DISS. ETH NO. 29319

# USEABLE FULLY HOMOMORPHIC ENCRYPTION

A dissertation submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

ALEXANDER VIAND
M.Sc. in Computer Science, ETH Zurich

born on 26 July 1991
citizen of Germany

accepted on the recommendation of

Prof. Dr. Kenneth Paterson, examiner
Dr. Anwar Hithnawi, co-examiner
Prof. Dr. Raluca Ada Popa, co-examiner

2023

# ABSTRACT

Fully Homomorphic Encryption (FHE) enables arbitrary computations to be performed over encrypted data, eliminating the need to decrypt the data and expose it to potential risk while in use. FHE promises to significantly broaden the range of applications that can be secured with End-to-End encryption. In the last decade, FHE has undergone several breakthroughs and advancements that led to a leap in performance improvements, enabling a variety of applications and a first wave of real-world deployments. However, the complexity of developing an efficient FHE application still hinders deploying FHE in practice and at scale.

FHE presents unique challenges in *development* and *deployment*, which are moving to the foreground as FHE is transitioning from theory to practice. Secure computation techniques such as FHE are inherently interleaved with application logic, as they introduce both theoretical (e.g., data independence) and practical (e.g., cost model) paradigm changes. Programs need to be translated to the unique programming model of FHE, taking into account the security, expressiveness, and performance characteristics of the underlying schemes. Beyond performance, which has been the focus of the community for most of the last decade, and the challenges of development, practical deployments introduce further challenges that have so far received scant attention. Specifically, we need to carefully question to what extent the traditional threat models used in FHE (e.g., semi-honest servers and IND-CPA security) are sufficient for real-world deployments. In order to widen the set of scenarios in which FHE can be deployed effectively, we must define stronger notions of security and develop new constructions to achieve them efficiently.

This dissertation presents three contributions toward useable FHE: First, we study, categorize, and distill the challenges of FHE development, identifying key characteristics that define FHE's unique programming paradigm.

Second, we introduce HECO, a Fully Homomorphic Encryption compiler that translates high-level programs to optimized FHE implementations, enabling non-experts to develop secure and efficient FHE applications. Finally, we present verifiable FHE, a new notion of maliciously secure integrity-preserving FHE that addresses the challenges arising from the mismatch between the traditional threat models used in FHE and real-world deployment scenarios.

# ZUSAMMENFASSUNG

Fully Homomorphic Encryption (FHE) ermöglicht die Durchführung beliebiger Berechnungen über verschlüsselten Daten und beseitigt die Notwendigkeit, die Daten zu entschlüsseln und sie während der Verwendung potenziellen Risiken auszusetzen. FHE hat das Potenzial, die Arten von Anwendungen, die mit Ende-zu-Ende-Verschlüsselung geschützt werden kann, signifikant zu erweitern. Im letzten Jahrzehnt haben verschiedene Durchbrüche und Fortschritte in der FHE Forschung zu erheblichen Leistungsverbesserungen geführt und eine Vielzahl von Anwendungen sowie eine erste Welle von praktischen Einsätzen ermöglicht. Trotz dieser Fortschritte hindert die Komplexität der Entwicklung von effizienten FHE-Anwendungen weiterhin den praktischen Einsatz von FHE im grösseren Massstab.

Die Entwicklung und Implementierung von FHE bringt spezielle Herausforderungen mit sich, die mit dem Wechsel von der Theorie zur Praxis immer stärker in den Vordergrund treten. Methoden für verschlüsselte Berechnungen wie FHE sind untrennbar mit der Anwendungslogik verbunden, da sowohl theoretische Aspekte (wie z. B. Datenunabhängigkeit) als auch praktische Aspekte (wie z. B. Kostenmodelle) paradigmatische Veränderungen mit sich bringen. Programme müssen in das spezifische Programmiermodell von FHE übersetzt werden, wobei die Sicherheits-, Ausdrucks- und Leistungsmerkmale der zugrunde liegenden Schemata zu berücksichtigen sind. Neben der Leistungsverbesserung – die bis jetzt den Schwerpunkt der FHE Forschung bildete – und den Herausforderungen der Entwicklung, stehen bei praktischen Einsätzen weitere Aspekte im Fokus, die bisher nur wenig Beachtung gefunden haben. Insbesondere bedarf es einer sorgfältigen Überprüfung, inwiefern die traditionellen Bedrohungsmodelle, die bei FHE verwendet werden (z. B. semi-honest Server und IND-CPA-Sicherheit), für reale Einsätze ausreichend sind. Um den praktischen Einsatz von FHE auf ein breiteres Spektrum von Anwendungsszenarien auszuweiten, benötigen wir stärkere Sicherheitsdefinitionen und neue Konstruktionen, die diese effizient realisieren können.

Die vorliegende Dissertation präsentiert drei Beiträge zur praxistauglichen Verwendung von FHE: Zunächst studieren, konkretisieren, und kategorisieren wir die Herausforderungen, die sich in der FHE-Entwicklung ergeben. Dabei identifizieren wir die wesentlichen Merkmale, die das einzigartige Programmierparadigma von FHE definieren. Zweitens stellen wir HECO vor, einen FHE Compiler, der high-level Programme in optimierte FHE-Implementierungen übersetzt und es auch Laien ermöglicht, sichere und effiziente FHE-Anwendungen zu entwickeln. Schliesslich führen wir *verifiable FHE* ein, eine neue Definition für integritätswahrende verschlüsselte Berechnungen, dass auf die Herausforderungen, die sich aus den Unterschieden zwischen den traditionell verwendeten Bedrohungsmodellen und praktischen Einsatzszenarien für FHE ergeben, ausgelegt ist.

# ACKNOWLEDGEMENTS

First and foremost, I would like to thank my co-supervisors Anwar Hithnawi and Kenny Paterson for their support, guidance, and mentorship throughout this journey. Their enthusiasm for the field of research has been highly contagious and has fueled my own enthusiasm, making research an incredibly rewarding and enjoyable experience. Their insights, feedback, and encouragement, helped me explore the world of research and equipped me to tackle its challenges. I would also like to thank Friedemann Mattern for giving me the opportunity to start my academic journey in his group, giving me tremendous freedom in my research, and ensuring I would continue to be supported after his retirement. Furthermore, I would like to thank my co-examiner Raluca Ada Popa for her time and effort in reviewing this dissertation.

I would like to thank all my collaborates and co-authors, including the students I had the pleasure of supervising or otherwise working with. A special thank you goes to Patrick Jattke, Miro Haller, and Christian Knabenhans for their contributions to the research that underlies this thesis. I would also like to thank the members of the Distributed Systems Group, Applied Cryptography Group, and the Privacy-Preserving Systems Lab, for providing a great environment both at work and beyond. Thank you especially to Barbara von Allmen for her unrelenting support and for her help with the administrative challenges of this journey.

I have been fortunate to have been able to also spend time in the lab of Tobias Grosser at the University of Edinburg and with Dawn Song at the University of California, Berkeley. Thank you both for graciously hosting me and providing me with the opportunity to explore new areas of research and expand my understanding and skillset. Thank you also to the staff and students in Edinburgh and Berkeley that welcomed me into their groups and made me feel at home during these visits. Furthermore, I extend my thanks to Intel Labs, especially Rosario Cammarota, Michael Steiner, Charlotte Bonte, and the entire DPRIVE team that I worked with during my internship. I also want to acknowledge the wider FHE community, with special mention to Pascal Paillier, for their contributions, encouragement, and support.

Last but certainly not least, I would like to thank my friends and family for their unwavering support during this long journey, and for their encouragement, understanding, and patience, without which this journey would not have been possible.

# CONTENTS

# LIST OF FIGURES

# LIST OF TABLES

# 1 INTRODUCTION

In recent years, a series of high-profile incidents [44, 152] have brought the issue of security and privacy to the forefront of public consciousness, leading to the introduction of privacy regulations and a raised standard for industry best practices. This has led to the near-ubiquitous adoption of encryption at rest and during transit across industry. However, data generally still has to be decrypted before being used, requiring service providers to maintain and manage the associated decryption keys. While in use, the decrypted data is again at risk of compromise, potentially undermining other protections. The desire to address this issue has led to a surge in demand for secure and confidential computing solutions that protect data while in use. In addition to hardware-based confidential computing enclaves, a variety of cryptographic techniques for secure computation have emerged. These include secure Multi-Party Computation (MPC) and Zero-Knowledge Proofs (ZKP), which are both seeing increasing real-world deployment. While these techniques provide powerful primitives that address a variety of important settings, they are not an ideal fit for the standard client-server applications that make up the vast majority of (cloud) applications today. For example, clients in this context frequently do not have the communication bandwidth required to run interactive MPC protocols. More recently, these techniques have been joined by Fully Homomorphic Encryption (FHE), which allows a third-party server to perform computations on encrypted data without the need for interaction with the client. As FHE provides a more natural match for many cloud-based settings, interest in FHE has accelerated in recent years. Gartner projects [73] that "by 2025, at least 20% of companies will have a budget for projects that include fully homomorphic encryption."

FHE is a key technological enabler for secure computation by allowing arbitrary computations to be performed over encrypted data, eliminating the need to decrypt the data and expose it to potential risk while in use.

While first proposed in the 1970s [151], FHE was long considered impossible or impractical. In 2009, breakthrough work from Gentry proposed the first feasible FHE scheme [91], and a first implementation was presented the following year [92]. In the following decade, FHE evolved from a theoretical concept to reality. Thanks to advances in the underlying theory, general hardware improvements, and more efficient implementations, it has become increasingly practical. For example, times for a multiplication between ciphertexts dropped from 30 minutes to less than 20 milliseconds. While this is still around seven orders of magnitude slower than an `IMUL` instruction on a modern CPU, it is sufficient to make many applications practical. These advances have enabled a variety of applications covering a wide range of domains [56, 76, 108, 110, 128, 130, 145]. These include mobile applications, where FHE has been used to encrypt the back end of a privacy-preserving fitness app [135], while continuing to provide a real-time experience. In the medical domain, FHE has been used to enable privacy-preserving genome analysis applications over large datasets [115]. More generally, FHE has been used to solve various well-known problems like Private Set Intersection (PSI) [40], outperforming previous solutions by $2\times$ in running time. In the domain of machine learning, FHE has been used for tasks ranging from linear and logistic regression [114] to Encrypted Neural Network inference [71]. The latter can be used to run privacy-preserving ML-as-a-Service applications, for example, for private phishing email detection [57]. Beyond academic work, FHE is increasingly seeing deployment in real-world applications, such as in Microsoft Edge's password monitor [112]. With upcoming hardware accelerators for FHE promising further speedup [70, 153], FHE will soon be competitive for an even wider set of applications.

While FHE has proven its potential to enable new levels of privacy and security for sensitive information, realizing this potential in practice currently remains difficult. This is because both the *development* and *deployment* of FHE applications present unique challenges. As FHE is transitioning from theory to practice, these issues are rapidly moving to the foreground. FHE introduces issues beyond the inherent challenges of deploying cryptography in practice, such as implementation correctness, side-channel resistance, and key management. This is because traditional cryptography (e.g., secure communication and storage) can frequently be integrated into the transport and storage layer of systems. This essentially hides most of their complexity from applications and enables the simultaneous deployment of such techniques across a variety of applications. Secure computation techniques such as FHE, on the other hand, are inherently interleaved with application logic, as they introduce both theoretical (e.g., data independence) and practical (e.g., cost model) paradigm changes. This process must consider each application individually and requires a deep understanding of the application and the theory and 'folklore' of FHE. Currently, only a small handful of experts have the necessary experience to transform complex applications into efficient state-of-the-art FHE solutions. In order to realize the potential of FHE, we must democratize it by taming this complexity through the development of tools and abstractions. Beyond performance, which has been the focus of the community for most of the last decade, and the challenges of development, practical deployments introduce further challenges that have so far received scant attention. Specifically, we need to carefully question to what extent the traditional threat models used in FHE (e.g., semi-honest servers and IND-CPA security) are sufficient for real-world deployments. In order to widen the set of scenarios in which FHE can be deployed effectively, we must define stronger notions of security and develop new constructions to achieve them efficiently.

## 1.1 THESIS CONTRIBUTIONS

This work aims to unlock the potential of FHE for non-expert developers, democratizing access to FHE and making it accessible to a wider audience. We are the first to consider FHE usability holistically, addressing both the development and deployment aspects of FHE. Our work is the first to study, categorize and distill the challenges of FHE development, bringing them to the attention of a wider audience beyond the core FHE community. Based on these insights, we present a set of tools and abstractions that hide the underlying complexity of working with FHE, enabling non-experts to develop secure and efficient FHE applications. We demonstrate that useability does not have to come at the expense of performance by introducing a series of novel optimizations that map traditional high-level programs to FHE's unique programming paradigm. Finally, we address some of the challenges that arise from the mismatch between the traditional threat models used in FHE and real-world deployment scenarios. More concretely, we present three key contributions, which we discuss below.

*Part I: FHE Application Development (Chapter 3)*

Despite recent breakthroughs in FHE practicality, building secure and efficient FHE-based applications remains a challenging task. This is largely attributed to the differences between traditional programming paradigms and FHE's computation model. Working with FHE also introduces significant engineering challenges. Different schemes offer varying performance tradeoffs, and optimal choices are heavily application-dependent. To address some of the engineering challenges in this space, we have seen a surge of work on tools that aim to improve accessibility and reduce barriers to entry in this field [6, 14, 16, 34, 48, 65, 66, 71, 72]. However, we currently lack a comprehensive overview of the current state of FHE development. While it is clear that both significant advances have been made and many challenges remain open, there is no systematic understanding of the remaining engineering challenges that need to be addressed to help broaden FHE adoption. Therefore, this part of the thesis aims to fill in this knowledge gap by studying and surveying the current state-of-the-art of FHE tools. More concretely, this survey has two objectives: First, to assist developers looking to develop FHE-based applications in selecting a suitable approach and, second, to provide the community with valuable insights on both successes and remaining issues in this space.

Towards this goal, we conduct an extensive survey of existing tools and highlight their features and characteristics. Subsequently, we consider these tools in practice by experimentally evaluating them across a range of case study applications, contrasting usability, expressiveness, and performance.

In our experimental evaluation, we consider a selection of tools in more detail and provide an in-depth analysis of their usability and expressiveness in practice. We implement and benchmark three case-study applications that represent different domains of FHE-based computation. Our benchmarks allow us to study not only the overall performance of FHE for these applications across tools but also the relative strengths of different tools compared to each other. We provide an online repository[1] that includes Docker images for all the tools we evaluate, our automated benchmarking framework, and the example applications.

*Part II: End-to-End Compilation for FHE (Chapter 4)*

Today, performance is no longer the major barrier to the adoption of FHE. Instead, it is the complexity of effectively translating applications to FHE that currently limits deploying FHE in practice and at scale. Several FHE compilers have emerged recently to ease FHE development. However, none of these answer how to automatically transform imperative programs to secure and efficient FHE implementations. This is a fundamental issue that needs to be addressed before we can realistically expect broader use of FHE. Automating these transformations is challenging because the restrictive set of operations in FHE and their non-intuitive performance characteristics require programs to be drastically transformed to achieve efficiency. Moreover, existing tools are monolithic and focus on individual optimizations and, as a result, fail to fully address the needs of end-to-end FHE development. In order to overcome these limitations, we need to fundamentally *rethink the architecture of FHE compilers* and *develop novel optimizations* that abstract away the complexity FHE and address the limitations of existing tools. In this part of the thesis, we present HECO, a new multi-stage optimizing FHE compiler. Our architecture provides, for the first time, a true end-to-end toolchain for FHE development. In addition, we propose novel transformations and optimizations that map imperative programs to the unique programming model of FHE.

1 `https://github.com/MarbleHE/SoK`

Based on the progression of abstraction levels that expert developers naturally consider when designing FHE applications, we identified four phases of converting an application to an efficient FHE implementation: *program transformation*, *circuit optimization*, *cryptographic optimization*, and *target optimization*. Compilers need to be able to accommodate a wide variety of optimizations across all levels of abstraction and we propose a set of Intermediate Representations (IRs) based on the requirements of each phase that allow us to naturally and efficiently express optimizations at these different levels. We realize these IRs using the MLIR compiler framework [118] which provides a standardized way to define and operate on domain-specific IRs. In our design, we take a broader view of FHE development, extending the scope of optimizations beyond the cryptographic challenges on which existing tools focus. Experts spend significant time considering *how* to best express an application in the FHE paradigm, only considering the other aspects once the program is *efficiently* expressible using native FHE operations. HECO supports the automatic transformation of high-level programs to FHE's unique programming paradigm, allowing developers to express their algorithms conveniently in the standard imperative paradigm. In our evaluation, we show that HECO can match the performance of expert implementations, providing up to 3500x speedup over naive implementations. We open-source HECO and we hope that it will help to advance the FHE development ecosystem.

*Part III: Verifiable Fully Homomorphic Encryption (Chapter 5)*

The same malleability that enables homomorphic computations also raises *integrity* issues, which have so far been mostly overlooked. Specifically, the FHE research community has historically made extensive use of the assumption that the server running an FHE application would be honest-but-curious, rather than actively malicious. This assumption may be reasonable in some deployment scenarios, but the necessity to trust the server to this extent limits the scope of application scenarios. In addition, even otherwise trusted parties can be compromised by malicious third parties, exposing this attack surface. A violation of the semi-honest assumption threatens not only correctness but also confidentiality: a class of attacks known as *key-recovery attacks* exploits the interactive nature of real-world deployments to construct (partial) decryption oracles, and practical key-recovery attacks have been developed for all major FHE schemes [37, 43, 167]. Therefore, there is an urgent need to strengthen FHE to maintain strong guarantees in the context of these attacks. While there has been work that aims to address this gap, these have remained isolated efforts considering only aspects of the overall problem and fail to fully address the needs and characteristics of modern FHE schemes and applications. This work is the first to consider FHE integrity in the context of real-world FHE deployment settings This part of the thesis aims to both highlight the dangers arising from the gap between existing notions and real-world scenarios, and to propose efficient instantiations of a new robust notion for FHE integrity that effectively addresses these challenges.

We are the first to compare and analyze the different existing approaches to FHE integrity. We show how these approaches fall short when considering how FHE is used in practice, highlighting the mismatch between the setting assumed in the existing integrity literature and the settings used for the vast majority of FHE applications. Finally, we show how this mismatch enables attacks on both correctness and confidentiality, even in the presence of existing integrity mechanisms. We define a new notion of integrity for FHE that captures real-world FHE deployment settings, addressing the issues we identified in our analysis. Existing notions are usually a complex combination of existing integrity notions and ad-hoc confidentiality properties, interleaving FHE and integrity aspects. In contrast, we present a natural clean-slate notion of verifiable FHE that composes the standard notion of FHE with modular integrity properties. This allows our notion to be adapted to the wide variety of FHE deployment settings we observe in practice. We show how to generically construct our notion from a standard FHE scheme, commitments, and Zero-Knowledge Proofs (ZKPs). and then instantiate our notion of using a variety of different state-of-the-art ZKP systems. In the process, we highlight a series of fundamental challenges in bringing together FHE and ZKP systems. We investigate several approaches to bridge this gap, evaluate our instantiations on a variety of different workloads and compare them to a hardware-attestation–based approach (FHE-in-TEE) as a point of comparison. We show that verifiable FHE can be practical but also highlight the need for future work on ZKP systems specifically designed for the unique characteristics of FHE.

## 1.2 PUBLICATIONS

The material in this thesis is based on the following publications:

- (Chapter 3) **Alexander Viand**, Patrick Jattke, Anwar Hithnawi, SoK: Fully Homomorphic Encryption Compilers, *In 2021 IEEE Symposium on Security and Privacy, SP 2021.* [161].
- (Chapter 4) **Alexander Viand**, Patrick Jattke, Miro Haller, Anwar Hithnawi, HECO: Fully Homomorphic Encryption Compile, *In 32nd USENIX Security Symposium (USENIX Security 23)* [4]
- (Chapter 5) **Alexander Viand***, Christian Knabenhans*, Anwar Hithnawi, Verifiable Fully Homomorphic Encryption, *In Arxiv arXiv:2023.07041 [cs.CR], Under Submission.* [162]

Any errors in this dissertation are, of course, mine alone. During the course of my doctoral studies I also co-authored the following publications below, in addition to a number of smaller works [106, 132, 137, 163].

- Lukas Burkhalter, Anwar Hithnawi, **Alexander Viand**, Hossein Shafagh, Sylvia Ratnasamy, TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control, *In 17th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2020.* [27]
- Lukas Burkhalter*, Nicolas Küchler*, **Alexander Viand**, Hossein Shafagh, and Anwar Hithnawi, Zeph: Cryptographic Enforcement of End-to-End Data Privacy, *In 15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021.* [28]

* Equal contribution.

- Hidde Lycklama*, Lukas Burkhalter*, **Alexander Viand**, Nicolas Küchler, Anwar Hithnawi, RoFL: Attestable Robustness for Secure Federated Learning, *In 2023 IEEE Symposium on Security and Privacy, SP 2023* [131]
- Nicolas Küchler, Emanuel Opel, Hidde Lycklama, **Alexander Viand**, Anwar Hithnawi, Cohere: Privacy Management in Large Scale Systems, *In Arxiv arXiv:2023.08517 [cs.CR], Under Submission.* [116]

## 1.3 SOFTWARE ARTEFACTS

We make the code for each work in this thesis available as open source.

- (Chapter 3)
  - `https://github.com/MarbleHE/SoK`
- (Chapter 4)
  - `https://github.com/MarbleHE/HECO`
- (Chapter 5)
  - `https://github.com/MarbleHE/ringSNARK`
  - `https://github.com/MarbleHE/FHE-in-TEE`
  - `https://github.com/MarbleHE/ZKP-FHE`

# 2 PRELIMINARIES

In this chapter, we provide an overview of relevant concepts, focusing on high-level descriptions and referring to the relevant literature for formal definitions.

## 2.1 FULLY HOMOMORPHIC ENCRYPTION

A *homomorphic* encryption scheme is a (most frequently public-key) encryption scheme where there exists a homomorphism between operations on the plaintext and operations on the ciphertext:

$$\mathrm{Dec}(\mathrm{Enc}(x+y)) = \mathrm{Dec}(\mathrm{Enc}(x) \oplus \mathrm{Enc}(y))$$

where $+$ and $\oplus$ are operations over the plaintext and ciphertext space, respectively. A *fully homomorphic* encryption scheme is one that is homomorphic in regards to both addition and multiplication. We refer to [90] for a formal definition, including several constraints that apply to exclude trivial constructions.

Addition and multiplication allow us to compute any polynomial function over the encrypted data but many frequently-used functions like comparisons or sorting are non-polynomial, i.e., cannot (easily) be expressed as polynomial functions. However, multiplication and addition in $\mathbb{Z}_2$ can be used to emulate `AND`- and `XOR`-gates, respectively. Together with memory, this can emulate arbitrary computations [155].

**Foundations of practical FHE.** Modern FHE schemes date back to 2009 when Gentry presented the first feasible FHE construction [90]. While the original scheme had impractically large constant overheads, follow-up work improved upon the scheme, enabling a first implementation [92]. All modern schemes follow the general approach laid out by Gentry's first scheme: In these schemes, public keys are values that cancel out to zero when combined with the secret key $\mathsf{sk}$. Encryption multiplies the public key $\mathsf{pk}$ with a random number $a$ and adds the message $m$. For two ciphertexts $x_i = \mathsf{pk} * a_i + m_i$, their addition $x_0 + x_1 = m_0 + m_1 + \mathsf{pk} * (a_0 + a_1)$ trivially decrypts to $m_0 + m_1$. Multiplications are generally more complex and are handled differently by different schemes. For a secure system, some *noise* $e$ must be added to public keys and ciphertexts. As long as $e$ is sufficiently small, $m + e'$ can be rounded to the correct value; applying the secret key then recovers $m$. During homomorphic operations, the noise in the ciphertext grows. While this effect is negligible during additions, multiplying two ciphertexts introduces significantly more noise. This limits computations to a (parameter-dependent) number of consecutive multiplications (multiplicative *depth*) before decryption fails. This limitation can be circumvented using *bootstrapping*, which resets the noise level of a ciphertext to a fixed lower level by homomorphically evaluating the decryption circuit with an encrypted secret key as input. However, the decryption circuit needs to be sufficiently low-depth to allow at least one additional multiplication before needing to bootstrap again.

**Second Generation Schemes.** While the first generation of FHE presented a significant academic breakthrough, it was too inefficient (e.g., around 30 min needed to compute a single homomorphic multiplication) to truly enable practical applications of FHE. In response, a second generation of schemes like the Brakerski-Gentry-Vaikuntanathan (BGV) [22] and Brakerski/Fan-Vercauteren (BFV) [23, 78] schemes evolved. In order to overcome the performance penalties of bootstrapping, they introduced the concept of *leveled* homomorphic encryption. Here, the parameters are chosen sufficiently large to evaluate the entire computation without bootstrapping. While there is a cut-off point after which bootstrapping is more efficient, this is unlikely to be reached by most programs. In addition, they introduced support for Single Instruction, Multiple Data (SIMD)-style *batching*. This exploits the fact that the plaintext space is a ring of polynomials with many coefficients. Using the Chinese Remainder Theorem [107], this can be reinterpreted as many different independent slots and many different messages (usually

$2^{13}$–$2^{16}$) can be packed into a single ciphertext. Automorphisms additionally enable homomorphically executable rotations between slots [103].

The Cheon-Kim-Kim-Song (CKKS) scheme [47] introduces a further optimization, considering homomorphic encryption for *approximate* numbers. While it follows a very similar construction to BGV, it is formally speaking not an FHE scheme since the result is only approximately the same as the equivalent plaintext operation, which can introduce subtle issues in practice. However, this relaxation has led to an extremely efficient scheme. CKKS is designed primarily for computations with fixed-point numbers, i.e., a number $x$ is represented as $m = \lfloor x * \Delta \rceil$ for *scale* $\Delta$, usually a large integer. CKKS introduces a homomorphic rounding operation to reduce the scale homomorphically, avoiding overflow issues.

**Third Generation Schemes.** More recently, a third generation of FHE schemes, based on the Gentry-Sahai-Waters (GSW) scheme [94], has emerged. These schemes mostly abandon batching and leveled HE and instead focus on fast bootstrapping. For example, implementations of the Chillotti-Gama-Georgieva-Izabachene (CGGI[1]) scheme [50, 51] can perform bootstrapping in less than 0.1 seconds, while bootstrapping for BFV or BGV usually takes several minutes even in efficient implementations. While initially limited to binary settings, recent follow-up work [56] extends this to arithmetic circuits. However, fast bootstrapping is incompatible with batching, introducing a trade-off between latency and throughput when compared to second-generation schemes.

**Security.** Modern FHE schemes rely on post-quantum hardness assumptions, widely believed to be secure for the foreseeable future. The community has developed estimates of their concrete hardness [3] and parameter choices for several FHE schemes have been standardized [2]. Nevertheless, some attention must be paid to security when using FHE. For example, FHE does not provide *integrity* by default, i.e., a server might perform a different calculation than requested or none at all. There exist techniques to address that, ranging from zero-knowledge-proofs to hardware attesta-

1 The CGGI scheme is more commonly known as TFHE, however we refer to it by the author initials in order to avoid confusion with the TFHE library.

tion [25]. Additionally, FHE does not provide by default *circuit privacy*, i.e., a client might be able to learn information about the applied circuit. Different techniques, varying in practicality and protection level, can be used to address this [19, 20, 75]. Finally, issues can appear when using *approximate* homomorphic encryption (e.g., CKKS), with attacks that can recover the secret key from the noise embedded in ciphertext decryptions [124]. Recent work has shown how adding differentially private noise can mitigate these attacks [125], but some concerns remain.

**FHE and MPC.** Finally, we briefly consider FHE in the wider context of secure MPC. While FHE could be used to realize many 2-party MPC protocols, it does not by default offer *circuit privacy*, i.e., does not hide the function being computed. Where desired, this is usually addressed in practice via *noise flooding* [90], i.e., adding large noise to the final result before returning it to the client. FHE can also be extended to multi-party or multi-key settings. In multi-party FHE, different entities generate a public key and shares of a secret key [139]. In multi-key FHE, each entity independently generates their secret and public key [39]. There are also hybrid schemes that combine FHE and MPC [111] or different FHE schemes [18]. We only consider the two-party FHE-only client-server setting, but many of the concepts transfer directly to the other settings.

## 2.2 FULLY HOMOMORPHIC ENCRYPTION SCHEMES

In this section, we briefly outline important FHE schemes. We focus primarily on aspects relevant to FHE application developers, i.e., plaintext spaces, encodings, and aspects that impact performance.

### 2.2.1 *CGGI*

The Chillotti-Gama-Georgieva-Izabachene scheme [50, 51] is part of a third generation of FHE schemes based on the Gentry-Sahai-Waters (GSW) scheme [94]. More commonly known as TFHE, we refer to it here by the author initials in order to avoid confusion with the TFHE library.

In CGGI, the plaintext and ciphertext space $T$ is a group of polynomials (modulo some irreducible polynomial) of degree up to $n-1$ over the torus $\mathbb{T} = \mathbb{R}/\mathbb{Z}$ (i.e., the real numbers $\mod 1$). The message space is generally chosen so that the computation emulates binary circuits and homomorphic addition becomes XOR and multiplication becomes AND. Since $T$ is not a ring, it supports addition but has no native multiplication operation. However, multiplications are defined between GSW ciphertexts and ciphertexts in $T$. This is used to perform multiplications and non-linear operations over ciphertexts in $T$ during the bootstrapping process, by encrypting the bootstrapping key as a GSW ciphertext. Multiplications between ciphertexts in $T$ are realized as one specific type of such a non-linear transformation applied during bootstrapping. In this *gate-bootstrapped* version of the scheme, every non-linear gate therefore inherently includes bootstrapping.

Chillotti et al. also show how to construct a MUX gate that selects between two ciphertexts in $T$ dependent on a GSW ciphertext and introduce efficient designs for Look-Up-Tables (LUTs). Finally, they show how to use weighted Finite Automata to emulate binary multiplication [53]. However, these techniques are not implemented in the Fast Fully Homomorphic Encryption Library over the Torus (TFHE) library.

### 2.2.2 *BFV*

The Brakerski/Fan-Vercauteren scheme is a second-generation scheme. Fan and Vercauteren [78] ported a scheme by Brakerski [21] to the ring-LWE domain and improved its performance. In BFV, the plaintext space $R_t$ is a ring of polynomials (modulo some irreducible polynomial) of degree up to $n-1$ with coefficients in $\mathbb{Z}_t$. Note that for $t = 2$, we are in the binary circuit setting. Messages $m \in \mathbb{Z}_t$ can be encoded into this plaintext space as a constant polynomial $f(x) = m$. However, this is inefficient as only one of $n$ coefficients is utilized. Simply encoding messages into additional coefficients raises issues when performing computations: while polynomial additions work coefficient-wise, multiplications combine different coefficients in undesired ways. Instead, one can achieve SIMD-style *batching* via the Chinese Remainder Theorem [107]. By choosing $n = \Pi_{i=0}^{k} n_i$ , a degree-$n$ polynomial can be reinterpreted as the multiplication of $k$ lower-degree polynomials. Using this technique, $k$ messages can be packed into a single plaintext, where $k \gg 1000$ in practice, while maintaining meaningful semantics. Automorphisms additionally enable homomorphic rotations of the elements [103].

The ciphertexts, meanwhile, are made up of at elements from $R_q$, which has the same structure as $R_p$, but with a different coefficient modulus $q$. Each ciphertexts consists of at least two elements, i.e., $c = [c_0, c_1]$. These polynomials $c_i$ can themselves be interpreted as coefficients of a polynomial $C(X)$. Homomorphic addition and multiplication between ciphertexts correspond to addition and multiplication between the $C(X)$'s, respectively. As a consequence, the result of a multiplication is a quadratic polynomial, i.e., a ciphertext with three elements $c = [c_0, c_1, c_2]$. During further multiplications the noise term would first become squared, then cubed, etc. growing excessively. Therefore, BFV and similar schemes introduce a *relinearization* procedure to transform ciphertexts back to linear form. We omit a description of bootstrapping and instead note that BFV is more commonly used in leveled mode where the parameters are chosen sufficiently large to complete the computation without bootstrapping.

Using the Chinese Remainder Theorem (CRT), it is possible to *encode* a vector of $n$ integers into a single polynomial, with addition and multiplication acting slot-wise (SIMD). Since the polynomial degree $n$ is usually between $2^{13}$ and $2^{16}$ for security, it can significantly reduce ciphertext expansion and computation cost. BFV also supports rotation operations over such *batched* ciphertexts, which cyclically rotate the vector's elements. Finally, BFV includes a variety of noise-management (or *ciphertext maintenance*) operations, which do not change the encrypted message but can reduce noise growth during computations.

### 2.2.3 *CKKS*

The Cheon-Kim-Kim-Song scheme [47], also known as Homomorphic Encryption for Arithmetic of Approximate Numbers (HEAAN), focuses on homomorphic encryption for *approximate* numbers. Formally speaking it is not an FHE scheme since it only fulfills the requirements approximately, i.e., $\text{Dec}(\text{Enc}(x+y)) \approx \text{Dec}(\text{Enc}(x) \oplus \text{Enc}(y))$, for some operations $+$ and $\oplus$. While this slight relaxation has led to an extremely efficient scheme, some care must be taken when using approximate FHE schemes [46, 124]. CKKS is designed primarily for computations with fixed point numbers, i.e., a number $x$ is represented as $m = \lfloor x * \Delta \rceil$ for *scale* $\Delta$, usually a large integer. While any integer-based scheme can be used for fixed-point computations, they quickly run into overflow issues. CKKS addresses this by introducing a homomorphic rounding operation that reduces the scale of a product back to the original scale $\Delta$.

In CKKS, the logical message space is $\mathbb{C}^n$, i.e., vectors over the complex numbers, although most applications use only the real part. The plaintext space $R$ is a ring of polynomials (modulo some irreducible polynomial) of degree up to $n-1$ with coefficients in $\mathbb{Z}$. Given a scaling factor $\Delta \in \mathbb{R}$, we represent $m \in \mathbb{R}$ as $m' = \lfloor \Delta m \rceil \in \mathbb{Z}$. For brevity, we skip a description of the encoding of such representations into a plaintext polynomial and simply note that the encoding introduces small additional approximation errors. During encryption, noise is intentionally introduced, but this noise overlaps with the least significant bits of the plaintext. Therefore, the approximation error and noise are treated as one, and rather than suddenly losing the message when the noise reaches a threshold, we gradually lose accuracy.

Like in BFV, ciphertexts in CKKS are arrays of elements $c_i \in R_q$ and multiplications require relinearization. However, different to BFV, the noise $e$ grows quadratically with each subsequent multiplication. After $\ell$ multiplications, it has grown to $e^{2^\ell}$ and a modulus $q \approx e^{2^\ell}$ would be required to decrypt the resulting ciphertext correctly. Instead, one can scale the ciphertext down by a factor $\omega$, i.e., go from $R_q$ to $R_{q/\omega}$. This is known as *rescaling* and is similar to the *modulus switching* operation in the Brakerski-Gentry-

Vaikuntanathan (BGV) scheme [22] but rescaling also affects the plaintext. Using rescaling, a modulus of size $(\ell + 1)\omega e$ suffices to evaluate $\ell$ subsequent multiplications. During this operation, the plaintext encrypted in the ciphertext is also effectively rescaled to $\Delta' = \lfloor \Delta/\omega \rceil$. Choosing $q = \Pi_{i=0}^{k} q_i$ where $q_i$ are roughly equally sized primes improves both performance [45] and, by setting $\omega = q_i$, ensures a (nearly) constant scale throughout the computation.

### 2.2.4 *Other Schemes*

We introduced the most commonly implemented and used schemes above. In the following, we briefly highlight additional schemes in the FHE domain. BFV and CKKS are part of a larger family that also includes the Brakerski-Gentry-Vaikuntanathan (BGV) scheme [22] and a variety of predecessor schemes. We skipped an introduction to BGV as it offers similar features to BFV while the latter usually outperforms it in practice [61]. For CGGI, there is also a leveled version of the scheme [50], however, we focused on the gate-bootstrapping version that is currently available in implementations of the scheme. There are also hybrid schemes that combine FHE and MPC [111] or even different FHE schemes [18], offering transformations between ciphertexts in different schemes. These can be used by experts to build highly efficient, special-purpose protocols [115] but are out of scope for this work.

Finally, FHE can be extended to multi-party or multi-key settings. In multi-party FHE, different entities participate an interactive MPC protocol to generate a public key and shares of a secret key. Encryption and computation remain unmodified, but decryption requires running another MPC protocol [139]. In multi-key FHE, each entity independently generates their secret and public key. Homomorphic computation between ciphertexts encrypted under different keys is possible but slower, and the resulting ciphertext can be decrypted only with access to both secret keys [39]. In the following, we will consider only the two-party client-server setting.

# 3 FHE APPLICATION DEVELOPMENT

Though there has been a surge of works on FHE tools and accessibility [6, 14, 16, 34, 48, 65, 66, 71, 72], we currently lack a comprehensive overview of the current state of FHE development. While it is clear that both significant advances have been made and many challenges remain open, there is no systematic understanding of the remaining engineering challenges that need to be addressed to help broaden FHE adoption. Therefore, we aim to fill in this knowledge gap by studying and surveying the current state-of-the-art of FHE tools. More concretely, this survey has two objectives: First, to assist developers looking to develop FHE-based applications in selecting a suitable approach and, second, to provide the community with valuable insights on both successes and remaining issues in this space. Towards this goal, we conduct an extensive survey of existing tools and highlight their features and characteristics. Subsequently, we consider these tools in practice by experimentally evaluating them across a range of case study applications, contrasting usability, expressiveness, and performance. In our experimental evaluation, we consider a selection of tools in more detail and provide an in-depth analysis of their usability and expressiveness in practice. We implement and benchmark three case-study applications that represent different domains of FHE-based computation. Our benchmarks allow us to study not only the overall performance of FHE for these applications across tools but also the relative strengths of different tools compared to each other. We conclude this chapter with a discussion of the current state of FHE and FHE tools. We discuss applications for which FHE is likely practical today and show gaps between state-of-the-art results and what non-expert users can realistically implement. Based on the insights gained through our study, we highlight successes in the FHE tool space and identify gaps that remain to be addressed. Finally, we discuss a possible road map for the next generation of FHE tools.

## 3.1 CHALLENGES OF DEVELOPING FHE APPLICATIONS

The intricacy of the underlying schemes still limits developing FHE-based applications predominantly to experts. Each scheme presents a new set of configurations and performance tradeoffs, and achieving state-of-the-art results requires a high familiarity with the underlying schemes. In addition, FHE imposes a fundamentally different programming paradigm, not only because of the need for data-independent programs but also because efficient solutions frequently require complex vectorization approaches.

Throughout the last decade, a significant amount of *folklore* knowledge around optimization methods and best practices has been built up in the FHE community. However, these techniques and insights are often scattered across the literature or only referred to in passing. As a result, there is a vast gap between state-of-the-art performance results and what non-experts can achieve themselves.

In this section, we provide an overview of the key engineering challenges that developers face today. The community is starting to identify these accessibility issues as a major roadblock to the broader adoption of FHE. Recent works are trying to address these challenges by proposing higher-level interfaces, better abstractions, and automated optimizations. There will most likely always be specific applications that impose additional challenges requiring expert input. However, improved tools can benefit a variety of common application patterns and help ease the path to FHE for many applications.

### 3.1.1 *Parameter Selection*

Selecting secure and efficient instantiations of the underlying cryptographic problems is hard for most encryption schemes. In standard public-key cryptography, we circumvent this by standardizing particular instantiations, e.g., selecting certain elliptic curves, to avoid security issues arising when the underlying hardness assumptions do not hold for poor choices.

FHE introduces the additional challenge of computation-specific parameters. More complex computations require larger plain- and ciphertext moduli to avoid overflow or noise issues. However, as these parameters increase, the Learning With Errors (LWE) problem that security is based on for most schemes becomes easier, and the dimension of the problem space (i.e., polynomial degree) must be increased to compensate. As a result, we cannot standardize a single set of secure parameter choices. Instead, the standardization effort [2] aims to provide a conservative estimate of the security of different combinations of moduli and dimensions. However, since this does not address efficiency, parameter selection remains an issue in developing FHE-based applications.

The time to evaluate homomorphic operations, for a given polynomial degree $n$, is roughly proportional to the ciphertext space modulus $q$, and a smaller $q$ also gives higher security. Therefore, we want to select the smallest $q$ that still correctly decrypts the computation result. However, effectively computing this minimal $q$ remains an open challenge. While formal analyses of the ciphertext noise growth exist for a variety of schemes, these worse-case analyses are frequently too conservative, giving parameters many times larger than the experimentally determined optimum [61]. Also, the plaintext space modulus $t$ required to avoid overflows depends on the size of the actual inputs, which likely come from a smaller subset of $\mathbb{Z}_t$ in practice. Here, again, worst-case analyses lead to impractically conservative parameters. Instead, the community's accepted method is to incrementally decrease $q$ until the computation (on some representative input values) fails to decrypt correctly, then choosing the previous $q$ plus some "safety margin" determined by experience.

### 3.1.2 *Encoding*

With encryption schemes like AES or protocols like TLS, developers do not generally have to consider the plaintext spaces of the underlying encryption schemes. As long as a message can be serialized into a binary string, only padding concerns arise. However, in FHE, the semantics of the plaintext space determine the effect of the homomorphic computations. These semantics, however, frequently do not match the intended application semantics exactly. While this is already a concern in traditional programming, with floating-point accuracy errors or integer overflows, FHE introduces a significantly stronger deviation from the 'ideal' computation model.

For example, while we generally consider $\mathbb{Z}_t$ as the message space for most schemes, most support additional, more complex spaces. For example, BGV supports Galois Fields $\mathbf{GF}(2^d)$ which can be used to efficiently realize AES-FHE transciphering, i.e., converting a standard AES ciphertext to an FHE ciphertext given an encryption of the AES key [93].

Conceptually, binary plaintext spaces (i.e., $\mathbb{Z}_2$) are the easiest to work with since the semantics of homomorphic computations directly correspond to binary circuits. However, working directly with binary circuits is complicated as even trivial functions like addition and multiplication of bit-wise encoded integers require complex algorithms (e.g., Sklansky or Kogge-Stone adders) to implement arithmetic operations efficiently. Therefore, the conceptual ease-of-use is negated by a significant engineering overhead for even simple algorithms.

While using advanced encoding schemes will most likely remain predominantly an expert technique, existing FHE tools have already shown that they can be employed automatically to some extent. For example, nGraph-HE [14] also uses the imaginary part of the CKKS message space when no ciphertext multiplications are required, roughly doubling throughput.

### 3.1.3 *Data-Independent Computation*

Virtually all standard programming paradigms rely on some form of data-dependent execution branching. Traditional iterative programming relies heavily on if/else statements and loops, and even functional programming requires data-dependent branching to terminate recursion.

FHE computations, on the other hand, are by definition data-independent, or they would violate the privacy guarantees. Therefore, FHE computations are frequently conceptualized as *circuits*, i.e., gates (or operations) connected by wires, where the execution follows the same steps, no matter what values the input has. While it is possible to emulate, e.g., if/else branches by calculating the result for both branches and performing a multiplexing selection afterward, this requires evaluating both branches. Simulating (bounded) dynamic-length loops could be achieved by following a similar approach; however, this quickly becomes infeasible in practice.

In addition, many schemes offer the best performance when using integer plaintext spaces ($t \gg 2$). These *arithmetic circuits* are limited to computing polynomial functions. However, many applications, including neural-network inference, can be approximated very well. Therefore, a significant part of developing an FHE-based solution is to consider first whether there exists a polynomial approximation for the task to be performed. Sometimes, this even requires completely switching the approach, e.g., standard algorithms for genomic sequence analysis are not suitable for polynomial approximation, but alternative approaches exist that can be expressed much more easily [119].

### 3.1.4 *SIMD Batching*

One of the major breakthroughs in achieving practical performance in FHE-based solution was the introduction of *batching* or *packing* in second-generation schemes, i.e., allowing one to pack many different messages into a single ciphertext. The resulting SIMD parallelism can trivially be used to improve throughput by packing many different inputs into a single computation run.

However, many FHE applications are limited in their practicality by latency, i.e., non-amortized runtime. State-of-the-art FHE-based solutions virtually always apply batching inside a computation, even on a single instance of the input. Exploiting SIMD batching to reduce latency requires novel programming paradigms and algorithms that do not have equivalents outside FHE. For example, matrix-vector-products can be expressed more efficiently if we encode each of the matrix diagonals into a SIMD vector [100], rather than row- or column-wise.

SIMD batching is, for those schemes that support it, potentially the most important optimization technique, as the large size of the vectors can lead to runtime improvements of many orders of magnitude. However, it is also one of the more complex techniques, requiring a deep understanding of both the application and the performance-tradeoffs of the FHE scheme in question. While some domains, such as machine learning, are inherently heavily vectorized and can therefore be automatically transformed into SIMD-friendly forms, this remains an open problem for more general applications.

### 3.1.5 *Ciphertext Maintenance*

Different schemes use a variety of solutions to manage the growth of the ciphertext noise during homomorphic computations. However, virtually all schemes feature some form of *ciphertext maintenance* operations. These are operations like relinearization, mod-switching/rescaling or bootstrapping that must be called explicitly by the developer in order to manage the noise growth optimally. For example, while one might be tempted to apply relinearization immediately after each multiplication, doing so is suboptimal. This is most obvious for the last multiplication in a computation: with no further multiplications following, the benefit of reducing future noise growth is lost. Similar issues appear when considering when to rescale in the CKKS scheme.

Bootstrapping is frequently not efficient when a leveled approach can be used. However, there are some applications for which it is the more suitable approach. In general, there is a continuum of choices between the minimal parameters that allow only a single operation before bootstrapping is needed and the (potentially infeasibly large) parameters required to execute the entire computation without bootstrapping.

One of the major advantages of the CGGI scheme is that it inherently relies on bootstrapping to realize each operation. Therefore, it removes the developer's burden to consider parameters and bootstrapping. However, it is worth noting that a leveled version of the scheme is, in fact, faster for certain applications, once again demonstrating a trade-off between simplicity and performance.

While a variety of tools have included automatic ciphertext maintenance [34, 66, 163], these were usually naive heuristics that did not improve performance. Developing efficient strategies is difficult because there are usually multiple degrees of freedom. For example, for the rescaling operations in CKKS one needs to consider both what scale to rescale to and where to insert the operations. Recently, however, there have been increasing efforts to automate this process [71].

## 3.2 SURVEY OF FHE FRAMEWORKS

### 3.2.1 *Methodology*

We split our analysis of the FHE tool space into two parts. First, we present an extensive survey of existing tools and highlight their features and characteristics. Second, we consider these tools in practice by experimentally evaluating them across a range of case study applications, contrasting usability, expressiveness, and performance. We combine our quantitative performance analysis with a qualitative assessment, describing the challenges of developing applications in the different tools.

The secure computation ecosystem includes many different types of tools. On the low-level side, there are math libraries that simplify building implementations of FHE schemes, e.g., by efficiently implementing techniques useful for general lattice cryptography. Then, there are FHE libraries that implement specific schemes and offer slightly higher-level APIs, e.g., `keygen`, `encode`, `encrypt`, `add`, `mult`. Finally, there are compilers that abstract aspects like parameter selection, encryption and decryption by offering a higher-level language that developers can use to specify their computation.

In our survey, we consider FHE libraries and compilers. While some of the underlying math libraries provide implementations of FHE schemes as examples [65], we consider only tools that natively offer an API for FHE operations. Throughout the last decade, there has been significant development in schemes and implementations, with some being discarded or replaced for security or efficiency reasons. We only consider tools based on schemes that are currently still considered viable candidates (i.e., BFV, BGV, CKKS, or GSW-based constructions) and consider only the latest version of each tool, including "spiritual successors" where they exist. We also consider only unique implementations, i.e., we do not list wrappers or ports of existing tools. FHE techniques are used internally in several MPC protocols, and there are a variety of tools that specifically target hybrid protocols combining FHE and MPC [111]. However, for this survey, we consider only tools that support using purely FHE, requiring no interaction during the computation itself.

We focus on three design aspects: *(i)* settings and configurations, e.g., which input languages or schemes a tool supports; *(ii)* features and optimizations, e.g., support for batching or automated parameter selection; *(iii)* accessibility, e.g., documentation and examples.

In our experimental evaluation, we consider a selection of tools in more detail. Through using the tools to implement different case study applications, we can provide an in-depth analysis of their usability and expressiveness in practice. In addition, our benchmarks allow us to study not only the overall performance of FHE for these applications but also the relative strengths of different tools compared to each other. We select three applications that represent different domains of FHE-based computation. Our first application is a risk score calculation that requires comparisons and, therefore, binary circuit emulation. Second, we consider a statistical $\chi^2$-test, in a formulation that simplifies it to polynomial functions over integers. Finally, we consider machine learning, specifically neural network inference, for a range of network architectures. We evaluate these applications across the different tools and report on usability, expressiveness, and performance.

### 3.2.2 *FHE Tooling Ecosystem*

Without tool support, realizing FHE-based computations by implementing the required mathematical operations directly or using an arbitrary-

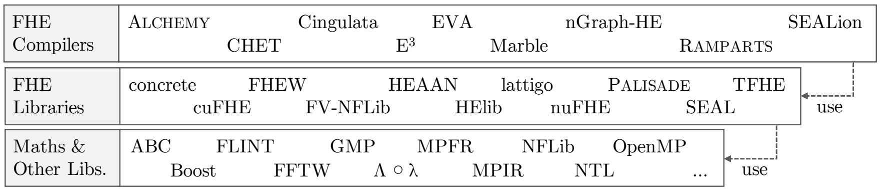


Figure 3.1: Overview of the FHE tool space.

precision arithmetic library is complex, requiring considerable expertise in both cryptography and high-performance numerical computation. Therefore, FHE libraries like the Simple Encrypted Arithmetic Library (SEAL) [156] or TFHE [51] implement the underlying cryptographic operations and expose a higher-level API. In addition to key generation, encryption, and decryption, these APIs also expose at least homomorphic addition and multiplication.

In practice, however, library APIs often include dozens of additional functionalities for ciphertext maintenance and manipulation. Since schemes vary in features, these APIs differ significantly not just in their implementation but also conceptually. Efforts are being made to standardize APIs for FHE schemes [2] and, simultaneously, there are first steps towards interoperability via wrappers around existing libraries [8]. However, achieving competitive performance frequently still requires working with libraries directly.

While FHE libraries make the process of writing FHE-based applications substantially more efficient, they still require significant expertise and understanding of the underlying scheme since they remain relatively low-level cryptographic libraries. Therefore, recent years have seen the development of higher-level tools, frequently known as FHE *compilers*, that aim to translate standard programs into FHE-based implementations. These tools focus on making FHE accessible to non-experts by improving usability and increasingly offering advanced optimizations previously accessible only to experts. Compilers generally rely on FHE libraries to realize the actual en-/decryption and homomorphic computation. FHE libraries, in turn, frequently employ existing libraries for fast numerical computations, parallelization, or other non-FHE-specific features. Figure 3.1 depicts different FHE tools and where they fit into this dependency hierarchy.

| Name | Input Lang. | Supported Schemes | | | Features | | Accessibility | | | Last Major Update |
|---|---|---|---|---|---|---|---|---|---|---|
| | | BFV | CKKS | GSW | Bootstrapp. | Levels | Code | Ex. | Doc. | |
| concrete ([55]) | Rust | ○ | ○ | ● | ● | ○ | ● | ◐ | ● | 11/2020 |
| FHEW ([74]) | C++ | ○ | ○ | ● | ● | ○ | ● | ○ | ○ | 05/2017 |
| FV-NFLlib ([67]) | C++ | ● | ○ | ○ | ○ | ● | ● | ◐ | ○ | 07/2016 |
| HEAAN ([47]) | C++ | ○ | ● | ○ | ● | ● | ● | ◐ | ○ | 09/2018 |
| HElib ([100]) | C++ | ● | ● | ○ | ● | ● | ● | ● | ● | 12/2020 |
| lattigo ([139]) | Go | ● | ● | ○ | ○ | ● | ● | ● | ○ | 12/2020 |
| PALISADE ([149]) | C++ | ● | ● | ● | ● | ● | ● | ● | ● | 04/2020 |
| SEAL ([156]) | C++, .NET | ● | ● | ○ | ○ | ● | ● | ● | ● | 08/2020 |
| TFHE ([51]) | C++ | ○ | ○ | ● | ● | ○ | ● | ● | ● | 05/2017 |
| cuFHE ([99]) | C++, Python | ○ | ○ | ● | ● | ○ | ● | ◐ | ○ | 08/2018 |
| nuFHE ([143]) | C++, Python | ○ | ○ | ● | ● | ○ | ● | ● | ● | 07/2019 |

Table 3.1: Overview of existing FHE CPU-targeting (top) and GPU-targeting (bottom) libraries. Note that similar schemes are summarized into categories, e.g., BFV/BGV as BFV and CGGI/TFHE/FHEW as GSW.

### 3.2.3 *FHE Libraries*

FHE libraries implement the underlying cryptographic operations of an FHE scheme and expose a higher-level API. They minimally provide key generation, encryption, decryption, homomorphic addition, and multiplication interfaces. In practice, however, library APIs often include dozens of additional functionalities for ciphertext maintenance and manipulation.

Using these libraries generally requires a deep understanding of the underlying scheme and its supported operations. While many libraries include powerful advanced features that can significantly improve performance, developers must employ them manually while ensuring correctness and efficiency.

In Table 3.1, we present an overview of FHE libraries and list supported languages, schemes, features, and accessibility aspects. We group schemes into families of related schemes for conciseness and consider support for bootstrapping and leveled-FHE. For accessibility, we consider whether an implementation (Code) is available, whether examples (Ex.) describe usage (●) or usage can be inferred from, e.g., tests (◐), and whether or not API documentation (Doc.) is available. Finally, we give a rough indication of age and activity by giving the date of the last release or major update.

Due to space constraints, we present only a small subset in more detail. We start by discussing HElib, SEAL, and Palisade, which appear to be the most active and widely supported libraries. We also discuss TFHE here since it is used by some of the compilers we evaluate. Finally, we discuss performance differences and briefly discuss the remaining libraries.

### 3.2.4 *HElib*

The Homomorphic Encryption Library (HElib), presented in 2013 by Halevi and Shoup, was the first FHE library [100]. The library is implemented in C++ and uses the NTL library [157] for the underlying mathematical operations. While it initially only implemented the BGV scheme, more recent releases of this library also support the CKKS scheme. The library offers leveled FHE operations and, for BFV, also supports bootstrapping [101]. The source code is available under the Apache License v2.0, and includes extensive examples. In addition to the standard documentation, several reports describing the design and algorithms of HElib [100, 101, 103] are available.

### 3.2.5 *PALISADE*

PALISADE, first released in 2014, is developed primarily by NJIT and Duality Technologies [149]. It is implemented in C++ and optionally uses the NTL library [157] to accelerate underlying mathematical operations. PALISADE supports a wide range of schemes, including BFV, BGV, CKKS, and CGGI. In addition, it supports multi-party extensions of certain schemes and other cryptographic primitives like proxy re-encryption and digital signatures. The library offers both leveled and bootstrapped operations, where supported by the scheme. PALISADE's source code is available under a BSD 2-clause license and includes examples and documentation.

### 3.2.6 *SEAL*

The Simple Encrypted Arithmetic Library (SEAL), first released in 2015, is developed by Microsoft Research [156]. It is implemented in C++, with an official wrapper for `.NET` languages (e.g., C#). SEAL is thread-safe and heavily multi-threaded itself. It implements the BFV and CKKS schemes, with a majority of the API being common to both. SEAL offers leveled FHE operations and does not implement bootstrapping for either scheme. Earlier versions of SEAL included automated parameter selection based on estimating the noise growth [148]. Since the estimated parameters were frequently non-competitive, this feature was removed. However, SEAL still ensures that the chosen parameters offer 128-bit security. The source code is available under an MIT license, is well documented, and includes a wide range of examples for both schemes. In addition, there are several demo applications (e.g., AsureRun [135]) that demonstrate more complex use cases.

### 3.2.7 *TFHE*

The Fast Fully Homomorphic Encryption Library over the Torus (TFHE) was proposed in 2016 by Chillotti et al. [51] and can be considered the successor of the FHEW library [74]. It is implemented in C++ and supports a variety of different libraries for underlying FFT operations. TFHE is based on the CGGI scheme and offers gate-by-gate bootstrapping with significantly reduced bootstrapping times, resulting in times of less than 0.1 s compared to 6 min for bootstrapping in the HElib library. TFHE implements a variety of logic gates like OR, NOR, MUX that are generally implemented more efficiently than naive constructions from XOR and AND would be. However, the library provides no assistance with building more complex logic circuits like efficient comparators and adders. TFHE's source code is available under the Apache License v2.0 and includes examples and documentation.

### 3.2.8 *Other Libraries*

In addition to the libraries we discussed above, we considered a large variety of other libraries [47, 55, 67, 74, 138]. We also conducted a series of microbenchmarks to compare how different implementations of the same scheme perform. However, due to space considerations we refer to our accompanying online repository for details. Finally, GPU-based libraries like cuFHE [99] and nuFHE [143] can offer significant speedups, improving the already fast TFHE bootstrapping times by around two orders of magnitude. However, as GPUs remain considerably more expensive and less common in enterprise datacenters, these speedups must be considered in context.

| Name | Input | Schemes | | | Ptxt. | Features & Optimizations | | | | Accessbility | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Lang. | BFV | CKKS | GSW | Space | SIMD | Params. | Ctxt. Mnt. | $\times$ Depth | Code | Ex. | Doc. |
| ALCHEMY ([66]) | Haskell | ● | ○ | ○ | Arithm. | ◐ | ◐ | ◐ | ○ | ● | ◐ | ○ |
| Cingulata ([34]) | C++ | ● | ○ | ◐ | Binary | ○ | ● | ◐ | ● | ● | ● | ○ |
| $E^3$ ([49]) | C++ | ● | ○ | ● | Both | ◐ | ○ | ◐ | ○ | ● | ● | ● |
| EVA ([71]) | Python | ○ | ● | ○ | Arithm. | ● | ● | ● | ○ | ● | ● | ○ |
| Marble ([163]) | C++ | ● | ○ | ○ | Both | ◐ | ◐ | ◐ | ○ | ● | ● | ○ |
| RAMPARTS ([6]) | Julia | ● | ○ | ○ | Arithm. | ○ | ● | ◐ | ● | ○ | – | – |
| CHET ([72]) | C++ | ○ | ● | ○ | Arithm. | ◐ | ● | ◐ | ○ | ○ | – | – |
| nGraph-HE ([14]) | Python | ● | ● | ○ | Arithm. | ● | ○ | ◐ | ● | ● | ● | ● |
| SEALion ([160]) | Python | ● | ○ | ○ | Arithm. | ◐ | ● | ◐ | ○ | ○ | ● | ○ |

Table 3.2: Overview of existing general-purpose FHE compilers (top) and those specializing on machine learning (bottom). Note that similar schemes are summarized into families, e.g., BFV/BGV as BFV and CGGI/TFHE/FHEW as GSW.

| | FHEW | HEAAN | HElib | PALISADE | SEAL | TFHE |
|---|---|---|---|---|---|---|
| Alchemy | ○ | ○ | ○ | ○ | ○ | ○ |
| Cingulata | ○ | ○ | ○ | ○ | ○ | ● |
| E$^3$ | ● | ○ | ● | ○ | ● | ● |
| EVA | ○ | ○ | ○ | ○ | ● | ○ |
| Marble | ○ | ○ | ● | ○ | ● | ○ |
| Ramparts | ○ | ○ | ○ | ● | ○ | ○ |
| CHET | ○ | ● | ○ | ○ | ● | ○ |
| nGraph-HE | ○ | ○ | ○ | ○ | ● | ○ |
| SEALion | ○ | ○ | ○ | ○ | ● | ○ |

Table 3.3: Use of existing FHE libraries by FHE compilers. Note that Alchemy implements BGV internally using the $\Lambda \circ \lambda$ lattice cryptography library, and Cingulata also includes a custom implementation of BFV.

### 3.2.9 *FHE Compilers*

This section provides an overview of existing FHE compilers, i.e., tools that provide a high-level abstraction to develop FHE-based applications, so developers do not have to deal directly with homomorphic operations on ciphertexts. These tools generally manage key setup, encryption, decryption, and ciphertexts maintenance operations in the background. The term *compiler* is used loosely in the context of FHE, as some function more like interpreters or libraries to link against.

In Table 3.2, we provide an overview of the various FHE compilers and their properties. FHE compilers can roughly be divided into generic tools for general purpose use and tools that target specific applications. In the latter category, we see compilers targeted at building Machine Learning (ML) applications. In addition to supported schemes, which we again group for conciseness, we also consider the plaintext spaces supported by the tool. Even when the underlying scheme and implementation support different plaintext spaces, compilers generally only target binary or arithmetic plaintext spaces.

We consider a wide range of features and generally differentiate between three states indicating full support (●), partial support (◐), or no support (○). For SIMD-Batching (SIMD), we differentiate between tools that merely enable batching and those that actively assist in working with vectorized data. Similarly, we differentiate between manual, partially assisted, and fully automated parameter selection (Params.). While all tools include some form of automated ciphertext maintenance operations (Ctxt. Mnt.), we segment tools into those that use naive heuristics and those using more advanced strategies. Additionally, we note whether or not tools try to reduce the multiplicative depth ($\times$ Depth) of the circuits they generate. For accessibility, we consider the same metrics as for libraries, i.e., whether an implementation (Code) is available, whether examples (Ex.) describe usage, and whether or not API documentation (Doc.) is available. Similarly, we again give a rough indication of age and activity by giving the date of the last release or major update. Where no source code is available to us, we have to omit these metrics ("–"). Finally, Table 3.3 associates compilers with the libraries they use. Here we can see SEAL being targeted by a significantly larger number of compilers than any other library. In the following, we introduce each compiler in more detail.

### 3.2.10 Alchemy

A Language and Compiler for Homomorphic Encryption Made easY (Alchemy) was proposed by Crockett et al. in 2017 [64]. Input programs are specified in a special Domain-Specific Language (DSL) implemented in Haskell and executed as arithmetic circuits using a custom BGV implementation using the $\Lambda \circ \lambda$ lattice crypto library [65]. While it supports SIMD batching, it does not offer an encoding/decoding API, making it difficult to use. Alchemy automatically selects suitable parameters by statically tracking the upper bound of the ciphertext error but requires user-supplied modulus candidates. However, this approach, based on type-level arithmetic, leads to excessively long compilation times and makes Alchemy impractical for complex programs. While open-source, the minimal examples are insufficient to allow non-Haskell-experts to use the library, and it is therefore excluded from our experimental evaluation.

### 3.2.11 *Cingulata*

Cingulata (previously *Armadillo*) was proposed in 2015 by Carpov et al. [34]. The compiler takes C++ code as input and generates a corresponding Boolean circuit. Cingulata implements the BFV scheme directly, using the Flint and Sage libraries for operations on polynomials. We refer to this built-in BFV implementation as CinguBFV. Cingulata also supports the CGGI scheme via the TFHE library, but advanced optimizations are not supported in this mode. Recent versions include CinguParam [105], which automatically determines parameters for BFV. Cingulata inserts relinearization operations naively but tries to reduce the circuit's multiplicative depth using the circuit optimization tool ABC [136], which was originally designed for hardware synthesis. However, follow-up work has introduced novel FHE-specific depth-reduction heuristics [7, 33, 122]. Cingulata's source code is available under the CeCILL license and includes many examples.

### 3.2.12 *Encrypt-Everything-Everywhere*

The Encrypt-Everything-Everywhere ($E^3$) framework was proposed by Chielle et al. [49] in 2018. $E^3$ uses C++ as its input language and supports both arithmetic and boolean circuits in BFV, BGV, and CGGI. $E^3$ supports SIMD operations but does not expose rotation operations, severely limiting the expressiveness. Users must provide parameters as part of the configuration, and ciphertext maintenance operations are inserted naively. It uses the Synopsys Design Compiler, a proprietary tool for hardware design, to try to reduce the circuit's multiplicative depth. Internally, it supports a variety of libraries, including TFHE, FHEW, HElib, and SEAL. $E^3$'s source code is available online and includes both examples and documentation.

### 3.2.13 *EVA & CHET*

The Encrypted Vector Arithmetics Language and Compiler (EVA) was presented by Dathathri et al. [71] in 2019. It introduces a novel input language explicitly designed for vector arithmetic and targets arithmetic circuits in CKKS using the SEAL library. It is inherently batched and focuses on automating parameter selection and ciphertext maintenance. The program is converted into a term graph, and during multiple passes, graph rewriting rules transform it, e.g., by inserting relinearization and rescaling operations at the optimal locations. However, EVA does not consider depth-reducing transformations. While EVA can be used for any (vectorized) application, the focus is primarily on neural network inference. Towards this end, EVA includes and subsumes prior work in the form of the Compiler and Runtime for Homomorphic Evaluation of Tensor Programs (CHET) [72], which focuses on optimizing matrix-vector operations. EVA and its examples are available under the MIT license. CHET, however, is not.

### 3.2.14 *Marble*

Marble, presented by Viand et al. in 2018 [163] offers a high-level interface for FHE in C++ by overloading built-in operators. For arithmetic circuits, it targets BFV via the SEAL library, and for binary circuits, BGV as implemented in the HElib library. Marble exposes a batched version of the API, allowing relatively efficient implementation, but it requires that the developer provides a suitably vectorized program. However, Marble provides only rudimentary parameter selection, inserts ciphertext maintenance operations naively, and does not apply any program optimizations. While a version of Marble is available online, the available code supports only binary circuits. Since Marble targets an outdated version of HElib and focuses on usability over optimizations, we do not include it in our experimental evaluation.

### 3.2.15 *Ramparts*

Ramparts was proposed in 2019 by Archer et al. [6]. It uses Julia, a language for interactive scientific computing, as its input language, and targets arithmetic circuits in BFV using the PALISADE library. Ramparts does not support batching, but includes noise-growth-estimation based parameter selection. Ciphertext maintenance operations are inserted naively, but a symbolic simulator simplifies the circuit by applying sub-expression elimination, constant folding, and partial evaluation (e.g., loop unrolling, function inlining). Ramparts is not publicly available. Therefore, we were unable to include it in our experimental evaluation. However, Rampart's evaluation compares it against Cingulata and a baseline using PALISADE directly. The evaluation showed significant performance benefits compared to Cingulata; however, in exchange, Ramparts is limited to programs that can be expressed as polynomial functions and the symbolic evaluation approach significantly increases compilation times.

### 3.2.16 *nGraph-HE*

The nGraph-HE framework, proposed by Boemer et al. [16] in 2019, is based on Intel's nGraph ML compiler [69] and translates standard TensorFlow computations into arithmetic circuits in BFV or CKKS using the SEAL library. It enables inference on pre-trained models over encrypted inputs, applying FHE-specific optimizations (e.g., constant folding, SIMD-packing, and graph-level optimizations such as lazy rescaling and depth-aware encoding), and run-time optimizations (e.g., bypassing special plaintext values). However, it inserts rescaling operations naively and requires the user to define the parameters. In subsequent work [14], nGraph-HE was extended to support non-polynomial activation functions. However, these are computed in an interactive protocol with the client, which introduces significant latency and is out of scope for our study. nGraph-HE is available under the Apache License v2.0 and includes examples and documentation.

### 3.2.17 *SEALion*

The framework SEALion, proposed by Van Elsloo [160] in 2019, exposes a custom Python API for specifying ML models, which are trained using TensorFlow. SEALion then enables inference over encrypted data using arithmetic circuits in BFV using the SEAL library. SEALion supports batching to increase inference throughput by performing inference over multiple data simultaneously but does not consider non-trivial batching optimizations. Further, it features automatic parameter selection using a heuristic search algorithm to find an optimal parameter set. However, it inserts ciphertext maintenance operations naively and does not consider depth-reducing optimizations. SEALion is not currently publicly available; however, the authors shared their implementation with us, and the code includes well-commented examples.

## 3.3 EVALUATION

In the following, we present our experimental evaluation, where we investigate FHE compilers in more detail. We use these tools to implement and benchmark selected case study applications. This allows us to provide an in-depth analysis of their usability and expressiveness in practice, and to compare the performance characteristics of current FHE compilers.

Since there are no standardized benchmarks for FHE, comparing performance across tools is generally difficult without implementing a task in a variety of tools. Motivated by that, we selected three applications that represent different domains of FHE-based computation. Each is designed to showcase complex issues arising when working with FHE, yet also remain simple and easy to reproduce across tools. First, we present a risk score calculation that requires comparisons and, therefore, binary circuit emulation. While simple, this represents a class of heavily branched programs that is common in traditional programming but hard to express in FHE. Second, we consider a statistical $\chi^2$-test, in a formulation that simplifies it to polynomial functions over integers. This represents a variety of interesting analysis methods that are ill-suited to FHE by default but can be reformulated or approximated to allow efficient implementations. We focus only on the core computation, however in practical deployments, this would probably be preceded by a homomorphic aggregation over user data. Finally, we consider machine learning, specifically neural network inference for a range of network architectures. We evaluate a range of increasingly complex models and show how commonly used architectures are adapted for FHE.

In our evaluation, we consider three dimensions: usability, expressiveness, and performance. We start by describing each application in detail, then report on the process of implementing these applications in the different tools, highlighting strengths and challenges. Where required, we describe adjustments made to the applications due to limits in expressiveness. Finally, we present our benchmarking results and highlight the impact of specific techniques or optimizations.

### 3.3.1 *Applications*

**Cardio.** The cardio risk factor assessment (cardio) application computes a score representing a patient's risk of cardiac disease. The application takes metrics such as age, gender, weight, drinking habits and smoking behavior where some are integer-valued and others boolean flags as input. As illustrated in Listing 3.1, the computation consists of a series of simple rules over the inputs that use comparisons and boolean operators. The algorithm is derived from an implementation in [32]. Due to its reliance on comparison operations, the program requires emulation using binary circuits.

```
+1  if man && age > 50 years
+1  if woman && age > 60 years
+1  if smoking
+1  if diabetic
+1  if high blood pressure
+1  if HDL cholesterol < 40
+1  if weight > height-90
+1  if daily physical activity < 30
+1  if man && alcohol cons. > 3 glasses/day
+1  if !man && alcohol cons. > 2 glasses/day
```

Listing 3.1: The Cardio application.

We encode the inputs as 8 bit numbers and encrypt them at the client-side. The server receives the encrypted data, computes the risk score, and returns the encrypted score back to the client. While the inputs are obviously sensitive information, the cardio risk assessment algorithm is public and could easily be calculated client-side. However, it is easy to imagine other applications where a service provider might not want to share the algorithm with a client. For example, similar algorithms are still widely used for risk assessment or fraud detection, and knowledge of the criteria considered makes it easier to circumvent these checks. For simplicity, we omit the noise flooding required to provide (practical) circuit privacy in our example.

**Chi-Squared Test.** $\chi^2$ or chi-squared tests are common statistical tests. For our application, we specifically consider Pearson's Goodness-of-Fit test as it can be used to test for deviation from the Hardy-Weinberg equilibrium in Genome-Wide Association Studies (GWAS).

We split the computation into a polynomial part on the server and a final set of divisions on the client, as proposed by Lauter et al. [119]. First, the server receives the encrypted genotype counts $N_0, N_1, N_2$, then it computes $\alpha = (4N_0N_2 - N_1^2)^2, \beta_1 = 2(2N_0 + N_1)^2, \beta_2 = (2N_0 + N_1)(2N_2 + N_1), \beta_3 = 2(2N_2 + N_1)^2$ and returns the encrypted results to the client. Decrypting these, the client can compute the test statistic as $X^2 = \frac{\alpha}{2N}(\frac{1}{\beta_1} + \frac{1}{\beta_2}\frac{1}{\beta_3})$. This transformation introduces some slight leakage of intermediate values but in return enables an application that would otherwise be infeasible. A more realistic deployment scenario would most likely first see the server calculate the genotype counts over an encrypted genomic database. While this application is comparatively simple, it is nevertheless practically relevant as seen by its application to genomic studies. Additionally, its simplicity allows us to focus more clearly on the overheads introduced by each tool.

**NN Inference.** The neural-network inference application demonstrates FHE's capabilities for privacy-preserving machine learning. Specifically, we consider inference (or prediction) on a simple image recognition task, i.e., recognizing handwritten digits from the MNIST dataset [121]. MNIST is a common benchmark in machine learning applications and can be solved effectively by many techniques. In MNIST, individual inputs are $28 \times 28$ pixel images containing a single handwritten digit. First, the network is trained over a large number of plaintext images. Later, a client submits an encrypted input and the model owner returns the encrypted prediction. This guarantees the privacy of the input and gives strong practical protections for the privacy of the model. When only the model parameters, but not the general architecture, need to be protected, formal circuit privacy is not required.

### 3.3.2 *Implementation Considerations*

In this section, we explain our selection of tools for each application and briefly discuss implementation challenges we faced. A more detailed documentation of our implementations and design choices is available in our online repository[1].

**Cardio.** The cardio risk factor assessment requires computing several comparisons between integers, which are hard to approximate polynomially and therefore require binary circuit emulation. As a baseline, we implemented the programs manually in SEAL and TFHE. Since EVA targets CKKS, which is less well suited to binary emulation, we do not consider it

1 `https://github.com/MarbleHE/SoK.`

here. The Cingulata and $E^3$ compilers, on the other hand, support binary plaintext spaces natively.

In SEAL and TFHE, we needed to manually implement binary adders and comparators. This is significantly easier in TFHE, where multiplicative depth is not a concern and a simple ripple-carry-adder is sufficient. Therefore, our optimized TFHE implementation merely improves the final summation of risk factors by using a tree of adders. While our naive SEAL implementation also uses a ripple-carry-adh.kjulder, we also implemented an optimized version where we implemented a Sklansky-adder, which trades off additional operations for lower depth. In the optimized version, we also made heavy use of in-place and plaintext-ciphertext versions of the homomorphic operations, simplified expressions as much as possible, and manually determined optimal parameters. Finally, we implemented an optimized batched variant, which required significant changes to the computation structure, i.e., transforming all ten conditions into the form `a && b < c` by introducing dummy values and operations.

Cingulata makes the implementation significantly more straight-forward as it contains built-in circuits for common operations such as addition, multiplication, and comparisons. Therefore, the program is virtually identical to its plaintext counterpart. However, the compilation process is complex, and the interactions between the compiler and runtime system are not well documented. This made it hard to integrate the different mult-depth reduction techniques available, and it required significant amounts of trial-and-error to determine, e.g., how Cingulata differentiates between secret and plaintext inputs in the circuits it generates.

$E^3$ offers a similar and even arguably more powerful API than Cingulata. For example, it supports both binary and arithmetic plaintext spaces and can switch ciphertexts between them. In a similar vein, very few changes were needed to re-target our SEAL (BFV) implementation to TFHE (CGGI). However, an initial lack of documentation and very long compile times made developing and debugging applications difficult. While $E^3$ features some support for batching, this is quite limited. Specifically, it does not include rotation operations that are essential to fully express the program's batched version. Therefore, the $E^3$ batched version remains somewhat incomplete.

**Chi-Squared Test.** The Chi-Squared test, at least as re-formulated in our application, uses only addition and multiplication over integers, making it ideally suited for integer-based FHE schemes. Nevertheless, we also consider implementations targeting binary emulation for comparison. We manually implemented the application in SEAL, targeting the BFV schemes and an integer plaintext space. In our optimized version, we manually select optimal parameters, use in-place operations where possible and reuse common sub-expressions. Our manual implementations in SEAL closely match the mathematical description as all operations are native operations. Nevertheless, both the naive and the optimized implementation required over 100 lines of code. Our TFHE-based manual implementations additionally required implementing a binary adder and multiplier to support the computation, resulting in several hundred lines of code. EVA, in contrast, allowed us to easily express the same computation in around a dozen lines of code. While the EVA implementation targets CKKS, the precision is sufficient to ensure that, when rounding back to integers, the result perfectly matches the other BFV/integer-based implementations. While Cingulata supports the BFV scheme, it only supports binary plaintext spaces. Therefore, it must also emulate integer multiplications using binary circuits. However, since it hides the complexity of generating efficient circuits from the user, this matters only for performance, not for usability. Both Cingulata and $E^3$ can target integer-based BFV and binary CGGI with minimal changes required. Note that batching this application would be trivial but only impacts throughput, not latency, and is therefore omitted.

**NN-Inference.** The MNIST problem is comparatively easy to solve, with simple approaches easily achieving more than 90% accuracy and even small neural networks achieving around 95% accuracy. State-of-the-art networks achieve up to 99.5% test accuracy. However, increasing accuracy quickly requires exponentially more complex models. In our evaluation, we used three different model architectures of increasing complexity. First, we used a simple Multi-Layer Perceptron (MLP) as a baseline, i.e., two fully connected layers with a non-linear activation. Next, we consider a more complicated Convolutional Neural Network (CNN), specifically the Cryptonets architecture [95] designed specifically for FHE, which consists of 5 layers and two activations. Finally, we also evaluated a LeNet-5-like [120] network, which is a significantly more complex design and more representative of networks used to solve challenging tasks in practice. This network consists of 7 layers and three activations. We use a technique from [72] and learn a degree-two polynomial approximation of the ReLU activation function during training.

SEALion and nGraph-HE focus exclusively on machine learning inference, directly using TensorFlow programs or TensorFlow-like programs as their inputs. While SEALion can currently only express a simple MLP network, nGraph-HE seems to support the full TensorFlow feature set. Both make FHE-based development nearly as easy as working with standard TensorFlow. While EVA does not directly support machine learning tasks, the CHET tool can be re-targeted to EVA, and we consider an EVA program for a LeNet-5 model generated by CHET, in addition to a manually implemented MLP. We complemented the comparison between the tools with a baseline implementation of an MLP in SEAL, using the CKKS scheme and manually implementing matrix-vector-product optimizations from [111], which required significant engineering effort.

### 3.3.3 *Effects of Optimizations*

This section presents the results of our benchmarks, with a particular focus on the effect that automation and optimization have on runtime. All benchmarks run on an AWS instance (m5n.xlarge), equipped with 4 vCPUs and 16 GB RAM. The reported results are mean values computed over 10 test runs.

**Cardio.** In Figure 3.2, we report the run time for the cardio risk factor assessment application in different setups. We see a large span of results, between less than 5 seconds for the manual optimized implementation and over three minutes for the slowest tool-generated implementations. E$^3$ seems to introduce significant overheads, even when compared to naive implementations targeting the same library. Cingulata's BFV implementation (CinguBFV) seems considerably slower than SEAL's, but we can still observe the effect of the different depth-reduction approaches, with multi-start (E) cutting computation time in half. Comparing our manual implementations, we see both of our TFHE implementations outperforming the naive and (non-batched) optimized SEAL implementation as expected. Cingulata's TFHE implementation actually further outperforms our manually optimized TFHE implementation, even when our manual program uses fewer gates. This speedup might be due to better memory management or due to slightly different TFHE environments. However, by far the best performance is achieved when using batching in SEAL, even though this application is inherently binary-based and ten conditions are a relatively small number to batch in the context of FHE.

**Chi-Squared Test.** In Figure 3.3, we present the runtimes for the chi-squared test application, using a logarithmic scale due to the large range of values. We contrast manually- and tool-generated implementations targeting SEAL and TFHE and compare this against Cingulata's implementation targeting the built-in BFV implementation. The manually optimized SEAL implementation and EVA-generated implementation outperform the others by a large margin, requiring less than a second. With 16.46 s, a slowdown of more than 10×, the E3 program targeting SEAL is significantly slower, but the overhead compared to the naive solution is negligible. Meanwhile, Cingulata targeting CinguBFV suffers from both using binary emulation unnecessarily and a generally slower BFV implementation. Since the program already has minimal depth, we omit a discussion of the different depth-reduction heuristics here. Similarly, our TFHE optimizations seem to have no positive effect on this simple program, while Cingulata is again faster per-gate in TFHE, possibly due to configuration differences. Finally, we note that the TFHE implementation generated by E3 is around 9× slower than native implementations, which are already non-competitive compared to integer-based solutions. In combination with the cardio benchmarking results, this indicates that $E^3$ generates binary adder/multiplier circuits inefficiently when using binary emulation.

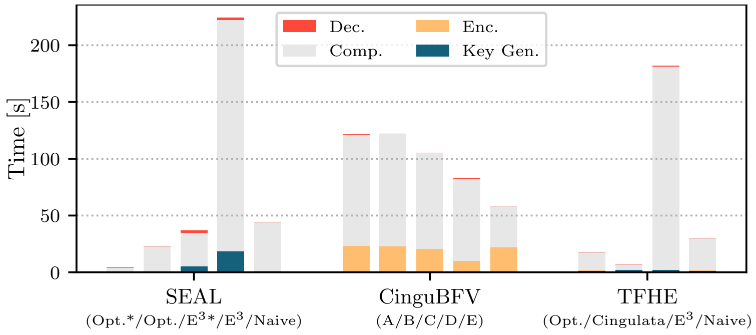


Figure 3.2: Runtime of the cardio benchmark. We group compiler generated and manually optimized and naive programs by the FHE implementation they target. For CinguBFV, we consider circuits using different depth-optimization approaches (A: baseline, B: ABC, C: Lobster, D: Cingulata, E: Multi-Start). * indicates batching was used.

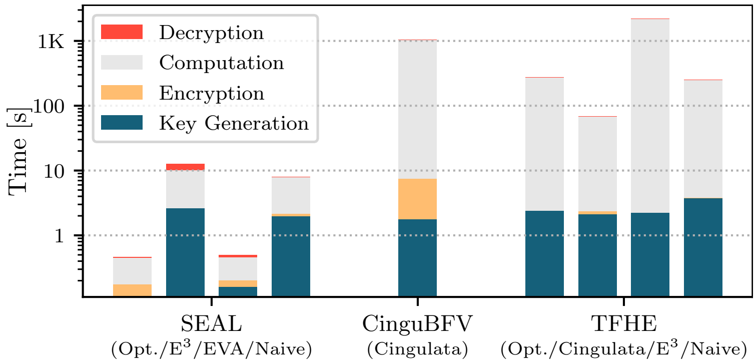


Figure 3.3: Runtime of the chi-squared test benchmark using a logarithmic scale. We group compiler generated, optimized and naive programs by the FHE implementation they target.

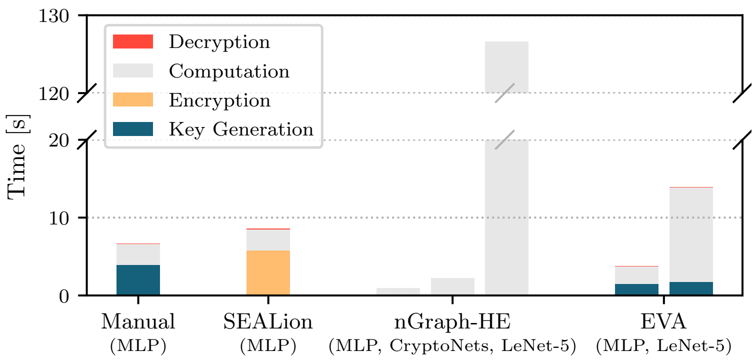


Figure 3.4: Runtime of the neural network inference benchmark, i.e., recognizing handwritten digits from the MNIST dataset. All implementations target SEAL. We implement a simple multi-layer-perceptron (MLP). For nGraph-HE and EVA, we also consider more complex models (CryptoNets, LeNet-5).

**NN Inference.** We present the evaluation results for the neural-network inference task in Figure 3.4, reporting latency, i.e., the time to run encrypted prediction on a single image. Note that SEALion and nGraph-HE use SIMD-style batching to achieve higher throughput at the same latency. For nGraph-HE, it was not possible to provide individual sub-timings, as key-generation, encryption, and decryption are invisible to the application code. We first compare our manual implementation of an MLP both directly in SEAL and using EVA against the same network architecture implemented in SEALion and nGraph-HE, which offer much higher-level interfaces. All models achieved around 95% accuracy, nearly identical to their plaintext equivalents. Note that for SEALion, the overall runtime is artificially inflated because the tool encrypts the input against a range of possible parameter sets instead of only the targeted one. Taking this into account, we can see that despite us implementing several optimization techniques from the literature, the higher-level tools clearly outperform the manual implementation. In the case of SEALion, this appears to be due to automatic sparsification, which reduces the network's size. Finally, we explored more complex models using nGraph-HE and EVA, using CHET-generated programs for the latter. The Cryptonets CNN architecture significantly increases accuracy (to 98%) at a minimal increase in computation cost. However, achieving state-of-the-art network performance (99+%) requires a considerably more complex LeNet-5-like network, which takes around 13 seconds to run using EVA and more than two minutes using nGraph-HE.

## 3.4 DISCUSSION

In this section, we discuss some key questions in the space of FHE and FHE tools:

### 3.4.1 *What applications can be developed using FHE today?*

While FHE can be practical for a wide variety of applications, there remain many applications that are not yet feasible using FHE. Applications that make sense for FHE generally feature a client-server scenario where both the input data and the algorithm need to be kept private. In addition, there are practical limits to the complexity of the applications that can be outsourced. As a very rough heuristic, computations that take more than a few hundred milliseconds without FHE are unlikely to be practical once translated into FHE as of today. However, this very much depends on the application scenario. Generally, online computations where immediate feedback is expected are more challenging. For example, face recognition applications at an airport might tolerate a few seconds of delay at most. On the other hand, offline tasks like computing statistics over the results of a year-long medical study can be considered practical even if taking considerable time.

For non-expert users, the range of applications that can be realized in practice also depends significantly on the available tools. Using libraries like SEAL, PALISADE, or HElib makes it easy to implement simple computations that can be expressed as low-degree polynomials (e.g., the modified $\chi^2$ test), and tools like nGraph-HE enable novice users to easily implement linear ML models, simple statistics, and neural network inference. For more experienced users, this question becomes increasingly difficult to address in general terms. The implementation challenges we describe for our case studies show that application complexity and FHE implementation complexity do not necessarily correlate. Finally, some applications require modifications or extensions of the underlying cryptographic primitives. These include computations that require switching between different schemes or between FHE and MPC homomorphically. Many applications can already be solved practically using these or other novel programming paradigms. Frequently, success in implementing an efficient FHE-based solution for an application depends less on the performance of the underlying FHE tools but on how the application is translated. Exploiting the advantage of SIMD-batching, e.g., using EVA, requires designing heavily vectorized programs for a setting with significantly more restrictions than, e.g., AVX vector instructions. In addition, many applications become feasible only after slight modifications,e.g., using polynomial approximations or rewriting expressions so that hard-to-compute operations (e.g., square roots) are delayed until the end to allow them to be performed client-side after decryption. By presenting these paradigms more clearly and targeting an audience beyond the crypto community, the set of applications that developers can expect to realize successfully using FHE will expand significantly.

### 3.4.2 *When to use which of the FHE tools?*

Given the choice of different tools that each present slightly different features and strengths, selecting the appropriate tool for a given application can be non-trivial. However, not all tools that *can* implement a solution are necessarily suitable choices, as demonstrated in our evaluation. Current tools generally excel at specific workloads or application domains, and here we try to provide some recommendations for tools to consider for common application scenarios.

For generic applications that compute non-polynomial functions or require binary emulation, there are multiple options with different tradeoffs. If working primarily with integers, the programmable bootstrapping offered by the *concrete library* is an obvious choice. While compilers like Cingulata (CinguBFV) or $E^3$ are easier to work with, the performance overhead they introduce might be unacceptable for many applications. For applications requiring a true binary plaintext space, *Cingulata (TFHE)* is most likely the easiest approach.

For applications that compute (polynomial) statistics over large amounts of data, we recommend the *EVA compiler* targeting CKKS for applications requiring approximate numbers. If working with integers only, we recommend working directly with the *SEAL library* targeting BFV, since BFV is less complex to work with and current compilers targeting it introduce significant slowdowns. The batching offered by these schemes can be a natural fit when computing aggregate statistics or retrieving information from encrypted databases.

For applications that involve or use machine learning inference, the recommended approach depends on the complexity of the used ML model. Where training a model with polynomial activation functions produces sufficient accuracy, we recommend using the *nGraph-HE compiler* targeting the CKKS scheme. nGraph-HE supports virtually all TensorFlow features, including the Keras model definition API, making it trivial to port existing models. In addition, nGraph-HE offers excellent performance that can easily outperform even a fairly involved manual implementation. Where deeper/recursive networks or standard activation functions (e.g., ReLU) are required to achieve the desired accuracy, the programmable bootstrapping functionality offered by the *concrete library* makes it the most suitable choice. However, this will require significantly more engineering effort as there are currently no higher-level compilers targeting concrete.

### 3.4.3 *Where should FHE tools go from here?*

Both FHE compilers and libraries remain complex to use, and there are obvious low-hanging fruits in terms of usability that include better documentation and more extensive examples. In addition, there is a general lack of interoperability, not just technically but also conceptually. For example, even libraries implementing the same scheme can offer surprisingly different APIs. The ongoing standardization efforts are trying to create a unified view of the most popular schemes, including standardized APIs for the most common operations. However, this does not address the various extension of the API, e.g., optimizations for squaring rather than multiplying or performing operations in-place. This would be solved ideally by introducing a common intermediate representation language that compilers can target and libraries can implement.

The existing tools have successfully reduced the complexity of working with complex FHE schemes. There is a large choice of libraries providing secure and efficient implementations of current schemes. In addition, compilers have emerged that make it significantly easier to realize computations efficiently, e.g., by automatically choosing parameters or inserting ciphertext maintenance operations. However, this still leaves the user with the significant challenge of translating an application into an appropriate FHE computation in the first place. For example, tools could automatically vectorize iteratively written programs or offer suggestions on aspects of the computation that would be beneficial to extract to the client-side.

Finally, it is worth noting that we have considered only FHE tools in our analysis and discussion. However, real-world problems are frequently complex and require a combination of techniques, including FHE, Multi-Party Computation (MPC), and Zero-Knowledge Proofs (ZKP). In the long term, the secure computation community could gain tremendously by considering these problems more holistically and building tools that support a wider range of techniques.

# 4 END-TO-END COMPILATION FOR FHE

Fully Homomorphic Encryption is a nascent field and still actively evolving, with ongoing research on the cryptography, software implementations, and, increasingly, on hardware accelerators. As a result, tools must be designed to accommodate and adapt to to this fast moving field. Existing compilers (cf. Section 4.4), however, are mostly rigid, monolithic tools with a narrow focus on individual sub-optimizations. Whereas experts usually transform applications in ways that accelerate them by orders of magnitude, existing tools have mostly focused on smaller-scale optimizations that result in small constant-factor speedups. While these represent important contributions, they are insufficient to make the kind of qualitative performance difference that is necessary to achieve practical FHE. In order to overcome these limitations, we need to fundamentally *rethink the architecture of FHE compilers* and *develop novel optimizations* that abstract away the complexity FHE and address the limitations of existing tools. In this chapter, we present HECO, a new multi-stage optimizing FHE compiler. Our architecture provides, for the first time, a true end-to-end toolchain for FHE development. In addition, we propose novel transformations and optimizations that map imperative programs to the unique programming model of FHE.

## 4.1 END-TO-END FHE COMPILER DESIGN

This section first introduces relevant aspects of the FHE programming paradigm, then provides a system overview of HECO before presenting its core components, beginning with the overall framework design, followed by a discussion of our compiler architecture, and finally, gives an overview of the transformations and optimizations that constitute the compilation pipeline.

### 4.1.1 *FHE Programming Paradigm*

FHE imposes a variety of restrictions on developing programs: some derive from the definition of FHE and its security guarantees, while others result from scheme restrictions and cost models. For example, FHE's security guarantees make it necessarily data-independent, hence preventing branching based on secret inputs. While some forms of branching can be *emulated*, all branches must be evaluated, resulting in a potentially significant degradation of performance. In addition, FHE schemes only offer a limited set of data types and operations, with addition and multiplication as basic operations. Applied over binary plaintext spaces ($\mathbb{Z}_2$), this technically enables arbitrary computation. However, the best performance is usually achieved with larger plaintext spaces (e.g., $\mathbb{Z}_t$ for $t \gg 2$). In this setting, computations are equivalent to arithmetic circuits, which can only compute polynomial functions. Non-polynomial functions can be approximated, but this is typically prohibitively inefficient. While recent works have explored homomorphic conversions between binary and arithmetic settings [18, 129] and introduced *programmable bootstrapping* to approximate non-polynomial functions [56], these approaches are not yet practical enough for widespread adoption.

As a result, developing FHE applications requires fundamentally rethinking how programs are written. Generally, developers need to rethink their approach, e.g., using branch-free algorithms well-suited to low-degree polynomial approximations. In addition, the large size of FHE ciphertexts, which is required for security reasons, is a significant source of both communication and computation overhead. However, it also presents an opportunity, as many[1] schemes support *batching*, which allows encrypting many values into the same ciphertext. This reduces ciphertext expansion and enables element-wise operations in a Single Instruction, Multiple Data (SIMD) fashion. Data movement in FHE is incredibly restricted, affording no efficient ways to permute the batched data after encryption, with the exception of cyclical rotations. As a result, efficient FHE algorithms are usually dras-

1 Specifically, schemes from the Ring-LWE family that B/FV, BGV and CKKS belong to.

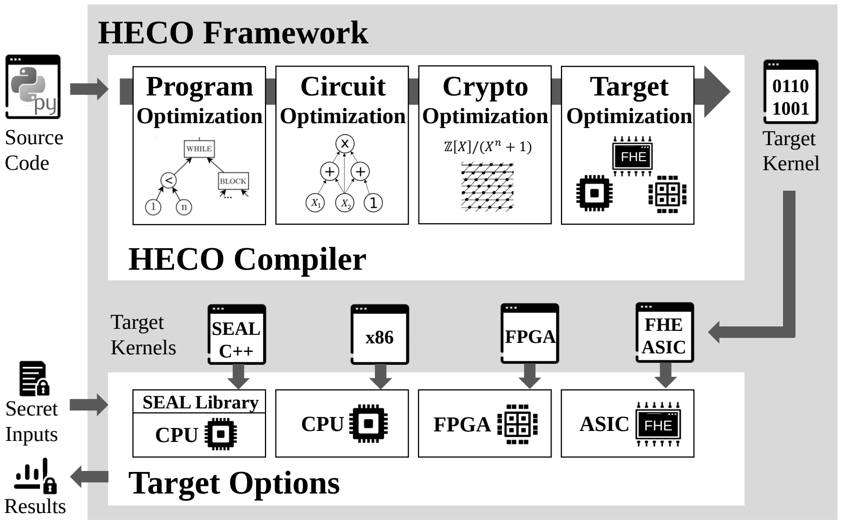


Figure 4.1: Overview of our end-to-end design, showing the compilation flow from a high-level input program to an efficient FHE kernel running on a target backend.

tically different from their plaintext equivalents. Adapting to this unique programming paradigm requires a lot of experience and poses a significant barrier to entry for non-experts.

### 4.1.2 *System Overview*

HECO proposes a multi-staged approach to FHE compilation that encompasses: *(i) Program Transformations*, which restructure high-level programs to be efficiently expressible using native FHE operations, *(ii) Circuit Optimizations*, which primarily focuses on changes that reduce noise growth in the FHE computation, *(iii) Cryptographic Optimizations*, which instantiate the underlying scheme as efficiently as possible for the given program, and *(iv) Target Optimizations*, which map the computation to the capabilities of the target. We propose a set of Intermediate Representations (IRs) designed to provide a suitable abstraction of each stage, allowing us to naturally and efficiently express optimizations at these different levels. In contrast, existing compilers usually abstract FHE computations as circuits[2] which does not allow them to fully express many optimizations at the high-level (program transformation) or at the lowest level (target optimization) because these need to consider aspects such as data-flow or memory management which have no natural correspondence in a circuit representation. In HECO, high-level programs are lowered through a series of transformations, using multiple increasingly lower-level IRs to produce the target kernel. These kernels can then be targeted and run against various back-end options. We provide a user-facing Python framework that abstracts away the complexities of this process, supports a Python-embedded Domain Specific Language for FHE, and provides a unified experience for development, compilation and execution. We provide an overview of our end-to-end design in Figure 4.1. In the remainder of this section, we describe HECO's components, abstractions, and compilation stages.

### 4.1.3 *HECO Framework*

HECO's framework ties together the front end, compiler, and the various back ends into a unified development experience. It allows developers to edit, compile and deploy their applications from a familiar Python environment. In order to provide an intuition of the developer experience in our system, we provide an example of using HECO to compile and run an FHE program in Listing 1 (Server) and Listing 2 (Client). By wrapping FHE functions in `with` blocks, we can operate on them as first-class entities, making compilation explicit. Our framework provides the necessary infrastructure to run programs directly from the front end, allowing developers to integrate FHE functionality into larger applications easily.

**Python-Embedded DSL.** HECO uses Python to host its Domain-Specific Language (DSL), inheriting Python's syntax and general semantics. We want to allow developers to write programs in as natural a fashion as possible, and merely require type annotations to denote inputs that are

2 This is natural since FHE computations are usually modeled mathematically as arithmetic circuits.

```python
def server(x_enc, y_enc, public_context):
  p = FrontendProgram()
  with CodeContext(p):
    def euclidean_sq(x: Tensor[8, Secret[int]],
                     y: Tensor[8, Secret[int]])
                     -> Secret[int]:
      sum: Secret[int] = 0
      for i in range(8):
        d = x[i] - y[i]
        sum = sum + (d * d)
      return sum

  # compile FHE code
  f = p.compile(context=public_context)

  # run FHE code using SEAL
  r_enc = f(x_enc, y_enc)
  return r_enc
```

Listing 1: Example server-side code using HECO.

`Secret`. In order to facilitate this, HECO supports (statically sized) loops, access to vector elements, and many other high-level features that do not have a direct correspondence in FHE. Since our compilation approach requires a high-level representation of the input program, including these non-native operations and the control-flow structure, we cannot follow the approach used by most existing tools. These tend to execute the program using placeholder objects that record operations performed on them, which is equivalent if considering FHE programs as circuits but removes most of the high-level information about the program structure. Instead, we use Python's extensive introspection features to parse the input program and translate the resulting Abstract Syntax Tree (AST) directly to our high-level IR.

### 4.1.4 *Compiler Infrastructure*

The core of HECO is an optimizing compiler that translates and optimizes programs by lowering them through a series of progressively lower-level

```
def client(x : Tensor[int], y : Tensor[int]):
  # Select SEAL backend, scheme and params
  context = SEAL.BFV.new(poly_mod_degree=2048)

  # encrypt input
  x_enc = context.encrypt(x)
  y_enc = context.encrypt(y)

  # send enc input to server
  r_enc = server(x_enc, y_enc, context.pub())
  result = context.decrypt(r_enc, context)
```

Listing 2: Corresponding client-side code, outsourcing the computation of the (squared) euclidean distance.

Intermediate Representations (IRs). This section describes how we build upon the MLIR framework to realize HECO's compiler design.

**Multi-Level Intermediate Representations.** HECO's middle end exposes multiple levels of abstractions to facilitate our multi-stage compilation & optimization approach. This is realized through a series of Intermediate Representations (IRs), as seen in Figure 4.2. We leverage the MLIR framework [118], which was designed specifically to facilitate *progressive lowering*, introducing additional IRs to reduce the complexity of each lowering step. MLIR defines a common syntax for IR operations, for example, an addition might be represented as `%2 = artih.addi(%0, %1): (i16, i16) -> i16`. MLIR is strongly typed, however, for conciseness, we will omit the details of type conversions when discussing transformations. Intermediate Representations in MLIR are composed of sets of operations known as *dialects*. We define a custom dialect for our high-level abstraction of FHE (`heco::fhe`) and combine this with built-in dialects for vector operations (`mlir::tensor`), plaintext arithmetic (`mlir::arithmetic`) and basic program structure (`mlir::affine`, `mlir::func`) to realize our High-Level Intermediate Representation (HIR). In addition, we define dialects for each of the supported FHE schemes, mirroring their natively supported operations (`heco::bfv`, `heco::bgv`, `heco::ckks`). MLIR also includes a variety of standard simplification passes, which can be extended to custom dialects by defining appropriate interfaces.

**Supporting Different Back-Ends.** FHE is actively evolving, and as such, tools need to be able to adapt to new and improved implementations, both in software and hardware. This requires a high level of modularity and flexibility from the compiler. In HECO, we achieve this by using target-specific dialects, which can be customized and extended as new back ends are introduced. While traditional library-based implementations targeting CPUs and GPUs share a common API (conceptually, if not technically), upcoming FPGA and ASIC accelerators for FHE [70, 81, 88, 154]) feature a much lower-level interface. These systems are designed to efficiently realize the required mathematical operations in the modular rings of polynomials that underly most FHE schemes, and as a result, their Instruction Set Architectures (ISA) operate on this level. In order to support this, HECO is designed to be easily extended to match this abstraction level, featuring MLIR dialects for both bignum polynomial ring operations (`heco::poly`) and for the commonly used Residue Number System (RNS) approach using the Chinese Remainder Theorem (CRT) to split these large datatypes into hardware-sized elements (`heco::rns`). In addition to targeting hardware accelerators, the ability to lower to this level also allows targeting `x86` directly via LLVM IR and the LLVM toolchain.

### 4.1.5 *Transformation & Optimization*

The transformations and optimizations in HECO are grouped according to the four stages of compilation we identified, and we present them accordingly in the following.

**Program Transformations.** The first phase of compilation focuses on high-level transformations and optimizations. This includes a wide variety of general (e.g., constant folding, common sub-expression elimination) and FHE-specific optimizations that allow developers to write code more naturally by removing the need for menial hand-optimization. Most importantly, however, it focuses on optimizations that map the input program to FHE's unique programming paradigm, such as the automated batching optimizations, which we present in more detail in Section 4.2. Previous work has

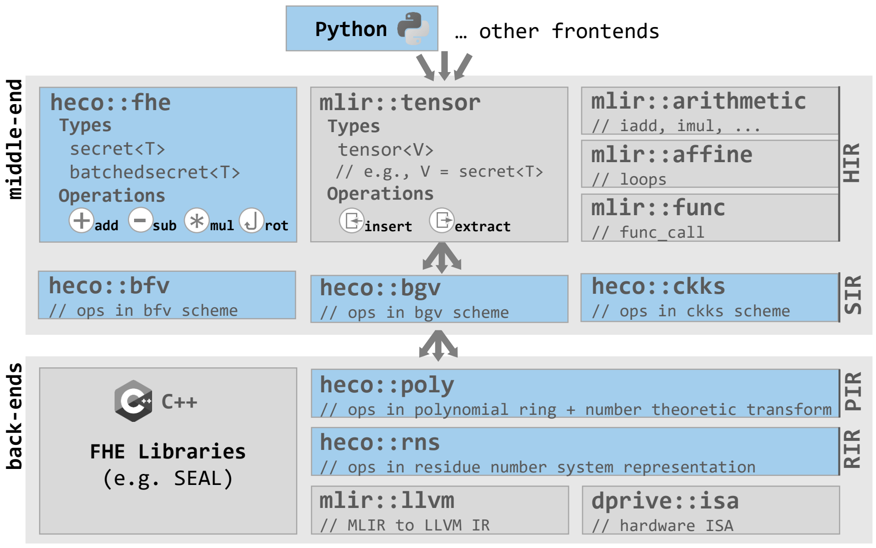


Figure 4.2: Overview of HECO's dialects, which define the operations used in the Intermediate Representations (IRs).

shown that performance differences between the runtimes of well-mapped and naively-mapped implementations can easily reach several orders of magnitude [161]. As a result, a significant part of our focus in HECO is on this level of abstraction, which existing FHE tools generally do not support.

**Circuit Optimizations.** After mapping to the FHE paradigm, the program is conceptually equivalent to an arithmetic circuit of native FHE operations. This is the level of abstraction considered by the vast majority of existing tools. Optimizations at this stage are mostly concerned with managing the noise growth in the computation. For example, a variety of optimizations that try to re-arrange the arithmetic operations to reduce the number of sequential multiplications have been proposed [6, 7, 33, 34]. However, even state-of-the-art optimizations in this style are likely to accelerate a program by only around 2x in practice [161]. We omit a detailed description of these techniques here as they are not the focus of this work. More importantly, this level of abstraction is also where we must consider ciphertext maintenance operations. These do not modify the encrypted messages, but significantly affect future noise growth, making them essential for practical FHE. HECO uses a traditional approach of inserting relinearization operations [78] between all consecutive multiplications. This is always correct but not necessarily strictly optimal, and more sophisticated strategies [58, 71, 123] could offer further improvements. The modularity of our design makes it a straightforward future work to include these techniques, but since these have been explored in the past, we do not focus on them in this work.

**Cryptographic Optimizations.** In the third phase, we consider *cryptographic optimizations* focused on instantiating the underlying FHE scheme as efficiently as possible. When targeting existing FHE libraries, the primary challenge is parameter selection: identifying the smallest (i.e., most efficient) parameters that still provide sufficient noise capacity to perform the computation correctly. Different techniques have been proposed to estimate the expected noise growth of an FHE program [60, 62, 107]. These include theoretical noise analysis, where recent work has achieved tighter bounds for some schemes [60], but which generally tend to significantly overestimate noise growth [62], leading to unnecessarily large parameter choices. As a result, experts primarily still rely on a trial-and-error process to experimentally determine the point at which noise invalidates the results. HECO includes basic automatic parameter selection based on a simple multi-depth heuristic but also allows experts to easily override these suggestions. When targeting hardware directly, rather than through libraries, further optimization opportunities open up. For example, many ciphertext maintenance operations can be instantiated in different ways, offering trade-offs between runtime, memory consumption, and noise behavior. While libraries tend to implement a general-purpose compromise, compilers can adaptively choose the most appropriate approach for a given computation. However, this requires re-expressing the complex underlying logic of FHE schemes inside the compiler. HECO inherits a powerful system of abstractions and optimizations for computationally intense mathematics from the MLIR framework, allowing our system to be easily extended with such optimizations in the future.

**Target Optimization.** Finally, in the fourth phase, we consider *target-specific optimizations*. In addition to general code generation optimizations, there is a significant opportunity for FHE-specific optimizations at this level. For example, when available, FHE benefits greatly from instruction set extensions such as `AVX512`. This concept has already been explored in the context of libraries [15], and implementing similar techniques in HECO should be straightforward given our modular design. When targeting hardware, FHE accelerators impose non-trivial constraints on memory and register usage, due to complex memory hierarchies. Initial work in this space has already shown that code generation and scheduling can have a significant impact on accelerator performance [88, 154]. HECO supports optimizations at this level through our low-level dialects for the underlying math, and can easily be extended with target-specific dialects for the Instruction Set Architectures (ISAs) of upcoming accelerators, e.g., those developed by the DARPA DPRIVE program [70].

## 4.2 AUTOMATIC SIMD BATCHING

In this section, we introduce our automated SIMD batching optimization, which is part of our program transformation stage and maps traditional imperative programs to the restrictive SIMD-like setting of state-of-the-art FHE schemes.

### 4.2.1 *SIMD Batching*

Effective use of batching is arguably the single most important optimization for many applications and is omnipresent in most state-of-the-art FHE results. Due to the large capacity of FHE ciphertexts (usually $2^{13} - 2^{16}$ slots), applying batching has the potential to drastically reduce ciphertext expansion overhead and computation time. While batching can be used to trivially increase throughput, most FHE applications are constrained by latency. However, employing batching effectively to improve performance on a single input is non-trivial due to the restrictions imposed by FHE's unusual programming paradigm. Therefore, unlocking the performance potential of batching currently requires significant expertise and experience in writing FHE applications. In the following, we present a simple example that showcases the drastic transformations that can be required to achieve efficient batching, followed by a brief introduction of a folklore technique that demonstrates common patterns of FHE batching optimizations.

**Example Application – Image Processing.** We consider a simple image processing application (see Listing 3a), which nevertheless features a complex loop nest structure and non-trivial index patterns. Specifically, we consider a Laplacian Sharpening filter, i.e., a convolution of a $(3x3)$ kernel over an image, implemented with wrap-around padding. The function is compatible with efficient arithmetic-circuit based FHE, as it does not use data-dependent branching and only requires homomorphic addition and multiplications operations. However, its current form makes use of nested loops accessing a complex set of indices, which is not very amenable to efficient batching as there appears to be little opportunity for operations over entire ciphertexts.

Nevertheless, there exists a significantly more efficient *batched* design, as seen in Listing 3b. In the optimized version, the input image is batched into a single ciphertext, and all homomorphic operations make full use of their SIMD nature. Instead of iterating the kernel over the image, nine copies of the image are made and each is rotated so that all elements interacting at a specific kernel position align at the same index. This is possible, because the relative offset between different pixels in the kernel remains static, even though the indices themselves are different for each iteration. The transformation enables the the runtime of the program to depend on the (small) kernel size, rather than the image size. As a result, the batched version is more than an order of magnitude more efficient than a naive implementation. These types of drastic transformations are common in state-of-the-art FHE applications and significant experience is required to develop an intuition for this unusual programming paradigm.

**Rotate-and-Sum.** In the example above, only interactions between values in different ciphertexts were required. However, it is also possible to efficiently

```python
def foo(img: Tensor[N, Secret[f64]]):
  img_out = img.copy()
  w = [[1, 1, 1], [1, -8, 1], [1, 1, 1]]
  for x in range(n): # loop over pixels
  for y in range(n):
    t = 0
    for j in range(-1, 2): # apply kernel
    for i in range(-1, 2):
      t += w[i+1][j+1] * img[((x+i)*n+(y+j))%N]
      img_out[(x*n+y)%N] = 2*img[(x*n+y)%N] - t
  return img_out
```

(a) Textbook implementation of a simple sharpening filter

```python
def foo_batched(img: BatchedSecret[f64]):
  r0 = img * -8
  r1 = img << -n-1
  r2 = img << -n
  r3 = img << -n+1
  r4 = img << -1
  r5 = img << 1
  r6 = img << n-1
  r7 = img << n
  r8 = img << n+1
  return 2*img-(r0+r1+r2+r3+r4+r5+r6+r7+r8)
```

(b) Optimized batched solution of the same program

Listing 3: Both of these functions apply a simple sharpening filter to an encrypted image of size $n \times n = N$, by convolving a $3 \times 3$ kernel ($-8$ in the center, 1 everywhere else) with the image. The version on the left encrypts each pixel individually, and follows the textbook version of the algorithm, operating over a vector of $N$ ciphertexts. The version on the right batches all pixels into a single ciphertext and uses rotations (<<) and SIMD operations to compute the kernel over the entire image at the same time. Designing batched implementations requires out-of-the-box thinking in addition to significant expertise and experience.

**Algorithm 1** Rotate-and-Sum

```
1: Algorithm SumVectorPowerTwo($x, n$)
2:     for $i \leftarrow \frac{n}{2}$ downto 1 by $i \leftarrow \frac{i}{2}$
3:         $x \leftarrow x + \text{Rotate}(x, i)$
       return $x$
```

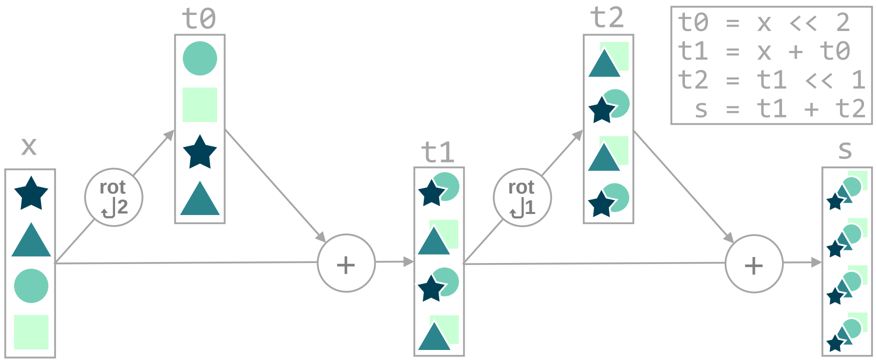


Figure 4.3: Illustration of how repeated copying and rotating can be used to compute the sum of all elements in a ciphertext in a logarithmic, rather than linear, number of steps.

realize certain operations on the elements of a single ciphertext; we now describe a common folklore technique used to achieve this: The *rotate-and-sum* algorithm allows us to efficiently sum up the elements of a ciphertext, using $\mathcal{O}\left(()\log n\right)$ rotations (where $n$ is the number of ciphertext slots). The algorithm proceeds by creating a copy of the current vector, rotating it and then adding both before repeating the same procedure with a lower offset (c.f. Algorithm 1). This is visualized in Figure 4.3 for a vector size of four. While this technique is applicable, the performance benefits are usually overshadowed by more radical transformations, such as the example shown above. However, using rotate-and-sum and similar rotation-based approaches can be worthwhile if it enables other parts of the program to remain slot-aligned.

### 4.2.2 *Automatic Batching Approach*

Experts generally rely on their experience with the FHE programming paradigm to transform and optimize programs for batching, posing a high barrier to entry for non-expert developers. Instead, formal methods to automatically translate traditional imperative programs into efficient batched FHE solutions are required. We assume the input program computes the elements from vectors of secret values in a non-SIMD fashion (e.g., Listing 3a). Of course, this can be naively realized by encrypting each vector element into one ciphertext , but this usually does not achieve acceptable performance due to the high overhead of FHE. The goal of automated batching is to amortize the cost of each FHE operation by utilizing as many ciphertext slots as possible for meaningful computation. In the following, we discuss two potential alternative approaches and their drawbacks before introducing our approach.

**Strawman Approach.** Batching each vector in the input program into a ciphertext will trivially achieve a 'batched' solution. However, this raises the question of how to execute the computations over individual elements present in the program. Element-wise access (`extract` and `insert`) are not native FHE operations and must instead be emulated, requiring several rotations and ciphertext-plaintext multiplications. For example, Listing 4 shows how `x[i] = x[i] + y[j]` can be emulated in the batched setting. However, this replaces each FHE operation from the naive, vector-of-ciphertexts approach, with multiple expensive FHE operations. As a result, unless ciphertext expansion is significantly more important than runtime, this approach is virtually always ill-advised. Therefore, most existing FHE tools use the vector-of-ciphertexts approach rather than attempting to perform batching.

**Alternative Approaches.** There have been initial attempts at performing automated batching for FHE using Synthesis-based approaches [63]. While these can, in theory, achieve the drastic transformations required to exploit batching, they are not suitable for practical use in real-world code development, as they do not scale beyond toy-sized program snippets, and

```
%1=tensor.extract %x[i]
%2=tensor.extract %y[j]
%3=fhe.add(%1, %2)
%4=tensor.insert(%3,%z[i])
```

(a) Input Program

```
%1=fhe.mul(%x, %m_i)
%2=fhe.rotate(%1, -i)
%3=fhe.mul(%y, %m_j)
%4=fhe.rotate(%3, -j)
%5=fhe.add(%2, %4)
%6=fhe.rotate(%5, i)
%7=fhe.mul(%z, %mn_i)
%8=fhe.add(%6, %7)
```

(b) Naive Batching

Listing 4: Strawman batching approach for `z[i] = x[i] + y[j]`, showing the necessary rotations and multiplications with masking vectors: `%m_i`, `%m_j` are zero everywhere except at $i$ or $j$, respectively; `%mn_i` is one everywhere except at $i$.

even those can take minutes to optimize. Alternatively, one might consider applying traditional Superword-level Parallelism (SLP) vectorization algorithms [42, 117, 134], as these try to group operations into SIMD instructions. However, these generally rely on the ability to efficiently scatter/gather elements into and out of vectors, which is only possible at a high cost in FHE. While some recent work can reason about the cost of data movement, it does not consider how data movement introduced at the beginning of the program might affect later parts of the program [42].

**HECO's Approach.** HECO's batching transformation starts with the core idea of the strawman approach, i.e., batching vectors of secrets into ciphertexts but eliminates the overhead of emulating `insert`/`extract` operations for batching amenable programs. Rather than directly emulating these operations, we instead translate the homomorphic operations in which they appear as operands. While this still requires inserting rotation operations, it allows operations with compatible index access patterns to be mapped to the same emulated code if using an appropriate algorithm. As a result, a simple built-in simplification pass can eliminate the duplicates. In the case of well-structured programs, this can completely eliminate emulation related code. For example, for the program from the previous section (Listing 3a), HECO produces exactly the optimized code seen in Listing 3b, which does not contain any emulation-related code.

**HECO Batching Pipeline.** HECO's batching transformation is composed of a series of smaller passes, each interleaved with built-in simplification passes. Before the main batching passes, we first perform a series of preprocessing steps that unroll statically-sized loops, merge sequential associative binary operations into $n$-ary group operations, and perform a type conversion from vectors of secrets to `BatchedSecrets`, which are HECO's high-level abstraction of ciphertexts. Following this, the main pass walks through the program and transforms each operation over secret vector elements into operations over entire vectors. Similar to the strawman approach, this involves introducing rotations, but our approach does not require multiplications with masks. Additionally, the way we perform these translations allows us to 'chain' them so that consecutive operations on the same vector elements do not result in separate emulation code. After the main pass and the associated simplification pass, we apply the rotate-and-sum technique where applicable, which is enabled by both the merging of operations during the preprocessing phase and the exposure of same-ciphertext operations by the main pass. In the following, we first outline some key preprocessing steps before explaining the two main optimization steps.

### 4.2.3 *Preprocessing*

In addition to standard simplifications and canonicalization of the IR (i.e., bringing operations into a standardized 'canonical' form to reduce the complexity of the IR), we also apply two more specialized transformations.

**Merging Arithmetic Operations.** During the preprocessing stage, we combine chained applications of (associative and commutative) binary operations into larger arithmetic operations with multiple operands (e.g., merging `x=a+b; y=x+c` to `y=a+b+c`). This removes chains of dependencies and replaces them with a single operation, making it easier to identify *rotate-and-sum* optimization opportunities. In addition, it can also allow more efficient direct lowerings, such as when performing the product over $n$ elements: which can be lowered efficiently to a multiplication tree with depth $\log n$.

**Type Conversion.** While tensors can be arbitrarily (re)shaped, all FHE ciphertexts used in a homomorphic computation must have compatible parameters, which implies a fixed number of slots. Therefore, we perform type conversion, converting all vector operations over secret values to operations over the `BatchedSecret` type, an abstract representation of ciphertexts. We convert multi-dimensional tensors to vectors using column-major encoding and scale up any secret vector operands to the size of the largest (secret) vector present, padding the plaintext as necessary. This does not impact the result of the computation, as the existing code will never access these additional elements.

### 4.2.4 *Automatic SIMD-fication*

This pass replaces scalar operations over vector elements (e.g., `x[i] + y[j]`) with SIMD operations, applying the same operation to each element of the ciphertext. At its core, the pass is a linear walk over all (arithmetic) homomorphic operations in the program, as seen in Algorithm 2. For each operation, we *(i)* identify in which ciphertext slot the result should be computed *(ii)* transform the operands so that they are suitable for such a SIMD-operation, and *(iii)* insert `extract` operations in situations where a scalar is expected (including uses in later operations that have yet to be transformed). Note that, by itself, this transformation does not actually remove any code. However, it will expose common patterns (e.g., such as those occurring in loops) and cause operations over *compatible* indices to be translated to the same SIMD operations. This allows the following clean-up pass to remove these now-redundant operations, which frequently includes duplicate arithmetic operations and many or all of the operations inserted to ensure consistency.

**Target Slot Selection.** When translating an operation with operands corresponding to different vector positions (e.g., `x[i]+y[j]`), we must bring the elements of interest into alignment by issuing a rotation for at least some of the operands. However, there are usually multiple valid solutions (e.g., `rot(x, j-i)` vs. `rot(y, i-j)`), especially for operations with multiple

**Algorithm 2** Batching Pass

```
1: Algorithm BatchPass($\mathcal{G}$)
2:     $\mathcal{V}, \mathcal{E} \leftarrow \mathcal{G}$
3:     foreach $op \in \mathcal{V} \wedge \text{type}(op) = \text{fhe.secret}$:
4:         $ts \leftarrow$ SelectTargetSlot($op, \mathcal{V}, \mathcal{E}$)
5:         OperandConversion($op, ts, \mathcal{V}, \mathcal{E}$)
6:         foreach $v \in \mathcal{V} \wedge (op, v) \in \mathcal{E}$:
7:             $u \leftarrow \text{fhe.extract}[v, ts]$
8:             Replace($v, u, \mathcal{V}, \mathcal{E}$)
9: procedure SelectTargetSlot($op, \mathcal{V}, \mathcal{E}$)
10:     foreach $v \in \mathcal{V} \wedge (op, v) \in \mathcal{E}$:
11:         switch $v$:
12:             case $\text{fhe.insert}[\_, i]$: return $i$
13:             case func.return: return 0
14:     foreach $v \in \mathcal{V} \wedge (v, op) \in \mathcal{E}$:
15:         switch $o$:
16:             case $\text{fhe.extract}[\_, i]$:
17:                 return $i$
18:     return $\perp$
19: procedure OperandConversion($op, ts, \mathcal{V}, \mathcal{E}$)
20:     foreach $v \in \mathcal{V} \wedge (v, op) \in \mathcal{E} \wedge \text{type}(v) = \text{fhe.secret}$:
21:         switch $v$:
22:             case $\text{fhe.extract}(x, i)$:
23:                 $u \leftarrow \text{fhe.rotate}(x, i - ts)$
24:                 Replace($v, u, \mathcal{V}, \mathcal{E}$)
25:             case $\text{fhe.ptxt}[p]$:
26:                 $p' \leftarrow$ Repeat($p$)
27:                 $u \leftarrow \text{fhe.ptxt}(p')$
28:                 Replace($v, u, \mathcal{V}, \mathcal{E}$)
29: procedure Replace($v, u, \mathcal{V}, \mathcal{E}$)
30:     $\mathcal{V} \leftarrow (\mathcal{V} \setminus \{v\}) \bigcup \{u\}$
31:     foreach $w \in \mathcal{V} \wedge (v, w) \in \mathcal{E}$:
32:         $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{(v, w)\}) \bigcup \{(u, w)\}$
33:     foreach $w \in \mathcal{V} \wedge (w, v) \in \mathcal{E}$:
34:         $\mathcal{E} \leftarrow (\mathcal{E} \setminus \{(w, v)\}) \bigcup \{(w, u)\}$
```

operands, which occur frequently as a result of our preprocessing stage An obvious approach would be to rotate each operand (e.g., `x[i]`) so that the element of interest is moved to slot 0. Then, performing the SIMD operation over the rotated operands produces the result in the same slot. While this approach is straightforward and correct, it is unsuitable for an optimizing compiler, as it does not set the program up for further simplification. For example, in the case of a loop `for i in 0..10: z[i]=x[i]+y[i]`), each iteration would result in a unique operation with distinct operands, each rotated by different amounts.

Instead, HECO introduces the notion of a *target slot*, determined by the further uses of the result. For example, if the result of a computation is assigned to `z[k]`, we select $k$ as the target slot, eliminating the rotation required afterward. If no clear target slot can be derived from the uses of the result, we use one of the operand indices as the target slot, removing the need to rotate that operand. Selecting the target slot this way reduces the immediate number of rotation operations created. More importantly, it reliably maps operations with the same *relative* index access patterns to the same set of rotations and SIMD operations, allowing them to be eliminated by the following simplification pass. Since this approach is based purely on index access patterns, it works equally well for complex loop nests and heavily interleaved code.

**Operand Conversion.** In order to convert a scalar operation to a fully-batched SIMD operation, all non-batched inputs must be converted, as described in Algorithm 2 (OperandConversion). Note that, due to the linear-walk nature of the pass, all previous FHE operations have already been converted to fully-batched operations. As a result, any operand of type `fhe.secret` must be the result of an `fhe.extract` operation. This invariant allows us to 'chain' batched operations together by replacing the extraction-based operand with a rotation-based operand. Specifically, an operand extracted from slot $i$ of vector $x$, is replaced with a rotation of $x$ by $i - ts$, where $ts$ is the target slot determined before.

**Ensuring Consistency.** We maintain the consistency and correctness of the program at each step of the optimization. Towards this, we first construct the new rotations and batched operations as additions to the program. We only replace occurrences of the old operation with the optimized version after we replace uses of the old operation with an `extract` operation that extracts the target slot of the new batched result. This ensures that, even if no further batching opportunities are found, the program remains correct. Since batched FHE schemes do not support true scalar values, we simply interpret scalars as ciphertexts where only slot 0 contains valid data. With this convention, any remaining extractions will eventually be converted to a rotation by $-ts$. However, in practice, this is rare as most of the `extract` operations we insert will in turn be converted to rotations when the next homomorphic operation is processed. As a result, these consistency-related `extract` operations are frequently eliminated completely at the end of the batching pass.

### 4.2.5 *Rotate-and-Sum Pass*

After the main pass and the associated simplification pass, we apply the *rotate-and-sum* technique where applicable. Since this optimization requires a holistic view of the operation, this would be significant if we did not merge sequential operations during preprocessing. While we used a sum over all elements when explaining the technique in the previous section, the technique can be generalized to any subset with a consistent stride. Additionally, it can also be used to compute products rather than sums. When applied to multiplication, it additionally has the benefit of automatically reducing the multiplicative depth of the expression as a side-effect.

Note that the pre-processing combination of binary operations into larger operations *must* happen before the main pass described above, as that pass would otherwise insert rotations between the different operations, making them no longer directly chained. The actual translation to a series of rotations and native binary operations, meanwhile, has to be performed *after* that pass, since it requires the operands to be entire ciphertexts, rather than scalars. Additionally, the de-duplication simplifications that can take place after the batching transformation can widen the applicability of this transformation by reducing the number of distinct ciphertexts appearing in the program.

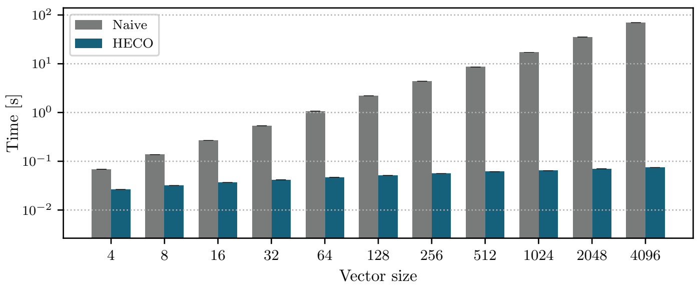
## 4.3 IMPLEMENTATION

We build HECO on top of the open-source MLIR framework [118], which is rapidly establishing itself as the go-to tool for domain-specific compilers and opening up the possibility of exchanging ideas and optimizations even beyond the FHE community. HECO consists of roughly 15k LOC of C++, with around 2k LOC of Python for the Python front-end. HECO uses the Microsoft Simple Encrypted Arithmetic Library (SEAL) as its FHE backend. SEAL, first released in 2015, is an open-source FHE library implemented in C++ that is thread-safe and heavily multi-threaded itself. SEAL implements the BFV, BGV and CKKS schemes.

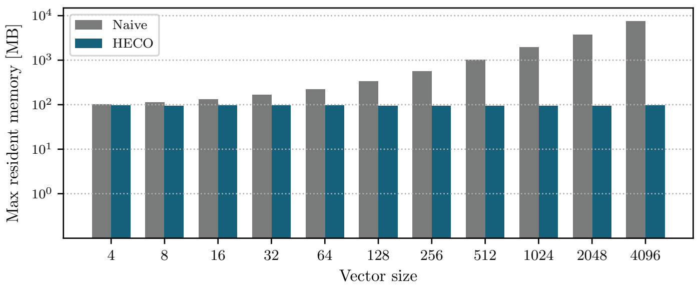
In contrast to existing monolithic compilers, HECO is highly modular and designed to be flexible and extensible. We decouple optimizations from front-end logic, allowing for a wide variety of domain-specific front-ends and the ability to easily replace back-ends to target different FHE libraries or hardware accelerators as they become available. The toolchain can easily be adapted to different needs, with certain optimizations enabled or disabled as required.

### 4.3.1 *Evaluation*

HECO is designed to compile high-level programs, written by non-experts in the standard imperative paradigm, into highly efficient batched FHE implementations that achieve the same performance as hand-crafted implementations by experts. HECO achieves its usability goals through a well-integrated Python front-end and by requiring developers to alter their code only minimally (annotating variables as secret). However, ease-of-use becomes moot when the performance of the generated code is not competitive. Therefore, we focus our evaluation on the performance of HECO and the code it generates, trying to answer whether or not automatic op-



(a) Hamming Distance benchmark runtime (in seconds).



(b) Hamming Distance benchmark memory usage (in MB).

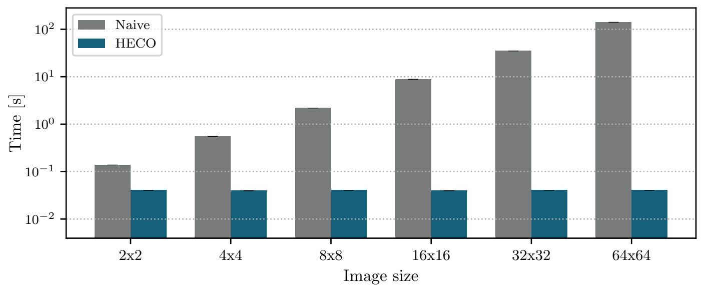
Figure 4.4: Log-log plot of the runtime and memory consumption of the hamming distance benchmarks for different vector sizes, comparing a naive non-batched solution with the batched solution generated by our system.

timizations can bring naive code to the same performance level as expert implementations. In this section, we first show the effect of the batching optimizations on benchmark workloads designed to demonstrate different batching patterns, then compared against synthesized optimal batching patterns, and finally discuss a real-world application example.

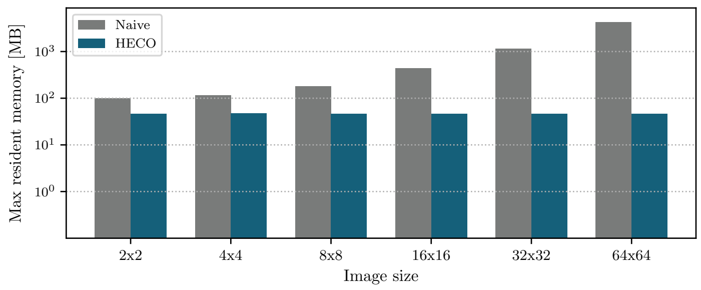
### 4.3.2 *Benchmarks*

We evaluate HECO in terms of the speedup reduction in memory overhead gained over non-optimized implementations and the compile time required.

**Applications.** We demonstrate the speedup achieved by our batching optimization on two applications that are representative of common batching



(a) Roberts Cross benchmark runtime (in seconds).



(b) Roberts Cross benchmark memory usage (in MB).

Figure 4.5: Log-log plot of the runtime and memory consumption of the roberts cross benchmark for different vector sizes, comparing a naive non-batched solution with the batched solution generated by our system.

opportunities. The *roberts cross operator* is an edge-detection feature used in image processing. It approximates the gradient of an image as the square root of the sum-of-squares of two different convolutions of the image, which compute the differences between diagonally adjacent pixels. As in all other kernel-based benchmarks, wrap-around padding is used, which aligns well with the cyclical rotation paradigm of FHE. In order to enable a practical FHE evaluation, the final square root is omitted, since it would be prohibitively expensive to evaluate under encryption. The *hamming distance*, meanwhile, computes the edit distance between two vectors, i.e., the number of positions at which they disagree. Here, we consider two binary vectors of the same length, a setting in which computing (non-)equality can be done efficiently using the arithmetic operations available in FHE. Specifically, this makes use of the fact that $\text{NEQ}(a, b) = \text{XOR}(a, b) = (a - b)^2$ for $a, b \in \{0, 1\}$.

**Baseline.** Our baseline is a naive implementation of the application without taking advantage of batching, as one might expect FHE novices to implement. In this setting, vectors of secrets are directly translated to vectors of ciphertexts. While this approach introduces significant ciphertext expansion and increases the memory required, it is actually preferable in terms of run time over a solution that batches vector data into ciphertexts, but does not re-structure the program to be batching-friendly. This is because such a solution adds the overhead of rotations, masking, etc., to the base runtime of the non-batched solution.

**Environment.** All benchmarks are executed on AWS `m5n.xlarge` instances, which provide 4 cores and 16 GB of RAM. We used Microsoft SEAL [156] as the underlying FHE library, targeting its BFV [21, 78] scheme implementation. All experiments are run using the same parameters, which ensure at least 128-bit security. We report the run time of the computation itself, omitting client-side aspects such as key generation or encryption/decryption. All results are the average of 10 iterations, discarding top and bottom outliers.

**Runtime & Memory Overhead.** In Figure 4.5a we show the runtime of the Roberts Cross benchmark for varying instance sizes, comparing the non-batched baseline with the batched solution generated by HECO. While the run time of the naive version increases linearly with the image size, the batched solution maintains the same performance (until parameters must be increased to accommodate even larger images). Instead, the run time is more closely tied to the size of the kernel than to that of the image. This highlights the dramatic transformations achieved by HECO, fundamentally changing the structure of the program. As a result of these transformations, HECO achieves a speedup of 3454x over the non-batched baseline for 64x64 pixel images, demonstrating the extraordinary impact that effective use of batching can have on FHE applications: while the non-batched solution is borderline impractical at over two minutes, the batched solution takes only

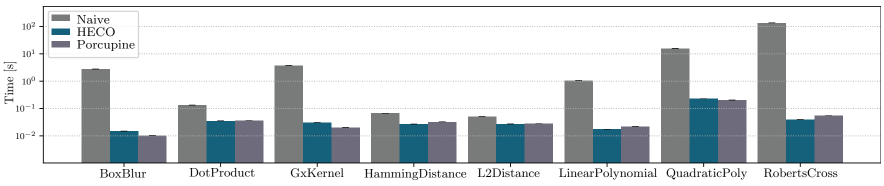


Figure 4.6: Runtime of example applications (in seconds), comparing a naive non-batched baseline, the solution generated by our system (HECO), and an optimally-batched solution synthesized by the Porcupine tool.

a fraction of a second (0.04 s). The runtime of the generated code in this case does depend directly on the vector length. However, due to the fold-style optimization (cf. Section 4.2.5), this dependence is only logarithmic.

In Figure 4.4b we can see that, for realistic problem sizes, the performance advantage of batching becomes significant, resulting in a speedup of 934x for 4096-element vectors. We also see that, while the runtime of the batched solution does increase with the vector length, this is nearly imperceptible when compared to the non-batched baseline. Finally, in Figure 4.5b/4.4a, we show the memory overhead of the non-batched baseline and the batched solution. While FHE introduces a non-negligible baseline overhead due to the large amount of key- and other context-data that must be maintained, the reduced number of ciphertexts in the batched solution has a clear impact on memory usage, increasingly so as the problem sizes increase.

**Compile Time.** HECO achieves these fundamental transformations efficiently, with compile times that are amenable to interactive development. This is in contrast to synthesis-based tools, which require more than 10 minutes to synthesize a batched solution for the Roberts Cross benchmark even for toy-sized instances and do not scale to the sizes we consider here at all [63].

### 4.3.3 *Comparison with Synthesized Solutions*

We compare the baseline and HECO to synthesized optimal batching patterns. Synthesis based approaches explore the space of all possible programs, constrained by a reference specification describing input-output behavior. While this tends to be computationally expensive, it has the potential to find optimal solutions featuring highly non-intuitive optimizations. We use a set of nine benchmark applications that represent a variety of common code patterns relevant to batching for our evaluation. These programs range from having no inter-element dependencies (e.g., Linear Polynomial), to simple accumulator patterns (e.g., Hamming Distance), and complex dependencies across a multitude of different vector elements (e.g., Roberts Cross). As a result, they provide a useful benchmark, especially for batching optimizations. These benchmarks were first proposed in [63], which introduces Porcupine [63], a synthesis-based compiler for batched FHE. In addition to a reference specification, it requires a developer-provided sketch of an initial possible batched approach. Our 'synthesis' solutions are based on pseudo-code made available in an extended version of the paper [63].

Synthesis based tools can require the significant search time to find solutions, limiting them to toy-sized workloads. For example, Porcupine requires over 10 minutes to synthesize a program with ten instructions and will fail to synthesize a solution at all for sufficiently complex programs. As a result, we consider the following problem sizes here: The synthesized dot-product code targets 8-element vectors, while those for Hamming Distance and L2 Distance were provided for 4-element vectors. For the other applications, we use vectors of length 4096, representing 64x64 pixel images

As we can see from Figure 4.6, HECO dramatically improves performance over the non-batched baseline approach. For example, for the Roberts Cross benchmark, our batched solution is over 3500 times faster than the non-batched solution, taking less than 0.04 seconds instead of over 2.35 minutes. More importantly, our results are nearly equivalent to the optimally batched solutions synthesized by Porcupine, especially when considering the stark contrast between the non-batched and the two batched solutions. In some cases (e.g., Box Blur), Porcupine has an advantage because it finds non-intuitive solutions beyond traditional batching patterns. Interestingly, for some applications (e.g. Hamming Distance) HECO actually

| $n$ | 4 | 16 | 64 | 256 | 1024 | 4096 |
|---|---|---|---|---|---|---|
| **rc** | 0.06 | 0.07 | 0.08 | 0.12 | 0.30 | 1.28 |
| **hd** | 0.03 | 0.04 | 0.06 | 0.09 | 0.25 | 1.85 |

Table 4.2: Compile time (in seconds) of the Roberts Cross (rc) and Hamming Distance (hd) benchmarks for different problem sizes ($n$).

outperforms Porcupine. This is because Porcupine provides an optimally batched solution but does not necessarily handle ciphertext management optimally, inserting unnecessary relinearization operations.

### 4.3.4 *Real-World Application*

The previous benchmarks demonstrated HECO's effectiveness and performance for different and common batching patterns. We now evaluate an application that more closely resembles the complexity that real-world settings exhibit. Specifically, we consider an application computing private statistics over two databases that might contain duplicate entries. We use this to demonstrate that HECO can produce efficient FHE code for non-trivial programs, while also highlighting that there is further room for optimizations exploiting application semantics.

**Application.** Privacy regulations frequently prohibit entities from combining sensitive datesets directly. Instead, they could employ *threshold* FHE, which extends FHE with multi-party key generation, to securely compute on the (encrypted) joint dataset. In this setting, neither party has sole access to the secret key, and they must collaborate to decrypt the results of approved queries. However, in practice their datasets might *overlap* (e.g., agencies at different levels of government collecting similar data), introducing duplicate items into the joint dataset. Due to the duplicates, analytics (e.g., counting queries) will return incorrect results. Therefore, we must first de-duplicate the encrypted databases before executing the analytics. Since threshold FHE does not affect the server-side execution of the com-

```python
def encryptedPSU(a_id: Tensor[128,8,Secret[int]],
                 a_data: Tensor[128,Secret[int]],
                 b_id: Tensor[128,8,Secret[int]],
                 b_data: Tensor[128,Secret[int]])
                 -> Secret[int]:
  sum: Secret[int] = 0
  for i in range(0, 128):
    sum = sum + a_data[i]
      for i in range(0, 128):
        unique: Secret[int] = 1
          for j in range(0, 128):
      # compute a_id[i] /= b_id[j]
    eq: sf64 = 1
    for k in range(0, 8):
       # a xor b == (a-b)^2
       x = (a_id[i][k] - b_id[j][k])**2
       nx = 1 - x  # not x
       eq = eq * nx # eq and nx
       neq = 1 - eq # not eq
       unique = unique * nequal

  sum = sum + unique * a_data[i]
  return sum
```

Listing 5: Computing statistics over duplicated data.

putation, HECO can be used directly to develop such an application. This first computes the *Private Set Union (PSU)* of two databases A,B indexed by unique IDs consistent across both (e.g., a national identifier like the SSN). For simplicity, we consider databases with one data column and a simple `SUM` aggregation. However, the presented approach trivially extends to larger databases and more complex statistics. Listing 5 shows the application expressed using the HECO Python frontend, for a database size of 128 elements and 8-bit identifiers. We split the identifiers into individual bits, allowing us to compute the equality function even while working with arithmetic circuits. The program begins by aggregating A's data and then proceeds to check each element of B's database for potential duplicates in A. In order to compute the equality function, we compute $\bigwedge_k a_k \oplus b_k$, using the fact that, for inputs $a, b \in \{0, 1\}$, `xor` can be computed as $(a-b)^2$, `and` directly via multiplication, and `not` as $1 - a$. If a duplicate is found, the element of B is multiplied with `unique == 0`, i.e., it does not contribute to the overall statistics.

**Performance & Discussion.** We evaluated both a naive baseline and the HECO-optimized batched implementation using the same setup described earlier in this section. The naive approach has a run time of several minutes (11.3 *min*) and requires the sending of over 2000 ciphertexts between the server and the clients. The batched solution produced by our system requires not only significantly less data to be transmitted (only 4 ciphertexts), but also runs an order of magnitude faster (57.6 *s*), confirming the trend we observed when evaluating on smaller benchmarks. While the results achieved by HECO are more than practical already, a state-of-the art hand-written implementation designed for this task can improve this even further, requiring only 1.4 *s*. However, arriving at this solution requires a significant rethinking of the program and an application-specific batching pattern, which HECO intentionally does not consider to avoid a search space explosion. Note that synthesis based tools such as Porcupine also cannot capture these kinds of transformations.

Specifically, instead of batching the identifiers for each database into a single ciphertext, the expert solution instead creates one ciphertext per bit, using significantly oversized ciphertexts with $128^2 = 2^{14}$ slots. This enables the expert solution to batch every possible permutation of the identifier set into one ciphertext. By applying this to the encryption of set B, while simply encrypting 128 non-permuted repetitions of set A, the expensive $\mathcal{O}\left(()\, n^2\right)$ duplication check can be performed in parallel on all elements at the same time. Computing the `unique` flag then uses a rare application of the rotate-and-*multiply* pattern. As a trade-off, the equality computation is no longer batched, but since the number of bits in the identifiers is, by necessity, at most logarithmic in the number of database elements, this is a profitable trade-off.

In general, exploiting application-specific packing patterns can unlock additional performance gains and is a frequently combined with client-side processing in expert-designed FHE systems. However, the decision on which client-side processing (e.g., removing outliers, computing permutations, etc) is sensible is not a well-defined problem that automated solutions can tackle. At the same time, the performance improvements demonstrated by HECO provide a first jump from the regime of prohibitive overheads to one of practical solutions. While further optimizations are likely frequently possible, they offer quickly diminishing returns. For example, scaling this application to real-world sizes (which, incidentally, is more complex with the expert approach) means that a naive solution might take days to compute while HECO's solution would complete in a few hours, which is a reasonable runtime for these kinds of secure statistics.

## 4.4 RELATED WORK

In this section, we briefly discuss related work in the domain of FHE compilation (Section 4.4.1) followed by a discussion of differences to existing MPC and ZKP compilers (Section 4.4.2).

### 4.4.1 *FHE Compilers*

The complexity of implementing FHE operations efficiently led to the development of dedicated libraries [102, 156] early on. Today, a large number of libraries [1, 55, 102, 139, 156] provide efficient implementations of state-of-the-art schemes. These libraries mostly provide comparatively low-level APIs that allow developers to extract the best possible performance but require significant expertise to utilize effectively. As a result, a first wave of FHE tools and compilers emerged that tried to improve the usability of FHE [34, 48, 71, 161, 163]. These mostly target a circuit-level abstraction and are focused on circuit optimizations [161].

For example, Microsoft's EVA [71] offers a user-friendly high-level interface and automatically inserts ciphertext maintenance operations into the circuit. EVA uses a custom circuit-based IR and requires developers to manually map their program to the FHE programming paradigm. In order to ease this process, recent versions [58] include a library of expert-implemented batched kernels for frequently used patterns, e.g., summing all elements in a vector. However, this still requires developers to manually transform an application to the batched paradigm.

A series of tools including Cingulata [34], E3 [48], SyFER-MLIR [97], and Google's Transpiler [96] attempt to translate arbitrary programs without the usual restrictions of FHE. They achieve this by translating their input programs into binary circuits, encrypting each input bit individually. However, programs translated in this way are virtually always too inefficient to be of practical use because they do not support the type of high-level transformations that HECO employs to achieve practical efficiency.

Domain-specific compilers [14, 16, 72], e.g., targeting encrypted Machine Learning applications, rely on a large set of hand-written expert-optimized kernels for common functionality (mostly linear algebra operations). Since these tools rely on pre-determined mappings rather than automatically identifying optimization opportunities, they do not transfer to other domains, such as the general-purpose setting HECO targets. Besides that, their lack of flexibility prevents their use when developers' needs are even slightly misaligned.

The Porcupine compiler [63] is closest to our work in that it also considers translating imperative programs to FHE's batching paradigm. However, their tool has a significantly different focus, using a heavy-weight synthesis approach that tries to identify optimal solutions that can outperform even state-of-the-art approaches used by experts. Since it explores a large state space in the search for an optimal solution, compile times tend to be long (up to many minutes) and programs can contain at most a handful of statements before the approach becomes infeasible. Additionally, Porcupine requires that developers provide a sketch of the structure of the batched program, making it less suitable for non-expert users.

Finally, we want to highlight that the MLIR framework is rapidly establishing itself as the gold standard for FHE tooling. Early attempts such as SyFER-MLIR [97] relied primarily on built-in optimizations, adding only a few binary-circuit-based FHE-specific rewrite rules. No evaluation is provided for SyFER-MLIR, but prior work studying similar simple rewrite rule-based tools [161] leads us to predict that it would produce only relatively minor speedups. More recently, however, a variety of concurrent work has successfully realized different aspects of the FHE ecosystem using

MLIR. For example, Zama's Concrete-ML [153] internally uses an MLIR-based compiler to translate Machine Learning tasks expressed as Numpy programs to the TFHE scheme. HECATE [123], meanwhile, improves upon the rescale-allocation optimizations presented in the EVA compiler [71]. However, where the latter uses a custom Python implementation that does not provide for interoperability with other tools, HECATE builds upon common MLIR abstractions. Thanks to the modular nature of MLIR, these tools could easily be integrated into HECO's end-to-end toolchain.

### 4.4.2 *MPC & ZKP Compilers*

Compilers for both Multi-Party Computation (MPC) and Zero-Knowledge Proof (ZKP) systems face similar challenges to FHE compilation, such as the need for data-independent computations and a general tendency towards trying to achieve small and low-depth circuits. However, in practice we find these similarities are too superficial to allow techniques from one domain to be lifted to another. Due to the heavy reliance on selectively revealing (potentially blinded) data during an MPC computation – a feature that has no direct correspondence in FHE – many of the optimization approaches are unlikely to transfer. State-of-the art MPC compilers [29, 104] also frequently make heavy use of hybrid approaches, i.e., switching between different MPC settings. While there has been significant work on scheme switching for FHE [18, 129], practical applications remain rare and few libraries currently support these techniques. As these approaches start to mature, investigating to what extent scheme-switching optimizations from the domain of MPC can transfer to FHE will present an interesting avenue for future work.

Zero-Knowledge Proof Compilers also face the challenge of mapping complex operations to arithmetic circuits with limited expressiveness. However, their setting fundamentally differs from that of FHE, as the prover generally has access to all data in the computation in the clear. This allows ZKP computations to heavily rely on witness-based computation, which allows compilers to shift virtually all non-arithmetic operations outside the core ZKP computation. Additionally, ZKP compilers mostly use intermediate representations based on Rank-1 Constraint Systems (R1CS) or other constraint specification systems that are not suitable for expressing FHE computations. Finally, we note that compiler frameworks trying to accommodate MPC, ZKP and potentially also FHE are emerging [146]. However, their reliance on a circuit-like IR makes them unsuitable for the high-level transformations we use in our work.

## 4.5 DISCUSSION

As FHE has emerged into practicality, it has drawn the interest of a significantly wider audience bringing new perspectives, requirements and backgrounds to the area. While traditionally, FHE applications were mostly developed by the same experts that designed, optimized and implemented the underlying cryptographic schemes, this will soon no longer be true beyond the world of cutting-edge academic research. In recent years, a variety of tools has emerged in an attempt to address the needs of future non-expert developers. While some domain specific tools have proven to be very effective [16, 56, 161], most general purpose tools have fallen short of delivering on the promise of usable FHE. While they simplify the development process, they generally produce naive implementations which provide little real-world benefit due to their significant overhead compared to more optimal implementations. However, as performance is key for practical deployment, we believe that usability without sufficient performance is mostly meaningless.

HECO aims to bridge the gap between usability and performance for general-purpose workloads. It offers non-expert developers the ability to express applications in a familiar high-level paradigm *without* paying the extreme performance penalty this would usually incur. Beyond optimizing this high-level transformation, HECO proposes a new end-to-end architecture for FHE compilers based on the distinct stages of FHE optimization we identify. HECO's modular architecture is designed to allow it to interoperate with other toolchains and easily integrate future optimization techniques. While HECO represents an important step in FHE usability, many challenges remain to be addressed. For example, HECO considers RLWE-based schemes offering SIMD operations, but recent developments in LWE-based fast-bootstrapping schemes makes them an attractive alternative. Today, these worlds remain mostly separate and use significantly different paradigms. Future work needs to consider how to unify these, especially in the context of scheme-switching, i.e., the ability to move between schemes inside a single application. Finally, upcoming dedicated FHE hardware accelerators promise to deliver significant performance improvements but require sophisticated scheduling to unlock their potential. While existing work on accelerators already incorporates automated scheduling, there are likely significant further optimization opportunities in considering compilation for these systems from an end-to-end perspective. More generally, we believe that there is significant potential for interdisciplinary research that combines techniques from compiler and programming language research with insights from cryptography.

# 5 VERIFIABLE FULLY HOMOMORPHIC ENCRYPTION

Computing on encrypted data inherently requires malleable ciphertexts (e.g., the addition of two ciphertexts is also valid ciphertext). However, this malleability also raises the issue of *integrity*, as the server can deviate from the computation requested by the client. This has obvious implications for correctness but can also have more severe consequences: a malicious server can exploit the malleability of FHE to carry out key-recovery attacks [37, 43, 54, 79, 167], undermining the confidentiality of FHE. So far, most work on FHE schemes and applications has chosen to side-step this issue by making assumptions on the setting and threat model. However, as FHE is starting to be deployed to protect critical information, we must move beyond these assumptions to a threat model that can withstand real-world adversaries.

Historically, the FHE research community has made extensive use of the assumption that the server running an FHE application would be honest-but-curious, rather than actively malicious [22, 52, 74, 78]. This assumption may be reasonable in some deployment scenarios (e.g., when FHE is used only to ensure regulatory compliance or when dealing with trusted institutions cooperating on their own data). However, the necessity to trust the server to this extent is very limiting to the scope of application scenarios, since a violation of the assumption threatens not only correctness but also confidentiality. In addition, even otherwise trusted parties can be compromised by malicious third parties, exposing this attack surface. While FHE protects against passive attacks, a malicious or compromised server taking part in an FHE application can leverage this to undermine data confidentiality (c.f. Section 5.1.3). In order to remediate these attacks, a line of research has emerged that constructs more robust FHE schemes that achieve indistinguishability against chosen ciphertext attacks (IND-CCA1). These schemes remain secure even in the presence of decryption oracles. Unfortunately, many of these constructions assume the presence of cryptographic primitives even stronger than FHE and/or are too inefficient to implement in practice. A different line of research focuses on achieving integrity for FHE; guaranteeing a function was correctly executed on the ciphertext while preserving the confidentiality of inputs. We review and discuss these approaches, showing that there is a significant gap between the assumptions made by existing work and the way state-of-the-art FHE schemes are used in practice.

This chapter presents a taxonomy of existing approaches, highlighting the gap between their assumed settings and real-world FHE deployments. Based on these insights, we then define a new notion of integrity for FHE that captures real-world FHE deployment settings, addressing the issues we identified in our analysis. We show how to generically construct our notion from a standard FHE scheme, commitments, and Zero-Knowledge Proofs (ZKPs). Finally, we instantiate our construction using a range of state-of-the-art ZKP protocols and propose a series of optimizations to improve the efficiency of verifiable FHE in practice.

## 5.1 ANALYSIS OF FHE INTEGRITY CONSTRUCTIONS

In this section, we aim to answer the question: *to what extent do the recent developments in FHE integrity address the needs of real-world FHE deployments, and where and why do they fall short?* Recently, a number of approaches for FHE integrity have been proposed, covering a variety of settings and introducing a plethora of subtly different notions and properties. In this section, we are the first to holistically analyze FHE integrity across the boundaries of the different approaches. Towards this, we unify the existing constructions into a set of general paradigms that form a taxonomy of FHE integrity approaches. This allows us to consider the strengths and weaknesses of the proposed notions independently of individual instantiations. We then show how these notions fall short when considering how FHE is used in practice, highlighting the mismatch between the setting assumed in the existing integrity literature and the settings used for the vast majority of FHE applications. Finally, we show how this enables attacks on both correctness and confidentiality even in the presence of these integrity mechanisms.

### 5.1.1 *Taxonomy of FHE Integrity Paradigms*

We provide a complete taxonomy of all the literature on FHE integrity that we are aware of at the time of writing. We analyze them according to the underlying techniques, grouping them into Message-Authentication-Code–, Zero-Knowledge-Proof– and Attestation–based approaches. We discuss the existing constructions at a level of abstraction that allows us to directly compare them and focus on their suitability for practical FHE deployments. We provide a summary of our analysis in Table 5.2 on page 93.

**Homomorphic Message Authentication Codes.** Message Authentication Codes (MACs) have long been used to ensure integrity for traditional symmetric-key encryption. A Message Authentication Code (MAC) is a short unforgeable tag used to verify the authenticity and integrity of a message. They are generated using a secret MAC key and then sent along with the corresponding message. Note that verifying a MAC also requires access to the MAC key. While MACs are usually intentionally non-malleable, *homomorphic* MACs allow for some operations similar to homomorphic encryption. In the context of FHE, special (fully) *homomorphic* MACs are required, so that the server can combine valid MACs on the inputs to an FHE operation to a valid MAC of the output. Existing constructions fall into three different paradigms based on how they combine the MAC and the underlying FHE scheme: *(i)* Encrypt-and-MAC (EaM), *(ii)* Encrypt-then-MAC (EtM), and *(iii)* MAC-then-Encrypt (MtE).

In the Encrypt-and-MAC (EaM) paradigm, the initial MAC and FHE ciphertext are computed from the same plaintext and are then processed in parallel (but independently) to produce the output MAC and ciphertext pair. Since the MAC in EaM is not encrypted under FHE, it must itself provide strong security, i.e., be semantically secure. In addition, the MAC must offer the same homomorphic operations as the underlying FHE scheme. Li et al. construct such an EaM scheme using multilinear maps, assuming a generic FHE scheme [126]. While it describes how to support addition and multiplication operations, it is not clear how this scheme can be extended to handle the complex ciphertext maintenance operations (e.g, relinearization) that are necessary for modern FHE schemes to achieve state-of-the-art efficiency. In general, it is unclear how to expand the expressiveness of homomorphic MACs while maintaining the strong security guarantees required for the EaM approach.

The Encrypt-then-MAC (EtM) approach firsts encrypts the plaintext using FHE and then applies the MAC to the resulting ciphertext. This removes the need for the MAC to preserve confidentiality, but the MAC does still need to be homomorphic with respect to operations on ciphertexts (including ciphertext maintenance operations). Fiore et al. make use of this paradigm in [82], instantiating their MAC using pairings. In order to enable this, they needed to introduce a homomorphic hash function to bridge the gap between FHE ciphertexts and the MACs, i.e., polynomial rings and pairing groups. In order to achieve efficient verification, they also rely on amortized closed-form efficient PRFs [12]. However, the combination of these primitives limits the expressiveness of the resulting construction. Specifically, it only supports quadratic circuits, i.e., circuits with at most one multiplication gate. In general, it is unclear whether it is possible to create MACs that support both arbitrarily deep circuits and complex operations on the ciphertexts.

Finally, the MAC-then-Encrypt (MtE) paradigm first computes a MAC over the plaintext and then encrypts the MAC-augmented plaintext under FHE. This removes the need to support ciphertext maintenance operations, as these do not affect the encrypted message. Gennaro and Wichs [89] provided one of the first FHE integrity construction based on this paradigm. However, the construction is not efficiently verifiable, i.e., verifying the MAC requires recomputation that is as expensive as computing the original result. The work proposed potential ways to solve this issue, but did not instantiate a solution. Catalano and Fiore [35] addressed the verification efficiency, but in turn their construction is limited to arithmetic circuits of a bounded depth. More recently, Chatel et al. [36] have generalized these two approaches, providing the first FHE integrity scheme that can efficiently support arbitrary circuits and modern state-of-the-art schemes.

**Zero-Knowledge Proofs.** A Zero-Knowledge Proof (ZKP) is a protocol that allows a *prover* to convince a *verifier* of the truth of a mathematical statement without revealing additional information. Non-interactive ZKPs allow the prover to generate a *proof* that the verifier can check independently, and there exist techniques to turn any ZKP non-interactive. In the context of integrity, ZKPs can be used to show that $y = f(x, w)$ for a given $x, y$ and $f$ without revealing $w$.

Zero-Knowledge Proofs are a natural primitive to explore for FHE integrity constructions. In this approach, the server first computes the FHE circuit, storing intermediate ciphertext results, and then computes a Succinct Non-interactive ARgument of Knowledge (SNARK) that asserts that the server knows an assignment of intermediate values to the circuit so that, for the given input, the circuit results in the output ciphertext. In theory, any generic ZKP system could be used to generate this proof. However, a trivial instantiation would introduce prohibitively large additional overhead in emulating the complex ring operations used in FHE schemes. Recent work has therefore focused instead on developing ZKP systems tailored to FHE.

One line of work uses (homomorphic) hashing to bring the size of FHE ciphertexts down into a range that can be handled more efficiently with ZKP techniques. This includes the first SNARK for FHE presented by Fiore et al. [83] and follow-up work by Bois et al. [17]. However, the homomorphic hashing requirement limits this approach to simple schemes such as the BV scheme [24] which does not feature the complex ciphertext maintenance operations that are necessary to achieve the practical efficiency enjoyed by state-of-the-art FHE schemes. An alternative approach by Ganesh et al. [85] instead focuses on constructing a generic ZKP system that natively operates on the rings used in FHE schemes. This drastically improves the efficiency of proving the ring operations that make up basic homomorphic operations such as addition and multiplication. However, ciphertext-maintenance operations generally require either switching between different rings or non-ring operations such as rounding, which are not supported by the current construction. While such tailored approaches hold the promise of significant performance improvements, they cannot currently support efficient modern state-of-the-art FHE schemes. Therefore, when we consider FHE integrity in practice in Section 5.3, we will focus on generic ZKP systems and discuss how to efficiently instantiate it for FHE.

**Trusted Execution Environment Attestation.** Trusted Execution Environments (TEEs) are hardware components capable of isolating code running on them from the rest of the machine. Code running on a TEE cannot be tampered with by other processes, even the operating system or hypervisor. TEEs are commercially available in commodity hardware provided by all major hardware vendors [5, 87, 109]. TEEs are frequently used to provide confidentiality from either the server operators or other VMs running on the same hardware, but they can also provide integrity through code *attestation*, which generates a certificate that the outputs were generated through a correct execution of the program.

Trusted Execution Environments (TEEs) such as Intel SGX [109] can be used to provide confidentiality, but a series of attacks [80, 142] has put their suitability for this task in question. However, their integrity protections, i.e., their ability to *attest* to the program running in the enclave, have so far mostly resisted practical attacks [140]. Therefore, it is natural to augment the confidentiality properties of FHE with the integrity protections of TEEs by running FHE inside an enclave. However, the computational complexity of FHE and especially the large sizes of ciphertexts and evaluation keys pose a challenge to TEEs, which are usually more restricted in terms of memory and available computational power than the underlying untrusted hardware. More fundamentally, they can only be employed in settings where the additional trust assumption on the specific hardware vendor is acceptable. Natarajan et al. [141] present an FEE-in-TEE design and implementation, that, because of the ability of TEEs to express arbitrary computations, can easily support modern state-of-the-art FHE schemes with little to no required modifications.

|  |  | Ctxt. Maint. | Circuit | Srv. Inputs | Srv. Privacy | Approx. FHE | Adversarial Model |  | Impl. |
|---|---|---|---|---|---|---|---|---|---|
|  |  |  |  |  |  |  | Verif. Oracles | Dec. Oracles |  |
|  | [89] | ? | Any | ○ | ○ | ○ | ◑ | ○ | ○ |
| MtE | [35] | ? | Any | ○ | ○ | ○ | ● | ○ | ○ |
|  | [36] | ● | Any | ○ | ○ | ○ | ○ | ○ | ● |
| EtM | [82] | ○ | Quadratic | ○ | ○ | ○ | ○ | ○ | ● |
| EaM | [126] | ? | Any | ○ | ○ | ○ | ○ | ○ | ○ |
|  | [83] | ○ | Any | ○ | ● | ○ | ● | ○ | ○ |
| ZKP | [17] | ○ | LogspaceUnif | ● | ● | ○ | ● | ○ | ○ |
|  | [85] | ◑ | Any | ● | ● | ○ | ○ | ○ | ◑ |
| TEE | [141] | ● | Any | ● | ○ | ● | ● | ◑ | ◑ |

Table 5.2: Characteristics and limitations of existing FHE integrity paradigms and approaches. A ? indicates insufficient details are given while ◑ indicates partial support.

### 5.1.2 *FHE Integrity in Practice*

In this section, we consider to what extent the assumptions and guarantees proposed in the existing FHE integrity literature fulfill the requirements of real-world FHE deployments. We highlight the differences between the deployment settings assumed by the existing literature and the needs of real-world FHE deployments, uncovering significant mismatches. In the following (Sections 5.1.3 and 5.1.4), we discuss the implications of this mismatch for correctness and confidentiality.

**FHE Deployment Settings.** The existing literature on FHE integrity assumes the *outsourced computation* setting, where a client provides an encrypted input $x$ and a function (or circuit) $f$ to the server, which then computes $f(x)$ homomorphically and returns the encrypted result. While this setting is the most natural to define FHE in, it is not the only setting or even the most widespread one. In practice, the server has the ability to both choose the circuit to compute and to provide additional inputs. This opens up a variety of important additional use cases that enable a form of two-party computation. For example, this can be used to offer privacy-preserving Machine Learning as a Service (MLaaS) where the server has a model that it wants to make available as a service and the client wants to receive a homomorphically computed inference on its private input [133, 150, 165].

These real-world deployment scenarios fundamentally change the attack surface which we need to consider. Public and private server inputs have significant implications for integrity, with the latter posing more fundamental challenges. Nevertheless, even public inputs prevent the use of some integrity approaches, such as MAC-based solutions [35, 36, 82, 89, 126]. This is inherent in the concept of MACs, which only the client can generate for fresh messages. On the other hand, TEE- and ZKP-based approaches can usually support public inputs with straightforward extensions, even though these are not considered in the existing literature. Since these inputs are public, they can be transmitted to the client along with the result and proof (or attestation) of correctness, allowing them to complete the verification. Private server inputs, on the other hand, are more challenging because they raise fundamental questions about the meaning of correctness in their presence. For example, in the privacy-preserving MLaaS setting, it is not immediately clear what statements on the correctness of the computation can be made without revealing the model to the client. This highlights a fundamental gap between the *application-level* correctness that we usually want to achieve and the *circuit-level* correctness that existing integrity notions work on. We discuss this challenge in further detail in Section 5.1.3. More importantly, and perhaps unexpectedly, the ability to introduce private inputs opens up a new attack surface and allows a malicious server to undermine the *confidentiality* of FHE. Intuitively, these issues arise because the private nature of the inputs allows the server to choose malformed inputs that 'poison' the output in a way that will lead to a decryption failure at the client, which the server can observe through the client's reaction. While TEE- and ZKP-based approaches can technically be extended to support private inputs, straightforward extensions are ineffectual in preventing such attacks. We discuss these issues in more detail in Section 5.1.4.

### 5.1.3 *Attacks on Correctness*

The ability of the server to provide inputs opens up a new attack surface since integrity issues can now arise not only from deviations in the computation but also from malicious inputs. In this setting, the server can produce 'incorrect' results even when executing the circuit 'correctly,' allowing them to produce a valid proof of correctness for these 'incorrect' results. We identify two different cases: first, the computation can result in an *invalid ciphertext* that fails to decrypt correctly. Second, the computation can return a valid ciphertext but nevertheless fail to satisfy *application-level* expectations.

**Invalid Ciphertexts.** The noise inherent in FHE ciphertexts grows during computation and, for all but trivial circuits, must be managed carefully using ciphertext-maintenance operations. When the server inputs a ciphertext[1] with larger noise than expected, the noise will overflow and garble

1 This attack still works when the server provides a plaintext, since message size also impacts noise growth.

the message. As a result, decryption will no longer return the correct result. Note, however, that this is not necessarily observable by the client: the decryption will return an essentially random value, but without application-level constraints, this is indistinguishable from a correct result. Existing integrity notions do not consider this aspect since they are designed for the outsourced computation setting, where all inputs are from the client and can therefore be assumed to be well-formed. As a result, the server can provide an incorrect result together with a proof of 'correctness,' clearly undermining the idea of integrity. More importantly, this also has severe implications on confidentiality, which we discuss in Section 5.1.4.

**Application-Level Correctness.** Using existing integrity notions, a proof of 'correctness' only guarantees that the result is the output of the specified circuit applied to the client's input and *some* input provided by the server. When the server inputs are public, the client can verify that they are well-formed and meaningful. However, when the inputs are private, which is necessary for most applications in the two-party setting, this is no longer possible. This allows the server to affect the correctness of the result on an application level. For example, consider a Private Information Retrieval (PIR) application, where a client wants to run a private query against a database stored on the server. The server could compromise application-level correctness by censoring individual entries in the database to further some malicious objective. For example, using existing integrity notions, the server could hide specific Wikipedia articles or individual transactions from a blockchain history while still providing a proof of 'correctness'.

### 5.1.4 *Attacks on Confidentiality (Key Recovery)*

In addition to impacting correctness, the mismatch between existing notions and real-world requirements causes a considerably more serious issue: a malicious server can exploit the interactive nature of practical FHE deployments to compromise confidentiality. In fact, using *key-recovery attacks*, a server could potentially recover the client's full secret key. These attacks fall outside the scope of the semi-honest–server setting assumed by FHE schemes but should be addressed in a setting where the server is not trusted to execute the computation faithfully, such as that considered by integrity notions. Existing integrity notions for FHE, however, fail to do so.

**Key Recovery Attacks.** Intuitively speaking, Key-Recovery Attacks exploit the fact that the decryption operation combines the ciphertext and the secret key. While a decryption of a valid ciphertext will only ever output the encrypted message, a malformed ciphertext can result in (parts of) the secret key being returned instead. For example, we briefly outline a simple key recovery attack by Chenal and Tang [43] against the BV scheme [24]. In BV, decryption is defined as $\mathsf{Dec}_{\mathsf{sk}}(ct) = [ct_0 + ct_1 \cdot \mathsf{sk}]_t$, where the inner operations are performed over $R_q$ (see [24] for details). When decrypting the special ciphertext $ct = (0, 1)$, one trivially recovers the secret key $\mathsf{Dec}_{\mathsf{sk}}(ct) = [0 + 1 \cdot \mathsf{sk}]_t = [\mathsf{sk}]_t = \mathsf{sk}$ (under some assumptions on the parameters; we refer to [43] for the details). In this simple attack, the client can easily detect that this ciphertext has been maliciously crafted to be a (trivial) encryption of the secret key. However, FHE also features encryptions of the secret key that are indistinguishable from standard ciphertexts. For example, key-recovery attacks can take advantage of the *evaluation keys* provided to the server in most schemes. These are encryptions of (functions of) keys, which allow the server to perform crucial ciphertext maintenance operations (e.g., relinearization, key-switching, bootstrapping). If given access to a decryption oracle, an adversary could decrypt these evaluation keys and recover the secret key. For example, we present how to exploit the relinearization key used in the BGV scheme. We recall that BGV [22] uses a relinearization key $\mathsf{rk} = (a \cdot \mathsf{sk} + t \cdot e + \mathsf{sk}^2, -a)$, which is a valid encryption of $\mathsf{sk}^2$ under $\mathsf{sk}$. Using a decryption oracle, we can recover $\mathsf{sk}^2$ and learn information about $\mathsf{sk}$.

**Beyond Decryption Oracles.** Full decryption oracles that would allow an adversary to exploit the straightforward attacks we discussed above are not commonplace in practice. However, more sophisticated attacks [54] do not require such strong oracles and instead target decryption *failure* oracles. These oracles arise when the server manipulates the result to have noise overflow, leading to an invalid decryption. When the client detects and reacts to this unexpected result (e.g., by requesting a re-run of the computation or aborting further interactions), this leaks information to the adversary. Because most FHE application scenarios involve multiple queries to the server (e.g., MLaaS, PIR, etc), avoiding these oracles is frequently impossible in practice. Common patterns in FHE applications frequently allow the server to have full control over the shape of the ciphertext result, even in the presence of existing integrity notions. For example, consider $f(x, w_1, w_2) := \langle x; w_1 \rangle + w_2$, which appears when implementing a (biased) matrix-vector multiplication (this is a common pattern in ML applications). The adversary can choose $w_1 = 0$ and an arbitrary $w_2$ as inputs, and prove that $f(x, w_1, w_2) = w_2$ for any $x$ provided by the client. However, these attacks usually do not even require full control of the output: because state-of-the-art FHE applications are heavily optimized for efficiency, they usually have very limited additional noise capacity, so simply providing any outsized inputs is likely to cause a noise overflow. Since existing integrity notions only consider the correct execution of the operations in the circuit, they do not address this issue. Although some provide what appear to be strong guarantees of input privacy [17, 83] that one might expect to exclude such attacks, even these do not prevent such attacks. This is because their privacy guarantees only hold against a much weaker adversary limited to verification oracles (i.e., informing the adversary whether an attempted proof passed verification). However, since decryption (failures) oracles are essentially impossible to avoid, any system trying to achieve FHE integrity in the real world must consider them. In the following section, we, therefore, introduce a new robust notion of FHE integrity that is specifically designed to accommodate real-world deployment settings and threat models.

## 5.2 MALICIOUSLY-SECURE VERIFIABLE FHE

In this section, we define a new notion of integrity for FHE. Specifically, it captures real-world FHE deployment settings, including those with actively malicious server inputs. One of the key insights of our notion is that we consider a stronger and arguably more realistic threat model. Existing notions result in an inconsistent view of the server, which is both expected to deviate from the computation (otherwise, integrity notions would not be necessary) yet, at the same time, assumes that the server will not use these deviations to exploit common decryption (failure) oracles that arise in real-world FHE to undermine confidentiality. This can result in a false sense of security, leading users of constructions achieving these notions to believe that they have stronger protections than these notions in fact offer. Because of this inconsistency, we believe that the natural threat model beyond the semi-honest server assumption should be an actively malicious server with full access to a decryption oracle. In addition, we also consider an unbounded verification oracle in order to give similar strength to the integrity guarantees. Finally, we discuss how to generically construct a vFHE scheme from an IND-CPA secure FHE scheme, commitments, and a compatible ZKP system before studying how to instantiate this construction efficiently in the next section (Section 5.3).

### 5.2.1 *Defining Maliciously-Secure Verifiable FHE*

We present a natural notion of verifiable FHE that cleanly composes FHE with integrity properties. This is in contrast to existing notions which are monolithic combinations of existing integrity notions (e.g., MAC unforgeability [126] or Verifiable Computation [82]) with ad-hoc confidentiality properties. As a result, they generally interleave FHE and integrity aspects, which makes them hard to reason about or extend. We propose a modular framework that allows us to easily define extensions of the core notion, enabling us to support a wide range of FHE deployment settings with varying requirements. We first provide a definition of Fully Homomorphic Encryption that is sufficiently detailed to express important details of real-world deployments of FHE. We then present the core definition of a maliciously secure verifiable FHE scheme (vFHE) and present the soundness, completeness, and security properties it must fulfill. Then, we describe how to extend this notion with privacy for the server inputs, as well as with input predicates.

**Definition 5.2.1 (FHE)**
*A* Fully Homomorphic Encryption *(FHE) scheme for a class of circuits*[2] $\mathcal{F} := \{f : \mathcal{M}^{k+l} \to \mathcal{M}^m \mid k, l, m \in \mathbb{N}\}$*, is a tuple of PPT algorithms* $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Dec})$*:*

- $\mathsf{KGen}(1^\lambda) \to (\mathsf{pk}, \mathsf{sk})$ *where $\lambda$ is the security parameter.*
- $\mathsf{Enc}_{\mathsf{key}}(x) \to c_x \in \mathcal{C}_{\mathsf{key}} \subseteq \mathcal{C}$ *for $x \in \mathcal{M}$, where* key *is either* sk *or* pk[3]
- $\mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}) \to \vec{c_y} \in \mathcal{C}^m_{out} \subseteq \mathcal{C}^m$ *for $f \in \mathcal{F}$, $\vec{c_x} \in \mathcal{C}^k_{in}$ and $\vec{w} \in \mathcal{M}^l$*
- $\mathsf{Dec}_{\mathsf{sk}}(c_y) \to y \in \mathcal{M} \cup \{\perp\}$*, for $c_y \in \mathcal{C}$*

*where $\mathcal{M}$ is the message space*[4] *and $\mathcal{C}$ is the ciphertext space (with $\mathcal{C}_{in} \subseteq \mathcal{C}$). The scheme must satisfy the* correctness, *(relaxed)* compactness, *and (IND-CPA)* security *properties defined below.*

We define both exact and *approximate* correctness. Informally, a scheme is correct if any honest computation will decrypt to the expected result. For approximate correctness, a slight error $\varepsilon$ between the decryption and the expected results is accepted. A scheme that achieves the latter, but not the former, is said to be an *approximate FHE* scheme. In the following, we will use $\mathsf{Enc}_{\mathsf{key}}(\vec{x}) := [\mathsf{Enc}_{\mathsf{key}}(x_1), \ldots, \mathsf{Enc}_{\mathsf{key}}(x_k)]$ for $\vec{x} \in \mathcal{M}^k$ and $\mathrm{Dec}_{\mathsf{sk}}(\vec{c}) := [\mathrm{Dec}_{\mathsf{sk}}(c_1), \ldots, \mathrm{Dec}_{\mathsf{sk}}(c_k)]$ for $\vec{c} \in \mathcal{C}^k$ for conciseness.

**Definition 5.2.2 (Correctness)**
*A scheme is* correct *if for all circuits $f \in \mathcal{F}$, and for all $\vec{x}$ and $\vec{w}$ so that their concatenation $\overrightarrow{xw}$ is in the domain of $f$:*

$$\Pr\left[\mathsf{Dec}_{\mathsf{sk}}(\vec{c_y}) = f(\overrightarrow{xw}) \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ \vec{c_x}, \leftarrow \mathsf{Enc}_{\mathsf{key}}(\vec{x}) \\ \vec{c_y} \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}) \end{array}\right] = 1$$

2 A circuit is a DAG where leaf nodes are labeled as inputs or outputs, and internal nodes represent operations. We slightly abuse notation in the following and use arrow notation to describe the sets of possible inputs and outputs for circuits.

3 Note that we can black-box construct a public-key scheme from a secret-key FHE scheme.

4 Modern FHE schemes distinguish between message and plaintext spaces which are frequently, but not always, isomorphic. If the mapping is non-surjective, not all ciphertexts decrypt to a valid message, and Dec will return $\perp$.

**Definition 5.2.3 ($\epsilon$-Correctness)**
*A scheme is* approximately *(or $\varepsilon$-)*correct *if for all circuits $f \in \mathcal{F}$, and for all $\vec{x}$ and $\vec{w}$ so that their concatenation $\overrightarrow{xw}$ is in the domain of $f$:*

$$\Pr\left[ \|\mathsf{Dec}_{\mathsf{sk}}(\vec{c_y}) - f(\overrightarrow{xw})\| \leq \varepsilon \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ \vec{c_x}, \leftarrow \mathsf{Enc}_{\mathsf{key}}(\vec{x}) \\ \vec{c_y} \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}) \end{array} \right] = 1$$

*where $\|\cdot\|$ is a scheme-specific norm, and $\varepsilon$ is a scheme-specific upper bound on the decoding error (which may depend on $f$, $\mathsf{pk}$, or other quantities of the scheme).*

Without the requirement of *compactness*, a definition of FHE would allow trivial constructions that simply concatenate the circuit description to the input ciphertext and perform the evaluation as part of the decryption. Therefore, we traditionally require the output of $\mathsf{Eval}_{\mathsf{pk}}(f, \ldots)$ to have a size independent of the size of $f$ (more formally, Dec must be implementable by a circuit of size polynomial in $\lambda$). However, relaxations of this property are common in practice. For example, leveled FHE is frequently only *quasi-compact*, with the size of the keys (and therefore, Dec) dependent on the depth of the circuit. When considering integrity guarantees, as we will do next, the need to verify the correctness of a computation requires that Dec take (a description of) the input ciphertexts used to compute a result. Therefore, we further relax compactness to allow a dependence on the number of inputs of $f$ (but not the size of $f$ more generally). Specifically, we use the notion of *compactness w.r.t. circuit complexity* by Canetti et al. [31].

**Definition 5.2.4 (Compactness [31])**
*A scheme is* compact *(w.r.t. circuit size) if there exists a polynomial $p$ so that the size of the ciphertexts in $\vec{c_y} \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w})$ is bounded by $p(\lambda, k, l)$ for all $f \in \mathcal{F}, \vec{c_x} \in \mathcal{C}_{in}^k, \vec{w} \in \mathcal{M}^l$. In particular, $p$ is independent of the size of the circuit.*

We restate the traditional IND-CPA security assumption, i.e, a PPT adversary with access to the usual encryption oracle $\mathcal{O}_{\mathsf{Enc}}(x) := \mathsf{Enc}_{\mathsf{key}}(x)$. Note that we do not require an evaluation oracle since $\mathsf{Eval}_{\mathsf{pk}}$ is public and can be executed by the adversary directly. For split adversaries $\mathcal{A} = (\mathcal{A}_1, \ldots, \mathcal{A}_n)$, we implicitly allow the adversary to retain state between the invocation of $\mathcal{A}_i$ and $\mathcal{A}_{i+1}$, unless specified otherwise.

**Definition 5.2.5 (IND-CPA Security)**
*A scheme is* IND-CPA secure *if for any PPT adversary $\mathcal{A}$ the advantage* $\mathsf{Adv}^{\text{IND-CPA}}[\mathcal{A}](\lambda) = 2\left|\Pr\left[b = \widehat{b}\right] - \frac{1}{2}\right|$ *of the attacker in the following game is negligible in the security parameter $\lambda$:*

```
IND-CPA for FHE
$(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda)$
$(m_0, m_1) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Enc}}}(1^\lambda, \mathsf{pk})$
$(c^*, \tau^*) \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_b)$
$\widehat{b} \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Enc}}}(c^*)$
```

Our definition of FHE can express both "true" FHE (where $\mathcal{F}$ can contain arbitrary circuits) and leveled FHE, by limiting $\mathcal{F}$ to circuits of depth at most $L$. Note that, like most definitions of FHE, ours is not inherently composable. Specifically, it is possible that $\mathcal{C}_{out} \cap \mathcal{C}_{\mathsf{in}} = \varnothing$. For example, in leveled FHE, we might want to set $\mathcal{C}_{\mathsf{in}} = \mathcal{C}_{\mathsf{pk}} \cup \mathcal{C}_{\mathsf{sk}}$, i.e., allow only fresh encryptions as input.

**Definition 5.2.6 (Verifiable FHE)**
*A* verifiable *Fully Homomorphic Encryption (vFHE) scheme for a class of circuits $\mathcal{F} := \{f : \mathcal{M}^{k+l} \to \mathcal{M}^m \mid k, l, m \in \mathbb{N}\}$ is an FHE scheme for which there exists a PPT algorithm* Verify*:*

- $\mathsf{Verify}_{\mathsf{sk}}(f, \vec{c_y}) \to \{0, 1\}$ *for* $f \in \mathcal{F}$, $\vec{c_y} \in \mathcal{C}$

*and which satisfies the* IND-CCA1 security, completeness, *and* soundness *properties defined below.*

In addition to the encryption oracle $\mathcal{O}_{\mathsf{Enc}}(x) := \mathsf{Enc}_{\mathsf{key}}(x)$, the adversary has access to an unbounded verification and decryption oracle $\mathcal{O}_{\mathsf{Dec}}(c) := b$ which returns $b = \perp$ if $\nexists f, \vec{c}$ s.t. $c \in \vec{c} \wedge \mathsf{Verify}_{\mathsf{sk}}(f, \vec{c}) = 0$ and $b = \mathsf{Dec}_{\mathsf{sk}}(c)$ otherwise.

**Definition 5.2.7 (IND-CCA1 Security)**
*A scheme is* IND-CCA1 secure *if for any PPT adversary $\mathcal{A}$ the advantage* $\mathsf{Adv}^{\text{IND-CCA1}}[\mathcal{A}](\lambda) = 2\left|\Pr\left[b = \widehat{b}\right] - \frac{1}{2}\right|$ *of the attacker in the following game is negligible in the security parameter $\lambda$:*

$$\begin{array}{l}
\underline{\text{IND-CCA1 } \textit{for vFHE}} \\
(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\
(m_0, m_1) \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Enc}}, \mathcal{O}_{\mathsf{Dec}}}(1^\lambda, \mathsf{pk}) \\
(c^*, \tau^*) \leftarrow \mathsf{Enc}_{\mathsf{pk}}(m_b) \\
\widehat{b} \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Enc}}}(c^*)
\end{array}$$

**Definition 5.2.8 (Completeness)**
*A scheme is* complete *if for all circuits $f \in \mathcal{F}$, and for all $\vec{x}$ and $\vec{w}$ so that their concatenation $\overrightarrow{xw}$ is in the domain of $f$:*

$$\Pr\left[\mathsf{Verify}_{\mathsf{sk}}(f, \vec{c_y}) = 1 \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ \vec{c_x} \leftarrow \mathsf{Enc}_{\mathsf{key}}(\vec{x}) \\ \vec{c_y} \leftarrow \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}) \end{array}\right] = 1$$

**Definition 5.2.9 (Soundness)**
*A scheme is* sound *if for any PPT adversary $\mathcal{A}$ the following probability is negligible in the security parameter $\lambda$:*

$$\Pr\left[\begin{array}{c} \mathsf{Verify}_{\mathsf{sk}}(f, \vec{c_y}) = 1 \\ \wedge \\ \nexists \vec{w} \in \mathcal{M}^l s.t. \mathsf{Dec}_{\mathsf{sk}}(\vec{c_y}) = f(\overrightarrow{xw}) \end{array} \;\middle|\; \begin{array}{c} (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ x, f \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Enc}}, \mathcal{O}_{\mathsf{Dec}}}(\mathsf{pk}) \\ \vec{c_x} \leftarrow \mathsf{Enc}_{\mathsf{key}}(x) \\ \vec{c_y} \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Enc}}, \mathcal{O}_{\mathsf{Dec}}}(\vec{c_x}) \end{array}\right]$$

We present our notion with *designated verifiability*, i.e., some secret key material might be required to verify a ciphertext. We believe that this is natural for the FHE setting, where only the secret key holder can decrypt a ciphertext and therefore has any benefit from verifying it. Note that even in a threshold FHE or multi-key FHE setting, designated verifiability is sufficient, since the key holders can extend the MPC protocol they run to decrypt to also realize verification.

Our approach allows us to detangle confidentiality and integrity guarantees. In contrast, correctness and completeness (i.e., verifiability) are usually combined in existing notions. This allows us to reason about constructions more cleanly and also allows us to easily define extensions of our core notion. For example, our notion trivially extends to *approximate* verifiable FHE, and we are the first to extend integrity notions to this setting. Because we split correctness and completeness, changing between verifiable FHE and verifiable approximate FHE has no knock-on-effects on the remainder of the notion. In the next section, we discuss how to extend our notion to support private server inputs and input predicates.

### 5.2.2 *Server Input Privacy and Input Predicates*

Verifiable FHE as defined in Definition 5.2.6 addresses key-recovery attacks and ensures that client inputs are protected as expected. However, it does not address the privacy of server inputs. We therefore provide an extension of the notion which can be used in settings where formal guarantees for server privacy are required. Informally speaking, it requires that the result of an evaluation reveals nothing to the client beyond the output of the function. Existing work that considers hiding server inputs does not explicitly state the threat model for this setting. We address this issue and define an adversary model that considers the client as the adversary. For example, considering the perspective of the client, the indistinguishability has to hold even when the adversary has access to the secret key. We assume that the client generates keys and encrypts honestly, i.e., we assume a semi-honest client. We could strengthen the threat model at the cost of requiring proofs of correct key generation and encryption. However, we believe that this would be prohibitively expensive and not appropriate for most settings. We note the parallels between this setting and generic 2-party MPC. However, in MPC, parties usually have equal protection. Here, the guarantee offered to the client is stronger, since the server never learns the output of the function and therefore has *no* information on the client input, not even that which would be derivable from the function output.

**Definition 5.2.10 (vFHE with Server Input Privacy)**
*A verifiable FHE scheme with* server input privacy *for a class of circuits* $\mathcal{F} := \{f : \mathcal{M}^{k+l} \to \mathcal{M}^m \mid k, l, m \in \mathbb{N}\}$ *is a vFHE scheme that additionally satisfies the server privacy property defined below.*

**Definition 5.2.11 (Server Input Privacy)**
*A scheme offers* server input privacy *when, for all* $f \in \mathcal{F}$*, the following are (statistically) indistinguishable*[5] *for all* $\vec{w}, \vec{w}'$*:*

$$(f(\overrightarrow{x w}), \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w})) \approx (f(\overrightarrow{x w'}), \mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}'))$$

*where* $x := \mathrm{Dec}_{\mathsf{sk}}(\vec{c_x})$.

Guarantees about the correct execution of a circuit are usually not sufficient to guarantee desirable application-level properties. Arguably, there is little difference to the client between a 'correct' execution of the circuit on 'incorrect' inputs and an incorrect execution of the circuit. While some amount of possible deviation is inherent in any 2-party computation, a rich history of MPC and ZKP applications has shown how to use input checks to ensure that a computation achieves higher-level properties [77]. For example, in Machine Learning as a Service (MLaaS), a client can run several validation queries on inputs with known labels to ensure the model has sufficient accuracy before querying it with real inputs. However, in the privacy-preserving MLaaS setting, where the model weights are private to the server, the server could switch out or degrade the model without the client noticing. This can be solved efficiently by requiring the server to commit to the model weights before the validation and then ensuring that the provided model weights match the commitment as part of each query. Alternatively, the client might want to ensure that the server inputs lie inside a valid range that is a subset of the possible plaintext space, and, more generally, we can consider arbitrary predicates on the server inputs that must hold for the client to accept the result.

We note that one could, in theory, integrate such input checks into the function $f$. However, this would require evaluating them under FHE and

5 Note that this must hold even for an adversary with access to sk.

potentially cause issues when trying to combine this with the notion of circuit privacy. We define our notion as an extension of vFHE with private server inputs, as it only really applies in this setting: public inputs are modeled as part of the circuit and can be trivially checked for consistency by the client. We model the input checks as a predicate $\phi : w \mapsto \{0,1\}$ and extend the notion as follows:

**Definition 5.2.12 (vFHE with Input Predicates)**
*A verifiable FHE scheme with* input predicates $\phi$ *for a class of circuits* $\mathcal{F} := \{f : \mathcal{M}^{k+l} \to \mathcal{M}^m \mid k,l,m \in \mathbb{N}\}$ *is a vFHE scheme with server input privacy that additionally satisfies the* input predicates *property defined below.*

**Definition 5.2.13 (Input Predicates)**
*A scheme satisfies* input predicates $\phi : \mathcal{M}^l \to \{0,1\}$ *if for any PPT adversary* $\mathcal{A}$ *the following probability is negligible in the security parameter* $\lambda$*:*

$$\Pr\left[\begin{array}{c|c} \mathsf{Verify}_{\mathsf{sk}}(f, \vec{c_y}) = 1 & (\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{KGen}(1^\lambda) \\ \wedge & x, f \leftarrow \mathcal{A}_1^{\mathcal{O}_{\mathsf{Enc}}, \mathcal{O}_{\mathsf{Dec}}}(\mathsf{pk}) \\ \nexists \vec{w} \in \mathcal{M}^l s.t. & \vec{c_x} \leftarrow \mathsf{Enc}_{\mathsf{key}}(x) \\ \mathsf{Dec}_{\mathsf{sk}}(\vec{c_y}) = f(\overrightarrow{xw}) \wedge \phi(\vec{w}) = 1 & \vec{c_y} \leftarrow \mathcal{A}_2^{\mathcal{O}_{\mathsf{Enc}}, \mathcal{O}_{\mathsf{Dec}}}(\vec{c_x}) \end{array}\right]$$

## 5.2.3 *Generic Construction*

We show how to generically construct a maliciously secure verifiable FHE scheme (Definition 5.2.6) from a standard FHE scheme and a generic ZKP system. In order to achieve security in this malicious setting (especially when considering private server inputs) we need to combine a proof of circuit correctness (i.e., of correct computation) with well-formedness checks on the inputs that ensure the result will be safe to decrypt. This requires us to make the concept of *valid* decryptions explicit, which will usually have an application-dependent component. For example, a logit vector returned by a machine learning model should consist of probabilities that sum to one. Therefore, we do not hard-code a concept of validity into our construction, but instead, define it relative to a set of circuits that return valid results (for a certain range of inputs). In the following, we will assume $\mathcal{F}$ to be defined so that, for all $f \in \mathcal{F}$ and all $\vec{c_x}$ and $\vec{w}$ so that $\overrightarrow{xw}$ is in the image of $f$, $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Eval}(f, \vec{c_x}, \vec{w}))$ results in such a "valid" decryption.

We build our construction from a standard IND-CPA FHE scheme, which includes all modern state-of-the-art schemes. While it might initially appear more attractive to utilize an IND-CCA1 FHE scheme, since our security notion is similarly defined, this is not ideal. Comparatively few IND-CCA1 FHE schemes are known [31, 127, 164] and the existing schemes are computationally much more expensive than IND-CPA schemes. This is because existing IND-CCA1 schemes either already explicitly include integrity protections [31] or rely on powerful primitives that, in turn, would require integrity-like protections [127, 164] to realize. As a result, starting our construction with an IND-CCA1 scheme and then adding integrity protections for circuit correctness would lead to duplicated efforts when instantiating the construction in practice.

Note that constructing a scheme for the core vFHE definition is comparatively straightforward, as this setting only features public server inputs: for any $f$ (which includes the public inputs) that results in a valid ciphertext for any freshly encrypted client inputs, we can combine an IND-CPA secure FHE scheme with a MAC-then-Encrypt scheme such as Chatel et al. [36] or a generic ZKP of circuit correctness. Therefore, we start with this construction, and incrementally extend it to achieve the more complex notions of verifiable FHE, including approximate correctness, server input privacy, and input predicates. Finally, we discuss concrete instantiations of these constructions in the following section (Section 5.3).

**Building Blocks.** In the following, we formally introduce the building blocks used in our generic construction. We refer to Definition 5.2.1 for a definition of Fully Homomorphic Encryption.

**Definition 5.2.14 (SNARK)**
*Let $\mathcal{R}$ be an efficiently computable binary relation which consists of pairs of the form $(x, w)$, where $x \in \mathcal{X}$ is a statement, and $w \in \mathcal{W}$ is a witness. Let $L$ be the language associated with the relation $\mathcal{R}$, i.e., $L = \{x \mid \exists w.\mathcal{R}(x, w) = 1\}$.*
*A triple of polynomial time algorithms $\Pi = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ is a SNARK for an NP relation $\mathcal{R}$, if the following properties are satisfied:*

*Completeness: For every true statement for the relation $\mathcal{R}$, an honest prover with a valid witness always convinces the verifier:* $\forall (x, w) \in \mathcal{R}$ :

$$\Pr\left[\mathsf{Verify}_{\mathsf{vk}}(x, \pi) = 1 \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^{\lambda}) \\ \pi \leftarrow \mathsf{Prove}_{\mathsf{crs}}(x, w) \end{array}\right] = 1$$

*Knowledge Soundness: For every PPT adversary, there exists a PPT extractor that gets full access to the adversary's state (including its random coins and inputs). Whenever the adversary produces a valid argument, the extractor can compute a witness with high probability:* $\forall \mathcal{A} \exists \mathcal{E}$ :

$$\Pr\left[\begin{array}{c} \mathsf{Verify}_{\mathsf{vk}}(\tilde{x}, \tilde{\pi}) = 1 \\ \wedge \mathcal{R}(\tilde{x}, w') = 0 \end{array} \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^{\lambda}) \\ ((\tilde{x}, \tilde{\pi}); w') \leftarrow \mathcal{A}|\mathcal{E}(\mathsf{crs}) \end{array}\right] = \mathsf{negl}(\lambda)$$

*We stress here that this definition requires a* non-black-box *extractor, i.e., the extractor gets full access to the adversary's state.*
*Succinctness: For any $x$ and $w$, the length of the proof is given by* $|\pi| = \mathsf{poly}(\lambda) \cdot \mathsf{polylog}(|x| + |w|)$.

**Definition 5.2.15 (zk-SNARK)**
*A zk-SNARK for a relation $\mathcal{R}$ is a SNARK for $\mathcal{R}$ with the following additional property:*
*Zero-Knowledge: There exists a PPT simulator $\mathcal{S} = (\mathcal{S}_1, \mathcal{S}_2)$ such that $\mathcal{S}_1$ outputs a simulated CRS* $\mathsf{crs}$ *and a trapdoor* $\mathsf{td}$*; On input* $\mathsf{crs}$, *$x$, and* $\mathsf{td}$, *$\mathcal{S}_2$ outputs a simulated proof $\pi$, and for all PPT adversaries $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$, such that*

$$\left| \Pr\left[\begin{array}{c} (x, w) \in \mathcal{R} \\ \wedge \\ \mathcal{A}_2(\pi) = 1 \end{array} \;\middle|\; \begin{array}{c} (\mathsf{crs}, \mathsf{vk}) \leftarrow \mathsf{Setup}(1^{\lambda}) \\ (x, w) \leftarrow \mathcal{A}_1(1^{\lambda}, \mathsf{crs}) \\ \pi \leftarrow \mathsf{Prove}_{\mathsf{crs}}(x, w) \end{array}\right] - \right.$$
$$\left. \Pr\left[\begin{array}{c} (x, w) \in \mathcal{R} \\ \wedge \\ \mathcal{A}_2(\pi) = 1 \end{array} \;\middle|\; \begin{array}{c} (\mathsf{crs}', \mathsf{td}) \leftarrow \mathcal{S}_1(1^{\lambda}) \\ (x, w) \leftarrow \mathcal{A}_1(1^{\lambda}, \mathsf{crs}') \\ \pi \leftarrow \mathcal{S}_2(\mathsf{crs}', \mathsf{td}, x) \end{array}\right] \right| = \mathsf{negl}(\lambda)$$

**Constructions for vFHE.** We begin with a formal description of the construction for our core notion. As described above, the ZKP system could also be replaced with homomorphic MACs due to the lack of server inputs.

**Construction 1 (vFHE from FHE and ZKP)**

- Let $\mathcal{E}$ = ($\mathcal{E}$.KGen, $\mathcal{E}$.Enc, $\mathcal{E}$.Dec, $\mathcal{E}$.Eval) be an (IND-CPA secure) FHE scheme for a class of circuits $\mathcal{F}$.
- Let $\Pi$ = ($\Pi$.Setup, $\Pi$.Prove, $\Pi$.Verify) be a SNARK for the relation

$$\mathcal{R} = \left\{ \left( (f, \vec{c_x}, \vec{c_y}), \vec{w} \right) : \vec{c_y} = \mathcal{E}.\mathsf{Eval}_{\mathsf{pk}_\mathcal{E}}(f, \vec{c_x}, \vec{w}) \wedge \vec{w} \in \mathcal{W} \right\}$$

We construct a vFHE scheme $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Verify}, \mathsf{Dec})$ for $\mathcal{F}$ satisfying Definition 5.2.6 from these building blocks as follows:

- $\mathsf{KGen}(1^\lambda) \to ((\mathsf{pk}_\mathcal{E}, \mathsf{pk}_\Pi), (\mathsf{sk}_\mathcal{E}, \mathsf{sk}_\Pi))$,
  where $(\mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{E}) = \mathcal{E}.\mathsf{KGen}(1^\lambda)$ and $(\mathsf{pk}_\Pi, \mathsf{sk}_\Pi) = \Pi.\mathsf{Setup}(1^\lambda)$
- $\mathsf{Enc}_{\mathsf{key}}(x) \to c_x$, where $c_x = \mathcal{E}.\mathsf{Enc}_{\mathsf{key}_\mathcal{E}}(x)$
- $\mathsf{Eval}_{\mathsf{pk}}(f, \vec{c_x}, \vec{w}) \to (\vec{c_y}, \tau)$ for $\vec{c_x} \in \mathcal{C}_{\mathsf{in}}^k$ and $\mathcal{C}_{\mathsf{in}} = \mathcal{C}_{\mathsf{sk}} \cup \mathcal{C}_{\mathsf{pk}}$
  where $\vec{c} = \mathcal{E}.\mathsf{Eval}(f, \vec{c_x}, \vec{w})$ and $\tau = \Pi.\mathsf{Prove}(f, \vec{c_y}, \vec{c_x}, \vec{w})$
- $\mathsf{Verify}_{\mathsf{sk}}(f, (\vec{c_y}, \tau)) = \Pi.\mathsf{Verify}_{\Pi.\mathsf{sk}}\left(f, (\vec{c_y}, \tau)\right)$.
- $\mathsf{Dec}_{\mathsf{sk}}((c_y, \tau)) \to y$, where $y = \mathcal{E}.\mathsf{Dec}_{\mathsf{sk}_\mathcal{E}}(c_y)$

We note that we can construct a scheme with $\epsilon$-correctness following the same approach if we require $\mathcal{E}$ to be an approximate FHE scheme with IND-CPA$^\mathrm{D}$ security [125].

**Proofs for Construction 1**

*Correctness*: For honestly generated keys, encryptions, and evaluations, correctness directly follows from the correctness of the underlying FHE scheme. In the approximate case, $\epsilon$-approximativity follows directly from the approximate correctness of the FHE scheme.

*Completeness*: vFHE completeness follows directly from the completeness of the underlying ZKP scheme.
*Soundness*: Given an adversary $\mathcal{A}$ against the soundness of the vFHE scheme, we construct an adversary $\mathcal{B}$ against the knowledge-soundness of $\Pi$ as follows: $\mathcal{B}$, on input $\mathsf{pk}_\Pi$, generates the FHE keys $\mathsf{pk}_\mathcal{E}, \mathsf{sk}_\mathcal{E}$, and runs $\mathcal{A}$ with input $\mathsf{pk} = (\mathsf{pk}_\mathcal{E}, \mathsf{pk}_\Pi)$. $\mathcal{B}$ answers to $\mathcal{A}$'s queries to the encryption oracle using $\mathsf{pk}_\mathcal{E}$. For $\mathcal{A}$'s queries to its decryption oracle, $\mathcal{B}$ first relays the proof element of the query to its own verification oracle; $\mathcal{B}$ then answers with the decryption of $\mathcal{A}$'s ciphertext if the verification passes, and with $\perp$ otherwise. $\mathcal{B}$ can thus perfectly simulate the vFHE soundness game for $\mathcal{A}$. $\mathcal{A}$ wins if and only if there is no witness for its output $c_x = \tau_x, c_y$, in which case there cannot exist an extractor for $\mathcal{B}$, and $\mathcal{B}$ breaks the knowledge soundness of $\Pi$ exactly when $\mathcal{A}$ breaks the soundness of the vFHE scheme.
*Security*: Informally, the addition of a (sound) proof of correct computation allows us to reduce the IND-CCA1 security of the vFHE scheme to the IND-CPA$^\mathrm{D}$ security of the underlying FHE scheme. Formally, for a given IND-CCA1 adversary $\mathcal{A}$ against vFHE, we construct an adversary $\mathcal{B}$ against the knowledge soundness of $\Pi$ and an IND-CPA$^\mathrm{D}$ adversary $\mathcal{C}$ against $\mathcal{E}$ such that $\mathsf{Adv}^{\text{IND-CCA1}}[\mathcal{A}](\lambda) \leq \mathsf{Adv}^{\mathrm{KS}}[\mathcal{B}](\lambda) + \mathsf{Adv}^{\text{IND-CPA}^\mathrm{D}}[\mathcal{C}](\lambda)$.

Let $G_0$ denote the IND-CCA1 game for vFHE, and let $G_1$ be the same game with a slightly different decryption oracle, modified as follows: on receiving a query $(c_y, \tau_x, \tau_y)$, the oracle returns $\perp$ if $\mathsf{Verify}_\mathsf{sk}(c_y, \tau_x, \tau_y) = 0$. If the verification passes, the oracle uses the knowledge extractor over $\mathcal{A}$ to retrieve the witness $w$ corresponding to the proof $\tau_y$; now, instead of returning a decryption of $c_y$, the oracle computes the expected ciphertext output $\hat{c_y} = \mathsf{Eval}_\mathsf{pk}(\tau_x, w)$, and returns $\mathsf{Dec}_\mathsf{sk}(\hat{c_y})$. Due to the correctness and completeness of $\mathcal{E}$, $G_0$ and $G_1$ are identical up until $\mathcal{A}$ issues a decryption query that is accepted by the verifier, but for which $c_y \neq c'_y$. The probability of this event is bounded by the advantage of an adversary $\mathcal{B}$ against the knowledge-soundness of $\Pi$; we can construct $\mathcal{B}$ from any $\mathcal{A}$ that is able to distinguish $G_0$ from $G_1$ as in the proof of soundness above.

We can construct an adversary $\mathcal{C}$ against the IND-CPA$^\mathrm{D}$ security of the underlying FHE scheme, by utilizing $\mathcal{A}$ in a perfect simulation of $G_1$. To do so, $\mathcal{C}$ generates the $\Pi$ keying materials, and uses the (encryption and decryption) oracles from its IND-CPA$^{\mathrm{D}}$ game to answer $\mathcal{A}$'s queries in $G_1$.

To conclude, for any IND-CCA1 adversary $\mathcal{A}$, we can either construct a knowledge-soundness adversary $\mathcal{B}$ against $\Pi$, or an IND-CPA$^{\mathrm{D}}$ adversary against $\mathcal{E}$. $\square$

Before presenting our generic construction, we recall the definition of FHE circuit privacy:

**Definition 5.2.16 (FHE Circuit Privacy [90])**
*An FHE scheme $\mathcal{E}$ is* circuit-private *for circuits in $\mathcal{F}_{\mathcal{E}}$ if, for any key-pair ($\mathsf{pk}$, $\mathsf{sk}$) output by $\mathcal{E}.\mathsf{KGen}(1^\lambda)$ any circuit $f \in \mathcal{F}_{\mathcal{E}}$, and any fixed inputs $c_x$ in the image of $\mathcal{E}.\mathsf{Enc}$ for inputs $x$, the following are (statistically) indistinguishable:*

$$\mathcal{E}.\mathsf{Enc}_{\mathsf{pk}}(f(x)) \approx \mathcal{E}.\mathsf{Eval}_{\mathsf{pk}}(f, c_x)$$

**Construction 2 (vFHE with Private Server Input)**

- Let $\mathcal{E}$ = ($\mathcal{E}$.KGen, $\mathcal{E}$.Enc, $\mathcal{E}$.Dec, $\mathcal{E}$.Eval) be an (IND-CPA secure) FHE scheme for $\mathcal{F}$ with (FHE) circuit privacy (see Definition 5.2.16).
- Let $\Pi$ = ($\Pi$.Setup, $\Pi$.Prove, $\Pi$.Verify) be a zkSNARK for

$$\text{ZKPPoK}\left\{\vec{w} : c_y = \mathcal{E}.\mathsf{Eval}_{\mathsf{pk}_{\mathcal{E}}}(f, \vec{c_x}, \vec{w}) \wedge \ \vec{w} \in \mathcal{W}\right\}$$

For $f \in \mathcal{F}$, we construct a vFHE scheme for $\mathcal{F}$ with server input privacy $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Verify}, \mathsf{Dec})$ satisfying Definition 5.2.10 from these building blocks following the approach of Construction 1.

**Proofs for Construction 2**
The proofs for *Correctness*, *Completeness*, *Soundness*, and *Security* are identical to the corresponding proofs for the core protocol above.
*Server Privacy*: we need to show that an adversary with access to all keys does not learn anything about $w$ from $(c_y, \tau_y) \leftarrow \mathsf{Eval}_{\mathsf{pk}}(c_x, w)$. The zero-knowledge property of $\Pi$ ensures that the proof $\tau_y$ hides $w$, while the circuit privacy of $f$ ensures that no information about $w$ can be derived from $c_y$.$\square$

Since we use a generic ZKP system to enforce validity checks on the server inputs, extending our construction to verifiable FHE with input predicates is straight-forward:

**Construction 3 (vFHE with Input Predicates)**

- Let $\mathcal{E}$ = ($\mathcal{E}$.KGen, $\mathcal{E}$.Enc, $\mathcal{E}$.Dec, $\mathcal{E}$.Eval) be an IND-CPA secure FHE scheme with (FHE) circuit privacy (see Definition 5.2.16).
- Let $\phi$ be a predicate on the inputs.
- Let $\Pi$ = ($\Pi$.Setup, $\Pi$.Prove, $\Pi$.Verify) be a SNARK for the relation

$$\mathcal{R} = \left\{ \left( (f, \vec{c_x}, \vec{c_y}), \vec{w} \right) : \vec{c_y} = \mathcal{E}.\mathsf{Eval}_{\mathsf{pk}_{\mathcal{E}}}(f, \vec{c_x}, \vec{w}) \wedge \vec{w} \in \mathcal{W} \wedge \phi(\vec{w}) = 1 \right\}.$$

We construct a vFHE scheme for $\mathcal{F}$ with input predicates $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Eval}, \mathsf{Verify}, \mathsf{Dec})$ satisfying Definition 5.2.12 from these building blocks following the approach of Construction 1.

**Proofs for Construction 3**

Definition 5.2.12 does not introduce a dedicated new property, but instead slightly modifies correctness, soundness, and completeness so that each also requires the predicate to hold. The definition of *Security* remains unchanged and the proof is identical to the corresponding proofs for Construction 1 above. The other proofs require slight modifications, but remain essentially the same:

*Correctness*: As for Construction 1, (approximate) correctness follows directly from the FHE scheme, as the addition of the predicate $\phi(w) = 1$ does not influence the computation.

*Completeness*: vFHE completeness follows directly from the completeness of the underlying ZKP scheme, as the predicate $\phi$ is incorporated in the proof system.

*Soundness*: The proof is nearly identical to the one for Construction 1, with the relation for $\Pi$ being augmented by the predicate $\phi$. □

## 5.3 INSTANTIATING VERIFIABLE FHE IN PRACTICE

In this section, we instantiate our notion of maliciously-secure verifiable FHE using state-of-the-art FHE and ZKP schemes. We highlight a series of challenges in bringing together modern FHE and ZKP systems, including the mismatch between the large polynomial rings used in most state-of-the-art FHE schemes and the integer fields used in the vast majority of ZKP systems. We investigate several approaches to bridge this gap and introduce a new optimization for emulating ring arithmetic inside ZKPs.

We consider a wide range of ZKP systems and identify four promising candidates that are best suited to the characteristics of FHE verification. In addition, we also discuss how to instantiate our notion with hardware attestation primitives, introducing an optimization that allows us to accelerate FHE-in-TEE by a factor of two over the existing state-of-the-art for multiplications. We evaluate our ZKP- and TEE-based instantiations for a variety of different workloads going far beyond the type of circuits that existing FHE integrity notions can express.

### 5.3.1 *Verifiable FHE via ZKP*

In the following, we discuss how to instantiate our construction for verifiable FHE with private server inputs (c.f. Section 5.2.3).

**Bridging FHE and ZKP.** The areas of FHE and ZKP have been maturing mostly independently, and state-of-the-art ZKP systems have primarily been tailored to applications that share few characteristics with FHE. In addition, FHE computations are inherently large and complex, making proofs non-trivial. Most of this complexity arises from the advanced ciphertext maintenance operations used by state-of-the-art schemes. As a result, previous work on FHE integrity frequently chose to use simple schemes such as the BV scheme [24] to avoid this complexity. However, the applicability of these schemes is limited in practice because of their prohibitive overhead. We instead choose to target modern state-of-the-art FHE schemes which offer the performance necessary to realize real-world FHE applications.

FHE schemes fall into two main families, with the LWE-based FHEW [74] and TFHE [52] schemes focusing primarily on evaluating binary circuits, while RLWE-based schemes such as BGV [22], B/FV [21][78] and CKKS [47]) focus on arithmetic circuits. Initially, LWE-based schemes might appear promising since they use smaller ciphertexts and offer faster computation. However, efficient implementations of schemes in this family usually make heavy use of floating-point operations, which would introduce significant overhead in the ZKP proofs. The RLWE schemes, in turn, feature larger ciphertexts but also offer a powerful form of data parallelism [158] that is at the core of most state-of-the-art FHE results. While our construction can be instantiated with any scheme, we select BGV because it is amenable to integer-based ZKP and requires the least amount of non-arithmetic operations to realize its ciphertext-maintenance operations.

The RLWE setting introduces a fundamental mismatch between the *ring* based FHE computation and the mostly *field* based ZKP systems. Specifically, BGV uses rings of the form $R_q := \mathbb{Z}_q[X]/(X^N+1)$, i.e., polynomials with degree up to $N$ (usually $N > 2^{13}$) and coefficients in $\mathbb{Z}_q$. Most ZKP systems, on the other hand, mostly use large prime fields (i.e., $\mathbb{Z}_p$ where $p$ is usually a 254-bit prime). Note that multiplying polynomials efficiently requires converting them into a form that allows element-wise multiplication via the Number Theoretic Transform (NTT), a discrete analog to the Fast Fourier Transformation (FFT). As a side effect, this also removes the need to explicitly compute the reduction modulo $(X^N+1)$. However, we must still take care to achieve modular reduction with regard to the coefficient modulus $q$.

**Matching the Coefficient Modulus.** Existing work mostly assumes that it is possible to instantiate the FHE and ZKP schemes so that the ZPK field modulus $p$ is the same as the FHE coefficient modulus $q$. While FHE already needs to support a wide range of ciphertext moduli since this is an application-dependent parameter, most ZKP systems offer less flexibility. There is also an inherent tension between the FHE and ZKP parameters here, since FHE security reduces with larger $q$ (requiring increases in $N$ to compensate) while many ZKP systems rely on elliptic curves becoming more secure as $p$ increases. As a result, while it is possible to instantiate FHE with a matching coefficient, this results in an unnecessarily inefficient FHE scheme, especially for smaller circuits. On the other hand, for larger FHE circuits, $q$ might outgrow the ZKP modulus. More fundamentally, this approach does not support the *modulus switching* used by BGV and other state-of-the-art FHE schemes to manage noise. Both of these issues can be addressed by using the Chinese Remainder Theorem (CRT) to decompose the coefficient modulus $q$ into $L$ moduli $q_1, \ldots, q_L$, working on each *Residue Number System* (RNS) limb independently. This is already frequently used in FHE implementations to improve performance on existing hardware, as computing ten 60-bit operations is significantly cheaper than computing one 600-bit operation. However, even with the RNS approach, matching ZKP and FHE parameters remains difficult and inherently limited to trivial circuits, as it cannot express more complex ciphertext maintenance operations.

**Emulating the Coefficient Modulus.** Removing the need to match FHE and ZKP parameters allows more efficient instantiations of both, but requires us to emulate the FHE coefficient modulus inside the ZKP field. This can be done by explicitly computing the modulus (mod $q$) after each arithmetic operation, which is comparatively cheap. However, we also need to prove the correctness of the result, which requires two expensive range proofs. We introduce an optimization that reduces the need to perform these expensive modulus emulations. We observe that, in practice, the ZKP modulus is frequently significantly larger than the FHE modulus, especially for smaller circuits. Because modular reduction produces the same result whether it is applied to the inputs or the outputs of an operation, we can wait until just before the results could overflow and only compute and prove the modulo reduction then. Specifically, we can perform $\lfloor \frac{p}{q} - 1 \rfloor$ (multiplication) operations in sequence before needing to reduce. For small circuits with $q \sim 60$ *bits* and a standard field-based ZKP with $p \sim 254$ *bits*, this enables a 3x reduction in the number of modulo operations.

Without our optimization, there is little difference between the RNS and non-RNS approaches with respect to the effort required to prove modulus gates. This is because the cost of proving a modular reduction is roughly linear in the bit-width of the modulus. However, with our optimization, using the RNS approach allows us to reduce the number of modular reductions even further. Considering an FHE circuit with $k$ arithmetic operations, a non-RNS, non-optimized implementation requires $k$ modular reductions of size $\log q$. Our optimization reduces this to roughly $\frac{q}{p} * k$ modular reductions of size $\log q$. With RNS splitting each element into $L$ limbs, the size of each gate is reduced to $\frac{\log q}{L}$ but, without any optimizations, the number of modular reductions increases to $kL$, negating the benefits. However, with our optimization, we require only $\frac{q_i}{p} * k * L = \frac{q}{L} * \frac{1}{p} * k * L$, i.e., $\frac{q}{p} * k$ modular reductions of the reduced size. Therefore, with our optimization, RNS allows us to reduce the cost per modular reduction even while already reducing their number. For example, with $q \sim 60$ *bits* and $p \sim 254$ *bits* as above, an RNS approach splitting $q$ into two $q_i \sim 30$ *bits* would halve the cost again, giving us a total 6x decrease in modular reduction overhead. As a result, we can construct comparatively efficient ZKP circuits for a wide range of FHE applications while allowing freedom in FHE and ZKP parameter selection.

**Arithmetization.** Given a computation defined as a circuit, different ZKP systems follow different approaches in translating the correctness of computation into an arithmetic proof system. Since the circuits arising in FHE integrity follow similar patterns, we can generically select an appropriate approach independent of the concrete FHE application. We propose to use R1CS [10], one of the most widespread arithmetization approaches, which converts circuits into a system of Rank-1 constraints. Basic arithmetic operations such as additions and multiplications can be realized directly using a single constraint. However, more complex operations (e.g., rounding) require a larger number of constraints. Since the majority of operations in (NTT-based) FHE are simple arithmetic operations, they can be efficiently translated to R1CS. While PLONK-like systems [84] extend R1CS with support for custom gates that can model commonly recurring sub-circuits, this is only efficient for low-degree sub-circuits[6] In our setting, the only candidate for such extraction would be the modulo sub-circuit, but this is not a low-degree circuit. Beyond R1CS, STARKs [9] present an alternative way to express computations not as circuits but as a series of data manipulations. The efficiency of STARKs is directly related to the size of the state space and the complexity of the transition function describing each step. Since FHE features large ciphertext expansion and a high-degree modulo function, this makes it a poor candidate for realization using STARKs. Recent work [166] can also forgo explicit arithmetization and instead directly express arithmetic circuits. However, this approach scales poorly with the number of inputs, which is high for FHE circuits due to the expansion from individual ciphertexts to $2N$ field elements (where $N \geq 2^{13}$). As a result, R1CS currently remains the most appropriate choice for FHE integrity applications, and we consider only ZKP systems with efficient support for R1CS arithmetizations.

**ZKP Scheme Selection.** In recent years, a large variety of efficient ZKP systems has been proposed, with many seeing widespread use in real-world deployments. However, when selecting suitable candidates for our instantiation, we need to consider that FHE has slightly different requirements than, e.g., blockchain applications. For example, due to the large ciphertext expansion, proof size is less of a concern in FHE, which already introduces a noticeable communication overhead. Instead, we are mostly concerned with achieving a good trade-off between prover and verifier time. Note that we do not require public verifiability and can therefore exploit potentially more efficient designated verifiability schemes. We select four candidate approaches that are especially suitable for verifiable FHE:

First, we consider Groth16 [98], which represented a major breakthrough in practical ZKP research, showing that zkSNARKs can be achieved with good concrete efficiency. It requires a trusted setup, which presents difficulties in settings where the set of parties is not known in advance, such as in blockchain applications. However, in verifiable FHE, the client and

6 Very recent work [38] promises to address this issue by improving efficiency for higher-degree sub-circuits. However, only an incomplete closed-source proof-of-concept implementation exists at this point.

server can easily realize this trusted party during a one-time setup via a 2-party maliciously-secure MPC protocol. Follow-up work has introduced new schemes with different tradeoffs, such as removing the need to re-run the setup for each circuit. However, this is not relevant to FHE, where parameters are usually already circuit-specific.

On the other hand, we do evaluate transparent SNARKs that completely remove the need for such a set-up. Here, we consider Bulletproofs [26] which is part of a generation of zkSNARKS without trusted setup that brings them into the same realm of performance as traditional efficient constructions that require (universal or per-circuit) setup. While most implementations of Bulletproofs focus on using it for efficient range proofs, for FHE we also require the ability to support more generic R1CS systems. We also evaluate Aurora [11], which is part of the same generation and trades off asymptotically worse prover times and proof sizes for a move to post-quantum secure assumptions, which matches nicely with the post-quantum security of FHE schemes.

Finally, we also consider Rinocchio [85] by Ganesh et al., which we briefly discussed in Section 5.1.1. Rinocchio offers native support for FHE-friendly rings, potentially giving significant performance benefits over systems that need to emulate these rings. We extend Rinocchio with a more optimized encoding scheme, which we describe in more detail below. However, its expressiveness is limited, as Rinocchio only supports arithmetic ring operations, whereas some FHE operations (e.g., relinearization) use component-wise rounding operations internally. Additionally, the soundness of Rinocchio relies on the size of the exceptional set, which for FHE-friendly cyclotomic rings is $|A| = q_1 \approx 2^{60}$, i.e., a single run of Rinocchio provides only around 60 bits of (computational) soundness. We use a simple soundness amplification strategy, running three separate instances of the protocol to achieve stronger soundness guarantees. Overall, Rinocchio is much more FHE-friendly than previous proof or argument systems, but still struggles to efficiently represent state-of-the-art FHE optimizations natively. For example, it is unclear how to implement a sound check of plaintext inputs while staying in the efficient NTT representation we use for our evaluation.

Specifically, we need to bound the size of the plaintext coefficients pre-NTT and then verify that the post-NTT input is indeed the result of applying the NTT to this input. However, without the ability to verify the application of automorphisms, we cannot compute this NTT without assuming some of the server inputs are given in specific forms. Nevertheless, we include it because it represents an interesting avenue for future work and promises significantly improved performance for the circuits it can support. In Section 5.3.3, we evaluate our construction when instantiated using these ZKP schemes and also compare it against a TEE-based instantiation.

**Optimizations to the Rinocchio Protocol.** In the following, we describe our optimizations for the Rinocchio protocol by Ganesh et al. [85]. The original paper introduces two possible encodings for the cyclotomic rings used by FHE. The first one (dubbed "Regev-style" encoding) encodes each of the $N$ coefficients in $\mathbb{Z}_q$ by encrypting it into an element of $\mathbb{Z}_Q^n$ using a LWE cryptosystem scheme; the parameters of the encoding scheme are chosen to ensure that the encodings are $k$-linearly-homomorphic, where $k$ is determined by the circuit. The second construction ("Torus encoding") uses a variant of the TFHE cryptosystem.

The Regev encoding has an expansion factor of $\frac{N \cdot n \cdot \log_2(Q)}{N \cdot \log_2(q)} = n \cdot \log_q(Q)$, as it encodes each of the $N$ coefficients in $\mathbb{Z}_q$ as an element of $\mathbb{Z}_Q^n$. However, most FHE implementations will not be able to support a plaintext modulus of the size of $q$ (typically hundreds of bits), and in practice one would need to encode each of the $l$ CRT components individually, leading to an expansion factor of $l \cdot n \cdot \log_q(Q)$. Using this encoding will thus slow down the prover and verifier significantly, as all encodings, decodings, and computations over the encoding space will be slow.

The TFHE encoding, on the other hand, requires using floating-point arithmetic to encode and decode, unlike the FHE scheme used for computation. Additionally, this encoding does not allow us to use the (potentially heavily optimized) ring arithmetic libraries provided by the FHE library.

Therefore, we propose a new RLWE Regev-style encoding for Rinocchio, taking advantage of the batching technique commonly used in FHE. For many FHE schemes, if the plaintext modulus $t$ satisfies the condition $t = 1$

mod $2N$, one can use an efficient encryption that packs $N$ plaintext values (interpreted as an element of $R_t$) into a single ciphertext in $R_q^2$. For our encoding, we take an input in $R_q$ as $l$ polynomials in $R_{q_1}, \ldots, R_{q_l}$ (this decomposition is already used natively by the FHE scheme for efficiency reasons), and encode each of those polynomials as an element in $R_Q$. The expansion factor in this case is $\frac{l \cdot \log_2(Q)}{\log_2(q)} = l \cdot \log_q(Q)$, improving on the Regev encoding by a factor of $N \geq 2^{10}$. Using this batching technique imposes the requirement $q_i = 1 \mod 2N$ on the ciphertext moduli of the FHE scheme; this condition is already necessary for some schemes (e.g., RNS-optimized BGV [113]), and can be easily satisfied for all other schemes.

### 5.3.2 *Verifiable FHE via TEE*

In this instantiation, we use hardware attestation rather than cryptographic proofs to provide the integrity component. We note that, while there has been a plethora of attacks on TEE-based *confidentiality* guarantees [80, 140, 142], there have been significantly fewer issues with the attestation-based integrity guarantees [140]. Natarajan et al. presented the first implementation of this FHE-in-TEE approach [141] and we extend their work to our notion with server inputs. In addition, we introduce an optimization that allows us to efficiently compute subfunctions on untrusted hardware. Since TEEs can directly attest to the program that they are running, there is no need for explicit arithmetization. However, programs will frequently require adjustments to properly interface with the enclave SDK and to remove unsupported operations and performance bottlenecks. Nevertheless, TEEs support most FHE schemes more naturally than ZKP systems, including offering native support for, e.g., rounding or floating-point operations. However, computations and especially memory operations inside the enclave are significantly slower than operations in the untrusted domain. We address this issue partially by introducing a new optimization that accelerates FHE-in-TEE by a factor of two for batched multiplications. The key insight in our optimization is that the server can also rely on the guarantees of the enclave to not leak its inputs to the client. Specifically, it can use lightweight cryptographic proofs that do not have the zero-knowledge property to prove to the enclave that computations performed on the untrusted hardware are indeed correct and can be relied upon by the enclave. This enables our optimization which computes computationally expensive batched multiplications on untrusted hardware and then uses an efficient Schwartz-Zippel-based proof of correctness to move them back to the trusted domain.

In the following, we describe our optimization for FHE-in-TEE: Executing any code inside a TEE incurs a slowdown (due to reduced computational power and memory), especially in the case of FHE computations, which are typically compute- and memory-intensive. To alleviate this slowdown, we propose a new method to accelerate FHE computations inside TEEs by taking advantage of faster (but untrusted) hardware (e.g., a vanilla untrusted CPU, a CPU with specialized vector instructions repurposed for FHE [15], a GPU accelerator for FHE [147], or even a dedicated hardware accelerator).

The key insight to our improvement is that both the TEE and the untrusted hardware are on the side of the (malicious) server. Therefore, the server's input does not need to be protected from the server, and can be stored on the server's own untrusted hardware; the client's inputs are only available in their encrypted form, and can thus also be stored outside the enclave. This insight allows us to devise a protocol for *verifiably* outsourcing certain FHE operations. In order to do this efficiently, we rely on a very lightweight, information-theoretic argument of equality, based on the generalized Schwartz–Zippel lemma over rings:

**Theorem 5.3.1 (Generalized Schwartz-Zippel Lemma over Rings [13, 85])** *For a ring $R$, let $f : R^n \rightarrow R$ be a $n$-variate non-zero polynomial, let $A \subseteq R$ be a finite exceptional set, and let* $\deg(f)$ *denote the total degree of $f$. Then:*

$$\Pr_{\vec{a} \leftarrow A^n}[f(\vec{a}) = 0] \leq \frac{\deg(f)}{|A|}$$

For a given computation, we encode the expected result in one polynomial ($f$), and the actual result computed on untrusted hardware in another polynomial ($g$). The trick for efficiency, then, is to compute the compute and compare $f(a)$ and $g(a)$ faster than computing the full representation of $g$ in the first place.

Consider, for example, the *tensoring* operation, which is the most computationally expensive part of FHE multiplication (for the B/FV, BGV, and CKKS schemes). In the following, we will interpret a ciphertext $ct = (ct_0, \ldots, ct_{k-1}) \in R_q^k$ as a polynomial of degree $k-1$ over $R_q$, where $ct_i$ is the $i$-th coefficient.

The tensoring operation takes as input two ciphertexts $ct = (ct_0, ct_1), ct' = (ct'_0, ct'_1) \in R_q^2$, and outputs $c_{\text{out}} = ct \cdot ct' = (ct_0 \cdot ct'_0, ct_0 \cdot ct'_1 + ct'_0 \cdot ct_1, ct_1 \cdot ct'_1) \in R_q^3$. Now, evaluating the expected result $ct \cdot ct'$ at a random point $a \in A$ can be done efficiently as follows:

$$f(a) := (ct \cdot ct')(a) = ct(a) \cdot ct'(a) = (ct_0 + a \cdot ct_1) \cdot (ct'_0 + a \cdot ct'_1)$$

Evaluating the untrusted result $ct_{\text{out}}$ at this same point can be done by using Horner's rule:

$$g(a) := ct''(a) = ct''_0 + a \cdot (ct''_1 + a \cdot ct''_2)$$

After checking that $f(a) = g(a)$, we know that $ct \cdot ct' = ct''$ with high probability (for $R = \mathbb{Z}_{q_1 \cdot \ldots \cdot q_l}[X]/f[X]$, we have $|A| = q_1 \approx 2^{60}$, i.e., 60 bits of soundness). While computing the result has a concrete complexity of $1\ {}_{R}{+}_{R}$, $4\ {}_{R}{\times}_{R}$, verifying the result as outlined above only requires $4\ {}_{A}{\times}_{R}$, $4\ {}_{R}{+}_{R}$, $1\ {}_{R}{\times}_{R}$.

This approach can also be extended as follows to verify $k$ tensoring operations at the same time. Let $f(a_1, \ldots, a_k) := \sum_{i=1}^{k} (ct_i \cdot ct'_i)(a_i) = \sum_{i=1}^{k} (ct_{i,0} + a_i \cdot ct_{i,1}) \cdot (ct_{i,0} + a_i \cdot ct'_{i,1})$, and define $g(a_1, \ldots, a_k) := \sum_{i=1}^{k} ct''_i(a_i)$
$= \sum_{i=1}^{k} (ct''_{i,0} + a_i \cdot (ct''_{i,1} + a_i \cdot ct''_{i,2}))$.

Computing $k$ tensoring operations has a concrete complexity of $k\ {}_{R}{+}_{R}$, $4k\ {}_{R}{\times}_{R}$, while verifying the result by computing $f(\vec{a})$ and $g(\vec{a})$ has a complexity of $4k\ {}_{A}{\times}_{R}, (6k-2)\ {}_{R}{+}_{R}, k\ {}_{R}{\times}_{R}$ . By trading expensive $R$-$R$ multiplications for cheaper $R$-$R$ additions and $A$-$R$ multiplications, we are able to achieve a non-negligible speed-up, which we quantify in the next section.

We can view our protocol as a much more efficient, non-zero-knowledge version of Rinocchio; indeed, Rinocchio also uses Theorem 5.3.1, but requires significantly more protocol machinery in order to achieve zero-knowledge. In addition, Rinocchio offers roughly $\log_2(q_1) \approx 60$ bits of computational soundness (and thus requires a soundness amplification strategy), while our protocol offers $\log_2(q_1)$ bits of *statistical soundness*, and can therefore provide a satisfactory level of security by itself.

We note that this optimization is similar to the Slalom framework by Tramèr and Boneh [159], which offloads matrix multiplications (over unencrypted) values to untrusted hardware by using Freivalds' algorithm. Slalom relies on the TEE both for integrity and data confidentiality and only supports matrix multiplication, whereas our protocol does not require confidentiality of the data stored on the TEE, and can handle arbitrary polynomial computations.

### 5.3.3 *Performance Analysis*

In this section, we evaluate our ZKP- and TEE-based instantiations across a range of workloads representing different levels of complexity. We conclude our analysis with a brief outlook on the future of FHE integrity.

**Implementation & Setup.** We use the Microsoft SEAL [41] implementation of the BGV scheme [22], which is a state-of-the-art RNS- and NTT-based implementation. We express our ZKP circuits using Circom [86] which translates its custom specification language to R1CS. We rely on a variety of state-of-the-art ZKP implementations for the backends: we use the arkworks library [59] to implement Groth16 [98], for Aurora [11] we use the libiop library by the same authors, and for Bulletproofs [26] we use the Dalek library [68]. We implement Rinocchio [85] and extend it with an optimized encoding scheme (cf. Section 5.3.1). We make our implementation available as open-source[8]. For the TEE-based approach, we re-implement CHEX-MIX [141] and extend it with our optimization (cf. Section 5.3.2), targeting

6 We omit setup results when no additional setup is required.
8 `https://github.com/MarbleHE/ringSNARK`

Intel SGX via the OpenEnclave SDK [144]. We make our FHE-in-TEE framework available as open-source[9]. We evaluate our implementations on an AWS `c5d.4xlarge` instance with 16 vCPUs and 32 GB of RAM. We make our instantiations and evaluation setup publicly available[10].

**Workloads.** We consider three different circuits, each representing a different level of complexity: *(i)* Our Toy circuit computes a ciphertext-ciphertext multiplication on two inputs provided by the client, i.e., in the outsourced computation setting considered by previous work. *(ii)* The Small circuit represents a more realistic low-depth two-party computation, computing $x \cdot v + w$ for an encrypted client input $x$ and private server inputs $v$ and $w$. The presence of the server inputs requires input checks to ensure validity and prevent key-recovery attacks. Their private nature requires performing and proving noise-flooding as part of the computation. We follow the approach described in [17], which repeatedly adds encryptions of zero to increase the noise of the ciphertext without modifying the message. *(iii)* Finally, our Medium circuit introduces ciphertext-maintenance operations, computing $\mathsf{ModSwitch}\left((x-w)^2\right)$ for a client input $x$ and a private server input $w$. As in the previous task, this requires proving the validity of the client input and ensuring server privacy via noise-flooding. In addition, it requires computing and proving the modulus switching ciphertext operation, going well beyond what prior work on FHE integrity is able to express.

9 `https://github.com/MarbleHE/FHE-in-TEE`
10 `https://github.com/MarbleHE/ZKP-FHE`

| ZKP System | Toy | | | Small | | | Medium | | |
|---|---|---|---|---|---|---|---|---|---|
| | Setup | Prover | Verifier | Setup | Prover | Verifier | Setup | Prover | Verifier |
| FHE [No Integrity] | 0.003 s | 0.002 s | 0.001 s | 0.807 s | 0.011 s | 0.009 s | 1.053 s | 0.014 s | 0.010 s |
| Bulletproofs [26] | -[6] | 7569.799 s | 552.079 s | - | 3957.122 s | 278.433 s | - | 8697.741 s | 575.792 s |
| Aurora [11] | - | 1554.589 s | 32.880 s | - | 3750.477 s | 79.323 s | - | 5028.085 s | 106.345 s |
| Groth16 [98] | 198.640 s | 195.941 s | 0.002 s | 479.222 s | 472.711 s | 0.002 s | 642.470 s | 633.741 s | 0.002 s |
| Rinocchio [85] | 0.485 s | 0.320 s | 0.096 s | 46.700 s | 305.000 s | 0.153 s | 56.90 s | 443.000 s | 0.181 s |
| TEE [Section 5.3.2] | - | 0.154 s | - | - | 1.100 s | - | - | 1.260 s | - |

Table 5.3: Performance results for different instantiations of verifiable Fully Homomorphic Encryption.
For FHE, Setup = Key Generation, Prover = Homomorphic Computation and Verifier = Encryption/Decryption

**Performance.** In Table 5.3 we present the performance results for the different instantiations, compared against a baseline of non-verified FHE. The FHE parameters for the workload are $N = 8192$ and $\log_2 q = 137 = 45 + 46 + 46$, and the zero-knowledge proofs have between $2^{22}$ and $2^{24}$ R1CS constraints. We observe that the runtimes fall into three different categories: practical (seconds), acceptable (minutes), and impractical (hours). We note that verifier times are always either practical or at least acceptable. However, prover time varies wildly between the different instantiations. For example, transparent SNARKs (Bulletproofs and Aurora) take several hours to compute the proofs. In addition, they have the slowest verifier times, making them overall unattractive for FHE verification, especially when considering that FHE permits a straightforward realization of trusted setups. Groth16 offers the best verification time, being nearly indistinguishable from pure FHE. At the same time, it offers acceptable prover runtimes in the order of minutes. This comes at the cost of several minutes of trusted setup, but that can be amortized over many client queries. Rinocchio[11] offers slightly slower but still very practical client verification but improves significantly on prover time and especially setup times. However, these performance gains only apply when instantiating Rinocchio over $\mathbb{Z}_q^n$, where the plaintext input checks cannot be fully expressed. Finally, FHE-in-TEEoffers by far the most practical performance at the cost of additional trust assumptions and hardware requirements.

**Outlook.** Our work shows that, while the cost of robustness is clearly non-negligible, it is, today, already acceptable for high-value applications in challenging settings. FHE applications currently mostly focus on settings where latency is not critical, which allows them to tolerate the additional overhead more easily. While our constructions and instantiations explored (and optimized) existing state-of-the-art FHE and ZKP schemes, there are further opportunities to improve performance through the co-design of FHE and ZKP schemes. More fundamentally, we need to improve our understanding of how to construct efficient and expressive ring-native ZKP systems. Specifically, next-generation FHE-friendly ZKPs should be able to

11 We report only the performance of our optimized version.

accommodate switching between different rings, as is common in modern FHE, and be able to express the automorphisms that underly rotation operations that are essential when using SIMD packing. We believe that the increasing demand to deploy FHE in challenging real-world environments will also drive future research on these issues.

# 6 SUMMARY

Fully Homomorphic Encryption is reaching an important milestone in practicality and deployability. Recent cryptographic and algorithmic advances have enabled a first wave of real-world deployments [112]. These initial deployments demonstrate the potential of FHE to enable a new generation of privacy-preserving applications. With upcoming hardware accelerators promising significant further speedup [30, 70, 153, 154], FHE is poised to become a key enabler for a wide range of privacy-preserving applications. However, development techniques and tooling have not kept up with the rapid advances of FHE. As a result, the development of FHE solutions has continued to require cryptographic experts and specialized knowledge.

This requirement for highly specialized expertise has limited the practical deployment of FHE to a small number of research-oriented use cases, preventing the technology from reaching its full potential in solving real-world problems. While commercial off-the-shelf solutions (e.g., for privacy-preserving ML inference) will unlock the benefits of FHE for some use cases, there remains a long tail of use cases that require custom FHE solutions to be developed. While experts will continue to play an important role in developing cutting-edge solutions for highly challenging applications, this leaves out a vast middle ground of use cases that require solutions beyond off-the-shelf products yet are comparatively simple enough to not require novel cryptographic techniques.

Addressing this gap requires the development of tools and resources that allow non-experts to develop and deploy efficient FHE-based solutions. This thesis presents a series of steps toward solving this challenge by providing a set of practical approaches and techniques that can be used by non-experts to implement FHE-based solutions effectively. More concretely, this thesis presented the following contributions.

**FHE Application Development (Chapter 3).** We provided a systematic understanding of the engineering challenges that need to be addressed to help broaden FHE adoption. Towards this, we studied and surveyed the current state-of-the-art of FHE tools to understand which of these challenges have been addressed and which remain to be solved. We considered these tools in practice by experimentally evaluating them across a range of case study applications, contrasting usability, expressiveness, and performance. For a selection of promising tools, we provided an in-depth analysis of usability and expressiveness in practice. We implemented and benchmarked case-study applications that represent different domains of FHE-based computation and allowed us to study not only the overall performance of FHE for these applications across tools but also the relative strengths of different tools compared to each other. Based on the insights gained through our study, we highlighted successes in the FHE tool space and identified gaps that remained to be addressed. We identified the unique programming paradigm of FHE and the lack of tool interoperability as key challenges preventing wider adoption.

**End-to-End Compilation for FHE (Chapter 4).** Building upon the gained understanding, we presented a new end-to-end compiler design that addresses the shortcomings of current tools and resources. Our architecture provides, for the first time, a true end-to-end toolchain for FHE development. We identified four phases of converting an application to an efficient FHE implementation and designed a set of Intermediate Representations (IRs) based on the requirements of each phase that allow us to naturally and efficiently express optimizations at these different levels. HECO, our Homomorphic Encryption Compiler, allows non-experts to develop efficient FHE-based solutions. In order to enable this, we proposed novel transformations and optimizations that map imperative programs to the unique programming model of FHE. HECO automatically applies batching, a restricted type of parallelism found in many FHE schemes, which is used by experts to drastically reduce ciphertext expansion and computational overhead. In our evaluation, we showed that HECO can match the performance of expert implementations, providing up to 3500x speedup over naive non-batched implementations.

**Verifiable Fully Homomorphic Encryption (Chapter 5).** Finally, we highlighted the challenges of deploying FHE-based solutions in real-world settings. We showed how existing integrity notions fall short, highlighting the mismatch between the setting assumed in the existing integrity literature and the settings used for the vast majority of FHE applications. We presented a new notion, verifiable FHE, that addresses the issues we identified, composing the standard notion of FHE with modular integrity properties. As a result, our notion easily adapts to a wide variety of FHE deployment settings. Verifiable FHE can be constructed generically from FHE and ZKP, and we explored a variety of possible instantiations, comparing them to a hardware-attestation–based approach (FHE-in-TEE). We highlight a series of fundamental challenges in bringing together FHE and ZKP systems and introduced a new optimization for emulating FHE ring arithmetic inside field-based ZKPs. We showed that verifiable FHE can be practical, but also highlighted the need for future work on ZKP systems specifically designed for the unique characteristics of FHE.

**Impact.** The research presented in this thesis has brought to light crucial FHE development issues within the community and has garnered significant interest. Our work has already influenced the direction of the next generation of FHE tools which are currently being developed [96]. The work has also attracted collaborations with industry leaders such as Google and Intel. In particular, Google has been actively involved in our efforts to standardize intermediate representations (IRs) across the FHE community and is transitioning its toolchain to an MLIR-based one following our design. Meanwhile, Intel is planning to adopt an MLIR-based toolchain based on HECO for its upcoming FHE accelerator. In conclusion, this work, along with concurrent efforts, has made significant progress in addressing many of the issues we identified in our initial investigation. However, as we have worked to overcome some of the most critical shortcomings, we have uncovered new challenges that must be addressed.

## 6.1 FUTURE WORK

Despite the progress that has been made toward achieving the vision of useable FHE, many issues still require attention and further research. Below, we separate these into three categories, focussing on potential future directions for FHE tooling, challenges around FHE integrity and verifiable FHE, and finally, potential future directions in accessible secure computation beyond the scope of FHE.

### *The Future of FHE Compilers*

There are a plethora of possible improvements to FHE compilers both in algorithmic design (optimizations) and in tooling (user experience) improvements. Here, we focus on two more fundamental directions that address key challenges in the current FHE ecosystem.

Currently, the FHE ecosystem is conceptually split between two mostly incompatible paradigms. On the one hand, RLWE-based schemes allow additions and multiplications over (large vectors of) integers. On the other hand, LWE-based schemes feature comparatively fast bootstrapping and can use functional bootstrapping to evaluate non-polynomial functions. Most FHE tools today tie the developer to the use of one of these two approaches. However, which approach is best to use depends on the target application specifics. Hence, this necessitates work on compiler designs that offer common abstractions and allow developers to re-target applications seamlessly. Future FHE compilers should automate such decisions so that developers no longer have to consider the underlying approach used for their applications.

As FHE hardware starts to emerge, they will require new compilation techniques to fully exploit their potential. Tools should be able to generate code for a variety of hardware architectures that will likely differ significantly in the parts of the computation that they optimize. The next generation of FHE compilers should be able to automatically translate programs to heterogeneous hardware. In addition, large-scale applications will have to be scheduled across compute clusters made of heterogeneous hardware, ranging from CPUs and GPUs to FPGA-accelerators and custom co-processors. These challenges need to be addressed with compilation tools, code generations, and scheduling techniques that abstract all these issues from developers yet deliver compelling performance.

*Practical FHE Integrity*

While this work has presented proof-of-concept implementations of ZKP-based verifiable FHE that shows its feasibility, enabling their practical use for more complex workloads requires significant further work. While Rinocchio [85] shows the potential of dedicated FHE-friendly Ring-ZKP schemes, further work is required to achieve practicality. For example, there is a need to investigate how more complex operations, such as relinearization, which are key to the performance of RLWE-based FHE schemes, can be realized efficiently. LWE-based FHE, on the other hand, might be more amenable to traditional ZKP techniques due to its smaller size. Investigating how traditional field-based ZKPs perform in this application is an interesting avenue for future work. However, it is likely that here, too, dedicated ZKP schemes will be required to achieve practical performance. For example, schemes optimized to handle complex operations such as boostrapping, which are essential operation in LWE-based FHE schemes.

In addition, verifiable FHE fundamentally changes the cost model of FHE computations, as the prover overhead frequently dominates over the FHE computation itself. This invalidates the heuristics currently used by experts and tools and can result in non-intuitive approaches significantly outperforming traditional wisdom. For example, avoiding relinearization operations by increasing parameter sizes will result in a significant slowdown in the FHE computation, but the resulting reduction in ZKP complexity will likely more than make up for this overhead. As a result, FHE tools that want to incorporate verifiable FHE need to introduce novel optimizations targeted at the performance characteristics of the underlying schemes.

*Beyond FHE: Developing & Deploying Advanced Cryptography*

Protecting user privacy is an ongoing challenge, and while FHE is a useful tool for secure computation, it is just one of many techniques that can be employed. As more secure computation techniques achieve practicality, the development and deployment of privacy-enhancing technologies (PETs) will only grow more complex. Future PETs will likely incorporate a variety of techniques, including FHE, Zero-Knowledge Proofs (ZKP), and Multi-Party Computation (MPC). This further raises the complexity of development, both cryptographically and from an engineering point of view. Currently, complex protocols incorporating multiple secure computation techniques are virtually always hand-crafted by experts familiar with the underlying techniques. While compilers for the respective techniques could help implement each sub-component of the puzzle, such an approach does not take into account the possible performance tradeoffs that arise when combining these techniques. In order to address this, we require tools that can reason and optimize across the boundaries of different schemes and techniques. This thesis has highlighted the potential of combining cryptographic insights with techniques and tools from the world of Programming Languages research. While we demonstrated the efficacy of this approach in the domain of FHE, we believe that this approach holds significant potential for the accessibility of secure computation in general.

# BIBLIOGRAPHY

[1] Ahmad Al Badawi et al. "OpenFHE: Open-Source Fully Homomorphic Encryption Library". In: *Cryptology ePrint Archive*. WAHC'22 (2022), 53. DOI: 10.1145/3560827.3563379. URL: https://doi.org/10.1145/3560827.3563379.

[2] Martin Albrecht et al. *Homomorphic Encryption Security Standard*. Tech. rep. Toronto, Canada: HomomorphicEncryption.org, 2018. URL: https://homomorphicencryption.org.

[3] Martin R Albrecht, Rachel Player, and Sam Scott. "On the Concrete Hardness of Learning with Errors". In: *Journal of Mathematical Cryptology* 9.3 (2015), 169. ISSN: 1862-2976, 1862-2984. DOI: 10.1515/jmc-2015-0016. URL: https://www.degruyter.com/view/j/jmc.2015.9.issue-3/jmc-2015-0016/jmc-2015-0016.xml.

[4] Anwar Hithnawi Alexander Viand Patrick Jattke. "HECO: Fully Homomorphic Encryption Compiler". In: 2023.

[5] *AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More*. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf.

[6] 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 - WAHC'19*. New York, New York, USA: ACM Press, 2019, 57. ISBN: 978-1-4503-6829-2. DOI: 10.1145/3338469.3358945. URL: http://dl.acm.org/citation.cfm?doid=3338469.3358945.

[7] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. “Faster Homomorphic Encryption Is Not Enough: Improved Heuristic for Multiplicative Depth Minimization of Boolean Circuits”. In: *Topics in Cryptology – CT-RSA 2020*. Springer International Publishing, 2020, 345. DOI: `10.1007/978-3-030-40186-3\\\_15`. URL: `http://dx.doi.org/10.1007/978-3-030-40186-3%5C%5F15`.

[8] Nick Barlow, Tomas Lazauskas, Oliver Strickson, and Adria Gascon. “SHEEP: A Homomorphic Encryption Evaluation Platform”. In: (2019). URL: `https://github.com/alan-turing-institute/SHEEP`.

[9] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. “Scalable, transparent, and post-quantum secure computational integrity”. In: *Cryptology ePrint Archive* (2018). URL: `https://eprint.iacr.org/2018/046.pdf`.

[10] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. “SNARKs for C: Verifying Program Executions Succinctly and in Zero Knowledge”. In: *Cryptology ePrint Archive* (2013). URL: `https://eprint.iacr.org/2013/507.pdf`.

[11] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P Ward. “Aurora: Transparent Succinct Arguments for R1CS”. In: *Advances in Cryptology – EUROCRYPT 2019*. Springer International Publishing, 2019, 103. DOI: `10.1007/978-3-030-17653-2\_4`. URL: `http://dx.doi.org/10.1007/978-3-030-17653-2%5C%5F4`.

[12] Siavosh Benabbas, Rosario Gennaro, and Yevgeniy Vahlis. “Verifiable Delegation of Computation over Large Datasets”. In: *Advances in Cryptology – CRYPTO 2011*. Springer Berlin Heidelberg, 2011, 111.

[13] Anurag Bishnoi, Pete L Clark, Aditya Potukuchi, and John R Schmitt. “On Zeros of a Polynomial in a Finite Grid”. In: *Comb. Probab. Comput.* 27.3 (2018), 310.

[14] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. “nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data”. In: *Proceedings*

*of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC'19. New York, NY, USA: Association for Computing Machinery, 2019, 45. ISBN: 978-1-4503-6829-2. DOI: 10.1145/3338469.3358944. URL: https://doi.org/10.1145/3338469.3358944.

[15] Fabian Boemer, Sejun Kim, Gelila Seifu, F D de Souza, Vinodh Gopal, et al. *Intel HEXL (release 1.2)*. https://github.com/intel/hexl. 2021. arXiv: 2103.16400 [cs.CR]. URL: http://arxiv.org/abs/2103.16400.

[16] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. "nGraph-HE: A Graph Compiler for Deep Learning on Homomorphically Encrypted Data". In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. CF '19. New York, NY, USA: ACM, 2019, 3. ISBN: 978-1-4503-6685-4. DOI: 10.1145/3310273.3323047. URL: https://dl.acm.org/doi/10.1145/3310273.3323047.

[17] Alexandre Bois, Ignacio Cascudo, Dario Fiore, and Dongwoo Kim. "Flexible and Efficient Verifiable Computation on Encrypted Data". In: *Public-Key Cryptography – PKC 2021*. Springer International Publishing, 2021, 528.

[18] C Boura, N Gama, and M Georgieva. "Chimera: A Unified Framework for B/FV, TFHE and HEAAN Fully Homomorphic Encryption and Predictions for Deep Learning". 2018. DOI: doi:10.1515/jmc-2019-0026. URL: https://eprint.iacr.org/2018/758.

[19] Florian Bourse, Rafaël Del Pino, Michele Minelli, and Hoeteck Wee. "FHE Circuit Privacy Almost for Free". In: *Advances in Cryptology – CRYPTO 2016*. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2016, 62. ISBN: 978-3-662-53007-8 978-3-662-53008-5. DOI: 10.1007/978-3-662-53008-5\\\_3. URL: https://link.springer.com/chapter/10.1007/978-3-662-53008-5%5C%5F3.

[20] Florian Bourse and Malika Izabachène. "Plug-and-play sanitization for TFHE". In: *Cryptology ePrint Archive* (2022).

[21] Zvika Brakerski. “Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP”. In: *Advances in Cryptology – CRYPTO 2012*. Vol. 7417. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, 868. ISBN: 978-3-642-32008-8. DOI: 10.1007/978-3-642-32009-5_50. URL: http://dx.doi.org/10.1007/978-3-642-32009-5%5C%5F50.

[22] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Trans. Comput. Theory* 6.3 (2014), 1. ISSN: 1942-3454. DOI: 10.1145/2633600. URL: http://doi.acm.org/10.1145/2633600.

[23] Zvika Brakerski and Vinod Vaikuntanathan. “Efficient Fully Homomorphic Encryption from (Standard) LWE”. In: *2011 IEEE 52nd Annual Symposium on Foundations of Computer Science*. Palm Springs, CA, USA: IEEE, 2011, 97. ISBN: 978-0-7695-4571-4. DOI: 10.1109/FOCS.2011.12.

[24] Zvika Brakerski and Vinod Vaikuntanathan. “Fully Homomorphic Encryption from Ring-LWE and Security for Key Dependent Messages”. In: *Advances in Cryptology – CRYPTO 2011*. Springer Berlin Heidelberg, 2011, 505. DOI: 10.1007/978-3-642-22792-9\\\_29. URL: http://link.springer.com/chapter/10.1007/978-3-642-22792-9%5C%5F29.

[25] Lars Brenna, Isak Sunde Singh, Håvard Dagenborg Johansen, and Dag Johansen. “TFHE-rs: A library for safe and secure remote computing using fully homomorphic encryption and trusted execution environments”. In: *Array* 13 (2022), 100118. ISSN: 2590-0056. DOI: 10.1016/j.array.2021.100118. URL: https://www.sciencedirect.com/science/article/pii/S2590005621000564.

[26] B Bünz, J Bootle, D Boneh, A Poelstra, P Wuille, and G Maxwell. “Bulletproofs: Short Proofs for Confidential Transactions and More”. In: *2018 IEEE Symposium on Security and Privacy (SP)*. 2018, 315. DOI: 10.1109/SP.2018.00020. URL: http://dx.doi.org/10.1109/SP.2018.00020.

[27] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. "TimeCrypt: Encrypted Data Stream Processing at Scale with Cryptographic Access Control". In: *USENIX NSDI*. 2020.

[28] Lukas Burkhalter, Nicolas Küchler, Alexander Viand, Hossein Shafagh, and Anwar Hithnawi. "Zeph: Cryptographic Enforcement of End-to-End Data Privacy". In: *USENIX OSDI*. 2021.

[29] Niklas Büscher and Stefan Katzenbeisser. *Compilation for Secure Multi-party Computation*. SpringerBriefs in Computer Science. Springer International Publishing, 2017. DOI: 10.1007/978-3-319-67522-0. URL: https://link.springer.com/book/10.1007/978-3-319-67522-0.

[30] Rosario Cammarota. "Intel HERACLES: Homomorphic Encryption Revolutionary Accelerator with Correctness for Learning-oriented End-to-End Solutions". In: *Proceedings of the 2022 on Cloud Computing Security Workshop*. CCSW'22. Los Angeles, CA, USA: Association for Computing Machinery, 2022, 3. ISBN: 9781450398756. DOI: 10.1145/3560810.3565290. URL: https://doi.org/10.1145/3560810.3565290.

[31] Ran Canetti, Srinivasan Raghuraman, Silas Richelson, and Vinod Vaikuntanathan. "Chosen-Ciphertext Secure Fully Homomorphic Encryption". In: *Public-Key Cryptography – PKC 2017*. Springer Berlin Heidelberg, 2017, 213.

[32] S Carpov, T H Nguyen, R Sirdey, G Constantino, and F Martinelli. "Practical Privacy-Preserving Medical Diagnosis Using Homomorphic Encryption". In: *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*. 2016, 593. DOI: 10.1109/CLOUD.2016.0084. URL: http://dx.doi.org/10.1109/CLOUD.2016.0084.

[33] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. "A Multi-Start Heuristic for Multiplicative Depth Minimization of Boolean Circuits". In: *International Workshop on Combinatorial Algorithms*. Springer. Springer International Publishing, 2017, 275. DOI: 10.1007/978-3-

319-78825-8_23. URL: https://link.springer.com/chapter/10.1007/978-3-319-78825-8%5C%5F23.

[34] 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*. SCC '15. Singapore, Republic of Singapore: ACM, 2015, 13. ISBN: 978-1-4503-3447-1. DOI: 10.1145/2732516.2732520. URL: http://doi.acm.org/10.1145/2732516.2732520.

[35] Dario Catalano and Dario Fiore. "Practical Homomorphic MACs for Arithmetic Circuits". In: *Advances in Cryptology – EUROCRYPT 2013*. Springer Berlin Heidelberg, 2013, 336.

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

[37] Bhuvnesh Chaturvedi, Anirban Chakraborty, Ayantika Chatterjee, and Debdeep Mukhopadhyay. "A Practical Full Key Recovery Attack on TFHE and FHEW by Inducing Decryption Errors". In: *Cryptology ePrint Archive* (2022).

[38] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. "HyperPlonk: Plonk with Linear-Time Prover and High-Degree Custom Gates". In: *Cryptology ePrint Archive* (2022). URL: https://eprint.iacr.org/2022/1355.pdf.

[39] Hao Chen, Wei Dai, Miran Kim, and Yongsoo Song. *Efficient Multi-Key Homomorphic Encryption with Packed Ciphertexts with Application to Oblivious Neural Network Inference*. Tech. rep. New York, NY, USA, 2019, 395. DOI: 10.1145/3319535.3363207. URL: https://doi.org/10.1145/3319535.3363207.

[40] Hao Chen, Zhicong Huang, Kim Laine, and Peter Rindal. "Labeled PSI from Fully Homomorphic Encryption with Malicious Security". In: *Cryptology ePrint Archive* (2018), 1223. DOI: 10.1145/3243734.3243836. URL: https://eprint.iacr.org/2018/787.

[41] Hao Chen, Kim Laine, and Rachel Player. "Simple Encrypted Arithmetic Library - SEAL v2.1". In: *Financial Cryptography and Data Security*. Springer International Publishing, 2017, 3. DOI: 10.1007/978-3-319-70278-0_1. URL: http://dx.doi.org/10.1007/978-3-319-70278-0%5C%5F1.

[42] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. "VeGen: a vectorizer generator for SIMD and beyond". In: *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '21. Virtual, USA: Association for Computing Machinery, 2021, 902. ISBN: 9781450383172. DOI: 10.1145/3445814.3446692. URL: https://doi.org/10.1145/3445814.3446692.

[43] Massimo Chenal and Qiang Tang. "On Key Recovery Attacks Against Existing Somewhat Homomorphic Encryption Schemes". In: *Progress in Cryptology - LATINCRYPT 2014*. Springer International Publishing, 2015, 239.

[44] Long Cheng, Fang Liu, and Danfeng Daphne Yao. "Enterprise Data Breach: Causes, Challenges, Prevention, and future Directions". In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 7.5 (2017).

[45] 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*. Springer International Publishing, 2019, 347. DOI: 10.1007/978-3-030-10970-7\\\_16. URL: http://dx.doi.org/10.1007/978-3-030-10970-7%5C%5F16.

[46] Jung Hee Cheon, Seungwan Hong, and Duhyeong Kim. *Remark on the Security of CKKS Scheme in Practice*. Cryptology ePrint Archive, Report 2020/1581. https://eprint.iacr.org/2020/1581. 2020.

[47] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. "Homomorphic Encryption for Arithmetic of Approximate Numbers". In: *Advances in Cryptology – ASIACRYPT 2017*. Vol. 10624.

Cham: Springer International Publishing, 2017, 409. ISBN: 978-3-319-70693-1. DOI: 10.1007/978-3-319-70694-8_15. URL: https://www.springerprofessional.de/homomorphic-encryption-for-arithmetic-of-approximate-numbers/15266370.

[48] E Chielle, N G Tsoutsos, O Mazonka, and M Maniatakos. "Encrypt-Everything-Everywhere: ISA Extensions for Private Computation". In: *IEEE Transactions on Dependable and Secure Computing* (2020), 1. ISSN: 1941-0018. DOI: 10.1109/TDSC.2020.3007066. URL: http://dx.doi.org/10.1109/TDSC.2020.3007066.

[49] Eduardo Chielle, Oleg Mazonka, Nektarios Georgios Tsoutsos, and Michail Maniatakos. "E3: A Framework for Compiling C++ Programs with Encrypted Operands". In: *IACR Cryptology ePrint Archive* 2018 (2018), 1013. URL: https://eprint.iacr.org/2018/1013.

[50] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "Faster Packed Homomorphic Operations and Efficient Circuit Bootstrapping for TFHE". In: *Advances in Cryptology – ASIACRYPT 2017*. Springer International Publishing, 2017, 377. DOI: 10.1007/978-3-319-70694-8\\\\_14. URL: https://link.springer.com/chapter/10.1007/978-3-319-70694-8%5C%5F14.

[51] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption Library". In: (2016). URL: https://tfhe.github.io/tfhe.

[52] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption Over the Torus". In: *J. Cryptology* 33.1 (2020), 34.

[53] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. "TFHE: Fast Fully Homomorphic Encryption over the Torus". In: (2018). URL: https://eprint.iacr.org/2018/421.

[54] Ilaria Chillotti, Nicolas Gama, and Louis Goubin. "Attacking FHE-based applications by software fault injections". In: *Cryptology ePrint Archive* (2016). URL: https://eprint.iacr.org/2016/1164.

[55] Ilaria Chillotti, Marc Joye, Damien Ligier, Jean-Baptiste Orfila, and Samuel Tap. "CONCRETE: Concrete operates oN ciphertexts rapidly by extending TfhE". In: *WAHC 2020 – 8th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. 2020. URL: `https://homomorphicencryption.org/wp-content/uploads/2020/12/wahc20%5C%5Fdemo%5C%5Fdamien.pdf`.

[56] Ilaria Chillotti, Marc Joye, and Pascal Paillier. *Programmable Bootstrapping Enables Efficient Homomorphic Inference of Deep Neural Networks*. Tech. rep. `https://ia.cr/2021/091`. Zama, 2020.

[57] E. J. Chou, A. Gururajan, K. Laine, N. K. Goel, A. Bertiger, and J. W. Stokes. "Privacy-Preserving Phishing Web Page Classification via Fully Homomorphic Encryption". In: *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. 2020, 2792.

[58] Sangeeta Chowdhary, Wei Dai, Kim Laine, and Olli Saarikivi. "EVA Improved: Compiler and Extension Library for CKKS". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 43. ISBN: 9781450386562. DOI: `10.1145/3474366.3486929`. URL: `https://doi.org/10.1145/3474366.3486929`.

[59] arkworks contributors. `arkworks` *zkSNARK ecosystem*. 2022. URL: `https://arkworks.rs`.

[60] Anamaria Costache, Benjamin R Curtis, Erin Hales, Sean Murphy, Tabitha Ogilvie, and Rachel Player. *On the precision loss in approximate homomorphic encryption*. `https://eprint.iacr.org/2022/162.pdf`. Accessed: 2022-2-26. URL: `https://eprint.iacr.org/2022/162.pdf`.

[61] Anamaria Costache, Kim Laine, and Rachel Player. "Evaluating the Effectiveness of Heuristic Worst-Case Noise Analysis in FHE". In: (2019). URL: `https://eprint.iacr.org/2019/493`.

[62] Anamaria Costache, Kim Laine, and Rachel Player. “Evaluating the Effectiveness of Heuristic Worst-Case Noise Analysis in FHE”. In: *Computer Security – ESORICS 2020*. Springer, 2020, 546. ISBN: 978-3-030-59012-3. DOI: 10.1007/978-3-030-59013-0_27. URL: https://doi.org/10.1007/978-3-030-59013-0%5C%5F27.

[63] Meghan Cowan, Deeksha Dangwal, Armin Alaghi, Caroline Trippel, Vincent T. Lee, and Brandon Reagen. “Porcupine: A Synthesizing Compiler for Vectorized Homomorphic Encryption”. In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Virtual, Canada: Association for Computing Machinery, 2021, 375. ISBN: 9781450383912. DOI: 10.1145/3453483.3454050. URL: https://doi.org/10.1145/3453483.3454050.

[64] Eric Crockett. “Simply Safe Lattice Cryptography”. PhD thesis. Georgia Institute of Technology, 2017. URL: https://smartech.gatech.edu/handle/1853/58734.

[65] Eric Crockett and Chris Peikert. “$\Lambda o \lambda$: Functional Lattice Cryptography”. In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2016, 993. URL: https://dl.acm.org/doi/abs/10.1145/2976749.2978402.

[66] 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*. ACM. 2018, 1020.

[67] CryptoExperts. *FV-NFLlib*. 2016. URL: https://github.com/CryptoExperts/FV-NFLlib.

[68] Dalek Cryptography. *Rust Bulletproofs Library*. Online: https://github.com/dalek-cryptography/bulletproofs. 2020.

[69] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. “Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep

Learning". In: *arXiv preprint arXiv:1801.08058* (2018). URL: https://arxiv.org/abs/1801.08058.

[70] DARPA. *Data Protection in Virtual Environments (DPRIVE)*. https://sam.gov/opp/16c71dadbe814127b475ce309929374b/view. 2020. URL: https://sam.gov/opp/16c71dadbe814127b475ce309929374b/view.

[71] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal 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*. 2019. URL: http://arxiv.org/abs/1912.11951.

[72] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. "CHET: an optimizing compiler for fully-homomorphic neural-network inferencing". In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2019, 142. ISBN: 9781450367127. DOI: 10.1145/3314221.3314628. URL: https://dl.acm.org/citation.cfm?doid=3314221.3314628.

[73] Mark Driver. *Emerging Technologies: Homomorphic Encryption for Data Sharing With Privacy*. Tech. rep. Gartner, Inc, 2020.

[74] Léo Ducas and Daniele Micciancio. "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second". In: *Advances in Cryptology – EUROCRYPT 2015*. Springer Berlin Heidelberg, 2015, 617. DOI: 10.1007/978-3-662-46800-5_24. URL: http://dx.doi.org/10.1007/978-3-662-46800-5%5C%5F24.

[75] Léo Ducas and Damien Stehlé. "Sanitization of FHE Ciphertexts". In: *Advances in Cryptology – EUROCRYPT 2016*. Vol. 9665. Lecture notes in computer science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2016, 294. ISBN: 978-3-662-49889-7 978-3-662-49890-3. DOI: 10.1007/978-3-662-49890-3_12. URL: https://eprint.iacr.org/2016/164.pdf.

[76] Enveil. *Enveil Raises $10 Million in Series A Funding*. Accessed: 2020-12-21. 2020. URL: `https://www.globenewswire.com/news-release/2020/02/18/1986152/0/en/Enveil-Raises-10-Million-in-Series-A-Funding.html`.

[77] David Evans, Vladimir Kolesnikov, and Mike Rosulek. "A Pragmatic Introduction to Secure Multi-Party Computation". In: *Foundations and Trends® in Privacy and Security* 2.2-3 (2018), 70. ISSN: 2474-1558. DOI: `10.1561/3300000019`. URL: `http://dx.doi.org/10.1561/3300000019`.

[78] Junfeng Fan and Frederik Vercauteren. "Somewhat Practical Fully Homomorphic Encryption". In: *Cryptology ePrint Archive* 2012 (2012), 144. URL: `https://eprint.iacr.org/2012/144`.

[79] Prastudy Fauzi, Martha Norberg Hovd, and Håvard Raddum. "On the IND-CCA1 Security of FHE Schemes". In: *Cryptology ePrint Archive* (2021).

[80] Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. "Security Vulnerabilities of SGX and Countermeasures: A Survey". In: *ACM Comput. Surv.* 54.6 (2021), 1.

[81] Axel Feldmann, Nikola Samardzic, Aleksandar Krastev, Srini Devadas, Ron Dreslinski, Karim Eldefrawy, Nicholas Genise, Christopher Peikert, and Daniel Sanchez. "F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption (Extended Version)". In: (2021). arXiv: `2109.05371 [cs.CR]`. URL: `http://arxiv.org/abs/2109.05371`.

[82] Dario Fiore, Rosario Gennaro, and Valerio Pastro. "Efficiently Verifiable Computation on Encrypted Data". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. CCS '14. Scottsdale, Arizona, USA: Association for Computing Machinery, 2014, 844.

[83] Dario Fiore, Anca Nitulescu, and David Pointcheval. "Boosting Verifiable Computation on Encrypted Data". In: *Public-Key Cryptography – PKC 2020*. Springer International Publishing, 2020, 124.

[84] Ariel Gabizon, Zachary J Aztec, and Aztec Oana Williamson. *PlonK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. `https://eprint.iacr.org/2019/953.pdf`. Accessed: 2022-12-2. 2022. URL: `https://eprint.iacr.org/2019/953.pdf`.

[85] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. "Rinocchio: SNARKs for Ring Arithmetic". In: *Cryptology ePrint Archive* (2021).

[86] Hermenegildo García Navarro. "Design and implementation of the Circom 1.0 compiler". MA thesis. Universidad Complutense de Madrid, 2020. URL: `https://eprints.ucm.es/id/eprint/62475/`.

[87] Jon Geater. "ARM® TrustZone®". In: *Trusted Computing for Embedded Systems*. Ed. by Bernard Candaele, Dimitrios Soudris, and Iraklis Anagnostopoulos. Cham: Springer International Publishing, 2015, 35.

[88] Robin Geelen et al. "BASALISC: Flexible Asynchronous Hardware Accelerator for Fully Homomorphic Encryption". In: (2022). arXiv: `2205.14017 [cs.CR]`. URL: `http://arxiv.org/abs/2205.14017`.

[89] Rosario Gennaro and Daniel Wichs. "Fully Homomorphic Message Authenticators". In: *Advances in Cryptology - ASIACRYPT 2013*. Springer Berlin Heidelberg, 2013, 301.

[90] Craig Gentry. "A fully homomorphic encryption scheme". PhD thesis. Stanford University, 2009. URL: `https://crypto.stanford.edu/craig`.

[91] Craig Gentry. "Fully homomorphic encryption using ideal lattices". In: *Proceedings of the forty-first annual ACM symposium on Theory of computing*. STOC '09. Bethesda, MD, USA: Association for Computing Machinery, 2009, 169. ISBN: 978-1-60558-506-2. DOI: `10.1145/1536414.1536440`. URL: `https://doi.org/10.1145/1536414.1536440`.

[92] Craig Gentry and Shai Halevi. "Implementing Gentry's Fully-Homomorphic Encryption Scheme". In: *EUROCRYPT*. 2011.

[93] Craig Gentry, Shai Halevi, and Nigel P Smart. "Homomorphic Evaluation of the AES Circuit". In: *Annual Cryptology Conference*. Springer. 2012, 850. URL: https://link.springer.com/chapter/10.1007/978-3-642-32009-5%5C%5F49.

[94] 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*. Vol. 8042. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, 75. ISBN: 978-3-642-40040-7. DOI: 10.1007/978-3-642-40041-4_5.

[95] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy". In: *Proceedings of The 33rd International Conference on Machine Learning*. Ed. by Maria Florina Balcan and Kilian Q Weinberger. Vol. 48. Proceedings of Machine Learning Research. New York, New York, USA: PMLR, 2016, 201. URL: http://proceedings.mlr.press/v48/gilad-bachrach16.html.

[96] Shruthi Gorantala et al. "A General Purpose Transpiler for Fully Homomorphic Encryption". In: *CoRR* abs/2106.07893 (2021). arXiv: 2106.07893. URL: https://arxiv.org/abs/2106.07893.

[97] Sanath Govindarajan and William S Moses. "SyFER-MLIR: Integrating Fully Homomorphic Encryption Into the MLIR Compiler Framework". In: (2020).

[98] Jens Groth. "On the Size of Pairing-Based Non-interactive Arguments". In: *Advances in Cryptology – EUROCRYPT 2016*. Springer Berlin Heidelberg, 2016, 305.

[99] Vernam Group. *cuFHE*. 2018. URL: https://github.com/vernamlab/cuFHE.

[100] Shai Halevi and Victor Shoup. "Algorithms in HElib". In: *Advances in Cryptology – CRYPTO 2014*. Ed. by Juan A Garay and Rosario

Gennaro. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, 554. ISBN: 978-3-662-44370-5 978-3-662-44371-2. DOI: 10.1007/978-3-662-44371-2_31. URL: http://dx.doi.org/10.1007/978-3-662-44371-2%5C%5F31.

[101] Shai Halevi and Victor Shoup. "Bootstrapping for HElib". In: *Advances in Cryptology – EUROCRYPT 2015*. Ed. by Elisabeth Oswald and Marc Fischlin. Lecture Notes in Computer Science. Springer. Springer, Berlin, Heidelberg / Springer, 2015, 641. ISBN: 978-3-662-46799-2 978-3-662-46800-5. DOI: 10.1007/978-3-662-46800-5\\\_25. URL: http://link.springer.com/chapter/10.1007/978-3-662-46800-5%5C%5F25.

[102] Shai Halevi and Victor Shoup. "Design and Implementation of a Homomorphic-Encryption Library". In: *IBM Research (Manuscript)* 6 (2013), 12.

[103] Shai Halevi and Victor Shoup. "Faster Homomorphic Linear Transformations in HElib". In: *Advances in Cryptology – CRYPTO 2018*. Vol. 10991. Cham: Springer International Publishing, 2018, 93. ISBN: 978-3-319-96883-4. DOI: 10.1007/978-3-319-96884-1_4. URL: https://link.springer.com/chapter/10.1007/978-3-319-96884-1%5C%5F4.

[104] Marcella Hastings, Brett Hemenway, Daniel Noble, and Steve Zdancewic. "SoK: General Purpose Compilers for Secure Multi-Party Computation". In: *IEEE Symposium on Security and Privacy (SP)*. Los Alamitos, CA, USA: IEEE Computer Society, 2019, 479. DOI: 10.1109/SP.2019.00028. URL: https://www.computer.org/csdl/proceedings/sp/2019/6660/00/%2066000a462-abs.html.

[105] Vincent Herbert. "Automatize Parameter Tuning in Ring-Learning-With-Errors-Based Leveled Homomorphic Cryptosystem Implementations". In: (2019). URL: https://eprint.iacr.org/2019/1402.

[106] Alberto Ibarrondo and Alexander Viand. "Pyfhel: PYthon For Homomorphic Encryption Libraries". In: *Proceedings of the 9th on Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. WAHC

'21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, 11. ISBN: 9781450386562. DOI: 10.1145/3474366.3486923. URL: https://doi.org/10.1145/3474366.3486923.

[107] Ilia Iliashenko. "Optimisations of Fully Homomorphic Encryption". PhD. Thesis, KU Leuven. PhD thesis. PhD thesis, KU Leuven, 2019. URL: https://www.esat.kuleuven.be/cosic/publications/thesis-316.pdf.

[108] Inpher. *J.P. Morgan leads USD $10 million financing in leading data security and machine learning provider, Inpher*. Accessed: 2020-12-21. 2018. URL: https://www.prnewswire.com/news-releases/jp-morgan-leads-usd-10-million-financing-in-leading-data-security-and-machine-learning-provider-inpher-300743090.html.

[109] *Intel® Software Guard Extensions (Intel® SGX)*. https://www.intel.com/content/www/us/en/architecture-and-technology/software-guard-extensions.html. Accessed: 2022-11-2.

[110] Riddhi Jain. *Data encryption provider IXUP appoints new CEO & MD Marcus Gracey*. Accessed: 2020-12-21. URL: https://itmunch.com/data-encryption-provider-ixup-appoints-new-ceo-md-marcus-gracey/.

[111] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference". In: *Proceedings of the 27th USENIX Conference on Security Symposium*. SEC'18. Berkeley, CA, USA: USENIX Association, 2018, 1651. ISBN: 978-1-931971-46-1. URL: https://www.usenix.org/conference/usenixsecurity18/presentation/juvekar.

[112] Sreekanth Kannepalli, Kim Laine, and Radames Cruz Moreno. *Password Monitor: Safeguarding passwords in Microsoft Edge*. https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge/. Accessed: 2021-7-5. 2021. URL: https://www.microsoft.com/en-us/research/blog/password-monitor-safeguarding-passwords-in-microsoft-edge.

[113] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. “Revisiting Homomorphic Encryption Schemes for Finite Fields”. In: *Cryptology ePrint Archive* (2021).

[114] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. *Semi-parallel Logistic Regression for GWAS on Encrypted Data*. Tech. rep. 2019, 294. URL: https://eprint.iacr.org/2019/294.

[115] Miran Kim et al. “Ultra-Fast Homomorphic Encryption Models Enable Secure Outsourcing of Genotype Imputation”. 2020. DOI: 10.1101/2020.07.02.183459. URL: https://www.biorxiv.org/content/10.1101/%202020.07.02.183459v2.

[116] Nicolas Küchler, Emanuel Opel, Hidde Lycklama, Alexander Viand, and Anwar Hithnawi. “Cohere: Privacy Management in Large Scale Systems”. In: (2023). arXiv: 2301.08517 [cs.CR]. URL: http://arxiv.org/abs/2301.08517.

[117] Samuel Larsen and Saman Amarasinghe. “Exploiting superword level parallelism with multimedia instruction sets”. In: *SIGPLAN Not.* 35.5 (2000), 145. ISSN: 0362-1340. DOI: 10.1145/358438.349320. URL: https://doi.org/10.1145/358438.349320.

[118] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. “MLIR: A Compiler Infrastructure for the End of Moore’s Law”. In: (2020). arXiv: 2002.11054 [cs.PL]. URL: http://arxiv.org/abs/2002.11054.

[119] Kristin Lauter, Adriana López-Alt, and Michael Naehrig. “Private Computation on Encrypted Genomic Data”. In: *Progress in Cryptology - LATINCRYPT 2014*. Springer International Publishing, 2014, 3. DOI: 10.1007/978-3-319-16295-9_1. URL: http://dx.doi.org/10.1007/978-3-319-16295-9%5C%5F1.

[120] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. “Gradient-Based Learning Applied to Document Recognition”. In: *Proceedings of the IEEE* 86.11 (1998), 2278. URL: https://ieeexplore.ieee.org/document/726791.

[121] Yann LeCun and Corinna Cortes. “MNIST Handwritten Digit Database”. In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[122] Dongkwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. “Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Term Rewriting”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2020. New York, NY, USA: Association for Computing Machinery, 2020, 503. ISBN: 978-1-4503-7613-6. DOI: 10.1145/3385412.3385996. URL: https://doi.org/10.1145/3385412.3385996.

[123] Yongwoo Lee, Seonyeong Heo, Seonyoung Cheon, Shinnung Jeong, Changsu Kim, Eunkyung Kim, Dongyoon Lee, and Hanjun Kim. “HECATE: Performance-Aware Scale Optimization for Homomorphic Encryption Compiler”. In: *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2022, 193. DOI: 10.1109/CGO53902.2022.9741265. URL: http://dx.doi.org/10.1109/CGO53902.2022.9741265.

[124] Baiyu Li and Daniele Micciancio. *On the Security of Homomorphic Encryption on Approximate Numbers*. Cryptology ePrint Archive, Report 2020/1533. https://eprint.iacr.org/2020/1533. 2020.

[125] Baiyu Li, Daniele Micciancio, Mark Schultz, and Jessica Sorrell. “Securing Approximate Homomorphic Encryption Using Differential Privacy”. In: *Cryptology ePrint Archive* (2022). URL: https://eprint.iacr.org/2022/816.pdf.

[126] Shimin Li, Xin Wang, and Rui Zhang. “Privacy-Preserving Homomorphic MACs with Efficient Verification”. In: *Web Services – ICWS 2018*. Springer International Publishing, 2018, 100.

[127] Zengpeng Li, Steven D Galbraith, and Chunguang Ma. “Preventing Adaptive Key Recovery Attacks on the GSW Levelled Homomorphic Encryption Scheme”. In: *Provable Security*. Springer International Publishing, 2016, 373.

[128] Mary Loritz. *Paris-based Cosmian raises €1.4 for its platform that analyses encrypted data while keeping it private*. Accessed: 2020-12-21. URL: https://www.eu-startups.com/2019/03/paris-based-cosmian-raises-e1-4-for-its-platform-that-analyses-encrypted-data-while-keeping-it-private/.

[129] Wen-jie Lu, Zhicong Huang, Cheng Hong, Yiping Ma, and Hunter Qu. “PEGASUS: Bridging Polynomial and Non-polynomial Evaluations in Homomorphic Encryption”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021. DOI: 10.1109/sp40001.2021.00043. URL: https://doi.org/10.1109/sp40001.2021.00043.

[130] Ingrid Lunden. “Duality, a security startup co-founded by the creator of homomorphic encryption, raises $16M”. In: *TechCrunch* (2019). URL: http://techcrunch.com/2019/10/30/duality-cybersecurity-16-million/.

[131] Hidde Lycklama, Lukas Burkhalter, Alexander Viand, Nicolas Küchler, and Anwar Hithnawi. “RoFL: Attestable Robustness for Secure Federated Learning”. In: *2023 IEEE Symposium on Security and Privacy (SP)*. 2023.

[132] Hidde Lycklama, Nicolas Küchler, Alexander Viand, Emanuel Opel, Lukas Burkhalter, and Anwar Hithnawi. “Cryptographic Auditing for Collaborative Learning”. In: *NeurIPS ML Safety Workshop*. 2022. URL: https://openreview.net/forum?id=sTrIkf20aTw.

[133] Chiara Marcolla, Victor Sucasas, Marc Manzano, Riccardo Bassoli, Frank H P Fitzek, and Najwa Aaraj. “Survey on fully homomorphic encryption, theory and applications”. 2022.

[134] Charith Mendis and Saman Amarasinghe. “goSLP: globally optimized superword level parallelism framework”. In: *Proc. ACM Program. Lang.* 2.OOPSLA (2018), 1. DOI: 10.1145/3276480. URL: https://doi.org/10.1145/3276480.

[135] Microsoft. *AsureRun*. 2019. URL: https://github.com/microsoft/SEAL-Demo/tree/master/AsureRun.

[136] Alan Mishchenko. “ABC: System for Sequential Logic Synthesis and Formal Verification”. In: (2018). URL: https://github.com/berkeley-abc/abc.

[137] Travis Morrison, Sarah Scheffler, Bijeeta Pal, and Alexander Viand. “Private Outsourced Translation for Medical Data”. In: *Protecting Privacy through Homomorphic Encryption*. Ed. by Kristin Lauter, Wei Dai, and Kim Laine. Cham: Springer International Publishing, 2021, 107. ISBN: 978-3-030-77287-1. DOI: 10.1007/978-3-030-77287-1_7. URL: https://doi.org/10.1007/978-3-030-77287-1%5C%5F7.

[138] Christian Mouchet, Jean-Philippe Bossuat, and Juan Troncoso-Pastoriza. *Lattigo: A multiparty homomorphic encryption library in Go*. https://homomorphicencryption.org/wp-content/uploads/2020/12/wahc20_demo_christian.pdf. Accessed: 2022-11-2. 2020. URL: https://homomorphicencryption.org/wp-content/uploads/2020/12/wahc20%5C%5Fdemo%5C%5Fchristian.pdf.

[139] Christian Mouchet, Juan Troncoso-Pastoriza, and Jean-Pierre Hubaux. “Multiparty Homomorphic Encryption: From Theory to Practice”. In: (2020). URL: https://eprint.iacr.org/2020/304.

[140] Kit Murdock, David Oswald, Flavio D Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. “Plundervolt: Software-based Fault Injection Attacks against Intel SGX”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, 2020, 1466.

[141] Deepika Natarajan, Andrew Loveless, Wei Dai, and Ronald Dreslinski. “CHEX-MIX: Combining Homomorphic Encryption with Trusted Execution Environments for Two-party Oblivious Inference in the Cloud”. In: *Cryptology ePrint Archive* (2021).

[142] Alexander Nilsson, Pegah Nikbakht Bideh, and Joakim Brorsson. “A Survey of Published Attacks on Intel SGX”. In: (2020). arXiv: 2006.13598 [cs.CR].

[143] NuCypher. *nufhe*. 2019. URL: https://github.com/nucypher/nufhe.

[144] *Open Enclave SDK*. https://github.com/openenclave/openenclave.

[145] Charlie Osborne. *IBM launches experimental homomorphic data encryption environment for the enterprise*. Accessed: 2020-12-21. 2020. URL: https://www.zdnet.com/article/ibm-launches-experimental-homomorphic-data-encryption-environment-for-the-enterprise/.

[146] Alex Ozdemir, Fraser Brown, and Riad S Wahby. "CirC: Compiler infrastructure for proof systems, software verification, and more". In: *2022 IEEE Symposium on Security and Privacy (SP)*. 2022, 2248. DOI: 10.1109/SP46214.2022.9833782. URL: http://dx.doi.org/10.1109/SP46214.2022.9833782.

[147] Ozgun Ozerk, Can Elgezen, Ahmet Can Mert, Erdinc Ozturk, and Erkay Savas. "Efficient Number Theoretic Transform Implementation on GPU for Homomorphic Encryption". In: *Cryptology ePrint Archive* (2021).

[148] Rachel Player. "Parameter Selection in Lattice-Based Cryptography". PhD thesis. PhD thesis, Royal Holloway, University of London, 2018. URL: https://pure.royalholloway.ac.uk/portal/files/29983580/2018playerrphd.pdf.

[149] Yuriy Polyakov, Kurt Rohloff, and Gerard W Ryan. *PALISADE Lattice Cryptography Library User Manual (v1.6.0)*. Tech. rep. 2019, 54. URL: https://palisade-crypto.org/documentation.

[150] Luis Bernardo Pulido-Gaytan, Andrei Tchernykh, Jorge M Cortés-Mendoza, Mikhail Babenko, and Gleb Radchenko. "A survey on privacy-preserving machine learning with fully homomorphic encryption". In: *Latin American High Performance Computing Conference*. Springer, 2021, 115.

[151] Ronald L Rivest, Len Adleman, and Michael L Dertouzos. "On Data Banks and Privacy Homomorphisms". In: *Foundations of secure computation* 4.11 (1978), 169. URL: https://people.csail.mit.edu/rivest/RivestAdlemanDertouzos-OnDataBanksAndPrivacyHomomorphisms.pdf.

[152] Hamza Saleem and Muhammad Naveed. “SoK: Anatomy of Data Breaches”. In: *Proceedings on Privacy Enhancing Technologies* 2020.4 (2020), 153. URL: http://isyou.info/jowua/papers/jowua-v10n4-4.pdf.

[153] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. “F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption”. In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO ’21. Virtual Event, Greece: Association for Computing Machinery, 2021, 238. ISBN: 9781450385572. DOI: 10.1145/3466752.3480070. URL: https://doi.org/10.1145/3466752.3480070.

[154] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, and Daniel Sanchez. “CraterLake: a hardware accelerator for efficient unbounded computation on encrypted data”. In: *ISCA*. 2022, 173.

[155] John E. Savage. *Models of Computation: Exploring the Power of Computing*. 1st. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997. ISBN: 0-201-89539-0.

[156] “Microsoft SEAL (Release 3.5)”. In: (2020). URL: https://github.com/Microsoft/SEAL.

[157] Victor Shoup et al. “NTL: A Library for Doing Number Theory”. In: (2016). URL: http://www.shoup.net/ntl/.

[158] N P Smart and F Vercauteren. “Fully homomorphic SIMD operations”. In: *Designs, Codes and Cryptography. An International Journal* 71.1 (2014), 57. ISSN: 0925-1022. DOI: 10.1007/s10623-012-9720-4. URL: https://doi.org/10.1007/s10623-012-9720-4.

[159] Florian Tramèr and Dan Boneh. “Slalom: Fast, Verifiable and Private Execution of Neural Networks in Trusted Hardware”. In: (2018). arXiv: 1806.03287 [stat.ML].

[160] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. “SEALion: A Framework for Neural Network Inference on Encrypted Data”. In: *arXiv preprint arXiv:1904.12840* (2019). URL: `https://arxiv.org/abs/1904.12840`.

[161] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. “SoK: Fully Homomorphic Encryption Compilers”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, 1092. DOI: `10.1109/SP40001.2021.00068`. URL: `http://dx.doi.org/10.1109/SP40001.2021.00068`.

[162] Alexander Viand, Christian Knabenhans, and Anwar Hithnawi. “Verifiable Fully Homomorphic Encryption”. In: (2023). arXiv: `2301.07041 [cs.CR]`. URL: `http://arxiv.org/abs/2301.07041`.

[163] Alexander Viand and Hossein Shafagh. “Marble: Making Fully Homomorphic Encryption Accessible to All”. In: *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. ACM, 2018, 49. ISBN: 978-1-4503-5987-0. DOI: `10.1145/3267973.3267978`. URL: `https://dl.acm.org/citation.cfm?doid=3267973.3267978`.

[164] Biao Wang, Xueqing Wang, and Rui Xue. “CCA1 secure FHE from PIO, revisited”. In: *Cybersecurity* 1.1 (2018), 1.

[165] Alexander Wood, Kayvan Najarian, and Delaram Kahrobaei. “Homomorphic Encryption for Machine Learning in Medicine and Bioinformatics”. In: *ACM Comput. Surv.* 53.4 (2020), 1.

[166] Tiancheng Xie, Jiaheng Zhang, Yupeng Zhang, Charalampos Papamanthou, and Dawn Song. “Libra: Succinct Zero-Knowledge Proofs with Optimal Prover Computation”. In: *Cryptology ePrint Archive* (2019). URL: `https://eprint.iacr.org/2019/317.pdf`.

[167] Zhenfei Zhang, Thomas Plantard, and Willy Susilo. “Reaction Attack on Outsourced Computing with Fully Homomorphic Encryption Schemes”. In: *Information Security and Cryptology - ICISC 2011*. Springer Berlin Heidelberg, 2012, 419.