EPFL

Thèse n° 8846

Multiparty Homomorphic Encryption: from Theory to Practice

Présentée le 29 septembre 2023

Faculté informatique et communications
Laboratoire d'ingénierie de sécurité et privacy
Programme doctoral en informatique et communications

pour l'obtention du grade de Docteur ès Sciences

par

Christian Vincent MOUCHET

Acceptée sur proposition du jury

Prof. O. N. A. Svensson, président du jury
Prof. C. González Troncoso, Prof. J.-P. Hubaux, directeurs de thèse
Prof. F. Kerschbaum, rapporteur
Prof. C. Orlandi, rapporteur
Prof. B. Ford, rapporteur

École polytechnique fédérale de Lausanne

2023

For there was a moment when anything was possible.
And there will be a moment when nothing is possible.
But in between we can create.
— Mohsin Hamid

Abstract

Multiparty homomorphic encryption (MHE) enables a group of parties to encrypt data in a way that (i) enables the evaluation of functions directly over its ciphertexts and (ii) enforces a joint cryptographic access-control over the underlying data. By extending traditional (single-party) homomorphic encryption (HE), MHE schemes support the design and deployment of highly efficient protocols for secure multiparty computation (MPC).

MPC protocols based on MHE have highly desirable properties: They generally require less communication than traditional MPC techniques and have a fully public transcript. Hence, most of their execution-related costs can be outsourced to an untrusted external party (such as a cloud server). Although promising in theory, MHE-based MPC solutions have not yet been implemented in any of the 30+ existing MPC frameworks, thus revealing a gap between theory and practice. This dissertation summarizes our work toward closing this gap, by proposing contributions to both sides.

On the theoretical side, we propose two MHE constructions that extend the new generation of HE schemes to the multiparty setting. Our first construction is an N-out-of-N-threshold MHE scheme that revisits the seminal lattice-based MHE construction by Asharov et al. (EUROCRYPT'12). Notably, we improve the efficiency of its setup phase, and we generalize its decryption procedure into a generalized *key-switching* operation that further enables re-encryption, conversion to secret-shares, and the interactive bootstrapping of its ciphertexts. Our second construction extends the first with fault-tolerance capabilities. This extension provides a T-out-of-N-threshold MHE scheme that stands as a compact and efficient alternative to the threshold scheme of Boneh et al. (CRYPTO'18), when synchronous communication can be assumed.

On the practical side, we propose the Lattigo library and the Helium system. Lattigo is an open-source Go package that implements the state-of-the-art HE schemes, along with their multiparty extensions. It is also the first maintained library to implement the bootstrapping procedure for approximate homomorphic encryption. Helium builds on top of Lattigo and provides the first end-to-end open-source implementation of an MHE-based MPC protocol. We exploit the theoretical properties of this protocol to propose a helper-assisted setting, where the parties delegate most of the protocol execution cost to an honest-but-curious third party (e.g., a cloud service). As a result, Helium is also the first open-source system to support MPC with sub-linear cost for the parties, without assuming non-collusion between the multiple delegate nodes.

Résumé

Le chiffrement homomorphe multipartite (MHE) permet à un groupe composé de multiple parties de chiffrer des données de manière à (i) permettre l'évaluation de fonctions directement sur les données chiffrées et (ii) d'appliquer un contrôle d'accès cryptographique conjoint sur les données sous-jacentes. En étendant le chiffrement homomorphe (HE) classique (à une seule partie), les systèmes de cryptage MHE permettent d'élaborer des protocoles très efficaces pour le calcul multipartite sécurisé (MPC) dans le modèle d'un adversaire passif contrôlant une majorité des parties.

Les protocoles de calcul multipartite basés sur le MHE présentent des propriétés intéressantes : ils nécessitent généralement moins de communication que les techniques de calcul multipartite traditionnelles et ils reposent entièrement sur l'échange de messages pouvant être publiquement divulgués. Par conséquent, la plupart des coûts liés à leur exécution peuvent délégués de façon sécurisée à une tierce partie externe (telle qu'un serveur dans le *cloud*). Bien qu'en théorie prometteuses, les solutions MPC basées sur le MHE n'ont encore été mises en œuvre dans aucune des 30+ solution logicielles MPC existantes, ce qui révèle un fossé entre la théorie et la pratique. Cette thèse résume notre travail visant à combler ce fossé, en proposant des contributions sur ses deux fronts.

Du côté théorique, nous proposons deux nouvelles constructions MHE qui étendent la dernière génération de systèmes de cryptage HE avec des fonctionnalités multipartites. Notre première construction est un système de cryptage à seuil N-sur-N qui revisite la construction basée sur les réseaux euclidiens introduite par Asharov et al. (EUROCRYPT'12). Nous améliorons notamment l'efficacité de sa phase d'initialisation et nous généralisons sa procédure de décryptage en une opération de *key-switching* généralisée qui permet en outre la re-encryption, la conversion en parts secrètes et le rafraichissement des textes chiffrés. Notre deuxième construction étend la première avec des capacités de tolérance aux pannes. Cette extension fournit un MHE à seuil T-sur-N qui constitue, dés lors qu'il est possible de supposer une communication synchrone, une alternative compacte et efficace à la proposition de Boneh et al. (CRYPTO'18).

Du côté pratique, nous proposons la bibliothèque logicielle Lattigo et le système Helium. Lattigo est un *package* Go open-source qui implémente les systèmes de cryptage HE de pointe ainsi que leurs extensions multipartites MHE. C'est également la première bibliothèque logicielle maintenue à mettre proposer une implémentation pour le *bootstrapping* du chiffrement homomorphe approximatif proposé par Cheon et al. (ASIACRYPT'17). Helium s'appuie sur Lattigo et fournit la première implémentation open-source d'un protocole MPC générique basé sur MHE. Nous nous appuyons sur les propriétés théoriques de ce protocole pour proposer un système dans lequel les parties délèguent la majeure partie du coût d'exécution du protocole à une tierce partie. Helium est ainsi le premier système open-source qui permet le MPC avec un coût pour les parties en dessous de linéaire (asymptotiquement, dans le nombre de parties), sans avoir à supposer la non-collusion entre de multiples tierces parties.

Acknowledgements

This dissertation concludes a chapter of my life, and there are so many people I want to thank. It feels natural to start from the beginning. But when did this all begin ?

During high school, Jean-Marc Falcoz was the first teacher to explain mathematics to me in a way that I could understand them. Around the same time, Boris Ischi taught me my first programming lectures, starting the flame of a passion that I was, still today, not yet able to put out. At the end of high school, Francois Lombard supervised my first ever research project on computer security. My fate as a computer scientist was sealed for good.

Curiosity is good but you need the opportunity to enter a PhD program and the appropriate guidance to complete it. I'm immensely grateful to Jean-Pierre Hubaux for providing both. Jean-Pierre, you gave me the freedom to explore and define my own project, and supported me in the most difficult moments. I wish you all the best for your retirement. I'm also immensely grateful to Carmela Troncoso for welcoming me (along with my colleagues) in the Security and Privacy Engineering Laboratory for my final year at EPFL. There are many bright academics, but fewer that inspire from their leadership and mentorship qualities. Carmela, you are one of them. Among many things, I learned from you that great supervisors do not only ask "*how is it going ?*", but also "*what can I do for you ?*".

During my time at EPFL, I had the chance to collaborate with extremely bright and talented people. Juan Troncoso Pastoriza was my first co-author and scientific mentor. Thank you Juan for taking the time to answer all my questions about lattices and for the countless hours spent brainstorming our cryptographic protocols. Later, I collaborated with Apostolos Pyrgelis, whose wisdom, knowledge and critical thinking was of precious support and guidance. For accepting to sit on my defense committee, I would like to thank Florian Kerschbaum, Claudio Orlandi, Ola Svensson and Bryan Ford. Bryan, thank you for providing insights on my work on several occasions throughout my PhD. I was extremely lucky to meet Henry Corrigan-Gibbs during his stay as a post-doc at EPFL. What started as a heated discussion about why two non-colluding cloud-servers is one cloud-server too many (I'm happy that we now agree...) evolved into a friendship (and more heated discussion, usually around an equally heated piece of Bagne cheese). Thank you Henry for all your advice and for greatly contributing to shape my view of research. Thanks to my friend Charles Bédard, for being such an amazing human being and a model of sane passion for research and science.

Sometimes, in a career or a life, some completely non-obvious decisions turn out to be pivotal in said career or life. One such moment occurred when a law student who was following some EPFL cryptography courses *for fun* wanted to work on my project of implementing a lattice-based FHE library. This is how I met Jean-Philippe Bossuat, and this is how we now maintain Lattigo, one of the most successful FHE libraries available today. Thank you Jean-Philippe, it was and still is quite a ride!

To implement cryptographic systems is a lot of work, and many of our practical results would never have seen the light of day without the great work of the many excellent Master and Bachelor

students who completed their semester projects with me: Johan Lanzrein, Björn Guðmundsson, Elliott Bertrand, Elia Anzuoni, Elie Daou, Clémence Altmeyerhenzien, Anas Ibrahim, Vincent Parodi, Hedi Sassi, Walid Ben Naceur, Adrien Laydu, Manon Michel, Adrian Cucos and Giovanni Torrisi. I also want to thank Linus Gasser and Christian Grigis from (respectively, formerly from) the EPFL Center for Digital Trust. Linus, you taught me a lot about the Go language and you were always available to chat about our prototypes.

I'm eternally grateful to Holly Cogliati-Bauereis for the time spent trying to improve my English writing. Holly, no AI can do what you do. I never had to worry much about administrative tasks, and this is thanks to the always friendly support of Patricia Hjelt, Angela Devenoge and Isabelle Coke.

I want to thank my colleagues in the PhD program; These extremely bright and kind people who made the everyday life of this difficult undertaking actually enjoyable. Starting with my friends and past members of the Laboratory for Data Security: David Froelicher, João Paul-Thierry Sá Sousa Sousa and Mickaël Misbach. David, thanks for being my *PhD Buddy* and a great *Coulage* companion. João, your stories are always worth listening to (at least the first few times you tell them). Mickaël, I miss your tartiflette. After the covid pandemic, the number of PhD students in the lab drastically went down to three. This is how I had the chance to get to know Sylvain Chatel and Sinem Sav much better. Sylvain, thanks for our fruitful and enriching collaborations, and all these great conference trips we had. I will definitely miss our coffee discussions and even your somewhat critical (not to say, cynic) views on basically everything, ranging from cafeteria logistics to the meaning of computer-science research. Sinem, thank you for being, with David, among the first researchers to employ the constructions presented in this thesis in concrete applications. I will remember the Italian "LDS on Tour" trip with you, delivering talks but also absorbing a substantial amount of pasta along the way. I also want to thank our friends from the DeDis Laboratory: Ceyhun Alp for the good laughs, Kirill Nikitin for the deep discussions about academia and the purpose of scientific research (and for this memorable time when you hosted me in NYC!) and Simone Colombo for the good times in Japan. Lastly, I want to thank the members of the SPRING laboratory, for the warm welcome they gave me during the final months of my PhD. I'm especially grateful to Bogdan Kulynych, Theresa Stadler, Mathilde Raynald, and Vera Rimmer for the insightful and timely discussions on research careers, to Kasra Edalatnejadkhamene for being always so positive and cheerful, to Dario Pasquini and Maria Grazia for showing me around in Rome, and to Wouter Lueks for his always spot-on inputs on my work and presentations.

Finalement, j'aimerais remercier ma famille. Leur amour et soutien inconditionnels ont été la base sur laquelle il m'a été possible de construire tout le reste. Merci à mes grands-parents de m'avoir laissé jouer avec leur ordinateur quand j'étais petit. Merci à ma mère Corinne, qui a fait tant de sacrifices, sans jamais le mentionner, pour offrir à ma sœur et moi nos études et les conditions idéales pour leur réussite. Merci à mon père Claude, dont j'admire la sagesse et qui sera toujours un modèle pour moi.

Merci à Emilie, qui me remplit de joie chaque jour et, je l'espère, pour encore beaucoup de jours.

Lausanne, August 5, 2023 C.M.

Table of Contents

Introduction

Historically, cryptography has played a crucial role in ensuring the confidentiality and authenticity of information, during its transmission between entities. Already at the time of its early military applications, cryptography wielded significant political and societal influence, with the outcome of wars hinging on the acquisition of timely and accurate information. Today, as the Internet connects two-thirds of the world population [1] and our digital footprint carries an almost complete picture of our daily life, the stakes of implementing cryptographically secure systems have never been higher. Indeed, the ability for individual members of a society to ensure the confidentiality of their communications and information is an indisputable pre-condition to guaranteeing the fundamental rights to privacy and freedom of speech.

Global interconnections also provide an incredible opportunity: the ability, not only to communicate, but also to *compute* over the Internet. From the early *Web 2.0*, which enables users to contribute content to websites, to *cloud computing*, the practice of delegating computations to externally managed machines, computation has become a new paradigm for the Internet. In this paradigm, users are no longer merely senders and recipients of messages, they are also actors in computations.

The Internet provides a powerful *logical* computation platform. But, it requires its users to trust the machine where the *physical* computations occur and, by extension, the entity that owns and manages it. From this stems the question that initiated a significant extension of focus in modern cryptographic research: *Can cryptography protect the confidentiality and integrity of computation performed within an untrusted infrastructure?*

Modern Cryptographic Research

Over the last few decades, this question motivated the study of *secure computation* in untrusted environments. From the *correctness* perspective, for example, the field of *verifiable computing* [BFLS91; Mic00; GGP10; GKR15] introduced techniques for a computationally weak party to be able to delegate a computation to an untrusted but computationally powerful party while being able to verify that the returned result is computed correctly. From the *confidentiality* perspective, which is the focus of this thesis, the field of *homomorphic encryption* (HE) provides techniques for encrypting data in a way that still enables computation to be performed, without requiring decryption. HE therefore enables computationally weak parties to delegate both the storage and the processing of sensitive data to an untrusted third party, while ensuring the confidentiality of these data.

Beyond delegated computation, the field of *secure multiparty computation* addresses the problem of securing distributed computations among multiple parties, when these parties' inputs need to be kept confidential. From data-driven research by large medical institutions to a group of

[1] `https://www.internetworldstats.com/stats.htm`

friends willing to privately synchronize their calendars, the setting of secure multiparty computation encompasses a large range of real-world problems. Interestingly, achieving secure multiparty computation *inherently* requires computation to be performed in *some* untrusted environment: be it an assisting third party or the parties themselves, as a distributed system. From a practical standpoint, this requirement is much stronger than for simple (single-party) delegated computations, where clients might still choose not to delegate. For example, if the cost of the cryptographically secure outsourced computation outweighs the cost of the local one (e.g., the cost of acquiring and managing more powerful client-side hardware). In contrast, secure multiparty computation cannot be achieved as a cost allocation trade-off. Hence, it is not surprising that finding efficient protocols to perform secure multiparty computation was, and still is, a significant focus of cryptographic research.

Multiparty Homomorphic Encryption

Multiparty homomorphic encryption (MHE) enables computation over encrypted data, while enforcing *joint* cryptographic access-control over the underlying data (we will further define this primitive in Chapter 1). By generalizing traditional single-party homomorphic encryption to multiple data providers, MHE techniques constitute a promising family of solutions for performing secure multiparty computation (MPC).

More specifically, MHE techniques can be used to construct efficient MPC protocols, commonly referred to as *two-round MPC*: In the first round, the parties encrypt their sensitive input data with the MHE scheme, and the function is homomorphically evaluated over the ciphertexts, either by the parties themselves or by an assisting third-party. The output of this round is the computation result, encrypted under the MHE scheme. In the second round, the parties obtain the computation output by engaging in a multiparty decryption protocol. These MHE-based MPC protocols are characterized by their low communication complexity, as well as their amenability to the paradigms of cloud-computing [AJLT+12; MTBH21] such as service-based infrastructures and light-client/powerful-server architectures.

Several generations of MHE schemes have been proposed over the years, generally following the advances of single-party HE constructions. Among these multiparty schemes, *threshold* schemes [Des93] have been demonstrated to be particularly efficient due to their compactness. Threshold-HE techniques have also been long-known for reducing the communication complexity of MPC protocols [FH96; CDN01; DPSZ12; CDES+18], yet these techniques were initially constrained by the lack of an HE scheme that supports two arithmetic operations hence arbitrary circuits. This changed with the introduction of the first fully homomorphic encryption scheme, by Gentry in 2009 [Gen09], which was quickly followed by the proposition of concrete two-round MPC protocol construction [AJLT+12] based on MHE by Asharov et al.

Made possible by advances in the cryptanalysis of lattice-based cryptography, these constructions are based on the *learning-with-error* (LWE) assumption [Reg05]. Although promising in theory, these constructions have so far remained theoretical. Despite the existence of more than 30 frameworks exploiting various other approaches to MPC [Rot17], there is no implementation yet of an MHE-based MPC system. This has severely limited security researchers in using the MHE techniques in practice and demonstrates a gap between theory and practice. In this dissertation, we close this gap, with contributions to both the theoretical and practical sides of MHE research.

Contributions

On the theoretical side, we propose two MHE constructions that extend the current generation of HE schemes, based on the *ring learning-with-error* (RLWE) assumption [LPR10], to the multiparty setting. On the practical side, we propose the Lattigo library, which implements our MHE construction, and the Helium system, which implements the MHE-based MPC protocol. We now provide a summary of these contributions, and relate them to the chapters of this dissertation in which they are discussed.

Chapter 2: A Multiparty Homomorphic Encryption Scheme Our first construction is an N-out-of-N-threshold HE scheme that revisits the LWE construction by Asharov et al. [AJLT+12]. In addition to adapting their construction to RLWE, we propose several improvements that make it more practical. Notably, we improve the efficiency of its setup phase and propose new procedures that facilitate the instantiation of the scheme as an MPC protocol. We achieve this by generalizing the scheme's decryption operation into a *key-switching* operation that enables the ciphertexts to be re-encrypted, converted to secret-shares (e.g., to be processed by other MPC approaches), and to be refreshed without the need for a costly bootstrapping operation. We implement our solution in the Lattigo library (presented in Chapter 4) and evaluate the instantiation of our MHE scheme into an MPC solution for several circuits and system models.

This chapter is based on our work on [MTBH21]; it was presented at the 21st Privacy Enhancing Technologies Symposium (PETS'21).

Chapter 3: A Fault-Tolerant Multiparty Homomorphic Encryption Scheme Our second construction extends our N-out-of-N-threshold MHE scheme into a t-out-of-N-threshold scheme that enables a fault-tolerant MPC instantiation. This scheme depends on synchronous communication hence requires more assumptions than the state-of-the-art construction by Boneh et al. [BGGJ+18]. Yet, we show that our construction has several practical advantages. Notably, our scheme requires only a constant size state per party and does not affect the size of the ciphertexts, whereas [BGGJ+18] requires at least a $\Omega(N^4)$ state or requires non-compact ciphertexts. Moreover, our scheme does not require a trusted dealer and is considerably simpler, which makes it easy to implement: Its implementation in Lattigo requires less than a hundred lines of code on top of our N-out-of-N-threshold scheme implementation.

The chapter is based on our work on [MBH23]; It was published in issue number 36 of the IACR Journal of Cryptology.

Chapter 4: Lattigo: a Multiparty Homomorphic Encryption Library in Go Our first practical contribution is Lattigo, an open-source Go package that implements the state-of-the art HE schemes, along with their MHE extensions proposed in Chapters 2 and 3. As such, Lattigo is the first HE library to emphasize the multiparty setting for HE, and to implement all the functionalities required by the MHE-based MPC protocol. Lattigo is also the first open-source library to implement a bootstrapping procedure for the approximate homomorphic encryption scheme of Cheon et al., CKKS [CKKS17], in its optimized variant based on residue number system (RNS) [CHKK+19].

This chapter is based on our work on the Lattigo library that is a collaboration with Jean-Philippe Bossuat and Juan Troncoso-Pastoriza. Most of the chapter content was written for the present thesis, with some parts from [MBTH20].

Chapter 5: Helium: an MHE-based MPC Framework Our second practical contribution, Helium, builds on top of Lattigo and provides the first end-to-end open-source implementation of an MHE-based MPC protocol. In designing this system, we fill several gaps left unaddressed in the theoretical literature on MHE (including our own), such as how to securely implement the MHE-based MPC protocol with weak computing-resources and/or churning participants. We utilize the properties of MHE-based MPC to propose a helper-assisted setting, where the parties delegate most of the protocol execution costs to an honest-but-curious third party. Therefore, Helium is also the first implemented MPC system to support, without assuming non-collusion between the multiple delegate nodes, sub-linear-cost MPC for the input parties.

This chapter is based on our most recent work on the HElium system, for which a submission based on this chapter is currently under preparation.

Other Contributions

In addition to the contributions presented in this thesis, we have made several contribution related to homomorphic encryption in general. We now briefly describe these contributions.

Efficient Bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys We propose an efficient bootstrapping approach for the approximate homomorphic-encryption scheme of Cheon et al., CKKS [CKKS17].

The CKKS scheme is a *leveled* HE scheme that is capable of homomorphically evaluating arbitrary polynomial functions over encrypted complex-number vectors. Although the family of leveled cryptosystems enables only a finite multiplicative depth, with each multiplication *consuming* one level, the CKKS scheme enables the homomorphic re-encryption of an exhausted ciphertext into an almost *fresh* one. This capability, commonly called *bootstrapping*, theoretically enables the evaluation of arbitrary-depth circuits. In practice, however, the bootstrapping procedure for CKKS is approximate, and its precision and performance determine the actual (practical) maximum depth of a circuit. Consequently, for practical reasons, all previous CKKS bootstrapping approaches [CHKK+18; CCS19a; HK20] had to rely on a special secret-key structure known as *sparse secret-keys*, to reduce the depth of their circuit representation, and none of them has proposed parameters with an equivalent security of at least 128 bits under the recent attacks on sparse RLWE secrets [CHHS19; SC19].

To alleviate these issues, we propose several improvements to the CKKS bootstrapping in [BMTH21]. Compared to the previously proposed procedures, our proposed bootstrapping procedure is more precise, more efficient (in terms of CPU cost and number of consumed levels), and has a lower failure probability. As a result, and unlike the previous approaches, it does not require the use of sparse secret-keys and can be instantiated with 128-bit-secure parameters.

We achieve this efficiency and precision by introducing four novel contributions: (i) We propose a generic algorithm for homomorphic polynomial-evaluation that takes into account the specific semantic of the CKKS scheme, such as its approximate rescaling and the fact that each ciphertext-cihpertext multiplication consumes a level. Notably, we show that our homomorphic polynomial evaluation has optimal level-consumption. (ii) We propose a novel approach to computing homomorphic rotations over encrypted vector coefficients, and we propose a new technique, which we call *double hoisting*, for computing linear transformations as matrix-vector products. (iii) We propose a systematic approach to parameterize the bootstrapping, including a precise way to assess its failure probability. (iv) We implemented our improvements and bootstrapping procedure in the open-source Lattigo library. Note that (i) and (ii) are generic

encrypted-arithmetic algorithms, hence are of independent interest in HE research (i.e., beyond the bootstrapping procedure itself).

These results were presented at the 2021 Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT'21) [BMTH21].

PELTA–Shielding Multiparty-FHE against Malicious Adversaries Another contribution, more directly related to MHE, is the PELTA construction [CMSP+23]. The purpose of PELTA is to extend MHE to the active adversary model, by requiring parties to provide (zero-knowledge) proof of correct execution of the MHE local operations. Although such an extension is proposed by Asharov et al., the authors did not specify the concrete construction of such proofs. Our contribution to this result is a systematization of the current MHE schemes, under which a generic method for proving their correct execution can be proposed. The actual construction being the work of our collaborators is not discussed in this dissertation.

This result will be presented at the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23) [CMSP+23].

Impact

Contemporary literature has now successfully employed MHE schemes to build practical systems, for domain-specific tasks: for example, in distributed analytics [FTRC+21; CSSM+22; YZWL+22], and federated machine-learning [CDKS19; SPTF+21; ATP21; SBTC+22; XHXZ+22; SDPB+22; XLGZ+23; ISPB+23]. Most of these works employed schemes and/or implementations that are the results of this thesis.

In 2021, the MHE scheme and its implementation were used as the core technical building block of a commercial product developed by the start-up *Tune Insight SA*[2]. Today, this EPFL spin-off develops prototypes for early adopters of MHE-based MPC solutions in domains where data-driven decision-making can be improved by secure collaborations on sensitive data. Notably, the company is active in healthcare, in cyber-threat intelligence, and in insurance risk-analysis.

On the implementation side, Lattigo is now a well-established contribution to the HE-software landscape. According to Google Scholar[3], the library was mentioned in more than 160 scientific works between 2020 and the summer of 2023, several of which have used its HE and MHE implementations to build system research prototypes [BSA21; CPTH21; FTRC+21; KSJH21; ICDÖ22; TMBM+22; PPV22; CP23a; ERLT23; KG23; FCES+23]. Moreover, the standalone math layer of Lattigo was used by cryptography researchers to implement and evaluate new primitives [CHKL+21; HKLL+22; KKLS+22; KLKS+22; LLKK+22; GHHJ22; ACYJ+23; CP23b; KLSS23], such as multi-key and multi-group MHE schemes.In February 2022, the maintenance of Lattigo was transferred to Tune Insight, and the library is still being actively developed.

Organization

This dissertation is organized as follows:

– In Chapter 1, we present the definitions and constructions relevant to MHE schemes. We recall the necessary notions and introduce our systematization of (M)HE constructions.

– In Chapters 2 to 5, we present our main aforementioned contributions.

– In Chapter 6, we conclude this dissertation and discusses open problems, as well as future research directions.

[2] `https://tuneinsight.com`
[3] `https://scholar.google.com`

Chapter 1

Definitions and Constructions

Chapter Content

In this chapter, we introduce the notation, definitions and constructions relevant to this dissertation. The definition of MHE is taken from our work in [MTBH21]. We present the current generation of arithmetic HE schemes constructions based on RLWE under a novel abstraction: the *core & front-end* paradigm. This abstraction will enable the generalization of our constructions proposed in Chapters 2 and 3 across all these schemes, and will provide the main rational for the architecture of the Lattigo library.

1.1 Terminology

Our MHE constructions rely on both interactive and non-interactive cryptographic functionalities. For the sake of clarity, it is often useful to distinguish between the two cases in order to provide the correct intuition about the constructions. Hence, we use the following terminology to refer to cryptographic functions:

Operation refers to a cryptographic function that is performed as part of a larger process. This term is used to describe simple actions for which it is unspecified whether they require interaction with other parties or not.

Algorithm refers to a non-interactive operation, meaning that it does not involve any form of communication between different parties. It can be executed locally without the need for network connectivity or interaction with other devices.

Protocol refers to an interactive operation, meaning that, unlike an algorithm, it involves communication between different parties. Protocols usually involve the execution of algorithms composed with message exchanges.

Procedure describes processes that involve multiple operations to accomplish a specific task. Hence, procedures can be either interactive or non-interactive, depending on their underlying operations.

Using these terms, we describe multiparty cryptographic systems. For example, we refer to the "key-generation operation" as a specific action or function that is part of a larger cryptographic process. We also refer to the "key-generation algorithm" as a set of instructions that specify how the key-generation operation should be performed, without any communication or interaction between different parties. If the key-generation process involves communication between different parties or devices, we refer to it as a "key-generation protocol". Finally, if a process such as a cryptographic setup involves several operations, such as multiple key generation protocols, we refer to it as a "setup procedure".

1.2 Notation

We use regular letters for integers and polynomials, and boldface letters for vectors of integers and of polynomials. $\boldsymbol{a}^T$ denotes the transpose of a vector $\boldsymbol{a}$. We denote $[\cdot]_q$ the reduction of an integer modulo q, and $\lceil \cdot \rfloor$, $\lfloor \cdot \rfloor$, $\lfloor \cdot \rceil$ the rounding to the next, previous, and nearest integer, respectively. When applied to polynomials, these operations are performed coefficient-wise. When considering operations between elements of the same quotient ring, modular reductions are omitted. Given a probability distribution α over a ring R, $a \leftarrow \alpha$ denotes the sampling of an element $a \in R$ according to χ, and $a \leftarrow R$ implicitly denotes uniform sampling in R. We denote $\mathbf{a} \leftarrow \chi^l$ the sampling of an l-dimensional vector $\mathbf{a}$ according to a coefficient distribution χ (each coefficient being independently sampled). For a polynomial a, we denote its infinity norm (i.e., its largest

coefficient in absolute value) by $\|a\|$. Finally, $\mathsf{poly}[x]$ denotes a polynomial function in x and $\mathsf{negl}(x)$ a negligible function in x.

1.3 Definitions

We provide definitions for the primitives and notions relevant to this work. We consider an abstract security parameter λ and require that an adversary's advantage in attacking the schemes be a negligible function in λ. HE schemes also require proper parameterization in order to support the evaluation of the desired circuits. We model this dependency by introducing an abstract *homomorphic capacity* parameter κ and require that the probability of incorrect decryption be a negligible function in κ.

1.3.1 Homomorphic Encryption

Let $\mathcal{M}$ be a plaintext space with arithmetic structure, a homomorphic encryption scheme over $\mathcal{M}$ is a tuple of algorithms $\mathsf{HE} = (\mathsf{Setup}, \mathsf{SecKeyGen}, \mathsf{EncKeyGen}, \mathsf{EvalKeyGen}, \mathsf{Encrypt}, \mathsf{Eval}, \mathsf{Decrypt})$ with the following syntax:

- **Setup** $pp \leftarrow \mathsf{HE.Setup}(\lambda, \kappa)$:
  Given the security and the homomorphic-capacity parameters, HE.Setup outputs a public parameterization (passed implicitly to the other procedures).
- **Secret-key generation** $\mathsf{sk} \leftarrow \mathsf{HE.SecKeyGen}()$:
  HE.SecKeyGen generates and outputs a secret key sk.
- **Public-keys generation** $\mathsf{pk} \leftarrow \mathsf{HE.EncKeyGen}(\mathsf{sk})$ and $\mathsf{evk} \leftarrow \mathsf{HE.EvalKeyGen}(\mathsf{sk})$:
  From the secret-key, HE.EncKeyGen outputs the public encryption key pk and HE.EvalKeyGen outputs the evaluation-key evk.
- **Encryption** $\mathsf{ct} \leftarrow \mathsf{HE.Encrypt}(\mathsf{pk}, m)$:
  From a plaintext $m \in \mathcal{M}$, HE.Encrypt outputs a ciphertext ct encrypting m.
- **Evaluation** $\mathsf{ct}_{res} \leftarrow \mathsf{HE.Eval}(f, \mathsf{evk}, \mathsf{ct}_1, ..., \mathsf{ct}_I)$:
  From an arithmetic function $f : \mathcal{M}^I \rightarrow \mathcal{M}$, the public evaluation key evk and a I-tuple of ciphertexts $\mathsf{ct}_1, ..., \mathsf{ct}_I$ such that $\mathsf{HE.Decrypt}(\mathsf{ct}_i) = m_i$, HE.Eval outputs a ciphertext ct_{res} such that $\mathsf{HE.Decrypt}(\mathsf{ct}_{res}) = m_{res} = f(m_1, ..., m_I)$.
- **Decryption** $m' \leftarrow \mathsf{HE.Decrypt}(\mathsf{sk}, \mathsf{ct})$:
  From a ciphertext ct encrypting m and the secret key sk, HE.Decrypt outputs m'.

Moreover, scheme HE satisfies the following properties:

1. **Semantic Security**: for pp a public parameterization, any two messages $m_0, m_1 \in \mathcal{M}$, $\mathsf{sk} \leftarrow \mathsf{HE.SecKeyGen}$ and $\mathsf{pk} \leftarrow \mathsf{HE.EncKeyGen}(\mathsf{sk})$, then, for every probabilistic polynomial-time adversary $\mathcal{A}$ it holds that

$$\Pr[\mathcal{A}(\mathsf{HE.Encrypt}(\mathsf{pk}, m_b)) = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

2. **Correctness**: for any arithmetic function $f : \mathcal{M}^I \rightarrow \mathcal{M}$ and input messages $m_1, ..., m_I$, there exists a public parameterization pp such that, for $\mathsf{sk} \leftarrow \mathsf{HE.SecKeyGen}$, $\mathsf{pk} \leftarrow \mathsf{HE.EncKeyGen}(\mathsf{sk})$, $\mathsf{evk} \leftarrow \mathsf{HE.EvalKeyGen}(\mathsf{sk})$ and $\mathsf{ct}_i \leftarrow \mathsf{HE.Encrypt}(\mathsf{pk}, m_i)$,

$$\Pr[\mathsf{HE.Decrypt}(\mathsf{sk}, \mathsf{HE.Eval}(f, \mathsf{evk}, \mathsf{ct}_1, ..., \mathsf{ct}_I)) \neq f(m_1, ..., m_I)] \leq \mathsf{negl}(\kappa).$$

1.3.2 Multiparty Homomorphic Encryption

We now provide the definition of multiparty homomorphic encryption that we use in this work. Note that we tailor this definition for the *threshold* family of MHE schemes, as our proposed construction belongs to this family. The other family of MHE schemes, *multi-key* schemes [LTV12; CCS19b], will only be briefly discussed in Chapter 2, where we compare our MHE construction to those schemes. For consistency throughout this work, we use the more general *multiparty homomorphic encryption* term for referring to *threshold homomorphic encryption*.

Multiparty schemes rely on distributed state among the parties, such as secret keys and private plaintext inputs. In the definition below and throughout this work, we will use $\{x_i\}_{P_i \in \mathcal{P}}$ to denote the set of all values x_i held, respectively, by each party P_i in a given set of party $\mathcal{P}$.

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be a set of N parties, $T \leq N$ a *threshold* parameter, and $\mathcal{M}$ be a plaintext space with arithmetic structure. A *multiparty homomorphic encryption*-scheme over $\mathcal{P}$ and $\mathcal{M}$ is a tuple $\mathsf{MHE} = (\mathsf{Setup}, \Pi_{\mathsf{SecKeyGen}}, \Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \mathsf{Encrypt}, \mathsf{Eval}, \Pi_{\mathsf{Decrypt}})$ of algorithms and multiparty protocols with the following syntax:

- **Setup** $pp \leftarrow \mathsf{MHE.Setup}(\lambda, \kappa, \mathcal{P}, T)$:
  Takes the security and homomorphic capacity parameters and outputs a public parameterization. pp is an implicit argument to the other procedures.

- **Secret-key generation** $\mathsf{sk}_i \leftarrow \mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}()$:
  All parties in $\mathcal{P}$ take part in the $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ protocol that privately outputs to each party its own secret-key.

- **Public-keys generation** $\mathsf{pk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$, $\mathsf{evk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EvalKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$:
  Any sub-group $\mathcal{P}' \subseteq \mathcal{P}$ with $|\mathcal{P}'| \geq T$ holding keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ executes the public encryption- and evaluation-key generation protocols and outputs $(\mathsf{pk}, \mathsf{evk})$.

- **Encryption** $\mathsf{ct} \leftarrow \mathsf{MHE.Encrypt}(m, \mathsf{pk})$:
  Given the public-key pk, and a plaintext $m \in \mathcal{M}$, outputs a ciphertext encrypting m.

- **Evaluation** $\mathsf{ct}_{res} \leftarrow \mathsf{MHE.Eval}(f, \mathsf{evk}, \mathsf{ct}_1, ..., \mathsf{ct}_I)$:
  Given a function $f : \mathcal{M}^I \rightarrow \mathcal{M}$, the public key pk and a I-tuple of ciphertexts encrypting $(m_1, ..., m_I) \in \mathcal{M}^I$, outputs a result ciphertext encrypting $m_{res} = f(m_1, ..., m_I)$.

- **Decryption** $m \leftarrow \mathsf{MHE}.\Pi_{\mathsf{Decrypt}}(\mathsf{ct}, \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$:
  Given a ciphertext ct encrypting m, any sub-group of parties $\mathcal{P}' \subseteq \mathcal{P}$ with $|\mathcal{P}'| \geq T$ holding key $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ executes the decryption protocol and outputs m.

Moreover, scheme MHE satisfies the following properties:

1. **Semantic Security**: for a public parameterization pp, any two messages $m_0, m_1 \in \mathcal{M}$, for any setup subset $\mathcal{P}_{setup} \subseteq \mathcal{P}$ with $|\mathcal{P}_{setup}| \geq T$, $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}} \leftarrow \mathsf{MHE.}\Pi_{\mathsf{SecKeyGen}}$ and $\mathsf{pk} \leftarrow \mathsf{MHE.}\Pi_{\mathsf{EncKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}_{setup}})$, then, for every probabilistic polynomial-time adversary $\mathcal{A}$, any adversarial subset $\mathcal{P}_{\mathcal{A}} \subseteq \mathcal{P}$ with $|\mathcal{P}_{\mathcal{A}}| < T$, it holds that

$$\Pr[\mathcal{A}(\mathsf{MHE.Encrypt}(\mathsf{pk}, m_b), \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}_{\mathcal{A}}}) = b] \leq \frac{1}{2} + \mathsf{negl}(\lambda).$$

2. **Correctness**: for any arithmetic function $f : \mathcal{M}^I \to \mathcal{M}$ and input messages $m_1, ..., m_I$, there exists a public parameterization pp such that, for $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}} \leftarrow \mathsf{MHE.}\Pi_{\mathsf{SecKeyGen}}$ and any two subsets of parties $\mathcal{P}_{setup}, \mathcal{P}_{dec} \subseteq \mathcal{P}$ of size at least T, $\mathsf{pk} \leftarrow \mathsf{MHE.}\Pi_{\mathsf{EncKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}_{setup}})$, $\mathsf{evk} \leftarrow \mathsf{MHE.}\Pi_{\mathsf{EvalKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}_{setup}})$ and $\mathsf{ct}_i = \mathsf{MHE.Encrypt}(\mathsf{pk}, m_i)$,

$$\Pr[\mathsf{MHE.}\Pi_{\mathsf{Decrypt}}(\mathsf{MHE.Eval}(f, \mathsf{evk}, \mathsf{ct}_1, ..., \mathsf{ct}_I), \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}_{dec}}) \neq f(m_1, ..., m_I)] \leq \mathsf{negl}(\kappa).$$

1.3.3 Real-Ideal Scheme Formulation

It is common to formulate multiparty functionalities as abstract functionalities in an ideal environment in which a single party acts on behalf of the real-world parties. This applies to HE and MHE schemes as previously defined: We can model the functionality of an MHE scheme as an ideal environment that instantiates a single-party HE scheme. The semantic security property of the MHE definition requires that no single party should be able to decrypt messages alone. Hence, in the ideal world, the environment holds the secret-key to the HE scheme and responds to key-generation and decryption queries by calling the $\mathsf{EncKeyGen}$, $\mathsf{EvalKeyGen}$ and $\mathsf{Decrypt}$ procedures of the single-party HE scheme. In the real world, the parties must emulate the ideal-world secret key with their local secrets and emulate the secret-key dependent operations with the $\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{EvalKeyGen}}$ and Π_{Decrypt} multiparty protocols.

This abstraction is particularly useful in our context, because single-party HE schemes have many operations that do not require the secret-key, hence can be implemented as local operations in the multiparty scheme. In our exposition, we refer to the ideal-world scheme as *the ideal scheme*. By extension, we refer to the ideal scheme's secret-key as the *ideal secret key*. As this key concretely exists only through interactions between the parties, it is often convenient to model it as a function of the parties' individual secrets that we denote $\mathcal{S}(\mathsf{sk}_1, ..., \mathsf{sk}_N)$, where sk_i denotes the secret owned by party P_i.

1.3.4 MHE-based MPC

Multiparty homomorphic encryption techniques can be used to construct efficient secure multiparty computation protocols, commonly referred to as *two-round MPC* [AJLT+12]. The construction of such protocols for the passive adversary model is fairly natural from the MHE definition. Hence we defer the detailed description to Section 2.4, where we instantiate it with our MHE constructions and, in this section, we provide only a high-level description.

In the *Setup* phase, the parties execute the $\mathsf{MHE.}\Pi_{\mathsf{SecKeyGen}}$ protocol to generate their shares of an ideal secret-key. Then, they execute the $\mathsf{MHE.}\Pi_{\mathsf{EncKeyGen}}$ and $\mathsf{MHE.}\Pi_{\mathsf{EvalKeyGen}}$ protocols to obtain the corresponding public encryption and evaluation keys. The input-dependent *computation* phase consists of three steps: *Input*, *Evaluation*, and *Output*. During the *Input* step (round one), the parties use the $\mathsf{MHE.Encrypt}$ algorithm to encrypt their inputs and disclose the resulting ciphertexts to the other parties. Then, the desired computation is carried out by using

the $\mathsf{MHE.Eval}$ algorithm. Finally, the parties execute the $\mathsf{MHE.}\Pi_{\mathsf{Decrypt}}$ protocol to decrypt the result ciphertext(s) in the *Output* step (round two).

1.3.5 Ring Learning With Error

We recall the *ring learning-with-error* (RLWE) cryptographic assumption that we use throughout this work. It is a variant over rings of the *learning with error* (LWE) assumption introduced by Regev [Reg05]. The RLWE assumption is defined as two computational problems about a distribution over polynomial rings [LPR10]. Note that we particularize these definitions for the ring structure and parameters that we rely on in this work.

The RLWE Distribution Let $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ be the cyclotomic ring of degree-$n-1$ polynomials with coefficients modulo q and let $\mathsf{Err}(R_q)$ be an error distribution over R_q where the coefficients are sampled from a bounded discrete Gaussian distribution of small variance σ^2 and small bound B (w.r.t. q). For a given secret $s \in R_q$, the *ring learning with error distribution* $\mathsf{RLWE}_{s,\mathsf{Err}}$ is sampled by sampling $a \leftarrow R_q$ and $e \leftarrow \mathsf{Err}$, and outputting $(a, sa + e)$.

The Search-RLWE Problem Given $m = \mathsf{poly}[n]$ independent samples $(a_i, b_i) \leftarrow \mathsf{RLWE}_{s,\mathsf{Err}}$ from the RLWE distribution for secret s, find s.

The Decision-RLWE Problem Given $m = \mathsf{poly}[n]$ independent samples (a_i, b_i) where samples are either all sampled from $\mathsf{RLWE}_{s,\mathsf{Err}}$ or all sampled uniformly from R_q^2, distinguish between the two cases.

Hardness Assumption Informally, both problems are assumed to be computationally hard enough to be used for cryptography. For the presented parameterization (in which R_q is a cyclotomic ring), it has been demonstrated that the search-RLWE problem can be reduced to the *approximate shortest vector problem* (a-SVP) on ideal lattices and to the decision-RLWE problem [LPR10].

1.4 Constructions

We now recall the cryptographic constructions relevant to this work. We first present the construction of HE schemes from the RLWE assumption [LPR10]. Although HE schemes are typically presented as single monolithic schemes, such as the BFV [FV12], BGV [BGV14], and CKKS [CKKS17] schemes, they share common functionalities and sub-procedures. Therefore, we proceed in two steps: We first introduce a *core* RLWE-based construction for which the correctness property is only approximate and on top of which the BFV, BGV, and CKKS schemes can be constructed. As such, we see the current HE schemes as *front-end schemes*, each one enabling a different kind of homomorphic arithmetic (i.e., a different plaintext space) on top of the core, approximate scheme. For an example of such a front-end scheme, we introduce the BFV [FV12] encryption scheme that enables exact integer arithmetic. Finally, we discuss how this separation between the core and front-end schemes provides a useful systematization in both theory and practice.

1.4.1 Core RLWE-based Homomorphic Encryption

We detail the core RLWE construction as scheme HE (Scheme 1). Recall that this scheme represents the common functionalities of the BFV [FV12], BGV [BGV14], and CKKS [CKKS17] schemes. We introduce it directly in its more efficient variant based on a residue number system (RNS). Due to its practicality, this variant is implemented in most of the current lattice-based cryptographic libraries[1][2][3].

Let the ciphertext space be the polynomial quotient ring $R_q = \mathbb{Z}_q[X]/(X^n + 1)$, where the polynomial degree n is a power of two and where the polynomial-coefficient modulus q is a product of L different primes $q_1, ..., q_L$. Hence, we can use the isomorphism $R_q \cong R_{q_1} \times ... \times R_{q_L}$ provided by the Chinese remainder theorem (CRT) to perform the operations in the residue rings, without resorting to arbitrary-precision integer arithmetic. Moreover, we choose each q_i such that $q_i \equiv 1 \mod 2n$, which enables the representation of elements of R_{q_i} (i.e., polynomials) under the number-theoretic transform domain (NTT) under which both ring operations are performed coefficient-wise. We use $[-\frac{q}{2}, \frac{q}{2})$ as the set of representatives for the congruence classes modulo q. Unless otherwise stated, we consider the arithmetic in R_q and polynomial reductions are omitted in the notation. Let $\mathsf{Key}(R_q)$ be a secret-key distribution over R_q for which the coefficients are sampled uniformly in $\{-1, 0, 1\} \mod q$, let $\mathsf{Err}(R_q)$ be an error distribution where the coefficients are sampled from a bounded discrete Gaussian distribution of small variance σ^2 and small bound B (w.r.t. q).

Scheme 1 is an approximate HE scheme over the message space R_q: its ciphertexts contain an error which increases during the homomorphic operations, and its decryption procedure outputs approximations of the R_q messages. Some operations require multiplying ciphertexts with large R_q elements (e.g., HE.Relinearize involves multiplying the ciphertexts elements by the elements of rlk), which would result in the error blowing up. To mitigate such an effect on the error, the scheme relies on a prior decomposition of R_q elements into an auxiliary basis to reduce their norm. The decomposition basis is a parameter of the scheme that we denote $\mathbf{w}$, and its dimension is denoted by l. For an element $a \in R_q$, we denote $a^{(i)}$ its i-th coefficient (or coordinate) in the decomposed basis. Hence, we have $a = \sum_{i=0}^{l} w^{(i)} a^{(i)}$. A common choice is to use a base-w power basis for some integer $w < q$, i.e., $\mathbf{w} = (w^0, w^1, ..., w^{l-1})^T$ and $l = \lceil \log_w(q) \rceil$. Another option is to use the natural linear decomposition basis provided by the RNS, or even a hybrid between the two approaches.

Homomorphic Evaluation The HE.Eval procedure of the core scheme corresponds to the tuple (HE.Add, HE.Mul, HE.Relinearize) and the target arithmetic function must be expressed in terms of these operations. The HE.Mul operation outputs a ciphertext that consists of three R_q elements that can be seen as a *degree two* ciphertext. This higher-degree ciphertext can be further operated on and decrypted. Yet it is often desirable to reduce this degree back to one by using the HE.Relinearize operation. This operation requires the generation of a specific public key referred to as the *relinearization key* (rlk) that becomes part of HE.EvalKeyGen.

Likewise, a plaintext coefficient rotation by k can be operated as an homomorphic automorphism [GHS12] that requires rotation-specific *rotation-keys* (i.e., a key for each needed rotation parameter k). Although generating a single key for $k = 1$ would suffice for operating any rotation in theory, it is more efficient to generate keys for all (or most) of the rotations required by the circuit, in order to operate all (or most) rotations in constant-time.

[1] `https://github.com/Microsoft/SEAL`
[2] `https://palisade-crypto.org/`
[3] `https://github.com/tuneinsight/lattigo`

Plaintext Encoding Due to the inherent error of the encryption scheme, the $\mathsf{HE.Decrypt}$ procedure outputs an approximate message of the form $c_0 + c_1 s = m + e_{\mathsf{ct}}$ for an error term e_{ct} that depends on the various error terms introduced in the encryption and evaluation steps. Although noisy messages might be enough for some use cases, most applications typically require a fixed precision or even exact arithmetic. This is usually done by relying on plaintext encoding and decoding techniques that can be then formulated as *front-end* schemes that provide plaintext-space guarantees on top of the core RLWE scheme. Common strategies include scaling the plaintext up by some factor Δ and relying on quantization and rounding for the decoding [CKKS17; FV12]. Furthermore, it is common to apply FFT-like transforms to the plaintext polynomials in order to enable coefficient-wise encrypted arithmetic. Such techniques, often referred to as *packed* encoding, enable users to encode up to n messages in $\mathbb{Z}_q$ into n independent ciphertext *slots*, where n is the polynomial degree. The chosen encoding strategy often requires also defining front-end homomorphic operations (i.e., a front-end-specific Eval algorithm), as these operations must preserve the encoding. In the next section, we recall the BFV scheme as an example of a front-end scheme.

1.4.2 The BFV Front-End Scheme

The BFV scheme [FV12] relies on message-space scaling for its encoding/decoding strategy, in order to obtain an exact HE scheme over the integers from the core HE scheme (Scheme 1). More specifically, its plaintext space is the ring $R_t = \mathbb{Z}_t[X]/(X^n + 1)$ for $t < q$, and we denote $\Delta = \lfloor q/t \rfloor$, the integer division of q by t rounded down. We detail the front-end operations (i.e., that are redefined from the core scheme) in Scheme 2.

The BFV decryption of a ciphertext (c_0, c_1) can be seen as a two-step process. The first step runs the core $\mathsf{HE.Decrypt}$ operation to compute a noisy plaintext in R_q with

$$[c_0 + sc_1]_q = \Delta m + e_{\mathsf{ct}}, \tag{1.1}$$

where e_{ct} is the ciphertext overall error, or *ciphertext noise*. In the second step, the message is decoded from the noisy term in R_q to a plaintext in R_t, by rescaling and rounding

$$[\lfloor \frac{t}{q} (\Delta m + e_{\mathsf{ct}}) \rceil]_t = [\lfloor m + at + v \rceil]_t, \tag{1.2}$$

where $m \in R_t$, a has integer coefficients, and v has rational coefficients. Provided that $\|v\| < \frac{1}{2}$, Eq. (1.2) outputs m. Hence, the correctness of the scheme is conditioned on the noise magnitude $\|e_{\mathsf{ct}}\|$ that must be kept below $\frac{q}{2t}$ throughout the homomorphic computation, notably by choosing a sufficiently large q.

1.4.3 Notes on the Core & Front-End Paradigm

The main advantage of the core/front-end paradigm is that it enables us to define a common structure for RLWE-based HE schemes. Such a systematization is relevant from both a practical and theoretical perspective: On the practical side, it provides a meaningful structure for implementations by highlighting common functionalities, thus avoiding code duplication. For example, the BFV front-end scheme (Scheme 2) (as for all other front-ends to date) does not redefine the key-generation operations of the core scheme. Hence, an implementation of the core scheme can be used for generating keys of its front ends directly. From the theoretical perspective, the systematization is also useful in that attacks, improvements, and extensions to

Scheme 1: $\mathsf{HE}(n, q, \mathbf{w}, \mathsf{Key}, \mathsf{Err})$ ▷ *The core RLWE-based HE scheme*

$\underline{\mathsf{HE.SecKeyGen}}$:

1. Sample $s \leftarrow \mathsf{Key}$ and output: $\mathsf{sk} = s$

$\underline{\mathsf{HE.EncKeyGen}(\mathsf{sk} = s)}$:

1. Sample $p_1 \leftarrow R_q$, and $e \leftarrow \mathsf{Err}$ and output $\mathsf{pk} = (p_0, p_1) = (-sp_1 + e, p_1)$

$\underline{\mathsf{HE.RelinKeyGen}(\mathsf{sk} = s, \mathbf{w})}$:

1. Sample $\mathbf{r}_1 \leftarrow R_q^l$, $\mathbf{e} \leftarrow \mathsf{Err}^l$ and output: $\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1) = (s^2\mathbf{w} - s\mathbf{r}_1 + \mathbf{e},\ \mathbf{r}_1)$

$\underline{\mathsf{HE.Encrypt}(\mathsf{pk} = (p_0, p_1), m)}$:

1. Sample $u \leftarrow \mathsf{Key}$ and $e_0, e_1 \leftarrow \mathsf{Err}$ and output: $\mathsf{ct} = (m + up_0 + e_0\ ,\ up_1 + e_1)$

$\underline{\mathsf{HE.Add}(\mathsf{ct} = (c_0, c_1), \mathsf{ct}' = (c_0', c_1'))}$:

1. Output: $\mathsf{ct}_{\text{add}} = (c_0 + c_0', c_1 + c_1')$

$\underline{\mathsf{HE.Mul}(\mathsf{ct} = (c_0, c_1), \mathsf{ct}' = (c_0', c_1'))}$:

1. Output: $\mathsf{ct}_{\text{mul}} = (c_0c_0'\ ,\ c_0c_1' + c_0'c_1\ ,\ c_1c_1')$

$\underline{\mathsf{HE.Relinearize}(\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1), \mathsf{ct}_{\text{mul}} = (c_0, c_1, c_2))}$:

1. Decompose c_2 in base $\mathbf{w}$ as $\mathbf{c}_2$
2. Output $\mathsf{ct}_{\text{lin}} = (c_0 + \sum_{i=0}^{l} \mathbf{r}_0^{(i)}\mathbf{c}_2^{(i)},\ c_1 + \sum_{i=0}^{l} \mathbf{r}_1^{(i)}\mathbf{c}_2^{(i)})$

$\underline{\mathsf{HE.Decrypt}(\mathsf{sk} = s, \mathsf{ct} = (c_0, c_1))}$:

1. Output: $m' = c_0 + c_1 s$

Scheme 2: $\mathsf{BFV}(t, n, q, w, \mathsf{Key}, \mathsf{Err})$ ▷ *The BFV front-end scheme*

$\underline{\mathsf{BFV.Encrypt}(m \in R_t)}$:

1. Scale the message up in R_q as $m_{\text{enc}} = \Delta m$
2. Output $\mathsf{ct} = \mathsf{HE.Encrypt}(m_{\text{enc}})$

$\underline{\mathsf{BFV.Decrypt}(\mathsf{sk}, \mathsf{ct})}$:

1. Set $m'_{\text{enc}} = \mathsf{HE.Decrypt}(\mathsf{sk}, \mathsf{ct})$
2. Output $m' = [\lfloor \frac{t}{q} m'_{\text{enc}} \rceil]_t$ in R_t

$\underline{\mathsf{BFV.Mul}(\mathsf{ct}, \mathsf{ct}')}$:

1. Set $(c_0, c_1, c_2) = \mathsf{HE.Mul}(\mathsf{ct}, \mathsf{ct}')$
2. Output: $\mathsf{ct}_{\text{mul}} = ([\lfloor \frac{tc_0}{q} \rceil]_q, [\lfloor \frac{tc_1}{q} \rceil]_q, [\lfloor \frac{tc_2}{q} \rceil]_q)$

the core scheme will also directly apply to its front ends. Indeed, although it might seem that the core/front-end paradigm is purely functional, it highlights an important security consideration: approximate HE schemes are not de-facto secure against adversaries that have access to a decryption oracle [LM21] (referred to as the IND-CPA$^{\mathrm{D}}$ security model in [LM21]). Informally, this is because the approximation error after decryption is related to the error terms added to the keys and ciphertexts, and this error terms must remain secret for the RLWE assumption to hold. Although the first demonstrated attack specifically targeted the CKKS front-end scheme, it in fact affects the core scheme. Another way to look at the question is therefore that, as of today, IND-CPA$^{\mathrm{D}}$ security has been provided by the front-end schemes. Fortunately, not only attacks but also improvements to the core scheme are carried out to the front-end schemes. For example, countermeasures to ensure IND-CPA$^{\mathrm{D}}$ security at the core scheme level would transfer to future approximate front-end schemes, without being specific to CKKS. Similarly, core scheme extensions to new functionalities can transfer, hopefully with few adaptations, to the front-ends schemes. This will be the case for our multiparty extension, which we present in the next chapter.

1.5 Related Work

In this section, we provide a high-level discussion of MPC protocols in the dishonest majority setting: We discusses the predominantly implemented approach based on linear secret-sharing scheme (LSSS) and its several drawbacks, then we provide some historical context about the MHE-based solutions. We will discuss the state-of-the-art methods in MHE in more detail in the following chapters, when discussing our contributions.

1.5.1 MPC Applications

This last few decades have seen the established theoretical field of MPC evolve into an applied one, notably due to its potential for securing *data-sharing* scenarios in the financial [BCDG+09; BTW12], biomedical [JWBB+17; RTMS+18] and law-enforcement [BJSV15; KFB14] sectors, as well as for protecting digital assets [ABLK+18]. In particular, the use of passively-secure MPC techniques in such scenarios has been demonstrated to be effective and practical [JWBB+17; CWB18; AMP18], notably in the medical sector, where data collaborations are mutually beneficial and well-regulated yet legally require a certain level of data-protection [CWB18; RTMS+18].

1.5.2 LSSS-based MPC

According to a survey by Hastings et al. [HHNZ19], most of MPC frameworks available today for the dishonest-majority setting are based on the secret-sharing [Sha79] of the input data, according to a linear secret-sharing scheme (LSSS). In these systems, the evaluation of arithmetic circuits is generally enabled by the homomorphism of the LSSS and by interactive protocols such as Beaver's triple-based protocol [Bea92]. Notable example of such implementations include the Sharemind [BLW08] framework and the MP-SPDZ library [Kel20] that implements the SPDZ protocol [DPSZ12] and several of its subsequent improvements and derivatives [DKLP+13; KOS16; KPR18].

The LSSS-based approaches have several practical limitations: (i) These approaches require a per-party communication that is linear in the size of the circuit, which can be problematic for parties with low-bandwidth. (ii) They require a number of communication rounds, that is proportional to the circuit depth. This can be problematic for network with high-latency and

requires the parties to be online and active for the computation to make progress. (iii) For each circuit evaluation, these approaches require the prior distribution of a single-use correlated randomness, the size of which is proportional to the circuit size times the number of parties. When such a distribution cannot be performed by a trusted dealer (any collusion between the dealer and a curious party breaks the security of the protocol), LSSS-based approaches take the form of hybrids that generate the correlated randomness by relying on other techniques such as oblivious transfer [KOS16], plain HE [KPR18] and multiparty-HE [DPSZ12] in an *offline phase*. In practice, the cost of this offline phase quickly dominates the cost of the protocol, as the numbers of parties grow.

As a result of the aforementioned constraints, many current applications of LSSS-based MPC target the *outsourced* models where the actual computation is delegated to two parties [NWIJ+13; JWBB+17; MZ17; CB17; AMP18; CWB18] that are assumed to not collude (e.g., the *two-cloud model*). Unfortunately, this assumption might not be realistic in some contexts and the parties could require more guarantees such as those provided by running the MPC protocol themselves (i.e., without delegating trust).

1.5.3 MHE and MHE-based MPC

The idea of reducing communication costs in MPC by using (early constructions of) threshold homomorphic encryption can be traced back to a work by Franklin and Haber [FH96] and later improved by Cramer et al. [CDN01]. However, the lack (at that time) of efficient homomorphic schemes that preserve two distinct algebraic operations ruled out complete non-interactivity at the evaluation phase. More recent works have nonetheless demonstrated the practicality of multiparty *additive*-homomorphic encryption for task-specific instances, such as distributed machine learning [ZPGS19; FTSH20], thus suggesting the high potential that a *fully* homomorphic encryption solution could have. This is the idea behind the line of work by Asharov et al. [AJLT+12] and López-Alt et al. [LTV11; LTV12]. Following the introduction of the first practical FHE scheme by Gentry [Gen09], these work propose the first generic MPC protocols based on solely on MHE schemes.

Although of great interest, this line of work did not find much of an echo in applications as LSSS-based approaches have. One possible reason is the lack (at the time) of efficient and usable implementations of the contemporary HE schemes; these schemes are based on the *Learning with Errors* (LWE) assumptions [Reg05]. Today, multiple ongoing efforts aim at standardizing homomorphic encryption [ACCD+18] and at making its implementations available to a broader public. This new generation of schemes, based mostly on the *Ring Learning with Errors* assumptions [LPR10] (see Section 1.3.5), has brought HE-based techniques from being practical to being efficient.

As a result, we argue that MHE-based approaches are now efficient and flexible enough to support a broad range of MPC scenarios. In this dissertation, we therefore present our contributions that bring these techniques from theory to practice.

Chapter 2

A Multiparty Homomorphic Encryption Scheme

Chapter Content

We propose our first construction, which is an RLWE-based MHE scheme with an N-out-of-N-threshold access-structure, as per the definition of Section 1.3.2 (with $T = N$). We follow the blueprint of Asharov et al. [AJLT+12], adapt it to the core RLWE scheme as presented in Section 1.4.1, and introduce several improvements:

– We propose a novel protocol for the generation of relinearization keys. This protocol adds significantly less noise in the resulting relinearization key, therefore enables a less noisy relinearization algorithm.
– We propose a generalization of the scheme's output procedure, the decryption protocol, into a re-encryption one.
– From this generalization, we derive novel protocols for enabling external computation-receivers, for bridging between the MHE-based and LSSS-based MPC protocols, and for efficiently refreshing the ciphertext noise.

This chapter is organized as follows: We first propose an extended definition for MHE schemes; it models the generalized output procedures (Section 2.1). Then, we propose our construction (Section 2.2) and instantiate it into a generic, passively secure, MPC protocol for dishonest-majority setting (Section 2.4). We discuss this MPC approach and show that it has several advantages over its LSSS-based counterparts: Notably, its per-party communication complexity is only linear in the circuit's inputs and outputs, and its execution does not require private party-to-party communication channels. Finally, we demonstrate the efficiency of the latter instantiation for three examples of MPC circuits (Section 2.5).

2.1 Extended MHE Scheme Definition

We first extend the MHE scheme definition of Section 1.3.2 with the syntax and properties of our generalized output procedure. These output procedures support *re-encryption* of MHE ciphertexts to another secret-key. They are used as a part of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol in order to re-encrypt a ciphertext from the input to the computation receiver party's key. We further elaborate on these procedures and their practical use, when introducing their constructions in Section 2.2.5. For the sake of the exposition, we formulate the following definitions for the case of $T = N$ (i.e., $\mathcal{P}$ is the only qualifying set in the access-structure of the scheme). We also formulate the following correctness properties for an exact scheme (i.e., assuming exact encoding/decoding and evaluation procedures). The formulation for the core HE scheme (which is approximate) is similar but requires that the output decryption deviates only from $f(m_1, ..., m_I)$ by more than a defined threshold with negligible probability.

Recall that, in MHE, N parties in a set $\mathcal{P}$ hold secret-keys $\mathsf{sk}_1, ..., \mathsf{sk}_N$ that can be seen as shares of an ideal secret key $\mathcal{S}(\mathsf{sk}_1, ..., \mathsf{sk}_N)$. The generalized output procedures are as follows:

- **Key Switching** $\mathsf{ct}' \leftarrow \Pi_{\mathsf{KeySwitch}}(\mathsf{ct}, \mathsf{sk}_1, ..., \mathsf{sk}_N, \mathsf{sk}'_1, ..., \mathsf{sk}'_N)$:
  Given a ciphertext ct encrypted under secret-key $\mathsf{sk} = \mathcal{S}(\mathsf{sk}_1, ..., \mathsf{sk}_N)$ (the *input* ideal secret key), the parties in $\mathcal{P}$ re-encrypt ct under secret-key $\mathsf{sk}' = \mathcal{S}'(\mathsf{sk}'_1, ...\mathsf{sk}'_N)$ (the *output* ideal secret key), where sk'_i is a secret-key share known by party P_i.

- **Public-Key Switching** $\mathsf{ct}' \leftarrow \Pi_{\mathsf{PubKeySwitch}}(\mathsf{ct}, \mathsf{sk}_1, ..., \mathsf{sk}_N, \mathsf{pk}')$:
  Given a ciphertext ct encrypted under $\mathsf{sk} = \mathcal{S}(\mathsf{sk}_1, ..., \mathsf{sk}_N)$ (the *input* ideal secret key) and an output public-key pk' (the *output* public key) for secret-key sk' (the *output* secret key), the parties in $\mathcal{P}$ re-encrypt ct under sk'.

Additionally, the generalized output procedures satisfy the following correctness properties:

1. (KeySwitch-**Correctness**): For all arithmetic functions $f : \mathcal{M}^I \to \mathcal{M}$ over the parties' inputs $m_1, \ldots, m_I$, there exists $pp = (n, q, w, \mathsf{Err}, \mathsf{Key})$ such that, for $\mathsf{sk}' = \mathcal{S}'(\mathsf{sk}'_1, ..., \mathsf{sk}'_N)$ an output secret-key and

$$\begin{aligned}
\mathsf{sk}_i &\leftarrow \Pi_{\mathsf{SecKeyGen}} \quad i \in 1...N, \\
\mathsf{cpk} &\leftarrow \Pi_{\mathsf{EncKeyGen}}(\mathsf{sk}_1, ..., \mathsf{sk}_N), \\
\mathsf{rlk} &\leftarrow \Pi_{\mathsf{RelinKeyGen}}(\mathsf{sk}_1, ..., \mathsf{sk}_N), \\
\mathsf{ct}_i &\leftarrow \mathsf{MHE.Encrypt}(\mathsf{cpk}, m_i) \quad i \in 1...I, \\
\mathsf{ct}_f &\leftarrow \mathsf{MHE.Eval}(f, \mathsf{rlk}, \mathsf{ct}_1, ..., \mathsf{ct}_I), \\
\mathsf{ct}'_f &\leftarrow \Pi_{\mathsf{KeySwitch}}(\mathsf{ct}_f, \mathsf{sk}_1, ..., \mathsf{sk}_N, \mathsf{sk}'_1, ..., \mathsf{sk}'_N),
\end{aligned}$$

   it holds that $\Pr[\mathsf{HE.Decrypt}(\mathsf{sk}', \mathsf{ct}'_f) \neq f(m_1, ..., m_I)] < 2^{-\kappa}$.

2. (PubKeySwitch-**correctness**): For all arithmetic functions $f : \mathcal{M}^I \to \mathcal{M}$ over the parties' inputs $m_1, \ldots, m_I$, there exists $pp = (n, q, w, \mathsf{Err}, \mathsf{Key})$ such that, for sk' an output secret-key and $\mathsf{pk}' = \mathsf{HE.EncKeyGen}(\mathsf{sk}')$ and

$$\begin{aligned}
\mathsf{sk}_i &\leftarrow \Pi_{\mathsf{SecKeyGen}} \quad i \in 1...N, \\
\mathsf{cpk} &\leftarrow \Pi_{\mathsf{EncKeyGen}}(\mathsf{sk}_1, ..., \mathsf{sk}_N), \\
\mathsf{rlk} &\leftarrow \Pi_{\mathsf{RelinKeyGen}}(\mathsf{sk}_1, ..., \mathsf{sk}_N), \\
\mathsf{ct}_i &\leftarrow \mathsf{MHE.Encrypt}(\mathsf{cpk}, m_i) \quad i \in 1...I, \\
\mathsf{ct}_f &\leftarrow \mathsf{MHE.Eval}(f, \mathsf{rlk}, \mathsf{ct}_1, ..., \mathsf{ct}_I), \\
\mathsf{ct}'_f &\leftarrow \Pi_{\mathsf{PubKeySwitch}}(\mathsf{ct}_f, \mathsf{sk}_1, ..., \mathsf{sk}_N, \mathsf{pk}'),
\end{aligned}$$

   it holds that $\Pr[\mathsf{HE.Decrypt}(\mathsf{sk}', \mathsf{ct}'_f) \neq f(m_1, ..., m_I)] < 2^{-\kappa}$.

2.2 N-out-of-N-Threshold Scheme Construction

Here, we present our MHE construction. In order to abstract the actual system model, we assume an abstract channel over which the parties can disclose shares to their peers. In Section 2.4.3, we present concrete system models and discuss their features. Let $\mathsf{CRS}(R_q)$ be the uniform distribution in R_q, according to a common random string, i.e., elements sampled from this distribution are uniformly distributed and the same for all parties.

2.2.1 Ideal-Secret-Key Generation

We propose to use an additive structure for the combined secret-key, denoted as s in the following. We denote by s_i the secret key share of party P_i, thus

$$\mathsf{sk} = s = \sum_{P_i \in \mathcal{P}} s_i. \tag{2.1}$$

This enables a simple ideal-secret-key generation procedure $\Pi_{\mathsf{SecKeyGen}}$ in which each party samples independently its own share from the RLWE key-distribution.

Thus, the resulting $\Pi_{\mathsf{SecKeyGen}}$ procedure is non-interactive (which, actually, makes it an *algorithm* in our terminology). Equation. (2.1) applies, but this does not result in a *usual*

sharing of s, in the sense that the distribution of the shares is not uniform in R_q. This is not an issue because the security of our scheme (analyzed in Section 2.3.3) does not rely on this property. However, the norm of the resulting ideal secret key grows with $\mathcal{O}(N)$, which has an effect on the noise growth (analyzed in Section 2.3.2). By using techniques such as those described in [RSTV+22], it might be possible to generate ideal secret keys in R_3 as if they were produced in a trusted setup (e.g., as an additive secret-sharing of a usual RLWE secret-key over R_q). However, this would introduce the need for private channels between the parties.

2.2.2 Collective Encryption-Key Generation

The collective encryption-key generation, detailed in Protocol $\Pi_{\mathsf{EncKeyGen}}$, emulates the ideal scheme's $\mathsf{HE.EncKeyGen}$ procedure. In addition to the public parameters of the cryptosystem (which we will omit in the following), the procedure requires a public polynomial p_1, uniformly sampled in R_q and to be agreed upon by all the parties. For this purpose, they sample its coefficients from the common random string (CRS). After the execution of the $\Pi_{\mathsf{EncKeyGen}}$ protocol, the parties have access to the collective public encryption key

$$\mathsf{cpk} = \big(\big[\sum_{P_i \in \mathcal{P}} p_{0,i}\big]_q,\ p_1\big) = \big(\big[-(\sum_{P_i \in \mathcal{P}} s_i)p_1 + \sum_{P_i \in \mathcal{P}} e_i\big]_q,\ p_1\big), \tag{2.2}$$

that has the same form as the ideal public key pk in the BFV scheme, with larger worst-case norms $\|s\|$ and $\|e\|$. The norm increases only linearly in N hence is not a concern (see Section 2.3.2), even for a large number of nodes. Another notable feature of the $\Pi_{\mathsf{EncKeyGen}}$ protocol is that it applies to any kind of linear sharing of s, as long as the shares are valid RLWE secrets and the norm of the reconstruction is small enough. This includes uniformly random sharing over R_q of a traditional RLWE secret key in R_3.

2.2.3 Relinearization-Key Generation

Protocol $\Pi_{\mathsf{RelinKeyGen}}$ emulates the centralized $\mathsf{HE.RelinKeyGen}$. Informally, it produces pseudo-encryptions of $s^2\mathbf{w}^{(i)}$ for each element $i = 0, ..., l-1$ of the decomposition basis parameter $\mathbf{w}$. It requires a public element $\mathbf{a}$ to be uniformly sampled in R_q^l from the CRS. We use vector notation to express that these pseudo-encryptions are generated in parallel for every element of the decomposition base $\mathbf{w}$.

Asharov et al. propose a method to generate relinearization keys for multiparty schemes based on the *learning-with-errors* (LWE) problem [AJLT+12]. This method could be adapted to our scheme but results in significantly increased noise in the rlk (hence, higher noise in relinearized ciphertexts) with respect to the centralized scheme. One cause for this extra noise is the use of the public encryption algorithm to produce the mentioned pseudo-encryptions. By observing that the collective encryption key is not needed for this purpose (because the secret key is collectively known), we propose Protocol $\Pi_{\mathsf{RelinKeyGen}}$ as an improvement over the method by Asharov et al.[1]

After completing the $\Pi_{\mathsf{RelinKeyGen}}$ protocol, the parties have access to a public relinearization key of the form

$$\mathsf{rlk} = \ (\mathbf{r}_0, \mathbf{r}_1) = (-s\mathbf{b} + s^2\mathbf{w} + s\mathbf{e}_0 + u\mathbf{e}_1 + \mathbf{e}_2 + \mathbf{e}_3 \ \ , \ \ \mathbf{b}\,), \tag{2.3}$$

where $\mathbf{b} = s\mathbf{a} + \mathbf{e}_1$ and $\mathbf{e}_k = \sum_j \mathbf{e}_{k,j}$ for $k = 0, 1, 2, 3$.

[1] Park subsequently proposed another improvement on the relinearization-key generation protocol; it requires a single round of communication [Par21].

Protocol 1. $\Pi_{\mathsf{SecKeyGen}}$ ▷ *The secret-key generation protocol (non-interactive)*

Private Output: $\mathsf{sk}_i = s_i$ (ideal secret-key share of party i)

Round 1:
Each party P_i:
1. samples $s_i \leftarrow \mathsf{Key}$.

Output:
(For party P_i): $\mathsf{sk}_i = s_i$.

Protocol 2. $\Pi_{\mathsf{EncKeyGen}}$ ▷ *The collective public encryption-key generation protocol*

Private Input for P_i: $s_i = \mathsf{sk}_i$ (secret key share)
Public Output: $\mathsf{cpk} = (p_0, p_1)$ (collective encryption key)

Round 1:
Each party P_i:
1. samples $e_i \leftarrow \mathsf{Err}$ and $p_1 \leftarrow \mathsf{CRS}$,
2. discloses $p_{0,i} = -p_1 s_i + e_i$.

Output:
From $p_0 = \sum_{P_j \in \mathcal{P}} p_{0,j}$, outputs $\mathsf{cpk} = (p_0\ ,\ p_1)$.

Protocol 3. $\Pi_{\mathsf{RelinKeyGen}}$ ▷ *The relinearization-key generation protocol*

Public Input: $\mathbf{w}$ the decomposition basis of size l
Private Input of P_i: $s_i = \mathsf{sk}_i$
Output: $\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1)$

Round 1:
Each party P_i:
1. samples $u_i \leftarrow \mathsf{Key}$, $\mathbf{e}_{0,i}, \mathbf{e}_{1,i} \leftarrow \mathsf{Err}$ and $\mathbf{a} \leftarrow \mathsf{CRS}$,
2. discloses $(\mathbf{h}_{0,i}\ ,\ \mathbf{h}_{1,i}) = (-u_i\mathbf{a} + s_i\mathbf{w} + \mathbf{e}_{0,i}\ \ ,\ \ s_i\mathbf{a} + \mathbf{e}_{1,i})$.

Round 2:
Each party P_i:
1. sets $\mathbf{h}_0 = \sum_{P_j \in \mathcal{P}} \mathbf{h}_{0,j}$ and $\mathbf{h}_1 = \sum_{P_j \in \mathcal{P}} \mathbf{h}_{1,j}$,
2. samples $\mathbf{e}_{2,i}, \mathbf{e}_{3,i} \leftarrow \mathsf{Err}^l$,
3. discloses $(\mathbf{h}'_{0,i}\ ,\ \mathbf{h}'_{1,i}) = (s_i\mathbf{h}_0 + \mathbf{e}_{2,i}\ \ ,\ \ (u_i - s_i)\mathbf{h}_1 + \mathbf{e}_{3,i})$.

Output:
Set $\mathbf{h}'_0 = \sum_{P_j \in \mathcal{P}} \mathbf{h}'_{0,j}$ and $\mathbf{h}'_1 = \sum_{P_j \in \mathcal{P}} \mathbf{h}'_{1,j}$, output $\mathsf{rlk} = (\mathbf{h}'_0 + \mathbf{h}'_1\ ,\ \mathbf{h}_1)$.

Hence, compared to the keys generated by adapting the approach of Asharov et al. (we provide the corresponding protocol in Appendix A.1), our keys have a lower error in $\mathbf{r}_0$ and no error at all in $\mathbf{r}_1$ (i.e., they have the same form as those of the centralized scheme, although with larger noise terms). This significantly reduces the noise induced by relinearization.

A relevant feature of the proposed $\Pi_{\mathsf{RelinKeyGen}}$ protocol is its independence from the actual decomposition basis $\mathbf{w}$: It is compatible with other decomposition techniques, such as the one used for Type II relinearization [FV12], those based on the Chinese Remainder Theorem (as proposed by Bajard et al. [BEHZ16] and Cheon et al. [CHKK+18]), and the hybrid approach of Bossuat et al. [BMTH21].

2.2.4 Packed-Encoding and Rotation Keys

One of the most powerful features of RLWE-based schemes is the ability to embed vectors of plaintext values into a single ciphertext. Such techniques, commonly referred to as *packing*, enable arithmetic operations to be performed in a *single-instruction multiple-data* fashion, where encrypted arithmetic results in element-wise plaintext arithmetic. Provided with public *rotation keys*, arbitrary rotations over the vector components [CHKK+18] can be operated homomorphically. Generating these rotation keys (which, similarly to the relinearization key, can be seen as pseudo-encryptions of rotations of the secret-key coefficients) can be done in the multiparty scheme, by means of an $\Pi_{\mathsf{RotKeyGen}}$ sub-protocol. We do not detail this protocol, as it is a straightforward adaptation of $\Pi_{\mathsf{EncKeyGen}}$. This enables a vast family of homomorphically computable linear and non-linear transformations on ciphertexts. We will make use of rotations in the input-selection example circuit in Section 2.5.2.

2.2.5 Collective Key-Switching Protocols

The key-switching functionality enables the oblivious re-encryption operation. Given a ciphertext ct encrypted under an *input key* s, along with an *output key* s', the key-switching procedure outputs $\mathsf{ct}' = \mathsf{Enc}(s', \mathsf{Dec}(s, \mathsf{ct}))$. Because the single-party decryption (Eq. (1.1)) is equivalent to switching from the ideal secret-key to an output key $s' = 0$, this protocol generalizes the decryption protocol.

Smudging We observe that the aforementioned decryption procedure, and the MHE key-switching procedures in general, provide the output-key owner(s) with the ciphertext noise. As this noise depends on intermediate values in the encryption, homomorphic computation, and key-switching procedures, it could be exploited as a side-channel by curious receivers. [2] The *smudging* technique, as introduced by Asharov et al. [AJLT+12], aims at making the ciphertext-noise unexploitable by flooding it with some freshly sampled noise terms in a distribution of larger-variance.

We achieve this by sampling the relevant error terms in the key-switching protocols from a discrete Gaussian distribution of variance $\sigma^2_{\mathrm{smg}} = 2^\lambda \sigma^2_{\mathsf{ct}}$ where σ^2_{ct} is the key-switched ciphertext's noise variance (see Eq. (1.1)) and λ the desired security level (e.g., $\lambda = 128$, see Section 2.3.3). Hence, this technique assumes that the system keeps track of the ciphertext noise-level and has access to this property. For a ciphertext ct, we denote the variance of its noise term as $\mathsf{var}(\mathsf{ct})$, and the smudging distribution with variance σ^2 as $\mathsf{Smudge}(\sigma^2)$.

[2] Subsequently to our work, concrete attacks on the decryption noise were more thoroughly analysed, e.g., by Li and Micciancio in the context of CKKS [LM21]. Adapting these new results to MHE smudging would result in a much more practical approach: By studying the noise leakage in a computational setting, the variance of the smudging noise can be significantly reduced for the same security level [LMSS22; CHIV+22].

Receiver The protocol's instantiation depends on whether the parties performing the re-encryption have collective access to the output secret-key directly (e.g., in the case of internal receivers) or whether they have only its corresponding public-key (e.g., in the case of external receivers). Both these settings are relevant when instantiating the MHE scheme as an MPC protocol, which we discuss in Section 2.4. Therefore, we develop protocols that perform key-switching for these two settings: When s' is collectively known, the $\Pi_{\mathsf{KeySwitch}}$ protocol is used. When only a public key is known, the $\Pi_{\mathsf{PubKeySwitch}}$ protocol is used.

Collective Key-Switching

Protocol $\Pi_{\mathsf{KeySwitch}}$ details the steps for performing a key switching when the input parties collectively know the output secret key s'. This protocol can be used as a decryption protocol ($s' = 0$) or for updating the access-structure (see Section 2.2.8), and it is the basis for bridging MHE-based and LSSS-based approaches (see Section 2.2.6).

After the execution of the $\Pi_{\mathsf{KeySwitch}}$ protocol on input $\mathsf{ct} = (c_0, c_1)$ such that $c_0 + sc_1 = m + e_{\mathsf{ct}}$ where e_{ct} is the ciphertext's error, the parties have access to ct' such that

$$\begin{aligned}
\mathsf{HE.Decrypt}(s', \mathsf{ct}') &= c'_0 + s'c_1 \\
&= c_0 + \sum_j \big((s_j - s'_j)c_1 + e_j\big) + s'c_1 \\
&= c_0 + (s - s')c_1 + e_{\mathsf{CKS}} + s'c_1 \\
&= m + e_{\mathsf{ct}} + e_{\mathsf{CKS}}, \qquad (2.4)
\end{aligned}$$

where $e_{\mathsf{CKS}} = \sum_j e_j$. Hence, as long as $m + e_{\mathsf{ct}} + e_{\mathsf{CKS}}$ remains below the threshold for successful decoding by the concrete HE scheme in use, the $\mathsf{KeySwitch}$-correctness property is satisfied. For example, we require that $\|e_{\mathsf{ct}} + e_{\mathsf{CKS}}\| < q/(2t)$ for the correctness to hold when considering the BFV front-end scheme.

The use of the $\Pi_{\mathsf{KeySwitch}}$ protocol is limited to the cases where parties have collective knowledge of the output secret key s'. Yet, this might not be the case, for example, when considering an external receiver R for the key-switched ciphertext (we elaborate on external receivers in Section 2.4.1). This situation would require confidential channels between the receiver and each party in $\mathcal{P}$, in order either (i) to collect decryption shares from all parties, or (ii) to distribute an additive sharing of its secret key to the system. However, (i) would become expensive for a large number of parties, and (ii) would require R to trust at least one party in $\mathcal{P}$. Furthermore, both approaches would require confidential channels between each of the parties and the receiver, and this might not fit the system model. Instead, it would be more desirable that the parties re-encrypt the target ciphertext under s' given a public-key for s', in such a way that the ciphertext can be retrieved and decrypted with a single interaction from the receiver (and possibly at a later stage).

Collective Public-Key Switching

Protocol $\Pi_{\mathsf{PubKeySwitch}}$ details the steps for performing a collective key-switching when the input parties know only a public key for the output secret key s'. As it requires only public input from the receiver, $\Pi_{\mathsf{PubKeySwitch}}$ enables an *external* party (i.e., that is not part of an input access-structure) to obtain an output without the need for private channels with the parties.

Let $\mathsf{ct} = (c_0, c_1)$ be an input ciphertext such that $c_0 + sc_1 = m + e_{\mathsf{ct}}$ and $\mathsf{pk}' = (p'_0, p'_1)$ be a public key such that $p'_0 = -(s'p'_1 + e_{\mathsf{pk}'})$. After the execution of the $\Pi_{\mathsf{PubKeySwitch}}$ protocol on ct

Protocol 4. $\Pi_{\mathsf{KeySwitch}}$ ▷ *The collective key-switching protocol*

Public input: $\mathsf{ct} = (c_0, c_1)$ with $\mathsf{var}(\mathsf{ct}) = \sigma^2_{\mathsf{ct}}$
Private input for P_i: s_i, s'_i
Public output: $\mathsf{ct}' = (c'_0, c_1)$

Round 1:
Each party P_i:
1. samples $e_i \leftarrow \mathsf{Smudge}(\sigma^2_{\mathsf{ct}})$,
2. discloses $h_i = (s_i - s'_i)c_1 + e_i$.

Output:
Set $h = \sum_{P_j \in \mathcal{P}} h_j$ and output $\mathsf{ct}' = (c'_0, c_1) = (c_0 + h, c_1)$.

Protocol 5. $\Pi_{\mathsf{PubKeySwitch}}$ ▷ *The collective public-key switching protocol*

Public input: $\mathsf{pk}' = (p'_0, p'_1)$, $\mathsf{ct} = (c_0, c_1)$, $\mathsf{var}(\mathsf{ct}) = \sigma^2_{\mathsf{ct}}$
Private input for P_i: s_i
Public output: $\mathsf{ct}' = (c'_0, c'_1)$

Round 1:
Each party P_i
1. samples $u_i \leftarrow \mathsf{Key}$, $e_{0,i} \leftarrow \mathsf{Smudge}(\sigma^2_{\mathsf{ct}})$, $e_{1,i} \leftarrow \mathsf{Err}$,
2. discloses $(h_{0,i}\ ,\ h_{1,i}) = (s_i c_1 + u_i p'_0 + e_{0,i}\ \ ,\ \ u_i p'_1 + e_{1,i})$.

Output:
Set $h_0 = \sum_j h_{0,j}$ and $h_1 = \sum_{P_j \in \mathcal{P}} h_{1,j}$, output $\mathsf{ct}' = (c'_0, c'_1) = (c_0 + h_0, h_1)$.

Protocol 6. $\Pi_{\mathsf{ColBootstrap}}$ (BFV variant) ▷ *The collective refresh protocol*

Public input: $\mathsf{ct} = (c_0, c_1)$ $\mathsf{var}(\mathsf{ct}) = \sigma^2_{\mathsf{ct}}$
Private input for P_i: s_i
Public output: $\mathsf{ct}' = (c'_0, c'_1)$ with noise variance $N\sigma^2$

Round 1:
Each party P_i
1. samples $M_i \leftarrow R_t$, $e_{0,i} \leftarrow \mathsf{Smudge}(\sigma^2_{\mathsf{ct}})$, $e_{1,i} \leftarrow \mathsf{Err}$ and $a \leftarrow \mathsf{CRS}$,
2. discloses $(h_{0,i}\ ,\ h_{1,i}) = (s_i c_1 - \Delta M_i + e_{0,i}\ ,\ -s_i a + \Delta M_i + e_{1,i})$.

Output:
Set $h_0 = \sum_j h_{0,j}$, $h_1 = \sum_j h_{1,j}$ and output $(c'_0, c'_1) = ([\lfloor \frac{t}{q}([c_0 + h_0]_q)\rceil]_t \Delta + h_1\ ,\ a)$.

with output public key pk', the parties hold ct' satisfying

$$\begin{aligned}\mathsf{HE.Decrypt}(s', \mathsf{ct}') &= c'_0 + s'c'_1 \\ &= c_0 + \sum_j \left(s_j c_1 + u_j p'_0 + e_{0,j}\right) + s' \sum_j \left(u_j p'_1 + e_{1,j}\right) \\ &= c_0 + sc_1 + up'_0 + s'up'_1 + e_0 + s'e_1 \\ &= m + e_{\mathsf{ct}} + e_{\mathsf{PCKS}}, \end{aligned} \tag{2.5}$$

where $e_d = \sum_j e_{d,j}$ for $d = 0, 1$, $u = \sum_j u_j$, and the total added noise $e_{\mathsf{PCKS}} = e_0 + s'e_1 + ue_{\mathsf{pk}}$ depends on both the protocol-induced and the target-public-key noises. Similarly as for the $\Pi_{\mathsf{KeySwitch}}$ protocol, the $\mathsf{PubKeySwitch}$-correctness property is satisfied if $m + e_{\mathsf{ct}} + e_{\mathsf{PCKS}}$ can be successfully decoded. In the case of BFV, this holds if $\|e_{\mathsf{ct}} + e_{\mathsf{PubKeySwitch}}\| < q/(2t)$.

2.2.6 Bridging MPC Approaches

The flexibility of the $\Pi_{\mathsf{KeySwitch}}$ protocol can be harnessed to bridge the MHE-based and LSSS-based MPC approaches. More precisely, we consider the task of switching from an MHE encryption to an additive sharing of the message among the parties in $\mathcal{P}$, as well as the reverse operation. For the sake of this exposition, we will consider the BFV front-end scheme with the plaintext space $\mathcal{M} = R_t$ for t an NTT-friendly prime. Note that the procedure can be adapted to non-integer plaintext space [FTPS+21].

Encryption-to-Shares Given an encryption (c_0, c_1) of a plaintext $m \in R_t$, the $\Pi_{\mathsf{Enc2Share}}$ protocol outputs an additive sharing of m over R_t. Intuitively, this protocol corresponds to the parties masking their share in the decryption (i.e., $\Pi_{\mathsf{KeySwitch}}$ with $s' = 0$) protocol: Each party $P_i \in \{P_2, \ldots, P_N\}$ samples its own additive share $M_i \leftarrow R_t$ and adds a $-\Delta M_i$ term to its decryption share h_i before disclosing it. Party P_1 does not disclose its decryption share h_1 and derives its own additive share of m as

$$M_1 = \mathsf{BFV.Decrypt}(s_1, (c_0 + \sum_{i=2}^{N} h_i, c_1)) = m - \sum_{i=2}^{N} M_i.$$

Shares-to-Encryption Given a secret-shared value $m \in R_t$ such that $m = \sum_{i=1}^{N} M_i$, the $\Pi_{\mathsf{Share2Enc}}$ protocol outputs an encryption $\mathsf{ct} = (c_0, c_1)$ of m. This is, each party P_i samples a from the CRS and produces a $\Pi_{\mathsf{KeySwitch}}$ share for the ciphertext $(\Delta M_i, a)$ with input key 0 and output key s. The ciphertext that encrypts m is then given by $\mathsf{ct} = (\sum_{i=1}^{N} c_{0,i}, a)$.

2.2.7 Collective Bootstrapping

We combine the $\Pi_{\mathsf{Share2Enc}}$ and $\Pi_{\mathsf{Enc2Share}}$ protocols into a *multiparty bootstrapping* procedure (Protocol $\Pi_{\mathsf{ColBootstrap}}$) that enables the reduction of a ciphertext noise to further compute on it. This is a crucial functionality for RLWE schemes (such as the BFV scheme) for which the centralized bootstrapping procedure is expensive. Intuitively, the $\Pi_{\mathsf{ColBootstrap}}$ protocol consists of a conversion from an encryption to secret-shares and back, implemented as a parallel execution of the $\Pi_{\mathsf{Enc2Share}}$ and $\Pi_{\mathsf{Share2Enc}}$ protocols. It is an efficient single-round interactive protocol that the parties can use during the evaluation phase, instead of a computationally heavy bootstrapping procedure. In practice, a broad range of applications would not (or seldom) need to rely on this primitive, as the circuit complexity enabled by the practical parameters of the scheme suffices.

Table 2.1: Comparison with multi-key schemes: dependency in the number of parties N

	Size		Time	
Scheme	Ciphertext	Switching-key	Mult.+Relin.	Rotate
Multi-key MHE (Chen et al.[CDKS19])	$O(N)$	$O(N)$	$O(N^2)$	$O(N)$
Threshold MHE (this work)	$O(1)$	$O(1)$	$O(1)$	$O(1)$

But the $\Pi_{\mathsf{ColBootstrap}}$ protocol offers a trade-off between computation and communication (we demonstrate this in Section 2.5.3). As for the LSSS-bridge protocols, Froelicher et al. adapted this procedure to the CKKS instantiation of the MHE scheme [FTPS+21].

2.2.8 Dynamic Access-Structure

The scenario of parties joining and leaving the system corresponds to a secret-key update and is handled by the $\Pi_{\mathsf{KeySwitch}}$ and $\Pi_{\mathsf{PubKeySwitch}}$ protocols. More specifically, we consider the task of transferring a ciphertext from an input set of parties $\mathcal{P}$ to an output set $\mathcal{P}'$. If $\mathcal{P}' \subset \mathcal{P}$, the parties in $\mathcal{P} - \mathcal{P}'$ can simply use the $\Pi_{\mathsf{KeySwitch}}$ protocol with output key $s' = 0$. Otherwise, the parties use the $\Pi_{\mathsf{PubKeySwitch}}$ protocol with pk' set to the collective public-key of $\mathcal{P}'$.

2.3 MHE Scheme Analysis

We now analyze the complexity, noise growth and security aspects of our proposed MHE scheme construction.

2.3.1 Comparison with Multi-key-HE

Multi-key HE schemes, as introduced by López-Alt et al. [LTV12], enable the evaluation of homomorphic operations directly over ciphertexts encrypted under different secret-keys. The access-structure of these schemes can be seen as dynamic; they include *on-the-fly* each new party in the computation circuit. Hence, these schemes do not require the generation of a collective public encryption-key. In their current instantiation, however, they require the generation of public relinearization and rotation keys for which the size depends on the number of parties N. Furthermore, their ciphertext size and homomorphic operation complexity also increase with N. Chen et al. [CDKS19] propose multi-key extensions for the BFV and CKKS schemes for which these dependencies are reported in Table 2.1.

We observe that, when *on-the-fly* computation is not required by the application (e.g., the set of nodes is known in advance), threshold schemes result in a more efficient construction. However, note that the multi-key and threshold approaches are not mutually exclusive. Hybrid constructions, where the threshold scheme is used to emulate a single key within a multi-key setting, are promising ways of tailoring the access structure to the sought security and functionality requirements[3]. For example, in an encrypted federated learning system, a fixed group of parties could train a model under threshold encryption and enable the prediction to be evaluated *on-the-fly* under multi-key encryption.

[3] A hybrid scheme was subsequently proposed by Kwak et al.[KLSW21].

2.3.2 Noise Analysis

We analyze the effect of distributing the BFV cryptosystem on the ciphertext noise. As the distribution affects only the magnitude of the scheme's secrets (key and noise), the original cryptosystem analysis [FV12] directly applies, though with a larger worst-case error norm that we express as a function of the number of parties N in the following. The derivations for the equations presented in this section can be found in Appendix A.2.

Ideal Secret-Key and Encryption-Key As a result of the secret-key generation procedure, where each additive share s_i is sampled from R_3, we know that $\|s\| \leq N$. As a result of the $\Pi_{\mathsf{EncKeyGen}}$ protocol, the collective public key noise is $e_{\mathsf{cpk}} = \sum_{i=1}^{N} e_i$ (see Eq. (2.2)), which implies that $\|e_{\mathsf{cpk}}\| \leq NB$, where B is the worst-case norm for an error term from Err.

Fresh Encryption Let $\mathsf{ct}= (c_0, c_1)$ be a fresh encryption of a message m under a collective public key. The first step of the decryption (Eq. (1.1)) under the *ideal* secret key outputs $c_0 + sc_1 = m + e_{fresh}$, where

$$\|e_{fresh}\| \leq B(2nN + 1). \tag{2.6}$$

Thus, for a key generated by the $\Pi_{\mathsf{EncKeyGen}}$ protocol, the worst-case fresh ciphertext noise is linear in the number N of parties.

Collective Key-Switching Let $\mathsf{ct} = (c_0, c_1)$ be an encryption of m under the collective secret key s, and $\mathsf{ct}' = (c'_0, c_1)$ be the output of the $\Pi_{\mathsf{KeySwitch}}$ protocol on ct with target key s'. Then, $c'_0 + s'c_1 = m + e_{fresh} + e_{\mathsf{CKS}}$ with

$$\|e_{\mathsf{CKS}}\| \leq B_{smg}N, \tag{2.7}$$

where B_{smg} is the bound of the smudging distribution. We observe that the additional noise does not depend on the destination key s'.

Public Collective Key-Switching Let $\mathsf{ct} = (c_0, c_1)$ be an encryption of m under the collective secret key s, and $\mathsf{ct}' = (c'_0, c'_1)$ be the output of the $\Pi_{\mathsf{PubKeySwitch}}$ protocol on ct and target public key $\mathsf{pk}' = (p'_0, p'_1)$, such that $p'_0 = -s'p'_1 + e_{\mathsf{pk}'}$. Then, $c'_0 + s'c'_1 = m + e_{fresh} + e_{\mathsf{PCKS}}$ with

$$\|e_{\mathsf{PCKS}}\| \leq N(nB_{\mathsf{pk}'} + n\|s'\|B + B_{\text{smg}}), \tag{2.8}$$

where $\|e_{\mathsf{pk}'}\| \leq B_{\mathsf{pk}'}$, and B_{smg} is the bound on the smudging noise. Note that in this case, the smudging noise should dominate this term.

2.3.3 Standalone MHE Security

Here,we state the security theorem for the MHE scheme in the passive-adversary model. At this stage, we consider the standalone security of the scheme, i.e., the security of each protocol separately, and we defer the discussion about the composition of multiple MHE protocols to Section 2.4, where we discuss the full MHE-based MPC protocol. We formulate the security in the ideal/real simulation formalism [Lin17], as Theorem 1. We state that the adversary's view in the execution of the protocols can be simulated, in such a way that it is indistinguishable from the real view, by an ideal world simulator that has access only to the adversary's inputs. For two distributions D and $\tilde{D}$, we denote computational indistinguishability as $D \overset{c}{\equiv} \tilde{D}$ and statistical indistinguishability as $D \overset{s}{\equiv} \tilde{D}$. For a protocol Π, we denote $\Pi(\{x_i\}_{P_i \in \mathcal{P}})$ the output of the protocol execution under the parties' inputs (where x_i is the private input of party P_i), and we denote $\mathsf{view}_{\mathcal{A}}^{\Pi}$ the adversary's view in the protocol execution.

Theorem 1 (MHE Security for semi-honest model)**.** *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be a set of N parties and let $\mathcal{A} \subset \mathcal{P}$ be a set of corrupted parties (the adversary) where $|\mathcal{A}| \leq N-1$. For each possible set $\mathcal{A}$ of corrupted parties:*

– *There exists a simulator $S^{\mathsf{SecKeyGen}}$ such that, for $\{s_i\}_{P_i \in \mathcal{P}} \leftarrow \Pi_{\mathsf{SecKeyGen}}$,*

$$S^{\mathsf{SecKeyGen}}(\{s_i\}_{P_i \in \mathcal{A}}) \stackrel{c}{\equiv} \mathit{view}_{\mathcal{A}}^{\Pi_{\mathsf{SecKeyGen}}}.$$

– *There exists the simulators $S^{\mathsf{EncKeyGen}}, S^{\mathsf{RelinKeyGen}}$ such that, for $\{s_i\}_{P_i \in \mathcal{P}} \leftarrow \Pi_{\mathsf{SecKeyGen}}$, $\mathsf{cpk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}(\{s_i\}_{P_i \in \mathcal{P}})$ and $\mathsf{rlk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{RelinKeyGen}}(\{s_i\}_{P_i \in \mathcal{P}})$,*

$$S^{\mathsf{EncKeyGen}}(\{s_i\}_{P_i \in \mathcal{A}}, \mathsf{cpk}) \stackrel{c}{\equiv} \mathit{view}_{\mathcal{A}}^{\Pi_{\mathsf{EncKeyGen}}},$$
$$S^{\mathsf{RelinKeyGen}}(\{s_i\}_{P_i \in \mathcal{A}}, \mathsf{rlk}) \stackrel{c}{\equiv} \mathit{view}_{\mathcal{A}}^{\Pi_{\mathsf{RelinKeyGen}}}.$$

– *There exists a simulator $S^{\mathsf{KeySwitch}}$ such that, for $\{s_i\}_{P_i \in \mathcal{P}} \leftarrow \Pi_{\mathsf{SecKeyGen}}$ some input secret-keys, $\{s'_i\}_{P_i \in \mathcal{P}} \leftarrow \Pi_{\mathsf{SecKeyGen}}$ some target secret-keys, ct a ciphertext encrypting a message $m \leftarrow \mathcal{M}$ under secret-key $\sum_{i=1}^{N} s_i$ with ciphertext noise variance σ^2_{ct}, smudging noise terms $\{e_i^{smg}\}_{P_i \in \mathcal{P}} \leftarrow \mathsf{Smudge}(\sigma^2_{\mathsf{ct}})$ and $(c'_0, c'_1) = \mathsf{ct}' \leftarrow \Pi_{\mathsf{KeySwitch}}(\{s_i, s'_i, e_i^{smg}\}_{P_i \in \mathcal{P}}, \mathsf{ct})$, it holds that*

$$S^{\mathsf{KeySwitch}}(\{s_i, s'_i\}_{P_i \in \mathcal{A}}, \mathsf{ct}) \stackrel{c}{\equiv} \mathit{view}_{\mathcal{A}}^{\Pi_{\mathsf{KeySwitch}}},$$
$$m + \sum_{i=1}^{N} e_i^{smg} \stackrel{s}{\equiv} c'_0 + c'_1 \sum_{i=1}^{N} s'_i.$$

Proof (Intuition) We provide the actual construction for each simulator program in Appendix A.3, and we briefly discuss the intuition for the proof of Theorem 1 here. We begin by observing that, as we assume that parties publicly disclose their share for each protocol round in the MHE scheme, the task of each simulator is to output simulated round shares for all parties. To simulate the shares of the honest parties (which it cannot compute), it generates all shares but one by sampling elements from R_q, uniformly at random. The last share is generated by subtracting the sum of all randomized shares to the protocol output (which is provided to the simulator in the ideal world and generally corresponds to the sum of all shares in the real-world) in order that the simulated transcript produces the correct output. As all protocol shares in the MHE scheme have the form of an RLWE sample (or a vector thereof), these fake shares are computationally indistinguishable from the real-world ones by the adversary, by the decision-RLWE assumption [LPR10] (see Section 1.3.5). Note that in the worst case in which $|\mathcal{A}| = N-1$, the simulator as described above generates the real-world transcript.

2.4 MHE-Based Secure Multiparty Computation

We discuss the instantiation of our MHE scheme construction in a generic secure-multiparty-computation (MPC) protocol. Using MHE schemes to achieve MPC is not new [CDN01; AJLT+12], but each new generation of HE schemes makes this approach more efficient and flexible. However, to the best of our knowledge, no generic MPC solution has yet been implemented to exploit these ideas. We discuss how MHE-based solutions can lead to a new generation of MPC systems: not only in a traditional peer-to-peer setting but also in an outsourced setting where parties are assisted by a semi-honest entity, without relying on non-collusion assumptions.

2.4.1 The $\Pi_{\mathsf{MHE\text{-}MPC}}$ Protocol

Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be a set of N *input parties* holding respective inputs $(x_1, \ldots, x_N)$, let R be an *external receiver* and let $\mathcal{C}$ be a set of *computing parties* (which could have non-empty intersection with $\mathcal{P} \cup \{R\}$). Given a public arithmetic function f over the parties' inputs, the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol privately computes $y = f(x_1, \ldots, x_N)$ and outputs the result to R. Informally, this means that, for $\mathcal{A} \subset (\mathcal{P} \cup \mathcal{C} \cup \{R\})$ a set of corrupted parties (*the adversary*) in the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol where $|\mathcal{A} \cap \mathcal{P}| \leq N-1$, we require that the adversary does not learn anything about $\{x_i\}_{P_i \notin \mathcal{A}}$ more than that which can be learnt from its own inputs $\{x_i\}_{P_i \in \mathcal{A}}$ and, if $R \in \mathcal{A}$, from the output.

$\Pi_{\mathsf{MHE\text{-}MPC}}$ Protocol Overview The $\Pi_{\mathsf{MHE\text{-}MPC}}$protocol (Protocol 7) has two *phases*, Setup and Compute, that are themselves decomposed into several *steps*. The Setup phase instantiates the MHE scheme: Its goal is to generate the secret keys (in the SecKeyGen step, which is non-interactive, see Section 2.2.1), and to generate the necessary public keys (in the PubKeyGen step) for the rest of the protocol. In the PubKeyGen step, the parties generate the collective encryption key cpk (to be used in the Input step), and the evaluation keys (to be used in the Eval step): the relinearization key rlk, and one rotation key per rotation *amount* in the circuit (denoted $\mathsf{Rot}(f)$). For instance, if the circuit f performs (any number of) rotations by 1 and 2, then $\mathsf{Rot}(f) = \{1, 2\}$.

Note that the Setup phase is independent from the parties' inputs to the circuit f. Hence, it can be compared, to some extent, to the *offline* phase of the LSSS-based approaches (which generates the correlated randomness for the input-dependent phase). However, Setup phase of the MHE-based protocol is fundamentally different in that it produces public keys that can be used for an unlimited number of circuit evaluations. It has to be run only once for a given set of parties and for a given choice of public cryptographic parameters $pp = (n, q, \mathsf{Key}, \mathsf{Err})$. This means that the cost of Setup phase does not directly depend on the number of multiplication gates in the circuit but on the maximum circuit depth (which determines the HE parameters) and on the number of rotations in $\mathsf{Rot}(f)$ (which determines the number of necessary $\Pi_{\mathsf{RotKeyGen}}$ protocol execution).

In the Input step, the parties use the MHE.Encrypt algorithm to encrypt their inputs and to provide them to $\mathcal{C}$ for evaluation. For our construction, this corresponds to using the single-party HE.Encrypt procedure.

The Eval step consists of the evaluation of the circuit representation of f, using the MHE.Eval set of homomorphic operations. Again, our construction relies on the unmodified HE.Eval set of procedures. As this step requires no secret input from the parties, it can be performed by any semi-honest entity $\mathcal{C}$. In purely peer-to-peer settings, the parties themselves assume the role of $\mathcal{C}$, either by distributing the circuit computation, or by delegating it to one designated party. In the cloud-assisted setting, a semi-honest cloud provider can assume this role. Although it is frequent to define the role of *computing party* in current MPC applications [JWBB+17; AMP18; ABLK+18], it is usually a part of the N-party to 2-party problem reduction that introduces non-collusion assumptions. In the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, the computing parties are not required to be part of the computation data access-structure, thus removing the need for such assumptions.

The Output step enables the receiver R to obtain its output. This requires collaboration among the parties in $\mathcal{P}$ to re-encrypt the output under the key of R. This is achieved with the $\Pi_{\mathsf{PubKeySwitch}}$ protocol, which does not require online interaction between the input parties and the receiver.

Protocol 7. $\Pi_{\mathsf{MHE\text{-}MPC}}$ ▷ *The MHE-Based MPC Protocol (for External Receivers)*

Public input: f the ideal functionality, pk_R the receiver's public-key
Private input: x_i for each $P_i \in \mathcal{P}$, sk_R for R
Output for R: $y = f(x_1, x_2, \ldots, x_N)$

Setup:

The input parties in $\mathcal{P}$:

1. ($\mathsf{SecKeyGen}$) execute the secret-key generation protocol

$$\mathsf{sk}_i \leftarrow \mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}},$$

2. ($\mathsf{PubKeyGen}$) execute the required public-key generation protocols:

$$\mathsf{cpk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_N),$$

$$\mathsf{rlk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{RelinKeyGen}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_N),$$

$$\mathsf{rtk}_r \leftarrow \mathsf{MHE}.\Pi_{\mathsf{RotKeyGen}}(r, \mathsf{sk}_1, \ldots, \mathsf{sk}_N)\ \forall r \in \mathsf{Rot}(f).$$

Compute:

1. (Input) each input party in $\mathcal{P}$ encrypts its input as

$$c_i \leftarrow \mathsf{MHE.Encrypt}(\mathsf{cpk}, x_i)$$

and send c_i to $\mathcal{C}$,

2. (Eval) the computing parties in $\mathcal{C}$ compute the encrypted output as

$$c' \leftarrow \mathsf{MHE.Eval}(f, \mathsf{rlk}, c_1, c_2, \ldots, c_N)$$

and send the result c' to the parties in $\mathcal{P}$,

3. (Output) the parties in $\mathcal{P}$ re-encrypt the output under the receiver's key:

$$c'_R \leftarrow \mathsf{MHE}.\Pi_{\mathsf{PubKeySwitch}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_N, \mathsf{pk}_R, c')$$

and send c'_R to the receiver R.

2.4.2 $\Pi_{\mathsf{MHE\text{-}MPC}}$ Protocol Security

We now state the security theorem for the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol in the passive adversary model, as Theorem 2. To simplify the theorem, but without the loss of generality, we assume that the output receiver R is corrupted by the adversary. As a result, the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol output is provided to the simulator in clear text. Indeed, the opposite case (where the adversary does not know the receiver secret-key sk_R) is less relevant from a security point of view, because the simulator can simply output a uniformly random R_q^2 tuple as ct_R.

Theorem 2 ($\Pi_{\mathsf{MHE\text{-}MPC}}$ security for semi-honest model)**.** *Let $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ be a set of N parties holding private inputs $\{x_1, x_2, \ldots, x_N\}$ where x_i is held by party P_i, $f\mathcal{M}^N \to \mathcal{M}$ be a public arithmetic function. For each possible set $\mathcal{A} \subset \mathcal{P}$ of corrupted parties (the adversary) where $|\mathcal{A}| \leq N-1$, there exists simulator program $S^{\mathsf{MHE\text{-}MPC}}$ such that, for $y \leftarrow \Pi_{\mathsf{MHE\text{-}MPC}}(\{x_i\}_{P_i \in \mathcal{P}})$, it holds that*

$$S^{\mathsf{MHE\text{-}MPC}}(\{x_i\}_{P_i \in \mathcal{A}}, f(\{x_i\}_{P_i \in \mathcal{P}})) \overset{c}{\equiv} \mathit{view}_{\mathcal{A}}^{\Pi_{\mathsf{MHE\text{-}MPC}}},$$
$$f(\{x_i\}_{P_i \in \mathcal{P}}) \overset{s}{\equiv} y.$$

Proof (Intuition) Asharov et al. prove the security of MHE-based MPC protocol [AJLT+12] given a semantically secure MHE scheme, hence we only briefly provide an intuition for it here. We observe that the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol is *privately reducible* to the $\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{RelinKeyGen}}$ and $\Pi_{\mathsf{KeySwitch}}$ protocols. It privately computes its functionality f when provided with oracle access to $f_{\mathsf{EncKeyGen}}$, $f_{\mathsf{RelinKeyGen}}$ and $f_{\mathsf{KeySwitch}}$. In fact, this property directly follows from the fact that (a) these functionalities have public outputs (i.e, all query-response to the corresponding oracle can be simulated) and (b) the view of the resulting $\Pi_{\mathsf{MHE\text{-}MPC}}^{f_{\mathsf{EncKeyGen}}, f_{\mathsf{RelinKeyGen}}, f_{\mathsf{KeySwitch}}}$ protocol are valid keys and ciphertexts for the single-party scheme, yet with higher-norm secret-key and error bounds (which indeed preserves its semantic security). Then, the security of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol follows from the standalone security of each protocol (Theorem 1) by applying the composition theorem for semi-honest model [Gol09].

2.4.3 $\Pi_{\mathsf{MHE\text{-}MPC}}$ Protocol Features

In the following subsections, we discuss the properties of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, as well as the various system models these properties enable.

Public Non-interactive Circuit Evaluation

Although the homomorphic operations of HE schemes are computationally more expensive than local operations of secret-shared arithmetic, the former do not require private inputs from the parties. Hence, as long as no output or ciphertext refreshing is needed, the circuit evaluation procedure is non-interactive and can be performed by any semi-honest entity. This not only enables the evaluation to be efficiently distributed among the parties in the usual peer-to-peer setting but also enables new computation models for MPC:

Cloud-Outsourced Model The homomorphic circuit evaluation can be outsourced to a cloud-like service, by providing it with the inputs and necessary evaluation keys. The parties can arbitrarily go offline during the evaluation and reconnect for the final output step. In this model, the overhead for each input party is independent of the total number of parties. This enables even resource-constrained parties to take part in large-scale MPC tasks, that would

involve tens and even hundreds of parties. We demonstrate two instances of the cloud-assisted setting as a part of our evaluation (Sections 2.5.2 and 2.5.3).

Smart Contracts A special case of an outsourced MPC task is the execution of a smart contract over private data; this is feasible by means of the MHE-based MPC solution. In this scenario, the contract stakeholders (any party that has private input to the contract) are the MHE secret-key owners, and the smart-contract platform acts as an oblivious contract evaluator.

Public-Transcript Protocols

All the protocols of our MHE scheme construction have public transcripts, which removes the need for direct party-to-party communication. Hence, not only the Eval step, but the whole $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol can be executed over any public authenticated channel. This also brings new possibilities to designing MPC systems, which we describe below.

Efficient Network Communication Patterns The proposed protocols rely solely on the ability of the parties to publicly *disclose* their shares and to aggregate them. This gives flexibility for using efficient communication patterns: The parties can be organized in a topological way, as nodes in a tree, where each node interacts solely with its parent and children nodes. We observe that, for all the protocols, the shares are always combined by computing their sum. Hence, for a given party in our protocols, a round consists in computing its own share in the protocol, collecting and aggregating the share of each of its children and its own share, and sending the result *up the tree* to its parent (or outputting if the party is the root). Such an execution enables the parties to compute their shares in parallel and results in network traffic that is constant at each node.

By trading off some latency, the inbound traffic can be kept low by ensuring that the branching factor of the tree (i.e., the number of children per node) is manageable for each node. As the share aggregation can also be computed by any semi-honest third-party, the tree can contain nodes that are not part of $\mathcal{P}$ (i.e., nodes that would not have input in the MPC hence have no need of being part of the secret-key access-structure) and simply aggregate and forward their children's shares. We demonstrate the efficiency of the tree topology in the multiplication triple generation example benchmark, in Section 2.5.4.

Cloud-Assisted MPC Model The special case of a single root node that does not hold a share of the key can be mapped to a cloud-assisted setting where parties run the protocols interacting solely with a central node. This model complements the circuit-evaluation outsourcing feature by removing the need for synchronous and private party-to-party communication and the need for the input parties to be online and active for the protocol to progress. Hence, the cloud-assisted $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol has a clear advantage in terms of tolerance to unreliable parties, which is a significant step toward large-scale MPC. We use the cloud-assisted model for the first two example circuits of Section 2.5 and demonstrate its practicality for computations involving thousands of parties.

2.5 Performance Analysis

In order to analyze the performance of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol in both the cloud-assisted and the peer-to-peer settings, we evaluate three generic circuits over prime fields. These circuits represent common building blocks for more complex functionalities (which we briefly discuss)

Table 2.2: Experimental cryptographic parameters: Overview

Set	$\log_2 t$	$\log_2 n$	$\log_2 q$	$\log_2 w$	σ	sec. (bits)
I	32	13	218	26	3.2	128
II-A	32	14	438	110	3.2	128
II-B	16	14	438	110	3.2	128
II-C	16	15	880	180	3.2	128
III	32	13	218	55	3.2	128

and represent a comparison basis that requires no application-specific knowledge. Thus, these circuits enable a compact and reproducible comparison with a baseline generic MPC system.

In the cloud-assisted setting, we consider two example circuits: (i) A multiparty input-selection circuit and its application to multiparty private-information-retrieval (Section 2.5.2) and (ii) the element-wise product of integer vectors and its application as a simple multiparty private-set-intersection protocol (Section 2.5.3).

For both circuits, we compare the performance against a baseline system that uses an LSSS-based approach: the MP-SPDZ library implementation [Kel20] of the Overdrive protocol [KPR18] for the semi-honest, dishonest majority setting. In the peer-to-peer setting, we consider the task of generating Beaver multiplication triples (i.e., the "offline" phase of LSSS-based approaches, Section 2.5.4). We compare the performance against the SPDZ2K [CDES+18] oblivious-transfer-based and the Overdrive [KPR18] HE-based triple-generation protocols.

2.5.1 Experimental Setup and Parameters

We used our Lattigo implementation (see Chapter 4) of the multiparty BFV scheme. For the cloud-assisted setting, the client-side timings were measured on a MacBook Pro with a 3.1 GHz Intel i5 processor. The server-side timings were measured on a 2.5 GHz Intel Xeon E5-2680 v3 processor (2x12 cores). For the peer-to-peer setting, we ran all parties on the latter machine, over the localhost interface.

We measured the network-related cost in terms of the number of communicated bytes (upstream + downstream), which does not account for network-introduced delays. However, we observe that this would not advantage our solution with respect to the baseline. On the contrary, the constant and low number of rounds in the MHE-based approach makes it much less sensitive to network delays than its LSSS-based counterparts.

Each experiment represents a different circuit hence uses a different set of parameters. Therefore, we discuss the choice of parameters for each experiment. For convenience, we summarize all the parameters in Table 2.2, along with their security levels, according to the HomomorphicEncryption.org standardization document [ACCD+18].

2.5.2 Multiparty Input Selection

Setting We consider N input parties in the cloud-assisted setting. Party P_1 (the *Requester*) seeks to select one among $N-1$ bit-string inputs $x_2, \ldots, x_N$ held by other parties $P_2, \ldots, P_N$ (the *Providers*), while keeping the selector index r private. This corresponds to the ideal functionality $f(r, x_2, \ldots, x_N) = x_r$ for an *internal* receiver P_1.

This selection circuit can be seen as a generalization of an oblivious transfer functionality to the N-party setting, and can directly implement an N-party PIR system where the *requester* party retrieves a row in a database populated by the *providers*. We represent inputs as d-

Algorithm 1. $\mathsf{InputSelection}(\mathsf{rlk}, \mathsf{rtk}, \mathsf{ct}_r, \mathsf{ct}_2, ..., \mathsf{ct}_N)$

1 : **for** $i = 2...N$ **do**
2 : $\quad \mathsf{mask}_i \leftarrow \mathsf{BFV.PlainMul}(\mathsf{ct}_r, \mathbf{u}_i)$
3 : $\quad$ **for** $j = 1... \log(d)$ **do**
4 : $\quad\quad \mathsf{mask}_i \leftarrow \mathsf{BFV.Sum}(\mathsf{mask}_i, \mathsf{BFV.Rotate}(\mathsf{rtk}, \mathsf{mask}_i, 2^j))$
5 : $\quad \mathsf{ct}_{out} \leftarrow \mathsf{BFV.Sum}(\mathsf{ct}_{out}, \mathsf{BFV.Mul}(\mathsf{ct}_i, \mathsf{mask}_i))$
6 : **return** $\mathsf{BFV.Relinearize}(\mathsf{rlk}, \mathsf{ct}_{out})$

dimensional vectors in $\mathbb{Z}_p^d$ for p a 32-bit prime and d a power of two. We denote $\mathbf{u}_i$ the plaintext-space encoding of a vector in $\mathbb{Z}^N$ for which all components are equal to 0, except for the i-th component that is equal to 1.

$\Pi_{\mathsf{MHE\text{-}MPC}}$ Protocol Instantiation To realize the input selection functionality, we instantiate the $\Pi_{\mathsf{MHE\text{-}MPC}}$ as follows:

Setup: The parties run $\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{RelinKeyGen}}$ and $\Pi_{\mathsf{RotKeyGen}}$ to generate the encryption (cpk), relinearization (rlk), and the rotation keys (rtk).

Input: Each *Provider* P_i embeds its input in the coefficients of a polynomial in R_t, encrypts it using the cpk as ct_i and sends it to the cloud.
The *Requester* generates its selector as $\boldsymbol{u}_r$, encrypts it as ct_r, and sends it to the cloud.

Eval: The cloud computes $\mathsf{ct}_{out} = \mathsf{InputSelection}(\mathsf{ct}_r, \mathsf{ct}_2, ..., \mathsf{ct}_N)$ (Algorithm 1).

Output: The *Providers* execute the $\Pi_{\mathsf{KeySwitch}}$ protocol with target ciphertext ct_{out}, input key s and output key 0. By aggregating the decryption shares, the cloud computes an encryption of x_r under the *Requester* secret key. This encryption can be retrieved and decrypted by the receiver.

Parameterization We use the parameter set I in Table 2.2 for all system sizes N. This set uses a 32-bits modulus t (packing-compatible) to match the default computation domain of the baseline system [Kel20], and a modulus q supporting the depth-one $\mathsf{InputSelection}$ circuit.

Results Table 2.3 shows a comparison with the baseline system. The generation of rotation keys accounts for approximately 75% of the setup cost and is the main overhead of the protocol. For two parties, this setup takes more time and communication than the baseline's offline phase. For four parties, the MHE setup is slightly faster than the triple generation but still requires 1.7 times more communication. For eight parties, the MHE setup is 5.2× faster and requires 2.4× less communication. These results illustrate how the MHE-based solution, by having constant cost for the parties, has a lower asymptotic cost and provides significantly better scalability in practice. Moreover, comparing the MHE setup to the baseline's offline phase is valid only when considering a single, isolated circuit execution. This is because the MHE keys can be reused for an unlimited number of circuit evaluations and the cost of generating them can be amortized. When considering non-amortizable costs (**Total** and Compute in Tables 2.3), the MHE-based solution has a lower response time and a lower communication cost per party than the baseline. Moreover, the per-party communication cost of the MHE approach does not depend on N.

Table 2.3: Input selection: Baseline comparison (Set I)

		Time [s]			Com./party [MB]		
#Parties		2	4	8	2	4	8
LSSS-based (MP-SPDZ [Kel20])	Offline	0.35	1.04	3.56	6.58	25.74	101.82
	Online	0.02	0.04	0.07	1.31	4.72	17.83
	Total	**0.37**	**1.08**	**3.66**	**7.89**	**30.46**	**119.65**
MHE-based (this work)	Setup	0.59	0.58	0.69	42.93	42.93	42.93
	Compute	**0.27**	**0.28**	**0.31**	**1.31**	**1.31**	**1.31**

Table 2.4: Input selection: Cost for each phase (Set I)

	Party		Cloud		
	Time [ms]	Com. [MB]	Wall time|CPU time [s]		
#Parties	*indep.*	*indep.*	32	64	128
Setup	262.58	42.93	0.85	1.68	3.38
Input	6.22	0.52	0.01	0.01	0.02
Eval	0.00	0.00	0.4|8.1	0.8|23.4	1.6|62.1
Output	3.34	0.79	0.01	0.02	0.02

Table 2.4 shows the cost of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ phases for a larger number of parties. We were unable to instantiate the baseline system for such a large number of parties, as its cost was larger than the amount our experimental setting could handle. The parallelization of the circuit computation over multiple threads yields a very low response-time. Our choice for t enables 32.8 kilobytes of raw application data to be packed into each ciphertext (i.e., to be retrieved at each request). For the eight-party setting, this yields a plaintext throughput of 105.7 kB/s (baseline: 9.0 kB/s) and a bandwidth usage of only $40\times$ the size of an insecure plaintext system (baseline: $3650\times$). We ran the same experiment for $N = 8000$ parties; the response time was 61.7 seconds. These results show that the MHE approach can solve large MPC instances, even for resource-constrained clients, by delegating the storage and the heavy computation to a cloud.

2.5.3 Element-Wise Vector Product

Setting. We consider N input parties (with ideal secret key s) in the cloud-assisted setting. Each party holds a private integer vector $\boldsymbol{x}_i$ of dimension $d = 2^{14}$ and they all seek to provide an *external* receiver R (with secret key s_R) with the element-wise product (which we denote $\odot$) between the N private vectors. Thus, the ideal functionality is $f(\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N) = \boldsymbol{x}_1 \odot \boldsymbol{x}_2 \odot \cdots \odot \boldsymbol{x}_N = \boldsymbol{y}$ with *external receiver* R.

$\Pi_{\mathsf{MHE\text{-}MPC}}$ **Protocol Instantiation** We instantiated the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol as follows:

Setup: The parties use the $\Pi_{\mathsf{EncKeyGen}}$ and $\Pi_{\mathsf{RelinKeyGen}}$ protocols to generate the public encryption key cpk and the relinearization key rlk.

Input: Each input party $P_i \in \mathcal{P}$ encodes its input vector $\boldsymbol{x_i}$ as a polynomial x_i using *packed* plaintext encoding. Then, it encrypts this vector under the collective public key and sends $\mathsf{Enc}_s(x_i) = \mathsf{BFV.Encrypt}(\mathsf{cpk}, x_i)$ to the cloud.

Eval: The cloud computes the product by using the BFV.Mul operation (with intermediary BFV.Relinearize operations). This results in $\mathsf{Enc}_s(y)$ where y encrypts $\boldsymbol{y}$ under the collective secret-key s.

Output: The parties use the $\Pi_{\mathsf{PubKeySwitch}}$ protocol to re-encrypt $\mathsf{Enc}_s(y)$ into $\mathsf{Enc}_{s_R}(y)$.

Parameterization This is a demanding circuit, as its multiplicative depth is equal to $\lceil \log N \rceil$. Therefore, the choice of parameters depends on the number of parties. For up to 8 parties (Table 2.5), we use the parameter set II-A from Table 2.2 and compare the MHE solution against the baseline system. This set uses a 32-bits t (packing-compatible) to match the default computation domain of the baseline system [Kel20]. For up to 128 parties (Table 2.6), we use the parameter set II-B that differs from II-A in its smaller plaintext-space, which enables the circuit to have a depth of up to 9. For 1024 parties (Table 2.7), a circuit of depth 10 is required. We present two approaches to this problem: (i) Increase the size of q; this forces us to increase n to preserve the security level (parameter set II-C). (ii) Keep the same parameter set II-B and use the $\Pi_{\mathsf{ColBootstrap}}$ protocol to refresh the ciphertexts when reaching depth 9 in the circuit.

Results Table 2.5 shows the comparison with the baseline. We observe very similar results between the MHE approach and the baseline for the two-party case and a clear advantage for the former for larger numbers of parties. Table 2.6 shows the performance of the MHE approach for large numbers of parties. This demonstrates how re-balancing the cost of MPC toward computation time enables efficient multi-core processing and yields very low response times (e.g., $<$ 1 sec. of end-to-end computations for 32 parties). Finally, Table 2.7 illustrates how the $\Pi_{\mathsf{ColBootstrap}}$ protocol (used with the set II-B but not with the set II-C) introduces a trade-off between network usage and CPU usage. In this case, for an additional 4.7 MB of communication per party in the Compute phase, refreshing ciphertexts is more cost-effective (for bandwidth and CPU, by a factor between 4× and 5×) than using larger parameters, even if it requires one more communication round.

This circuit could be used, for example, as a multiparty private-set-intersection protocol for a large number of parties. In its most simple instantiation, the parties could encode their sets as binary vectors and use this functionality to compute the bit-wise AND between them. By mapping the results to this application, we can compare our circuit with the special purpose multiparty PSI protocol by Kolesnikov et al. [KMPR+17]. For the standard semi-honest model with a dishonest majority, the set size 2^{12} and 15 parties (the largest evaluated value in [KMPR+17]), the MHE solution is 1029× faster (in the LAN setting) and requires 15.3× less communication. However, our set encoding limits the application to finite sets. More advanced encodings should be investigated to match the flexibility of the approach by Kolesnikov et al.

Table 2.5: Element-wise product: Baseline comparison (Set II-A)

		Time [s]			Com./party [MB]		
#Parties		2	4	8	2	4	8
LSSS-based (MP-SPDZ [Kel20])	Offline	0.21	1.19	5.33	3.42	29.13	156.06
	Online	0.02	0.04	0.10	1.05	6.29	29.36
	Total	**0.24**	**1.24**	**5.52**	**4.47**	**35.42**	**185.42**
MHE-based (this work)	Setup	0.18	0.20	0.25	25.17	25.17	25.17
	Compute	**0.29**	**0.41**	**0.64**	**4.72**	**4.72**	**4.72**

Table 2.6: Element-wise product: Phase costs (Set II-B)

	Party		Cloud		
	Time [ms]	Com. [MB]	Wall time|CPU time [s]		
#Parties	*indep.*	*indep.*	32	64	128
Setup	96.41	25.17	0.49	0.85	1.99
Input	20.02	1.57	0.04	0.04	0.15
Eval	0.00	0.00	0.8|4.5	1.0|10.3	1.5|22.7
Output	25.38	3.15	0.05	0.10	0.21

Table 2.7: Element-wise product: $N = 1024$ parties, comparison between Set II-B (with use of the $\Pi_{\mathsf{ColBootstrap}}$ protocol during Eval) and Set II-C (non-interactive Eval)

	Party				Cloud	
	CPU Time [ms]		Com. [MB]		Wall|CPU Time [s|m]	
	II-B	II-C	II-B	II-C	II-B	II-C
Setup	110.2	467.5	25.2	121.8	13s	57s
Input	21.6	78.4	1.6	6.3	1s	3s
Eval	202.4	0.0	18.9	0.0	6s|3.8m	29s|19.2m
Output	27.2	107.5	3.1	12.6	1.2s	4.3s

2.5.4 Multiplication Triples Generation

In a peer-to-peer setting, we apply the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol to LSSS multiplication-triples generation. We compare the performance against the SPDZ2K [CDES+18] oblivious-transfer-based and the Overdrive [KPR18] HE-based triple-generation protocols. We used the *Multi-Protocol SPDZ* library [Kel20] implementation of SPDZ2K (in semi-honest mode) and implemented the HE and MHE approaches with Lattigo.

Setting We consider N parties that seek to generate multiplication triples in a peer-to-peer setting. They use the tree-based communication pattern described in Section 2.4.3. Let $\mathbf{x_i} = (\mathbf{a_i}, \mathbf{b_i}) \in \mathbb{Z}_p^{n\times 2}$ be the input of party P_i, where n is the number of generated triples and p is a prime. The ideal functionality for each party P_i is $f_i(\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_N}) = \mathbf{c_i}$ such that $\sum_{i=1}^{N} \mathbf{c_i} = (\sum_{i=1}^{N} \mathbf{a_i}) \odot (\sum_{i=1}^{N} \mathbf{b_i}) = \mathbf{a} \odot \mathbf{b}$.

$\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol instantiation To realize the multiplication-triple-generation functionality, we instantiate the $\Pi_{\mathsf{MHE\text{-}MPC}}$ as follows:

Setup The parties run the $\Pi_{\mathsf{RelinKeyGen}}$ protocol to generate a the rlk.

Input: The parties use the $\Pi_{\mathsf{Share2Enc}}$ protocol to obtain encryptions of $\mathbf{a}$ and $\mathbf{b}$. Hence, the root node holds $\mathsf{ct}_a = \mathsf{Enc}(\mathbf{a})$ and $\mathsf{ct}_b = \mathsf{Enc}(\mathbf{b})$.

Eval: The root computes $\mathsf{ct}_c = \mathsf{BFV.Relinearize}(\mathsf{rlk}, \mathsf{BFV.Mul}(\mathsf{ct}_a, \mathsf{ct}_b))$ and sends ct_c down the tree.

Output: The parties use the $\Pi_{\mathsf{Enc2Share}}$ protocol to obtain an additive sharing of $\mathbf{c}$ from $\mathsf{Enc}(\mathbf{c})$.

Parameterization We target the 32-bit integers as our LSSS-computation domain, hence we set t as a 32-bit prime (parameter set III for the HE and MHE methods). The OT-based generator produces $\mathbb{Z}_{2^{32}}$ triples[4].

Results Figure 2.1 plots the results for the three techniques, with a varying number of parties. To report on the steady regime of the systems, we do not include the Setup phase costs of all methods in the measurements. After the MHE setup phase, the parties can loop over the Input-Eval-Output steps to produce a stream of triples in batches of $n = 2^{13}$. Except for the two-party throughput, the MHE approach outperforms the HE-based and OT-based approaches.

[4]At the time of writing, MP-SPDZ does not implement a benchmark for the OT-based triple-generation in a prime field.

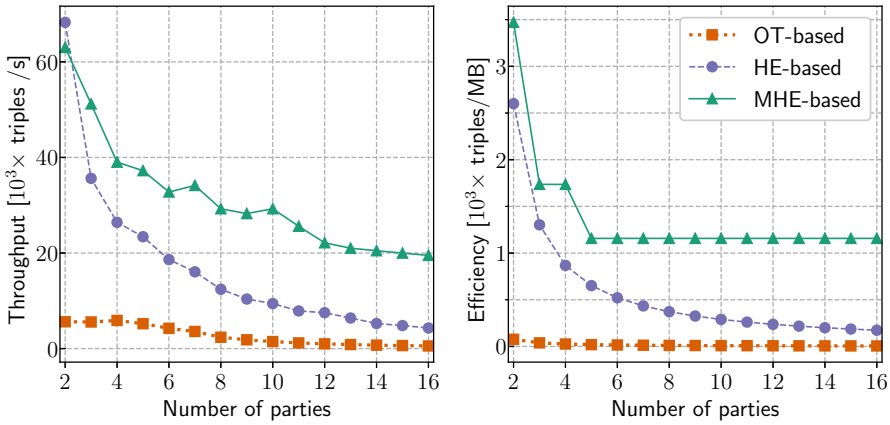

Figure 2.1: Number of generated triples per second (throughput, left) and per megabyte of communication (efficiency, right).

2.5.5 Discussion

We observe that the main cost of MHE-based solutions is the network load of their setup phase, primarily due to the generation of evaluation keys (e.g., relinearization, rotation). Hence, in scenarios with a single evaluation of a circuit with few multiplication gates and a small number of input parties, the MHE-based solution would not be as efficient as an LSSS-based approach that generates triples *on-the-fly*. However, as the MHE setup is performed only once, it is quickly amortized when considering circuits with a few thousand multiplication gates and with more than two parties; in this scenario, the cost of the LSSS-based approach is dominated by the generation of multiplication triples.

2.6 Chapter Summary

In this chapter, we introduced a N-out-of-N-threshold MHE scheme based on RLWE. The construction improves on the previous work by Asharov et al. by porting it to RLWE, by reducing the cost and increasing the precision of the relinearization key generation, by enabling external computation receivers, and by proposing a protocol to refresh ciphertexts. The construction is also generic and can be instantiated with most of the commonly implemented schemes: BFV, BGV and CKKS. We observed that the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol has a public-transcript, which permits computation models for MPC beyond the traditional peer-to-peer model. Cloud-assisted models are key to scaling MPC to large sets of parties, as they enable constant overhead for the parties without relying on non-collusion assumptions.

One limitation of our proposed scheme, in the context of large-scale MPC, would be its N-out-of-N-threshold access-structure. As the number of parties in a system increases, so does the probability of temporary disconnections or crashes. It is common to adapt the security requirements of such large systems by relaxing the access structure to a T-out-of-N-threshold one for $T < N$. In the following chapter, we show how to extend our construction to this corruption model.

Chapter 3

A Fault-Tolerant Multiparty Homomorphic Encryption Scheme

Chapter Content

In this chapter, we address the shortcomings of the MHE-based MPC protocol, presented in Chapter 2, in terms of fault tolerance. More specifically, we want to enable parties in this protocol to pursue its execution even in the presence of failing or unresponsive parties, as long as a *threshold* number $T \leq N$ of parties are online and responsive. This is generally achieved by employing Shamir's secret-sharing scheme [Sha79] on the parties' secrets. Informally, the parties represent their secrets as the constant element of a degree-$T-1$ polynomial in some field, and they distribute the shares of the secret as N evaluations of this polynomial at N different points. Then, a secret is recovered by reconstructing the polynomial through the interpolation of at least T shares. However, applying the Shamir secret-sharing to the RLWE-based MHE scheme and its associated MPC protocol presents some unique challenges.

Existing constructions, which we review in Section 3.2, either leak the failing parties' share of the ideal secret-key (i.e., it permanently alters the scheme's access structure) [AJLT+12], or require that a choice be made between non-compact party-states or non-compact ciphertexts [BGGJ+18], with both options resulting in a significant overhead in practice. The underlying reason for these shortcomings is noise: Recall that MHE protocols compute values of the form $h = as + e \in R_q$ where $s = \sum_{i=1}^{N} s_i$ is the sum of the parties' secret-keys and $e \ll q$ is small noise term by (i) having each party P_i disclose a public share of the form $h_i = as_i + e_i$ and (ii) adding all the public shares together. To avoid reconstructing the failing parties' secret shares s_i, the parties have to perform the Shamir reconstruction *within* the MHE protocols. In the literature, this is generally done by performing the interpolation directly over the non-failing parties' public share, by exploiting the linearity both of Shamir secret-sharing and the protocol [Ped91]. Unfortunately, this is not possible for MHE public shares h_i, for which the linear relation to the desired term h is only approximate (i.e., noisy): their multiplication with the interpolation coefficients (which may be large) would result in an error term e that is no longer small with respect to q. Hence, applying Shamir secret-sharing in our setting requires other techniques.

3.1 Our Results

We propose an efficient MHE scheme that tolerates temporary disconnection and/or failures from $T-1 < N$ of the parties. We formulate this scheme as an extension to the N-out-of-N-threshold scheme presented in Chapter 2 by *relaxing* its N-out-of-N-threshold access-structure to a T-out-of-N-*threshold* one. We follow the approach of *re-sharing* the additive secret-key shares with the Shamir secret-sharing scheme [Sha79], but with a specially adapted instance of the Shamir secret-sharing that we define over the ciphertext-space ring. Then, due to the linearity of the N-out-of-N-threshold MHE scheme's secret-key operations (e.g., threshold decryption), we obtain a compact and efficient scheme. Notably, we show that the re-shares can be pre-aggregated, thus resulting in a constant-size party state, and that the T-out-of-N secret-key-reconstruction can be performed efficiently *within* the secure MHE protocols themselves, i.e., without leaking any of the failing party's secret-key. We show that, in the synchronous setting, this requires a simple non-interactive pre-computation to the corresponding operation in the N-out-of-N scheme, yet performed among $N = T$ parties. As for the MHE scheme, our construction applies to the core RLWE encryption scheme and can therefore be used to instantiate T-out-of-N-threshold variants of the BGV, BFV, and CKKS front-end schemes.

We implemented our constructions in Lattigo [MBTH20], our open-source library for multiparty homomorphic encryption, which we introduce in Chapter 4. We report on the benchmark performance for our implementation and analyze the results in the context of MHE-based MPC. Furthermore, we show how to harness the T-out-of-N-threshold access-structure to accelerate

the execution of *batches* of secret-key operations in both the offline (Setup) and online (Compute) phases. We exemplify this through an application case-study: the end-to-end-encrypted federated neural network training of Sav et al. [SPTF+21].

This chapter is organized as follows: We review the existing works on threshold encryption for lattice-based HE construction in Section 3.2, and we provide the system model and some background on secret-sharing techniques in Section 3.3. Then, we develop the main technique, in Section 3.4, and its implementation and evaluation, in Section 3.5.

3.2 Related Work

In this work, our main object of study is a threshold encryption-scheme, an important part of which is the distributed key-generation (DKG) procedures. DKG techniques have been extensively studied for discrete-log-based (DL) constructions [Ped91; GJKR99; CKLS02; KG09; KG21], but their lattice-based counterparts have received less attention. However, they present their own set of challenges, especially when considering HE constructions: Notably, these challenges include the aforementioned issue of managing the noise and the fact that (M)HE constructions are expressed over highly structured rings. Also, from the practical perspective, the fact that (M)HE applications often require many public keys for their Eval algorithm calls for highly efficient solutions. Our work takes inspiration from DL-based DKG techniques and addresses these challenges.

Bendlin and Damgård considered the case where the parties obtain Shamir secret-shares of a secret key by means of pseudo-random secret-sharing (PRSS) techniques [BD10]. This results in a non-interactive secret-key-generation procedure, but it is non-compact as it requires one key per possible subset of adversarial parties. Due to this factorial expansion, this scheme would not be practical for a large number of parties.

Asharov et al. noticed that share-re-sharing could be used to achieve a T-out-of-N-threshold access structure in (the extended version of) their seminal work on LWE-based multiparty homomorphic encryption [AJLT+12]. However, they did not specify the concrete secret-sharing scheme and assumed an extra round of interaction, prior to the decryption round, to reconstruct a failing party's share. This solution, however, has shortcomings. For example, reconstructing the share of a party crashing during the Setup phase would require the input to be provided under weaker security (a $T-1$-out-of-N-threshold access-structure). Similarly, reconstructing a share during the Compute phase would prevent further iteration of this phase (as enabled by the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, see Section 2.4.3) under the same security assumptions. Instead, it would be preferable to let parties disconnect and reconnect without compromising their shares. Our work provides this capability, by performing the reconstruction *within* the secure decryption protocol directly.

Boneh et al. propose a T-out-of-N-threshold HE scheme based on learning-with-errors that also relies on re-sharing the secret-key shares, yet in a stronger *asynchronous* setting where parties are unable to determine, at the time of generating their decryption shares, which other parties are online [BGGJ+18]. This additional constraint is necessary for the composability of their scheme that they use as a building-block for higher-level cryptographic primitives in their work. However, it comes with a significant complexity and performance overhead, and their setup phase requires a trusted dealer to perform the sharing. In their work, Boneh et al. observe that enabling the parties to determine which other parties are online, before the decryption phase, would lead to a simpler scheme. We confirm this observation by showing that, in the semi-honest model with failures, there indeed exists a simpler, more compact and more efficient scheme that

does not require a trusted dealer. We elaborate on these differences in Section 3.5.1, where we provide a comparison between their construction and our scheme.

Concurrently to our work, Urban and Rambaud propose an alternative MHE-based MPC approach that provides guaranteed output delivery while minimizing the number of synchronous rounds needed in the setup phase and requiring no synchronous communication during the evaluation phase. Their approach is also based on a linear secret-sharing scheme over RLWE rings [UR22], but their construction targets generality rather than efficiency, as it enables the FHE coefficient modulus to be a composite with factors that are smaller than the number of parties. Our construction targets efficiency for the parameterization supported by the current FHE implementations, for which the structure of the coefficient modulus is already constrained for efficiency reasons.

3.3 Preliminaries

First, we present our system and adversary model, as well as the main system goals. Then, we present the main building blocks of our solution.

3.3.1 Adversary Model and System Goals

We consider a set $\mathcal{P}$ of N parties $\{P_1, ..., P_N\}$ (*the system*) in a secure-multiparty-computation setting, where an adversary $\mathcal{A}$ is able to corrupt up to $T-1$ parties. We assume that the adversary is static and passive, yet we further allow the adversary to take the corrupted parties offline for an arbitrary amount of time. The parties can communicate through private authenticated channels and through a public, synchronous, authenticated channel. Finally, and as for the previous construction in Chapter 2, we assume that the parties have access to a public common random string (CRS).

System Goals Let x_i be the private input of party P_i in some message space $\mathcal{M}$, let $f : \mathcal{M}^N \to \mathcal{M}$ be a public arithmetic function over the message space, and let λ be a security parameter. We formulate the following system goals:

- *Functionality.* The system must compute $y = f(x_1, ..., x_N)$ through a multiparty protocol.
- *Privacy.* There must exist a simulator program SIM_f that can simulate all the interactions between the parties (the *transcript*), when provided only with the output y and the inputs from the adversary. For an attacker to distinguish between the real and simulated interaction, the success probability must be a negligible function in the security parameter λ.
- *Fault Tolerance.* After the inputs are received for all parties, the output y should be delivered to the honest parties, as long as at least T parties are online and active.

Informally, the protocol execution should not reveal anything more about the inputs than that which can be deduced from the output y alone. We also observe that the fault-tolerance requirement, *guaranteed output-delivery*, is limited to the case where faulty parties eventually provide their inputs. This is because not all functions can be successfully computed under partial inputs.

The MHE-MPC (Section 2.4) protocol naturally provides *some* fault tolerance against parties going offline for a finite amount of time. As opposed to its LSSS-based counterparts, a party that goes offline after providing its inputs does not prevent the computation from making progress, as the homomorphic evaluation can be performed non-interactively. The same is true for a party

that momentarily goes offline after the Setup phase, except that, similarly to the plaintext case, the party's input are not available to the computation (whereas parties in an LSSS protocol based on multiplication hold shares of these triples, hence need to be online). In both cases, the main drawback is that all parties need to connect *eventually* (to participate in the decryption protocol of the output step) for the output to be delivered. This might be problematic in settings where a group of parties seek to tolerate a fraction of them going offline for an undetermined amount of time. In our construction, we use the Shamir secret-sharing scheme [Sha79] to solve this problem.

3.3.2 Shamir Secret-Sharing

We recall the secret-sharing scheme of Shamir that implements a T-out-of-N-threshold access-structure on its secrets, based on polynomial interpolation in a finite field [Sha79]. For the sake of notation, but without the loss of generality, we consider the reconstruction from the first T shares. Indeed, the procedure generalizes to any set of at least T shares.

- Shamir.Setup: The parties agree on a field K and each party $P_i \in \mathcal{P}$ is associated with a non-zero element $\alpha_i \in K$ such that if $i \neq j$ then $\alpha_i \neq \alpha_j$.
- Shamir.Share(s, T, $\alpha_1, ..., \alpha_N$): To share a message $s \in K$ among N parties such that T shares are needed to reconstruct s, sample $c_1, ..., c_{T-1} \leftarrow K$ and send $s_i = s + \sum_{k=1}^{T-1} c_k \alpha_i^k$ to party P_i.
- Shamir.Combine($s_1, ..., s_T$, $\alpha_1, ..., \alpha_T$): To reconstruct s from shares $s_1, ..., s_T$, compute

$$s = \sum_{i=1}^{T} s_i \prod_{j=1, j\neq i}^{T} \frac{\alpha_j}{\alpha_j - \alpha_i}. \tag{3.1}$$

We observe that the Shamir.Share procedure samples a degree-$(T-1)$ polynomial $S(X) \in K[X]$ such that $S(0) = s$ and distributes $S(\alpha_i)$ to party P_i, and the Shamir.Combine procedure computes the Lagrange interpolation at point $X = 0$ to reconstruct the secret. We refer to the sequence of public points $(\alpha_1, ..., \alpha_N)$ as the *Shamir public-points*.

3.4 T-out-of-N-Threshold Encryption for RLWE

Here, we present the main contribution of this chapter. We provide an overview of the main ideas behind the scheme in Section 3.4.1. Then, we present the secret-sharing scheme that we use for the share re-sharing in Section 3.4.2. Finally, we present our T-out-of-N-Threshold Encryption for RLWE in Section 3.4.

3.4.1 Overview

We start from the well-known *share re-sharing* approach that is to apply the Shamir secret-sharing scheme to the additive shares of the *ideal secret-key* s of the MHE scheme. Intuitively, this technique enables any set of at least T parties to reconstruct the shares of the missing parties and to take their place in the decryption procedure. However, a naive instantiation of this idea, over an arbitrary secret-sharing space, would be inefficient: It would require the non-failing parties to either reconstruct the shares of the failing parties (which would forever remove them from the access structure and add a communication round) or to compute their shares by running

a secure computation over the secret-sharing space (which would require costly emulation of the MHE scheme ciphertext space over the secret-sharing space). Also, it would require each party to store all N re-shares throughout the entire protocol, resulting in a non-constant-size state.

Instead, we perform the Shamir re-sharing directly over the MHE cipephertext-space ring R_q (See Section 1.4.1). In this way, we can exploit the linearity of both the ideal secret-key and the re-sharing scheme to obtain a more compact and communication-efficient scheme. More specifically, assuming R_q is our Shamir secret-sharing space, we denote $S_i \in R_q[X]$ the secret degree-$(T-1)$ polynomial sampled by party P_i during the Shamir.Share procedure, and $\lambda_i = \prod_{j=1, j\neq i}^{T} \frac{\alpha_j}{\alpha_j - \alpha_i}$ be the i-th Lagrange coefficient in the reconstruction using the Shamir public-points $\alpha_1, ..., \alpha_T$. Then, we observe that the Shamir.Combine operation commutes with the ideal-secret-key reconstruction:

$$s = \sum_{i=1}^{N} s_i = \sum_{i=1}^{N} \sum_{j=1}^{T} S_i(\alpha_j)\lambda_j = \sum_{j=1}^{T} \lambda_j \sum_{i=1}^{N} S_i(\alpha_j) = \sum_{j=1}^{T} s'_j. \tag{3.2}$$

This presents several challenges and opportunities for our construction; we outline them below, as Remarks 1 to 3.

Remark 1. The Shamir secret-sharing scheme is usually defined over an arbitrary field that guarantees the correctness and security of the Lagrange interpolation for enforcing the access structure. However, there are no such guarantees over arbitrary rings. For Equation (3.2) to hold and the resulting scheme to be secure, we need to show that these properties hold in the ring R_q.

Remark 2. From Equation (3.2), we observe that the new sharing over T parties has an additive structure for which the j-th term can be locally (pre-)computed by each $P_j \in \mathcal{P}_t$, if the set of parties that are participating in the secret-key operation is known.

Remark 3. The right-hand side of Equation (3.2) can be seen as a new T-out-of-T additive sharing of s and can simply be used by the parties instead of s_i (their N-out-of-N counterpart) in the MHE key-generation and decryption protocols.

We present the concrete Shamir secret-sharing scheme in Section 3.4.2 and show that it satisfies the requirements of a secret-sharing scheme (as per Remark 1). Then, we present our T-out-of-N-threshold scheme; we can formulate it as a direct extension of the N-out-of-N-threshold MHE scheme for RLWE, which only requires a constant-size additional state for each party (due to Remarks 2 and 3).

3.4.2 Shamir Secret-Sharing in R_q

The usual Shamir secret-sharing scheme is instantiated over a field. This guarantees that all non-zero elements are units, hence that Lagrange coefficients exist. Indeed, computing a Lagrange coefficient requires inverting elements of the form $\alpha_i - \alpha_j$, where α_i and α_j are the Shamir public-points. However, working in a field is not a requirement. In fact, it is a known result that using a ring is possible, as long as the set of Shamir public-points forms an *exceptional sequence* [ACDE+19; CDN15]. Here, we briefly present this result in our notation and terminology.

Definition 1. (From [ACDE+19]) For a ring R, the sequence $\alpha_1, ..., \alpha_N$ of elements of R is an *exceptional sequence* if $\alpha_i - \alpha_j$ is a unit in R for all $i \neq j$.

Theorem 3. *(From [ACDE+19]) Let R be a commutative ring and $\alpha_1, ..., \alpha_N$ be an exceptional sequence in R. Then, a Shamir secret-sharing scheme instantiated in R with Shamir public-points, $\alpha_1, ..., \alpha_N$, is correct and secure.*

Let us assume that $\alpha_1, ..., \alpha_N$ is an exceptional sequence for R_q. Then, by instantiating a T-out-of-N Shamir secret-sharing scheme that uses the elements of this exceptional sequence as the Shamir public-points, we obtain from Theorem 3 that our secret-sharing scheme for R_q is correct and secure for a threshold access-structure. Hence, we now define how to choose our Shamir public-points from R_q in such a way that guarantees an exceptional sequence and enables a highly efficient implementation.

Choice of Shamir public-points We first observe that checking whether an arbitrary sequence of R_q elements forms an exceptional sequence is easy: For each non-zero pairwise difference, it suffices to check that all coefficients of the difference polynomial under the CRT and NTT representation are non-zero. This holds because the inverse of each non-zero coefficient can be computed individually by Fermat's little theorem. However, computing these inverses for arbitrary elements of R_q is a costly operation that would result in an inefficient $\mathsf{Combine}$ operation.

Instead, we propose to restrict the choice of Shamir public-points to constant polynomials in $R_q = \mathbb{Z}_q[X]/(X^n + 1)$ (i.e., a monomial of the form αX^0 for $\alpha \in \mathbb{Z}_q^*$). On the one hand, it yields a significant performance boost, as computing the Lagrange coefficient now only requires scalar multiplications in $\mathbb{Z}_q$. On the other hand, this provides us with a simple procedure for choosing Shamir public-points that guarantee an exceptional sequence: Let $q_{min} = \mathsf{min}(q_1, ..., q_L)$ with $q_1, ..., q_L$ the prime factors of q. We observe that for $N < q_{min}$, choosing N distinct values in $\mathbb{Z}_{q_{min}}$ as the Shamir public-points will guarantee an exceptional sequence. Indeed, for any $i \neq j$, $-q_{min} < \alpha_i - \alpha_j < q_{min}$, $\alpha_i - \alpha_j \neq 0$ and the residue $\mod q_k$ is non-zero for any prime factor q_k of q. Then, a simple application of the CRT on R_q is enough to prove that $\alpha_i - \alpha_j$ is a unit in R_q. Therefore, any mapping from $\mathcal{P}$ onto $\mathbb{Z}_{q_{min}}$ can be used, including the *textbook* one that commonly uses i for party P_i, if i is a positive integer. We observe that it is critical for implementations to check that Shamir public-points are non-zero.

Choosing the Shamir public points from the restricted set has the side effect of limiting the number of parties to $q_{min} - 1$. But this is not an issue in most cases, because the factors of q are already constrained by the encryption scheme's parameterization requirements: They have to be primes congruent to $1 \mod 2n$ where n is the degree of the ring (which is typically larger than 2^{11} in the FHE setting). However, this could be a limitation in a setting where parties independently and randomly sample their own public points, as the probability of a collision would be too high. For such use-cases it might be preferable to sample points in $\mathbb{Z}_q$ where the probability of collision is negligible, then check that the sequence forms an exceptional sequence, which occurs with high probability.

3.4.3 The Share Re-sharing Scheme

For a set of parties $\mathcal{P}$ in the MHE scheme where $P_i \in \mathcal{P}$ holds secret-key share s_i, we define our re-sharing scheme as the three-tuple of procedures $\mathsf{T} = (\mathsf{Setup}, \mathsf{Thresholdize}, \mathsf{GetAdditiveShare})$. Intuitively, Scheme T applies the Shamir secret-sharing scheme over R_q introduced in Section 3.4.2 to the parties' key, which *relaxes* the N-out-of-N access-structure of the MHE scheme of Section 2.2 to a T-out-of-N-threshold one.

We observe that the output of the $\mathsf{T.Thresholdize}$ is only one ring element per party, due to the re-share being aggregatable. This is because the summation in N on the right-hand side of

Equation (3.2) does not depend on which T of the N parties participate in the reconstruction and can be pre-computed by each party P_i, after it receives all the $S_j(\alpha_i)$ from its peers.

We also observe that only the $\mathsf{T.Thresholdize}$ procedure is interactive and that it requires a single round of pairwise interactions between the parties over confidential channels. Once performed, the parties have access to Shamir shares $(\tilde{s}_1, ..., \tilde{s}_N)$, from which each party P_i can locally compute its share s'_i in an additive sharing $(s'_1, ...s'_T)$ of s among any subgroup of at least T parties in $\mathcal{P}$ (as per remark 2). Consequently, each party P_i can simply use its new share s'_i directly in the MHE procedures. This is the main idea for our next construction.

3.4.4 The T-out-of-N-Threshold MHE scheme

We propose our construction as the union tuple $\mathsf{T} \cup \mathsf{MHE}$, that provides a T-out-of-N-threshold encryption scheme. We detail this construction as the TMHE scheme (Scheme 4). For the sake of conciseness, we omit the $\Pi_{\mathsf{RelinKeyGen}}$ and $\Pi_{\mathsf{RotKeyGen}}$ protocols, as they are straightforward adaptations of the $\Pi_{\mathsf{EncKeyGen}}$ protocol. Similarly, we consider the Π_{Decrypt} special case of the $\Pi_{\mathsf{KeySwitch}}$ protocol, but the construction and discussion also apply to the latter and to its public-key variant $\Pi_{\mathsf{PubKeySwitch}}$. As per Remark 2, the $\mathsf{T.GetAdditiveShare}$ procedure requires the parties to obtain the set of participating parties from the environment. We formalize this requirement by providing the parties with an oracle access to the set of online parties. We denote $\mathcal{P}_{online} \leftarrow \mathsf{Env}$ such an oracle query where $\mathcal{P}_{online} \subseteq \mathcal{P}$ is the set of online parties at the time the environment is queried. We assume that the oracle returns the same set to all parties for a given secret-key operation. However, we do not assume this across different secret-key operations, and the set of parties performing the setup could differ from the set performing decryption. Indeed, as per Equation 3.2, any set of at least T parties can reconstruct s. In our (synchronous) model, this oracle can be realized with a simple broadcast round of communication to gather the identities of online parties, yet with the small caveat that, after this broadcast round, the parties might fail. In Section 3.4.5, we discuss how to deal with faulty oracles that return an incorrect set of online parties.

TMHE-based MPC protocol The instantiation of an MPC protocol from our scheme is the same as for the MHE scheme, yet it satisfies the *fault tolerance* requirement of Section 3.3.1. It tolerates up to $N - T$ parties going offline for an undetermined amount of time, as long as the failing parties completed the $\mathsf{TMHE}.\Pi_{\mathsf{SecKeyGen}}$ procedure and provided their encrypted inputs to the computation. We elaborate on the differences between the TMHE and MHE instantiations in Section 3.5.1.

3.4.5 Dealing with Faulty Oracles

Our model does not exclude the possibility of a party crashing after the oracle response. In such a case, step 4 of the $\mathsf{TMHE}.\Pi_{\mathsf{Decrypt}}$ cannot be completed due to missing share(s) in the disclosure step of the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ protocol. In practice, such a failure is generally detected and resolved by setting a time limit (timeout) for the parties to provide their decryption shares, and by defining the parties' behaviour in the case of such timeouts. Whereas the exact values for the timeout are indeed application dependent, we now discuss how parties can react to such timeouts to guarantee the eventual decryption of a ciphertext in a secure way.

Let $\mathcal{P}_{timeout}$ be the set of parties that did not provide their share in time during a secret-key operation; a partial yet insecure solution is to repeat steps 3 and 4 of the operation, with $\mathcal{P}'_{online} \leftarrow \mathcal{P}_{online} \setminus \mathcal{P}_{timeout}$ where $\setminus$ denotes the set difference. As such, this solution is insecure,

Scheme 3: T ▷ *The share re-sharing scheme*

$\mathsf{T.Setup}$:

1. Each party $P_i \in \mathcal{P}$ is associated with a public point $\alpha_i \in R_q$ such that $\alpha_i - \alpha_j$ is a unit for all i, j, $i \neq j$.

$\mathsf{T.Thresholdize}(T, s_1, ..., s_N, \alpha_1, ..., \alpha_N)$:

1. Each party P_i samples $c_{i,1}, ..., c_{i,T-1} \leftarrow R_q$.
2. Each party P_i sends $\tilde{s}_{i,j} = s_i + \sum_{k=1}^{T-1} c_{i,k}\alpha_j^k$ to each party P_j.
3. Each party P_i receives $\tilde{s}_{j,i}$ from each party P_j and computes:

$$\tilde{s}_i = \sum_{j=1}^{N} \tilde{s}_{j,i}.$$

$\mathsf{T.GetAdditiveShare}(\tilde{s}_1, ..., \tilde{s}_T, \alpha_1, ..., \alpha_T)$:

1. For $\mathcal{P}' \subseteq \mathcal{P}$, $|\mathcal{P}'| \geq T$, each party $P_i \in \mathcal{P}'$ computes

$$s_i' = \tilde{s}_i \prod_{j=1, i\neq j}^{T} \frac{\alpha_j}{\alpha_j - \alpha_i}.$$

Scheme 4: TMHE ▷ *The T-out-of-N-threshold HE scheme*

$\mathsf{TMHE.Setup}$:

Run the $\mathsf{T.Setup}$ procedure.

$\mathsf{TMHE.}\Pi_{\mathsf{SecKeyGen}}$:

1. Run $(s_1, ..., s_N) \leftarrow \mathsf{MHE.}\Pi_{\mathsf{SecKeyGen}}$.
2. Run $\mathsf{T.Thresholdize}(t, s_1, ..., s_N, \alpha_1, ..., \alpha_N)$.

$\mathsf{TMHE.}\Pi_{\mathsf{EncKeyGen}}$:

1. Otain $\mathcal{P}_{online} \leftarrow \mathsf{Env}$ and
2. if $|\mathcal{P}_{online}| < T$, return $\perp$.
3. Choose T parties $\mathcal{P}_{online}$ and run $(s_1', ..., s_T') \leftarrow \mathsf{T.GetAdditiveShare}$.
4. Execute the $\mathsf{MHE.}\Pi_{\mathsf{EncKeyGen}}(s_1', ..., s_T')$ protocol.

$\mathsf{TMHE.}\Pi_{\mathsf{Decrypt}}(\mathsf{ct})$:

1. Obtain $\mathcal{P}_{online} \leftarrow \mathsf{Env}$ and
2. if $|\mathcal{P}_{online}| < T$, return $\perp$.
3. Choose T parties $\mathcal{P}_{online}$ and run $(s_1', ..., s_T') \leftarrow \mathsf{T.GetAdditiveShare}$.
4. Execute the $\mathsf{MHE.}\Pi_{\mathsf{Decrypt}}(\mathsf{ct}, s_1', ..., s_T')$ protocol.

because the underlying $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ procedure is not secure under the composition of several decryptions of the same ciphertext $\mathsf{ct} = (c_0, c_1)$ (informally, $(sc_1 + e_1,\ sc_1 + e_2)$ leaks information about sc_1 when e_1 and e_2 are sampled independently). However, the key observation is that obtaining a new ciphertext ct' such that $\mathsf{Decrypt}(\mathsf{ct}) = \mathsf{Decrypt}(\mathsf{ct}')$ is easy with any asymmetric additive HE scheme. Hence, our solution is to operate a re-randomization step, by adding a fresh encryption of zero to the target ciphertext before repeating the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ step.

3.4.6 Accelerating Batched Multiparty Secret-Key Operations

The T-out-of-N-Threshold access-structure also enables the group of key-share holders to efficiently parallelize batches of secret-key operations, when more than T participants are online. Performing batches of secret-key operations is common in MHE-based MPC protocols:

- at the Setup phase - when the parties have to generate a number of key-switching keys to support non-linear operations such as ciphertext-ciphertext multiplication (the relinearization key) and ciphertext-slot rotations (the rotation keys).
- at the Eval step - if the parties rely on interactive protocols to reduce the noise or to raise the level of ciphertexts as a part of the circuit in order to avoid the overhead of using bootstrapping [MTBH21; SPTF+21]. These protocols can be abstracted as performing a masked decryption and a re-encryption, hence are secret-key operations.
- at the Output step - when the function's output consists of multiple ciphertexts. This could be by design (of the ideal functionality), or because the encryption parameters do not enable packing enough values in one ciphertext.

Let k be the number of secret-key operations to be performed (e.g., the number of rotation keys to be generated), and let $\mathcal{P}_{online}$ be the set of online parties. The parties in $\mathcal{P}_{online}$ can be organized into k subgroups of T distinct parties, and the work can be distributed among the subgroups. As observed in the previous chapter, the cost of running each MHE secret-key operation protocol within each subgroup of size T can be made constant for each party, by relying on one or multiple aggregators (e.g., in a tree-like topology) [MTBH21]. Hence, the total overhead for each party in performing the k secret-key operations can be reduced to $(kT)/|\mathcal{P}_{online}|$, which is $T/|\mathcal{P}_{online}|$ times the overhead of performing these same k operations in the N-out-of-N-threshold scheme. In Section 3.5.3, we evaluate the effect of using this technique in the setup and in the evaluation phase of a concrete instance of the $\mathsf{MHE\text{-}MPC}$ protocol: the federated neural network training algorithm of Sav et al. [SPTF+21].

3.5 Evaluation

Here, we discuss our proposed construction from the theoretical and practical standpoints.

3.5.1 Theoretical Evaluation

We first study the overhead and additional assumptions of the threshold scheme, with respect to the original MHE scheme. Then, we discuss the main differences between the threshold scheme of Boneh et al. and our proposed construction.

Comparison with the Base MHE Scheme From the system-model standpoint, the main difference between the TMHE scheme, and the base MHE scheme is indeed that our construction enables T-out-of-N access structures. Hence, instantiating the MHE-based MPC protocol with our scheme satisfies the *fault tolerance* requirement of Section 3.3.1. Moreover, the TMHE-based

Table 3.1: Threshold extension costs, measured in the number of R_q elements per party for the internal state and network communication, and in the asymptotic function of N and T for the per-party computational cost. We distinguish between the costs associated with the generation ($\mathsf{SecKeyGen}$) operation and the usage ($\mathsf{SecKeyOp} \in \{\mathsf{PubKeyGen}, \mathsf{Decrypt}\}$) of the secret-key.

	Party's state		Network cost per party		Comp. cost	
	SecKeyGen	SecKeyOp	SecKeyGen	SecKeyOp	SecKeyGen	SecKeyOp
MHE	1	1	0	1	$\mathcal{O}(1)$	$\mathcal{O}(1)$
TMHE	T	1	$2(N-1)$	1	$\mathcal{O}(T+NT+N)$	$\mathcal{O}(T)$

instantiation retains most of the features from the MHE-based one: (a) Its *offline* phase is reusable and has to be performed only once for a given set of parties and encryption parameters. (b) Its *online* phase has a fully public transcript and consists of only two rounds of interaction among the parties. However, the $\mathsf{TMHE}.\Pi_{\mathsf{SecKeyGen}}$ protocol relies on confidential communication channels between the parties (to execute the T.Thresholdize re-sharing procedure), which is not the case for the original $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ procedure. As a result, the TMHE-based $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol does not have a fully public transcript in its *offline* phase, whereas the MHE-based one does. However, private communication is required for only a single asynchronous round of communication, hence is not a major obstacle in many peer-to-peer and cloud-assisted models.

From the computational-cost standpoint, the threshold extension requires an additional state to be stored and exchanged by each party. We summarize the related costs in Table 3.1. The $\mathsf{TMHE}.\Pi_{\mathsf{SecKeyGen}}$ is the only operation where this overhead is not negligible: It requires each party to store a degree-$(T-1)$ polynomial in $R_q[X]$, to evaluate this polynomial N times (for X a degree-0 polynomial), and to send and receive $N-1$ Shamir secret shares. Whereas, the base scheme does not require any interaction to generate the secret key. The fact that the Setup phase is only a one-time offline key-generation phase that is *re-usable* for any number of circuit evaluations enables the amortization of this step in many applications. Regarding secret-key operations ($\Pi_{\mathsf{EncKeyGen}}$ and Π_{Decrypt}), the only overhead is the local computation of the GetAdditiveShare procedure that is $\mathcal{O}(T)$. This overhead, however, is close to negligible in practice. This is because the computation of the Lagrange coefficient, which is done over $\mathbb{Z}_q$ due to the compact Shamir public-points selection of Section 3.4.2, is the only part of this computation that depends on T. We demonstrate this empirically, in Section 3.5.2.

Comparison with the Scheme of Boneh et al. Boneh et al. propose a T-out-of-N-threshold scheme as an essential building block to their universal thresholdizer for cryptographic primitives [BGGJ+18]. However, they consider a stronger asynchronous setting, where parties are unable to determine (or optimistically guess) the set of online other parties when performing secret-key operations. Essentially, their solution is to perform the Lagrange interpolation homomorphically, when aggregating the shares. But, such an aggregation can be performed only when the Lagrange coefficients are small with respect to q. Therefore, their first solution is using a $\{0,1\}$-LSSS to share the secret key of the scheme. For a T-out-of-N-threshold access-structure, this implies a per-party state in $\mathcal{O}\big(N^{4.2}\big)$ to store the secret-key shares. Their second solution is using Shamir secret-sharing, which requires only a $\mathcal{O}(1)$ storage for the secret-key shares (assuming a trusted setup). However, this requires increasing the size of the modulus q by a $\mathcal{O}\big(N!^3\big)$ multiplicative factor, thus rendering the encryption scheme non-compact and more difficult to parameterize (increasing the coefficient modulus while keeping the other parameters fixed reduces the security of RLWE). In contrast, our scheme targets the synchronous setting, yet is much simpler and more efficient; this enables its implementation and its integration into

Table 3.2: Benchmarked HE Parameters. The polynomial degree n and coefficient modulus q size in bits are taken from the standardization document [ACCD+18]. L is the number of prime factors of q.

Set	Pol. deg. (n)	Coeff. size (L)	Coeff. size ($\log_2 q$)
I	2^{13}	4	218
II	2^{14}	8	438
III	2^{15}	15	881

the Lattigo library. Notably, it can be seen as an extension of an existing scheme, requires a $\mathcal{O}(1)$ storage for the secret-key shares, and has a negligible online overhead. Moreover, it does not require a trusted dealer of shares.

3.5.2 Basic Operations Benchmarks

We implemented the scheme extension T in our Lattigo open-source library [MBTH20] and benchmarked its performance on an AMD Ryzen 9 5900X CPU (3.7GHz clock, 6M of L2-cache), for several common choices of encryption parameters (summarized in Table 3.2) and several values of the threshold T. Note that our implementation itself uses no parallelization, but its interface enables a party to generate the shares for each other party separately in step 2 of the Thresholdize operation. Hence, this step can be parallelized and the actual latency divided by $\min(N, C)$, where C is the number of cores available. In the scope of this micro-benchmark, we report the total CPU time to abstract this setting-dependent variable and to show the actual cost of the computation (the latency being relatively low in the context of a networked system). We report the results for the threshold extension T in Table 3.3 and for the relevant operations of the TMHE scheme in Table 3.4.

We observe that the Thresholdize algorithm is the most expensive operation, with a consistently higher network cost. We also observe that the cost of the procedure increases in $\mathcal{O}(NT)$ as expected. Hence, for adversarial models admitting a fixed fraction $(T-1)/N$ of dishonest parties, the per-party CPU-cost of the setup will be quadratic in the number of participants. Due to the compact Shamir public-point technique described in Section 3.4, the T.GetAdditiveShare step is very efficient, and its cost is significantly lower than the operations of the MHE scheme to which it is a pre-processing (in the TMHE scheme). For example, the cost of generating a party's decryption share in the TMHE scheme for parameter set III with $N = 20$, $T = 7$ is 12.0 ms, only 0.4 ms of which are spent on the T.GetAdditiveShare operation. We conclude that, from a CPU-time perspective, the threshold access-structure comes at an almost negligible cost with respect to the non-threshold scheme. Consequently, the main overhead of the scheme remains the pairwise exchange of Shamir secret-shares during the one-time key-generation phase.

3.5.3 Case-study: Encrypted Federated Neural Network Training

The main application of our TMHE scheme is the MHE-MPC protocol, which is a generic MPC protocol. To further demonstrate the effects of using our construction in a concrete application of this protocol, we now consider a federated learning scenario in which multiple parties seek to train a neural network model on their joint datasets, under encryption.

Sav et al. used the MHE-MPC protocol to perform federated neural network training and inference under N-out-of-N-threshold encryption [SPTF+21]. Their approach relies on the CKKS variant of the MHE scheme and faces two important challenges: First, it relies heavily on ciphertext-slot rotations for many different rotation values (mostly for the matrix operations)

Table 3.3: Threshold extension T benchmarks (with per-step breakdown for Thresholdize, see Section 3.4.3) for $N = 20$ parties and threshold $T = 7, 14, 19$. These values represent the per-party CPU time in milliseconds.

	Param.	I			II			III		
	T	7	14	19	7	14	19	7	14	19
Thresholdize	**Step 1**	6.0	13.0	17.9	26.2	56.8	78.7	91.7	198.2	275.6
	Step 2	4.2	8.8	12.3	16.6	35.6	50.0	67.3	146.9	202.1
	Step 3		0.2			0.9			3.4	
	Total	10.4	22.0	30.4	43.7	93.2	129.5	162.4	348.5	481.2
Combine		<0.1	<0.1	<0.1	0.1	0.1	0.1	0.3	0.4	0.4

Table 3.4: Threshold scheme TMHE benchmarks in milliseconds for $N = 20$ parties and threshold $T = 7, 14, 19$. These values represent the per-party CPU time in milliseconds.

	Param.	I			II			III		
	T	7	14	19	7	14	19	7	14	19
SecKeyGen	**MHE.SecKeyGen**		0.5			2.1			7.4	
	T.Thresholdize	10.4	22.0	30.4	43.7	93.2	129.5	162.4	348.5	481.2
	Total	10.9	22.5	30.9	45.8	95.3	131.6	169.8	355.9	488.6
Decrypt	**T.GetAdditiveShare**	<0.1	<0.1	<0.1	0.1	0.1	0.1	0.3	0.4	0.4
	MHE.Decrypt		0.8			2.8			11.6	
	Total	0.8	0.8	0.9	2.9	2.9	2.9	11.9	12.0	12.0

hence requires many rotation-keys to be generated in the offline setup phase. Second, the high multiplicative depth of the training algorithm requires the parties to refresh the ciphertexts during the computation, by means of the $\Pi_{\mathsf{ColBootstrap}}$ protocol (to circumvent the high cost of a local bootstrapping). The use of secret-key operations in the training phase has two consequences: it limits the system to *synchronous learning* scenarios (where all parties have to be online for the whole training phase) and it introduces a significant communication overhead which constitutes the system's main bottleneck.

We now describe the effect of using our TMHE scheme (with a CKKS front-end) in Sav et al.'s system, assuming a T-out-of-N-threshold setting. In the scope of this case-study, we focus on their MNIST instantiation where $N = 10$ parties train a three-layer neural network to perform handwritten digit recognition. This scenario uses a polynomial degree $n = 2^{14}$, a coefficient modulus of $\log_2 q = 438$ bits with $L = 9$ primes, and requires 623 rotation keys to be generated[1], along with the public encryption- and relinearization-keys.

Setup Phase To generate the public encryption-, relinearization- and rotation-keys, we propose to equally distribute the set of keys to be generated among the online parties (up to a difference of one key per party). Each party then picks a random set of $T - 1$ other parties per key it is responsible for, queries these other parties for their shares, and aggregates them (as defined in the TMHE scheme). Finally, each party retrieves the aggregated share of the keys it is not responsible for (from the designated party for that key).

We implemented a proof of concept Go application for this setup procedure based on our open-source TMHE scheme implementation in Lattigo. The protocol interactions were implemented as a client-server application that enables the parties to query each other for their respective

[1]The work of Sav et al. actually abstracts the setup phase and their code is closed-source. This value was obtained through communication with the authors.

Table 3.5: Threshold MHE Setup cost for $N = 10$ parties, $T = 8, 6, 4$, 623 rotation keys. The values are the largest measured per-party costs among all parties, along with their ratio with respect to the $T = N$ case.

	Scheme	MHE	TMHE		
	t	10	7	5	3
Time [s]	**CPU time**	149.9 (100%)	120.1 (80.1%)	108.6 (72.4%)	88.7 (59.1%)
	Wall time	67.1 (100%)	53.7 (80.0%)	48.6 (72.4%)	35.1 (52.3%)
Network [GB]	**Sent**	5.3 (100%)	4.5 (84.9%)	3.9 (73.6%)	3.3 (62.2%)
	Received	5.3 (100%)	4.4 (83.01%)	3.8 (71.7%)	3.2 (60.4%)

shares, as well as for the aggregated shares they are responsible for. To estimate the minimum wall-time latency of the setup phase, the application performs all queries to the other parties in parallel. We benchmarked this implementation on a network of 10 machines equipped with an Intel Xeon E5-2680 v3 CPU (2.5 GHz, 30 MB cache) and 256 GB of RAM. To simulate a realistic WAN-like network, we limited the network's bitrate to 1 Gbits/sec and introduced a 10 ms latency. We instrumented our code to report the total amount of data sent and received for each party, as well as the total wall time for the execution of the setup phase, and we extracted the CPU time from the operating system's metric. Our experiment assumes that all parties are online to perform the setup.

The results for the MNIST setup are summarized in Table 3.5. Our experimental result confirms that the use of our TMHE scheme reduces the per-party cost when more than T parties are participating to the setup. We do not observe a factor $\frac{T}{N}$ reduction with respect to the $T = N$ case (which uses the MHE scheme directly). This is because the final phase (query of the aggregated shares) still depends on N when all parties are online. But the cost reduction remains significant, hence motivating the use of the T-out-of-N-threshold scheme when the threat model allows it. We also observe a larger gap between the CPU and wall times for $T = 3$, as the parallelization of batched secret-key operation described in Section 3.4.6 starts being effective. We note that this effect should be observed also for $T = 5$, but it is not. This suggests that more engineering would be needed: for example, by partitioning the set of parties into two groups operating individually for the share generation and aggregation phase.

Online phase The training algorithm used by Sav et al. is an iterative distributed gradient descent with two phases per iteration. The first phase is a local gradient descent, where each party computes its local gradients through forward-backward propagation. The second phase is a global model update where a designated party aggregates all the gradients and updates the model weights. The model weights and the gradients are encrypted throughout the whole process and the number of iterations is a parameter of the system. As the source code of their system is closed-source, we study its online phase from a theoretical perspective. More precisely, we focus on its communication complexity because it constitutes the main bottleneck of the algorithm.

This bottleneck is caused by the use of the interactive refresh protocol for ciphertexts that have reached a certain level L_{ref} (the smallest level at which the refresh protocol is correct and secure, see Section 5.F of [SPTF+21]). In phase 1, each party requires β refresh where β is a function of model size and encryption parameters (also see Section 5.F of [SPTF+21]). In phase 2, the designated party requires l refreshes (one per non-input layer). A single instance of the refresh protocol requires the initiator to broadcast the level-L_{ref} ciphertext to be refreshed and to collect one share per party. The ciphertext consists of two ring elements at level L_{ref} and each share consists of one ring element at level L_{ref} and one ring element at the largest level L.

Assuming 8-byte encoding for the coefficients, the transcript of a single refresh protocol is of size $E = 8n(3L_{ref} + L)$ bytes per party assisting the initiator in the protocol. In the N-out-of-N-threshold model, this represents a total communication of $N\beta(N-1)E$ bytes for the first phase and of $l(N-1)E$ in the second. For the MNIST instance, this represents a communication of 644.1 MB per iteration ($\beta = 4$, $l = 2$, $L = 7$ and L_{ref}).

We propose the following modification to the framework of Sav et al., which is again a straightforward application of the TMHE scheme: In the local gradient descent phase (1), each party picks a random subgroup of $T-1$ other parties in the set of online parties and performs all refresh protocols among this group. In the global-model update phase (2), the aggregator randomly partitions the set of online parties into $\lfloor \frac{|\mathcal{P}_{online}|}{T} \rfloor$ groups, and distributes the batch of l refresh protocols among the groups. The proposed changes extend the framework to the asynchronous learning scenario (with a tolerance of $N-T$ offline parties). In the case where all parties are online, it reduces the communication complexity for phases 1 and 2 to, respectively, $N\beta(T-1)E$ and $l(T-1)E$; this corresponds to a total of 286.3 MB per iteration for the MNIST instance. Additionally, it divides the latency of step 2 by $\lfloor \frac{|\mathcal{P}_{online}|}{T} \rfloor$. Hence, as for the setup phase, the use of our fault-tolerant scheme also comes with a general reduction of the online phase costs, especially when it relies on the refresh protocol.

3.6 Chapter Summary

In this chapter, we extended the MHE scheme of Chapter 2 to a T-out-of-N-threshold access-structure. We demonstrated that the approach of re-sharing the secret-key shares composes well with our initial approach, and that this yields an elegant and efficient solution. Notably, the extension introduces additional interaction only at the key-generation phase and, due to our technique for compact Shamir public-points, has only a negligible memory and CPU-time overhead with respect to the base scheme. As a result, not only does our scheme provide fault-tolerance to the MHE-based MPC protocol but, when the number of online parties is above the threshold, it also reduces the per-party costs and overall latency. Moreover, our solution enables parties that only temporarily go offline to re-integrate the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol execution.

Chapter 4

Lattigo: a Multiparty Homomorphic Encryption Library in Go

Chapter Content

In the previous chapters, we proposed several constructions for multiparty homomorphic encryption (MHE) and their instantiation in a MHE-based MPC protocol (the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol). We also discussed the features of this protocol, the way they make it highly relevant from a system-design point of view, and we demonstrated its concrete efficiency. As a result, there is a great interest in building concrete MPC systems that can employ MHE schemes. However, prior to the work of this thesis, no open-source library implementing the MHE scheme for RLWE existed. In this chapter, we report on one of the most renowned contributions of our work: the Lattigo library for multiparty homomorphic encryption.

Multiparty computation systems, by nature, are highly interactive, concurrent and cross-platform. They belong to the family of *distributed systems* that are notoriously difficult and costly to implement. Fortunately, the new generation of programming languages, such as Go [1], greatly reduces this effort, notably by featuring built-in concurrency primitives, extensive standard libraries, and comprehensive toolchains for building, testing, and analyzing code. Moreover, Go features a minimal set of fundamental concepts and associated syntax, which makes learning the language easy. This is especially relevant in the setting of academic research, as it generally does not have the resources associated with large-scale software development and often relies on students for the implementation work. However, at the time of starting our work on Lattigo, most of the existing HE libraries were written in C++. To provide an alternative, we design and develop a high-performance HE library in the Go language, and we make it open-source.

4.1 Building an (M)HE library in Go

The development of Lattigo began in March 2019, as part of our research on multiparty homomorphic encryption (MHE) and secure multiparty computation. In addition to seeing the scientific interest in being able to quickly integrate our research results into a code-base for their empirical evaluation, we saw an opportunity to cater to the needs of the community by bringing HE to a new programming language: Go. Our group currently uses Go for the implementation of several applied research projects. As these systems transitioned from proof-of-concept implementations to real-world prototypes deployed in operational settings, the need for a cryptographic layer supporting MHE became pressing.

4.1.1 Challenges

Besides fulfilling the aforementioned needs for an (M)HE library, our work on Lattigo addresses several technical challenges related to such an implementation.

Complexity of the Constructions Lattice-based HE constructions are comparatively more complex than traditional (i.e., non-homomorphic) encryption schemes. From a high-level perspective, their interface is that of an (asymmetric) encryption scheme, augmented with evaluation capabilities. At the very low-level, fast ring arithmetic requires multiple layers of algorithmic optimization, notably

- The decomposition of R_q with composite $q = \prod_{i=0}^{l} q_i$ into a residue number system (RNS), which enables the arithmetic to be performed in smaller sub-rings $R_{q_1} \times R_{q_2} \times \cdots \times R_{q_l}$.
- The number theoretic transform (NTT) [AB74] that enables fast polynomial multiplication.
- The Montgomery modular multiplication [Mon85] that enables fast modulo q multiplication in the polynomial-coefficient domain.

[1] `https://go.dev`

These optimizations rely on special representations of the polynomials and their coefficients, and they require specific parameterization of the ring. Moreover, not all operations can be performed in all representations (e.g., sampling cannot be performed in the NTT domain). This requires to sparingly apply the transforms during the computation. In the case of the NTT, the transform is costly (and often constitutes the bottleneck of the operations that require it), hence we need to avoid switching domains as much as possible. As a result, efficiently computing even a simple term of the form $ab + c$ could require different algorithms, depending on the domain of each operand and on the expected domain for the output.

Multiplicity of Schemes This last decade has been a prolific one for HE research and has led to a new generation of schemes based on RLWE. From the user perspective, these schemes differ mostly in their plaintext space and in the way the noise is managed throughout the computation. As the "right" scheme to use in a given context depends on the application, it is desirable for an HE library to implement as many of them as possible. In this regard, the good news is that these schemes share common functionalities (i.e., the *core* RLWE HE scheme, see Section 1.4.1), which enables the implementer to re-use code between schemes (by expressing these schemes as *front-ends* over the core scheme, see Section 1.4.1). However, the somewhat less good news is that the theoretical literature in which these schemes are proposed, due to the rapidly evolving set of techniques in use, do not present their contributions in such a modular way. Rather, they present the proposed schemes in a monolithic way, which makes it hard for newcomers to the field to extract such common functionality.

Our *core* + *front-end* formulation, from which the design of Lattigo arises, reflects this systematization effort.

Flexibility and Future-Proofness In addition to supporting the existing schemes with as little code as possible, our modular design is sound and flexible. This achieves a two-fold effect: First, it ensures that the library is able to integrate future constructions and improvements on RLWE-based FHE, hence remains relevant in the longer term. Second, and in conjunction with open-source, it enables *advanced* users to implement proof-of-concept of their new constructions *at the right layer*. This is indeed much faster than if they were starting from scratch, or if they were modifying a monolithic piece of software.

In Lattigo, good examples of the benefits a sound design provides are the introduction of our multiparty functionalities, as extensions of the single-party schemes, and the recently added support of the scheme by Kim and Song for approximate real-number arithmetic [KS19], as a specific parameterization of the CKKS scheme.

Performance and Parameterization For most applications, the cost of using HE dominates the overall application cost. Therefore, minimizing the CPU cost of the library's operation is paramount to maximize the application performance. Although this is first and foremost a matter of algorithmic optimization, such as the aforementioned ring-arithmetic optimizations, software optimization techniques also play an important role. One of the main reasons for this lies in the fact that HE schemes have application-specific parameterization. More precisely, whereas traditional encryption schemes are mostly concerned with setting the parameters for a fixed security level, HE schemes require to be parameterized according to a particular set of functions to be evaluated. As a result, their implementations cannot simply hard-code these parameters and their corresponding constants (e.g., as one would hard-code the constants of a given elliptic curve). Instead, HE implementations need to pre-compute these values at run-time,

and to ensure that their APIs enable a user to perform these pre-computation outside of the critical path of an application.

Our software and API design principles, which we present in Section 4.2.1, account for software-optimization techniques such as pre-computation caching and buffer pre-allocation. Additionally, our work on the Lattigo library yields several novel algorithmic optimizations that are relevant for HE research. We summarize these cryptographic optimizations in Section 4.2.2.

Interfaces Design and Usability It is essential for any library to have a well-designed application programming interface (API). In designing such an API, library designers face the challenge of providing a simple API (which is easy to use by novice users and reduces the probability of making errors) while supporting use-cases of advanced users.

One crucial aspect of Lattigo, in this regard, is that we choose to export each of its layers as a standalone package (we discuss these layers in the library overview in Section 4.2). As a result, more advanced users have the ability to implement new functionalities (such as new schemes) by relying on these lower-level layers.

4.2 Library Overview

Lattigo is an open-source Go module[2] available under Apache 2.0 license[3]. At the time of writing, its current version is `v4.1.0`. Lattigo is now actively maintained by the start-up *Tune Insight SA*[4] that uses the library as a core component of its products and solutions.

4.2.1 Internal Design

We now describe the notable aspects of the library's design; they address the challenges exposed in the previous section.

Layered Architecture To address the challenges arising from the multiplicity of schemes, and the need for maintainability, as well as flexibility, we use a four-layer architecture:

- The *Ring* layer is the lowest one; it implements the cyclotomic polynomial ring R_q, its operations, and sampling according to the various distributions required by RLWE.
- The *Core RLWE* layer implements the core functionalities that are common to all (or several) HE and MHE schemes. This includes the *core* RLWE scheme of Section 1.4.1, as well as the threshold schemes of Sections 2.2 and 3.4. Although this layer principally serves as a common base for implementing the various *front-end* HE and MHE schemes, it is already user-facing. This is because objects that are not redefined by the front-ends, such as cryptographic keys and ciphertexts, are imported directly from this layer.
- The *Scheme* layer implements the front-end schemes, i.e., the BFV, BGV, and CKKS schemes. Hence, this layer implements (i) the particular plaintext encoding(s) and decoding(s) of each scheme, and (ii) the homomorphic operations.
- The *Multiparty scheme* layer implements the front-ends for the multiparty schemes, based on both the *Core RLWE* layer (for the core MHE scheme constructions) and the *Scheme* layer (mostly for the plaintext encoding of the relevant front-end scheme). Similarly to the *Scheme*

[2] `https://go.dev/ref/mod`
[3] `http://www.apache.org/licenses/`
[4] `https://tuneinsight.com`

layer, this layer is mostly concerned with evaluation-time operations (i.e., the Compute phase in the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, see Section 2.4). The Setup phase functionalities (the collective generation of public encryption and evaluation keys) are common for all schemes and are directly imported by the user from the *Core RLWE* layer.

Standalone Packages at Each Layer Each layer of the Lattigo design consists of multiple *Go packages*[5]. By definition, packages in Go are standalone units and should provide a usable API to users. Lattigo complies with this paradigm and exposes the packages listed in Table 4.1. As a result, we provide a paradigmatic solution to the flexibility and usability challenges.

Pure Go Implementation All packages, including the low-level *ring* package, are implemented exclusively in Go. Although it might be tempting to rely on existing number-theory libraries such as NTL [6] or GMP [7], doing so would require using CGO, the C-Go linking interface. Unfortunately, doing so would hinder the usability, because C wrappers break the portability of the code to some architectures such as WebAssembly, which enables running Lattigo in a browser environment [8].

API Principles The operations of the front-end schemes are defined as methods (e.g., `Add`) over purpose-specific objects, `struct` types in Go, (e.g., `Evaluator`) that encapsulate the cryptographic parameters, temporary buffers and pre-computations. These objects are initialized from the cryptographic parameters, and all pre-computation and temporary-buffer allocations occur at initialization (e.g., with a `NewEvaluator(params)` method). For performance reasons, API functions (e.g., `Evaluator.Add(a, b, res)`) use these pre-allocated buffers and output registers, and they do not allocate memory, unless if explicitly labeled as doing so (e.g., `res := Evaluator.AddNew(a, b)`).

As of version v4.1.0, Lattigo provides a single-threaded implementation of its API and all types assume single-threaded use. Therefore, the API user controls the concurrency aspects of its application.

4.2.2 Cryptographic Optimizations and Features

We now summarize the aspects of Lattigo that are relevant from a cryptographic-research standpoint. These aspects are algorithmic improvements to critical operations in RLWE-based HE schemes.

Generalized Key-Switch Procedure One of these critical operations is commonly referred to as *key-switch*. It enables the evaluator to re-encrypt a ciphertext, which was encrypted under some function $f(s)$ of the secret-key, *back* to a ciphertext under the secret-key s. This operation is used for all homomorphic operations (besides addition) to invert their effect on the decryption structure, and requires a public-key (commonly referred to as an *evaluation-key*). For example, the homomorphic multiplication yields a ciphertext that can be seen as being encrypted under s^2 and requires a second step (commonly referred as the *relinearization*) to operate a key-switching from s^2 back to s.

[5] `https://go.dev/ref/spec#Packages`
[6] `https://libntl.org`
[7] `https://gmplib.org/`
[8] `https://webassembly.org/`

Table 4.1: The `github.com/tuneinsight/lattigo/v4 v4.1.0` Go module

	Ring layer
`ring`	implements the RNS-accelerated modular arithmetic over the cyclotomic ring $\mathbb{Z}_Q[X]/(X^n+1)$. This includes: the RNS basis extension, the RNS division (rescaling), the NTT, and the uniform, Gaussian, and ternary sampling.
	Core RLWE layer
`rlwe`	implements the *core* RLWE scheme (see Section 1.4.1) i.e., generic operations that are common to RLWE schemes.
`drlwe`	implements the *core* N-out-of-N-threshold RLWE scheme (see Section 2.2) and its T-out-of-N-threshold extension (see Section 3.4).
	Scheme layer
`bfv`	implements the RNS-accelerated Fan-Vercauteren version of Brakerski's scale-invariant homomorphic encryption scheme (BFV) [FV12; BEHZ16; HPS19], as a *front-end* of package `rlwe`.
`bgv`	implements the RNS-accelerated variant of the levelled scheme of Brakerski et al. (BGV) [BGV14], as a *front-end* of package `rlwe`.
`ckks`	implements the RNS-accelerated version of Cheon et al.'s HE scheme for arithmetic over approximate complex numbers (CKKS) [CKKS17; CHKK+19] as-well-as its variant for real numbers [KS19].
`ckks/advanced`	implements advanced homomorphic operations for the CKKS scheme, such as the homomorphic CKKS encoding/decoding, the plaintext-polynomial evaluation and the approximation of non-linear functions.
`ckks/bootstrapping`	implements the bootstrapping for the CKKS scheme [BMTH21; BTH22].
`rgsw`	implements the subset of the HE scheme of Gentry, Sahai and Waters (GSW) [GSW13] that is necessary for bridging between RLWE and LWE-based schemes and for supporting look-up table evaluation.
`rgsw/lut`	implements the generation and homomorphic evaluation of look-up tables for RLWE schemes.
	Multiparty scheme layer
`dbfv`	implements the BFV *front-end* for the threshold RLWE scheme.
`dbgv`	implements the BGV *front-end* for the threshold RLWE scheme.
`dckks`	implements the CKKS *front-end* for the threshold RLWE scheme.

In Lattigo, we employ a generalization of the key-switch procedure proposed by Han and Ki [HK20], which enables the user to tune a trade-off between the homomorphic capacity, the evaluation-key size, and the key-switch algorithmic complexity. Informally, this lets the user specify a number α of the l prime factors of q to be *specially reserved* for the key-switching operation. These specially reserved primes are commonly referred to as *special primes*, and their product is denoted by p. Although these reserved primes cannot be exploited during other operations, which reduces the ciphertext space modulus from $q = \prod_{i=0}^{l} q_i$ to $q' = qp^{-1} = \prod_{i=0}^{l-\alpha} q_i$ hence reduces the homomorphic capacity, they result in an increased throughput for the key-switching operation. Figure 4.1 shows that, by increasing α to 4, we achieve a $2\times$ increase in throughput and a $5\times$ decrease in the key-size. This shows that, in terms of throughput, the loss in homomorphic capacity is more than compensated by the run-time reduction.

We also further optimize the key-switch-key format and key-switch algorithm for the evaluation of automorphisms such as rotations, as proposed by Bossuat et al. [BMTH21].

Novel BFV Quantization The BFV homomorphic multiplication (even in its RNS variant [BEHZ16; HPS19]) is an expensive operation because it requires the use of a large secondary basis [FV12] to temporarily represent intermediate values in the ciphertext tensor-product. Lattigo follows a novel approach to handling this operation, by adapting the RNS-friendly quantization techniques proposed in the original full-RNS variant of the CKKS scheme [CHKK+19]. At the time of proposing this algorithm, this gives Lattigo a considerable performance advantage over existing implementations (see Section 4.3 for the benchmark comparisons).

CKKS Bootstrapping The Lattigo library is the first library to implement the bootstrapping operation for the RNS-variant of the CKKS scheme (the only previously implemented bootstrapping for CKKS being specific to the non-RNS variant [9]). At the time of writing, it is still the only open-source implementation of this procedure. Moreover, our implementation effort resulted in several theoretical improvements to various sub-procedures of the bootstrapping. Compared to the contemporary state-of-the-art, our bootstrapping procedure is both more efficient and more precise (as shown in Figure 4.2), and it does not require the use of sparse secret keys. These results [BMTH21; BTH22] were accepted and presented in two highly ranked venues of the cryptographic research. The results of [BMTH21] were presented in more detail in the introduction of this dissertation.

Homomorphic Polynomial Evaluation Several of the aforementioned sub-procedures are of independent relevance, i.e., outside the bootstrapping operation itself. One such procedure is a scale-invariant and depth-optimal homomorphic polynomial evaluation algorithm that can be applied to polynomials in both the standard and the Chebyshev basis. The procedure enables the user to provide the clear-text polynomial coefficient and a desired output scale. Then, the procedure starts by recursively computing the scale at which each monomial is computed in order to ensure that all rescalings in the evaluation will be exact [BMTH21]. Another such procedure is the homomorphic (plaintext) linear transform. The `ckks/advanced` package provides standalone implementations of these functions, upon which the `ckks/bootstrapping` package builds to provide the bootstrapping operation. Hence, in addition to being the first HE library to provide a state-of-the-art bootstrapping for CKKS, Lattigo is also the first library to make its core components available to the users.

[9] `https://github.com/snucrypto/HEAAN`

Table 4.2: CKKS Bootstrapping Parameters. n is the ring degree, h is the number of non-zero coefficients in the secret-key, $\log(q')$ is the bit-size of the ciphertext modulus, $\log(p)$ is the bit-size of the key-switching decomposition-basis (the security is based on $\log(pq')$), and C the ciphertext modulus consumption by bootstrapping (in bits).

Parameter Set	n	h	$\log(q')$	$\log(p)$	C
Best of [CCS19a]	2^{16}	64	1240	1240	1057
Best of [HK20]			1270	182	900
Set II		192	1248	305	743
Set III		2^{15}	1416	366	956

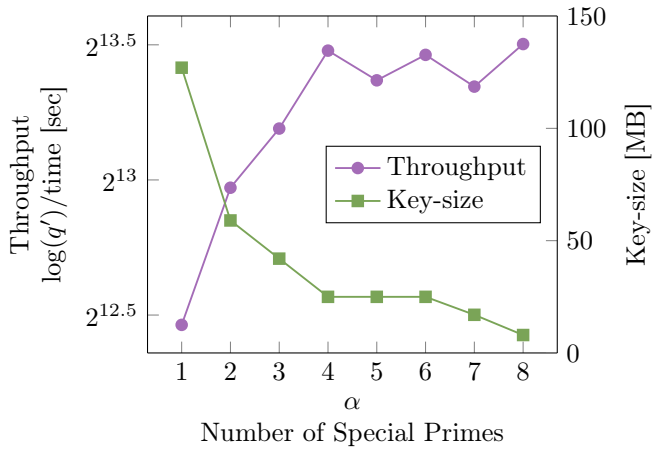

Figure 4.1: Comparison of the public key-switch operation throughput (in ciphertext-bits/sec.) and public switching-key size for $n = 2^{15}$ and variable $1 \leq \alpha \leq 8$ and $l = 16 - \alpha$ (Lattigo v2.1.0).

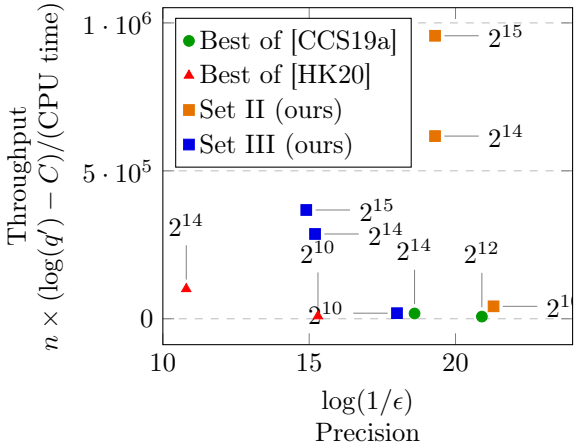

Figure 4.2: (From [BMTH21]) Comparison of the bootstrapping utility. See Table 4.2 for the parameters. We plot the results for our best-performing parameter set against the state of the art. Nodes are labeled with n, the number of plaintext slots, $\log(q') - C$ is the residual homomorphic capacity (in bits) after the bootstrapping, and the precision $\log(1/\epsilon)$ is defined as the negative log of the mean error across all the slots.

Multiparty Homomorphic Encryption One of the main purposes of the Lattigo library is to support the development of MHE-based MPC protocols. Lattigo implements the MHE scheme presented in Section 2.2 as well as its T-out-of-N-threshold extension presented in Section 3.4, for the BFV, BGV, and CKKS front-ends. As such, it is the first library to implement the complete set of local operations required to support the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol.

4.3 Performance Comparison

A crucial issue in implementing HE is that of runtime performance. In this regard, the C and C++ languages are often considered as the de-facto standard, as they enable their users to have control over their programs, down to machine-level considerations. The Go language is somewhat different and, like many recent languages, favors safety and simplicity over control. This raises the question of whether we can write fast cryptographic code in Go or not.

To answer this question, we perform benchmarks for the HE primitives implemented in Lattigo, against those of Microsoft's SEAL C++ library[10] as a baseline. We use Lattigo `v2.1.0` and SEAL `v3.6`, the latest version of both libraries at the time of performing these benchmarks.[11] All experiments are conducted single-threaded on an i5-6600k at 3.5 GHz with 32 GB of RAM running Windows 10. We use Go version 1.14.2 for building Lattigo and the MSVC++ compiler version 14.28 to compile the SEAL library and its benchmarks.

Parameters We define the benchmarked parameters as the triplet $\{n, l', \alpha\}$, where n is the ring degree, l' is the number of ciphertext moduli (prime factors of q') and α is the number of special primes for the key-switching (prime factors of p). These factors are the most relevant when comparing the library performance, as each individual modulus factor q_i fits within exactly one (64-bits) machine word. Both Lattigo and SEAL propose several default parameter sets for 128-bit security (according to the standardization document [ACCD+18]) and varying homomorphic capacity. However, at the time of writing, SEAL does not yet support the use of multiple moduli in the extended-basis p (it enforces $\alpha \leq 1$), hence does not support the default parameters of Lattigo. Consequently, we performed our benchmarks with the default parameters of SEAL.

Results Tables 4.3, 4.5, 4.4, and 4.6 summarize the timings of local operations for BFV and CKKS in a single-key setting, along with the corresponding baseline-system timings. We conclude from these benchmarks, that it is possible to produce, in Go, cryptographic code that matches and even surpasses the performance of a C++ implementation[12].

We believe that the good performance results of Lattigo can be attributed to the efficiency of the package `lattigo/ring` that relies on low-level Go-friendly optimizations (e.g., Montgomery and pointer arithmetic, lazy-reduction, loop unrolling), as well as scheme-specific high-level algorithmic optimizations (e.g, a novel BFV quantization, operation-specific plaintext encoding).

4.4 Applications

Here, we briefly discuss several of Lattigo's first applications, from our group's line of research. Lattigo has been successfully used in practical scenarios, for both its intended use-cases: in single-party HE for client-server applications, and more complex workflows requiring MHE.

[10] `https://github.com/Microsoft/SEAL`
[11] At the time of performing these benchmarks, the BGV scheme was not implemented by either libraries.
[12] For more recent benchmarking work confirming this observation, see the wortk of Gouert et al.[GMT22]

Table 4.3: BFV Timings in μs for $2^{10} \leq d \leq 2^{13}$.

Op	$d = 2^{11}, L = 1$		$d = 2^{12}, L = 2$		$d = 2^{13}, L = 4$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	29	26	60	55	122	123
Decode	29	30	73	56	129	112
Encrypt	803	226	2085	936	5711	2935
Decrypt	110	64	358	284	1374	1251
Add	7	3	28	11	126	46
Mul-Pt	129	90	482	380	2084	1652
Mul-Ct	1146	476	3721	2065	14987	9123
KeySwitch	-	-	775	745	3933	3781

Table 4.4: BFV Timings in ms for $2^{14} \leq d \leq 2^{16}$

Op	$d = 2^{14}, L = 8$		$d = 2^{15}, L = 15$		$d = 2^{16}, L = 31$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	0.2	0.2	0.5	0.5	1.1	1.2
Decode	0.2	0.2	0.6	0.6	1.2	1.3
Encrypt	18.5	10.5	65.4	39.2	253.5	153.0
Decrypt	5.6	5.3	23.5	22.4	115.7	95.6
Add	0.4	0.2	1.7	1.0	7.1	4.5
Mul-Pt	8.8	7.4	34.1	32.1	149.5	133.2
Mul-Ct	65.7	44.9	400.3	205.7	2822.6	1186.3
KeySwitch	24.3	24.1	147.0	154.5	1183.8	1235.5

Table 4.5: CKKS Timings in μs for $2^{10} \leq d \leq 2^{13}$.

Op	$d = 2^{11}, L = 1$		$d = 2^{12}, L = 2$		$d = 2^{13}, L = 4$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	112	103	305	247	854	668
Decode	63	82	345	174	1385	382
Encrypt	548	392	1816	1115	5329	3680
Decrypt	18	6	71	27	272	108
Add	7	3	28	10	124	46
Mul-Pt	14	7	52	27	210	126
Mul-Ct	45	15	187	61	795	242
Rescale	-	-	203	222	861	857
KeySwitch	-	-	807	731	3927	3619

Table 4.6: CKKS Timings in ms for $2^{14} \leq d \leq 2^{16}$

Op	$d = 2^{14}, L = 8$		$d = 2^{15}, L = 15$		$d = 2^{16}, L = 31$	
	SEAL	Lattigo	SEAL	Lattigo	SEAL	Lattigo
Encode	3.2	1.9	14.3	6.3	58.0	22.9
Decode	6.4	0.8	31.7	3.1	230.7	6.5
Encrypt	19.0	13.0	71.6	50.0	295.7	211.4
Decrypt	1.1	0.4	4.7	2.3	19.2	9.1
Add	0.4	0.2	1.7	0.9	7.2	4.5
Mul-Pt	0.8	0.5	3.1	2.2	13.2	9.1
Mul-Ct	3.1	1.1	12.0	4.6	49.2	19.4
Rescale	3.6	3.4	14.6	13.8	64.2	65.7
KeySwitch	23.4	22.8	146.5	143.5	1178.5	1205.8

Client-Server Applications A paradigmatic case of a secure service that operates on encrypted sensitive client data, performing private genotype imputation, was one of the proposed tasks in the 2019 iDash challenge [13]. Our group employed Lattigo for developing one of the three winning solutions: a multinomial logistic regression inference with CKKS-encrypted data that performs a batch prediction (1,000 patients with 80,000 to-be-imputed variants each) in seconds and has memory requirements and prediction accuracy comparable to clear-text state-of-the-art genotype imputation tools [KHBC+21].

Lattigo was also used for implementing a passively secure oblivious linear-function-evaluation (OLE) protocol [BEPS+20], which is a common building block for generic MPC protocols. This protocol generalizes oblivious transfer to linear functions, and its Lattigo implementation (on top of the `ring` package) is able to evaluate more than 1 million OLEs per second over the ring $\mathbb{Z}_m$, for a 120-bit m on standard hardware.

Large-Scale Multi-party Applications Lattigo has been used for implementing distributed training and evaluation of several machine-learning models, including generalized linear models [FTPS+21] and feed-forward neural networks [SPTF+21]. The systems built with Lattigo are capable of efficiently scaling up to thousands of parties and achieve a high training throughput, and they close the accuracy gap with respect to centralized clear-text systems. Examples of the achieved performance include training a logistic regression model on a dataset of 1 million samples with 32 features distributed among 160 data providers in less than three minutes [FTPS+21], and training a 3-layer neural network on the MNIST dataset with 784 features and 60,000 samples distributed among 10 parties in less than 2 hours [SPTF+21].

4.5 Chapter Summary

In this Chapter, we introduced the Lattigo library, a multiparty homomorphic encryption library written in Go. Lattigo greatly facilitates the development of new HE- and MHE-applications by enabling the use of these primitives in a modern language: Go. By considerably reducing the development time of such applications, Lattigo can be a catalyst in both the cryptography research and the adoption of HE and MHE in real systems.

At the time of writing, Lattigo is being used in many more projects than those presented in this chapter. The library is being used both by applied-cryptography researchers for building application prototypes [BSA21; CPTH21; FTRC+21; KSJH21; ICDÖ22; TMBM+22; PPV22; SBTC+22; SDPB+22; CP23a; ERLT23; KG23; FCES+23] and by HE researchers for implementing proof-of-concept of new constructions and optimization [CHKL+21; HKLL+22; KKLS+22; KLKS+22; LLKK+22; GHHJ22; ACYJ+23; CP23b; KLSS23].

However, by design, the scope of Lattigo is limited to the implementation of local operations. In other words, it provides the local operations of the HE and MHE schemes but not an *end-to-end* implementation of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. Building such a system requires implementing the network and application logic, but it is undesirable to implement this logic within the cryptographic library itself. Indeed, this could not be done without restricting the generality of cryptographic library and would considerably increase the cost of maintaining it. In general, it is good practice to define and implement this logic in a separate code-base. Therefore, as doing so is the last step separating the theory of MHE from its practical use, implementing the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol into a concrete *end-to-end* MPC system is the topic of the next chapter.

[13] `http://www.humangenomeprivacy.org/2023/`

Chapter 5

Helium: an MHE-based MPC Framework

Chapter Content

In this chapter, we propose a framework: HElium. It is an end-to-end implementation of the MHE-based MPC protocol ($\Pi_{\mathsf{MHE\text{-}MPC}}$, see Chapter 2) that supports lightweight and churning parties. To support churn, HElium relies on the T-out-of-N-threshold MHE scheme of Chapter 3 and builds on top of its Lattigo implementation (Chapter 4).

Low-Requirements MPC Performing MPC tasks among computationally weak and unreliably connected parties is a long-standing problem in MPC systems research [BHKL18; LYKG+19; DGKN09; CHP13; CP15]. The currently implemented approaches for the N-party setting, which are mostly based on linear secret-sharing schemes (LSSS) [HHNZ19], tend to impose stringent requirements on the systems that seek to employ them: From an implementation perspective, many MPC frameworks assume that they are in control of the network stack. Although this design is acceptable for performing standalone benchmarks, it is unrealistic when integrating the framework into larger systems that have their own transport layers. In addition, they require the protocol participants to be online for the computation to make progress, and require a large amount of bandwidth. However, this requirement cannot always be met (e.g., when the participants run on low-end hardware and experience unreliable network connection), which forces many frameworks and applications to introduce new assumptions in their security model. One common such assumption is that of *non-collusion* between a set of third-party delegates among which the parties secret-share their inputs and which run the MPC protocol on the parties' behalf (e.g., the two-clouds model in which N parties delegate the computation to two non-colluding cloud servers). However, this assumption could be unsatisfactory, and the corresponding model could be difficult to implement (e.g., it requires two or more cloud service providers to provide some guarantees of non-collusion). Hence, there is a need for MPC solutions that can operate with low requirements for the participants, yet that rely solely on cryptographic assumptions.

MHE-Based MPC Due to their low communication-complexity and small number of rounds, the MHE-based MPC approaches are an ideal choice for low-requirements MPC [AJLT+12]. In their theoretical formulation for the passive-adversary model, they have sub-linear communication cost [MTBH21] and require only two rounds: one to provide the inputs and another one to reveal the computation output [AJLT+12]. More precisely, their public-transcript property enables the parties to delegate most of the communication and computation cost to a single honest-but-curious third-party (e.g., a cloud server) yet without relying on additional non-cryptographic assumptions for this delegation.

Whereas more than thirty LSSS-based frameworks were built over the last two decades [Rot17], there exists no open-source implementation of the MHE-based MPC protocol. One of the reasons is that implementing the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol requires addressing several practical challenges that the typical computation- and communication-models of the theoretical works (including the work of Chapters 2 and 3) abstract from. More specifically, the current theoretical works consider a *monolithic* execution of the protocol: each round of the protocol ($\mathsf{SecKeyGen}$, $\mathsf{PubKeyGen}$, Input and Output in Protocol 7 of Chapter 2) is executed sequentially, and the protocol terminates after the Output round. Unfortunately, as we show in this chapter, such an ideal execution is undesirable in practice, and can even be impractical in scenarios involving resource-constraint parties.

Our Contributions

In this chapter, we isolate the challenges related to a realistic, *non-monolithic* execution of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, we address these challenges, and we propose HElium: the first open-source

implementation of an MHE-based MPC protocol.

- **Challenges** We show that the monolithic execution can be impractical for resource-constrained parties, and that it does not adequately capture the fact that the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol can be seen as a long-lived session that must only be established once for a given set of parties and HE parameters. The latter brings the challenge of handling churning clients in a non-trivial way, which requires securely instantiating the TMHE scheme of Chapter 3.

- **Generic Solution** We provide the design of a non-monolithic execution that addresses the efficiency and security challenges. This design bridges the theory-to-practice gap that is left open by theoretical work on MHE and is generic: it applies to both the peer-to-peer and cloud-assisted settings of the MHE-based MPC protocol.

- **The HElium Framework** We propose HElium, an end-to-end framework for MHE-based MPC in the cloud-assisted model. HEliumhas very low requirements for the parties: They can run on several hundreds of megabytes of RAM, their communication cost is sub-linear in the number of parties, and they do not need to be simultaneously online and reachable.

- **Implementation** We implement our generic solution and make it open-source. We provide a solution that can be easily integrated into larger applications, by expressing its transport layer as an abstract interface in the *remote procedure call* (RPC) paradigm (by default, our implementation uses a gRPC-based service as a concrete transport).

As such, HElium is the first framework for MPC under churn that does not require non-cryptographic assumptions besides the traditional passive-adversary setting.

5.1 System Specification

Here, we define the framework's operational setting and functionality. These are, from a high level, that of a generic MPC system. This definition sets the basic requirements for our desired MHE-based MPC solution.

Setting Let $\mathcal{P} = \{P_1, ..., P_N\}$ be a set of N parties. We consider an asymmetric setting where the parties in $\mathcal{P}$ are resource constrained, cannot listen for incoming connections, and could be inconsistently online. Thus, they receive assistance from a *helper* H that is assumed to run on high-end hardware, to have the ability to listen for incoming connections, and to be consistently online. Let $\mathcal{M}$ be a ring (*plaintext space*), and let $f : \mathcal{M}^N \to \mathcal{M}$ be an arithmetic function over $\mathcal{M}$. We assume that the parties and helper have access to an MHE scheme for which, given a function f (*target function*) as above, they can derive an HE circuit C_f (*target circuit*) and a set of public parameters pp such that C_f correctly and homomorphically computes f.

Functionality From an HE circuit C_f that computes f, the HE public parameters pp, and $(x_1, ..., x_N)$ where $x_i \in \mathcal{M}$ is a private input from party P_i, the system computes $f(x_1, ..., x_N)$.

Adversarial Model We assume a passive adversary that can statically corrupt a subset $\mathcal{A} \subset \mathcal{P}$ of $T - 1$ parties for a fixed *threshold* parameter T (i.e., T is the smallest subset size such that any subset of size T is guaranteed to contain an honest node). The adversary can observe the network traffic and the internal state of all parties in $\mathcal{A} \cup \{H\}$.

Churn Model We consider that the parties in $\mathcal{P}$ can be in either the *connected* state or the *disconnected* state. To model the transitions between the states, we view the time before a disconnection event (respectively, a re-connection event) as a random variable following some distribution D_d (respectively, D_r). These events are independent among the parties.

5.1.1 System Workflow Overview

A generic MPC framework is generally a sub-component of a larger distributed system that we refer to as the *user application*. We provide an overview of the HElium framework's workflow from the perspective of the user-application designer. The workflow consists of the *conception* phase that is carried out by the user-application designer, after which the framework and user application are autonomous.

0. *Conception* In the conception phase, the user application designer translates the setting $(\mathcal{P}, T)$ and the target function f into a homomorphic circuit C_f, and a set of HE parameters *pp*. C_f and *pp* are passed, along with other configuration-related information (e.g., the parties' identifiers and network addresses, if any), as input to the framework in the next phases.

1. *Setup*. In the setup phase, the framework generates the private and public key material required for the next phase. It is a preparation phase that requires only the parameters *pp* and the circuit C_f, hence can be performed *offline* (i.e., possibly before the inputs are available).

2. *Compute* In the computation phase, the framework performs the evaluation of the target circuit C_f and outputs the result. Hence, this phase begins with the user application providing the target function's private inputs to the framework, and it ends with the decryption of the final result.

Notes on the Conception Phase We observe that our functionality definition assumes that the target function f is translated into an HE circuit by the application designer. We voluntarily leave the aspects related to circuit design and HE parameterization outside the scope of this work. Although these aspects are important for assisting non-experts during the conception phase, they are orthogonal to the contributions of this work (which focus on the operational phases) and are addressed by existing literature on HE compilers [CDS15; CPS18; CMGT+18; DKSD+20; ACDM+19; VJHH23].

5.2 MHE-based Multiparty Computation

We recall the syntax and semantics of MHE schemes and the generic MHE-based MPC protocol they enable. On account of their practical efficiency, we focus on the threshold family of MHE schemes [AJLT+12; MTBH21]. For the sake of the exposition, we use a simplified MHE model for which (i) the evaluation-key generation protocol is a single protocol that takes the operation to be enabled as an argument (instead of the $\Pi_{\mathsf{RelinKeyGen}}$ and $\Pi_{\mathsf{RotKeyGen}}$ protocols) and (ii) for which we consider only the decryption protocol (instead of the generalized output protocols $\Pi_{\mathsf{KeySwitch}}$ and $\Pi_{\mathsf{PubKeySwitch}}$).

We consider a *security* parameter λ and require that the advantage of the adversary $\mathcal{A}$ in breaking input privacy is no more than $2^{-\lambda}$. We also consider a *homomorphic capacity* parameter and require that the multiparty computation outputs the correct result with probability at least $1 - 2^{-\kappa}$.

5.2.1 MHE Semantics

Let $\mathcal{P}$ be a set of N parties, and let the threshold T be the size of the smallest subset of $\mathcal{P}$ that is guaranteed to contain at least one honest party. Given a plaintext space with arithmetic structure $\mathcal{M}$, an MHE scheme over $\mathcal{P}$ and $\mathcal{M}$ is a tuple of algorithms and multiparty protocols $\mathsf{MHE} = (\mathsf{GenParam}, \Pi_{\mathsf{SecKeyGen}}, \Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \mathsf{Encrypt}, \mathsf{Eval}, \Pi_{\mathsf{Decrypt}})$ whose elements have the following syntax and semantic:

- **Public parameters gen.** $pp \leftarrow \mathsf{GenParam}(\lambda, \kappa, \mathcal{P}, T, \mathcal{F})$:
  Given the security parameter λ and the homomorphic capacity parameter κ, the identities of the set of parties in $\mathcal{P}$, the threshold T, and a set $\mathcal{F}$ of arithmetic functions $f : \mathcal{M}^I \to \mathcal{M}$, $\mathsf{GenParam}$ outputs a public parameterization pp. This parameterization is an implicit argument to the following algorithms and protocols.

- **Secret-key generation** $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}} \leftarrow \Pi_{\mathsf{SecKeyGen}}()$:
  From the public parameters, $\Pi_{\mathsf{SecKeyGen}}$ outputs a secret-key sk_i to each party $P_i \in \mathcal{P}$.

- **Encryption-key gen.** $\mathsf{cpk} \leftarrow \Pi_{\mathsf{EncKeyGen}}(\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$:
  From any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, $\Pi_{\mathsf{EncKeyGen}}$ outputs a *collective public encryption key* cpk.

- **Eval.-key gen.** $\mathsf{evk}_{\mathsf{op}} \leftarrow \Pi_{\mathsf{EvalKeyGen}}(\mathsf{op}, \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$:
  Given a homomorphic operation op to be supported by the Eval algorithm and any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, $\Pi_{\mathsf{EvalKeyGen}}$ outputs a public evaluation-key $\mathsf{evk}_{\mathsf{op}}$ for operation op.

- **Encryption** $\mathsf{ct} \leftarrow \mathsf{Encrypt}(m, \mathsf{pk})$:
  Given the public encryption key pk, and a plaintext $m \in \mathcal{M}$, $\mathsf{Encrypt}$ outputs a ciphertext ct that is the encryption of m.

- **Evaluation** $\mathsf{ct}_{\mathrm{res}} \leftarrow \mathsf{Eval}(f, \{\mathsf{evk}_{\mathsf{op}}\}_{\mathsf{op} \in f}, \mathsf{ct}_1, \ldots \mathsf{ct}_I)$:
  Given an arithmetic function $f : \mathcal{M}^I \to \mathcal{M}$, the evaluation key $\mathsf{evk}_{\mathsf{op}}$ for each homomorphic operation op used in f and an I-tuple of ciphertexts $(\mathsf{ct}_1, ..., \mathsf{ct}_I)$ encrypting $(m_1, \ldots, m_I) \in \mathcal{M}^I$, Eval outputs a ciphertext $\mathsf{ct}_{\mathrm{res}}$ that is the encryption of $m_{\mathrm{res}} = f(m_1, \ldots, m_I)$.

- **Decryption** $m \leftarrow \Pi_{\mathsf{Decrypt}}(\mathsf{ct}, \{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'})$:
  Given ct an encryption of m, and any subset of secret keys $\{\mathsf{sk}_i\}_{P_i \in \mathcal{P}'}$ such that $\mathcal{P}' \subseteq \mathcal{P}$ and $|\mathcal{P}'| \geq T$, Π_{Decrypt} outputs m.

Current MHE scheme constructions [MTBH21] are based on the *ring-learning with error* (RLWE) problem [LPR10]. The plaintext space of such schemes is a ring of polynomials of a fixed (power-of-two) degree. Their Eval algorithm supports additions and multiplications in this ring. They also support homomorphic rotations over the coefficients of the message. Each homomorphic operation (besides the addition) requires its own evaluation key to be provided to the Eval algorithm, hence it requires the execution of a separate instance of the $\Pi_{\mathsf{EvalKeyGen}}$ protocol. We note that the "rotation of k positions" operation is considered an individual operation for each required value of k in the circuit, and it is common for applications to generate many such evaluation keys. This is because rotations are costly and achieving a desired rotation by composition (e.g., of many rotations by $k = 1$) is often impractical. A notable aspect of the current MHE schemes is that all their protocols can be executed in a single round of communication [MTBH21; Par21]. We provide a unified model for these protocols in Section 5.3.3.

Protocol 8. $\Pi_{\mathsf{MHE\text{-}MPC}}$ ▷ *MHE-based MPC (helper-assisted, for public output)*

Private input: x_i for each party $P_i \in \mathcal{P}$, sk_R for receiver R
Public input: f the circuit, pk_R the receiver's public-key
Output for R: $y = f(x_1, x_2, \ldots, x_N)$

Setup:

1. (SecKeyGen) all parties in $\mathcal{P}$ execute the secret-key generation protocol

$$\mathsf{sk}_i \leftarrow \mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}(),$$

2. (PubKeyGen) any subset of parties $\mathcal{P}' \subseteq \mathcal{P}$, with $|\mathcal{P}'| \geq T$ executes the required public-key generation protocols:

$$\mathsf{cpk} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}),$$

$$\mathsf{evk}_{\mathsf{op}} \leftarrow \mathsf{MHE}.\Pi_{\mathsf{EvalKeyGen}}(\mathsf{op}, \mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}),\ \forall \mathsf{op} \in f.$$

Compute:

1. (Input) each party in $\mathcal{P}$ encrypts its input x_i as

$$c_i \leftarrow \mathsf{MHE.Encrypt}(x_i, \mathsf{cpk})$$

and sends c_i to H.

2. (Eval) the helper H computes the encrypted output as

$$c' \leftarrow \mathsf{MHE.Eval}(f, \{\mathsf{evk}_{\mathsf{op}}\}_{\mathsf{op} \in f}, c_1, c_2, \ldots, c_N),$$

and sends c' to the parties in $\mathcal{P}$.

3. (Output) any subset of parties $\mathcal{P}' \subseteq \mathcal{P}$ with $|\mathcal{P}'| \geq T$ re-encrypts the output c' under the receiver's key as

$$c'_R \leftarrow \mathsf{MHE}.\Pi_{\mathsf{Decrypt}}(\mathsf{sk}_1, \ldots, \mathsf{sk}_{|\mathcal{P}'|}, \mathsf{pk}_R, c').$$

5.2.2 The MHE-Based MPC Protocol

As discussed in Chapter 2, an MHE scheme can be used to construct a generic MPC protocol, over the scheme's plaintext space. We recall such a protocol as $\Pi_{\mathsf{MHE\text{-}MPC}}$ (for the helper-assisted setting and public output) as Protocol 8. The $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol has two phases, Setup and Compute, each of which consists in running several MHE protocols, i.e., as *sub-protocols*. During the Setup phase, the parties collectively run the MHE key-generation sub-protocols in order to generate the private and public key material required for the next phase. They generate a collective encryption-key cpk (with $\Pi_{\mathsf{EncKeyGen}}$) for encrypting the inputs and all the evaluation keys (with multiple calls to $\Pi_{\mathsf{EvalKeyGen}}$) required to evaluate the target function. During the Compute phase, the parties encrypt their inputs under the cpk, evaluate the target function under homomorphic encryption (with Eval), and collectively decrypt the result (with Π_{Decrypt}).

Cloud-Assisted $\Pi_{\mathsf{MHE\text{-}MPC}}$ In this work, we consider the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol in the cloud-assisted setting [MTBH21]. In this setting, a helper node H assists the parties in the protocol execution. The role of the helper is two-fold: (i) It computes the homomorphic circuit on the parties' encrypted inputs during the Eval step (i.e., it acts as an *evaluator*), and (ii) it assists the parties with the execution of the sub-protocols, by collecting their shares, aggregating them, and broadcasting the result (i.e., it acts as an *aggregator*).

Ideally, the Eval step of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol is non-interactive (i.e., the evaluator alone performs the HE operations on the parties' inputs). In practice, however, interacting with the helper enables the parties to improve performance by replacing the traditional FHE *bootstrapping* that is costly and requires a large number of rotation keys to be generated in the Setup phase, with a single-round multiparty sub-protocol that refreshes the ciphertexts. This sub-protocol functions as a simultaneous (single-round) decryption and re-encryption. In order to execute successfully, it requires at least T parties to be online.

Monolithic Execution Typically, theoretical works [AJLT+12; MTBH21; MBH23] assume that $\Pi_{\mathsf{MHE\text{-}MPC}}$-like protocols are composed by four broadcast rounds: SecKeyGen, PubKeyGen, Input, and Output (one for each interactive step of Protocol 8) which are executed as a monolith. This means that (i) the parties execute these rounds in a predefined order, (ii) for any round that involves multiple sub-protocols, the parties compute a single *round-share* as the concatenation of the involved sub-protocols' shares, and (iii) that the protocol terminates after the Output round.

Such a monolithic execution ensures that, by design, parties are synchronized. This makes it trivial to realize assumptions such as the access to a common random string (CRS) by, among others, the public-key generation sub-protocols ($\Pi_{\mathsf{EncKeyGen}}$, $\Pi_{\mathsf{EvalKeyGen}}$). This common string ensures that randomness (i) is the same for each party, (ii) is fresh for each sub-protocol execution, and (iii) is uniformly distributed in the ciphertext-space ring. Such a CRS is typically implemented by assuming that each party has access to a fixed-length random string that can be expanded into an arbitrary length string by using it as a seed to a keyed-PRF (e.g., BLAKE2b [ANWW13]). Synchronization ensures that all parties' keyed-PRFs are in the same state and thus they read the same (pseudo)random values when executing protocols.

However, although the monolithic execution is convenient in theory, it is hard to realize it in practice, as it leads to practical challenges and does not fulfill the requirements of Section 5.1 in terms of workflow. In contrast, a non-monolithic execution depicts more practical and realistic settings but requires addressing a number of challenges that we detail in Section 5.2.3.

5.2.3 Practical Challenges

Several challenges arise when we concretely implement the $\Pi_{\mathsf{MHE\text{-}MPC}}$ solution in practice. Here, we provide an overview of these challenges and outline our solutions.

Although Chapters 2 and 3 formulate the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol in terms of independent sub-protocols, our security and correctness analysis for these chapters still consider a monolithic execution (and, for the security analysis, only the $T = N$ case without retries).

Challenge 1. Resource-Constrained Clients: The first challenge is that some parties might have constrained hardware resources, e.g., low-power CPU and small RAM. As such, the monolithic execution of MHE sub-protocols in the PubKeyGen step (see step 2. in the Setup phase of Protocol 8), for which the round share per party is the concatenation of many $\Pi_{\mathsf{EvalKeyGen}}$ shares (i.e., one per homomorphic operation op to be supported by the Eval algorithm), can easily overflow the available memory of weak parties (e.g., when a circuit requires a large number of distinct rotation values). For a reference example, recall that the monolithic share size of a single party in the Poseidon [SPTF+21] encrypted neural network training circuit when executed for the MNIST dataset is 5.3 GB (see Section 3.5.3). The solution to this challenge is intuitive: Instead of computing a single monolithic share, we propose to run the sub-protocols independently and asynchronously. This enables the parties to limit the number of concurrently running protocols and to execute them in a streamlined way. However, this also creates new challenges in preserving the correctness and security properties of the MHE sub-protocols (we will expand on these challenges when further defining the structure of MHE protocols, as Challenge 5).

Challenge 2. Session-Like Execution: Another shortcoming of a monolithic execution flow is that it is overly restrictive in practice: A crucial aspect of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol is that, after the execution of the input-independent Setup phase, the parties can execute an arbitrary number of Compute phase iterations. The monolithic execution does not capture this aspect, as it is assumed to terminate after the output is received. In practice, the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol could be long-lived and, we argue, is better framed in terms of a *session*: a logical computation context for which the data access-control is cryptographically enforced (through a collective secret-key). This change of paradigm is not only a matter of terminology: To implement such a long-lived execution, we need to account for parties that temporarily disconnect from the system, and we require that the re-connection of such *churning* parties must be safe and efficient (see Challenge 3).

Challenge 3. Churning Clients: Asharov et al. propose to handle party failures by having each party re-share its share of the secret key with a T-out-of-N-threshold sharing scheme during the Setup phase. This approach enables the online parties to publicly reconstruct the shares of the failing parties in order to simulate them for the rest of the protocol execution [AJLT+12]. However, doing so *permanently alters* the security provided by the encryption under the cpk (which then provides $T-1$-out-of-N-threshold encryption) and does not enable the failing parties to reconnect to the session at a later point (Challenge 2). As a result, securely re-integrating the returning party would require interaction from all parties, to either renew the parties' secret-key shares (in a *proactive* fashion [HJKY95]) or to re-create a session from scratch.

Instead, we use the T-out-of-N-threshold scheme introduced in Chapter 3: It enables each MHE sub-protocol to be run with any T-subset of parties, yet without reconstructing the failing parties' share in plaintext. Our scheme is highly efficient, but it introduces two constraints on the system: (i) The sub-protocols require a list of at least T online parties as input to their

GenShare operation, and (ii) a given sub-protocol could fail in the event of a party from the list crashing before it provides its share. This introduces the challenge of synchronising the parties on the protocol outcomes and of securely implementing protocol retries (Challenge 4).

Challenge 4. Secure Sub-Protocols Retries: Failure and retries of sub-protocols are not captured by the existing security analysis of MHE-based MPC that assumes either the $T = N$ case [MTBH21] or the single monolithic execution case (in which fault-tolerance can be achieved by simply reconstructing the failing parties' share in plaintext [AJLT+12]). In our case, the use of a more complex scheme creates the additional challenge of securely instantiating the sub-protocols and their potential retries. Intuitively, these challenges arise because a system that does not emulate a monolithic execution is at risk of generating correlated shares; and this could enable cryptanalytic attacks. We elaborate on such attacks and propose a secure way of executing protocol retries in Section 5.3.

5.3 Solution Design

Here, we present our general solution for a practical execution of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. We begin by observing that the end goal of this solution is to instantiate an *environment* for the execution of the MHE sub-protocols in both the Setup and Compute phases. To achieve this, the environment must implement the following functionalities: (i) to securely initialize and orchestrate the execution of MHE sub-protocols, (ii) to transport the parties' shares in these sub-protocols, and (iii) to make the protocols' output available to the parties. Moreover, this environment must address the practical challenges discussed in Section 5.2.3: It must implement a non-monolithic execution to address Challenges 1 and 2, and it must handle churning parties and protocol retries to address Challenges 3 and 4.

The remainder of this section is organized as follows: In Section 5.3.1, we map the entities in our system definition (of Section 5.1) to *nodes* in our concrete solution. Then, in Section 5.3.2, we define the notion of a *session* in MHE-based MPC as a logical secure-computing context. In Section 5.3.3, we describe our proposed mechanism for executing, in a non-monolithic fashion, the MHE sub-protocols, and we extend this mechanism with a failure-handling mechanism in Section 5.3.4. Overall, this section describes a generic environment for executing MHE sub-protocols that we will, in Section 5.4, instantiate as the core part of a generic MPC system called HElium.

5.3.1 Nodes

We refer to all actors in our system as *nodes*. We assume that all nodes are associated with an identifier, which is a unique string of characters provided by the high-level user application. We also assume that the nodes hold certificates for these identifiers, hence they can authenticate themselves within a PKI model. As per our system model (Section 5.1), there are two types of nodes: *Session nodes* are the parties in $\mathcal{P}$, i.e., they have inputs to the computation hence hold a share of the collective secret-key (to enforce their inputs' access-control). *Helper nodes* are nodes that do not have inputs to the computation hence do not hold a share of the secret key yet assist the parties in the computation. Note that in our system model, there is only one such helper node (denoted by H).

5.3.2 Sessions

As highlighted in Challenge 2, the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol is most efficiently instantiated as a long-lived *logical* secure computing context. This is because the keys generated during the Setup phase can be reused for an unlimited number of iterations of the Compute phase that can amortize the potentially high cost of the key generation. Such a long-lived context is usually captured by the notion of a *session* in secure communications[1], and we extend this notion to secure computations.

The purpose of a session in the environment is two-fold. The first purpose is functional: it provides a logical context for the encrypted computation to take place: the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. This context comprises all the cryptographic objects related to a single execution of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, such as the cryptographic parameters, the secret and public key-material generated in the Setup phase, as well as the data encrypted under these keys that are the input to the Compute phase. The second purpose is security-critical: it provides context-separation between multiple $\Pi_{\mathsf{MHE\text{-}MPC}}$ executions. Context separation is crucial for safe crash-recovery, even when considering the execution of a single protocol instance (see Section 5.3.4).

A session is initialized (or re-loaded after a restart) by each node, from the *session parameters*:

$$\texttt{SessParams} := \{\texttt{SessionID}, \texttt{Nodes}, \texttt{HEParameters}, \texttt{PublicSeed}, \texttt{PrivateSeed}\},$$

where `SessionID` is a unique system-wide identifier for the session, `Nodes` the identities of the N session nodes in $\mathcal{P}$, `HEParameters` the MHE scheme parameters, including the threshold T, `PublicSeed` a public bit-string seed for the public randomness source initialization, and `PrivateSeed` a private bit-string seed for the private randomness source initialization. The `SessionID`, `Nodes`, and `HEParameters` are the result of the conception phase (Phase 0 in §5.1.1) of the higher-level application. The `PublicSeed` requires synchronization among the nodes on a unique random bit-string. One node could be entrusted to generate it. Alternatively, the higher-level application could use existing decentralized randomness beacons[2] [SJKG+17]. The `PrivateSeed` represents the master secret-key of the node and must be sampled from a secure randomness source.

The session parameters constitute the immutable part of the session and must be stored reliably at each node, and we require that they suffice for re-loading a session in a correct and secure way (e.g., after a node crash). Therefore, by minimizing the size of the critical information to be persisted by the clients (typically, to a few kilobytes), we significantly reduce the risk related to its storage (e.g., by reducing the cost of redundancy and enabling the possibility of secure storage solutions such as hardware wallets). The mutable part of the session consists of the cryptographic material generated during the Setup and Compute phases (i.e., the encryption key, the evaluation keys, and the ciphertexts). Importantly, this cryptographic material does not need to be reliably stored by the session nodes and can be re-derived from the session's immutable parameters and some interaction with the helper node. However, this cryptographic material can be stored by the session nodes for efficiency purposes.

From this point onward, our discussion will focus on a single session hence on a single instance of a long-lived $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. For the sake of conciseness, we will refer to the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol as *the session* and to its sub-protocols simply as *protocols*.

[1] `https://www.rfc-editor.org/rfc/rfc8446`
[2] `https://drand.love/`

5.3.3 Protocols

We now discuss how to execute the protocols in a non-monolithic way within the session. We proceed in three steps: First, in Section 5.3.3, we define an abstraction for the MHE protocols. Then, in Section 5.3.3, we discuss our execution mechanism. Finally, in Section 5.3.4, we extend this execution mechanism with failure handling.

MHE Protocol Abstraction

We now define an abstraction that captures the core functionality of the MHE protocols. This enables us to define our execution mechanism in a generic way.

Preliminaries We consider a ring R as the ciphertext space of the MHE scheme (i.e., R is a polynomial ring parameterized such that the RLWE problem is hard [LPR10]). Informally, for a a publicly known element, and for s and e two secret values sampled from low-norm distributions over R, (i.e., the coefficients of these values are small w.r.t. those of a), the distribution of $(as + e, a)$ is computationally indistinguishable from the uniform distribution over R^2.

The $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ protocol privately outputs, to each party $P_i \in \mathcal{P}$, a T-out-of-N-threshold secret-share $s_i \in R$ of the collective secret key s (see Chapter 3). More precisely, $\mathsf{MHE}.\Pi_{\mathsf{SecKeyGen}}$ outputs to each party P_i a point $(\alpha_i, S(\alpha_i))$ of some secret degree-$T-1$ polynomial $S \in R[X]$ for which $S(0) = s$. Hence, any subset $\mathcal{P}'$ of $\mathcal{P}$ with $|\mathcal{P}'| \geq T$ could reconstruct s from their shares $\{s_i\}_{P_i \in \mathcal{P}'}$ as

$$s = S(0) = \sum_{P_i \in \mathcal{P}'} \prod_{\substack{P_j \in \mathcal{P}' \\ P_j \neq P_i}} \frac{\alpha_j}{\alpha_j - \alpha_i} s_i = \sum_{P_i \in \mathcal{P}'} \lambda_i^{(\mathcal{P}')} s_i, \tag{5.1}$$

where $\lambda_i^{(\mathcal{P}')}$ denotes the Lagrange interpolation coefficient for the share of party P_i in the reconstruction among set $\mathcal{P}'$. Indeed, the secret key s is never reconstructed in practice.

The PAT Protocol Abstraction Although MHE schemes consist of many sub-protocols, the core functionality of these protocols is the same: to compute a noisy linear function of the collective secret key s of the form $as + e$, where a is a public polynomial and e is some small error term [MTBH21]. For example, the $\mathsf{MHE}.\Pi_{\mathsf{EncKeyGen}}$ protocol generates a collective public encryption key of the form $(p_0, p_1) = (sp_1 + e_{\mathsf{pk}}, p_1)$ by setting a to be a uniform value sampled from the common random string. Similarly, the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ protocol performs the decryption of a ciphertext (c_0, c_1) in two steps: It first computes a term $h = sc_1 + e_{\text{dec}}$ (i.e., $a = c_1$ in the core functionality); then it computes a noisy message as $m_{noisy} \approx c_0 + h$ which can be decoded into m (provided that the noise is not too large).

To compute this core functionality, the MHE scheme exploits the linearity of the Shamir secret-sharing scheme. It lets the participating parties compute their respective linear terms as shares, then let the parties obtain the result by summing up these shares. As these shares have the form of an RLWE sample, they can be publicly disclosed and aggregated without compromising the parties' secret-keys. Hence, we say that the MHE protocols have *public aggregatable transcripts*, and we refer to them as PAT protocols. More formally, MHE protocols have a common structure that can be expressed as a tuple $\mathsf{PAT} = (\mathsf{GenShare}, \mathsf{AggShare})$ of algorithms with the following syntax and semantic:

- **Share generation** $v_i \leftarrow \mathsf{PAT.GenShare}(s_i, a, \mathcal{P}'; \chi)$:
  From the secret-key share s_i, a publicly known polynomial a and a set of participating parties $\mathcal{P}'$, GenShare outputs a share

$$v_i = \lambda_i^{(\mathcal{P}')} s_i a + e_i, \quad \text{with } e_i \leftarrow \chi$$

- **Share aggregation** $v_{\text{agg}} \leftarrow \mathsf{PAT.AggShare}(\{v_i\}_{P_i \in \mathcal{P}'})$:
  From the shares $\{v_i\}_{P_i \in \mathcal{P}'}$ of the participating parties $\mathcal{P}'$, AggShare outputs a single aggregated share

$$v_{\text{agg}} = \sum_{P_i \in \mathcal{P}'} v_i = sa + \sum_{P_i \in \mathcal{P}'} e_i$$

- **Finalization** $\mathsf{out} \leftarrow \mathsf{PAT.Finalize}(v_{\text{agg}}, \mathsf{in})$:
  From the aggregation of all shares of the parties in $\mathcal{P}'$ and some public auxiliary input polynomial in, Finalize outputs the result out of the protocol.

For the key-generation protocols ($\Pi_{\mathsf{EncKeyGen}}$ and $\Pi_{\mathsf{EvalKeyGen}}$), the PAT.Finalize algorithm takes the publicly known polynomial a (sampled from the CRS) as input in, and outputs the resulting key as $\mathsf{out} = (v_{\text{agg}}, a)$. For the decryption operation, it takes the element c_0 of the ciphertext as auxiliary input in and outputs the noisy message $\mathsf{out} = m_{\text{noisy}} = c_0 + v_{\text{agg}}$.

Execution Mechanism

Under the PAT abstraction, we design a generic execution mechanism for PAT protocols. Here, we describe this execution mechanism in a form that addresses Challenges 1 and 2, and partially addresses Challenge 3. In Section 5.3.4, we augment the execution mechanism with failure handling, which fully addresses Challenges 3 and 4. Our mechanism relies on several *roles* assumed by the nodes, as well as several *objects* that correspond to synchronization messages between the nodes.

Role: Protocol Participants Each node that provides a share in a PAT protocol is a *protocol participant*. Observe that the PAT.GenShare algorithm requires a set of T protocol participants to generate the share (as a consequence of using the T-out-of-N scheme of Chapter 3 [MBH23]). Thus, for each protocol execution, the set of protocol participants is a fixed subset $\mathcal{P}'$ of the session nodes $\mathcal{P}$.

Role: Protocol Aggregator To avoid broadcast communications and to enable a sublinear communication for the session nodes, we exploit the *public* and *aggregatable* properties of the PAT protocol shares. We designate an *aggregator* that collects the T shares from all parties (generated with the PAT.GenShare method), and it aggregates them (with the PAT.AggShare method). Then, all nodes requiring the output of the protocol can query the aggregator for the aggregated share.

Role: Coordinator The coordinator ensures that the session (i.e., the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol) progresses, by coordinating the execution of the required protocols. To this end, the coordinator keeps track of the network view, i.e., which session nodes are online/offline, and assigns the different roles to the available nodes (both session and helper nodes). Moreover, the coordinator

retains the execution status of the various protocols and we specify further details after defining the two message objects on which the coordinated execution relies.

Object: Protocol Signature We observe that, from a functional perspective, each PAT protocol can be associated with a tuple

$$\texttt{PSig} := \{\texttt{PType}, \texttt{PArgs}\}$$

where $\texttt{PType} \in (\Pi_{\mathsf{SecKeyGen}}, \Pi_{\mathsf{EncKeyGen}}, \Pi_{\mathsf{EvalKeyGen}}, \Pi_{\mathsf{Decrypt}})$ designates the type of protocol and `PArgs` denotes the public inputs (i.e., the *arguments*) of the protocol. For example, the operation `op` for which the $\Pi_{\mathsf{EvalKeyGen}}$ must generate an evaluation key and the ciphertext that Π_{Decrypt} must decrypt, are protocol public inputs. For the sake of the exposition, we also consider the public polynomial a to be part of `PArgs` for now, but we will propose a more efficient way of providing it. We refer to the tuple $\{\texttt{PType}, \texttt{PArgs}\}$ as the protocol's *signature*.

Object: Protocol Descriptors By composing a protocol signature with the identities of the parties that assume the roles of the T participants and the aggregator, we obtain a complete, unequivocal description of a given PAT protocol execution. We refer to this description as a *protocol descriptor* and it is defined as a tuple

$$\texttt{PDesc} := \{\texttt{PSig}, \texttt{PParticipants}, \texttt{PAggregator}\}$$

with `PSig` the protocol signature, `PParticipants` the set of session nodes that provide a share in the protocol, and `PAggregator` the identity of the aggregator for this protocol. The protocol descriptor can be viewed as the *runtime* version of the protocol signature, because the role attribution to the nodes needs to take into account the current state of the system. Whereas, signatures correspond to the functional aspect of the PAT protocol. As a result of this distinction, we can define the notion of *equivalent* protocols, i.e., protocols whose descriptors have equal signatures. This will be useful when designing our fault-tolerance mechanism in Section 5.3.4.

Process: Protocol Execution The first step of both the Setup (resp. Compute) phase of $\Pi_{\mathsf{MHE\text{-}MPC}}$ (see Protocol 8) is to decompose the high-level description of the desired setup (resp. circuit) into a list of *protocol signatures* to be executed by the system. This is done by all the nodes. The second step is the actual execution of the list of protocol signatures, as independent PAT protocols. Doing so involves (i) generating protocol descriptors for each protocol signature, (ii) emitting them to the parties, and (iii) keeping track of which protocols have completed: these are the tasks of the coordinator. This requires deciding on the set of protocol participants $\mathcal{P}' \subset \mathcal{P}$ and on the aggregator (which can be any reachable node over the Internet) for each protocol; the coordinator can do this, based on its view of the network at runtime. Note that different strategies for choosing parties have various effects on the performance and cost figures of the environment. For example, a coordinator could favor performance by relying more on parties with high bandwidth, or it could favor fairness by distributing the load. On the participants' side, the parties receive the protocol descriptor and execute the corresponding protocol. Party $P_i \in \mathcal{P}'$ computes its respective share as $v_i = \Pi_{\texttt{PSig}}.\mathsf{GenShare}(s_i, a, \mathcal{P}'; \chi)$, where the actual protocol is determined by `PSig`, where s_i is provided by the session, and where a and $\mathcal{P}'$ as provided by the `PArgs` and `PParticipants` fields of the protocol descriptor, respectively. Note that this requires the party to initialize the secret error distribution χ. Then each party P_i sends its share v_i to the aggregator identified by the `PAggregator` field of the protocol descriptor. Upon receiving all

the shares for a participant set $\mathcal{P}'$, the aggregator reports to the coordinator that the protocol has been completed successfully.

In an ideal case with no participant failure, the protocol execution process, as described above, would simply emulate a monolithic execution by running each protocol once. However, the coordinator also needs to account for failures (Section 5.3.4).

Public Polynomial We observe that providing the full public polynomial a as a part of the protocol signature is unnecessary for the key-generation protocols, in which it is sampled from a CRS. Instead, the parties can sample it themselves, which divides the communication overhead of these protocols by a factor of 2. As this approach requires considering the case of protocol retries, we also discuss it along with our solution to failure handling.

5.3.4 Failure Handling and Randomness Initialization

Recall that the participant set is defined *before* the participants provide their shares. Hence, the failure of any participant in this set to provide its share would prevent the PAT protocol from completing. The rationale of our approach (and that of the TMHE scheme of Chapter 3) is that the probability of such a failure event can be minimized by having the coordinator decide on the set of participants at runtime, i.e., with knowledge of the current view of the network. For example, the coordinator could choose the participant set that is likely to be responsive, by selecting the nodes that are currently online. This is indeed insufficient, as failures could still occur if a participant disconnects between this decision and providing its share, with a probability that is determined by our failure model (see Section 5.1). This failure case requires a failure-handling mechanism.

To construct such a mechanism, we start from the observation that PAT protocols are single-round and do not require the participants to keep any state (only the aggregator does). As a result, a protocol retry is equivalent to running a fresh, *equivalent* protocol (i.e., a protocol corresponding to the same signature) with a different participant set. As the execution of multiple protocols is already provided by our previous mechanism, this reduces the problem of failure handling to that of defining the criteria and semantics of protocol termination, and to ensuring that protocol retries do not break the security of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol.

Protocol Termination Consistently with our described mechanism, protocols are either *running* or *completed*, and the transition from the former to the latter corresponds to the event that the aggregator has received a share from each protocol participant. As a result, it is the responsibility of the aggregator to report a successful termination to the coordinator, and the coordinator's responsibility (i) to decide which completed protocol output for a given signature should be used for the rest of the session, and (ii) to report the protocol completion to the other nodes. Note that we do not consider a *failed* status, because this would require defining the corresponding event and, ultimately, only a single party would rely on it (i.e., the aggregator, in order to clear its state).

Secure Protocol Retries This failure-handling mechanism preserves the semantics of PAT protocols and, from a functionality perspective, it only requires synchronization between the coordinator and the aggregator. From the security perspective, however, we observe that failure handling introduces a fundamental divergence between the non-monolithic execution and the monolithic one: A single protocol in the monolithic execution could require the execution of several related protocols in our approach. This leads to the challenge of ensuring that the extra

shares do not break the simulatability of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, which we present as Challenge 5. This challenge underlies Challenges 2, 3 and 4, and we now formulate it for the PAT abstraction.

Challenge 5. PAT Protocols Simulatability with Retries: Recall that the parties' shares in PAT protocols (§5.3.3) have the form $s_i a + e_i$, where a is a publicly known polynomial, s_i is the secret-key of the party $P_i \in \mathcal{P}$, and e_i is a fresh polynomial from the noise distribution. These shares are safe to disclose publicly under the RLWE assumption (which provides the public-transcript property), as long as the parties can be simulated as an RLWE challenger: an oracle that outputs either a tuple of form $(as + e, a)$ or a uniformly random tuple in R_q^2, in a distinguishing game. The challenge resides in the fact that the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol requires the environment to control the public polynomial a. More specifically, the public polynomials are freshly sampled from a common random string (CRS) during the Setup phase, and they are ciphertext elements in the Compute phase. This can lead to security issues if the initialization of the public polynomial and of the private randomness sources are not carefully performed. For example,

- *Failure case A*: Consider the case of a party generating two shares for the same protocol, for which the public polynomial is a. The disclosure of both $\lambda_i^{(\mathcal{P}')} s_i a + e_i$ and $\lambda_i^{(\mathcal{P}')} s_i a + e_i'$ cannot be simulated by the RLWE challenger and directly leads to an attack, where an adversary can gain information on $s_i a$, by averaging the two shares.

- *Failure case B*: Consider an (insecure) environment that executes two protocols but uses the same public polynomial over two different sets of parties $\mathcal{P}'$ and $\mathcal{P}''$. This would trigger the disclosure of two related shares $\lambda_i^{(\mathcal{P}')} s_i a + e_i$ and $\lambda_i^{(\mathcal{P}'')} s_i a + e_i'$, whose RLWE secrets are linearly related (by the ratio of their Lagrange coefficients).

We observe that the two failure cases described in Challenge 5 are not restricted to our non-monolithic execution, but they could occur by naively composing multiple monolithic executions (e.g., that use the same CRS but different seeds for the secret distributions). *Failure case A* could occur if the coordinator issues two identical protocol descriptors (as forbidding this would require stronger assumptions), or if a node crashes after providing its share and reboots (as we require nodes to be able to initialize from the session parameters only, and to re-join an existing session). *Failure case B* could occur even in non-malicious scenarios, e.g., in the case of naively retrying protocols, as we discuss below. We now describe how our solution addresses this challenge.

Protocol Public Randomness Initialization To generate the CRS without having to store it (which would be very inefficient), we use the common approach of expanding it from a shorter seed (e.g., a 256-bit string) to a keyed PRF. This seed can be public and used by all the nodes to initialize their PRF; reading from it will yield the same pseudo-random sequence of bits. Although all common random polynomials required by the session could be read, sequentially, from a single CRS in the monolithic execution (as suggested in [AJLT+12; MTBH21]), this is not suitable for our environment. Indeed, this would require the parties to synchronize their reads to the CRS, which is not possible in our model where parties might be offline and where the number of protocols depends on runtime parameters (to support re-tries). Moreover, as discussed above, we need to ensure that common random polynomials are fresh for each protocol. To achieve this, we use one keyed PRF per protocol that we seed with the key `PublicSeed`$||$`PSig`$||h($`PParticipants`$)$, with `PublicSeed` the session-wide seed from the session parameters, and `PSig` and `PParticipants` the protocol signature and the list of participants from the protocol descriptor. $h : \mathsf{Powerset}(\mathcal{P}) \to \{1,0\}^*$ is an injective function that maps participant lists to bit-strings. Through this initialization, the environment ensures that each protocol within

the session uses a fresh public polynomial. This is also valid for retries of equivalent protocols over a different participant set; these retries are consistently treated as new protocols and no longer produce correlated shares. Moreover, we observe that running twice the same protocol produces the same common random polynomial.

Ciphertext Re-randomization As for the key-generation protocols in the setup phase, the secret-key operations should be secure when multiple aggregation phases are performed in parallel. Informally, these protocols operate on an input ciphertext (c_0, c_1) by producing one or multiple shares of the form $sc_1 + e$ for some secret polynomials s and e (i.e., samples from the RLWE distribution). Hence, we cannot simply sample a different c_1 element for each parallel aggregation because c_1 is set by the ciphertext. Instead, we propose to *re-randomize* the ciphertext for each parallel aggregation; this can be done by homomorphically by adding a fresh encryption of zero to the input ciphertext. This is easily achieved from the collective public key (hence does not require interaction), by using the MHE.Encrypt algorithm. In our helper-assisted model, a solution could be to let H produce the re-randomizations when initiating each parallel aggregation. However, this is unsatisfactory for several reasons: (i) for l parallel aggregations, the helper would need to send l re-randomizations to the participants, and (ii) this would make H a single point of failure (which is fine in the strictly applied honest-but-curious model, but would be difficult to extend to more restrictive assumptions). HElium employs a more efficient and elegant solution: it lets the parties re-randomize the ciphertext by themselves, by running the MHE.Encrypt over common random coins. More specifically, for a participant list `PParticipants`, the parties sample the required secret polynomials from a keyed PRF with key

$$\texttt{PublicReRandSeed}|\texttt{CircuitID}|\texttt{ProtocolID}|H(\texttt{PParticipants}),$$

where `PublicReRandSeed` is the public session wide re-randomization seed. The security of using a *publicly re-randomized* ciphertext for generating RLWE samples follows from Lemma 4 in [BV11].

Protocol Private Randomness Initialization To generate the secret polynomials in a secure way and to prevent the attacks mentioned above, we also rely on a keyed PRF, yet this time it is seeded with a private seed from each session node (e.g., a 256-bit string). Similarly as for the public randomness (CRS), we use one PRF stream per protocol keyed with the seed $\texttt{PrivateSeed}||\texttt{PSig}||\texttt{dist}||h(\texttt{PParticipants})$, where `PrivateSeed` is the session-wide private seed, and `dist` is an identifier of the ring element to be sampled (some protocols require sampling multiple terms). Through this initialization, our environment ensures that the participants use fresh secret values when generating their shares for each protocol and that they never output two different shares for the same protocol in the same session (which breaks security as mentioned above).

Note on randomness initialization We note that the issue of secure randomness initialization does not appear in the current theoretical works on MHE schemes, due to the assumption of a monolithic execution. A notable symptom of this is that current libraries implementing MHE do not provide the user with control over the error distribution initialization (instead, they initialize them from the OS secure randomness source directly). As a result of this work, we added in Lattigo the possibility of initializing the secret distribution.

5.4 HElium

Here, we describe HElium our end-to-end implementation of the MHE-based MPC protocol. HElium instantiates the generic solution of Section 5.3 for running the MHE protocols in a non-monolithic way and in the helper-assisted setting described in Section 5.1.

Helper-Assisted Execution Recall that, in our helper-assisted model, the parties receive assistance from a high-end and reliably connected third party H (e.g., a server in the cloud). We, therefore, attribute the roles of aggregator and coordinator to the helper H. For the aggregator role, this ensures (1) that the aggregator is reachable, and (2) that the resource-constrained session nodes have optimal (constant) overhead. Moreover, this also means that protocol participants do not have to be simultaneously online during the protocol execution, only the helper does. For the coordinator role, this greatly simplifies HElium's design: It delegates all the complex synchronization mechanisms needed to drive the protocol execution, such as making decisions on the status of a protocol (which would otherwise require some form of consensus) and obtaining a consistent view over the network, to a single honest-but-curious node.

Two-Service Design The $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol consists of two phases, Setup and Compute, that can be executed in parallel and without terminating (as discussed in Challenge 2). To address this requirement, HElium relies on a two-services design:

The *Setup* service implements the Setup phase. This service consumes *setup descriptors* that are high-level descriptions of the setup phase. Intuitively, a setup descriptor is the list of keys required in the setup phase, along with the intended recipients of these keys (we further discuss setup descriptors in Section 5.4.2).

The *Compute* service implements the Compute phase. This service consumes *circuit descriptors* that define the function to be evaluated. Intuitively, the compute descriptor is a homomorphic circuit with its input gates labelled with the identities of the session nodes that must provide them, and with special gates for the MHE protocols (e.g., the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ gate for output gates). We further discuss circuit descriptors in Section 5.4.3.

5.4.1 Protocol Execution

Our implementation of the solution of Section 5.4 relies on the helper (as coordinator) sending synchronization messages to the nodes and three concurrent routines (+ one initialization routine) running at each node. The synchronization messages are tuples of the form:

$$\texttt{SynMsg} := \{\texttt{PDesc}, \texttt{PStatus}\}$$
$$\texttt{PStatus} \in \{\texttt{Started}, \texttt{Completed}\}$$

and serve the purpose of synchronizing the nodes that a protocol with descriptor `PDesc` has been started (`Started`) or has successfully completed (`Completed`). The routines are as follows:

1. *(initialization)* All nodes derive the list of protocol signatures to be executed, from either the setup description (in the Setup service) or from the circuit description (in the Compute service). The helper H puts all the protocol signatures to be executed in a queue.

2. *(registration)* Upon startup, each session node connects to the helper node and waits for protocol update messages. An incoming message with `PStatus` = `Started` is passed to the node's execution routine, whereas a message with `PStatus` = `Completed` is passed to

the node's finalization route. Upon connection of a session node, the helper node sends the list of currently started and completed protocols, and it leaves the connection open for sending new synchronization messages. Upon reaching the end of the signature queue, the helper closes the connection.

3. *(execution)* The helper node starts sub-protocols (i) by popping the head of the signature queue, (ii) by creating a protocol descriptor from the signature, and (iii) by sending a protocol synchronization message with the created protocol descriptor and `PStatus` = `Completed` to the connected session nodes. To create the protocol descriptor, the helper, according to its view of the network, chooses the participant list and sets itself as the aggregator. Upon reception of protocol aggregation messages, the session nodes that are in the participant set send their shares to the helper node. Upon reception of all the expected shares for an aggregation, the helper node sends a synchronization message with `PStatus` = `Completed` to the connected nodes.

4. *(finalization)* Upon receiving a synchronization message with status `Completed`, the session nodes that are the intended recipients of the protocol's output (as determined by the setup or circuit description) query the helper for the aggregation and compute the protocol's output with its PAT.Finalize algorithm.

From this design, we obtain a streamlined execution flow where nodes control the workload by setting a limit on the number of concurrently executing protocols. This execution flow addresses Challenge 1. Similarly, our implementation directly provides a natural congestion control mechanism on the helper side (as an aggregator); this mechanism can control its inbound traffic by limiting the number of parallel protocol executions. Finally, this three-routine solution already accounts for session nodes that connect at any time (including after all protocols have completed) and enables them to simply retrieve (if needed) the result of a protocol as a part of their finalization routine. Note also that, although the helper could send the result of a protocol directly with the completion message, it is preferable to let the nodes fetch the required protocol results themselves. Doing so keeps the synchronization messages light and lets the weaker node control the amount of incoming data.

Failure Handling Centralizing the coordination also greatly simplifies the implementation of a failure-handling mechanism, by removing the need for consensus on the protocol outcomes. Instead, the coordinator can rely on local timeouts to decide whether to retry a protocol. As per our solution to failure handling described in Section 5.3.4, retrying a protocol corresponds to starting a new *equivalent* one (i.e., that has the same protocol signature). We implement this mechanism by having the coordinator put signatures of timed-out protocols back into the signature queue (see the *initialization* routine above).

We use the following strategy for retries: Instead of considering timed-out protocols as failed, the helper simply considers them as running and keeps the related state, i.e., the partial aggregation of the participant's shares, in memory (which follows our general failure handling approach of Section 5.3.4). Then, instead of a simple signature queue, the helper uses a priority queue and executes the retries with higher priority. This is because the successful completion of a protocol retry enables the helper to unload the resources not only for that protocol, but for the timed-out protocols as well.

5.4.2 The Setup Service

This service implements the Setup phase of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol: It generates the public encryption and evaluation keys as required for the Input and Eval phases (Section 5.2). On the user-facing side, it provides an interface for specifying which keys must be generated. Internally, it runs the required PAT protocols, as described in Section 5.4.1.

Setup Descriptors The setup service takes as input a high-level description of the Setup phase, that we refer to as setup descriptors. Such a description corresponds to the desired outcome of this phase: a mapping from each possible public-key (i.e., the encryption key cpk and all the evaluation keys $\mathsf{evk}_{\mathsf{op}}$ for each operation op in the homomorphic circuit) to the set of nodes that require this key in the Compute phase. HElium assumes that this mapping is passed to each node by the user application and is agreed upon by the parties. This is a realistic assumption, because the setup descriptor can be derived unambiguously from the circuit description that, in turn, must also be agreed upon in any secure multiparty computation scenario (as parties must accept the leakage of the output). For this same reason, HElium also accepts circuit descriptions (which we further define in Section 5.4.3) as a setup description. It then relies on symbolic execution of the target circuit to infer the set of required operations and produces a corresponding setup description.

Persistence In our session-like execution, nodes need to be able to re-join the session after a reboot or a crash (Challenge 3). Although we require that a node can do so from the session parameters only (for security reasons, see Section 5.3.2), such a *stateless* startup requires interacting with the helper, to retrieve the necessary setup protocols' outputs (e.g., the collective encryption key cpk). This is prevented by enabling session nodes to store the required keys in a persistent storage.

Implementation Our helper-assisted protocol execution mechanism as described in Section 5.4.1 constitutes the essential part of the Setup service implementation. In the *initialization* routine, the nodes parse the setup descriptor and derive the list of corresponding protocol signatures to be executed for generating the corresponding keys. In addition to this, each node derives a list of protocol signatures for which it requires the output and passes this list to the *finalization* routine.

The only difference with the general execution mechanism is that nodes need to take into account the state in their persistent storage. We implement this behavior in the following way: In its *initialization* routine, the helper node checks its persistent storage for signatures that have already been successfully computed. For such signatures, the helper loads their corresponding protocol descriptor and output from the storage, marks the protocol as completed, and does not queue the corresponding signature for execution. In their *finalization* routine, the session nodes first attempt to use their persistent storage to load the outputs of the relevant protocols instead of querying them. In case the required result is not present in the storage, they query the helper for that result (upon receiving the completion message) and update their storage. Regardless of the source from which they retrieve the protocol output, the parties call the PAT.Finalize method to obtain the required key.

Observe that, for a node that connects to the network after the whole setup has completed, i.e., with an empty key-store, the execution of the setup corresponds to downloading the result of each required protocol (i.e., as all protocols have completed states, no action is performed during the *execution* routine).

Long-Running Setup Phase Consistent with the view of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol as a session, the Setup phase can continue running after the Compute one has begun to further populate the session with new evaluation-keys (i.e., that are required at a later stage, for new circuits). In fact, HElium runs the two phases in parallel; the compute service's routine waits for the required public keys to be provided by the setup service. As in the helper-assisted model, the session nodes only require the public encryption key, this enables them to start the Compute phase (i.e., to encrypt and provide their inputs) while the evaluation keys are being generated, potentially reducing the latency of the overall computation.

5.4.3 The Compute Service

This service implements the Compute phase of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. This phase corresponds to the homomorphic evaluation of the target circuit followed by a collective decryption (see Section 5.2). We start by observing that this phase can be modeled as two consecutive PAT protocols: In the first one, PAT.GenShare corresponds to the MHE.Encrypt operation (the Input step) and the PAT.AggShare corresponds to the homomorphic evaluation with MHE.Eval (the Eval step). The second one is the $\mathsf{MHE}.\Pi_{\mathsf{Decrypt}}$ protocol applied to the evaluation result ciphertext. More complex circuits might require refreshing the ciphertexts during the Compute phase, which can be done with the $\mathsf{MHE}.\Pi_{\mathsf{ColBootstrap}}$ (see Section 2.2.7) that is single-round PAT protocol. In such cases, the online phase is modelled as a $2 + R$ round PAT protocol, where R is the number of refresh rounds.

Role: Evaluator The aggregator of the first PAT protocol is commonly referred to as the *evaluator*, as it carries out the homomorphic evaluation of the circuit. In HElium, this task, along with the coordination of the overall evaluation process, is assumed by the helper node H.

Three-Plane Service-Design The compute service operates over three logical planes: the *circuit*, *data*, and *protocol* planes, which we detail below. The circuit plane is the top-level plane: it is responsible for handling the evaluation of the circuit from the circuit descriptor. To do so, the circuit plane makes calls to the data and protocol planes, to resolve the session nodes' inputs and the decryption/refresh protocols' results. We now describe each of these planes.

The Circuit Plane

From a high level, and as for any MPC system, the circuit plane takes as input the representation of the target circuit in *some* language and acts as an interpreter for this language. Unfortunately, there exists no well-established language that is specially designed to represent HE circuits (and although it would be an interesting extension to HElium, designing such language and its interpreter is outside the scope of this work). Therefore, we opt for a simpler solution in HElium: to provide the user application with a Go interface for building MHE circuits. This interface (named `helium.EvaluationContext`) exposes the usual HE operations (by including the Lattigo `Evaluator` interface, see Chapter 4) as well as IO primitives (i.e., labelled input and output gates) and the special gates for the MHE protocols used in the Compute phase (i.e., the Π_{Decrypt} and $\Pi_{\mathsf{ColBootstrap}}$ protocols). HElium programs are therefore Go functions that take as input a `helium.EvaluationContext` interface type, and their execution is directly handled by the Go language. Listing 5.1 provides an example of a simple HElium program for computing a component-wise vector product between two parties.

```
func(ec helium.EvaluationContext) {
    op1 := ec.Input("//node-a/in")    // read node-a's inputs
    op2 := ec.Input("//node-b/in")    // read node-b's inputs
    res := ec.MulNew(op1, op2)        // multiply the inputs
    ec.Relinearize(res, res)          // do relinearization
    resDec := ec.Decrypt(res)         // decrypt the result
    ec.Output("/out", resDec)         // output the result
}
```

Listing 5.1: The HElium program for two-party component-wise vector multiplication.

This Go-interface-based approach has several benefits: First, we do not need to define a specific interpreter. Instead, we exploit Go's execution directly. Second, we enable the user application to fully control the circuit execution flow, which includes exploiting Go's built-in parallelism primitives and hardware accelerators. Finally, our approach includes the possibility for the user to use or design its own language and interpreter, as long as this can be initialized and run from a Go function.[3]

The circuit plane executes the homomorphic operation methods of the `EvaluationContext` by making calls to the Lattigo library, and it relies on the *data* and *protocol* planes for the IO and protocol methods, respectively. Note that the circuit planes of all nodes (i.e., including the session) execute the HElium program, but the internal implementation of the `EvaluationContext` differs: Upon calling the *Input* function at a session node, the node encrypts its plaintext input and registers it in its data plane. Upon calling the *Input* function at the evaluator, the evaluator queries the corresponding ciphertext operands from its own data plane. We further specify the registration and queries of ciphertext operands when we discuss the data plane. Upon calling an FHE arithmetic function, the evaluator calls the corresponding method of the Lattigo library while the session nodes do nothing (i.e., they simply perform a *symbolic execution*). Upon calling a protocol function, the session nodes wait for the corresponding protocol descriptor from the evaluator; and the evaluator creates the corresponding protocol signature, passes it to its protocol plane, and waits for its completion. We further specify the protocol plane in the following.

The Data Plane

The data plane handles the transport and provisioning of data to the circuit plane. Its principal functionality is to coordinate the transport of ciphertext data. In this regard, we begin by observing that data transport in the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, due to the public nature of the transported objects (i.e., the ciphertexts), follows the same principles as an ordinary (plaintext) computing system: The data can be safely sent, made available, and queried between nodes (although we generally avoid moving data, as much as possible). As a result, we can reuse the paradigms and techniques of plaintext computing and data-retrieval systems directly in MHE-based MPC.

One such paradigm is that of a resource that can be identified and/or within a system from a *universal resource identifier/locator*, i.e., a URI/L. To support this paradigm, HElium employs a natural URI scheme for identifying ciphertexts within the session, where each ciphertext can be identified by its holder node identifier, circuit identifier, and ciphertext identifier:

`helium://<NodeID>/<SessionID>/<CircuitID>/<CiphertextID>`

Depending on the context, parts of the identifiers can be omitted by the user application

[3] One limitation of this approach is that runtime-defined functions loading requires the Go plugin mechanism that is not currently supported by all platforms.

designer (as in Listing 5.1), and expanded by the framework at execution time. This enables the application designer to define HElium programs in a generic way and to execute them several times (see Challenge 2). For instance, the `SessionID` and `CircuitID` fields can be expanded to the session identifier in which the circuit was executed and the circuit identifier that was attributed to it. Similarly, the `NodeID` field (the host part) can be omitted for intermediate values and the output, as it can be assumed to be the circuit's evaluator.

Note that when all three fields are present, the URI enables a ciphertext to be located unambiguously within the system, hence is also a URL. This URL, along with a traditional transport protocol that supports *get*- and *post*-types of queries (e.g., HTTP, FTP, or more advanced RPC protocols), constitute a sufficient solution for our data plane.

The Protocol Plane

Our helper-assisted protocol-execution mechanism, as described in Section 5.4.1, constitutes the essential part of the protocol plane implementation. The protocol plane maintains a queue of PAT protocol signatures to be executed at the helper side (as a coordinator), and runs the routines that issue/execute protocol descriptors at the helper (as aggregator)/session nodes (as protocol participants), respectively.

5.5 Implementation and Evaluation

We implemented HElium in Go. We rely on the Lattigo library (see Chapter 4) for the local cryptographic operations, such as the homomorphic operations and the GenShare methods of each MHE sub-protocol. For the transport layer, we use the gRPC framework[4]. This framework provides a service abstraction in the *remote procedure calls* (RPC) paradigm and enables the generation of client and server stubs from a high-level API definition language. The gRPC framework also supports message streaming that we employ to support streams of shares and protocol descriptors (as required by the protocol execution mechanism described in Section 5.4.1), as well as mutual TLS authentication.

5.5.1 Experimental Evaluation

To evaluate HElium as an MPC solution, we instantiate the framework for two MPC scenarios: the component-wise vector product (named Experiment I) and multiparty input-selection circuits (named Experiment II) scenarios of Chapter 2. Recall that, due to the lack (at the time) of an $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol implementation, Section 2.5 simulated the MHE solution in a single program, executed by a single machine. Hence, the present section revisits these experiments, by presenting a realistic execution between machines connected over a LAN network.

Parameters For consistency with the setting of Section 2.5, we consider the helper-assisted setting and the strictest threshold parameter $T = N$. We also consider the BFV scheme and employ the parameters of Table 5.1. Note that, due to several improvements of the Lattigo library that are posterior to the results presented in Section 2.5 (such as our improved key-switching procedure, see Section 4.2.2), we use a slightly different parameterization in the present experiments. This is, we reserve a *special modulus*, denoted p, for the key-switching operations.

[4] `https://grpc.io`

Table 5.1: BFV scheme parameters

Set	$\log_2 t$	$\log_2 n$	$\log_2 q$	$\log_2 p$	σ	sec. (bits)
I	32	14	328	110	3.2	128
II	32	13	162	55	3.2	128

Experimental Setup We run all nodes on two machines with Intel Xeon E5-2680 v3 processors (2.5 GHz, 2×12 cores) and 256 GB of RAM, i.e., the same machines as for the experiments of Section 2.5. To simulate a realistic setting, we run the helper in one of the machines and all the clients in the other one (recall that the clients only communicate with the helper in the helper-assisted model). The machines are connected using a LAN network of 30 Gbits/sec. with a 0.1ms latency. We take the MP-SPDZ values directly from Section 2.5 (note that these values might not reflect improvements brought to the MP-SPDZ library after January 2020). For each MPC solution, we report:

- *Setup-phase latency and communication costs*: These are the costs that are independent of how many circuit evaluations are performed.
- *Offline-phase latency and communication costs*: These are the costs that are related to a single circuit evaluation and that can be performed before the inputs are available.
- *Online-phase latency and communication cost*: These are the costs related to a single circuit evaluation after all the previous phases have been completed.

Note that, as the MHE-based solution has no offline phase, its latency for that phase is zero. Similarly, the LSSS-based solution has a negligible latency for the setup phase.

Results Figures 5.1 and 5.2 present the results for the component-wise vector product scenario (Experiment I). Recall that, in this scenario, each party holds a private vector $\mathbf{x}_i$ of dimension 2^{14} with 32-bit coefficients and the multiparty circuit computes the element-wise product between all the parties' vectors. For more details on the circuit and on the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol instantiation, see Section 2.5.3.

We observe that, consistently with the previously observed results of Chapter 2, HElium provides a clear advantage over the LSSS-based system for $N > 2$. This is even true (although with a smaller gap) when considering the MHE setup as part of the circuit evaluation cost (e.g., in a scenario where a single circuit execution is performed for the whole session). When considering the per-circuit-evaluation costs only (e.g., when performing multiple circuit evaluations within the session and amortizing the setup cost), the cost of HElium is comparable to the LSSS-based system for $N = 2$ and has a 4.2× and 13.5× smaller latency for $N = 4$ and $N = 8$, respectively. In Figure 5.2, we observe that HElium's communication costs are independent of the number of parties and result in a 1×, 7.5×, and 39.3× smaller communication size than MP-SPDZ for $N = 2$, $N = 4$, and $N = 8$ parties respectively.

Figures 5.3 and 5.4 present the results for the multiparty input selection scenario (Experiment II). Recall that this scenario lets party P_1 (the *requester*) select one input among N other parties' inputs (the *providers*), hence it corresponds to a generalized oblivious transfer functionality. Note that N stands for the number of providers in this scenario (hence this scenario is an $N+1$-party computation). The providers' inputs are vectors in $\mathbb{Z}_p^d$ for $d = 2^{13}$ for a 32-bit prime number p, and the requester's input is a unit vector for which all coefficients are 0 except the i-th one, to encode a query for the input of provider P_i (see Section 2.5.2 for more details on the circuit and the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol instantiation).

We observe that HElium has latency lower than the LSSS-based solution for all numbers of parties, even when considering the MHE setup as part of this latency. Interestingly, it is

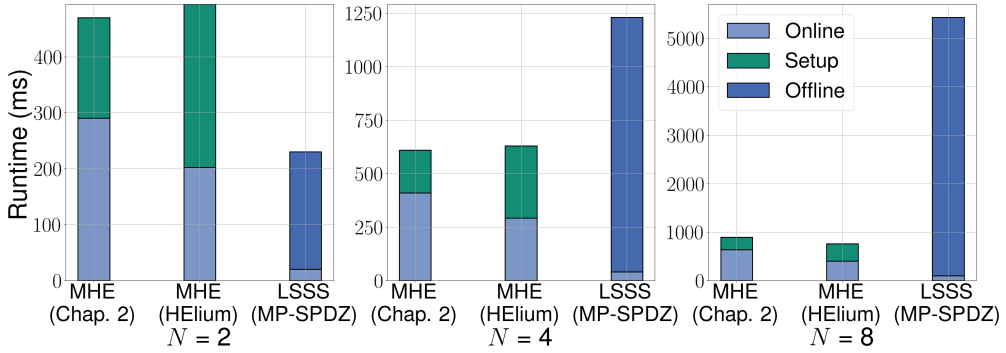

Figure 5.1: Experiment I (component-wise vector product) phase latency in milliseconds.

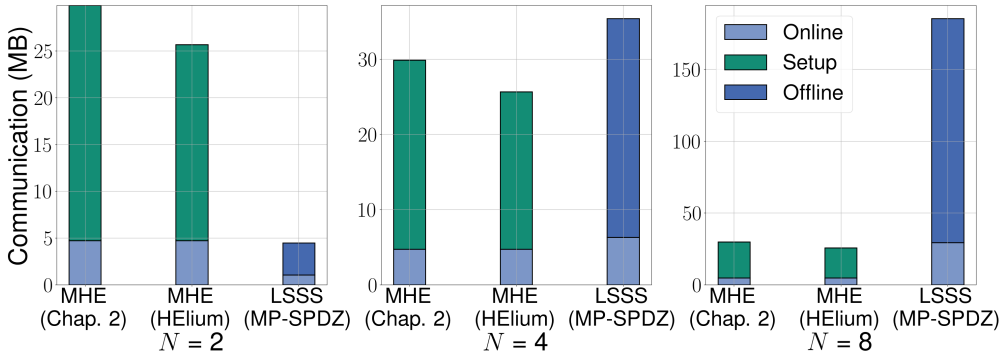

Figure 5.2: Experiment I phase communication cost (upload+download) in megabytes.

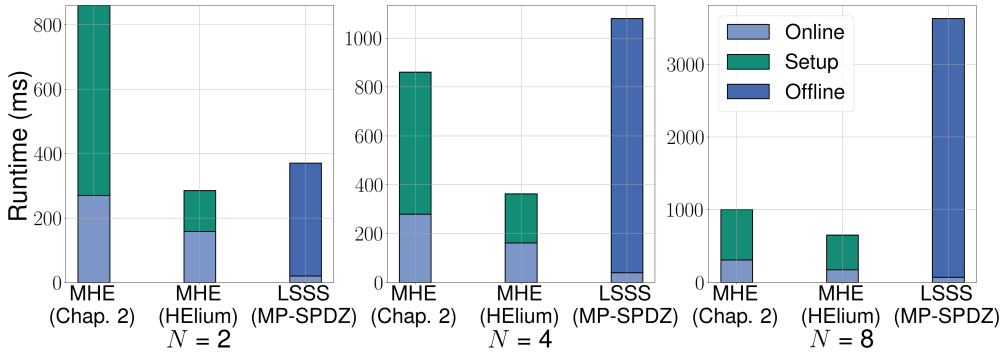

Figure 5.3: Experiment II (multiparty input selection) phase latency in milliseconds.

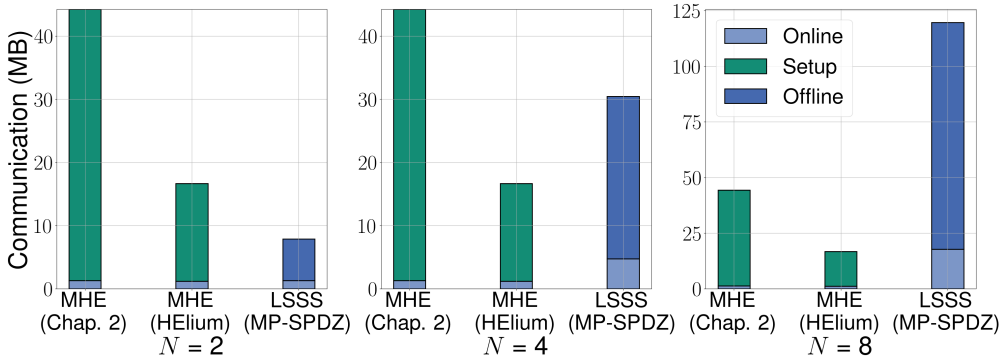

Figure 5.4: Experiment II phase communication cost (upload+download) in megabytes.

also significantly faster than the previous results obtained in Chapter 2, to a point where the previous observation now also applies to the $N = 2$ case (which was not the case for the previous results). This is mostly due to improvements to the key-switching algorithm and to the related evaluation-keys in Lattigo. When considering the per-circuit-evaluation latency, HElium is 2.3×, 6.7× and 21.1× faster than the LSSS-based system for $N = 2, 4, 8$, respectively. We also observe again that HElium's communication costs are increasingly more efficient than MP-SPDZ with the number of parties; HElium has 6.7×, 25.8×, and 101.4× less communication than MP-SPDZ for $N = 2$, $N = 4$, and $N = 8$ respectively.

5.6 Chapter Summary

We proposed HElium an end-to-end implementation of the MHE-based MPC protocol of Chapter 2. To propose a practical solution, we isolated the challenges related to executing this protocol in realistic environments, in which the parties might have limited hardware resources and/or unreliable network connection. We pointed out that a monolithic execution is not realistic in this setting, and that it is also undesirable when viewing the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol as a long-lived session. To address these challenges, we proposed a non-monolithic execution of this protocol, as well as solutions to the security and correctness challenges further introduced by this execution. We implemented our generic solution and evaluated it by revisiting the experiments presented in Chapter 2. We showed that HElium provides a highly efficient solution even in the churn-free setting already supported by existing tools. The short-term next steps in this regard are to evaluate our system for MPC under churn.

We will also make our implementation open-source, for the community to use. As such, HElium is the first implemented MPC framework based on MHE, which enables the use of these techniques in practice. Moreover, due to properties of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol and our non-monolithic execution, HElium is also the first framework to provide a solution to MPC under churn that does not require non-cryptographic assumptions (e.g., non-collusion) besides the traditional passive-adversary setting.

Chapter 6

Conclusion and Future Work

In this dissertation, we have addressed the problem of bringing MHE, and the MPC protocol it enables, from theory to practice. To do so, we have proposed several contributions to both the theoretical and practical sides.

We have introduced a N-out-of-N-threshold MHE scheme based on RLWE and have instantiated this scheme in an efficient and versatile MPC solution, the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. In doing so, we have re-visited the theoretical LWE-based construction by Asharov et al. [AJLT+12] by adapting it to RLWE, and by proposing several improvements to its sub-procedures. This includes a more efficient protocol for generating the relinearization key and a generalized decryption protocol that enables the interactive bootstrapping of a ciphertext and the conversion from/to LSSS shares. We have implemented and evaluated the resulting scheme, as an MPC solution, and we have demonstrated that it provides a concretely efficient solution that can outperform the implemented LSSS-based state-of-the-art solutions.

We have also extended our MHE scheme to T-out-of-N-threshold access-structures, in order to achieve fault tolerance at the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol level. Our extended scheme presents several advantages compared to the state-of-the-art construction by Boneh et al. [BGGJ+18]: Our solution is compact (i.e., requires a constant-size state at each party) and does not require a trusted dealer. Although our solution requires synchronous communication and the introduction of a failure-handling mechanism, we have shown that this is not an issue in the context of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol. We have also implemented this scheme in the Lattigo library and demonstrated the scheme's concrete efficiency through micro-benchmarks. Notably, we observe that the T-out-of-N-threshold access-structures only introduce a negligible computation overhead with respect to the underlying (N-out-of-N-threshold) MHE scheme's operations.

On the practical side, we have proposed Lattigo, a multiparty homomorphic encryption library in Go. In addition to the implementation of our MHE scheme and its T-out-of-N-threshold extension, Lattigo provides implementations for the state-of-the-art single-party HE schemes: BFV, BGV, and CKKS. The library has a modular, multi-layer design, and exposes primitives at all layers. As a result, it can be (and has been) used by HE researchers to implement new schemes and primitives.

Our second practical contribution, HElium, builds on top of Lattigo and provides the first end-to-end open-source implementation of an MHE-based MPC protocol. We have filled several gaps left unaddressed in the theoretical literature on MHE (including our own). In particular, we have shown how to securely implement the MHE-based MPC protocol with weak computing-resources and/or churning participants, by defining and proving secure a non-monolithic execution of this protocol. We have also proposed a concrete solution for the failure-handling mechanism required

by our T-out-of-N-threshold scheme. Finally, by exploiting the properties of the MHE-based MPC protocol, we have proposed a helper-assisted setting, where the parties delegate most of the protocol execution cost to an honest-but-curious third party (e.g., a cloud service). As a result, HElium is also the first implemented MPC system to support sub-linear-cost MPC, without assuming non-collusion between the multiple delegate nodes.

Future of MHE

In the current state of the art, it is no longer pertinent to consider HE as impractical. Therefore, and in the light of the results presented in this dissertation, we argue that it is time to start using MHE-based MPC techniques in MPC research and in practice.

We envision that two factors will further increase the efficiency and adoption of MHE-based techniques: First, the efficiency of HE evaluation is still an active and fruitful area of research, and we expect many improvements to further reduce the cost of critical operations such as bootstrapping. The next big step in this direction is that of hardware accelerators, for which prototypes have already demonstrated acceleration by three orders of magnitude [GVPH+22]. Second, although HE compilers are not yet as advanced as those available for other MPC approaches (see the corresponding open problem in the next session), they will continue to improve, as will the accessibility of MHE-based solution.

However, perhaps the most important transformative aspect of MHE-based techniques is that they will enable shifting the paradigms of MPC toward more centralized techniques. Although decentralization is often perceived as the holy grail when it comes to secure computing systems, decentralized systems are, in practice, incredibly difficult to build and operate. The MHE-based MPC protocol provides many opportunities for centralizing its functional components (e.g., the output of the offline phase, the circuit inputs and the circuit evaluation) while keeping the trust (i.e., the secret-keys) decentralized. As a result, applications relying on MHE could be considerably simpler to integrate and to operate. Moreover, they might give rise to a financially viable business model: providing MPC-as-a-service in a *standalone* model (i.e., without relying on several non-colluding servers). Whereas, such models are not yet viable in single-party HE scenarios (for which the cost of using HE generally out-weights the benefits of outsourcing), they provide a real added value in the multiparty case (for which the computation simply cannot occur otherwise).

Therefore, MHE-based MPC might well be among the first successful commercial application of HE research.

Open Problems and Future Research Directions

We have discussed the current limitations of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, as instantiated in this dissertation, and we have outlined potential solutions, as well as future research directions.

Circuit Privacy and Smudging Noise Our MHE construction relies on a somewhat crude approach to smudging: hiding the noise distribution by flooding it with fresh noise from a distribution of exponentially larger variance. This poses two challenges: First, it is not trivial to correctly and efficiently sample Gaussian noise of large variance. Second, it requires the ciphertext modulus to be large enough to accommodate for this extra noise. For these two aforementioned reasons, it is often impractical to obtain 128-bit of (statistical) security for the

smudging noise, and applications need to settle for smaller values. In fine, setting the smudging-noise power requires one to bound the *computational* advantage that leaking some information on the decryption noise gives to a malicious receiver. This observation led to *gentle smudging* approaches [CHIV+22; LMSS22] that could be adapted to the MHE setting. To achieve this adaptation, a crucial stepping stone is the derivation of accurate models for tracking the HE noise distribution for a given circuit and for a set of HE parameters [CLP20; CCHM+22; MP19].

Compilation to HE circuits and Parameterization In this dissertation, we have focused on the protocol-related aspects of MHE-based MPC. In doing so, we have assumed that the function to be computed was provided as an arithmetic circuit over the MHE scheme's plaintext space. However, expressing higher-level functionalities into MHE-friendly arithmetic circuits and parameterizing the MHE scheme for their evaluation is not an easy task. Consequently, application developers still require the assistance of HE experts in the design of their solutions. In particular, non-arithmetic functions, such as comparisons and branching programs, constitute a fundamental limitation that also applies to LSSS-based MPC (indeed, any branching in the flow of execution would leak information about the program state). However, the compilers of these solutions already propose workarounds, either by mapping them back to an arithmetic representation or by accepting the conditional variable leakage.

Compilers for HE are an active area of research [CDS15; CPS18; CMGT+18; DKSD+20; ACDM+19; VJHH23] that directly applies to MHE circuits. However, a significant limitation of the state of the art is that it does not address the parameterization of the HE scheme (beyond trivial cases) [VJH21]. Unfortunately, parameterization and circuit design are two intertwined tasks in practice, and it is unlikely that they can be addressed separately. Hence, a currently open problem is finding methods for generating a circuit *and* the encryption parameters from a high-level representation. In this regard, we observe that partial solutions might already be of high interest. Indeed, designing HE circuits is a tedious task even for HE experts who currently lack the basic tools to accomplish it efficiently. For example, the lack of a common representation for HE circuits forces HE users and tool-designers to target a specific library or a specific set of tools. This unfortunately fragments the design space into a multitude of point solutions that are difficult to compose. Hence, the most interesting aspect of the HE compiler research, in the short- to medium-term, is not their long-term goal of achieving generic compilation, without expertise, but the development of intermediate representations of HE circuits [VJHH23].

Another missing set of tools for HE circuit designers is noise/precision estimators that would provide debugging and correctness verification capabilities. The current work on models for tracking the noise distribution [CLP20; CCHM+22; MP19], once again, is a crucial step toward building these tools.

Active Adversary Model The constructions presented in this dissertation provide security in the passive adversary model. This might be sufficient in some settings, for example, in the medical sector where data collaborations are mutually beneficial and well-regulated, yet they legally require a certain level of data protection [RTMS+18]. Currently, honest-but-curious is the de-facto threat model to date for cloud services, and passively-secure MPC provides a way of protecting sensitive client-data in these scenarios. But, many applications might require more security guarantees when active adversaries are present among the computation participants. Asharov et al. show that the passively-secure MHE-based MPC protocol can be compiled into a maliciously secure one, by harnessing generic non-interactive zero-knowledge (NIZK) proofs techniques. However, they leave the concrete construction undefined, and the concrete cost of the resulting protocol is, therefore, unknown.

Zero-knowledge-proof systems for lattice-based schemes are another active research topic [BLS19; YAZX+19]. In [CMSP+23], we instantiated the NIZK-based solution of Asharov et al., implemented the first prototype, and obtained concrete performance results. We observe that as the local operations of the MHE scheme are of relatively low depth, proving their correct execution in zero-knowledge is efficient. However, lattice-based schemes rely on low-norm distributions that require costly range proofs. Although the creation and verification of these proofs are practical [CMSP+23], they are not yet efficient enough for general use, for example, when considering memory-limited parties. Reducing the overhead of these proofs is a crucial next step toward maliciously secure MHE.

Proving the correct execution of the homomorphic circuit evaluation is also necessary for a maliciously secure $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, which we leave unaddressed in [CMSP+23]. As the $\Pi_{\mathsf{MHE\text{-}MPC}}$ has a public transcript, a trivial solution is to publish this transcript as a proof. But this solution requires linear communication for the parties, which might be unsatisfactory in some applications. Another approach is to embed the verification mechanisms in the HE plaintext-space [CKPH22], in a MAC-then-encrypt fashion. It is plausible that this approach (proposed for the single-party setting) can be transposed to MHE in such a way that enables sub-linear communication for the parties. Another potential, yet non-cryptographic, solution could be to perform the evaluation in secure hardware enclaves such as Intel's SGX[1]. Although these platforms have been shown to be vulnerable to a number of attacks, these attacks have been mostly successful at breaking the confidentiality of the data (which is protected by HE), and no attack has yet been demonstrated, that could alter the computation itself in an exploitable way. Instead, current *active attacks* have been, to the best of our knowledge, limited to injecting faults that would make the enclave output an incorrect HE ciphertext and make decryption fail.

Final Remarks

It is a frequent observation that a vast amount of the state-of-the-art cryptographic research is not used in practice. In light of my work on bringing a cryptographic technique from theory to practice, this observation sounds rather unsurprising to me. Currently, there are very low incentives for researchers to implement their constructions (e.g., implementation work is often perceived as non-scientific), or to emphasize simple solutions (as simple solutions might be perceived as non-novel). There are some very valid reasons for detaching academic research from the constraints and costs related to concretely applying the results to the real world. But it also sounds a bit naive to expect that practitioners (or standardization bodies) will simply *do the job* when a concrete real-world problem requires a cryptographic solution. Indeed, the cryptographic research literature is far from being a structured catalogue of solutions. Consequently, the cost of implementing a state-of-the-art cryptographic solution, without access to a research prototype, might simply be too high for many practitioners. My view on this tension around practical work is not that researchers should implement their work, but that those who are interested in doing so should be able to find a venue for their results.

Implementation is, in fact, a great avenue for scientific research. First, it produces results that can be used by other researchers (to implement higher-level constructions, or to serve as a comparison baseline) and by practitioners (as a prototype demonstrating the construction's capability and operation). But it also fundamentally requires an in-depth exploration of the implemented primitive, which often highlights limitations that were hidden in the theoretical setting. The contributions of our work on the CKKS bootstrapping [BMTH21] and of our work

[1] `https://intel.com/sgx`

on the non-monolithic execution of the MHE-based MPC protocol (the HElium system) are examples of meaningful theoretical contributions arising from an implementation effort.

The ideas and constructions proposed in this thesis are all simple enough to be implemented, hence they have been. This considerable effort on the practical side of my research was, at the same time, the most frustrating and the most rewarding contribution of my work on this thesis. The frustrating part came from the difficulty in fulfilling the more traditional measurements of academic success: typically, scientific publications (or a number thereof) in highly selective venues that do not always value implementation as a decisive criteria. The rewarding part came from the impact of my work on Lattigo, as it has now become a well-established contribution to the HE-software landscape and is being used both by applied-cryptography researchers for building application prototypes [BSA21; CPTH21; FTRC+21; KSJH21; ICDÖ22; TMBM+22; PPV22; CP23a; ERLT23; KG23; FCES+23] and by HE researchers for implementing proof-of-concept of new constructions and optimization [CHKL+21; HKLL+22; KKLS+22; KLKS+22; LLKK+22; GHHJ22; ACYJ+23; CP23b; KLSS23].

In February 2022, the maintenance of Lattigo was transferred to the EPFL start-up *Tune Insight SA*[2], for which the work presented in this dissertation constitutes the core technical building block of their commercial product. As a result, the library is still being actively developed[3] and, at least in this regard, MHE techniques have made their way into practice.

[2] `https://tuneinsight.com`
[3] `https://github.com/tuneinsight/lattigo`

Bibliography

[AB74] RC Agarwal and C Burrus. "Fast convolution using Fermat number transforms with applications to digital filtering". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 22.2 (1974), pp. 87–97.

[ABLK+18] David W Archer, Dan Bogdanov, Yehuda Lindell, Liina Kamm, Kurt Nielsen, Jakob Illeborg Pagter, Nigel P Smart, and Rebecca N Wright. "From Keys to Databases—Real-World Applications of Secure Multi-Party Computation". In: *The Computer Journal* 61.12 (2018), pp. 1749–1771.

[ACCD+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, 2018.

[ACDE+19] Mark Abspoel, Ronald Cramer, Ivan Damgård, Daniel Escudero, and Chen Yuan. "Efficient Information-Theoretic Secure Multiparty Computation over via Galois Rings". In: *Theory of Cryptography: 17th International Conference, TCC 2019, Nuremberg, Germany, December 1–5, 2019, Proceedings, Part I*. Springer. 2019, pp. 471–501.

[ACDM+19] David W Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. "Ramparts: A programmer-friendly system for building homomorphic encryption applications". In: *Proceedings of the 7th acm workshop on encrypted computing & applied homomorphic cryptography*. 2019, pp. 57–68.

[ACYJ+23] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, and Ajay Joshi. "FAB: An FPGA-based accelerator for bootstrappable fully homomorphic encryption". In: *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE. 2023, pp. 882–895.

[AJLT+12] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. "Multiparty computation with low communication, computation and interaction via threshold FHE". In: *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings 31*. Springer. 2012, pp. 483–501.

[AMP18] Andreea B Alexandru, Manfred Morari, and George J Pappas. "Cloud-based MPC with encrypted data". In: *2018 IEEE Conference on Decision and Control (CDC)*. IEEE. 2018, pp. 5014–5019.

[ANWW13] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: simpler, smaller, fast as MD5". In: *International Conference on Applied Cryptography and Network Security*. Springer. 2013, pp. 119–135.

[ATP21] Andreea B Alexandru, Anastasios Tsiamis, and George J Pappas. "Encrypted distributed Lasso for sparse data predictive control". In: *IEEE Conference on Decision and Control (CDC)*. 2021.

[BCDG+09] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, et al. "Secure multiparty computation goes live". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2009, pp. 325–343.

[BD10] Rikke Bendlin and Ivan Damgård. "Threshold decryption and zero-knowledge proofs for lattice-based cryptosystems". In: *Theory of Cryptography: 7th Theory of Cryptography Conference, TCC 2010, Zurich, Switzerland, February 9-11, 2010. Proceedings 7*. Springer. 2010, pp. 201–218.

[Bea92] Donald Beaver. "Efficient multiparty protocols using circuit randomization". In: *Advances in Cryptology—CRYPTO'91: Proceedings 11*. Springer. 1992, pp. 420–432.

[BEHZ16] Jean-Claude Bajard, Julien Eynard, M Anwar Hasan, and Vincent Zucca. "A full RNS variant of FV like somewhat homomorphic encryption schemes". In: *International Conference on Selected Areas in Cryptography*. Springer. 2016, pp. 423–442.

[BEPS+20] Carsten Baum, Daniel Escudero, Alberto Pedrouzo-Ulloa, Peter Scholl, and Juan Ramón Troncoso-Pastoriza. "Efficient Protocols for Oblivious Linear Function Evaluation from Ring-LWE". In: *Security and Cryptography for Networks: 12th International Conference, SCN 2020, Amalfi, Italy, September 14–16, 2020, Proceedings*. 2020, pp. 130–149.

[BFLS91] László Babai, Lance Fortnow, Leonid A Levin, and Mario Szegedy. "Checking computations in polylogarithmic time". In: *Proceedings of the twenty-third annual ACM symposium on Theory of computing*. 1991, pp. 21–32.

[BGGJ+18] Dan Boneh, Rosario Gennaro, Steven Goldfeder, Aayush Jain, Sam Kim, Peter MR Rasmussen, and Amit Sahai. "Threshold cryptosystems from threshold fully homomorphic encryption". In: *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part I 38*. Springer. 2018, pp. 565–596.

[BGV14] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. "(Leveled) fully homomorphic encryption without bootstrapping". In: *ACM Transactions on Computation Theory (TOCT)* 6.3 (2014), pp. 1–36.

[BHKL18] Assi Barak, Martin Hirt, Lior Koskas, and Yehuda Lindell. "An end-to-end system for large scale P2P MPC-as-a-service and low-bandwidth MPC for weak participants". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 695–712.

[BJSV15] Dan Bogdanov, Marko Jõemets, Sander Siim, and Meril Vaht. "How the estonian tax and customs board evaluated a tax fraud detection system based on secure multi-party computation". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2015, pp. 227–234.

[BLS19] Jonathan Bootle, Vadim Lyubashevsky, and Gregor Seiler. "Algebraic techniques for short(er) exact lattice-based zero-knowledge proofs". In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I*. Springer. 2019, pp. 176–202.

[BLW08] Dan Bogdanov, Sven Laur, and Jan Willemson. "Sharemind: A framework for fast privacy-preserving computations". In: *European Symposium on Research in Computer Security*. Springer. 2008, pp. 192–206.

[BMTH21] Jean-Philippe Bossuat, Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. "Efficient bootstrapping for approximate homomorphic encryption with non-sparse keys". In: *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. [Paper]. Springer. 2021, pp. 587–617.

[BSA21] Liudmila Babenko, Alexander Shumilin, and Dmitry Alekseev. "Development of the algorithm to ensure the protection of confidential data in cloud medical information system". In: *2021 14th International Conference on Security of Information and Networks (SIN)*. Vol. 1. IEEE. 2021, pp. 1–4.

[BTH22] Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. "Bootstrapping for approximate homomorphic encryption with negligible failure-probability by using sparse-secret encapsulation". In: *Applied Cryptography and Network Security: 20th International Conference, ACNS 2022, Rome, Italy, June 20–23, 2022, Proceedings*. Springer. 2022, pp. 521–541.

[BTW12] Dan Bogdanov, Riivo Talviste, and Jan Willemson. "Deploying secure multi-party computation for financial data analysis". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2012, pp. 57–64.

[BV11] Zvika Brakerski and Vinod Vaikuntanathan. "Fully homomorphic encryption from ring-LWE and security for key dependent messages". In: *Advances in Cryptology–CRYPTO 2011: 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings 31*. Springer. 2011, pp. 505–524.

[CB17] Henry Corrigan-Gibbs and Dan Boneh. "Prio: Private, robust, and scalable computation of aggregate statistics". In: *14th Symposium on Networked Systems Design and Implementation (NSDI 17)*. 2017, pp. 259–282.

[CCHM+22] Anamaria Costache, Benjamin R Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. "On the precision loss in approximate homomorphic encryption". In: *Cryptology ePrint Archive* (2022).

[CCS19a] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Improved bootstrapping for approximate homomorphic encryption". In: *Advances in Cryptology–EUROCRYPT 2019: 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19–23, 2019, Proceedings, Part II*. Springer. 2019, pp. 34–54.

[CCS19b] Hao Chen, Ilaria Chillotti, and Yongsoo Song. "Multi-key homomorphic encryption from TFHE". In: *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part II 25*. Springer. 2019, pp. 446–472.

[CDES+18] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. "SPD$\mathbb{Z}_{2^k}$: Efficient MPC mod 2^k for Dishonest Majority". In: *Advances in Cryptology–CRYPTO 2018: 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19–23, 2018, Proceedings, Part II*. Springer. 2018, pp. 769–798.

[CDKS19] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. "Efficient multi-key homomorphic encryption with packed ciphertexts with application to oblivious neural network inference". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. ACM New York, NY, USA. 2019, pp. 395–412.

[CDN01] Ronald Cramer, Ivan Damgård, and Jesper B Nielsen. "Multiparty computation from threshold homomorphic encryption". In: *Advances in Cryptology–EUROCRYPT 2001: International Conference on the Theory and Application of Cryptographic Techniques Innsbruck, Austria, May 6–10, 2001 Proceedings 20*. Springer. 2001, pp. 280–300.

[CDN15] Ronald Cramer, Ivan Bjerre Damgård, and Jesper Buus Nielsen. "Secure Multiparty Computation and Secret Sharing". In: *Secure Multiparty Computation and Secret Sharing*. Cambridge University Press, 2015, pp. 236–298. DOI: 10.1017/CBO9781107337756.012.

[CDS15] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. "Armadillo: a compilation chain for privacy preserving applications". In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. 2015, pp. 13–19.

[CHHS19] Jung Hee Cheon, Minki Hhan, Seungwan Hong, and Yongha Son. "A hybrid of dual and meet-in-the-middle attack on sparse and ternary secret LWE". In: *IEEE Access* 7 (2019), pp. 89497–89506.

[CHIV+22] Leo de Castro, Carmit Hazay, Yuval Ishai, Vinod Vaikuntanathan, and Muthu Venkitasubramaniam. "Asymptotically Quasi-Optimal Cryptography". In: *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*. Springer. 2022, pp. 303–334.

[CHKK+18] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "Bootstrapping for approximate homomorphic encryption". In: *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part I 37*. Springer. 2018, pp. 360–384.

[CHKK+19] Jung Hee Cheon, Kyoohyung Han, Andrey Kim, Miran Kim, and Yongsoo Song. "A full RNS variant of approximate homomorphic encryption". In: *Selected Areas in Cryptography–SAC 2018: 25th International Conference, Calgary, AB, Canada, August 15–17, 2018, Revised Selected Papers 25*. Springer. 2019, pp. 347–368.

[CHKL+21] Jihoon Cho, Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Joohee Lee, Jooyoung Lee, Dukjae Moon, and Hyojin Yoon. "Transciphering framework for approximate homomorphic encryption". In: *Advances in Cryptology–ASIACRYPT 2021: 27th International Conference on the Theory and Application of Cryptology and Information Security, Singapore, December 6–10, 2021, Proceedings, Part III*. Springer. 2021, pp. 640–669.

[CHP13] Ashish Choudhury, Martin Hirt, and Arpita Patra. "Asynchronous multiparty computation with linear communication complexity". In: *Distributed Computing: 27th International Symposium, DISC 2013, Jerusalem, Israel, October 14-18, 2013. Proceedings 27*. Springer. 2013, pp. 388–402.

[CKKS17] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic encryption for arithmetic of approximate numbers". In: *Advances in Cryptology–ASIACRYPT 2017: 23rd International Conference on the Theory and Applications of Cryptology and Information Security, Hong Kong, China, December 3-7, 2017, Proceedings, Part I 23*. Springer. 2017, pp. 409–437.

[CKLS02] Christian Cachin, Klaus Kursawe, Anna Lysyanskaya, and Reto Strobl. "Asynchronous verifiable secret sharing and proactive cryptosystems". In: *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 2002, pp. 88–97.

[CKPH22] Sylvain Chatel, Christian Knabenhans, Apostolos Pyrgelis, and Jean-Pierre Hubaux. "Verifiable Encodings for Secure Homomorphic Analytics". In: *arXiv preprint arXiv:2207.14071* (2022).

[CLP20] Anamaria Costache, Kim Laine, and Rachel Player. "Evaluating the Effectiveness of Heuristic Worst-Case Noise Analysis in FHE". In: *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part II*. 2020, pp. 546–565.

[CMGT+18] Eduardo Chielle, Oleg Mazonka, Homer Gamil, Nektarios Georgios Tsoutsos, and Michail Maniatakos. "E3: A framework for compiling C++ programs with encrypted operands". In: *Cryptology ePrint Archive* (2018).

[CMSP+23] Sylvain Chatel, Christian Mouchet, Ali Utkan Sahin, Apostolos Pyrgelis, Carmela Troncoso, and Jean-Pierre Hubaux. "PELTA–Shielding Multiparty-FHE against Malicious Adversaries". In: *Cryptology ePrint Archive* (2023). To appear in: Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS'23).

[CP15] Ashish Choudhury and Arpita Patra. "Optimally resilient asynchronous MPC with linear communication complexity". In: *Proceedings of the 16th International Conference on Distributed Computing and Networking (ICDCN)*. 2015, pp. 1–10.

[CP23a] José Cabrero-Holgueras and Sergio Pastrana. "HEFactory: A symbolic execution compiler for privacy-preserving Deep Learning with Homomorphic Encryption". In: *SoftwareX* 22 (2023), p. 101396.

[CP23b] José Cabrero-Holgueras and Sergio Pastrana. "Towards automated homomorphic encryption parameter selection with fuzzy logic and linear programming". In: *Expert Systems with Applications* (2023), p. 120460.

[CPS18] Eric Crockett, Chris Peikert, and Chad Sharp. "Alchemy: A language and compiler for homomorphic encryption made easy". In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 2018, pp. 1020–1037.

[CPTH21] Sylvain Chatel, Apostolos Pyrgelis, Juan Ramón Troncoso-Pastoriza, and Jean-Pierre Hubaux. "Privacy and Integrity Preserving Computations with CRISP." In: *USENIX Security Symposium*. 2021, pp. 2111–2128.

[CSSM+22] Siddhartha Chowdhury, Sayani Sinha, Animesh Singh, Shubham Mishra, Chandan Chaudhary, Sikhar Patranabis, Pratyay Mukherjee, Ayantika Chatterjee, and Debdeep Mukhopadhyay. *Efficient Threshold FHE with Application to Real-Time Systems*. Cryptology ePrint Archive, Paper 2022/1625. 2022.

[CWB18] Hyunghoon Cho, David J Wu, and Bonnie Berger. "Secure genome-wide association analysis using multiparty computation". In: *Nature biotechnology* 36.6 (2018), p. 547.

[Des93] Yvo Desmedt. "Threshold cryptosystems". In: *Advances in Cryptology—AUSCRYPT'92: Workshop on the Theory and Application of Cryptographic Techniques Gold Coast, Queensland, Australia, December 13–16, 1992 Proceedings 3*. Springer. 1993, pp. 1–14.

[DGKN09] Ivan Damgård, Martin Geisler, Mikkel Krøigaard, and Jesper Buus Nielsen. "Asynchronous multiparty computation: Theory and implementation". In: *Public Key Cryptography–PKC 2009: 12th International Conference on Practice and Theory in Public Key Cryptography, Irvine, CA, USA, March 18-20, 2009. Proceedings 12*. Springer. 2009, pp. 160–179.

[DKLP+13] Ivan Damgård, Marcel Keller, Enrique Larraia, Valerio Pastro, Peter Scholl, and Nigel P Smart. "Practical covertly secure MPC for dishonest majority–or: breaking the SPDZ limits". In: *Computer Security–ESORICS 2013: 18th European Symposium on Research in Computer Security, Egham, UK, September 9-13, 2013. Proceedings 18*. Springer. 2013, pp. 1–18.

[DKSD+20] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madan Musuvathi. "EVA: An encrypted vector arithmetic language and compiler for efficient homomorphic computation". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 546–561.

[DPSZ12] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. "Multiparty computation from somewhat homomorphic encryption". In: *Advances in Cryptology–CRYPTO 2012: 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*. Springer. 2012, pp. 643–662.

[ERLT23] Kasra EdalatNejad, Mathilde Raynal, Wouter Lueks, and Carmela Troncoso. "Private Collection Matching Protocols". In: *Proceedings on Privacy Enhancing Technologies* 3 (2023), pp. 446–468.

[FCES+23] D. Froelicher, H. Cho, M. Edupalli, J. Sa Sousa, J. Bossuat, A. Pyrgelis, J. R. Troncoso-Pastoriza, B. Berger, and J. Hubaux. "Scalable and Privacy-Preserving Federated Principal Component Analysis". In: *2023 2023 IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2023, pp. 888–905.

[FH96] Matthew Franklin and Stuart Haber. "Joint encryption and message-efficient secure computation". In: *Journal of Cryptology* 9.4 (1996), pp. 217–232.

[FTPS+21] David Froelicher, Juan R Troncoso-Pastoriza, Apostolos Pyrgelis, Sinem Sav, Joao Sa Sousa, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. "Scalable Privacy-Preserving Distributed Learning". In: vol. 2. 2021, pp. 323–347.

[FTRC+21] David Froelicher, Juan R Troncoso-Pastoriza, Jean Louis Raisaro, Michel A Cuendet, Joao Sa Sousa, Hyunghoon Cho, Bonnie Berger, Jacques Fellay, and Jean-Pierre Hubaux. "Truly privacy-preserving federated analytics for precision medicine with multiparty homomorphic encryption". In: *Nature communications* (2021).

[FTSH20] David Froelicher, Juan Ramón Troncoso-Pastoriza, Joao Sa Sousa, and Jean-Pierre Hubaux. "Drynx: Decentralized, secure, verifiable system for statistical queries and machine learning on distributed datasets". In: *IEEE Transactions on Information Forensics and Security* 15 (2020), pp. 3035–3050.

[FV12] Junfeng Fan and Frederik Vercauteren. "Somewhat practical fully homomorphic encryption". In: *Cryptology ePrint Archive* (2012).

[Gen09] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. ACM New York, NY, USA. 2009, pp. 169–178.

[GGP10] Rosario Gennaro, Craig Gentry, and Bryan Parno. "Non-interactive verifiable computing: Outsourcing computation to untrusted workers". In: *Advances in Cryptology–CRYPTO 2010: 30th Annual Cryptology Conference, Santa Barbara, CA, USA, August 15-19, 2010. Proceedings 30*. Springer. 2010, pp. 465–482.

[GHHJ22] Aarushi Goel, Mathias Hall-Andersen, Aditya Hegde, and Abhishek Jain. "Secure Multiparty Computation with Free Branching". In: *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*. Springer. 2022, pp. 397–426.

[GHS12] Craig Gentry, Shai Halevi, and Nigel P Smart. "Fully Homomorphic Encryption with Polylog Overhead". In: *Advances in Cryptology–EUROCRYPT 2012: 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012, Proceedings*. Springer Berlin Heidelberg, 2012, pp. 465–482.

[GJKR99] Rosario Gennaro, Stanisław Jarecki, Hugo Krawczyk, and Tal Rabin. "Secure distributed key generation for discrete-log based cryptosystems". In: *Advances in Cryptology—EUROCRYPT'99: International Conference on the Theory and Application of Cryptographic Techniques Prague, Czech Republic, May 2–6, 1999 Proceedings 18*. Springer. 1999, pp. 295–310.

[GKR15] Shafi Goldwasser, Yael Tauman Kalai, and Guy N Rothblum. "Delegating computation: interactive proofs for muggles". In: *Journal of the ACM (JACM)* 62.4 (2015), pp. 1–64.

[GMT22] Charles Gouert, Dimitris Mouris, and Nektarios Georgios Tsoutsos. "New insights into fully homomorphic encryption libraries via standardized benchmarks". In: *Cryptology ePrint Archive* (2022). To appear in the Proceedings on Privacy Enhancing Technologies (PoPETs) 2023.

[Gol09] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, 2009, pp. 636–638.

[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: 33rd Annual Cryptology Conference, Santa Barbara, CA, USA, August 18-22, 2013. Proceedings, Part I*. Springer. 2013, pp. 75–92.

[GVPH+22] Robin Geelen, Michiel Van Beirendonck, Hilder VL Pereira, Brian Huffman, Tynan McAuley, Ben Selfridge, Daniel Wagner, Georgios Dimou, Ingrid Verbauwhede, Frederik Vercauteren, et al. "BASALISC: Programmable Asynchronous Hardware Accelerator for BGV Fully Homomorphic Encryption". In: *Cryptology ePrint Archive* (2022).

[HHNZ19] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. "SoK: General purpose compilers for secure multi-party computation". In: *2019 IEEE symposium on security and privacy (SP)*. IEEE. 2019, pp. 1220–1237.

[HJKY95] Amir Herzberg, Stanisław Jarecki, Hugo Krawczyk, and Moti Yung. "Proactive secret sharing or: How to cope with perpetual leakage". In: *Advances in Cryptology—CRYPT0'95: 15th Annual International Cryptology Conference Santa Barbara, California, USA, August 27–31, 1995 Proceedings 15*. Springer. 1995, pp. 339–352.

[HK20] Kyoohyung Han and Dohyeong Ki. "Better bootstrapping for approximate homomorphic encryption". In: *Topics in Cryptology–CT-RSA 2020: The Cryptographers' Track at the RSA Conference 2020, San Francisco, CA, USA, February 24–28, 2020, Proceedings*. Springer. 2020, pp. 364–390.

[HKLL+22] Jincheol Ha, Seongkwang Kim, Byeonghak Lee, Jooyoung Lee, and Mincheol Son. "Rubato: Noisy Ciphers for Approximate Homomorphic Encryption". In: *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*. Springer. 2022, pp. 581–610.

[HPS19] Shai Halevi, Yuriy Polyakov, and Victor Shoup. "An improved RNS variant of the BFV homomorphic encryption scheme". In: *Topics in Cryptology–CT-RSA 2019: The Cryptographers' Track at the RSA Conference 2019, San Francisco, CA, USA, March 4–8, 2019, Proceedings*. Springer. 2019, pp. 83–105.

[ICDÖ22] Alberto Ibarrondo, Hervé Chabanne, Vincent Despiegel, and Melek Önen. "Colmade: Collaborative Masking in Auditable Decryption for BFV-based Homomorphic Encryption". In: *Proceedings of the 2022 ACM Workshop on Information Hiding and Multimedia Security*. 2022, pp. 129–139.

[ISPB+23] Francesco Intoci, Sinem Sav, Apostolos Pyrgelis, Jean-Philippe Bossuat, Juan Ramon Troncoso-Pastoriza, and Jean-Pierre Hubaux. "slytHErin: An Agile Framework for Encrypted Deep Neural Network Inference". In: *arXiv preprint arXiv:2305.00690* (2023).

[JWBB+17] Karthik A Jagadeesh, David J Wu, Johannes A Birgmeier, Dan Boneh, and Gill Bejerano. "Deriving genomic diagnoses without revealing patient genomes". In: *Science* 357.6352 (2017), pp. 692–695.

[Kel20] Marcel Keller. "MP-SPDZ: A versatile framework for multi-party computation". In: *Proceedings of the 2020 ACM SIGSAC conference on computer and communications security*. 2020, pp. 1575–1590.

[KFB14] Joshua Kroll, Edward Felten, and Dan Boneh. "Secure protocols for accountable warrant execution". In: (2014).

[KG09] Aniket Kate and Ian Goldberg. "Distributed key generation for the internet". In: *2009 29th IEEE International Conference on Distributed Computing Systems*. IEEE. 2009, pp. 119–128.

[KG21] Chelsea Komlo and Ian Goldberg. "FROST: flexible round-optimized Schnorr threshold signatures". In: *Selected Areas in Cryptography: 27th International Conference, Halifax, NS, Canada (Virtual Event), October 21-23, 2020*. Springer. 2021, pp. 34–65.

[KG23] Dongwoo Kim and Cyril Guyot. "Optimized Privacy-Preserving CNN Inference With Fully Homomorphic Encryption". In: *IEEE Transactions on Information Forensics and Security* 18 (2023), pp. 2175–2187.

[KHBC+21] Miran Kim, Arif Ozgun Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, et al. "Ultrafast homomorphic encryption models enable secure outsourcing of genotype imputation". In: *Cell systems* 12.11 (2021), pp. 1108–1120.

[KKLS+22] Taechan Kim, Hyesun Kwak, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. "Asymptotically faster multi-key homomorphic encryption from homomorphic gadget decomposition". In: *Cryptology ePrint Archive* (2022).

[KLKS+22] Jongmin Kim, Gwangho Lee, Sangpyo Kim, Gina Sohn, Minsoo Rhu, John Kim, and Jung Ho Ahn. "Ark: Fully homomorphic encryption accelerator with runtime data generation and inter-operation key reuse". In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 1237–1254.

[KLSS23] Miran Kim, Dongwon Lee, Jinyeong Seo, and Yongsoo Song. "Accelerating HE Operations from Key Decomposition Technique". In: *Cryptology ePrint Archive* (2023).

[KLSW21] Hyesun Kwak, Dongwon Lee, Yongsoo Song, and Sameer Wagh. "A unified framework of homomorphic encryption for multiple parties with non-interactive setup". In: *Cryptology ePrint Archive* (2021).

[KMPR+17] Vladimir Kolesnikov, Naor Matania, Benny Pinkas, Mike Rosulek, and Ni Trieu. "Practical Multi-party Private Set Intersection from Symmetric-Key Techniques". In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM New York, NY, USA. 2017, pp. 1257–1272.

[KOS16] Marcel Keller, Emmanuela Orsini, and Peter Scholl. "MASCOT: faster malicious arithmetic secure computation with oblivious transfer". In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM New York, NY, USA. 2016, pp. 830–842.

[KPR18] Marcel Keller, Valerio Pastro, and Dragos Rotaru. "Overdrive: making SPDZ great again". In: *Advances in Cryptology–EUROCRYPT 2018: 37th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tel Aviv, Israel, April 29-May 3, 2018 Proceedings, Part III*. Springer. 2018, pp. 158–189.

[KS19] Duhyeong Kim and Yongsoo Song. "Approximate homomorphic encryption over the conjugate-invariant ring". In: *Information Security and Cryptology–ICISC 2018: 21st International Conference, Seoul, South Korea, November 28–30, 2018, Revised Selected Papers 21*. Springer. 2019, pp. 85–102.

[KSJH21] Miran Kim, Yongsoo Song, Xiaoqian Jiang, and Arif Harmanci. "SHiMMer: Privacy-Aware Alignment of Genomic Sequences with Secure and Efficient Hidden Markov Model Evaluation". In: (2021).

[Lin17] Yehuda Lindell. "How to simulate it–a tutorial on the simulation proof technique". In: *Tutorials on the Foundations of Cryptography*. Springer, 2017, pp. 277–346.

[LLKK+22] Yongwoo Lee, Joon-Woo Lee, Young-Sik Kim, Yongjune Kim, Jong-Seon No, and HyungChul Kang. "High-precision bootstrapping for approximate homomorphic encryption by error variance minimization". In: *Advances in Cryptology–EUROCRYPT 2022: 41st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Trondheim, Norway, May 30–June 3, 2022, Proceedings, Part I*. Springer. 2022, pp. 551–580.

[LM21] Baiyu Li and Daniele Micciancio. "On the security of homomorphic encryption on approximate numbers". In: *Advances in Cryptology–EUROCRYPT 2021: 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17–21, 2021, Proceedings, Part I 40*. Springer. 2021, pp. 648–677.

[LMSS22] Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. "Securing approximate homomorphic encryption using differential privacy". In: *Advances in Cryptology–CRYPTO 2022: 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15–18, 2022, Proceedings, Part I*. Springer. 2022, pp. 560–589.

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. "On Ideal Lattices and Learning with Errors over Rings". In: *Advances in Cryptology–EUROCRYPT 2010: 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, French Riviera, May 30-June 3, 2010, Proceedings*. Vol. 6110. Springer. 2010, p. 1.

[LTV11] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. "Cloud-assisted multiparty computation from fully homomorphic encryption". In: *Cryptology ePrint Archive* (2011).

[LTV12] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption". In: *Proceedings of the forty-fourth annual ACM symposium on Theory of computing*. ACM New York, NY, USA. 2012, pp. 1219–1234.

[LYKG+19] Donghang Lu, Thomas Yurek, Samarth Kulshreshtha, Rahul Govind, Aniket Kate, and Andrew Miller. "HoneyBadgerMPC and AsynchroMix: Practical asynchronous MPC and its application to anonymous communication". In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 887–903.

[MBH23] Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. "An Efficient Threshold Access-Structure for RLWE-Based Multiparty Homomorphic Encryption". In: *Journal of Cryptology* 36 (2023). [Paper].

[MBTH20] Christian Mouchet, Jean-Philippe Bossuat, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. "Lattigo: A multiparty homomorphic encryption library in Go". In: *WAHC 2020–8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. [Paper], [Library]. 2020.

[Mic00] Silvio Micali. "Computationally sound proofs". In: *SIAM Journal on Computing* 30.4 (2000), pp. 1253–1298.

[Mon85] Peter L Montgomery. "Modular multiplication without trial division". In: *Mathematics of computation* 44.170 (1985), pp. 519–521.

[MP19] Sean Murphy and Rachel Player. "A central limit framework for ring-lwe decryption". In: *Cryptology ePrint Archive* (2019).

[MTBH21] Christian Mouchet, Juan Troncoso-Pastoriza, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. "Multiparty Homomorphic Encryption from Ring-Learning-with-Errors". In: *Proceedings on Privacy Enhancing Technologies* 4 (2021). [Paper], pp. 291–311.

[MZ17] Payman Mohassel and Yupeng Zhang. "SecureML: A system for scalable privacy-preserving machine learning". In: *2017 38th IEEE Symposium on Security and Privacy (SP)*. IEEE. 2017, pp. 19–38.

[NWIJ+13] Valeria Nikolaenko, Udi Weinsberg, Stratis Ioannidis, Marc Joye, Dan Boneh, and Nina Taft. "Privacy-preserving ridge regression on hundreds of millions of records". In: *2013 IEEE symposium on security and privacy*. IEEE. 2013, pp. 334–348.

[Par21] Jeongeun Park. "Homomorphic encryption for multiple users with less communications". In: *IEEE Access* 9 (2021), pp. 135915–135926.

[Ped91] Torben Pryds Pedersen. "A threshold cryptosystem without a trusted party". In: *Advances in Cryptology—EUROCRYPT'91: Workshop on the Theory and Application of Cryptographic Techniques Brighton, UK, April 8–11, 1991 Proceedings 10*. Springer. 1991, pp. 522–526.

[PPV22] Alberto Pedrouzo-Ulloa, Fernando Pérez-González, and David Vázquez-Padín. "Secure Collaborative Camera Attribution". In: *Proceedings of the 2022 European Interdisciplinary Cybersecurity Conference*. 2022, pp. 97–98.

[Reg05] Oded Regev. "On lattices, learning with errors, random linear codes, and cryptography". In: *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing*. 2005, pp. 84–93.

[Rot17] Dragoș Rotaru. *awesome-mpc*. `https://github.com/rdragos/awesome-mpc`. 2017.

[RSTV+22] Dragos Rotaru, Nigel P Smart, Titouan Tanguy, Frederik Vercauteren, and Tim Wood. "Actively secure setup for SPDZ". In: *Journal of Cryptology* 35.1 (2022), p. 5.

[RTMS+18] Jean Louis Raisaro, Juan Troncoso-Pastoriza, Mickaël Misbach, João Sá Sousa, Sylvain Pradervand, Edoardo Missiaglia, Olivier Michielin, Bryan Ford, and Jean-Pierre Hubaux. "MedCo: Enabling Secure and Privacy-Preserving Exploration of Distributed Clinical and Genomic Data". In: *IEEE/ACM transactions on computational biology and bioinformatics* 16.4 (2018), pp. 1328–1341.

[SBTC+22] Sinem Sav, Jean-Philippe Bossuat, Juan R Troncoso-Pastoriza, Manfred Claassen, and Jean-Pierre Hubaux. "Privacy-preserving federated neural network learning for disease-associated cell classification". In: *Patterns* 3.5 (2022), p. 100487.

[SC19] Yongha Son and Jung Hee Cheon. "Revisiting the Hybrid Attack on Sparse Secret LWE and Application to HE Parameters". In: *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography* (2019).

[SDPB+22] Sinem Sav, Abdulrahman Diaa, Apostolos Pyrgelis, Jean-Philippe Bossuat, and Jean-Pierre Hubaux. "Privacy-Preserving Federated Recurrent Neural Networks". In: *arXiv preprint arXiv:2207.13947* (2022).

[Sha79] Adi Shamir. "How to share a secret". In: *Communications of the ACM* 22.11 (1979), pp. 612–613.

[SJKG+17] Ewa Syta, Philipp Jovanovic, Eleftherios Kokoris Kogias, Nicolas Gailly, Linus Gasser, Ismail Khoffi, Michael J Fischer, and Bryan Ford. "Scalable bias-resistant distributed randomness". In: *2017 IEEE Symposium on Security and Privacy (SP)*. Ieee. 2017, pp. 444–460.

[SPTF+21] Sinem Sav, Apostolos Pyrgelis, Juan R Troncoso-Pastoriza, David Froelicher, Jean-Philippe Bossuat, Joao Sa Sousa, and Jean-Pierre Hubaux. "POSEIDON: Privacy-preserving federated neural network learning". In: *28th Annual Network and Distributed System Security Symposium* (2021).

[TMBM+22] Juan R Trocoso-Pastoriza, Alain Mermoud, Romain Bouyé, Francesco Marino, Jean-Philippe Bossuat, Vincent Lenders, and Jean-Pierre Hubaux. "Orchestrating Collaborative Cybersecurity: A Secure Framework for Distributed Privacy-Preserving Threat Intelligence Sharing". In: *arXiv preprint arXiv:2209.02676* (2022).

[UR22] Antoine Urban and Matthieu Rambaud. *Share and Shrink: Ad-Hoc Threshold FHE with Short Ciphertexts and its Application to Almost-Asynchronous MPC*. Cryptology ePrint Archive, Paper 2022/378. `https://eprint.iacr.org/2022/378`. 2022. URL: `https://eprint.iacr.org/2022/378`.

[VJH21] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully homomorphic encryption compilers". In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2021, pp. 1092–1108.

[VJHH23] Alexander Viand, Patrick Jattke, Miro Haller, and Anwar Hithnawi. "HECO: Fully Homomorphic Encryption Compiler". In: *USENIX Security Symposium 2023*. 2023, (Pre–publication).

[XHXZ+22] Guowen Xu, Xingshuo Han, Shengmin Xu, Tianwei Zhang, Hongwei Li, Xinyi Huang, and Robert H Deng. "Hercules: Boosting the Performance of Privacy-preserving Federated Learning". In: *IEEE Transactions on Dependable and Secure Computing* (2022).

[XLGZ+23] Guowen Xu, Guanlin Li, Shangwei Guo, Tianwei Zhang, and Hongwei Li. "Secure Decentralized Image Classification with Multiparty Homomorphic Encryption". In: *IEEE Transactions on Circuits and Systems for Video Technology* (2023).

[YAZX+19] Rupeng Yang, Man Ho Au, Zhenfei Zhang, Qiuliang Xu, Zuoxia Yu, and William Whyte. "Efficient lattice-based zero-knowledge arguments with standard soundness: construction and applications". In: *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part I 39*. Springer. 2019, pp. 147–175.

[YZWL+22] Meng Yang, Chuwen Zhang, Xiaoji Wang, Xingmin Liu, Shisen Li, Jianye Huang, Zhimin Feng, Xiaohui Sun, Fang Chen, Shuang Yang, et al. "TrustGWAS: A full-process workflow for encrypted GWAS using multi-key homomorphic encryption and pseudorandom number perturbation". In: *Cell Systems* (2022).

[ZPGS19] Wenting Zheng, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. "Helen: Maliciously secure coopetitive learning for linear models". In: *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2019, pp. 724–738.

Appendix A
Derivations and Proofs

Appendix Content

A.1 Comparison between $\Pi_{\mathsf{RelinKeyGen}}$ and previous work

We show here how to adapt the classic method of Asharov et al. [AJLT+12] to our scheme, resulting in the Protocol 9.

Protocol 9. $\Pi_{\mathsf{RelinKeyGen}^{\mathsf{cpk}}}$ ▷ *The* cpk*-based relinearization-key generation protocol*

Public Input: $\mathsf{cpk}=(p_0, p_1)$, $\mathbf{w}$
Private Input of P_i: $\mathsf{sk} = s_i$
Output: $\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1)$

Round 1:
Each party P_i:
1. samples $\mathbf{e}_{0,i} \leftarrow \chi^l$ and $\mathbf{a} \in R_q^l$ from the CRS,
2. discloses $\mathbf{h}_i = -s_i\mathbf{a} + s_i\mathbf{w} + \mathbf{e}_{0,i}$.

Round 2:
Each party P_i:
1. sets $\mathbf{h} = \sum_{P_j \in \mathcal{P}} \mathbf{h}_j$,
2. samples $\mathbf{u}_i \leftarrow R_3^l$, $\mathbf{e}_{1,i}, \mathbf{e}_{2,i} \leftarrow \chi^l$ and,
3. discloses $\mathbf{h}'_{0,i} = s_i\mathbf{h} + \mathbf{u}_i p_0 + \mathbf{e}_{1,i}$ and $\mathbf{h}'_{1,i} = s_i\mathbf{a} + \mathbf{u}_i p_1 + \mathbf{e}_{2,i}$.

Output:
Set $\mathbf{h}'_0 = \sum_{P_j \in \mathcal{P}} \mathbf{h}'_{0,j}$ and $\mathbf{h}'_1 = \sum_{P_j \in \mathcal{P}} \mathbf{h}'_{1,j}$, outputs $\mathsf{rlk} = (\mathbf{h}'_0, \mathbf{h}'_1)$.

The relinearization key resulting from the $\Pi_{\mathsf{RelinKeyGen}^{\mathsf{cpk}}}$ protocol is of the form

$$\begin{aligned}\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1) &= (-(s\mathbf{b} + (s\mathbf{u} + \mathbf{v})e_{\mathsf{cpk}} - s\mathbf{e}_0 - \mathbf{e}_2) + s^2\mathbf{w}\ ,\ \mathbf{b} + s\mathbf{e}_1 + \mathbf{e}_3),\\ &= (-s\mathbf{a}+\mathbf{u}p_0+s^2\mathbf{w}+\mathbf{u}e_{\mathsf{cpk}}+s\mathbf{e}_0+\mathbf{e}_1\ ,\ s\mathbf{a}+\mathbf{u}_1p_1+\mathbf{e}_2).\end{aligned}$$

Hence, rlk holds a significantly increased noise with respect to the key produced by the (ideal) $\mathsf{MHE.RelinKeyGen}$, not only in $\mathbf{r}_0$, but also in $\mathbf{r}_1$, which is not *noisy* when generated in a centralized way. Our $\Pi_{\mathsf{RelinKeyGen}}$ solution (see Section 2.2.3) significantly improves on the simple method, by producing a noise-free $\mathbf{r}_1$ term and a less noisy $\mathbf{r}_0$ term.

A.2 Derivations of the Noise Analysis Equations

This appendix details the derivations of the noise growth equations presented in Section 2.3.2. The infinity norm of a polynomial p (i.e., its largest coefficient in absolute value) is denoted $\|p\|$ ($\|p\| \leq q/2$ for $p \in R_q$). We also recall that, since the polynomial modulus in R_q is a degree-n power of 2 cyclotomic, we have $\|ab\| \leq n\|a\|\|b\|$. We consider an instantiation of our distributed BFV scheme with N parties.

Derivation of Eq. (2.6)

From the ideal decryption of a fresh encryption of m under the collective public key $\mathsf{cpk} = (p_0, p_1)$:

$$\begin{aligned} c_0 + sc_1 &= \Delta m + p_0 u + e_0 + sp_1 u + se_1 \\ &= \Delta m - ue_{\mathsf{cpk}} + e_0 + se_1, \end{aligned}$$

where we substituted the expression of $\mathsf{BFV.Encrypt}$. As $\|u\| = 1$ and $\|e_i\| \leq B$ for $i = 0, 1$, Eq. (2.6) follows.

Derivation of Eq. (2.7)

From the decryption expression of ct',

$$\begin{aligned} c'_0 + s'c_1 &= c_0 + \sum_j ((-s'_j + s_j)c_1 + e_{\mathsf{CKS},j}) + s'c_1 \\ &= c_0 + sc_1 + \sum_j e_{\mathsf{CKS},j} \\ &= \Delta m + e_{fresh} + \sum_j e_{\mathsf{CKS},j}. \end{aligned}$$

As $e_{\mathsf{CKS},j} \leq B_{smg}$, Eq. (2.7) follows.

Derivation of Eq. (2.8)

From the decryption expression of ct',

$$\begin{aligned} c'_0 + s'c'_1 &= c_0 + \sum_j (s_j c_1 + u_j p'_0 + e_{0,j}) + s' \sum_j (u_j p'_1 + e_{1,j}) \\ &= c_0 + sc_1 + up'_0 + s'up'_1 + \sum_j e_{0,j} + se_{1,j} \\ &= \Delta m + e_{fresh} + \sum_j u_j e_{\mathsf{pk}'} + e_{0,j} + s'e_{1,j}, \end{aligned}$$

and Eq. (2.8) follows.

A.3 Proof of Theorem 1

First, we observe that Theorem 1 states that there is at least one honest player that we denote P_h. The choice for P_h, among multiple honest parties, does not reduce generality. We denote $\mathcal{H}$ the set $\mathcal{P} \setminus (\mathcal{A} \cup \{P_h\})$ of all other honest parties. Hence, the tuple $(\mathcal{A}, \mathcal{H})$ can represent any partition of $\mathcal{P} \setminus \{P_h\}$. In particular, both $\mathcal{A}$ and $\mathcal{H}$ can be empty in the following arguments. To simplify the notation, we consider the various error terms sampled as a part of the protocols as private inputs to these protocols (as if they were sampled before the protocol starts). We proceed by constructing simulators for each sub-protocol. For a given value x, we denote $\tilde{x}$ its simulated equivalent.

We observe that the $\Pi_{\mathsf{PubKeySwitch}}$, $\Pi_{\mathsf{Enc2Share}}$, $\Pi_{\mathsf{Share2Enc}}$ and $\Pi_{\mathsf{ColBootstrap}}$ protocols can all be derived from the $\Pi_{\mathsf{KeySwitch}}$ protocol and their associated simulators can be straightforwardly adapted from $S^{\mathsf{KeySwitch}}$.

Construction of $S^{\mathsf{EncKeyGen}}$ The output of the $\Pi_{\mathsf{EncKeyGen}}(\{s_i, e_i\}_{P_i \in \mathcal{P}})$ protocol is $\mathsf{cpk} = (p_0, p_1)$ as defined in Equation (2.2) and its transcript is the tuple $(p_{0,1}, p_{0,2}, \ldots, p_{0,N})$ of all the players' shares; this tuple corresponds to an additive sharing of p_0. $S^{\mathsf{EncKeyGen}}$ can simulate these shares by randomizing them under two constraints: (1) The simulated shares must sum up to p_0, and (2) the adversary shares must be equal to the real ones (otherwise, it could easily distinguish them). Hence, $S^{\mathsf{EncKeyGen}}$ generates the share $\widetilde{p}_{0,i}$ of party P_i as

$$\widetilde{p}_{0,i} = \begin{cases} [-s_i p_1 + e_i]_q & \text{if } P_i \in \mathcal{A} \\ \leftarrow R_q & \text{if } P_i \in \mathcal{H} \\ [p_0 - \sum\limits_{P_j \in \mathcal{A} \cup \mathcal{H}} \widetilde{p}_{0,j}]_q & \text{if } P_i = P_h \,. \end{cases}$$

Lemma 1. *For the adversary as defined in Theorem 1, it holds that* $(\widetilde{p}_{0,1}, \widetilde{p}_{0,2}, \ldots, \widetilde{p}_{0,N}) \stackrel{c}{\equiv} (p_{0,1}, p_{0,2}, \ldots, p_{0,N})$.

Proof (informal). We first observe that, when $\mathcal{H} = \emptyset$, $S^{\mathsf{EncKeyGen}}$ outputs the real view and the statement trivially holds. When $\mathcal{H} \neq \emptyset$, all $\widetilde{p}_{0,i},\ P_i \in \mathcal{H}$ are uniformly random in R_q and $\widetilde{p}_{0,h}$ is pseudo-random (because $[\sum_{P_j \in \mathcal{H}} \widetilde{p}_{0,j}]_q$ is pseudo-random). Indeed, any polynomial-time adversary distinguishing $(\widetilde{p}_{0,i}, p_1)$ from $(p_{0,i}, p_1)$ with non-negligible probability would directly yield a distinguisher for the decision-RLWE problem.

Construction of $S^{\mathsf{RelinKeyGen}}$ The output of the $\Pi_{\mathsf{RelinKeyGen}}(\{s_i, u_i, e_{0,i}, e_{1,i}, e_{2,i}, e_{3,i}\}_{P_i \in \mathcal{P}})$ protocol is $\mathsf{rlk} = (\mathbf{r}_0, \mathbf{r}_1)$, the relinearization key defined in Eq. (2.3). Its transcript consists of two rounds for which each party discloses a share in $R_q^{2\times l}$: $(\mathbf{h}_1, ..., \mathbf{h}_N, \mathbf{h}'_1, ..., \mathbf{h}'_N)$. These shares represent an additive sharing of values $\mathbf{h} = (\mathbf{h}^{(0)}, \mathbf{h}^{(1)})$ and $\mathbf{h}' = (\mathbf{h}'^{(0)}, \mathbf{h}'^{(1)})$, with the constraints that $\mathbf{r}_0 = \mathbf{h}'^{(0)} + \mathbf{h}'^{(1)}$ and $\mathbf{r}_1 = \mathbf{h}^{(1)}$. Hence, similar to $S^{\mathsf{EncKeyGen}}$, they can be generated for the honest parties by randomizing them under these constraints. Specifically, $S^{\mathsf{RelinKeyGen}}$ outputs $(\widetilde{\mathbf{h}}_1, ..., \widetilde{\mathbf{h}}_N, \widetilde{\mathbf{h}}'_1, ..., \widetilde{\mathbf{h}}'_N)$ where

$$\widetilde{\mathbf{h}}_i = \begin{cases} ([-u_i\mathbf{a} + s_i\mathbf{w} + \mathbf{e}_{0,i}]_q \ ,\ [s_i\mathbf{a} + \mathbf{e}_{1,i}]_q) & \text{if } P_i \in \mathcal{A} \\ \leftarrow R_q^{2\times l} & \text{if } P_i \in \mathcal{H} \\ (\leftarrow R_q^l \ ,\ [\mathbf{r}_1 - \sum\limits_{P_j \in \mathcal{A} \cup \mathcal{H}} \widetilde{\mathbf{h}}_j^{(1)}]_q) & \text{if } P_i = P_h \end{cases},$$

$$\widetilde{\mathbf{h}}'_i = \begin{cases} ([s_i\widetilde{\mathbf{h}}^{(0)} + \mathbf{e}_{2,i}]_q \ ,\ [(u_i - s_i)\widetilde{\mathbf{h}}^{(1)} + \mathbf{e}_{3,i}]_q) & \text{if } P_i \in \mathcal{A} \\ \leftarrow R_q^{2\times l} & \text{if } P_i \in \mathcal{H} \\ (\mathbf{b} \leftarrow R_q^l \ ,\ [\mathbf{r}_0 - \mathbf{b} - \sum\limits_{P_j \in \mathcal{A} \cup \mathcal{H}} \widetilde{\mathbf{h}}_j'^{(1)}]_q) & \text{if } P_i = P_h \end{cases},$$

Lemma 2. *For the adversary as defined in Theorem 1, it holds that*

$$(\widetilde{\mathbf{h}}_1, ..., \widetilde{\mathbf{h}}_N, \widetilde{\mathbf{h}}'_1, ..., \widetilde{\mathbf{h}}'_N) \stackrel{c}{\equiv} (\mathbf{h}_1, ..., \mathbf{h}_N, \mathbf{h}'_1, ..., \mathbf{h}'_N)$$

Proof (sketch). We first observe that, for the first round, $(-u_i\mathbf{a} + \mathbf{e}_{0,i}, \mathbf{a})$ and $(-s_i\mathbf{a} + \mathbf{e}_{1,i}, \mathbf{a})$ are two l-tuples of RLWE samples (with secrets s_i and u_i) and the same argument as for Lemma 1 applies (l times). Therefore, they can be considered pseudo-random and can be generated by the simulator. Next, we observe that $\mathbf{h}$, their sum in $R_q^{2\times l}$, is also pseudo-random. Hence, the shares of the second round can be considered as two sets of l *fresh* RLWE challenges $(s_i\mathbf{h}^{(0)} + \mathbf{e}_{2,i}, \mathbf{h}^{(0)})$ and $((u_i - s_i)\mathbf{h}^{(1)} + \mathbf{e}_{3,i}, \mathbf{h}^{(1)})$. This corresponds to recursively applying the RLWE assumption

to prove that, for $s, u \leftarrow R_3, e_0, e_1 \leftarrow \chi, a \leftarrow R_q$, the distribution $(usa + ue_0 + e_1, sa + e_0, a)$ is indistinguishable from the uniform distribution in R_q^3. For example, the core RLWE scheme encryption relies on this assumption for encrypting ciphertexts with an RLWE public-key.

Output of $\Pi_{\mathsf{KeySwitch}}$ The security argument for the $\Pi_{\mathsf{KeySwitch}}$ protocol is inherently more complex than the previous ones, as the real protocol output only approximates the ideal one. This enables us to formally express and characterize the need for *smudging* in (R)LWE-based MHE schemes. Given a ciphertext $\mathsf{ct} = (c_0, c_1)$ decrypting under s, the ideal functionality of the $\Pi_{\mathsf{KeySwitch}}$ protocol is to compute $\mathsf{ct}' = (c_0', c_1)$ such that $c_0' + s'c_1 = \mathsf{Decrypt}(s, \mathsf{ct}) + e_{smg}$, where e_{smg} is a *fresh* noise term sampled from an error distribution χ_{smg}. Indeed, this noise must be fresh in order to not leak the error terms in ct to the output-key holder. Hence, the *ideal* output of the $\Pi_{\mathsf{KeySwitch}}$ protocol is $f_{\mathsf{KeySwitch}}(\{s_i, s_i', e_i'\}_{P_i \in \mathcal{P}}, \mathsf{ct}) = \hat{h}$ such that $\mathsf{ct}' = (c_0 + \hat{h}, c_1)$ satisfies the above equation. However, the real output of the protocol $\Pi_{\mathsf{KeySwitch}}(\{s_i, s_i', e_i'\}_{P_i \in \mathcal{P}}, \mathsf{ct}) = h$ differs from the ideal one in that it contains the error of ct. Simulation-based proofs permit this difference, as long as it can be proven that the ideal and real outputs are indistinguishable for the adversary. We formulate the property as Lemma 3. Then, we show that, even when the adversary has access to the real output, the adversary cannot distinguish the simulated view from the real one. This is enunciated as Lemma 4.

Lemma 3. *Let $\hat{h} = f_{\mathsf{KeySwitch}}(\{s_i, s_i', e_i\}_{P_i \in \mathcal{P}}, \mathsf{ct})$ be the ideal output of the $\Pi_{\mathsf{KeySwitch}}$ protocol and $h = \Pi_{\mathsf{KeySwitch}}(\{s_i, s_i', e_i\}_{P_i \in \mathcal{P}}, \mathsf{ct})$ be its real output. For any adversary as defined in Theorem 1 provided with $s' = \sum_{P_i \in \mathcal{P}} s_i'$, it holds that $\hat{h} \overset{c}{\equiv} h$.*

Proof (sketch). The adversary's knowledge of s' enables the extraction of the noise of ct' as $e_{\mathsf{ct}'} = c_0' + h + s'c_1 - \Delta m$. This noise component has the form $e_{\mathsf{ct}} = e_{\mathsf{ct}} + e_{smg}$ where e_{ct} is the noise after a usual decryption of ct with s and $e_{smg} = \sum_{P_i \in \mathcal{P}} e_i$ is the sum of all the fresh noise terms added as a part of the $\Pi_{\mathsf{KeySwitch}}$ protocol. Hence, for Lemma 3 to hold, the distribution of $e_{\mathsf{ct}} + e_{smg}$ must be indistinguishable from that of e_{smg}.

We observe that both e_{ct} and e_{smg} follow centered Gaussian distributions of different variances that we denote σ_{ct}^2 and σ_{smg}^2, respectively. From the protocol definition, we know that each e_i is sampled from a $\chi_{\mathsf{KeySwitch}}$ of variance $2^\lambda \sigma_{\mathsf{ct}}^2$. Hence, the ratio $\sigma_{\mathsf{ct}}^2 / \sigma_{\mathsf{KeySwitch}}^2$ is negligible, and $e_{\mathsf{ct}} + e_{smg}$ is *statistically indistinguishable* from e_{smg} by the Smudging lemma [AJLT+12].

Construction of $S^{\mathsf{KeySwitch}}$ The transcript of the $\Pi_{\mathsf{KeySwitch}}$ protocol is a tuple of the parties' shares $(h_1, h_2, ..., h_N)$ that constitute an additive sharing of h in R_q. This transcript is simulated by $S^{\mathsf{KeySwitch}}$ as $(\widetilde{h}_1, \widetilde{h}_2, ..., \widetilde{h}_N)$ where

$$\widetilde{h}_i = \begin{cases} [(-s_i + s_i')c_1 + e_i']_q & \text{if } P_i \in \mathcal{A} \\ a_i \leftarrow R_q & \text{if } P_i \in \mathcal{H} \\ [h - \sum\limits_{P_i \in \mathcal{A} \cup \mathcal{H}} \widetilde{h}_i]_q & \text{if } P_i = P_H. \end{cases}$$

Lemma 4. *For the adversary as defined in Theorem 1, it holds that $(\widetilde{h}_1, \widetilde{h}_2, ..., \widetilde{h}_N) \overset{c}{\equiv} (h_1, h_2, ..., h_N)$.*

Proof (sketch). When considering the distribution of the simulated and real views alone, the usual decision-RLWE assumption suffices: $(-s_i c_1 + e_i', c_1)$ is indistinguishable from $(a \leftarrow R_q, c_1)$ for an adversary that does not know s_i and e_i'. However, we need to jointly consider this distribution and the real output. We recall that an adversary who has access to s' can extract $e + e'$ from the output and might be able to estimate e_i' for $i \notin \mathcal{A}$. Consequently, we need to make sure that

the uncertainty the adversary has in estimating e'_i is sufficiently large to protect each share h_i in the $\Pi_{\mathsf{KeySwitch}}$ protocol. We formalize this requirement as

Condition 1. *An input ciphertext* (c_0, c_1) *to the* $\Pi_{\mathsf{KeySwitch}}$ *protocol is such that* $c_0 + sc_1 = m + e_{\mathsf{ct}}$ *where* $e_{\mathsf{ct}} = e_{\mathcal{A}} + e_h$ *includes a term* e_h *that is unknown to, and independent from, the adversary. Furthermore,* e_h *follows a distribution according to the RLWE hardness assumptions.*

If Condition 1 holds, we know that $\mathcal{A}$ can only approximate the term e_h up to an error $e_{\mathsf{ct},h}$; this is enough to make (h_h, c_1) indistinguishable from $(a \leftarrow R_q, c_1)$. In the scope of the $\Pi_{\mathsf{MHE\text{-}MPC}}$ protocol, as long as all parties provide at least one input (for which the noise will be fresh), the requirement of Condition 1 is satisfied.

CHRISTIAN MOUCHET

christian.mouchet@bluewin.ch – https://cmct.ch

EDUCATION

École polytechnique fédérale de Lausanne (EPFL) Lausanne, Switzerland

– *Ph.D. in Computer Science* 2023

Advisor: Carmela Troncoso, Co-Advisor: Jean-Pierre Hubaux
Dissertation: *Multiparty Homomorphic Encryption: from Theory to Practice*

– *M.Sc. in Computer Science* 2017

Minor: Information Security
Master thesis: *Homomorphic Lattice-based Cryptography for Secure Distributed Computation*

– *B.Sc. in Computer Science* 2014

Collège Calvin Geneva, Switzerland

– *Swiss federal high school diploma* 2010

WORK EXPERIENCE

Kudelski Security, Kudelski Group Chesaux, Switzerland

– *Security Engineer Extern* Feb 2016 - Jul 2017

In the Managed Security Services department during the early stages of it's new *Threat Monitoring Service*, I developed a model and associated software solution to help them abstract the complexity and diversity of their customer's infrastructure and requirements.

– *Security Engineer Intern* Jul. 2016 - Feb 2017

In the Cyber Fusion Center, I evaluated the service-critical data-source monitoring solutions and demonstrated that they were, at the time, insufficient.

Swiss Armed Forces Switzerland

– *Mechanized Infantry Group Leader, Sergeant* 2011

ACTIVITIES

Lattigo: A Multiparty Homomorphic Encryption Library in Go 2018-Present

I am co-authoring and maintaining the Lattigo open-source library, an advanced cryptographic library implementing the main fully homomorphic encryption schemes and their multiparty variants. The library is now well established in the community and is now a collaboration between EPFL and Tune Insight SA.

Research Projects Supervision (EPFL)

- *Helium: Implementation of an end-to-end encrypted MPC framework* *Spring 2023*
  Semester project by Giovanni Torrisi
- *Design and implementation of a multiparty homomorphic encryption circuit evaluator* *Fall 2022*
  Semester project by Adrian Cucos
- *Implementation of a network layer for multiparty homomorphic encryption* *Fall 2021*
  Semester project by Manon Michel
- *Implementation of a threshold homomorphic encryption scheme* *Spring 2021*
  Semester project by Adrien Laydu
- *Implementation of a multikey homomorphic encryption scheme* *Spring 2021*
  Semester project by Hedi Sassi and Walid Ben Naceur
- *Cloud-based secure multiparty computation using homomorphic encryption* *Fall 2020*
  Semester project by Anas Ibrahim and Vincent Parodi

- *Implementation of multiparty homomorphic encryption schemes* *Fall 2020*
  Semester Project by Clémence Altmeyerhenzien
- *Profiling and optimization of an homomorphic encryption library* *Spring 2020*
  Semester project by Elie Daou
- *Implementation of secure multiparty computation using homomorphic encryption* *Spring 2020*
  Semester project by Elia Anzuoni
- *Practicality analysis of a threshold cryptosystem based on RLWE* *Spring 2020*
  Master thesis by Elliott Bertrand
- *Lattice-based signature and key-exchange protocols for the Onet library* *Fall 2019*
  Semester project by Björn Guðmundsson
- *Network layer for lattice-based secure multiparty computation protocols* *Fall 2019*
  Semester project by Johan Lanzrein

TEACHING EXPERIENCE

École polytechnique fédérale de Lausanne (EPFL) Lausanne, Switzerland

– *COM-402: Information security and privacy, Teaching assistant* Fall 2019, 2020, 2021

– *COM-405: Mobile networks, Teaching assistant* Spring 2019, 2020, 2021, 2022

– *CS-523: Advanced topics on privacy enhancing technologies, Teaching assistant* Fall 2018

– *MATH-111: Linear Algebra, Teaching assistant* Fall 2017

Swiss Academy of Engineering Sciences (SATW) Switzerland

– *TecDays module: "AI: Contrôle une colonie de fourmis artificielle", Lecturer* 2019, 2020

Swiss Armed Forces Switzerland

– *Milice Instructor* 2010-2020

ACADEMIC PUBLICATIONS

PELTA – Shielding Multiparty-FHE against Malicious Adversaries
S Chatel, **C Mouchet**, AU Sahin", A Pyrgelis, C Troncoso, JP Hubaux
Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security (CCS 2023, to appear)

An Efficient Threshold Access-Structure for RLWE-Based Multiparty Homomorphic Encryption
C Mouchet, E Bertrand, JP Hubaux
IACR Journal of Cryptology 2023 (JOC 2023)

Multiparty Homomorphic Encryption from Ring-Learning-with-Errors
C Mouchet, J Troncoso-Pastoriza, JP Bossuat, JP Hubaux
Proceedings on Privacy Enhancing Technologies 2021 (PETS 2021)

Efficient bootstrapping for Approximate Homomorphic Encryption with Non-sparse Keys
JP Bossuat, **C Mouchet**, J Troncoso-Pastoriza, JP Hubaux
International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT 2021)

Lattigo: A Multiparty Homomorphic Encryption Library in Go
C Mouchet, JP Bossuat, J Troncoso-Pastoriza, J Hubaux
Workshop on Encrypted Computing & Applied Homomorphic Cryptography (WAHC 2020)

UnLynx: A Decentralized System for Privacy-Conscious Data Sharing
D Froelicher, P Egger, J Sá Sousa, JL Raisaro, Z Huang, **C Mouchet**, B Ford, JP Hubaux
Proceedings on Privacy Enhancing Technologies 2017 (PETS 2017)

SKILLS

Languages	English (fluent), French (mother tongue), German (high-school level)
Programming	Go, Python, Scala, C/C++, Java, JavaScript
Software Tools	Git, LaTeX, Docker, MATLAB, SageMath

AWARDS

– *Teaching Assistant Award, Faculty of Computer and Communication Science, EPFL* 2021

– *Deloitte Zurich Hackaton, Winning team of the forensic track* 2017

– *Audiance Choice Award for the "Event-stream detection project", Big Data Course, EPFL* 2015