. In our experiments, we fix $\mathcal{K} = \{1, \ldots, 8\}$ and filter out the pairs (k, N) whose product exceeds 2^{14}. Thus, the product space for k and N is defined as $\{(k, N) \in \mathcal{K} \times \mathcal{N}_{\text{poly}} \mid k \cdot N \leq 2^{14}\}$.

- $\sigma_{\text{short}}, \sigma_{\text{long}}$: the standard deviation of the Gaussian distribution of fresh noise in the ciphertexts of dimension respectively n_{short} and n_{long} (or (k, N)). In the following, we will make σ_{short} and σ_{long} entirely dependent from the other parameters, so these sets will be reduced to a single element each.

- $\mathfrak{B}_{\mathsf{PBS}}$: the base used in the gadget decompositions occurring in the `BlindRotate` phase. We restrict the decomposition basis to powers of two so it divides q, avoiding rounding errors. A wide range for this parameter which we consider in our experiments is $\mathcal{B}_{\text{PBS}} = \{2^8, \ldots, 2^{25}\}$.

[2]Note that some LWE ciphertexts will have dimensions $n_{\text{long}} = k \cdot N$, namely the ones produced by `SampleExtract`on a PBS accumulator.

- $\mathfrak{B}_{\mathsf{KS}}$: similar to the previous one but for the `KeySwitch` phase. As we observed that `KeySwitch` requires smaller decompositions basis, we consider $\mathcal{B}_{\mathrm{KS}} = \{2^3, \ldots, 2^{15}\}$ in our experiments.

- ℓ_{PBS} and ℓ_{KS}: the levels of those decompositions. We can restrict to cases where $\mathfrak{B}_{\mathsf{PBS}}^{\ell_{\mathsf{PBS}}} \leq q$, because otherwise the precision of the gadget decomposition would uselessly exceed the precision of a ciphertext. So, we take $\ell_{\mathsf{PBS}} = \{1, \ldots, 4\}$ and filter out the pairs $(\mathfrak{B}_{\mathsf{PBS}}, \ell_{\mathsf{PBS}})$ that do not fit. Following the same reasoning, we take $\mathcal{L}_{\mathrm{KS}} = \{1, \ldots, 9\}$ and apply the filtering.

While the above ranges are representative examples which we use in our experiments, ORPHEUS allows the user to define these ranges as input of the search, enabling them to enlarge or reduce the search space according to their needs. Different examples of such ranges can be found in [Tap23].

Next, we explain the three metrics used to assess the relevance of a given parameter tuple: security, correctness, and performances. The two former come as constraints of the optimization problem while the latter comes as the cost function to be minimized.

8.1.2 The Security Constraint

The security of TFHE is built on top of the Learning with Errors problem, introduced in [Reg05] and recalled in Definition 2.1.1.

During TFHE's encryption phase, two types of fresh ciphertexts can be generated: *short* ciphertexts, which are LWE ciphertexts of dimension n_{short}, and *long* ciphertexts, which can either be GLWE ciphertexts of dimension k and degree N, or LWE ciphertexts of dimension $k \cdot N$. According to the current state of cryptanalysis, the hardness of $\mathsf{GLWE}_{q,k,N,\sigma_{\mathrm{long}}}$ is considered equivalent to that of $\mathsf{LWE}_{q,k\cdot N,\sigma_{\mathrm{long}}}$. Therefore, the parameters $(q, n_{\mathrm{short}}, \sigma_{\mathrm{short}})$ and $(q, k\cdot N, \sigma_{\mathrm{long}})$ for the instances $\mathsf{LWE}_{q,n_{\mathrm{short}},\sigma_{\mathrm{short}}}$ and $\mathsf{LWE}_{q,k\cdot N,\sigma_{\mathrm{long}}}$ must be chosen to guarantee a security level of at least λ bits.

These fresh ciphertexts, whether short or long, are provided to the server as input. Additionally, the evaluation keys used by the server for keyswitches and PBS are themselves collections of fresh ciphertexts that encrypt the bits of the secret key. Ensuring the LWE security of these ciphertexts is sufficient to secure the entire homomorphic encryption pipeline, as all subsequent ciphertexts are deterministically derived from the input ciphertexts and the evaluation keys. They will share the same parameters as the fresh ones apart from the noise which will be larger.

A widely recognized tool in the scientific community for assessing the security of lattice-based cryptographic schemes is the `lattice-estimator` [APS15]. This tool acts as an oracle, providing security level estimates for (G)LWE instances based on their parameters. In Section 8.2.1, we detail how this tool is integrated within our framework.

8.1.3 The Correctness Constraint

As homomorphic operations are carried out, the noise grows. If the noise exceeds a certain threshold, the client will obtain an incorrect result upon decryption. While injecting random noise into ciphertexts is essential to ensure security, it makes theoretically impossible to guarantee that some homomorphic computation will yield a correct result with absolute certainty. However, by selecting bigger parameters, the error probability can be reduced to an arbitrary small value, such as 2^{-40}, 2^{-64} or 2^{-128}.

Correctness is also closely tied to the security of the scheme. Recent works [Che+24a; Che+24b] have demonstrated that decryption errors can reveal information about the secret key. Specifically, an adversary could exploit a high error probability to build a key-recovery

attack by exploiting theses errors. Therefore, it is crucial to select parameter tuples that ensure sufficiently negligible error probabilities, typically 2^{-128}.

To address these correctness issues, it is important to analyze and compute the error probability of a given circuit, ensuring that the chosen parameters provide sufficient reliability throughout the homomorphic computation.

In purely linear circuits (*i.e.,* in noise-accumulating patterns), noise accumulates additively with each operation, causing error probability to increase monotonically. This means we only need to ensure the final noise level (at the circuit's end, right before decryption) remains below some safety threshold. However, introducing a PBS operation in the circuit makes things a bit more complex.

If the noise right before the `BlindRotate` step of the PBS is too high, the output of the PBS will still be a low-noise ciphertext but it will encrypt an incorrect value (because the rotation of the accumulator will land on the wrong torus sector). So, we must approach noise management a bit differently: rather than focusing solely on the final noise level, we must ensure the noise remains sufficiently low at each point preceding a `BlindRotate` operation (which are actually the points where the noise is maximal in the circuit). By maintaining noise below our safety threshold at these critical junctures, we can guarantee with overwhelming probability that the PBS operations will produce correct results.

To be able to analyze the noise in such critical points, we make the assumption that the distribution of the noise remains Gaussian after homomorphic operations. This assumption is widely accepted in the literature and is for example supported by the measurements presented in the work of [Ber+25].

Concretely, let us instantiate a bootstrapping circuit with a given tuple of TFHE parameters. Let us denote the variance of the noise right before `BlindRotate` by $\sigma^2_{\text{critical}}$. Using this value, it is possible to compute the error probability of this PBS with the following formula (that appears in [Chi+21]):

$$p_{\text{err}} = \mathsf{erfc}\left(\frac{\tau}{2\sqrt{2}\cdot\sigma_{\text{critical}}}\right) \tag{8.1}$$

where τ denotes the size of a torus sector, and erfc the complementary error function of Gauss, defined as

$$\mathsf{erfc}\,(z) = 1 - \frac{2}{\sqrt{\pi}}\int_0^z e^{-t^2}dt$$

Observe that the bootstrapping fails if the noise exceeds half a sector of the torus, which corresponds to $\frac{q}{p}$. So, we need to integrate the Gaussian outside of the bounds of the sectors. This gives the result.

All in one, the information we need about a circuit includes both the formula for the variance at the output, which is to be compared with either the maximum noise variance for decryption or the input constraint, and the formula for the noise variance just before the `BlindRotate` step. If the circuit contains multiple PBS, we focus on the one with the highest variance, as it is the most likely to lead to errors. In our methodology, we consider the first inequality on the noise variance at the output as a constraint and we rely on the critical noise variance to estimate the error probability. In particular, we can rewrite Equation 8.1 to compute the upper bound on the variance (referred to as the "safety threshold" in this section), ensuring a correct PBS with a probability of $1 - p_{\text{err}}$, using torus sectors of size τ:

$$\sigma_{\text{bound}}(p_{\text{err}}, \tau) = \frac{\tau}{2\sqrt{2}\cdot\mathsf{erfc}^{-1}(p_{\text{err}})}. \tag{8.2}$$

The size of the torus sectors τ is determined by the plaintext modulus p. A larger modulus means a thinner torus discretization, thus smaller torus sectors. In the most simple case, the torus is simply shared in p sectors, which fix $\tau = \frac{q}{p}$. Another recurrent scenario is the use of a

bit of padding (see Section 3.2) to deal with the negacyclicity problem, in which case τ becomes $\frac{q}{2p}$.

In Section 8.2.1, we explain how we use this bound in our tool to generate parameters that ensure correctness up to an error probability input by the user.

8.1.4 The Optimization Problem

To find good parameters, a naive approach would consist in relying solely on empirical measurements. Specifically, one could evaluate the atomic pattern AP for every possible tuple of parameters, record the runtime and noise levels, and construct an exhaustive database of results. To configure a TFHE instance of AP for a practical use-case (with predefined correctness probability and security level), one would then compute the safety variance threshold as outlined in Equation 8.2, filter the database to retain only the parameter tuples that meet both the correctness and security requirements, and finally select the fastest configuration.

More formally, let us denote by $\mathcal{P}^{(\mathsf{AP})}_{(p_{\mathsf{err}},\sigma_{\mathrm{in}},\sigma_{\mathrm{out}})}$ the set of all parameter tuples that evaluate correctly the atomic pattern APwith probability at least $(1 - p_{\mathsf{err}})$, when fed with inputs of maximal noise σ_{in} and producing results of maximal noise σ_{out}. The goal of this optimization algorithm is to compute:

$$\underset{p_i \in \mathcal{P}^{(\mathsf{AP})}_{(p_{\mathsf{err}},\sigma_{\mathrm{in}},\sigma_{\mathrm{out}})}}{\operatorname{argmin}} \quad \mathtt{Runtime}^{\mathsf{AP}}(p_i). \tag{8.3}$$

While conceptually simple, this approach does not scale. Running a PBS is notoriously slow, and the parameter space is too large. Moreover, each experiment would need to be repeated multiple times to gather enough data points for the average running time to be statistically meaningful. Therefore, we need a model capable of accurately predicting the runtime of a circuit for a given parameter set without requiring it to be executed. Since performance depends significantly on the machine and library used, it is unlikely that a generic model could capture the behavior of the circuit across all possible targets. Instead, we adopt a target-specific approach, developing a separate model for each architecture. To ensure practicality, the cost of training such a model must remain minimal. In the next section, we describe our method to speed-up the parameter search while basing it on an accurate prediction of AP's running time for a specific library on a specific machine.

8.2 Our Solution

In the previous section, we outlined the various challenges involved in selecting the appropriate parameters for a TFHE instance. Now, we present our framework to address these challenges.

To illustrate our methodology, we apply it to a simple atomic pattern, first introduced in [CJP21] and extensively studied since then, which consists of a dot product (DP) between a vector of ciphertexts and a vector of clear constants of maximal norm ν, followed by a programmable bootstrapping, as shown in Figure 8.1. We shall consider the second type of FHE graphs defined at the beginning of Section 8.1, with the constraint that the noise variance in output of the atomic pattern must not exceed the noise variance in input.

We denote Library the considered software library implementing the atomic pattern and Machine the machine on which the computation is run.

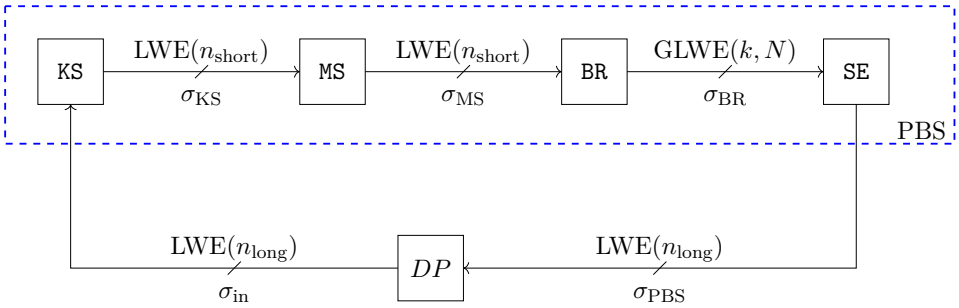


Figure 8.1: CJP Atomic pattern. On each wire is displayed the type and size of ciphertext, as well as the standard deviation of the noise on this point.

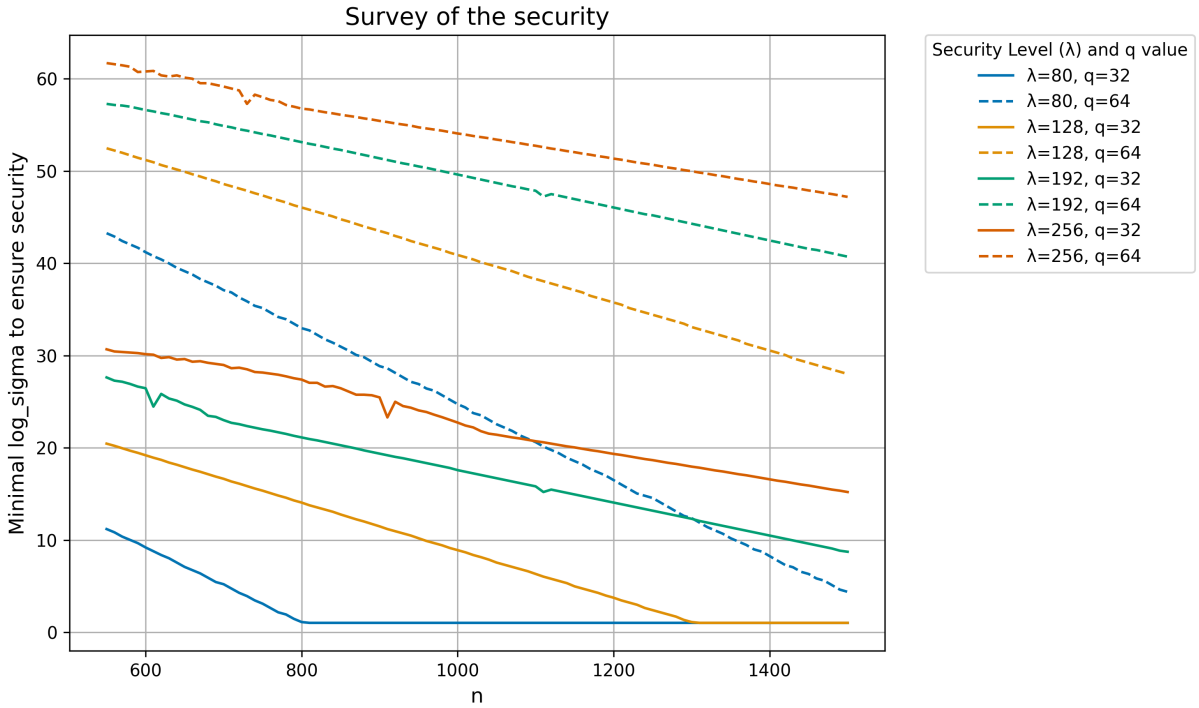


Figure 8.2: Isosecurity curves illustrating the minimum noise level (log scale) required for a given dimension n to achieve a security level λ with modulus q.

8.2.1 Reducing the Parameter Space

Security-Based Space Reduction

As discussed in Section 8.1.2, the security constraints establish a relationship between the parameters $(q, n_{\text{short}}, \sigma_{\text{short}})$ for LWE instances and $(q, k, N, \sigma_{\text{long}})$ for GLWE instances. A well-established result in the field, detailed in [Tap23], shows that if q is fixed and σ is computed as the smallest value required to achieve a given security level λ for a given n, the relationship between n and $\log_2(\sigma)$ can be closely approximated by an affine function. This observation allows us to fit a simple affine model to $(n, \log_2(\sigma))$ pairs, significantly simplifying the estimation of σ for any n. Such a fitting provides a computationally efficient way to enforce security constraints while navigating the parameter space.

Figure 8.2 illustrates this relationship for different values of security level λ and modulus q. On each *isosecurity* curve, any point $(n, \log_2(\sigma))$ that lies above it corresponds to a secure LWE instance. However, there is a crucial exception: zero noise is trivially insecure, so we impose a lower bound on $\log_2(\sigma)$. Following the reasoning outlined in [GHS12] and [Tap23], we fix this

Table 8.1: Pairs (a, b) for different security levels λ and modulus q as computed with the lattice estimator.

q	λ	a	b
2^{32}	80	-0.04086	33.71697
	128	-0.02576	34.66181
	192	-0.01880	36.60474
	256	-0.01754	40.78150
2^{64}	80	-0.04125	65.99926
	128	-0.02582	66.70935
	192	-0.01767	67.25340
	256	-0.01489	69.13875

lower bound at two bits.

Let a and b denote the parameters of the affine curve, such that for a fixed value of λ and a fixed value of q:

$$\log(\sigma) = \max(a \cdot n + b, 2). \tag{8.4}$$

Table 8.1 provides pairs (a, b) corresponding to several pairs of security levels λ and modulus q and computed from the experiments performed with the `lattice-estimator` [APS15].[3]

From this result, σ_{short} (resp. σ_{long}) can always be determined as a direct function of n_{short} (resp $n_{\text{long}} = k \cdot N$) using Equation 8.4. This approach removes both degrees of freedom $\mathcal{S}_{\text{short}}$ and $\mathcal{S}_{\text{long}}$. Consequently, the size of the search space is significantly reduced, and the parameter tuples are inherently restricted to secure configurations.

Correctness-Based Space Reduction

We have seen that the security constraints translate into relationships between certain parameters, thereby reducing the size of the search space. Interestingly, the same happens with the correctness constraint.

Recall that Equation 8.2 shows that for a given error probability p_{err} and a given size of torus sector τ, a safety threshold σ_{bound} can be defined for the noise variance that should not be exceeded. On the other hand, for a given atomic pattern, it is possible to derive a *noise formula* that expresses the highest noise variance σ_{critical} (for a PBS) as a function of the TFHE parameters and the features of this AP. Therefore, to ensure the correct error probability, the following inequality must hold:

$$\sigma_{\text{critical}} \leq \sigma_{\text{bound}}. \tag{8.5}$$

This inequality can be rewritten to derive bounds on one of the parameters with respect to the others. In practice, we choose to do that with n_{short}, because it is the parameter with the largest definition space (see the example of Section 8.2.1). By doing so, we eliminate the largest dimension of the search space, significantly reducing the cost of an exhaustive search.

The above principle can be extended to consider external constraints on the noise variances in input and output of the atomic pattern. Those additional constraints simply translate to further bounds on n_{short} (which might reduce the search space even more).

Wrapping-up the two above space reductions: for each partial tuple, we apply the security constraint to derive the noise parameters σ_{short} and σ_{long} (from Equation 8.4), then the constraint of correctness (from Equation 8.5) to get the range of possible values for n_{short}. The remaining parameters form so-called *partial parameters tuples*, which live in the space $\mathcal{Q} \times \mathcal{K} \times \mathcal{N}_{\text{poly}} \times \mathcal{B}_{\text{PBS}} \times \mathcal{L}_{\text{PBS}} \times \mathcal{B}_{\text{KS}} \times \mathcal{L}_{\text{KS}}$.

[3] We used commit `374f07331e6575d1856b2212f3b8aeac96e0295e`.

Based on the typical ranges defined in Section 8.1.1, the reduced space contains approximately 2^{17} tuples. This reduction makes exhaustive search feasible, which was previously impractical.

Performance-Based Space Reduction

The runtime of an AP naturally increases when n_{short} becomes larger. So we complete each partial tuple with the smallest value possible for n_{short} which complies with the correctness constraint (from Equation 8.5). At this point, all the partial tuples have been completed to form the most efficient corresponding full parameter tuple. For each one, we estimate their running time using a cost model fitted for the couple (Library, Machine) and select the fastest one. In Section 8.2.2, we explain how to construct such tailored cost models.

Example on CJP

To illustrate the above method, we apply it to the classic TFHE atomic pattern made of linear operations followed by a layer of PBS (referred to as CJP in [Ber+23a]). We consider the use case of a circuit solely composed of such atomic patterns, with the constraint that the output noise must not exceed the input noise. We explain in more details in Appendix A.1 how to derive σ_{critical} for this use-case. The concrete formula, which has been derived and proved in [Tap23], is:

$$\begin{aligned}\sigma^2_{\text{critical}} = \frac{4N^2}{q^2}\Bigg[\nu^2 \cdot \Bigg(& n_{\text{short}} \cdot (\ell_{\mathsf{PBS}}(k+1)N\frac{\mathfrak{B}^2_{\mathsf{PBS}}}{12}\sigma^2_{\text{long}}) \\ &+ n_{\text{short}} \cdot \frac{q^2 - \mathfrak{B}^{2\ell_{\mathsf{PBS}}}_{\mathsf{PBS}}}{24\mathfrak{B}^{2\ell_{\mathsf{PBS}}}_{\mathsf{PBS}}} \cdot \frac{kN}{2}\Bigg) + \frac{n_{\text{long}}}{2} \cdot \frac{q^2}{12\mathfrak{B}^{2\ell_{\mathsf{KS}}}_{\mathsf{KS}}} \\ &+ n_{\text{long}} \cdot \ell_{\mathsf{KS}} \cdot \sigma^2_{\text{short}} \cdot \frac{\mathfrak{B}^2_{\mathsf{KS}}}{12}\Bigg] + \frac{n_{\text{short}}}{24},\end{aligned}$$

where ν is the maximal norm of the vector containing the clear constants of the linear combination DP in the CJP atomic pattern (where the max is over the different instances of the CJP pattern involved in the FHE graph).

As explained in Section 8.2.1, we want to write this as a function of n_{short}. We use the security constraint to substitute σ_{short} by an expression function of n_{short}. The formula can be rewritten as:

$$\sigma^2_{\text{critical}} = \alpha \cdot 2^{2\cdot(a\cdot n_{\text{short}}+b)} + \beta \cdot n_{\text{short}} + \gamma \tag{8.6}$$

with:

$$\begin{aligned}\alpha &= \frac{4N^2}{q^2} \cdot n_{\text{long}} \cdot \ell_{\mathsf{KS}} \cdot \frac{\mathfrak{B}^2_{\mathsf{KS}}}{12} \\ \beta &= \nu^2 \cdot \frac{4N^2}{q^2} \cdot \left(\ell_{\mathsf{PBS}}(k+1)N\frac{\mathfrak{B}^2_{\mathsf{PBS}}}{12}\sigma^2_{\text{long}} + \frac{q^2 kN}{48\mathfrak{B}^{2\ell_{\mathsf{PBS}}}_{\mathsf{PBS}}}\right) + \frac{1}{24} \\ \gamma &= \frac{4N^2 n_{\text{long}}}{24\mathfrak{B}^{\ell_{\mathsf{KS}}}_{\mathsf{KS}}}.\end{aligned}$$

Studying this function with a simple differentiation shows that it admits only one minimum $\sigma_{\min}$ attained for $n_{\text{short}} = n_{\min}$, with:

$$n_{\min} = \frac{1}{a}\left(\frac{1}{2}\log_2\left(-\frac{\beta}{2\alpha\ln(2)a}\right) - b\right).$$

From there, we can compute $\sigma_{\min}$ and σ_{bound} for each partial parameter tuple. This leads to two possible cases:

- If $\sigma_{\min}$ is greater than σ_{bound}, then the considered partial parameter tuple is not suitable for the AP regardless of the value of n_{short}, and can therefore be discarded.
- Otherwise, we need to determine the smallest n_{short} such that the noise variance remains below σ_{bound}. Since the function defining σ_{critical} (introduced in Equation 8.6) is monotonically decreasing with respect to n_{short}, this optimal value can be efficiently found using a simple dichotomic search algorithm.

Once all the partial tuples have been either discarded or completed with the optimal n_{short}, we apply the cost model to all valid tuples and select the most efficient one.

8.2.2 Modeling the Execution Time

While the above approach allows to significantly reduce the research space, the latter remains too vast to run the target atomic pattern for each parameter set to build a database of the execution time. Indeed, a typical atomic pattern includes a (computationally heavy) PBS and it should be run several times for each configuration to get an accurate estimation of the running time. To tackle this issue, we need a cost model which, for a given parameter set, outputs an accurate estimation of the running time for the target atomic pattern.

The approach proposed in [Ber+23a] consists in counting the number of elementary operations required for a given parameter set, and use this count as a direct prediction of the running time. However, the main libraries that implement TFHE all use some advanced optimizations techniques, such as parallelization or exploitation of hardware optimizations. This makes the execution time vary significantly with respect to Library and Machine.

That is why we propose a measurements-driven approach, more realistic, that we break down into three steps:

1. *Measurements:* We construct a representative subset of the parameter space and measure the execution time of the atomic pattern for every parameter tuple of the subset on (Library, Machine).
2. *Training:* Using the database constructed in the previous step, we train a model to predict the execution time on (Library, Machine). We dive deeper into the inner workings of this model in the rest of this section.
3. *Inference :* Inferring an execution time of a parameter set is extremely fast using our model. Thus, we simply run the inference for every single possible parameter tuple to build a database, which our optimization algorithm will use as a reference.

The intuition behind our model is that a PBS can be broken down into smaller subroutines such as gadget decompositions, Fast-Fourier Transforms (FFT) or polynomial multiplications. So, in principle, we could measure the running time of these subroutines on (Library, Machine), and then deriving the number of calls to these subroutines for each parameter tuples. By summing it all together, estimating the running time should be easy and would only require a few measurements (namely, the running time of each subroutine). However, their computational complexities depend *from the parameter tuple itself.* For example, the running time of the FFT is in $0(N \log N)$ and the gadget decompositions are linear in ℓ_{KS} or in ℓ_{PBS}.

Our approach to modeling the runtime of an AP is as follows: for each parameter tuple, we consider each subroutine and count the calls to them. We also derive the computational complexity. Then, we try to construct a linear combination of the form given in Definition 8.2.1.

Definition 8.2.1. Let AP an atomic pattern. For a given parameter tuple $\mathcal{P}$ and a set $\{\mathcal{A}_1, \ldots, \mathcal{A}_a\}$ of subroutines, the cost of evaluating AP can be written as:

$$\sum_{i=1}^{a} \texttt{Count}_{\mathbf{AP}}(\mathcal{A}_i) \times \texttt{Complexity}_{\mathcal{P}}(\mathcal{A}_i) \times \texttt{LinearCoeff}(\mathcal{A}_i)$$

where:

- $\texttt{Count}_{\mathbf{AP}}(\mathcal{A}_i)$ represents the number of calls to the subroutine $\mathcal{A}_i$ within AP.
- $\texttt{Complexity}_{\mathcal{P}}(\mathcal{A}_i)$ is an expression of the computational complexity of the subroutine $\mathcal{A}_i$ with respect to the TFHE parameters tuple. This complexity refers to the expression in the $O(\cdot)$ expressing the runtime of $\mathcal{A}_i$ (e.g. $N \log N$ for the FFT). Specifically, it quantifies the number of elementary operations (e.g., additions, multiplications, or memory accesses) that the subroutine performs, depending on the parameters. Determining this expression is done by analyzing the algorithm of the subroutine.
- $\texttt{LinearCoeff}(\mathcal{A}_i)$ represents the hidden constant in the O notation, that captures the behavior of the target machine and library. We will determine it using a linear regression.

Counting the number of calls to a given subroutine is quite easy, as well as expressing the complexities. The linear coefficients are then estimated using a linear regression model.

We train the linear regression model from the measurements performed on the parameter subset, which allows us to determine the linear coefficients and thus a concrete formula to predict the execution time for *any* parameter tuple on (Library, Machine).

In Table A.1 (in Appendix A.2), we provide the formulas and coefficients obtained through linear regression for the CJP atomic pattern, evaluated with the library `tfhe-rs` and on the server[4] that will be used in Section 8.4 for our experiments.

Figures 8.3 and 8.4 presents experimental results for our training method applied to the `tfhe-rs` library on the same server. Figure 8.3 shows that our model quickly achieves a small mean squared error as the size of the training set increases. A subset as small as 50 already provides good results. Figure 8.4 further demonstrates the accuracy of our model for the whole range of inputs: we observe that the prediction errors are small relatively to the actual timings (about 4.3% in average).

Thanks to this model, we are now equipped with a prediction oracle for the runtime of an atomic pattern, instantiated with a given parameter tuple with a specific Library on a specific Machine. Next, we use this oracle to select the best parameter tuple.

8.2.3 The Optimization Process

In this section, we present our pipeline of operations to generate parameters. The optimization process can be separated in two phases: an offline phase and online phase.

The offline phase is the heaviest: we generate a random subset of the parameter space and measure the running time of the atomic pattern APusing Library on Machine. We then train a regression model to get a predictive model for the cost of APwith the couple (Library, Machine).

During the online phase, to generate a set of parameters for APon (Library, Machine), the user specifies the error probability p_{err} and the security level λ they target. The tool then constructs

[4]The server is equipped with an AMD Ryzen Threadripper PRO 7995WX with 96 cores, with a maximal frequency of 5.4 GHz and 528 GB of RAM.

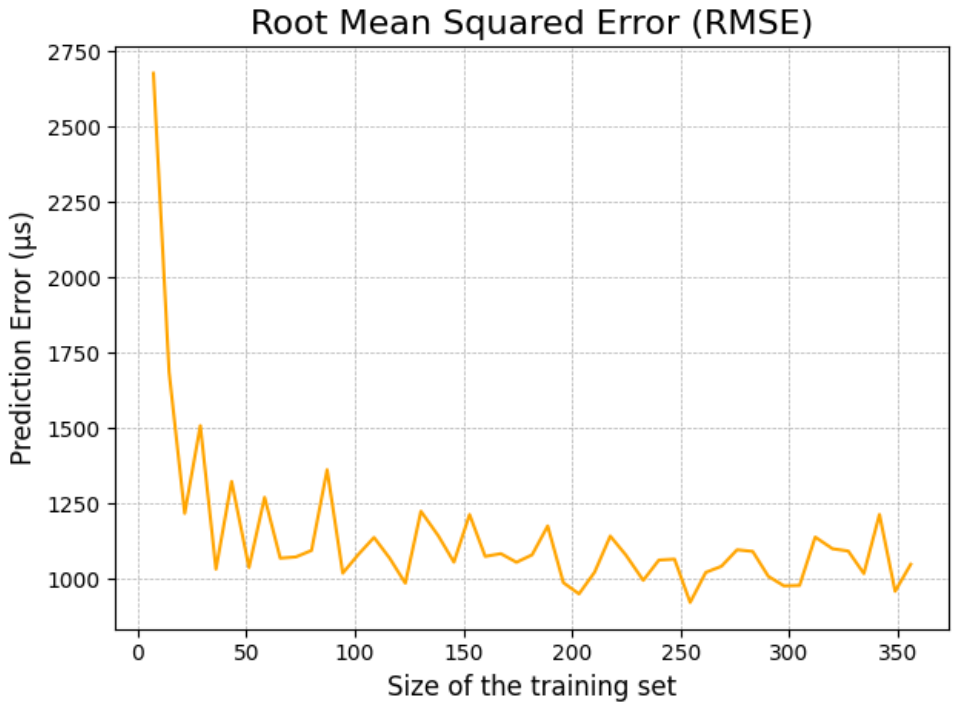


Figure 8.3: Root Mean Squared Error of the model with respect to the size of the training set.

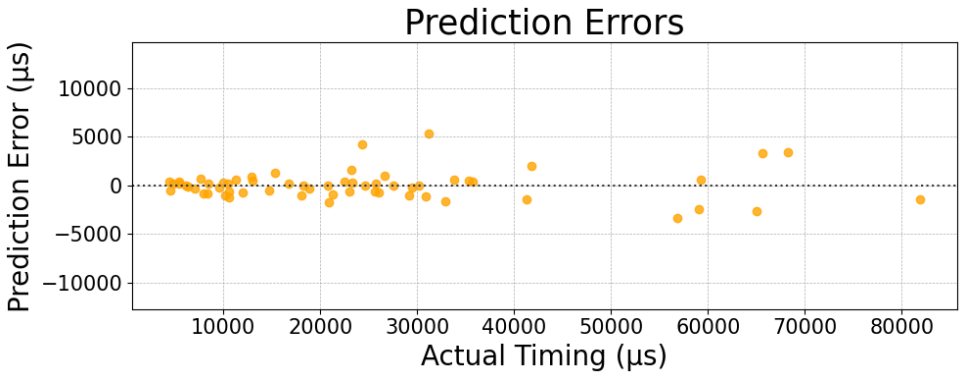


Figure 8.4: Residual plot for a training set of size 50.

a whole grid of partial parameter tuples spanning the entire space. Then, it uses the constraints presented in the previous subsection to determine the noise variances and the range of values of n_{short} that guarantee correct result for a probability up to $1 - p_{\text{err}}$.

Once all the parameters tuples of the grid have been completed, the model trained for (Library, Machine) is used to infer the running times of every possibilities and the best is selected.

8.3 Presentation of ORPHEUS

We now present ORPHEUS (for "Optimized Research of Parameters for Homomorphic Encryption made Universal and Simple"), a Python-based tool for monitoring and debugging homomorphic applications, as well as generating parameter sets for user-defined atomic patterns. The tool is available at the following link (and is planned to be open-sourced in the future):

https://git.cryptoexperts.net/research/tfhe_params_optimizer

In this section, we propose a tour of its capabilities and we explain how its modular structure allows a seamless extension to new libraries and atomic patterns.

ORPHEUS offers three main functionalities:

- *Probing the execution of homomorphic circuits:* One of the pain points encountered by developers of homomorphic applications is the debugging. Indeed, the values manipulated in the homomorphic circuit are encrypted, so when an error occurs it is hard to pinpoint the exact spot of the circuit where noise has overflown. A solution is to propagate the secret key along the circuit but this is a cumbersome and error-prone process. ORPHEUS offers a solution to this problem by allowing the developer to place *probes* in the circuit to monitor the magnitude of the noise at chosen locations. This is useful to trace and visualize the evolution of the noise along the execution. It can be used to build datasets to train cost model or study the noise behavior for a wide range of parameters. Under the hood, this probing functionality works by serializing ciphertexts and their respective keys into an universal format, enabling decryption outside the library.

- *Building machine-tailored cost models:* After producing datasets of experimental measurements for a couple (Library, Machine), ORPHEUS can train a cost model to predict the execution time of an atomic pattern with respect to a given parameter tuple, using the method of Section 8.2.2. By exporting these models, it is possible to use them "offline" on any other machine (such as a simple laptop) to predict running times on the target machine (typically, a production server).

- *Parameter selection:* ORPHEUS also implements the search method presented in Section 8.2.1 to produce parameters for an atomic pattern and a couple (Library, Machine). Users can specify their desired level of security, their target error probability and the plaintext modulus they need. ORPHEUS can then generate an optimal parameter tuple for the cost model corresponding to the target machine. If ORPHEUS is run directly on the target machine, it is able to dynamically validate the produced parameters by running the atomic pattern and directly measuring the actual error probabilities and performances.

Quick Tour of the API

- ```
  python app.py data_generation
  [--max_n_sets [MAX_N_SETS]]
  [--n_tests [N_TESTS]]
  library machine pattern modulus
  ```

  The user specifies the size of the dataset they want to build, the number of iterations to test for each parameter tuple, the target library, an identifier for the machine, an identifier of the atomic pattern and the plaintext modulus to run the experiments with. It outputs a csv file containing every parameter tuples tested, as well as the timings measurements and the noise measurements of each experiment. This file can then be used for easy plotting and visualizations.

- ```
  python app.py training_cost_model
  library machine pattern
  ```

  After having produced a dataset for a configuration (Library, Machine, AP), the user can train a cost model. This produces a serialized model, that can be reused later.

p (bits)	Set	n_{short}	k	N	σ_{short}	σ_{long}	$\mathfrak{B}_{\mathsf{PBS}}$	ℓ_{PBS}	$\mathfrak{B}_{\mathsf{KS}}$	ℓ_{KS}	Runtime server	Runtime laptop	$\log_2(p_{\text{err}})$
1 bit	Original `tfhe-rs`	720	6	256	$2^{48.7}$	$2^{28.4}$	2^{17}	1	2^4	3	7.2	11.4	-68
	ORPHEUS (server)	636	3	512	$2^{50.2}$	$2^{27.1}$	2^{18}	1	2^3	4	5.7	-	-69
	ORPHEUS (laptop)	736	3	512	$2^{47.7}$	$2^{27.1}$	2^{17}	1	2^4	3	-	11.7	-65
2 bits	Original `tfhe-rs`	775	3	512	$2^{47.4}$	$2^{28.4}$	2^{17}	1	2^4	3	6.7	11.0	-65
	ORPHEUS (server)	706	3	512	$2^{48.5}$	$2^{27.1}$	2^{17}	1	2^3	4	6.3	-	-75
	ORPHEUS (laptop)	736	3	512	$2^{48.5}$	$2^{27.1}$	2^{19}	1	2^4	3	-	11.8	-66
3 bits	Original `tfhe-rs`	857	2	1024	$2^{45.3}$	$2^{15.7}$	2^{23}	1	2^5	3	10.8	17.9	-64
	ORPHEUS (server)	750	2	1024	$2^{47.3}$	$2^{13.8}$	2^{22}	1	2^3	5	10.3	-	-65
	ORPHEUS (laptop)	823	2	1024	$2^{45.4}$	$2^{13.8}$	2^{20}	1	2^5	3	-	17.6	-64
4 bits	Original `tfhe-rs`	833	1	2048	$2^{45.9}$	$2^{15.7}$	2^{23}	1	2^3	5	13.2	22.4	-60
	ORPHEUS (server)	798	1	2048	$2^{46.1}$	$2^{13.8}$	2^{20}	1	2^3	5	12.6	-	-75
	ORPHEUS (laptop)	862	1	2048	$2^{47.1}$	$2^{13.8}$	2^{23}	1	2^5	3	-	22.3	-66
5 bits	Original `tfhe-rs`	946	1	4096	$2^{43.1}$	$2^{2.0}$	2^{22}	1	2^4	4	29.5	50.1	-64
	ORPHEUS (server)	862	1	4096	$2^{44.5}$	$2^{2.0}$	2^{22}	1	2^3	6	28.4	-	-68
	ORPHEUS (laptop)	899	1	4096	$2^{43.5}$	$2^{2.0}$	2^{25}	1	2^4	4	-	49.0	-68
6 bits	Original `tfhe-rs`	1005	1	8192	$2^{41.6}$	$2^{2.0}$	2^{22}	1	2^3	7	67.4	124.4	-69
	ORPHEUS (server)	930	1	8192	$2^{42.7}$	$2^{2.0}$	2^{23}	1	2^3	6	60.9	-	-65
	ORPHEUS (laptop)	1006	1	8192	$2^{40.7}$	$2^{2.0}$	2^{23}	1	2^5	4	-	118.2	-65

Table 8.2: Comparisons of the parameter sets `PARAM_MESSAGE_X_CARRY_O_KS_PBS_GAUSSIAN` of `tfhe-rs` and the ones produced by ORPHEUS for the server and the laptop. The timings and the noise are measured and averaged on 500 runs. The runtimes are expressed in milliseconds.

- ```
  python app.py parameter_generation [--retest]
  [--n_tests [N_TESTS]]
  library machine pattern perr p
  ```

  When the cost model is trained, optimized parameter sets can be generated. The user inputs the identifier of the library, the machine, the pattern and the plaintext modulus for which they want to generate parameters. They also specify the error probability they target. The `retest` allows to challenge the produced parameters by running `n_tests` iterations of the atomic pattern, to measure the critical noise and recompute the real error probability to compare with the one estimated by the tool.

ORPHEUS has been designed to be as modular as possible, in order to make it easily extendable to new use-cases. In the following, we present procedures to realize such extensions:

Using ORPHEUS with a new atomic pattern. To work with an atomic pattern, ORPHEUS only needs two components:

- *A declaration file:* In this file, the user gives a high-level description of the atomic pattern. They declare the probes placed in the code, the noise formula of the critical point and the count of basic operations occurring in the atomic pattern. Thanks to probing, it is possible to validate noise formulas and produce visualizations for the noise propagation.

- *The source file:* This is an implementation of the atomic pattern written in the target library. One can directly put probes in it, or in the inner files of the library itself. This file has then to be linked in the declaration file.
  ```

Importing a new library to ORPHEUS. ORPHEUS abstracts all the logic of decryption and noise measurements. While it natively supports `tfhe-rs`, further libraries can be imported. For this purpose, the user has to implement two components:

- *Probes*: For each library, ORPHEUS requires a *key probe* and a *ciphertext probe*. They are simply two pieces of code allowing to serialize a key and a ciphertext into a file following an universal format. This format supports both LWE and GLWE encryptions. ORPHEUS can then use these files to decrypt the ciphertexts and measure the noises. Probes for `tfhe-rs` are already implemented and integrated into the tool. To use ORPHEUS with a specific library, the latter must be recompiled to include the probes.
- *Binding*: ORPHEUS needs to be able to call the *benchmark files* of the atomic patterns. Thus, a binding from Python to the library language is required. Some boilerplates for Rust and C++ are already present in the tool, and they have been already implemented for `tfhe-rs`.

8.4 Experimental Results

We used ORPHEUS to experimentally check the parameter sets hardcoded in `tfhe-rs`. More specifically, we ran many instances of the CJP atomic pattern, measured the variance of the noise at the critical point, and computed the error probability to check that it matches the target values.

We then trained a cost model for two different machines: a server and a laptop. The server features an AMD Ryzen Threadripper PRO 7995WX with 96 cores, with a maximal frequency of 5.4 GHz and 528 GB of RAM. The laptop features an Intel Core i7-10870H @ 2.2 GHz with 8 cores, and 8 GB of RAM. After training the cost model on these machines, we produce parameter sets for both of them using ORPHEUS and compare their performances with the ones obtained with hardcoded set of parameters.

Table 8.2 provides the parameter sets obtained with ORPHEUS for the two considered machines. For the sake of comparison, we further provide the main parameter set named `PARAM_MESSAGE_X_CARRY_0_KS_PBS_GAUSSIAN` hardcoded in `tfhe-rs` (in the `shortint` API). These parameter sets are for a fixed $q = 2^{64}$ (as per the underlying implementation) and are used with varying bit-sizes of p, where `X` indicates the specific bit-size. They target a security level $\lambda = 128$ and error probability of 2^{-64}. We observe that the original sets comply with the claimed error probability of 2^{-64}, except for the case $|p| = 4$ bits which is slightly above. We also observe that the parameter sets obtained with ORPHEUS performs closely but generally slightly better than the hardcoded sets in terms of performances, which is a natural consequence of our more accurate cost model[5].

We also provide in Appendix A.3 some parameters sets generated for the server ensuring an error probability of 2^{-128}. According to the work of [Che+24a; Che+24b], such low error probabilities may be necessary to protect the scheme against attacks exploiting the decryption errors. We see that this increase correctness comes at the performance cost of roughly doubling the execution time.

[5]Note that our results on the server are more accurate compared to those on the laptop, where the predictions deviate more significantly from the actual timings. This discrepancy is likely due to the laptop's reduced stability during computations.

8.5 Conclusion

In this chapter, we presented and formalized the problem of parameter selection for FHE, and more specifically for the TFHE scheme. We explained and quantified the three key properties that a homomorphic application should satisfy: security, correctness and efficiency. We then presented a conceptually-simple procedure to select parameters, based on an exhaustive search made possible by an initial reduction of the parameter space. To bridge the gap between theory and practice, these techniques have been implemented in a tool called ORPHEUS.

ORPHEUS has been designed with flexibility in mind, and is meant to be easily extensible to new use-cases. Future work on this tool will consist in developing plug-ins to support other FHE libraries (such as OpenFHE) and more advanced homomorphic operators.

Conclusion

Fully Homomorphic Encryption is believed to be on the verge of practical usability. However, several challenges have still to be overcome before it can be integrated into everyday applications. The purpose of this thesis was to adress some concrete obstacles that hinder the practical deployment of FHE. In this concluding section, we list the main challenges and relate them to the contributions presented throughout this manuscript.

On Efficiency. FHE still incurs an significant computational overhead compared to traditional, unencrypted computation. While the FHE schemes themselves have been seen much improvement in the last years, an orthogonal direction is to design new algorithms tailored for specific use-cases. In Chapter 4, we developed a framework to accelerate the evaluation of Boolean functions. At the opposite end of the spectrum, Chapter 7 focus on accelerating the evaluation of LUT in larger plaintext spaces than those originally supported by TFHE. Chapter 5 further demonstrated how Boolean and arithmetic representations offer complementary advantages, and how efficient conversion mechanisms between both can enhance performance within homomorphic circuits.

Another way of improving performances lies in selecting appropriate parameters for the scheme. We propose such a procedure of parameter selection, which takes into account the computational circuit to evaluate as well as the environment of execution.

On Data Expansion and Transciphering. Data expansion is another well-known bottleneck in FHE. The literature has long proposed transciphering as a promising solution, however this technique necessitates homomorphic evaluation of the decryption function of a symmetric cipher. In Chapter 5, we have shown how far we could push to evaluate efficiently the AES cipher using TFHE. However, it is clear that standard ciphers such as AES cannot compete against schemes specifically designed with the use-case of transciphering in mind. In Chapter 6, we present such a cipher and show how its design combines the properties required to ensure both cryptographic security and homomorphic efficiency.

On Compilation and Development of Homomorphic Applications. Developing homomorphic applications remains a tedious task, requiring deep understanding of both cryptography and the internal workings of specific FHE schemes. It is clear that an adoption of FHE at scale will require an automated compilation toolchain that will abstract away cryptography complexities from non-expert programmers.

In Chapter 4, we proposed algorithm that automatically compiles Boolean functions into optimized sequences of homomorphic operations. We did a similar thing in the case of large LUTs in Chapter 7, where those LUTs are compiled into a more tractable algorithm, composed of smaller ones. Finally, our parameter selection framework presented in 8 further contributes to this effort.

The road ahead is likely still long before FHE becomes a ubiquitous technology. However, the pace of scientific progress in this field is accelerating rapidly, making it reasonable to believe (and hope) that such a technological revolution could occur in the not-so-distant future.

Bibliography

[24] *Lattigo v6*. Online: https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA. Aug. 2024 (cit. on p. 5).

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer Berlin Heidelberg, Germany, Apr. 2015, pp. 430–454. DOI: 10.1007/978-3-662-46800-5_17 (cit. on p. 62).

[Alb+18] Martin Albrecht, Melissa Chase, Hao Chen, Jintai Ding, Shafi Goldwasser, Sergey Gorbunov, Shai Halevi, Jeffrey Hoffstein, Kim Laine, Kristin Lauter, Satya Lokam, Daniele Micciancio, Dustin Moody, Travis Morrison, Amit Sahai, and Vinod Vaikuntanathan. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, Nov. 2018 (cit. on p. 5).

[AMT22] Tomer Ashur, Mohammad Mahzoun, and Dilara Toprakhisar. "Chaghri - A FHE-friendly Block Cipher". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 139–150. DOI: 10.1145/3548606.3559364 (cit. on p. 62).

[APS15] Martin R. Albrecht, Rachel Player, and Sam Scott. *On the concrete hardness of Learning with Errors*. Cryptology ePrint Archive, Paper 2015/046. 2015. URL: https://eprint.iacr.org/2015/046 (cit. on pp. 8, 88, 89, 113, 117, 121).

[Ara+24] Diego F. Aranha, Antonio Guimarães, Clément Hoffmann, and Pierrick Méaux. *Secure and efficient transciphering for FHE-based MPC*. Cryptology ePrint Archive, Report 2024/1702. 2024. URL: https://eprint.iacr.org/2024/1702 (cit. on p. 92).

[Bad+22] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. *OpenFHE: Open-Source Fully Homomorphic Encryption Library*. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915. 2022. URL: https://eprint.iacr.org/2022/915 (cit. on pp. 5, 114, 116).

[Bau+25] Jules Baudrin, Sonia Belaïd, Nicolas Bon, Christina Boura, Anne Canteaut, Gaëtan Leurent, Pascal Paillier, Léo Perrin, Matthieu Rivain, Yann Rotella, and Samuel Tap. "Transistor: a TFHE-friendly Stream Cipher". In: *Advances in Cryptology - CRYPTO* (2025). URL: https://eprint.iacr.org/2025/282 (cit. on pp. xii, xxi, 62, 77, 81, 84).

[Bea+15] Ray Beaulieu, Douglas Shors, Jason Smith, Stefan Treatman-Clark, Bryan Weeks, and Louis Wingers. "The SIMON and SPECK lightweight block ciphers". In: *Proceedings of the 52nd Annual Design Automation Conference, San Francisco, CA, USA, June 7-11, 2015*. ACM, 2015, 175:1–175:6. DOI: 10.1145/2744769.2747946. URL: https://doi.org/10.1145/2744769.2747946 (cit. on p. 51).

[Bel+24] Mariya Georgieva Belorgey, Sergiu Carpov, Nicolas Gama, Sandra Guasch, and Dimitar Jetchev. "Revisiting Key Decomposition Techniques for FHE: Simpler, Faster and More Generic". In: *Advances in Cryptology – ASIACRYPT 2024, Part I*. Ed. by Kai-Min Chung and Yu Sasaki. Vol. 15484. Lecture Notes in Computer Science. Kolkata, India: Springer, Singapore, Singapore, Dec. 2024, pp. 176–207. DOI: 10.1007/978-981-96-0875-1_6 (cit. on pp. 9, 15).

[Bel+25] Sonia Belaïd, Nicolas Bon, Aymen Boudguiga, Renaud Sirdey, Daphné Trama, and Nicolas Ye. "Further Improvements in AES Execution over TFHE". In: *IACR Commun. Cryptol.* 2.1 (2025), p. 39. DOI: 10.62056/AHMP-4TW9. URL: https://doi.org/10.62056/ahmp-4tw9 (cit. on pp. xi, xxi).

[Ben+22] Adda-Akram Bendoukha, Oana Stan, Renaud Sirdey, Nicolas Quero, and Luciano Freitas de Souza. "Practical Homomorphic Evaluation of Block-Cipher-Based Hash Functions with Applications". In: *Foundations and Practice of Security - 15th International Symposium, FPS 2022, Ottawa, ON, Canada, December 12-14, 2022, Revised Selected Papers*. Ed. by Guy-Vincent Jourdan, Laurent Mounier, Carlisle M. Adams, Florence Sèdes, and Joaquín García-Alfaro. Vol. 13877. Lecture Notes in Computer Science. Springer, 2022, pp. 88–103. DOI: 10.1007/978-3-031-30122-3_6. URL: https://doi.org/10.1007/978-3-031-30122-3%5C_6 (cit. on pp. 52, 59).

[Ber+23a] Loris Bergerat, Anas Boudi, Quentin Bourgerie, Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. "Parameter Optimization and Larger Precision for (T)FHE". In: *Journal of Cryptology* 36.3 (July 2023), p. 28. DOI: 10.1007/s00145-023-09463-5 (cit. on pp. 26, 77, 86, 88, 97, 99, 109, 111–115, 122, 123).

[Ber+23b] Jonas Bertels, Michiel van Beirendonck, Furkan Turan, and Ingrid Verbauwhede. *Hardware Acceleration of FHEW*. Cryptology ePrint Archive, Report 2023/618. 2023. URL: https://eprint.iacr.org/2023/618 (cit. on p. 5).

[Ber+25] Olivier Bernard, Marc Joye, Nigel P. Smart, and Michael Walter. "Drifting Towards Better Error Probabilities in Fully Homomorphic Encryption Schemes". In: *Advances in Cryptology – EUROCRYPT 2025, Part VIII*. Ed. by Serge Fehr and Pierre-Alain Fouque. Vol. 15608. Lecture Notes in Computer Science. Madrid, Spain: Springer, Cham, Switzerland, May 2025, pp. 181–211. DOI: 10.1007/978-3-031-91101-9_7 (cit. on pp. 19, 118, 148).

[BG07] Côme Berbain and Henri Gilbert. "On the Security of IV Dependent Stream Ciphers". In: *Fast Software Encryption – FSE 2007*. Ed. by Alex Biryukov. Vol. 4593. Lecture Notes in Computer Science. Luxembourg, Luxembourg: Springer Berlin Heidelberg, Germany, Mar. 2007, pp. 254–273. DOI: 10.1007/978-3-540-74619-5_17 (cit. on p. 79).

[BGV11] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. *Fully Homomorphic Encryption without Bootstrapping*. Cryptology ePrint Archive, Report 2011/277. 2011. URL: https://eprint.iacr.org/2011/277 (cit. on p. 8).

[BGV12] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ITCS 2012: 3rd Innovations in Theoretical Computer Science*. Ed. by Shafi Goldwasser. Cambridge, MA, USA: Association for Computing Machinery, Jan. 2012, pp. 309–325. DOI: 10.1145/2090236.2090262 (cit. on p. 62).

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) Fully Homomorphic Encryption without Bootstrapping". In: *ACM Trans. Comput. Theory* 6.3 (July 2014). ISSN: 1942-3454. DOI: 10.1145/2633600. URL: https://doi.org/10.1145/2633600 (cit. on pp. 2, 5, 8).

[Bha+19] Sauvik Bhattacharya, Oscar Garcia-Morchon, Rachel Player, and Ludo Tolhuizen. *Achieving secure and efficient lattice-based public-key encryption: the impact of the secret-key distribution*. Cryptology ePrint Archive, Report 2019/389. 2019. URL: https://eprint.iacr.org/2019/389 (cit. on p. 8).

[Bia+24] Beatrice Biasioli, Elena Kirshanova, Chiara Marcolla, and Sergi Rovira. *A Tool for Fast and Secure LWE Parameter Selection: the FHE case*. Cryptology ePrint Archive, Report 2024/1895. 2024. URL: https://eprint.iacr.org/2024/1895 (cit. on p. 113).

[Bit+12] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. "From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again". In: *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*. Ed. by Shafi Goldwasser. ACM, 2012, pp. 326–349. DOI: 10.1145/2090236.2090263. URL: https://doi.org/10.1145/2090236.2090263 (cit. on p. 6).

[Bon+22] Charlotte Bonte, Ilia Iliashenko, Jeongeun Park, Hilder V. L. Pereira, and Nigel P. Smart. "FINAL: Faster FHE Instantiated with NTRU and LWE". In: *Advances in Cryptology – ASIACRYPT 2022, Part II*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13792. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 188–215. DOI: 10.1007/978-3-031-22966-4_7 (cit. on p. 93).

[Bon+24] Antonina Bondarchuk, Olive Chakraborty, Geoffroy Couteau, and Renaud Sirdey. *Downlink (T)FHE ciphertexts compression*. Cryptology ePrint Archive, Report 2024/1921. 2024. URL: https://eprint.iacr.org/2024/1921 (cit. on p. 86).

[Bor+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. "PRINCE - A Low-Latency Block Cipher for Pervasive Computing Applications - Extended Abstract". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Beijing, China: Springer Berlin Heidelberg, Germany, Dec. 2012, pp. 208–225. DOI: 10.1007/978-3-642-34961-4_14 (cit. on p. 62).

[BOS23] Thibault Balenbois, Jean-Baptiste Orfila, and Nigel P. Smart. "Trivial Transciphering With Trivium and TFHE". In: *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*. Ed. by Michael Brenner, Anamaria Costache, and Kurt Rohloff. ACM, 2023, pp. 69–78. DOI: 10.1145/3605759.3625255. URL: https://doi.org/10.1145/3605759.3625255 (cit. on pp. 52, 53, 59, 91, 92).

[Bou+20] Christina Boura, Nicolas Gama, Mariya Georgieva, and Dimitar Jetchev. "CHIMERA: Combining Ring-LWE-based Fully Homomorphic Encryption Schemes". In: *J. Math. Cryptol.* 14.1 (2020), pp. 316–338. DOI: 10.1515/JMC-2019-0026. URL: https://doi.org/10.1515/jmc-2019-0026 (cit. on pp. 9, 15).

[BP10] Joan Boyar and René Peralta. "A New Combinational Logic Minimization Technique with Applications to Cryptology". In: *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*. Ed. by Paola Festa. Vol. 6049. Lecture Notes in Computer Science. Springer, 2010, pp. 178–189. DOI: 10.1007/978-3-642-13193-6_16. URL: https://doi.org/10.1007/978-3-642-13193-6%5C_16 (cit. on pp. 56, 57, 59).

[BPR12] Abhishek Banerjee, Chris Peikert, and Alon Rosen. "Pseudorandom Functions and Lattices". In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. Lecture Notes in Computer Science. Cambridge, UK: Springer Berlin Heidelberg, Germany, Apr. 2012, pp. 719–737. DOI: 10.1007/978-3-642-29011-4_42 (cit. on p. 92).

[BPR24] Nicolas Bon, David Pointcheval, and Matthieu Rivain. "Optimized Homomorphic Evaluation of Boolean Functions". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.3 (2024), pp. 302–341. DOI: 10.46586/tches.v2024.i3.302-341 (cit. on pp. xi, xxi, 26, 61, 62, 73).

[Bra12] Zvika Brakerski. "Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2012, pp. 868–886. DOI: 10.1007/978-3-642-32009-5_50 (cit. on p. 5).

[Bry89] Lennart Brynielsson. "A short proof of the Xiao-Massey lemma". In: *IEEE Trans. Inf. Theory* 35.6 (1989), p. 1344 (cit. on p. 84).

[Brz+25] Chris Brzuska, Sébastien Canard, Caroline Fontaine, Duong Hieu Phan, David Pointcheval, Marc Renard, and Renaud Sirdey. "Relations Among New CCA Security Notions for Approximate FHE". In: *IACR Communications in Cryptology (CiC)* 2.1 (2025), p. 20. DOI: 10.62056/aee0iv7sf (cit. on p. 6).

[Buc+16] Johannes A. Buchmann, Florian Göpfert, Rachel Player, and Thomas Wunderer. "On the Hardness of LWE with Binary Error: Revisiting the Hybrid Lattice-Reduction and Meet-in-the-Middle Attack". In: *AFRICACRYPT 16: 8th International Conference on Cryptology in Africa*. Ed. by David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi. Vol. 9646. Lecture Notes in Computer Science. Fes, Morocco: Springer, Cham, Switzerland, Apr. 2016, pp. 24–43. DOI: 10.1007/978-3-319-31517-1_2 (cit. on p. 8).

[Cam+20] Léopold Cambier, Anahita Bhiwandiwalla, Ting Gong, Mehran Nekuii, Oguz H. Elibol, and Hanlin Tang. "Shifted and Squeezed 8-bit Floating Point format for Low-Precision Training of Deep Neural Networks". In: *CoRR* abs/2001.05674 (2020). arXiv: 2001.05674. URL: https://arxiv.org/abs/2001.05674 (cit. on p. 95).

[Can+16] Anne Canteaut, Sergiu Carpov, Caroline Fontaine, Tancrède Lepoint, María Naya-Plasencia, Pascal Paillier, and Renaud Sirdey. "Stream Ciphers: A Practical Solution for Efficient Homomorphic-Ciphertext Compression". In: *Fast Software Encryption – FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Bochum, Germany: Springer Berlin Heidelberg, Germany, Mar. 2016, pp. 313–333. DOI: 10.1007/978-3-662-52993-5_16 (cit. on p. 62).

[CCS19] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved Bootstrapping for Approximate Homomorphic Encryption". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Cham, Switzerland, May 2019, pp. 34–54. DOI: 10.1007/978-3-030-17656-3_2 (cit. on p. 5).

[Che+17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yong Soo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Hong Kong, China: Springer, Cham, Switzerland, Dec. 2017, pp. 409–437. DOI: 10.1007/978-3-319-70694-8_15 (cit. on pp. xx, 4).

[Che+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for Approximate Homomorphic Encryption". In: *Advances in Cryptology – EUROCRYPT 2018, Part I*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10820. Lecture Notes in Computer Science. Tel Aviv, Israel: Springer, Cham, Switzerland, Apr. 2018, pp. 360–384. DOI: 10.1007/978-3-319-78381-9_14 (cit. on p. 5).

[Che+24a] Marina Checri, Renaud Sirdey, Aymen Boudguiga, and Jean-Paul Bultel. "On the Practical CPA^D Security of "exact" and Threshold FHE Schemes and Libraries". In: *Advances in Cryptology – CRYPTO 2024, Part III*. Ed. by Leonid Reyzin and Douglas Stebila. Vol. 14922. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2024, pp. 3–33. DOI: 10.1007/978-3-031-68382-4_1 (cit. on pp. 6, 76, 90, 117, 128).

[Che+24b] Jung Hee Cheon, Hyeongmin Choe, Alain Passelègue, Damien Stehlé, and Elias Suvanto. "Attacks Against the IND-CPA^D Security of Exact FHE Schemes". In: *ACM CCS 2024: 31st Conference on Computer and Communications Security*. Ed. by Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie. Salt Lake City, UT, USA: ACM Press, Oct. 2024, pp. 2505–2519. DOI: 10.1145/3658644.3690341 (cit. on pp. 6, 76, 90, 117, 128).

[Chi+16] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds". In: *Advances in Cryptology – ASIACRYPT 2016, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. Hanoi, Vietnam: Springer Berlin Heidelberg, Germany, Dec. 2016, pp. 3–33. DOI: 10.1007/978-3-662-53887-6_1 (cit. on p. 7).

[Chi+17] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Hong Kong, China: Springer, Cham, Switzerland, Dec. 2017, pp. 377–408. DOI: 10.1007/978-3-319-70694-8_14 (cit. on p. 7).

[Chi+20] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *Journal of Cryptology* 33.1 (Jan. 2020), pp. 34–91. DOI: 10.1007/s00145-019-09319-x (cit. on pp. xix, 4, 7, 8, 12, 15, 31, 33, 37, 116).

[Chi+21] Ilaria Chillotti, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. "Improved Programmable Bootstrapping with Larger Precision and Efficient Arithmetic Circuits for TFHE". In: *Advances in Cryptology – ASIACRYPT 2021, Part III*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13092. Lecture Notes in Computer

Science. Singapore: Springer, Cham, Switzerland, Dec. 2021, pp. 670–699. DOI: 10.1007/978-3-030-92078-4_23 (cit. on pp. 15, 22, 26, 33, 49, 74, 78, 87, 118).

[Chi18] Ilaria Chillotti. "Vers l'efficacité et la sécurité du chiffrement homomorphe et du cloud computing". PhD thesis. Université Paris-Saclay, 2018 (cit. on pp. 7, 9).

[CHK20] Jung Hee Cheon, Seungwan Hong, and Duhyeong Kim. *Remark on the Security of CKKS Scheme in Practice*. Cryptology ePrint Archive, Report 2020/1581. 2020. URL: https://eprint.iacr.org/2020/1581 (cit. on p. 6).

[Cho+24] Mingyu Cho, Woohyuk Chung, Jincheol Ha, Jooyoung Lee, Eun-Gyeol Oh, and Mincheol Son. "FRAST: TFHE-Friendly Cipher Based on Random S-Boxes". In: *IACR Transactions on Symmetric Cryptology* 2024.3 (2024), pp. 1–43. DOI: 10.46586/tosc.v2024.i3.1-43 (cit. on pp. 77, 92).

[CIM19] Sergiu Carpov, Malika Izabachène, and Victor Mollimard. "New Techniques for Multi-value Input Homomorphic Evaluation and Applications". In: *Topics in Cryptology – CT-RSA 2019*. Ed. by Mitsuru Matsui. Vol. 11405. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Cham, Switzerland, Mar. 2019, pp. 106–126. DOI: 10.1007/978-3-030-12612-4_6 (cit. on p. 65).

[CJP21] Ilaria Chillotti, Marc Joye, and Pascal Paillier. "Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks". In: *Cyber Security Cryptography and Machine Learning - 5th International Symposium, CSCML 2021, Be'er Sheva, Israel, July 8-9, 2021, Proceedings*. Ed. by Shlomi Dolev, Oded Margalit, Benny Pinkas, and Alexander A. Schwarzmann. Vol. 12716. Lecture Notes in Computer Science. Springer, 2021, pp. 1–19. DOI: 10.1007/978-3-030-78086-9_1. URL: https://doi.org/10.1007/978-3-030-78086-9%5C_1 (cit. on p. 119).

[CJS01] Vladimor V. Chepyzhov, Thomas Johansson, and Ben J. M. Smeets. "A Simple Algorithm for Fast Correlation Attacks on Stream Ciphers". In: *Fast Software Encryption – FSE 2000*. Ed. by Bruce Schneier. Vol. 1978. Lecture Notes in Computer Science. New York, NY, USA: Springer Berlin Heidelberg, Germany, Apr. 2001, pp. 181–195. DOI: 10.1007/3-540-44706-7_13 (cit. on p. 84).

[Cle+22] Pierre-Emmanuel Clet, Martin Zuber, Aymen Boudguiga, Renaud Sirdey, and Cédric Gouy-Pailler. *Putting up the swiss army knife of homomorphic calculations by means of TFHE functional bootstrapping*. Cryptology ePrint Archive, Report 2022/149. 2022. URL: https://eprint.iacr.org/2022/149 (cit. on p. 78).

[Cle+23] Pierre-Emmanuel Clet, Aymen Boudguiga, Renaud Sirdey, and Martin Zuber. "ComBo: A Novel Functional Bootstrapping Method for Efficient Evaluation of Nonlinear Functions in the Encrypted Domain". In: *AFRICACRYPT 23: 14th International Conference on Cryptology in Africa*. Ed. by Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne. Vol. 14064. Lecture Notes in Computer Science. Sousse, Tunisia: Springer, Cham, Switzerland, July 2023, pp. 317–343. DOI: 10.1007/978-3-031-37679-5_14 (cit. on p. 26).

[CLT14] Jean-Sébastien Coron, Tancrède Lepoint, and Mehdi Tibouchi. "Scale-Invariant Fully Homomorphic Encryption over the Integers". In: *PKC 2014: 17th International Conference on Theory and Practice of Public Key Cryptography*. Ed. by Hugo Krawczyk. Vol. 8383. Lecture Notes in Computer Science. Buenos Aires, Argentina: Springer Berlin Heidelberg, Germany, Mar. 2014, pp. 311–328. DOI: 10.1007/978-3-642-54631-0_18 (cit. on pp. 56, 57, 59).

[Con+22] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V. L. Pereira. "SortingHat: Efficient Private Decision Tree Evaluation via Homomorphic Encryption and Transciphering". In: *ACM CCS 2022: 29th Conference on Computer and Communications Security*. Ed. by Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi. Los Angeles, CA, USA: ACM Press, Nov. 2022, pp. 563–577. DOI: 10.1145/3548606.3560702 (cit. on p. 93).

[Con23] HEIR Contributors. *HEIR: Homomorphic Encryption Intermediate Representation*. https://github.com/google/heir. 2023 (cit. on p. 5).

[Cos+22] Orel Cosseron, Clément Hoffmann, Pierrick Méaux, and François-Xavier Standaert. "Towards Case-Optimized Hybrid Homomorphic Encryption - Featuring the Elisabeth Stream Cipher". In: *Advances in Cryptology – ASIACRYPT 2022, Part III*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13793. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 32–67. DOI: 10.1007/978-3-031-22969-5_2 (cit. on p. 62).

[Cou+23] David Bruce Cousins, Yuriy Polyakov, Ahmad Al Badawi, Matthew French, Andrew Schmidt, Ajey Jacob, Benedict Reynwar, Kellie Canida, Akhilesh Jaiswal, Clynn Mathew, Homer Gamil, Negar Neda, Deepraj Soni, Michail Maniatakos, Brandon Reagen, Naifeng Zhang, Franz Franchetti, Patrick Brinich, Jeremy Johnson, Patrick Broderick, Mike Franusich, Bo Zhang, Zeming Cheng, and Massoud Pedram. *TREBUCHET: Fully Homomorphic Encryption Accelerator for Deep Computation*. Cryptology ePrint Archive, Report 2023/521. 2023. URL: https://eprint.iacr.org/2023/521 (cit. on p. 5).

[CP19] Benjamin R. Curtis and Rachel Player. "On the Feasibility and Impact of Standardising Sparse-secret LWE Parameter Sets for Homomorphic Encryption". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC@CCS 2019, London, UK, November 11-15, 2019*. Ed. by Michael Brenner, Tancrède Lepoint, and Kurt Rohloff. ACM, 2019, pp. 1–10. DOI: 10.1145/3338469.3358940. URL: https://doi.org/10.1145/3338469.3358940 (cit. on p. 8).

[CRV14] Jean-Sébastien Coron, Arnab Roy, and Srinivas Vivek. "Fast Evaluation of Polynomials over Binary Finite Fields and Application to Side-Channel Countermeasures". In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Busan, South Korea: Springer Berlin Heidelberg, Germany, Sept. 2014, pp. 170–187. DOI: 10.1007/978-3-662-44709-3_10 (cit. on p. 95).

[CT00] Anne Canteaut and Michaël Trabbia. "Improved Fast Correlation Attacks Using Parity-Check Equations of Weight 4 and 5". In: *Advances in Cryptology – EUROCRYPT 2000*. Ed. by Bart Preneel. Vol. 1807. Lecture Notes in Computer Science. Bruges, Belgium: Springer Berlin Heidelberg, Germany, May 2000, pp. 573–588. DOI: 10.1007/3-540-45539-6_40 (cit. on p. 84).

[Das+18] Dipankar Das, Naveen Mellempudi, Dheevatsa Mudigere, Dhiraj D. Kalamkar, Sasikanth Avancha, Kunal Banerjee, Srinivas Sridharan, Karthik Vaidyanathan, Bharat Kaul, Evangelos Georganas, Alexander Heinecke, Pradeep Dubey, Jesús Corbal, Nikita Shustrov, Roman Dubtsov, Evarist Fomenko, and Vadim O. Pirogov. "Mixed Precision Training of Convolutional Neural Networks using Integer Operations". In: *CoRR* abs/1802.00930 (2018). arXiv: 1802.00930. URL: http://arxiv.org/abs/1802.00930 (cit. on p. 95).

[De 06] Christophe De Cannière. "Trivium: A Stream Cipher Construction Inspired by Block Cipher Design Principles". In: *ISC 2006: 9th International Conference on Information Security*. Ed. by Sokratis K. Katsikas, Javier Lopez, Michael Backes, Stefanos Gritzalis, and Bart Preneel. Vol. 4176. Lecture Notes in Computer Science. Samos Island, Greece: Springer Berlin Heidelberg, Germany, Aug. 2006, pp. 171–186. DOI: 10.1007/11836810_13 (cit. on pp. 52, 53).

[Deo+24] Amit Deo, Marc Joye, Benoit Libert, Benjamin R. Curtis, and Mayeul de Bellabre. *Homomorphic Evaluation of LWR-based PRFs and Application to Transciphering*. Cryptology ePrint Archive, Report 2024/665. 2024. URL: https://eprint.iacr.org/2024/665 (cit. on p. 92).

[Dij+10] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. "Fully Homomorphic Encryption over the Integers". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. French Riviera: Springer Berlin Heidelberg, Germany, May 2010, pp. 24–43. DOI: 10.1007/978-3-642-13190-5_2 (cit. on p. 2).

[DM15] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Sofia, Bulgaria: Springer Berlin Heidelberg, Germany, Apr. 2015, pp. 617–640. DOI: 10.1007/978-3-662-46800-5_24 (cit. on pp. 7, 17).

[Dob+18] Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. "Rasta: A Cipher with Low ANDdepth and Few ANDs per Bit". In: *Advances in Cryptology – CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2018, pp. 662–692. DOI: 10.1007/978-3-319-96884-1_22 (cit. on p. 93).

[Dob+19] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to Round 1 of the NIST Lightweight Cryptography project. 2019. URL: https://csrc.nist.gov/CSRC/media/Projects/Lightweight-Cryptography/documents/round-1/spec-doc/ascon-spec.pdf (cit. on p. 54).

[Dob+21] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. "Ascon v1.2: Lightweight Authenticated Encryption and Hashing". In: *Journal of Cryptology* 34.3 (July 2021), p. 33. DOI: 10.1007/s00145-021-09398-9 (cit. on pp. 54, 55).

[Dob+23] Christoph Dobraunig, Lorenzo Grassi, Lukas Helminger, Christian Rechberger, Markus Schofnegger, and Roman Walch. "Pasta: A Case for Hybrid Homomorphic Encryption". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.3 (2023), pp. 30–73. DOI: 10.46586/tches.v2023.i3.30-73 (cit. on p. 62).

[DR00] Joan Daemen and Vincent Rijmen. "The Block Cipher Rijndael". In: *Smart Card Research and Applications*. Ed. by Jean-Jacques Quisquater and Bruce Schneier. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, pp. 277–284. ISBN: 978-3-540-44534-0 (cit. on pp. 56, 64, 65).

[FV12] Junfeng Fan and Frederik Vercauteren. *Somewhat Practical Fully Homomorphic Encryption*. Cryptology ePrint Archive, Report 2012/144. 2012. URL: https://eprint.iacr.org/2012/144 (cit. on p. 5).

[GBA21] Antonio Guimarães, Edson Borin, and Diego F. Aranha. "Revisiting the functional bootstrap in TFHE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021.2 (2021), pp. 229–253. ISSN: 2569-2925. DOI: 10.46586/tches.v2021.i2.229-253. URL: https://tches.iacr.org/index.php/TCHES/article/view/8793 (cit. on pp. 22, 65, 67, 78).

[Gee+23] Robin Geelen, Michiel Van Beirendonck, Hilder V. L. Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios D. Dimou, Ingrid Verbauwhede, Frederik Vercauteren, and David W. Archer. "BASALISC: Programmable Hardware Accelerator for BGV Fully Homomorphic Encryption". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.4 (2023), pp. 32–57. DOI: 10.46586/tches.v2023.i4.32-57 (cit. on p. 5).

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *41st Annual ACM Symposium on Theory of Computing*. Ed. by Michael Mitzenmacher. Bethesda, MD, USA: ACM Press, May 2009, pp. 169–178. DOI: 10.1145/1536414.1536440 (cit. on pp. xviii, 2).

[GHS12] Craig Gentry, Shai Halevi, and Nigel P. Smart. "Homomorphic Evaluation of the AES Circuit". In: *Advances in Cryptology – CRYPTO 2012*. Ed. by Reihaneh Safavi-Naini and Ran Canetti. Vol. 7417. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2012, pp. 850–867. DOI: 10.1007/978-3-642-32009-5_49 (cit. on pp. 56, 57, 59, 62, 120).

[GMP19] Nicholas Genise, Daniele Micciancio, and Yuriy Polyakov. "Building an Efficient Lattice Gadget Toolkit: Subgaussian Sampling and More". In: *Advances in Cryptology – EUROCRYPT 2019, Part II*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11477. Lecture Notes in Computer Science. Darmstadt, Germany: Springer, Cham, Switzerland, May 2019, pp. 655–684. DOI: 10.1007/978-3-030-17656-3_23 (cit. on p. 13).

[Gou+17] Dahmun Goudarzi, Matthieu Rivain, Damien Vergnaud, and Srinivas Vivek. "Generalized Polynomial Decomposition for S-boxes with Application to Side-Channel Countermeasures". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Sept. 2017, pp. 154–171. DOI: 10.1007/978-3-319-66787-4_8 (cit. on pp. 95, 106).

[GR16] Dahmun Goudarzi and Matthieu Rivain. "On the Multiplicative Complexity of Boolean Functions and Bitsliced Higher-Order Masking". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2016, pp. 457–478. DOI: 10.1007/978-3-662-53140-2_22 (cit. on pp. 95, 106).

[GSW13] Craig Gentry, Amit Sahai, and Brent Waters. "Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based". In: *Advances in Cryptology – CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer Berlin Heidelberg, Germany, Aug. 2013, pp. 75–92. DOI: 10.1007/978-3-642-40041-4_5 (cit. on pp. xv, 15).

[HK20] Kyoohyung Han and Dohyeong Ki. "Better Bootstrapping for Approximate Homomorphic Encryption". In: *Topics in Cryptology – CT-RSA 2020*. Ed. by Stanislaw Jarecki. Vol. 12006. Lecture Notes in Computer Science. San Francisco, CA, USA: Springer, Cham, Switzerland, Feb. 2020, pp. 364–390. DOI: 10.1007/978-3-030-40186-3_16 (cit. on p. 5).

[HS20] Shai Halevi and Victor Shoup. *Design and implementation of HElib: a homomorphic encryption library*. Cryptology ePrint Archive, Report 2020/1481. 2020. URL: https://eprint.iacr.org/2020/1481 (cit. on p. 62).

[Inc20] Cryptolab Inc. *HEaaN : Fully homomorphic encryption with CKKS scheme*. https://github.com/virtualsecureplatform/TFHEpp. 2020 (cit. on p. 5).

[Joy21] Marc Joye. "Balanced Non-adjacent Forms". In: *Advances in Cryptology – ASIACRYPT 2021, Part III*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13092. Lecture Notes in Computer Science. Singapore: Springer, Cham, Switzerland, Dec. 2021, pp. 553–576. DOI: 10.1007/978-3-030-92078-4_19 (cit. on p. 13).

[Kim+22] Seonghak Kim, Minji Park, Jaehyung Kim, Taekyung Kim, and Chohong Min. "EvalRound Algorithm in CKKS Bootstrapping". In: *Advances in Cryptology – ASIACRYPT 2022, Part II*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13792. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 161–187. DOI: 10.1007/978-3-031-22966-4_6 (cit. on p. 5).

[Kös+17] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K. Bansal, William Constable, Oguz Elibol, Stewart Hall, Luke Hornof, Amir Khosrowshahi, Carey Kloss, Ruby J. Pai, and Naveen Rao. "Flexpoint: An Adaptive Numerical Format for Efficient Training of Deep Neural Networks". In: *CoRR* abs/1711.02213 (2017). arXiv: 1711.02213. URL: http://arxiv.org/abs/1711.02213 (cit. on p. 95).

[Kri+24] Florian Krieger, Florian Hirner, Ahmet Can Mert, and Sujoy Sinha Roy. *OpenNTT: An Automated Toolchain for Compiling High-Performance NTT Accelerators in FHE*. Cryptology ePrint Archive, Report 2024/1740. 2024. URL: https://eprint.iacr.org/2024/1740 (cit. on p. 5).

[KS23] Kamil Kluczniak and Leonard Schild. "FDFB: Full Domain Functional Bootstrapping Towards Practical Fully Homomorphic Encryption". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2023.1 (2023), pp. 501–537. DOI: 10.46586/tches.v2023.i1.501-537 (cit. on pp. 26, 78).

[LM21] Baiyu Li and Daniele Micciancio. "On the Security of Homomorphic Encryption on Approximate Numbers". In: *Advances in Cryptology – EUROCRYPT 2021, Part I*. Ed. by Anne Canteaut and François-Xavier Standaert. Vol. 12696. Lecture Notes in Computer Science. Zagreb, Croatia: Springer, Cham, Switzerland, Oct. 2021, pp. 648–677. DOI: 10.1007/978-3-030-77870-5_23 (cit. on pp. 6, 76).

[LMP22] Zeyu Liu, Daniele Micciancio, and Yuriy Polyakov. "Large-Precision Homomorphic Sign Evaluation Using FHEW/TFHE Bootstrapping". In: *Advances in Cryptology – ASIACRYPT 2022, Part II*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13792. Lecture Notes in Computer Science. Taipei, Taiwan: Springer, Cham, Switzerland, Dec. 2022, pp. 130–160. DOI: 10.1007/978-3-031-22966-4_5 (cit. on p. 26).

[Lof+12] Jake Loftus, Alexander May, Nigel P. Smart, and Frederik Vercauteren. "On CCA-Secure Somewhat Homomorphic Encryption". In: *SAC 2011: 18th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Ali Miri and Serge Vaudenay. Vol. 7118. Lecture Notes in Computer Science. Toronto, Ontario, Canada: Springer Berlin Heidelberg, Germany, Aug. 2012, pp. 55–72. DOI: 10.1007/978-3-642-28496-0_4 (cit. on p. 6).

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology – EUROCRYPT 2010*. Ed. by Henri Gilbert. Vol. 6110. Lecture Notes in Computer Science. French Riviera: Springer Berlin Heidelberg, Germany, May 2010, pp. 1–23. DOI: 10.1007/978-3-642-13190-5_1 (cit. on p. 8).

[Mat20] K. Matsuoka. *TFHEpp: pure C++ implementation of TFHE cryptosystem*. https://github.com/virtualsecureplatform/TFHEpp. 2020 (cit. on pp. 5, 74).

[Max19] Alexander Maximov. *AES MixColumn with 92 XOR gates*. Cryptology ePrint Archive, Report 2019/833. 2019. URL: https://eprint.iacr.org/2019/833 (cit. on pp. 57, 59, 69).

[Méa+16] Pierrick Méaux, Anthony Journault, François-Xavier Standaert, and Claude Carlet. "Towards Stream Ciphers for Efficient FHE with Low-Noise Ciphertexts". In: *Advances in Cryptology – EUROCRYPT 2016, Part I*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9665. Lecture Notes in Computer Science. Vienna, Austria: Springer Berlin Heidelberg, Germany, May 2016, pp. 311–343. DOI: 10.1007/978-3-662-49890-3_13 (cit. on p. 82).

[MN24] Mark Manulis and Jérôme Nguyen. "Fully Homomorphic Encryption Beyond IND-CCA1 Security: Integrity Through Verifiability". In: *Advances in Cryptology – EUROCRYPT 2024, Part II*. Ed. by Marc Joye and Gregor Leander. Vol. 14652. Lecture Notes in Computer Science. Zurich, Switzerland: Springer, Cham, Switzerland, May 2024, pp. 63–93. DOI: 10.1007/978-3-031-58723-8_3 (cit. on p. 6).

[MPP24] Pierrick Méaux, Jeongeun Park, and Hilder V. L. Pereira. "Towards Practical Transciphering for FHE with Setup Independent of the Plaintext Space". In: *IACR Communications in Cryptology (CiC)* 1.1 (2024), p. 20. DOI: 10.62056/anxrxrxqi (cit. on p. 93).

[MS88] Willi Meier and Othmar Staffelbach. "Fast Correlation Attacks on Stream Ciphers (Extended Abstract)". In: *Advances in Cryptology – EUROCRYPT'88*. Ed. by C. G. Günther. Vol. 330. Lecture Notes in Computer Science. Davos, Switzerland: Springer Berlin Heidelberg, Germany, May 1988, pp. 301–314. DOI: 10.1007/3-540-45961-8_28 (cit. on p. 84).

[NIS15] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf. 2015 (cit. on pp. 53, 79, 80).

[Niu+25] Chao Niu, Zhicong Huang, Zhaomin Yang, Yi Chen, Liang Kong, Cheng Hong, and Tao Wei. *XBOOT: Free-XOR Gates for CKKS with Applications to Transciphering*. Cryptology ePrint Archive, Report 2025/074. 2025. URL: https://eprint.iacr.org/2025/074 (cit. on pp. 92, 93).

[NLV11] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. "Can homomorphic encryption be practical?" In: *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*. Ed. by Christian Cachin and Thomas Ristenpart. ACM, 2011, pp. 113–124. URL: https://dl.acm.org/citation.cfm?id=2046682 (cit. on p. 62).

[Pai99] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology – EUROCRYPT'99*. Ed. by Jacques Stern. Vol. 1592. Lecture Notes in Computer Science. Prague, Czech Republic: Springer Berlin Heidelberg, Germany, May 1999, pp. 223–238. DOI: 10.1007/3-540-48910-X_16 (cit. on p. 2).

[RAD78] Ronald L. Rivest, Len Adleman, and Michael L. Dertouzos. "On Data Banks and Privacy Homomorphisms". In: (1978). Ed. by Richard A. DeMillo, David P. Dobkin, Anita K. Jones, and Richard J. Lipton, pp. 165–179 (cit. on p. 2).

[Reg05] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *37th Annual ACM Symposium on Theory of Computing*. Ed. by Harold N. Gabow and Ronald Fagin. Baltimore, MA, USA: ACM Press, May 2005, pp. 84–93. DOI: 10.1145/1060590.1060603 (cit. on pp. 7, 117).

[Sak+21] Kosei Sakamoto, Fukang Liu, Yuto Nakano, Shinsaku Kiyomoto, and Takanori Isobe. "Rocca: An Efficient AES-based Encryption Scheme for Beyond 5G". In: *IACR Transactions on Symmetric Cryptology* 2021.2 (2021), pp. 1–30. ISSN: 2519-173X. DOI: 10.46586/tosc.v2021.i2.1-30 (cit. on p. 84).

[Sha+24] Mingyao Shao, Yuejun Liu, Yongbin Zhou, and Yan Shao. *On the Security of LWE-based KEMs under Various Distributions: A Case Study of Kyber*. Cryptology ePrint Archive, Report 2024/1979. 2024. URL: https://eprint.iacr.org/2024/1979 (cit. on p. 8).

[Sly99] V. I. Slyusar. "A family of face products of matrices and its properties". en. In: *Cybernetics and Systems Analysis* 35.3 (May 1999), pp. 379–384. ISSN: 1060-0396, 1573-8337. DOI: 10.1007/BF02733426. URL: http://link.springer.com/10.1007/BF02733426 (cit. on p. 102).

[ST25] National Institute of Standards and Technology. *The NIST Threshold Call*. 2025. URL: https://csrc.nist.gov/Projects/threshold-cryptography (cit. on pp. 5, 61, 64).

[Ste+09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. "Efficient Public Key Encryption Based on Ideal Lattices". In: *Advances in Cryptology – ASIACRYPT 2009*. Ed. by Mitsuru Matsui. Vol. 5912. Lecture Notes in Computer Science. Tokyo, Japan: Springer Berlin Heidelberg, Germany, Dec. 2009, pp. 617–635. DOI: 10.1007/978-3-642-10366-7_36 (cit. on p. 8).

[Sun+19] Xiao Sun, Jungwook Choi, Chia-Yu Chen, Naigang Wang, Swagath Venkataramani, Vijayalakshmi Srinivasan, Xiaodong Cui, Wei Zhang, and Kailash Gopalakrishnan. "Hybrid 8-bit Floating Point (HFP8) Training and Inference for Deep Neural Networks". In: *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*. Ed. by Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett. 2019, pp. 4901–4910. URL: https://proceedings.neurips.cc/paper/2019/hash/65fc9fb4897a89789352e211ca2d398f-Abstract.html (cit. on p. 95).

[Tap23] Samuel Tap. "Construction de nouveaux outils de chiffrement homomorphe efficace". 2023URENS103. PhD thesis. 2023. URL: http://www.theses.fr/2023URENS103/document (cit. on pp. 12, 14, 15, 117, 120, 122, 147, 150).

[Tod+18] Yosuke Todo, Takanori Isobe, Willi Meier, Kazumaro Aoki, and Bin Zhang. "Fast Correlation Attack Revisited - Cryptanalysis on Full Grain-128a, Grain-128, and Grain-v1". In: *Advances in Cryptology – CRYPTO 2018, Part II*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10992. Lecture Notes in Computer Science. Santa Barbara, CA, USA: Springer, Cham, Switzerland, Aug. 2018, pp. 129–159. DOI: 10.1007/978-3-319-96881-0_5 (cit. on p. 84).

[Tra+23] Daphné Trama, Pierre-Emmanuel Clet, Aymen Boudguiga, and Renaud Sirdey. "A Homomorphic AES Evaluation in Less than 30 Seconds by Means of TFHE". In: *Proceedings of the 11th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, Copenhagen, Denmark, 26 November 2023*. Ed. by Michael Brenner, Anamaria Costache, and Kurt Rohloff. ACM, 2023, pp. 79–90. DOI: 10.1145/3605759.3625260. URL: https://doi.org/10.1145/3605759.3625260 (cit. on pp. 56, 57, 59, 61, 62, 64–67, 69, 72, 73).

[Wei+23] Benqiang Wei, Ruida Wang, Zhihao Li, Qinju Liu, and Xianhui Lu. "Fregata: Faster Homomorphic Evaluation of AES via TFHE". In: *ISC 2023: 26th International Conference on Information Security*. Ed. by Elias Athanasopoulos and Bart Mennink. Vol. 14411. Lecture Notes in Computer Science. Groningen, The Netherlands: Springer, Cham, Switzerland, Nov. 2023, pp. 392–412. DOI: 10.1007/978-3-031-49187-0_20 (cit. on pp. 61, 62, 73, 74).

[Wei+24] Benqiang Wei, Xianhui Lu, Ruida Wang, Kun Liu, Zhihao Li, and Kunpeng Wang. "Thunderbird: Efficient Homomorphic Evaluation of Symmetric Ciphers in 3GPP by combining two modes of TFHE". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2024.3 (2024), pp. 530–573. DOI: 10.46586/tches.v2024.i3.530-573 (cit. on pp. 61, 62, 73, 74).

[XM88] Guo-Zhen Xiao and James L. Massey. "A spectral characterization of correlation-immune combining functions". In: *IEEE Trans. Inf. Theory* 34.3 (1988), pp. 569–571. DOI: 10.1109/18.6037 (cit. on p. 84).

[Yan+21] Zhaomin Yang, Xiang Xie, Huajie Shen, Shiying Chen, and Jun Zhou. *TOTA: Fully Homomorphic Encryption with Smaller Parameters and Stronger Security*. Cryptology ePrint Archive, Report 2021/1347. 2021. URL: https://eprint.iacr.org/2021/1347 (cit. on p. 26).

[Zam22a] Zama. *concrete-optimizer : Concrete Optimizer is a Rust library that find the best cryptographic parameters for a given TFHE homomorphic circuit*. https://github.com/zama-ai/concrete/tree/main/compilers/concrete-optimizer. 2022 (cit. on pp. 49, 51, 52).

[Zam22b] Zama. *Concrete: TFHE Compiler that converts python programs into FHE equivalent*. https://github.com/zama-ai/concrete. 2022 (cit. on pp. 5, 114, 115).

[Zam22c] Zama. *TFHE-rs: A Pure Rust Implementation of the TFHE Scheme for Boolean and Integer Arithmetics Over Encrypted Data*. https://github.com/zama-ai/tfhe-rs. 2022 (cit. on pp. 5, 33, 49–51, 58, 75, 91, 114, 116).

Appendix

A Supplementary Material on Parameter Selection

A.1 More Details on the CJP Atomic Pattern

The goal of this appendix is to go through all steps of the CJP atomic pattern, to incrementally construct the final noise formula of Equation 8.6. All the formulas of this section as well as their proof can be found in [Tap23] and we use the notations introduced in Figure 8.1. To simplify the formulas, we make the assumption that the bits of the secret key are produced by a balanced uniform distribution in $\mathbb{B}$.

Let us start by a ciphertext at a "nominal" level of noise (the noise in output of a PBS). Its noise σ_{PBS} can be expressed as:

$$\sigma_{PBS}^2 = n_{\text{short}} \cdot \ell_{\mathsf{PBS}}(k+1)N\frac{\mathfrak{B}_{\mathsf{PBS}}^2}{12}\sigma_{\text{long}}^2 \tag{1}$$

$$+ n_{\text{short}} \cdot \frac{q^2 - \mathfrak{B}_{\mathsf{PBS}}^{2\ell_{\mathsf{PBS}}}}{24\mathfrak{B}_{\mathsf{PBS}}^{2\ell_{\mathsf{PBS}}}} \cdot \frac{kN}{2} \tag{2}$$

$$+ n_{\text{short}} \cdot \frac{kN}{32} \tag{3}$$

$$+ \frac{n_{\text{short}}}{16}\left(1 - \frac{kN}{2}\right)^2. \tag{4}$$

We estimated the noise for a wide range of parameters tuple to identify which terms can be safely neglected. Figure A.1 shows the magnitude of each of the four term of the sum for each parameter tuple. It is clear that Terms (3) and (4) are not useful.

As we are looping, we treat this noise as the input of the atomic pattern. Now, the noise gets multiplied by ν (which is the norm of the vector of coefficients of the linear combination). Thus, the noise incoming in the Atomic pattern is:

$$\sigma_{\text{in}}^2 = \nu^2 \cdot \sigma_{\text{PBS}}^2 \tag{A.1}$$

Within the PBS process, the first step to be done is the `KeySwitch`. The noise added in the ciphertext by this procedure is only additive. This yields the following theoretical formula:

$$\sigma_{\text{KS}}^2 = \sigma_{\text{in}}^2 \tag{1}$$

$$+ \frac{n_{\text{long}}}{2} \cdot \frac{q^2}{12\mathfrak{B}_{\mathsf{KS}}^{2\ell_{\mathsf{KS}}}} \tag{2}$$

$$+ \frac{n_{\text{long}}}{16} \tag{3}$$

$$+ n_{\text{long}} \cdot \ell_{\mathsf{KS}} \cdot \sigma_{\text{short}}^2 \cdot \frac{\mathfrak{B}_{\mathsf{KS}}^2}{12}. \tag{4}$$

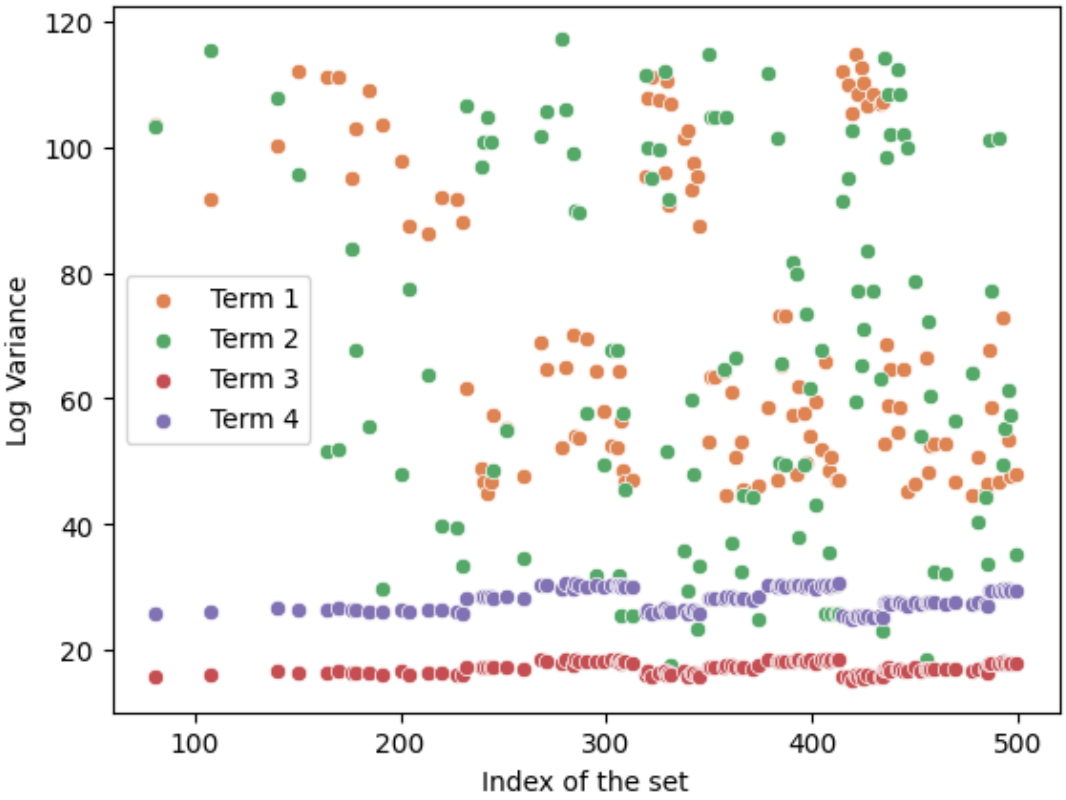


Figure A.1: Analysis of the noise after the PBS

A simulation analogous to the one we did for the PBS step is represented on Figure A.2. It shows that the third term can be safely discarded.

Then, the `ModSwitch`operation is performed on the ciphertext. This brings its components into the ring $\mathbb{Z}_{2N}$. The noise is accordingly scaled down, and an extra error due to the rounding errors (so-called *drift* and extensively studied in [Ber+25]) arises. This gives:

$$\sigma^2_{\text{MS}} = \frac{4N^2}{q^2}\sigma^2_{\text{KS}} \tag{1}$$
$$+ \frac{1}{12} \tag{2}$$
$$- \frac{4N^2}{12q^2} \tag{3}$$
$$+ \frac{n_{\text{short}}}{24} \tag{4}$$
$$+ \frac{n_{\text{short}} \cdot 4N^2}{48q^2}. \tag{5}$$

Figure A.3 shows that Terms (2), (3) and (5) can be neglected.

Note that [Ber+25] applies the same than us, but they work in $\mathbb{Z}_q$ instead of $\mathbb{Z}_{2N}$, so the formula looks different.

This point is the *critical point* in the atomic pattern, that is to say the point where the noise is maximal. We rename accordingly σ_{critical}.

Afterwards, the `BlindRotate`creates a ciphertext whose noise is independent of the input. Moreover, `SampleExtract`does not add any noise. So, we can write:

$$\sigma_{\text{BR}} = \sigma_{\text{PBS}} \tag{A.2}$$

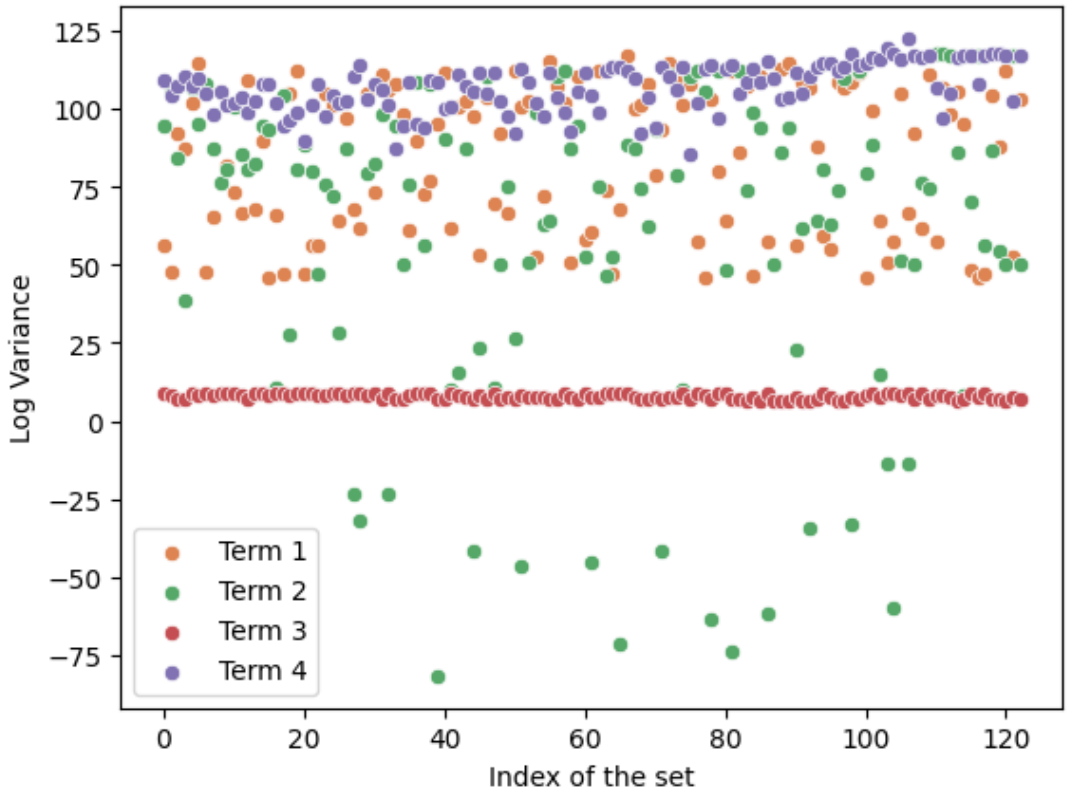


Figure A.2: Analysis of the noise after the KS

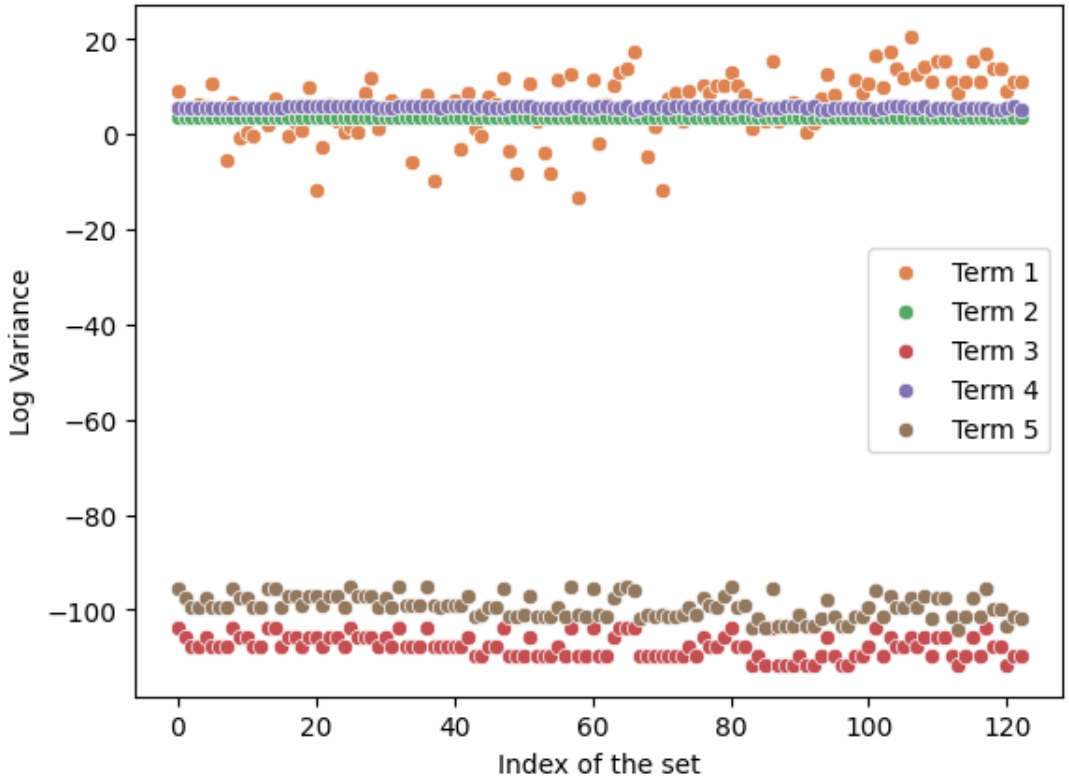


Figure A.3: Analysis of the noise after the MS

Putting it all together, we can write the formula of the critical variance with respect to a given parameter tuple:

$$\begin{aligned}\sigma^2_{\text{critical}} = \frac{4N^2}{q^2}\Bigg[\nu^2 \cdot \Bigg(n_{\text{short}} \cdot \Bigg(\ell_{\mathsf{PBS}}(k+1)N\frac{\mathfrak{B}^2_{\mathsf{PBS}}}{12}\sigma^2_{\text{long}}\Bigg) \\ + n_{\text{short}} \cdot \frac{q^2 - \mathfrak{B}^{2\ell_{\mathsf{PBS}}}_{\mathsf{PBS}}}{24\mathfrak{B}^{2\ell_{\mathsf{PBS}}}_{\mathsf{PBS}}} \cdot \frac{kN}{2}\Bigg) \\ + \frac{n_{\text{long}}}{2} \cdot \frac{q^2}{12\mathfrak{B}^{2\ell_{\mathsf{KS}}}_{\mathsf{KS}}} \\ + n_{\text{long}} \cdot \ell_{\mathsf{KS}} \cdot \sigma^2_{\text{short}} \cdot \frac{\mathfrak{B}^2_{\mathsf{KS}}}{12}\Bigg] \\ + \frac{n_{\text{short}}}{24}.\end{aligned} \tag{A.3}$$

A.2 Operations Counts and Complexities in CJP Atomic Pattern

Table A.1 gives an example of the cost formula for the CJP atomic pattern, as well as the linear coefficients found by the linear regression. Note that the data in Columns 2 and 3 can be found in [Tap23] while some notations are different. In particular, the keyswitch formula presented in there consider the case of switching a LWE ciphertext of dimension n to a GLWE ciphertext of dimension (k, N). In the CJP atomic pattern, we switch a LWE of dimension $k \cdot N$ to a LWE of dimension n_{short}.

Table A.1: Example of a cost formula for the CJP atomic pattern on the couple (tfhe-rs,server) with server a machine equipped with an AMD Ryzen Threadripper PRO 7995WX with 96 cores, with a maximal frequency of 5.4 GHz and 528 GB of RAM.

Subroutine $\mathcal{A}_i$	$\texttt{Count}_{\mathbf{AP}}(\mathcal{A}_i)$	$\texttt{Complexity}_{\mathcal{P}}(\mathcal{A}_i)$	$\texttt{LinearCoeff}(\mathcal{A}_i)$
Gadget decomposition in PBS	$n_{\text{short}} \cdot (k+1) \cdot N$	ℓ_{PBS}	0.00111174
Gadget decomposition in KS	$k \cdot N$	ℓ_{KS}	
Addition of ciphertexts	$(\ell_{\mathsf{KS}} - 1) \cdot k \cdot N$	n_{short}	0
Coeff-wise multiplication of ciphertexts	$\ell_{\mathsf{KS}} \cdot k \cdot N$	n_{short}	0.00014987
FFT	$n_{\text{short}} \cdot (k+1) \cdot \ell_{\mathsf{PBS}}$	$N \log N$	0
Multiplications in the FFT domain	$n_{\text{short}} \cdot \ell_{\mathsf{PBS}} \cdot (k+1)^2$	N	0.00034434
Additions in the FFT domain	$(k+1) \cdot (\ell_{\mathsf{PBS}} \cdot (k+1) - 1)$	N	0
iFFT	$(k+1)$	$N \log N$	0.1213151

A.3 Parameters for $p_{\mathsf{err}} = 2^{128}$

Using ORPHEUS, we also generated sets of parameters that ensures an error probability of 2^{128}. They are listed in Table A.2.

Table A.2: Parameters sets generated by ORPHEUS for `tfhe-rs` running on server, targeting $p_{err} = 2^{-128}$. The timings and the noise are measured and averaged on 500 runs. The runtimes are expressed in milliseconds.

p	Set	n_{short}	k	N	σ_{short}	σ_{long}	$\mathfrak{B}_{\mathsf{PBS}}$	ℓ_{PBS}	$\mathfrak{B}_{\mathsf{KS}}$	ℓ_{KS}	Runtime server	$\log_2(p_{\text{err}})$
1 bit	ORPHEUS (server)	660	3	512	$2^{49.7}$	$2^{27.1}$	2^{20}	1	2^3	4	7.0	-129
2 bits	ORPHEUS (server)	714	2	1024	$2^{48.3}$	$2^{13.8}$	2^{23}	1	2^3	4	10.1	-128
3 bits	ORPHEUS (server)	749	1	2048	$2^{47.4}$	$2^{13.8}$	2^{22}	1	2^3	5	12.9	-128
4 bits	ORPHEUS (server)	862	1	4096	$2^{45.6}$	$2^{2.0}$	2^{21}	1	2^3	5	27.3	-130
5 bits	ORPHEUS (server)	916	1	8192	$2^{43.1}$	$2^{2.0}$	2^{22}	1	2^4	4	58.2	-129
6 bits	ORPHEUS (server)	969	1	16384	$2^{41.7}$	$2^{2.0}$	2^{23}	1	2^4	5	125.6	-128

RÉSUMÉ

Dans cette thèse, nous étudions le chiffrement homomorphe, une technique cryptographique qui permet d'effectuer des calculs directement sur des données chiffrées, sans nécessiter de déchiffrement préalable. Ce domaine a connu un essor spectaculaire au cours des quinze dernières années, avec l'émergence de nombreux schémas de chiffrement de plus en plus performants. Néanmoins, les calculs homomorphes restent encore nettement plus coûteux que leurs équivalents classiques, ce qui freine leur adoption dans des applications concrètes.
Nous nous concentrons dans ce travail sur l'un des schémas les plus prometteurs : TFHE. Nous proposons de nouvelles techniques destinées à accélérer les calculs homomorphes pour différents cas d'usage. En exploitant un encodage innovant des messages, nous commençons par convevoir des algorithmes plus efficaces pour l'évaluation homomorphe de fonctions booléennes.
Dans un second temps, nous abordons le problème du transchiffrement, une approche visant à réduire la consommation de bande passante lors de la transmission de données chiffrées de manière homomorphe. Cela nécessite l'évaluation d'un algorithme de chiffrement symétrique dans le domaine homomorphe. Pour cela, et toujours en nous appuyant sur notre technique d'encodage, nous développons une implémentation homomorphe du chiffrement standard AES, plus rapide que celles de l'état de l'art, et contribuons à la conception d'un chiffrement par flot spécifiquement optimisé pour le transchiffrement.
Nous poursuivons avec une contribution qui étend les capacités de TFHE, en lui permettant de fonctionner sur des espaces de messages plus larges. Cette amélioration est possible grâce à un nouvel algorithme d'évaluation de table de correspondances dans ces espaces étendus.
Enfin, nous proposons une méthode conceptuellement simple et pratique pour générer des jeux de paramètres assurant sécurité, exactitude des calculs et efficacité, facilitant ainsi l'usage de TFHE dans les applications concrètes.

MOTS CLÉS

Mots-Clés ⋆ Cryptographie, Chiffrement complètement homomorphe, Calcul sécurisé

ABSTRACT

In this thesis, we study fully homomorphic encryption, a cryptographic technique that allows computations to be performed directly on encrypted data, without requiring prior decryption. This field has experienced remarkable growth over the past fifteen years, with the emergence of increasingly efficient encryption schemes. Nevertheless, homomorphic computations remain significantly more costly than their classical counterparts, which still hinders their adoption in practical applications.
In this work, we focus on one of the most promising schemes: TFHE. We propose new techniques aimed at accelerating homomorphic computations for various use cases. By leveraging an innovative message encoding strategy, we begin by designing more efficient algorithms for the homomorphic evaluation of Boolean functions.
Next, we address the problem of transciphering, an approach that seeks to reduce bandwidth consumption during the transmission of homomorphically encrypted data. This requires the evaluation of a symmetric encryption algorithm within the homomorphic domain. Still relying on our encoding technique, we develop a homomorphic implementation of the standard AES encryption scheme that outperforms state-of-the-art implementations, and present our contribution to the design of a stream cipher specifically optimized for transciphering.
We continue with a contribution that extends the capabilities of TFHE by enabling it to operate over larger message spaces. This improvement is made possible by a new algorithm for evaluating look-up tables in these extended spaces.
Finally, we propose a conceptually simple and practical method for generating parameter sets that ensure security, correctness, and efficiency, thereby facilitating the use of TFHE in real-world applications.

KEYWORDS

Keywords ⋆ Cryptography, Fully Homomorphic Encryption, Secure computation