UiT The Arctic University of Norway

Faculty of Science and Technology
Department of Computer Science

Safe and secure outsourced computing with fully homomorphic encryption and trusted execution environments

Isak Sunde Singh

INF-3981 Master's Thesis in Computer Science - June 2020

“Cryptography is typically bypassed, not penetrated.”
–Adi Shamir

“Yo I might be wrong, definitely trust the thesaurus over this dinosaurus”
–Mads Johansen

Abstract

Increases in data production and growing demands for more computing power leads to the current trend of outsourcing data and computation services to cloud providers. With data breaches and cases of data misuse becoming increasingly common, there is a high demand for secure systems. This, however, conflicts with the current data trust models. A solution to this that is becoming more common is the use of Trusted Execution Environment (TEE), aimed at guaranteeing code and data integrity and confidentiality.

However, it has been shown that TEEs such as Intel's Software Guard Extensions (SGX) are susceptible to several types of side-channel attacks where an adversary may gain information of the code and data within a secure environment, breaking the confidentiality property. There are some ways to counter this, such as using oblivious primitives to hide access patterns which may leak information, but these are inefficient and add performance overhead to computation.

Another way to ensure data confidentiality while simultaneously retaining the ability to perform computations on the data is through the use of Fully Homomorphic Encryption (FHE). FHE allows computing on encrypted data, preserving confidentiality and allowing outsourced computations to untrusted parties such as cloud providers. However, this type of encryption is malleable and lacks integrity protection, making it susceptible to integrity breaches where an adversary could modify the data resulting in a corrupt or incorrect plaintext after decryption.

This thesis implements a library for performing FHE in SGX, written in a memory-safe programming language to strengthen the internal safety of software in SGX and reduce its attack surface. We evaluate our design and show that one can feasibly combine these concepts while providing stronger security guarantees with a minimal development effort.

Acknowledgements

I would like to thank my supervisor Lars Brenna for his insights and feedback on this thesis. I also extend my gratitude to my co-supervisor Anders Gjerdrum for inspiring me to choose to work on the subjects of homomorphic encryption and Intel SGX. Without your recommendations, I would have suffered way less.

To all my classmates, without all of you, I would have not been able to finish this degree. Thank you all for the good and stressful times we have had together in this time and for enduring these past five years with me.

I would like to thank my family for their support over these years.

Mina, thank you for your love and support. You made the most challenging times of writing this thesis joyful.

Contents

Abstract **iii**

Acknowledgements **v**

List of Figures **xi**

List of Tables **xiii**

List of Abbreviations **xv**

1 Introduction **1**
1.1 Thesis Statement . . . 3
1.2 Scope and Assumptions . . . 3
1.3 Context . . . 4
1.4 Method and Approach . . . 5
1.5 Contributions . . . 6
1.6 Outline . . . 6

2 Background **9**
2.1 Homomorphic Encryption . . . 9
2.1.1 Fully Homomorphic Encryption . . . 11
2.1.2 TFHE . . . 13
2.1.3 Integrity Weakness . . . 14
2.1.4 Circuits . . . 15
2.2 Trusted Execution Environments . . . 16
2.2.1 Arm TrustZone . . . 18
2.2.2 Intel SGX . . . 18
2.2.3 SGX's Weaknesses . . . 19
2.2.4 Countermeasures to SGX Weaknesses . . . 20
2.3 Related Work . . . 22
2.4 Summary . . . 23

3 Design **25**
3.1 Homomorphic Encryption Library . . . 25

3.2 Porting the TFHE Library for Homomorphic Encryption . . . 26
3.3 Choosing a Programming Language 28
3.3.1 Undefined Behavior 30
3.3.2 Disambiguating Types and Self-Referential Structures 32
3.3.3 Choices and Alternatives 33
3.4 Summary 34

4 Implementation **35**
4.1 Porting TFHE to Rust 35
4.2 External Libraries 37
4.2.1 Development Tools 39
4.3 Code Makeup 40
4.3.1 Module Structure 41
4.4 Executing Code in SGX 43
4.5 Paillier PHE Implementation 44
4.6 Summary 44

5 Evaluation **47**
5.1 Experimental Setup 47
5.2 Library Evaluation 48
5.2.1 Encryption and Decryption 48
5.2.2 Key Generation 49
5.2.3 Bootstrapping 50
5.3 Evaluation Methodology 53
5.3.1 Yao's Millionaires' Problem 54
5.3.2 Socialist Millionaire Problem 55
5.3.3 Fused Millionaire Problem 55
5.4 Experiment Evaluation 56
5.5 Summary 59

6 Discussion **61**
6.1 Experiment Design Versus Real-World Programs 61
6.2 Improvements to TFHE Implementation 63
6.2.1 Internal Representation 63
6.2.2 Memory Management 64
6.2.3 SIMD and Parallelization 64
6.2.4 Ciphertext Packing 66
6.3 Recent Attacks on SGX 67
6.4 Circuit Optimization 68
6.5 Rust-SGX SDK 69
6.6 Implementation Correctness 70
6.7 Summary 71

7 Conclusion **73**

7.1 Concluding Remarks 73
7.2 Future Work 74

Bibliography **77**

List of Figures

2.1 Original bootstrapping idea 13
2.2 Improved bootstrapping 13
2.3 Homomorphic comparison circuit 16
2.4 Arm TrustZone architecture 18

4.1 Fortanix Rust EDP architecture 43

5.1 Encryption and decryption execution times 48
5.2 Encryption probability density function 49
5.3 Decryption probability density function 50
5.4 Key generation probability density function 51
5.5 Bootstrapping probability density function 52
5.6 Optimized compared to non-optimized bootstrapping 53
5.7 Memory usage of millionaires' problem solution program . . 57
5.8 Millionaires' problem experimental results representation . . 58

List of Tables

5.1 Millionaires' problem experiment results 58

List of Abbreviations

ADT	Abstract Data Type
API	Application Programming Interface
BGV	Brakerski-Gentry-Vaikuntanathan
CKKS	Cheon-Kim-Kim-Song
CPU	Central Processing Unit
DDB	Data-Dependent Branching
EPC	Enclave Page Cache
FFI	Foreign Function Interface
FFT	Fast Fourier Transform
FHE	Fully Homomorphic Encryption
GSW	Gentry-Sahai-Waters
HE	Homomorphic Encryption
HEAAN	Homomorphic Encryption Arithmetic of Approximate Numbers
IND-CCA2	Indistinguishability under Adaptive Chosen-Ciphertext Attack
IND-CPA	Indistinguishability under Chosen-Plaintext Attack
IO	Input/Output

IOT	Internet of Things
LFHE	Leveled Fully Homomorphic Encryption
LHE	Leveled Homomorphic Encryption
LUT	Lookup Table
LWE	Learning With Errors
ORAM	Oblivious RAM
OS	Operating System
PDF	Probability Density Function
PHE	Partially Homomorphic Encryption
PRAM	Protected RAM
RAM	Random-Access Memory
RAW	Read-After-Write
RLWE	Ring Learning With Errors
SDK	Software Development Kit
SEAL	Simple Encrypted Arithmetic Library
SGX	Software Guard Extensions
SHE	Somewhat Homomorphic Encryption
SIMD	Single Instruction, Multiple Data
SLOC	Source Lines of Code
SMPC	Secure Multi-Party Computation
SMT	Simultaneous Multithreading

TCB	Trusted Computing Base
TEE	Trusted Execution Environment
TPM	Trusted Platform Module
UB	Undefined Behavior
WAR	Write-After-Read
WAW	Write-After-Write

1 Introduction

The rise of cloud computing has introduced new ways to design systems. Cloud providers deliver both computational power and storage as a service to consumers, which allows outsourcing computations to a higher degree than before. Data confidentiality and security become increasingly important as consumers process more sensitive data in cloud settings. However, with the systems' current design, a malicious actor such as the cloud provider could access a consumer's data and even alter it and the running program.

Fully Homomorphic Encryption (FHE) is a way to process data while maintaining data confidentiality through encryption. Research in the area exploded after the 2009 doctoral thesis of Craig Gentry [1] showed the first technique for achieving FHE, after nearly 30 years of research since the idea's conception [2]. The concept is compelling, as it allows an untrusted party to compute on confidential, encrypted data, meaning one can securely outsource computations to an untrusted party. The technique allows a plethora of previously in-house computation settings to be outsourced to cloud providers, such as health data processing, financial processing, genome research, and more. The reality is a bit more complicated, as the computing party might still maliciously alter results, since all Homomorphic Encryption (HE) schemes are malleable by design. An altered result is theoretically indistinguishable from a correctly processed result. This situation is something non-homomorphic encryption schemes do not have to worry about, as their primary purpose is providing confidentiality. The actual computations performed on data encrypted using HE are also public, which might be unacceptable in some situations as the

operations themselves might be secret. FHE partially solves the problem of security in a cloud computing setting, in terms of data confidentiality, but leaves certain problems such as data integrity unsolved.

Another advancement in the field of trusted computing is the concept of a Trusted Execution Environment (TEE). A TEE is the concept of an environment that provides both data and code confidentiality. Additionally, it provides security guarantees such as data and code integrity, which resists attacks from unauthorized parties, both in software and hardware. It should provide these guarantees even against a malicious Operating System (OS) or hypervisor. As such, a TEE should, in theory, provide stronger security guarantees than FHE. However, as always, reality is more complicated. TEEs have had troubles with their implementation, as multiple works have found security flaws that breach security guarantees it by definition should provide. A prominent example is Intel's Software Guard Extensions (SGX), which is an attempt to utilize hardware and software support to isolate an environment away from the surrounding system. Several publications demonstrate attacks on Intel's processors and on SGX that breach confidentiality guarantees of the system, sometimes requiring hardware changes to mitigate the attacks [3], [4], [5], [6].

As Intel's SGX requires some hardware interaction in software, programs for the TEE are often written in systems programming languages such as C and C++. These languages are not memory safe, meaning they do not prevent a developer from introducing memory-related bugs that may be exploited by an attacker. Examples of these bugs are memory corruption errors, buffer overflows, uninitialized memory, data races, dereferencing pointers to unallocated memory (e.g., null-pointer dereferencing), or dereferencing a pointer causing an access violation, and much more. These types of errors are among the topmost dangerous security errors [7], and are in some cases the most common security vulnerabilities [8]. These errors might lead to worse results within SGX as they might cause unintended referencing of untrusted memory, cause data leakage to unauthorized parties, or allow a malicious host to hijack the process inside SGX [9]. A programming language that is memory safe disables the opportunity to make these mistakes, leading to stronger security.

In this thesis, we investigate the intersection between these three concepts, by combining the techniques of FHE, an implementation of a TEE, and a programming language that is memory safe. FHE provides cryptographically secure data confidentiality while lacking integrity guarantees, and its weaknesses align with the strengths of SGX, which is data and code integrity. At the same time, we mitigate the potential confidentiality weaknesses that SGX has. By also using a programming language that is memory safe, our design excludes another class of security-vulnerabilities that are memory-safety related. Specifically, we compare the relative performance metrics of a program using FHE

outside and within SGX, all written in a memory-safe programming language. By comparing the relative performance difference, we can determine if the hybrid system is feasible in terms of performance while retaining more robust security and safety guarantees than the alternative of using FHE or SGX on its own. To the best of our knowledge, this is the first work that combines a TEE with FHE to cover integrity weaknesses of FHE. Nor do we know of work that emphasizes on memory safety through compile-time guarantees relating to FHE. There exists earlier work using FHE schemes in a TEE, but for the sake of removing the need for the expensive bootstrapping operations that common FHE schemes require [10], by storing secret keys in a TEE. There exists other work which uses Partially Homomorphic Encryption (PHE) inside a TEE, though PHE is limiting in comparison to FHE, as we will show.

1.1 Thesis Statement

The integrity weaknesses of Homomorphic Encryption schemes are mitigable using a Trusted Execution Environment. Programming languages that are memory safe provide stronger security guarantees and can mitigate substantial classes of security vulnerabilities.

This thesis shall investigate the feasibility of developing a system for performing fully homomorphic encryption in a trusted execution environment, using a memory-safe programming language.

To investigate whether such a hybrid design is feasible or not, this thesis will build a prototype and compare its performance to a system using Fully Homomorphic Encryption outside the boundaries of a Trusted Execution Environment.

1.2 Scope and Assumptions

The underlying foundation of our system is bipartite by design. First, we rely on the mathematics and computational hardness assumptions behind the TFHE [11] FHE scheme to provide confidentiality of data encrypted with it. At the same time, this scheme, and as we show, any other FHE scheme, has weaknesses related to security, notably data integrity and malleability. Secondly, we rely on Intel SGX as a provider of a TEE, to guarantee both code and data confidentiality and integrity. However, as we show, this system also has several weaknesses, some of which entirely break the data confidentiality guarantees. Our focus in this thesis is a hybrid approach, where we combine the two

concepts to cover each of the weaknesses they entail. As such, we conjecture that the hybrid approach should be a safer alternative than using either system separately, as SGX provides data and code integrity, and TFHE provides data confidentiality. We also conjecture that the hybrid system will be relatively slow in execution time, as FHE is known for requiring a substantial amount of processing power. We will implement a prototype for performing FHE using the TFHE scheme within SGX, and evaluate the performance and compare the relative performance change of our hybrid system to a system using only FHE, at the loss of certain security guarantees.

To achieve this, we need to ensure we can perform homomorphic operations on ciphertexts from the TFHE schemes while using SGX. Ensuring memory safety is important, as one can still mistakenly experience memory corruption vulnerabilities considering most SGX software is still developed using unsafely typed languages such as C and C++. In security-related software, it is vital to ensure the validity and correctness of programs and the absence of security-related vulnerabilities, which often arises with languages that are not memory-safe. Thus we will explore the use of a memory-safe programming language in our construction while having a secondary focus that our construction does not present significantly reduced performance characteristics than a non-memory safe language would. Using this type of language will not only allow but also enforce stronger security guarantees at the application layer.

We will not perform a detailed security analysis of our design, as it is a complex and complicated task. It also requires comprehensive work and in-depth knowledge of security analysis techniques, which we do not have at hand nor do we have enough time.

1.3 Context

This thesis is written in the context of the Corpore Sano center[1], which conducts research in the convergence of computer science, sports science, and medical research.

As part of this work, Corpore Sano has conducted research projects into secure, distributed computation, including the Diggi [12] framework, and has come up with performance principles for Intel SGX [13].

1. https://corporesano.no/

1.4 Method and Approach

The Task Force on the Core of Computer Science presented in their final report a way to divide the discipline of Computing into distinct paradigms [14]. The three major paradigms are:

Theory which is rooted in mathematics and consists of four steps, followed in the development of a coherent and valid theory:

1. First, one characterizes the objects of study, or *definition*.
2. Then, hypothesize possible relationships between them, or *theorem*.
3. Further, determine whether the relationships are true, or *proof*.
4. Lastly, interpret the results.

A mathematician expects to iterate on these steps, as they encounter errors or inconsistencies.

Abstraction which is rooted in the experimental scientific method and follows four steps when investigating a phenomenon:

1. Form a hypothesis on the phenomenon.
2. Construct a model to make a prediction.
3. Design an experiment to collect data.
4. Analyze the results.

A scientist expects to iterate these steps, as they encounter problems such as when a model's predictions disagree with experimental evidence. *Modeling* is another word for this paradigm.

Design which is rooted in engineering and consists of four steps, followed in the construction of a system to solve a problem:

1. State the requirements.
2. State the specifications.
3. Design and implement the system.

4. Test and evaluate the system.

An engineer expects to iterate on these steps, as they encounter issues such as the system not upholding requirements to a satisfactory level.

In this thesis, we work within the last paradigm, design. We state the requirements and specifications of the system associated with our conjecture. Further, we present a design for a system, implement the requirements needed for our system, and then create a prototype based on the design. We then evaluate our prototype through a series of experiments and benchmarks, to assert the conformity to the requirements and specifications set.

1.5 Contributions

This thesis contributes by providing an implementation of the TFHE [11] cryptosystem, a FHE scheme, written in Rust. The library uses pure Rust and has few dependencies. Since the primary goal of this thesis is to create a safe way to perform FHE in SGX, the library works within SGX without modification. The library is open-source, continuously maintained, and made available online through a repository at the owner's GitHub account[2], and is additionally appended to this thesis as part of the source files.

1.6 Outline

The remainder of this thesis is structured as follows:

Chapter 2 on page 9 details the background of HE and FHE and its limitations, specifically through the TFHE scheme [11]. It further details the concept of a TEE, specifically Intel SGX [15], and its problems and weaknesses. It further describes the attractiveness of combining these concepts to achieve a more secure system. Additionally, it outlines some related work to our thesis.

Chapter 3 on page 25 details our choice of porting the existing TFHE [16] library, and the reasoning behind choosing the Rust programming language to implement this. Further, it describes the design of our system.

Chapter 4 on page 35 describes our implementation of the FHE library and

2. `https://github.com/IsakSundeSingh/tfhe`

the choices made to ensure it meets the requirements of execution within SGX.

Chapter 5 on page 47 details the setup of our experiments and systems, and the methodology behind the experiments. If further goes on to present and describe the results of our experiments.

Chapter 6 on page 61 details the findings of our experiments, possible reasons for the observed behaviors, the security aspects of our design, and choices and alternatives that could benefit our system.

Chapter 7 on page 73 summarizes this thesis and provides concluding remarks on our findings. Additionally, the chapter poses areas for improvement, and thoughts and possibilities for future work.

2 Background

Section 2.1 details how and what Homomorphic Encryption (HE) is and Section 2.2 on page 16 outlines a Trusted Execution Environment (TEE) and its weaknesses. Section 2.3 on page 22 describes related work. Finally Section 2.4 on page 23 summarizes this chapter.

2.1 Homomorphic Encryption

Homomorphic encryption allows performing computations on ciphertexts, outputting another ciphertext where the decryption of it is the same result as if one performs the computation to the unencrypted plaintext itself. HE is useful as it makes it possible for an untrusted party to perform operations on data while retaining the confidentiality of the data. Homomorphic properties of encryption functions are common and are seen in ElGamal [17], unpadded RSA [2], Paillier [18] and more. These encryption schemes, as with most encryption schemes, are homomorphic within some space. As an example, we can have a simple homomorphic system:

$$\mathcal{E}(a) \oplus \mathcal{E}(b) = \mathcal{E}(a + b) \mod q \tag{2.1}$$

Where $\mathcal{E}$ is some encryption function, a and b are some integer plaintexts, so

$\mathcal{E}(a)$ is the encryption of a. q is the modulus of the ciphertext space, meaning a ciphertext is always in the range $[0, q)$ and $q \in \mathbb{Z}$, or written as $\mathbb{Z}_q$. $\oplus$ is a binary operation on two ciphertexts, in this case, homomorphic to the plaintext addition operator. This ciphertext space has implications that affect computations, meaning a ciphertext that overflows this modulus and wraps around loses information. The example in equation 2.1 on the preceding page is an encryption system which is homomorphic under addition. As this system is homomorphic under a single operator, it is known as a Partially Homomorphic Encryption (PHE) system. These encryption systems are useful, but as it is homomorphic under a single operator, they are only instrumental in certain circumstances. PHE under addition can be used for encrypted summation systems, encrypted vote counting and more, but are not more generally applicable. If an encryption system is homomorphic under more than one operator, such as both addition (+) and multiplication ($\cdot$), they are called Somewhat Homomorphic Encryption (SHE) systems. Research articles often conflate PHE and SHE systems erroneously. SHE systems are a lot more applicable as one can do any Boolean operation if using integer arithmetic modulo 2, that is in the $\mathbb{B}$ space. The set arithmetic operators + and $\cdot$ in this space can be considered the logic operators exclusive-or (XOR) and conjunction (AND). The set of these two operators conjoined with a constant value of 1, $\{AND, XOR, 1\}$, is functionally complete, also called a universal set because it can generate one. If a set of operators is functionally complete, it means that all possible truth tables are expressible using a combination of the members of the set. That is, any logical circuit is constructible using this set. That means that if an encryption system is homomorphic under both addition and multiplication, it is functionally complete and can express any circuit. A complete guide to HE and its surrounding terms are found in [19].

As seen in equation 2.1 on the previous page, the ciphertexts are in some integer space, in this case $\mathbb{Z}_q$. This space leads to some limitations on the computations that are achievable. That is, say the encryption of two plaintexts, the plaintexts represented as two numbers a and b, are added together. That is: $\mathcal{E}(a) + \mathcal{E}(b) = \mathcal{E}(a + b)$. The resulting encryption is only correct if the encrypted sum $\mathcal{E}(a + b)$ is less than the modulus q. Otherwise the result will wrap around, and information is lost, similarly to integer overflow in computers. The wrap-around leads to some limitations with SHE as only a finite amount of computations can be performed on a ciphertext before losing information or producing incorrect results. This problem is inherent with the way these cryptosystems work, so a different technique is required to perform more than the limited amount of computations.

2.1.1 Fully Homomorphic Encryption

Fully Homomorphic Encryption (FHE) is another step up from SHE where there is no finite limit to the number of operations permitted on a ciphertext, allowing an arbitrary amount of computations. Gentry [1] proposed in 2009 the first method to create a FHE scheme by converting SHE schemes to FHE schemes through a process known as bootstrapping. There is also another type of HE known as Leveled Homomorphic Encryption (LHE), which is less powerful than FHE, but more efficient as it does not require the expensive bootstrapping procedure. LHE schemes allow computation up to a certain limit, usually depending on parameters such as the key size. SHE and LHE are often mistaken to be the same, but there are differences. A LHE scheme has the advantage of not having to perform the bootstrapping operation, so it has a fast evaluation but requires one to know the depth of a circuit to perform in advance to estimate the parameters of the setup. A FHE scheme, on the other hand, has slow evaluation times, due to requiring the bootstrapping procedure, but can evaluate any circuit, no matter the depth of it. There is also the matter of Leveled Fully Homomorphic Encryption (LFHE), which is a small step in-between LHE and FHE and often conflated with LHE. We do not detail LFHE further as it is not relevant to our work. A more detailed explanation and comparison between SHE, LHE, LFHE, and FHE is found in [19].

Bootstrapping

Explaining the bootstrapping process in the mathematical and technical sense is challenging, and the process also varies from different encryption schemes. It may also be confusing to see how processing data without having access to it even would work. Thus it is useful to look at a physical analogy. The analogy is paraphrased from [20].

Consider Alice, which owns a jewelry store. She has valuable materials that she makes jewelry from and wishes that her workers can make the jewelry for her. However, she distrusts her workers and does not wish them to have access to the materials or finished products. Alice has an idea for how to ensure her workers can produce jewelry without risking them stealing it. She locks her materials in a transparent impenetrable glovebox and locks it with a key for which only she has access. She gives the box to the workers and allows them to assemble jewelry using the gloves. As the box is impenetrable, the worker has no reason not to return it with the finished jewelry inside. Alice can then unlock the box with her secret key. This process is analogous to homomorphic encryption, where the glovebox with the materials locked inside is the encryption of some data. The glovebox with the finished jewelry inside is analogous to the encrypted result, which Alice decrypts using her secret key.

The gloves represent the homomorphism, i.e., the operations performable on the encrypted data. A lack of access to the data one computes defines HE. In this analogy, the lack of physical access to the materials inside the impenetrable box represents this.

The gloveboxes that Alice has defects and the gloves become stiff and useless after a certain amount of time when worked. Importantly, the gloves stiffen before the workers finish making the jewelry pieces. The gloveboxes do, however, have a one-way slot to insert things, allowing her to insert one box into another. Alice realizes she could have a worker complete a whole set of jewelry using several boxes and some keys. She gives a worker a glovebox, b_1, containing the required materials, while also handing them several additional boxes. These additional boxes contain keys, where the second box, b_2, contains the key to the first box, k_1, the third contains the key to the second (k_2), and so forth. A worker can then assemble the jewelry until the gloves become unusable, insert the box into the second, unlock it, extract the partly assembled jewelry and materials, and then continue the process, inserting b_i into b_{i+1} and then unlocking it with the key k_i contained within it. The process continues until the worker has finished producing the jewelry. One thing to notice is that this process does not work if the action of opening a glovebox within another box takes more than or equal to the time it takes for the gloves to stiffen.

The analogy of Alice's jewelry store is not perfect and has some flaws. Mainly, a worker may effortlessly determine if a glovebox contains a particular set of materials, so it lacks semantic security. Additionally, the input plaintexts, $m_1, m_2, ...$, may be larger than the output after computing a function f on the inputs, $f(m_1, m_2, ...)$. Moreover, it is highly unlikely that the boxes would physically fit within one another.

The process of opening a glovebox within another glovebox is analogous to evaluating the cryptosystem's decryption function while itself is encrypted. Any cryptosystem that has the self-referential property of being capable of handling its decryption function is called bootstrappable. If some system is bootstrappable, one can use it to construct a FHE scheme.

The old bootstrapping procedure proposed by Gentry [1], seen in Figure 2.1 on the facing page, is improved by work such as [21] and furthermore in FHEW [22], which TFHE builds upon, improving asymptotic time complexity of the decryption procedure in the security parameter to quasi-quadratic. An overview of the revised bootstrapping procedure is seen in Figure 2.2 on the next page.

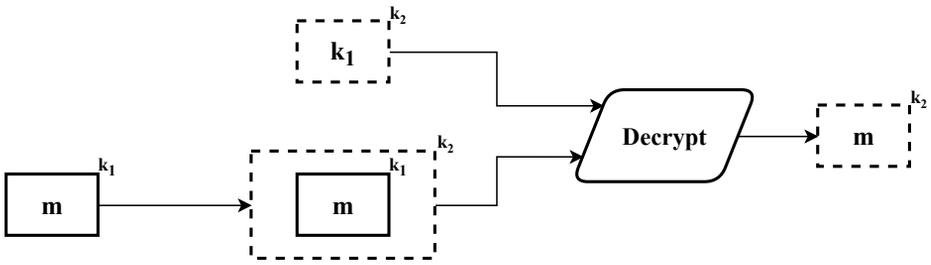


Figure 2.1: The original bootstrapping idea. The box with solid line around the message $\boldsymbol{m}$ represents the encryption of $\boldsymbol{m}$ with key $\boldsymbol{k}_1$, the same contraption inside a box with dotted outline represents the layered encryption of it with respect to the key $\boldsymbol{k}_2$. The dotted-line box with $\boldsymbol{k}_1$ inside represents the encryption of key $\boldsymbol{k}_1$ with respect to key $\boldsymbol{k}_2$. These encryptions are passed through the decryption circuit, resulting in an encryption of the original message $\boldsymbol{m}$ with respect to key $\boldsymbol{k}_2$. Security is preserved throughout the decryption circuit as the message $\boldsymbol{m}$ is encrypted by the second layer of homomorphic encryption.

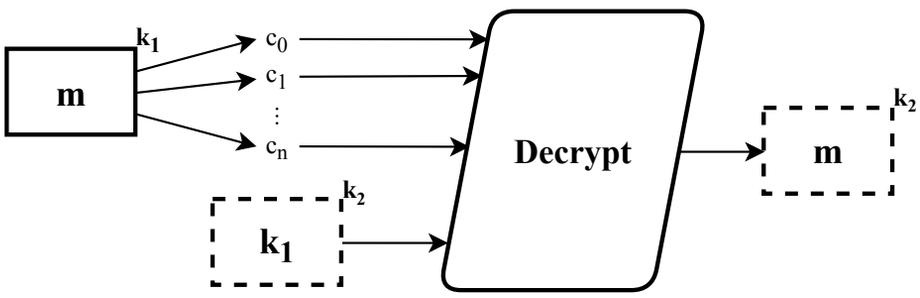


Figure 2.2: A new and revised bootstrapping procedure. Instead of re-encrypting noisy ciphertexts by adding an additional layer of homomorphic encryption, it evaluates the decryption circuit directly on the ciphertext's bits by using the encryption of the key $\boldsymbol{k}_1$ with respect to key $\boldsymbol{k}_2$.

2.1.2 TFHE

This work focuses on TFHE [11], or Fast Fully Homomorphic Encryption over the Torus, a symmetric lattice-based FHE scheme. It works by representing polynomials with coefficients over $\mathbb{T}$, the set of real numbers modulo 1, or $\mathbb{R}/\mathbb{Z}$. The original library implementation, also called TFHE [16], is mainly designed to work with computing on bits, in contrast to the other popular schemes Homomorphic Encryption Arithmetic of Approximate Numbers (HEAAN) (also commonly called Cheon-Kim-Kim-Song (CKKS), after the authors) [23] and Brakerski-Gentry-Vaikuntanathan (BGV) [24], named after the authors. The HEAAN scheme is designed for working with approximate numbers, as the plaintext space is within the complex numbers, $\mathbb{C}$. An implementation of the

HEAAN scheme can be found in the HEAAN library[1]. The BGV scheme is more appropriate than the others for use with integer arithmetic. The scheme is applicable for building circuits with, but is more complex in use and requires the developer to have considerable knowledge of the inner workings to establish an efficient HE program. An implementation of the BGV scheme can be found in Microsoft's Simple Encrypted Arithmetic Library (SEAL) library[2] and in HElib[3], both of which also implements the HEAAN scheme. All of these schemes build on the Learning With Errors (LWE) problem or the ring-variant, Ring Learning With Errors (RLWE). LWE is theorized to be post-quantum safe, meaning there are no known strong attacks using quantum computing, unlike other common encryption schemes such as RSA which are based on the prime number factorization problem, which can be solved easily on a quantum computer using Shor's algorithm [25].

2.1.3 Integrity Weakness

A homomorphic encryption scheme can be described more generally as:

$$E_k(x) \otimes E_k(y) = E_k(x * y) \tag{2.2}$$

In Equation 2.2 $E_k(x)$ is the encryption of the plaintext x with the key k. $*$ is some binary operation between plaintexts, and $\otimes$ is a *lifted* version of $*$, operating in the ciphertext space. Note that the lifted operator $\otimes$ does not necessarily involve the same operations as the $*$ operator, meaning it may have a higher complexity. Assume an attacker knows x and y in addition to their encryptions $E_k(x)$ and $E_k(y)$, and there exists some pair (x, y) such that $x * y \notin \{x, y\}$. The attacker can then compute $E_k(x) \otimes E_k(y)$ to obtain a ciphertext C, that corresponds to the encryption of $x * y$, which beforehand was assumed to be different than x and y. Because of this, the attacker has obtained a ciphertext that corresponds to a plaintext, $x*y$, they know, but whose ciphertext they have not observed previously. An encryption system is malleable when some attacker can transform a ciphertext into another ciphertext, and then decrypting it to a related plaintext. Thus, any HE system is, by definition, malleable, as opposed to non-malleable cryptosystems [26].

As HE allow processing encrypted data, they require a publicly known evaluation key to process the data. With this key, even symmetric HE schemes such as TFHE have similarities to asymmetric schemes in that one can publicly create ciphertexts using the evaluation key. Allowing the creation of ciphertexts by

1. https://github.com/snucrypto/HEAAN
2. https://www.microsoft.com/en-us/research/project/microsoft-seal/
3. https://github.com/homenc/HElib

a publicly known key means an attacker can use this to generate multiple ciphertexts and thus extract information about which plaintexts some ciphertexts encrypt (if they are encrypted using the same key). Thus an attacker could eventually break an encryption system by cracking the decryption key (using the information of the ciphertexts' similarity). To prevent this situation, a minimum required security notion is required. *Semantic security* is a notion which implies that only a negligible amount of information can be extracted from a ciphertext without access to the decryption key [27]. Semantic security implies that the encryption scheme is probabilistic, meaning two encryptions of the same plaintext are inequal. A further class of security notions is Indistinguishability under Chosen-Plaintext Attack (IND-CPA). This security means that if an adversary chooses two messages of the same length and sends them to an encryption oracle, and a ciphertext is returned, the adversary should not be able to guess which plaintext was encrypted with a higher probability than if the adversary randomly guessed. This means semantic security implies the encryption scheme is IND-CPA. It is a difficult concept to understand, but it means that semantically secure, but malleable encryption schemes are secure under standard IND-CPA but not secure under Indistinguishability under Adaptive Chosen-Ciphertext Attack (IND-CCA2) [28]. It has been shown that some encryption schemes that are IND-CPA become insecure when they encrypt their own decryption key, called *circular security* [29]. As HE schemes encrypt their decryption key as part of the bootstrapping process, they have circular security. The implications of circular security properties of HE schemes and their relation to the different notions of security are not fully understood, but a full explanation of different security notions and their implications are seen in [30] and [28]. By processing ciphertexts within a TEE, an adversary cannot modify nor even read the ciphertext, eliminating the issue of malleability and thus providing stronger security.

2.1.4 Circuits

For programs to be evaluable on homomorphically encrypted data, homomorphic primitives are required to represent them. As mentioned, having a functionally complete set of operations means any circuit is expressible. In this sense, a circuit, specifically a boolean circuit, is the mathematical model behind digital logic circuits used in integrated circuits. However, all data processed through this circuit is homomorphically encrypted. That all data is encrypted means that when a gate operates on a value or values, it does not know the values before processing, nor the result.

As a consequence of this, we cannot perform actions dependant on the resulting computation, or Data-Dependent Branching (DDB). That means it cannot perform unbounded loops or any conditional evaluation. Unfortunately, and

fortunately, in terms of security, this disallows an executor of a circuit with accompanying data to perform more than a single branch of a program. It makes it a lot harder to convert general programs into a circuit for use in HE and is limiting for the user. A possible way to handle this is to evaluate all branches of an execution path and transform the code accordingly. Transforming code in this matter is, however, challenging at best, if not impossible. It is also a manual process.

Although it may seem too limiting for anything to compute on homomorphically encrypted data, specific techniques are possible to use to achieve a higher degree of functionality. An example is string matching, file retrieval, and even sorting encrypted data, although relatively slowly [31]. As an example, Figure 2.3 depicts a circuit for comparing two bits homomorphically. A single circuit like this can have the input $r_{i-1} = 1$ to calculate $a_i \leq b_i$, where a_i and b_i are two bits. One can then chain these circuits together to account for the number of bits in a bit-string to compare.

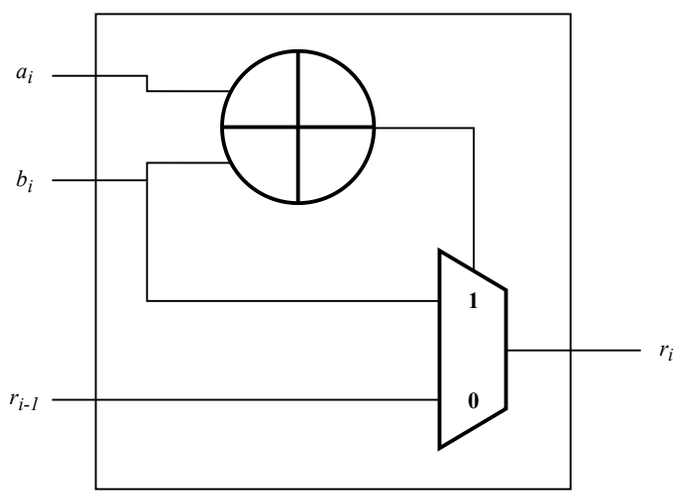


Figure 2.3: Circuit diagram for homomorphic comparison circuit. a_i and b_i are the current bits being compared, while r_{i-1} is the carried result (starting value 1), and r_i the new carried result. After a chain of these r_i will be 1 if B is greater than or equal to A.

2.2 Trusted Execution Environments

Trusted computing is a definition of systems that aid in achieving secure computations, privacy, and data protection. A Trusted Platform Module (TPM) is an example of a trusted computing component in hardware. It provides an interface that presents evidence if someone tampered with cryptographic keys

or some other data stored within it. A TPM only supplies a predefined API and does not provide an isolated execution environment, thus reducing its practicality. Later work in hardware introduces the concept of a TEE, reinforcing the definition of trusted computing. A TEE is an isolated environment guaranteeing to protect both code and data loaded within it in terms of confidentiality. Various definitions of TEEs have been given, that contradict each other in some areas, but are comparable in others [32], [33], [34], [35]. Sabt *et al.* compares these definitions and formalizes a description for TEEs by building on the notion of a *separation kernel*, first described by Rushby [36] and defines four main security policies:

Data (spatial) separation Data within one partition cannot be read or modified by other partitions.

Sanitization (temporal separation) Shared resources cannot be used to leak information to other partitions.

Control of information flow Communication between partitions cannot occur unless explicitly permitted.

Fault isolation Security breach in one partition cannot spread to other partitions.

Building on these definitions, they define a TEE as "[...] a tamper-resistant processing environment that runs on a separation kernel". A TEE should guarantee the authenticity of the executed code, including the integrity of the runtime state, such as Central Processing Unit (CPU) registers. It guarantees the confidentiality of code, data and runtime state persisted to secondary memory, e.g. through encryption. Moreover, a TEE should have the possibility of providing remote attestation, proving trustworthiness for third-parties. Contents within a TEE should be able to be updated securely. They should resist against all software attacks and physical attacks that are performed against main memory too. Attacks performed through backdoor security flaws are not possible. Consequently, a TEE should be secure in a way that even an Operating System (OS) is separated and cannot access nor modify it.

These conditions warrant that tasks can be sent to third-parties and executed within a TEE, without requiring trust in that party. Not requiring trusting the computing party allows for data-sensitive tasks to be outsourced, given they provide a TEE.

2.2.1 Arm TrustZone

Arm TrustZone [37] is a TEE implemented in Arm's processors. These TEEs are designed and targeted at embedded computing as they produce chipsets mostly for embedded platforms. TrustZone separates execution of code into two separate groups: the non-secure *normal* execution environment, or *world*, and a secure execution environment, that is trusted and certifiable. TrustZone provides isolation of separate hardware components while adding only a low impact on system performance. It allows securing a software library or an entire OS within the secure execution environment. TrustZone blocks software running in the non-secure environment from accessing the secure environment and any of its resources. Tasks that require the transmission of data or operations across the border between secure and non-secure environments must pass through monitor software, called *secure monitor* or *core logic* (in Cortex-A or Cortex-M processors, respectively). TrustZone also provides a secure boot sequence to verify secure boot images, and can be authenticated using cryptographic keys. An illustration of TrustZone's architecture overview can be seen in Figure 2.4.

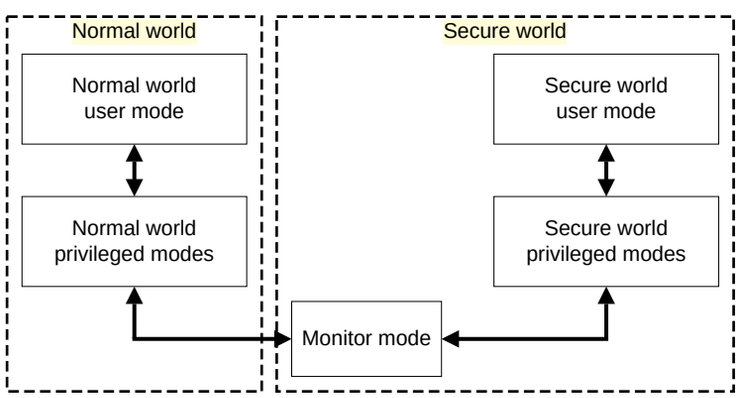


Figure 2.4: Arm TrustZone schema, with user-modes and privileged modes in both the normal and secure worlds, figure taken from [38, p. 38].

2.2.2 Intel SGX

Intel Software Guard Extensions (SGX) [15] is an implementation of a TEE in Intel's processors. SGX TEEs has execution environments which Intel refers to as *enclaves*. Intel's SGX is architecturally different from Arm's TrustZone, as TrustZone has execution split into two environments that are secure and non-secure, while an Intel CPU supporting SGX may have several enclaves at once. The support for multiple enclaves makes SGX more relevant to cloud computing in contrast to Arm TrustZone, in addition to Intel's narrower focus towards server and workstation processors.

One of the more essential features of SGX is known as remote attestation. By utilizing remote attestation, an enclave may prove its identity and authenticity to the user, that it has not been tampered with, that it is executing on genuine Intel hardware and is running the latest version of the software. When an enclave performs remote attestation, it communicates through an authenticated channel with some service provider which then verifies the enclave through an attestation service.[4] The attestation service is at the time of writing provided by Intel, but more recent work allows third-parties to provide such a service [39].

SGX needs to ensure that memory cannot be read nor modified by other processes, kernel threads and not any hypervisor either. To achieve this, SGX locks memory through a system called Protected RAM (PRAM) which can only be accessed by the owning enclave. Additionally, Intel provides a memory encryption engine that supports encrypted paging when memory requirements exceed the limits of PRAM, storing pages in a specific region of physical memory known as the Enclave Page Cache (EPC). SGX enables the OS to virtualize the EPC and page its encrypted contents securely to other storage.

2.2.3 SGX's Weaknesses

A TEE such as SGX provides several mechanisms to protect the contents within the enclaves, however many are rooted in hardware. Hardware modules such as the TPM are designed to be tamper-resistant, but work shows that they might still be susceptible to physical attacks, such as the cuckoo attacks where an adversary tries to gain access to hardware encryption keys [40]. There are several ways for an adversary to physically attack hardware components to extract information, through power-monitoring (or power-tweaking) attacks such as Plundervolt [41], acoustic cryptanalysis attacks such as in [42] where they placed microphones in the vicinity of a computer, electromagnetic attacks, optical attacks and more. We do not consider these types of physical hardware attacks in this thesis.

In addition to physical attacks, several software-based attacks exists. Formally, these software and physical attacks are known as side-channel attacks. These software-based side-channel attacks range from page-fault-based attacks, cache-based attacks and interface-based attacks [3], [4], [5], all of which impair the confidentiality of SGX. Some of these attacks are not specific to SGX and are general enough to work against processors supporting Simultaneous Multithreading (SMT) [43]. Many researchers focus on finding defences or

4. A complete explanation of this process can be found at `https://api.trustedservices.intel.com/documents/sgx-attestation-api-spec.pdf`

counters to these types of attacks, such as Déjà Vu [44], which tries to detect side-channel attacks conducted by a privileged attacker (such as a malicious OS or virus-infected OS).

Even when one does not consider these side-channel attacks possible, one still has to place trust in Intel, Arm, or another company, as the manufacturer of the TEE. In the case of Intel SGX, the TEE comprises of several hardware mechanisms in addition to software systems and a Software Development Kit (SDK). Trusted Computing Base (TCB) is introduced by Rushby in [36], who defines it as the set of all software- and hardware-critical components to some system's security. It is sensitive in that bugs and vulnerabilities might lead to a system failure, leak private information, or allow unprivileged individuals access. Essentially, a trusted computing base is an attack surface for malicious adversaries; hence one would want to minimize it if possible. As mentioned, SGX's TCB consists of the hardware and the enclave SDK. A TCB is frequently measured in terms of Source Lines of Code (SLOC), being the number of lines of actual code (excluding comments and insignificant white-space) a codebase contains. The Intel SGX Linux SDK consists of around 200 k SLOC at the time of writing.

2.2.4 Countermeasures to SGX Weaknesses

Some types of side-channel attacks that exploit access-pattern information leakage can be protected against using techniques such as Oblivious RAM (ORAM). ORAM [45] can be seen as a compiler that transforms memory accesses of a program into a program where the distribution of memory accesses differs (is independent) from the original program while preserving the semantics of the program. It may add accesses to external memory to hide real accesses. One can implement an Abstract Data Type (ADT) as an oblivious data structure. An example can be a balanced binary tree, where a lookup for an element usually would instantly be returned and has time complexity $O(\log n)$ (with n elements). An early return will leak the location of the element within the tree, either which node or which block of memory it is within, as the last fetched node contains the element. In an oblivious implementation, one could scan the entire tree before returning the element, which would access all nodes, hiding the element's location within the tree. However, this implementation is inefficient and turns the time complexity to $O(n)$ instead. ORAM hides memory accesses in a similar, but more general way.

Path ORAM [46] improves upon regular ORAM and has a low space overhead and in some cases, asymptotically improved performance compared to earlier work. Circuit ORAM [47] further improves the techniques and gives an implementation with a complexity near the theoretical lower-bound.

An example of oblivious primitives in action is seen in ZeroTrace [48], which strengthens security against access-pattern side-channel attacks in SGX. It provides an efficient block-level memory controller to hide access patterns. Both Path ORAM and Circuit ORAM are implemented and gave in some situations only a logarithmic overhead in bandwidth costs between enclave code and ORAM servers. ZeroTrace mitigates considerable weaknesses in SGX as it protects against shared resource and page-fault related attacks by converting programs into oblivious representations.

Another example of oblivious memory primitives in SGX is Oblix [49], an oblivious search index. The authors introduce something they call doubly-oblivious techniques. They call it doubly-oblivious as it ensures that accesses to external servers as well as the ORAM clients internal memory are oblivious. An ORAM client is a program which accesses an external resource (an ORAM server) through oblivious techniques. These doubly-oblivious techniques ensure that even if an adversary were to observe accesses to a client's internal memory, it could learn no information on the data. Oblix additionally designs oblivious algorithms that are more efficient than earlier work and implements a contact number discovery service akin to Signal's service implemented in SGX as a demonstration [50]. They use different techniques than Signal, but achieve speedups ranging from $\sim 9\times$ to $\sim 140\times$ faster while strengthening security, by utilizing the doubly-oblivious techniques, at the same time.

Although researchers always find some way to mitigate parts of attacks on SGX, there is always someone else finding a new way to breach the security guarantees. The CacheOut [6] attack, which exploits the fact that hardware-cache that is flushed and overwritten still can be recovered. Their attack can even selectively choose parts of data to leak with relatively high efficiency, unlike previous attacks where the attacker could only observe the leaked data the CPU enclave was currently accessing. This attack requires hardware fixes and proves once again that SGX enclaves do not fully protect the confidentiality of data and code in enclaves, and that other protective measures are required.

Another attack on SGX which build upon the CacheOut attack is SGAxe [51]. SGAxe exploits the CacheOut attack to compromise both the confidentiality but also the integrity of an enclave's memory. The attack extracts the secret attestation key used by enclaves to prove that they are genuine, meaning a malicious attacker such as a malicious cloud vendor could pass a fake enclave for a real one, tricking the client. This attack compromises many security guarantees needed in our hybrid TEE and FHE solution, but most importantly, it compromises the integrity guarantees required for our system to work.

Also very recently, a new class of attacks against Intel CPUs was discovered, with the attack known as LVI (Load Value Injection) [52]. The attack builds

on the Meltdown [53] attack to inject the attacker's data into the victim's data stream. This vulnerability breaches the data integrity guarantees that SGX should provide as it opens the possibility for the victim's code to execute on the attacker's data, breaking all the correctness guarantees of the user's code. Additionally, it might lead to a denial-of-service attack by injecting data structured in a way the user's code did not expect, leading to software crashes. The performance impact that software mitigations to this attack impose is potentially up to 19×.

2.3 Related Work

Drucker and Gueron [54] state that most secure cloud database solutions tend to provide confidentiality and integrity of data use either a TEE, encrypting data before sending it to the cloud or encrypt it using HE. They show that combining a TEE and using HE is feasible and does not need to rely on the TEE for confidentiality purposes. For security, they consider two attack types; attackers from outside the cloud server, or attackers from within, usually with administrator privileges. HE is malleable and is thus susceptible to integrity attacks. They compare their work to CryptDB [55] and MrCrypt [56] which both use PHE, but lack integrity security for both code and data. Drucker and Gueron combine the PHE scheme Paillier [18] (allows the addition of ciphertexts and multiplication of ciphertexts with plaintexts) and SGX, where SGX provides integrity of code and data (in addition to some confidentiality guarantees, side-channel attacks aside). The Paillier cryptosystem ensures data is private and provides confidentiality, even within the enclave. The combination allows the system to place less trust in Intel, as the Paillier cryptosystem guarantees confidentiality for the encrypted data while allowing some computations. They experiment with an employee salary database where the Paillier cryptosystem encrypts salaries. By selecting the sum of the encrypted salaries in the database, they only experience around 1.7× performance slowdown compared to not running in SGX with PHE. Execution time grows linearly with the number of summed entries, as expected. They present a potential electronic voting system with the combined model that ensures the anonymity of voters, at most once voting, the integrity of voting results, prevents vote duplication and can ensure the legitimacy of inputs. Additionally, they present other example usages for the combined model, with a similar ability (in terms of the complexity and scale of the system).

SAFETY [57] is a system which combines PHE and SGX to securely process genome data to identify genetic risk factors for diseases. This data is quite sensitive and often comes with strict regulations on how to process and store it. By combining Paillier encryption with SGX they created a system which achieved

a 4.8× speedup compared to existing secure computing techniques.

TEEFHE [10] is an example of combining FHE with SGX by performing the bootstrapping step within SGX. They use the BGV [24] scheme implemented in SEAL and modify the library to run within SGX. They distribute the work into several nodes, where some nodes process the ciphertexts using homomorphic operations in untrusted environments, and when they require the bootstrapping procedure, the processing nodes transmit them to a node which has the SEAL library running within SGX. The enclaves in these nodes have the secret key to decrypt and encrypt ciphertexts, so they first decrypt the incoming ciphertexts and re-encrypt them on the way out, and they are sent to the processing nodes to be further processed, that then can be transmitted back to the user when computations are complete. SGX enclaves perform encryption and decryption, preserving data and code integrity and confidentiality, as they do not consider side-channel attacks. Decrypting and encrypting a ciphertext removes the encoded noise and *refreshes* the ciphertext, effectively doing the same as a bootstrapping operation, but at a lower cost. As the untrusted compute servers perform computations on the encrypted data, they do not preserve data integrity in the case of an attack.

2.4 Summary

This chapter looks into the foundations of HE and more specifically, how to achieve FHE. It mentions the advantages of the technique, and the integrity weaknesses it entails that is malleability. Further, we explain the concept of a TEE, mention Arm's TrustZone and detail on Intel's SGX. SGX does have some weaknesses regarding data confidentiality as multiple attacks have shown. We mention how using a TEE such as SGX could cover the integrity weakness of FHE, which simultaneously provides stronger confidentiality guarantees to SGX. As TrustZone aims for embedded systems which frequently concentrate for the field of Internet of Things (IOT), microcontrollers or other embedded systems, this thesis focuses on Intel SGX.

3 Design

This chapter gives insight into some of our design choices. Section 3.1 lays out the various Fully Homomorphic Encryption (FHE) libraries that are available, their use-cases, and which we chose. Section 3.2 on the following page argues for our reasoning behind choosing to port the library, and Section 3.3 on page 28 follows by underpinning reasons for choosing Rust as a programming language. Finally, Section 3.4 on page 34 summarizes this chapter.

3.1 Homomorphic Encryption Library

There are several implementations of FHE schemes in libraries, most notably Simple Encrypted Arithmetic Library (SEAL) [58], HElib [59], Homomorphic Encryption Arithmetic of Approximate Numbers (HEAAN) [23] and TFHE [16]. The first two implement the Brakerski-Gentry-Vaikuntanathan (BGV) schemes and HEAAN [23] scheme. HEAAN implements the latter scheme, and TFHE implements a variation of the Gentry-Sahai-Waters (GSW) [60] scheme. HEAAN aims for and is most appropriate for approximate number calculations, similar to floating-point computations or numerical computations. The BGV scheme allows creating circuits and encoding programs with high performance, but at the same time, requires a substantial amount of domain-knowledge of Homomorphic Encryption (HE) and the scheme itself. Misusing it leads to large performance penalties as their library documentation also emphasizes. TFHE aims to be a simple library and has a very simple Application Programming

Interface (API), while still providing very fast bootstrapping times. Because of this, this work focuses on the TFHE scheme and library as it is flexible for prototyping without extensive research into the BGV scheme.

The TFHE library is written in a combination of C and C++. It has some external dependencies for a Fast Fourier Transform (FFT) implementation. As it dynamically loads libraries at runtime, it is unsuitable for use in Software Guard Extensions (SGX). Dynamically loading libraries is the responsibility of the Operating System (OS) and is thus not suitable in SGX, although work such as DynSGX [61] achieves loading functions dynamically. Thus, it is appropriate to rework the library to ensure it is statically linked, to allow running it within SGX without requiring contact with anything outside the enclave. However, due to the unsafe nature of C and C++, as we will further show, this allows a large class of software vulnerabilities to exist. C and C++ are languages without memory safety, meaning they allow the execution of several operations that cause dangerous Undefined Behavior (UB). Memory-safety disallows several security risks such as null-pointer dereferencing, leading to an adversary potentially hijacking an enclave, data leakage, or other *out-of-enclave bugs* [9]. Furthermore, it allows memory corruption errors, buffer overflows, accessing uninitialized memory, dereferencing pointers causing access violations (segmentation faults), data races and more. Memory safety in SGX is important as UB might lead to a breach of enclave security guarantees, and is why research into using Rust (a memory-safe language) in SGX is a studied area [62]. Thus, to achieve a higher degree of security, reworking the TFHE library to link dependencies for use in SGX statically is insufficient and still allows the user to write code that performs operations that are not memory safe and might cause security violations. We wish to prevent as much insecure behavior in SGX as possible.

3.2 Porting the TFHE Library for Homomorphic Encryption

The original TFHE library is written in C++ with C header files, so it seems to be easy enough to use in SGX. Unfortunately, it is not. The library has several dynamically linked libraries, which is problematic in SGX, as one needs to trust the OS since the kernel is responsible for loading and linking dynamic libraries.

Another reason is that the existing Software Development Kit (SDK) for SGX is difficult to use without extensive knowledge of how SGX works. As SGX is disentangled from any OS it requires the code to be non-dependent on OS-specific features. Additionally, SGX enclaves have a limited Protected RAM (PRAM), so

the code should have a small or non-existent runtime. Specifically, as programming languages with runtimes often require an underlying OS to function, any language with a runtime is practically out of the question. The programming language should probably not be garbage-collected as these languages often have a significant memory overhead, and paging is an expensive operation in SGX as pages need to be encrypted. Even though work such as Se-Lambda [63] use a garbage-collected language, specifically JavaScript, within SGX and with impressing efficiency, we argue that our language should not be garbage-collected. The reason is that FHE is very performance and resource-heavy, as an example can a single binary gate perform in around 13 *ms* in the TFHE library using hand-optimized assembly code [64], a relatively high number considering the effort. Using any language that applies costly abstractions or gives a large runtime overhead is infeasible.

C and C++ do not have memory safety, meaning it is up to the programmer to not make mistakes. Memory safety issues such as dangling pointers, double free, use after free, accessing uninitialized memory and dereferencing null pointers are common mistakes in C and C++. These issues are dangerous in SGX as it is easy to mistake enclave-pointers to non-enclave memory, potentially leading to a breach of confidentiality. Garbage-collected languages are often memory-safe, as they allocate memory for the user and track references to allocated memory, they will eventually deallocate unused memory, and ensures that code will not dereference unallocated memory, though in some languages it may be uninitialized. Garbage-collectors do, however, often have a significant memory-overhead, often requires multithreading to intercept threads and perform collection. They also use techniques of collection, leading to unpredictable performance spikes. In some cases this can lead to garbage-collection pauses of up to minutes or seconds, depending on the collection strategy and memory usage of a program, which is not at all negligible in situations where responsiveness or performance is critical [65].

Additionally, C is not developer-friendly with regards to adding higher-level abstractions. Writing in C often requires one to write performant, but slightly unreadable code, or more readable code with abstractions and several levels of indirection while sacrificing performance, though with more possibilities for unsafe code. It also skews developers to develop leaky abstractions, in some cases. C++ is better in this regard, allowing performant abstractions. However, since we want to explore a memory-safe language, they will not suffice.

3.3 Choosing a Programming Language

Rust[1] [66] is a strongly and statically typed systems programming language designed with a focus of C interoperability. A core principle of Rust is memory safety without garbage collection. Other memory-safe languages handle memory safety through garbage collection but impose a large runtime overhead. Rust has (close to, similar to C's runtime) no runtime overhead and manages memory through a borrow-checking- and lifetime-system. A value that can be de-allocated cannot be used after de-allocation, also known as an affine type (sometimes erroneously called a linear type), a type that can be used at most once. Each reference to some variable in Rust has an owner and variables are immutable by default. To be able to mutate a variable, it must be explicitly marked as mutable. References can be shared or exclusive. The borrow checker in Rust ensures that there can be only one owner of a variable at any time, any number of shared references at a time without any exclusive references, or only one exclusive reference without any shared reference at any point in a program. By doing this, the borrow checker ensures that memory is not read while being mutated, as well as ensuring only one thread can mutate while no one can read. Coincidentally, these properties ensure that the borrow-checker and type system of Rust fulfills what is known as Bernstein's Conditions [67]. These conditions state that if u and v are some operations on some memory regions, $M(v)$ is the set of memory regions mutated by v and $R(v)$ is the set of memory regions being read by v, we have the following equation:

$$M(v) \cap M(u) = R(v) \cap M(v) = R(u) \cap M(u) = \emptyset \tag{3.1}$$

If Equation 3.1 holds, then the operations v and u are safe to parallelize or run concurrently, with "safe" in this context meaning without modifying program results. Rust's borrow checker enforces this equation, leading to the next core principle of Rust: *fearless concurrency*. Because the borrow checker ensures the equation holds, any Rust program using the safe part of it (unsafe Rust is opt-in) is free of data-races such as Read-After-Write (RAW), Write-After-Read (WAR) or Write-After-Write (WAW), something the RustBelt project formally proved [68].

The last core principle of Rust is "abstraction without overhead", also known as zero-cost abstractions. This principle takes from C++ and essentially means that one should not pay for features one do not use and what one do use one could not hand-code any better oneself. Rust's traits are an example of this. Traits are equivalent to interfaces in many languages, but Rust's monomorphization strategy generates a single implementation for each type implementing a

1. https://www.rust-lang.org/

trait, permanently erasing the abstraction, leading to just as fast code as if the abstraction was not there.

A simple example shows the benefit of this. Consider a function checking if a given value is in an array of values, returning a boolean value. In C-style Rust this could be written as:

```
1 fn number_in(numbers: &[i32], value: i32) -> bool {
2   for i in 0..numbers.len() {
3     if numbers[i] == value {
4       return true;
5     }
6   }
7   return false;
8 }
```

Line 1 in the example is the function signature, with `fn` declaring a function, `numbers_in` the function name, `numbers: &[i32]` being the first argument, a value with the type slice (pointer to part of an array) of 32-bit integers, `value: i32` the argument to look for, and `-> bool` specifying the return value. The rest of the function body should be relatively easy to understand and is quite similar to C. In idiomatic Rust, the code snippet can be written as:

```
1 fn number_in(numbers: &[i32], value: i32) -> bool {
2   numbers.iter().any(|x| *x == value)
3 }
```

The signature is the same, but the body creates an iterator over the slice, uses the `any`-method which takes a closure applied to every element in the iterator, returning a boolean when the first closure returns true, otherwise false. In Rust, the last expression of a block is returned, so the `return` keyword can be omitted. This example is more concise, shorter and faster. The abstractions compile away, and a simple microbenchmark on the author's laptop showed that for a million integers where the number searched for is at the end of the array, the first example completes in 96 *ms* while the last example takes 64 *ms*. While not being a thorough benchmark, it shows how abstractions do not impose a high cost in Rust. The second version has a dereferencing operation in the closure (`*x`), while the first does not. This is because the first example is automatically dereferenced in the for-loop, while it is not in

the closure as `.iter()` returns an iterator that returns shared references to the values. This example is slightly contrived, as the same is achievable with `numbers.contains(&value)` which is implemented identically to our example[2], rendering the function unnecessary.

It has been shown that memory safety issues cause a large percentage of security bugs and vulnerabilities, up to 70% in Microsoft's products [8] and around 70% in the Chromium projects [69], so any attempt to mitigate these by ensuring memory safety is useful. Rust mitigates almost all of these as it is memory-safe by design, which the RustBelt project [68] formally proved for a subset of the Rust standard library. Thus, the fact that Rust provides these guarantees is vital, considering the broad range of security vulnerabilities it eliminates.

Rust libraries are known as *crates*. A crate consists of several modules, where each module contains the types, traits, functions, and submodules it might need. A crate that is published as a library is called a package, where the package contains the crate. We use the terms package, crate and library interchangeably in this thesis. Rust statically links all *crates*. Except for the C library implementation, which can be statically linked if one uses musl[3], all crates are statically linked, compiling down to a single, portable binary. Statically linking libraries allows for easy use in a multitude of platforms, including SGX.

3.3.1 Undefined Behavior

In C and C++, there are many cases where common code is UB. UB means that it is up to the compiler to handle the case, and various compilers make different decisions. Having UB in the code implies that code can execute quite differently than the developer expects, such as optimizing parts of the code away, a computation being optimized into a single constant value, or change execution flow [70]. Much of common code qualifies as UB, as there are lots of operations defined as undefined behavior. Examples of these are integer overflow and underflow, out of bounds accesses, dereferencing `NULL` pointers, dereferencing pointers to allocated memory of size zero, shifting values by amounts that are greater than or equal to the number of bits in the value, modifying a string literal, using a variable before assigning it, preprocessor numeric values which cannot be represented by `long int` and more. As cases of UB are so vast, it means that simple code examples like the ones below cause

2. https://doc.rust-lang.org/stable/src/core/slice/mod.rs.html#1399-1404
3. https://musl.libc.org/

UB:

```
// Assume a and i are defined
a[i] = i++; // UB

int64_t i = 1;
i <<= 65; // UB

int j;
printf("%d", j); // UB
```

Furthermore, the order in which function parameters evaluate is unspecified behavior, with the only requirement being that all parameters must evaluate completely before calling the function. The nonspecificity in evaluation order means that a simple example like this:

```
some_function(function1(), function2());
```

Can be evaluated as either of the following two examples:

```
auto first_value = function1();
auto second_value = function2();
some_function(first_value, second_value);

// OR

auto second_value = function2();
auto first_value = function1();
some_function(first_value, second_value);
```

Reordering of function calls means that if the two functions have side-effects, the program may produce different results based on what the compiler does. Producing differing results also holds with all the cases of UB as well, where some functions may return early with a hard-coded value, loops may be optimized away, and many other strange, unexpected things may happen. These are unintuitive, and the programmer may not discover these as most do not inspect the assembly or machine code after compilation.

In Rust, several of the behaviors that are UB in C and C++ is well-defined to ensure that code is as safe as possible. Incidents such as integer overflow are defined behavior, where signed integers overflow to their minimum (negative) value in release mode, but panic in debug mode, so that the developer may notice behavior that may not be intentional. It wraps in release mode as bound checks may become prohibitively expensive, and normal Rust code optimizes for speed. For explicit wrapping behavior the numeric types have explicit methods for all normal integer operations such as `2.wrapping_add(3)` for addition, `2.wrapping_sub(3)` for subtraction, etc. Additionally, Rust provides explicit methods for different behavior such as saturating operations (saturating the value at the numeric bounds), checked operations (for explicit handling of the overflow case), and overflowing operations, which returns the value and a boolean value determining if the value overflowed. Additionally, if all integer operations require the wrapping behavior, Rust provides wrapper types `Wrapping<T>` [4] for generic types `T` where all numeric binary operations wrap.

The original TFHE library [16] has a lot of integer manipulation, such as exploiting integer over- and underflow, expecting the integer to wrap around. This is UB in C and C++ but is quite commonly implemented as wrapping overflow by compilers. Converting this behavior to Rust is simple enough, as one could either use `Wrapping<i32>` for the `int32_t` type or use the explicit wrapping methods. Using explicit wrapping methods in Rust allows being explicit in integer overflow behavior while it also stops depending on UB for correctness. Furthermore, the original library uses a lot of typecasts between numeric types. In C, casting a floating-point value to an integer will result in truncating the decimal part, and the integral part represents the integer. However, if the integral part does not fit, the operation is UB. Potentially, a cast from a floating-point value to an integer may result in UB, which again, may produce varying results on different platforms and compilers.

3.3.2 Disambiguating Types and Self-Referential Structures

Rust is a language which heavily encourages the use of types and has a strong, static type system. C on the other hand, does not. C++ has classes, but they are not really used in the original TFHE library. This includes types such as standard library collections, such as vectors for dynamically-sized arrays. Due to this, it is difficult to disambiguate between pointers to values and actual arrays and one needs inspect use-cases and allocation-sites to figure out the type it should be converted to in Rust. Or even, double pointers may be interpreted as an optional pointer to an array, leading to more confusion.

4. https://doc.rust-lang.org/stable/core/num/struct.Wrapping.html

Furthermore, the original library uses structures where some fields point to specific parts of another fields in the same structure. These are called self-referential structures. Self-referential structures are not a problem in a language like C and C++ as they manage the memory manually, but Rust uses lifetimes to declare how long a value should live and self-referential structures cause problems for the borrow-checker and are quite difficult to model in Rust. Self-referential structures are possible to model in Rust using the `Pin<P>` API[5], designed for memory that cannot be moved in memory, although this is difficult to use. Moreover, one can utilize the `unsafe` keyword to escape from some memory-safety restrictions and allow more functionality, such as calling unsafe functions, performing Foreign Function Interface (FFI) calls, and dereferencing raw pointers. However, it is not necessary in this case and will sometimes allow cases of UB.

Most of these types in the original library are there to model two-dimensional vectors as a one-dimensional vector in row-major order, but having the second field accessing the next row within the flattened array. They are strictly not needed and are there mostly for convenience. In the Rust port, these two-dimensional vectors are modeled as such: two-dimensional vectors. That is vectors within vectors, `Vec<Vec<T>>`. Additionally, three-dimensional vectors flattened in the original library are three-level nested vectors in Rust, mostly done due to wishing for correctness before performance. However, it might give a substantial performance dip due to cache locality, where iterating through values of an inner vector will not update the cache with values from the next vector as they are spatially distant. Additionally, it might reduce performance due to creating more allocations, one for each vector, in comparison to a flattened, contiguosly allocated array. However, the port was designed for a proof-of-concept, not directly performance, as one can extrapolate some performance characteristics from the original library's performance.

3.3.3 Choices and Alternatives

To summarize, Rust is a programming language that is memory-safe, has strong type safety, is data race-free, and it is statically ahead-of-time compiled with a minimal runtime (similar to C's runtime, mostly used for handling program panic and abort) and no garbage-collection. Furthermore, it builds on abstractions that are zero-cost and encourages the use of these. It has a low memory footprint, akin to C++ and C, and targets bare-metal, meaning it can be used for low-level programming such as device drivers, OS kernels or programming for embedded systems. At the same time it provides performance that is on par with C and C++.

5. https://doc.rust-lang.org/std/pin/index.html

A possible alternative to the library is keeping it in the language it is (a mix of C and C++), but reworking it to work within SGX. This is possible, but Rust enables safe parallelization, without data races, something C and C++ does not. Additionally it provides better user ergonomics while keeping a strong type safety. To conclude, writing it in Rust seems like the best alternative.

3.4 Summary

This chapter discussed the properties of a few FHE libraries and outlined our argumentation for using the TFHE scheme. Further, it described our reasoning behind porting the library to another language, to provide better memory safety. The chapter further arguments for choosing Rust as a programming language, by describing several cases where conventional code in C and C++ is actually UB, and how Rust avoids several of these potential errors.

4

Implementation

This chapter describes the implementation of the Fully Homomorphic Encryption (FHE) library port which we will use in conjunction with Intel Software Guard Extensions (SGX) to evaluate our system. Section 4.1 describes choices made to allow certain concepts to translate into Rust. Further, Section 4.2 on page 37 outlines the external dependencies used in the implementation and why they are needed. Section 4.3 on page 40 describes the general layout and module structure of the code. Finally, Section 4.4 on page 43 details how Rust and the library implementation is integrated into SGX. We then outline our implementation of the Partially Homomorphic Encryption (PHE) Paillier cryptosystem, and finally summarize this chapter in Section 4.6 on page 44.

4.1 Porting TFHE to Rust

The implementation process of porting the TFHE library[1] from C++ to Rust began with describing data types. In the original library, many structures had fields that were strictly pointers to another struct type. In C and C++, this is indistinguishable from an array pointer, unless one looks at the initialization site. Dynamically allocated arrays such as these are equivalent in function-

1. Our port builds entirely on the code at this commit `https://github.com/tfhe/tfhe/commit/76db530cf736a25115ea0b0ccdb9267b401bb9a7`

ality to the `std::vec::Vec` [2] type, and are unambiguous in contrast to the original library's implementation. Structures with pointer-fields in the original library do not specify if they *own* the data they reference or if it references memory given to it in initialization, e.g., `struct Data { val: Vec<i32>}` versus `struct Data { val: &mut [i32]}` (lifetime annotations elided for brevity). This distinction is necessary for Rust, as it tracks ownership, but not C++. In the implementation, we chose the former, as it is more manageable than the latter, and it seems the original library chose this solution as well, based on their usage. Integer and floating-point data types have direct equivalents in Rust, and are thus translated directly.

The original library source code has some structures where a field is a pointer to values within a dynamically-allocated array that a different field in the same structure references. Referencing a field within the same structure is known as a self-referential structure. These are disallowed by the type system in Rust, and are challenging to attain without more advanced lifetime and type annotations. They are inherently dangerous and unsafe when it comes to memory safety due to Rust's move-semantics when taking ownership. When one moves a value in memory, the referenced value in the self-referential structure is invalidated, hence the unsafety. Because of this, the implementation chose to remove these fields and access these values directly, at the loss of some readability.

The TFHE library also has some occurrences of void pointers meant to be specialized by a Fast Fourier Transform (FFT) implementation. These are in the original library somewhat equivalent to Rust's trait system as they allow multiple implementations while providing a stable interface. In this implementation, we do not use the trait system for this as we do not aim to allow multiple implementations of the FFT.

The original library provides specific functions for serializing and deserializing data, including one set of functions for reading or writing a file, and another for streams. This set of functions is quite limiting and does not allow the developer to specify the serialization format. In our implementation, all data structures that might need to transmitted somewhere are serializable and deserializable, using the Rust package Serde[3]. Serde designs serialization and deserialization so that any data structure that implements one of two traits can be serialized or deserialized to one of the tens of different serialization formats supported. A macro allows deriving the implementation automatically, such as (line 3 highlights derive macro):

2. `https://doc.rust-lang.org/stable/std/vec/struct.Vec.html`
3. `https://crates.io/crates/serde` or their homepage `https://serde.rs/`

```rust
use serde::{Deserialize, Serialize};

#[derive(Deserialize, Serialize)]
struct Ciphertext {
  data: Vec<i32>,
}
```

Implementing these traits allows the user of the library implementation to choose the serialization format that fits the use-case best. As ciphertexts are quite large and contain many integers, a binary format might be best suitable.

Our implementation is written entirely in the safe subset of Rust, and will not compile if the `unsafe` keyword is used in our codebase. This is enforced by a crate diagnostics attribute, `forbid(unsafe_code)`, which also prevents overriding the attribute in our crate. However, some of our external dependencies require the use of the unsafe part of Rust to interact with low-level operations, such as providing randomness through assembly instructions.

4.2 External Libraries

When implementing our FHE library specifically to allow use in SGX, it is crucial to ensure that we use no dependencies that are incompatible with running in an enclave, such as requiring Input/Output (IO) access or use libraries that assume properties of the underlying hardware. Luckily, the cryptographic system does not require any specific software or hardware that SGX does not support. As the TFHE scheme is a semantically secure cryptosystem, it requires a source of randomness to function. However, this only applies in the encryption phase to generate secure ciphertexts. The evaluation and bootstrapping phase is entirely deterministic and computational and can thus function within SGX. Either way, the instructions `RDRAND` and `RDSEED` are available in SGX as a source of randomness if needed. Rust does not provide random number generation as part of its standard library, so the package Rand[4] is used instead, and it works within SGX if it should be required for some reason.

Homomorphic ciphertexts in the TFHE scheme rely on a polynomial representation of integers, and the implementation uses mostly integer arithmetic to perform the homomorphic operations between ciphertexts. The addition and

4. `https://crates.io/crates/rand`

subtraction of polynomials are simple and require only pairwise addition or subtraction of elements. Polynomial multiplication, on the other hand, requires a more complex implementation. A naïve implementation of the multiplication of two polynomials is possible in $O(n^2)$ operations. However, as these operations execute in a tight loop, they become a bottleneck. The complexity of the multiplication can be improved upon by utilizing FFT. The multiplication step of two polynomials $P(x)$ and $Q(x)$ can be divided into three steps:

- Take the coefficient representation of each of the polynomials and represent it as a vector of coefficients, with increasing order (e.g. $P(x) = 1 + 2x + 3x^2 \Rightarrow \vec{p} = [1, 2, 3]$), and perform a forward FFT transform on both these vectors. Each of these operations are $O(n \log n)$.

- Multiply the results' coefficients pairwise so that $\vec{p} = [1, 2, 3], \vec{q} = [4, 5, 6] \Rightarrow \vec{r} = [1 \cdot 4, 2 \cdot 5, 3 \cdot 6]$. This operation is $O(n)$.

- Perform the inverse FFT transform on $\vec{r}$ to get the coefficients of the polynomial representing the product of $P(x)$ and $Q(x)$. This operation is $O(n \log n)$.

The resulting time complexity is then $O(n \log n) + O(n) + O(n \log n)$ or simply $O(n \log n)$. Changing from the naïve implementation to a FFT-based multiplication procedure improves performance greatly on large polynomials. Achieving secure encryption using the TFHE scheme requires these polynomials to be on the order of $2^{10} = 1024$, so the improvement is substantial, with a relatively low increase in code complexity. Changing the implementation to use FFT-based multiplication gained an overall performance increase in our library, something which we will discuss in-depth in the coming chapter. This implementation uses the RustFFT[5] package for the FFT implementation.

Most operations in the TFHE library operates on 32-bit signed integers, to exploit the fact that wrapping behavior is the default on integer over- and underflow (although it is technically Undefined Behavior (UB) in C/C++). This port uses the Rust equivalent of `i32` and explicit wrapping-methods where an overflow could happen, instead of the `Wrapping<T>`-type which requires explicit packing and unpacking of values, and loses ergonomic value. A few parts of our implementation have generic functions to allow differentiating between polynomials of integers and polynomials of integers that represent numbers in the torus space, $\mathbb{T}$, even though their implementation is the same. These polynomial implementations also exploit Rust's trait system and macros for code generation used to implement the polynomials. However, when working with generic types that are numeric, Rust's standard library does not provide

5. `https://crates.io/crates/rustfft`

traits for numeric types, so we use an external package Num[6] for its numeric traits to mark generic type bounds in generic functions. Additionally, for FFT implementations, which operate on complex numbers, we use Num's complex number type. Complex numbers are also not in the Rust standard library.

Most of the operations performed in the library implementation are simple operations performed on sequences of integers (in our implementation, dynamically-sized arrays, or the Rust `Vec<i32>` type). Other than the FFT-based polynomial multiplications, most of the other operations are of simple pairwise arithmetic. The Rust compiler builds on LLVM[7] and allows the use of many of the LLVM optimizations. One of these optimizations is auto-vectorization, where the compiler inserts Single Instruction, Multiple Data (SIMD) instructions in place of code that apply simple arithmetic operations to elements of arrays. It applies optimizations if the arrays are of a compatible length, that is, often a multiple of 4. It can replace scalar operations with vector operations, often achieving great performance benefits if the hardware supports it. If the processor supports instruction set vector extensions such as AVX-512, as much as 16 ordinary instructions for 32-bit integers could be replaced with one instruction that operates on 512 bits, or 16 32-bit integers at once — achieving a speedup of potentially 16×.

Most of the arrays used in our implementation are 1024 elements in length, depending on the parameters used, but we determine the length and allocate these at runtime. Because of this, the Rust compiler does not apply auto-vectorization optimizations to our codebase. Relying on the compiler to auto-vectorize code for performance optimizations is flaky, so our code is better suited with a manual implementation of SIMD usage where appropriate. However, polynomial multiplication takes most of our runtime, and the RustFFT package performs it so the benefits might vary. Additionally, the time cost of implementing SIMD instructions is extensive, which is why we have not approached this optimization.

4.2.1 Development Tools

The library implementation uses the Criterion[8] package as a developer dependency, only used for benchmarking purposes. It is a statistic-driven benchmarking library that generates reports on performance changes from a previous run. The benchmarks performed are microbenchmarks, meaning they primarily test speeds for small portions of the library's functionality. We use these benchmarks

6. https://crates.io/crates/num
7. https://llvm.org/
8. https://crates.io/crates/criterion

throughout development to assert that small changes do not negatively affect performance, as a form of performance regression testing.

During development, Cargo Flamegraph[9], a tool to generate flamegraphs [71] based on performance metrics from the Linux tool Perf[10] was used. These graphs are used to analyze which parts of the library that spent the most time during execution, to find parts to optimize. Flamegraphs allows one to easily determine which functions take up the most time, by showing the cumulative proportion of time spent in each function, and their nested calls. The generated graphs allowed us to determine that polynomial multiplication alone took up an overwhelming majority of the time (70%) in both an example that encrypts and then decrypts data and in an example where we apply a homomorphic gate to a ciphertext, and that if we were to optimize, this would be the place to start. The results of these improvements are detailed in the evaluation chapter.

4.3 Code Makeup

The original TFHE library has 4400 Source Lines of Code (SLOC)[11] of mostly C++, but also some C and header files. These numbers do not include the FFT implementation. Additionally, it consists of 5743 SLOC of mostly C++ code for testing, but also some C code to test the C bindings of the library. Furthermore, it uses preprocessor directives for code generation, and these numbers do not include the expanded code. Our port of the library ends up on 2267 SLOC of Rust, including unit tests and documentation test examples, but integration tests add another 164 SLOC of Rust, although it has code generation, so integration tests expand to around 316 lines. Additional microbenchmark code and examples add another 100 SLOC, ending up at a total sum just below 2700 SLOC of Rust.

A lot of test code in the original library tested things that are unnecessary in Rust, such as ensuring an allocated array is not `NULL`, and that values are zeroized. Because the code has the style of mutating input arguments (caller allocation-style) instead of allocating a new result to return, it tests that certain specific input arguments are not mutated. Additionally, it has some tests which test implementation details, which is something one should not do, so we omit these in our port, thus contributing to our lower SLOC count.

Our implementation has more than 35 unit and integration tests, several

9. `https://crates.io/crates/flamegraph`
10. `https://github.com/torvalds/linux/tree/master/tools/perf`
11. All SLOC counts are counted using Tokei `https://github.com/XAMPPRocky/tokei`

benchmark programs and a few examples to show a user how to use the library. These tests acted as regression tests when we contiguously transformed our code into more idiomatic Rust. We used the benchmarks as performance regression tests to ensure our transformations did not hurt performance. The documentation tests ensure that the code's documentation conforms to the Application Programming Interface (API) in case we change any function or method signatures. Tests, documentation tests, code style (formatting) and lint checks, and benchmarks have been running in a continuous integration system on GitHub[12], where we have stored our project during implementation.

4.3.1 Module Structure

The port of the original library is structured somewhat differently from the original. As the original library implementation writes C-style C++, it structures files accordingly. It does this with a high degree of separation (declaration, allocation, initialization, and functions that operate on these types are separated) to not clutter namespaces as a header-file import includes all declarations. It could have used C++'s namespace system but does not, for the most part. Rust's module system allows importing specific items from modules instead of requiring a glob import, so we place data structure declarations next to their implementation and functions that operate on them.

`lib` The root module of the library. It is mostly empty except for crate documentation and re-exports encryption, bootstrapping, and homomorphic gate functions. It also re-exports each of the following modules: `bootstrapping`, `circuits`, `encryption`, `gates`, and `numerics`.

`encryption` Publicly exposes functions for encrypting and decrypting ciphertexts. It also exposes functions and types for generating encryption parameters and encryption keys with different levels of security. Additionally, it exposes the ciphertext type.

`gates` Publicly exposes functions that represent the basic homomorphic logical gates. Exposes all the logical gates that the original library exposes. That is, the normal logic gates `not`, `and`, `xnor`, `xor` `nand`, `or`, `nor`, and `mux`. Additionally, it exposes some specialized gates where one of the inputs is negated, `andny`, `andyn`, `orny`, and `oryn`. It also exposes the `constant`-function for bootstrapping a ciphertext with a publicly known constant.

`bootstrapping` Publicly exposes functions for performing the bootstrapping operation if the user requires control over when to perform this operation.

12. `https://github.com/`

`circuits` Publicly exposes more complex circuits such as a comparison circuit for comparing encrypted sequences of bits. This is a new module that does not have an equivalent in the original library. It exposes more complex circuits such as an equality circuit (`eq`) for two n-bit numbers, a comparison circuit (`compare`) for determining if $\boldsymbol{a} \leq \boldsymbol{b}$ is true (where $\boldsymbol{a}$ and $\boldsymbol{b}$ are n bits), a half-adder, a full-adder and a circuit for adding two n-bit numbers.

`numerics` Publicly exposes functions for encoding and decoding values into a torus-representation for manual use of the library's encryption functions.

`polynomial` Contains types and implementations for ergonomically working with integer polynomials. For internal use.

`lwe` Contains types and methods for working with the ciphertext representation. For internal use.

`tlwe` Contains types and methods for working with the torus version of the Learning With Errors (LWE) ciphertexts used within the TFHE scheme. For internal use.

`tgsw` Contains types and methods for working with the torus version of the Gentry-Sahai-Waters (GSW) ciphertexts used within the TFHE scheme. For internal use.

The implementation supports creating keys of different security levels. Choosing parameters for encryption schemes based on LWE is complicated, as choosing a parameter set with incompatible values might lead to an insecure system or a secure but less performant one. It is not as simple as measuring the security level of commonly used symmetric encryption systems such as AES, as it depends on different mathematical security assumptions. Our implementation currently supports the two parameter-sets defined in the original library, which have estimated security levels of 80-bits and 128-bits, known as bit security [72]. However, the key size is not directly proportional to the security level, as in AES, where a security level of 128-bit security equates to a 128-bit binary key. In TFHE, a security level of 128-bit equates to a $\sim$ 24 *MB* bootstrapping key [11]. The default parameter set in our library is the 128-bit security version as cryptographers do at the time of writing recommend 128-bit security to be safe until theoretically the year 2090 [72], but proper interpretations indicate safety for a few more decades, whereas 80-bit security was deemed safe until the year 2018 based on the same equations. These are based on encryption systems where a cryptographic *break* is not known, and a brute-force attack is the best known attack.

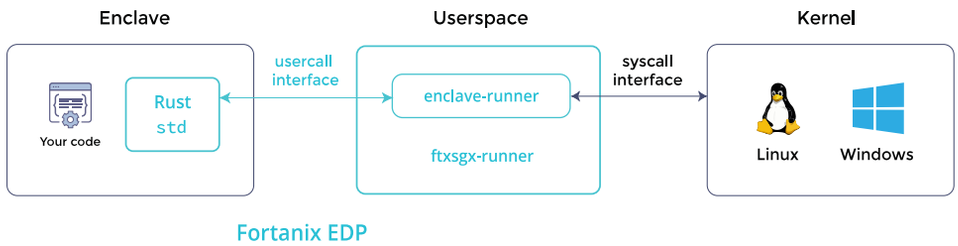


Figure 4.1: Fortanix's Enclave Development Platform architecture overview. Source: `https://edp.fortanix.com/docs/concepts/architecture/`

4.4 Executing Code in SGX

Executing programs in an SGX enclave is not straightforward. Programs need to be signed; enclaves require specifying stack size, heap size, and the number of threads beforehand. These requirements arise from the complexity of ensuring the security of enclaves and its design in hardware and software. SGX allows an enclave to exceed the memory limits of Protected RAM (PRAM), currently at 128 *MB* as of version 1 of SGX, by specifying the heap size before entering an enclave. However, an enclave with a large heap size will require the memory manager to page out encrypted data to regular Random-Access Memory (RAM) more often, resulting in a performance loss. Thus, one should investigate the memory requirements of a program before executing it within SGX, to limit memory specifications to just the needed memory.

Writing software to run within SGX is a difficult task and requires the developer to have some knowledge of how to use it properly. Additionally, Intel develops the SGX Software Development Kit (SDK) for use with C and C++, which requires us to deal with Foreign Function Interface (FFI) in Rust if we were to use it directly. Helpfully, there are projects in Rust to work with Intel SGX. Most notably, Rust SGX SDK[13], initially developed by Baidu, and Fortanix's Rust EDP[14]. The former is at the time of writing the most popular project in terms of *stars* on GitHub, and has published a paper on it [62], but aims for more advanced use-cases of SGX, where developers need a large degree of control of the SGX SDK. However, the latter aims for an easy way to write programs for SGX by being a platform compilation target, while the former is a library.

Additionally, Fortanix's project is recognized by Rust as a supported target platform and has at the time of writing an official tier 2 status[15]. Tier 2 support means code is guaranteed to build on the platform and is part of the language

13. `https://github.com/apache/incubator-teaclave-sgx-sdk`
14. `https://github.com/fortanix/rust-sgx` or their homepage `https://edp.fortanix.com/`
15. `https://forge.rust-lang.org/release/platform-support.html`

continuous build testing system. As such, regular Rust programs that do not use multiple processes or rely on Operating System (OS) functionality should work out of the box. These guarantees allow us to easily integrate our FHE library into a program that runs within an SGX enclave and is why we chose using Fortanix Rust EDP for working with SGX.

The architecture of Fortanix's Rust EDP is seen in Figure 4.1 on the previous page. As seen, it wraps the Rust standard library and intercepts user-calls through its enclave runner, making the experience for developers as simple as writing a normal Rust program.

Our example program using our ported library implementation and the Fortanix Rust EDP requires no special handling other than specifying the stack and heap size required for the program. The lack of special handling means that users of our ported library can easily use the hybrid solution of FHE and SGX in the cloud.

4.5 Paillier PHE Implementation

Most earlier work focuses on using PHE instead of FHE within a Trusted Execution Environment (TEE) such as SGX. They also commonly use the Paillier encryption scheme [18], a PHE scheme. This scheme allows the homomorphic functionality of the addition of ciphertexts. In contrast to most other PHE schemes, however, the Paillier cryptosystem also allows multiplication between ciphertexts and plaintexts. Having this feature makes it slightly more useful, however as it only allows multiplication with a publicly known plaintext, it is rather limited. We wished to see the efficiency of such a solution and its possible use-cases. Thus, we implemented a version of the Paillier PHE scheme in Rust. It functions in SGX using the Fortanix Rust EDP, similarly to our TFHE implementation. We did not take this any further as PHE proved to be quite limiting.

The implementation consists of 283 SLOC of Rust, while the tests are at just over 60 SLOC. The tests use property-based testing to generate values for testing a larger amount of the input space to find mistakes.

4.6 Summary

In this chapter, we described the process of translating the original library to Rust. We describe how we solved the challenges of translating parts of the

original library that relied on UB in their implementation, such as integer overflow behavior. Additionally, we described how we cleaned up some parts of our implementation to avoid self-referential structures that existed in the original library implementation. We described which external libraries we use and why, as we want to minimize the Trusted Computing Base (TCB) of our implementation and reduce memory consumption in SGX. We also give insight to an implementation optimization detail that improve performance, but increases code complexity. We mention the optimizations that we have not done but that we are aware of, such as using the Lagrange half-complex representation internally for polynomials, using SIMD instructions, and multithreading for dividing work onto several threads. These optimizations are out-of-scope for this thesis work, in terms of the development time needed to implement them.

Furthermore, we describe our implementation in terms of module structure, lines of code, tests, benchmarks, and examples of how to use our library. We also describe our additional module providing easy access to premade circuits, which the original library does not provide. Later, we describe the use of and choice of Fortanix Rust EDP to allow the use of our library within an SGX enclave. Lastly, we mention how we initially implemented the Paillier cryptosystem to be compatible with SGX, but have not further built on this idea as using PHE is limiting.

5 Evaluation

Section 5.1 describes our experimental hardware and software setup. Then, Section 5.2 on the next page describes the performance characteristics of our implementation of the TFHE scheme. Later, Section 5.3 on page 53 outlines how the methods of evaluating and choosing our experiments. Further, Section 5.4 on page 56 describes the results of our experiments and evaluate them. Finally, we give a summary of our evaluation findings in Section 5.5 on page 59.

5.1 Experimental Setup

We execute all our experiments, evaluations of these, and benchmarks on the same hardware while ensuring minimal interference by other processes. The computer has an Intel® Xeon® E3-1270 v6 Central Processing Unit (CPU), with 8 hardware threads, at 3.80 *GHz*, and 62 *GiB* Random-Access Memory (RAM). The Operating System (OS) running is Ubuntu 18.04.3 LTS with the Linux 4.15.0-58-generic kernel. All examples are evaluated using the following version of the Rust compiler (rustc): 1.46.0-nightly (118b50524 2020-06-06).

5.2 Library Evaluation

We continuously monitored our library implementation's performance throughout the porting process, to ensure we had no performance regressions when refactoring and cleaning up code to transform it to more idiomatic Rust. Because of this, we have detailed measurements of the performance of the different procedures of our library implementation.

5.2.1 Encryption and Decryption

Figure 5.1 depicts the performance of the encryption and decryption procedures in our library implementation. In particular, it represents the relationship between the running times of the encryption procedure versus the decryption procedure. As one can see from the plot, the encryption procedure is slower than the decryption procedure. The higher execution time is because the encryption procedure includes random number generation and allocation, whereas the decryption procedure is simply some cases of simple arithmetic. The encryption procedure takes an average of 1.65450 μs, but the decryption takes 797.620 ns, or 0.79762 μs (to compare in the same units), meaning the encryption running time is just over 2× the decryption running time.

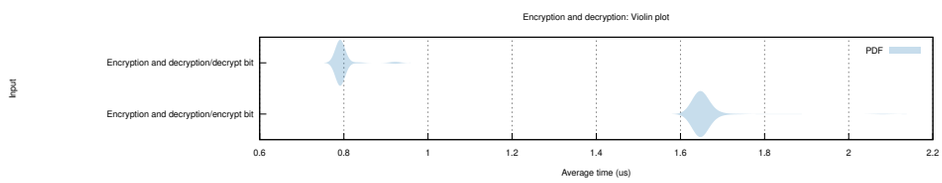


Figure 5.1: Violin plot of the encryption and decryption speeds of our implementation. The blue regions indicate the PDF of our implementation of the encryption and decryption procedures, estimated and smoothened using a kernel density estimator. The horizontal axis indicates the average time of the operation in microseconds (μs or us).

The encryption running time and its estimated Probability Density Function (PDF) can be seen more accurately in Figure 5.2 on the facing page, which also shows each of the samples' running times. As the encryption procedure encrypts individual bits in our setup, because of how we encode values (other setups are possible in the TFHE scheme), the execution time equates to an encryption throughput of 591.33 Kib/s (note bits, not bytes), or 73, 916 KiB/s (using the binary prefix, so kibibytes).

A more accurate view of the decryption running time and its estimated PDF is seen in Figure 5.3 on page 50. The decryption procedure has an execution time which equates to a decryption throughput of 1.1903 Mib/s, or 148, 79 KiB/s.

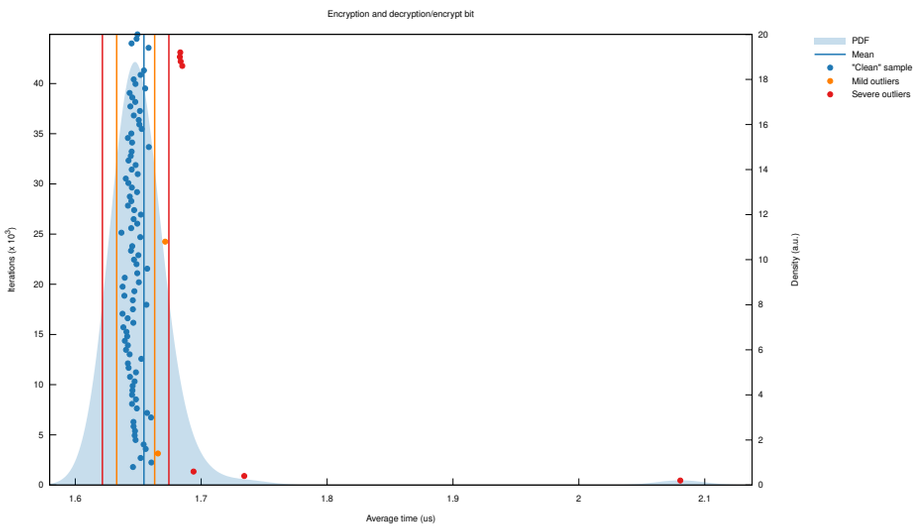


Figure 5.2: Detailed view of the estimated PDF of the encryption procedure. The mean estimate is 1.6545 μs and the median is 1.6467 μs with a standard deviation of 45.062 ns. As our encryption procedure encrypts individual bits, this equals a throughput rate of 591.33 Kib/s, or 73, 916 KiB/s (note the binary prefix).

Both Figure 5.2 and Figure 5.3 on the next page used Tukey's fences to calculate outliers ($k = 1.5$ for mild outliers and $k = 3$ for severe outliers). The outliers in Figure 5.2 are not that far from the mean. However, the plot shows a severe outlier near the 2.1 mark. A probable reason for this is interactions between other processes running at the time of execution. Even though we tried to isolate our experiment as well as possible, and limit the number of processes running while testing, the test is not perfect. The single worst outlier is severe to such an extent that it can be disregarded. The same holds for the results seen in Figure 5.3 on the next page, where a larger amount of outliers occur. These are also most likely due to interactions with different threads running at the same time. Additionally, because the execution times are on such a small scale, small perturbances may affect measurements in a more significant way.

5.2.2 Key Generation

The key generation procedure generates the secret and symmetric key used for encrypting and decrypting data in the TFHE scheme. In addition to this, it creates the bootstrapping key and the key-switching keys required during the bootstrapping process. We collectively name these the bootstrapping key for brevity, as it is the only process using them. As seen in Figure 5.4 on page 51, the key generation uses an average of 527.67 ms to generate the keys. As

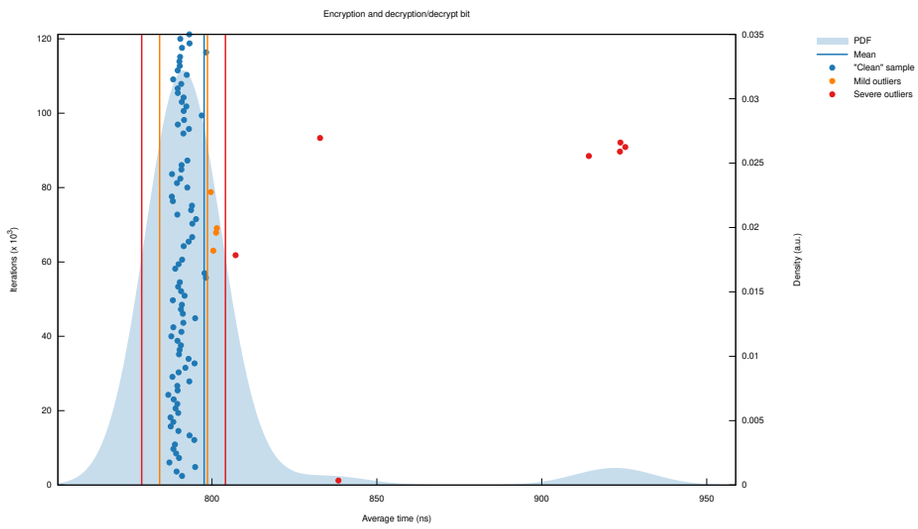


Figure 5.3: Detailed view of the estimated PDF of the decryption procedure. The mean estimate is 797.62 *ns* and the median is 790.77 *ns* with a standard deviation of 26.452 *ns*. As our decryption procedure decrypts individual bits, this equals a throughput rate of 1.1903 Mib/s or 148, 79 KiB/s (note the binary prefix).

this process depends heavily on random number generation, it is affected by fluctuations in time used to generate numbers.

5.2.3 Bootstrapping

Figure 5.5 on page 52 shows the performance characteristics of the bootstrapping procedure. The average execution time of a single bootstrapping procedure is 1.1937 *s*, significantly higher than the implementation of the original paper taking around 53 *ms* on similar hardware [11] and improved work leading to around 13 *ms* [16]. However, the original implementation implements the Lagrange half-complex representation, which reduces the number of multiplications required in the bootstrapping procedure by nearly a third. It also reduces the number of external products required, the expensive operation performed in the bootstrapping procedure. Additionally, the original implementation uses Fast Fourier Transform (FFT) processors based on Single Instruction, Multiple Data (SIMD) instruction sets such as AVX, providing large speedups. The outliers seen in the figure are, similarly to the outliers in the decryption and encryption procedures, related to interactions with other processes using the CPU. As most of the samples fall in a near-identical spot, it is reasonable to assume most results will lie in this range. Additionally, this procedure is deterministic and was benchmarked using the same inputs, so it is natural to

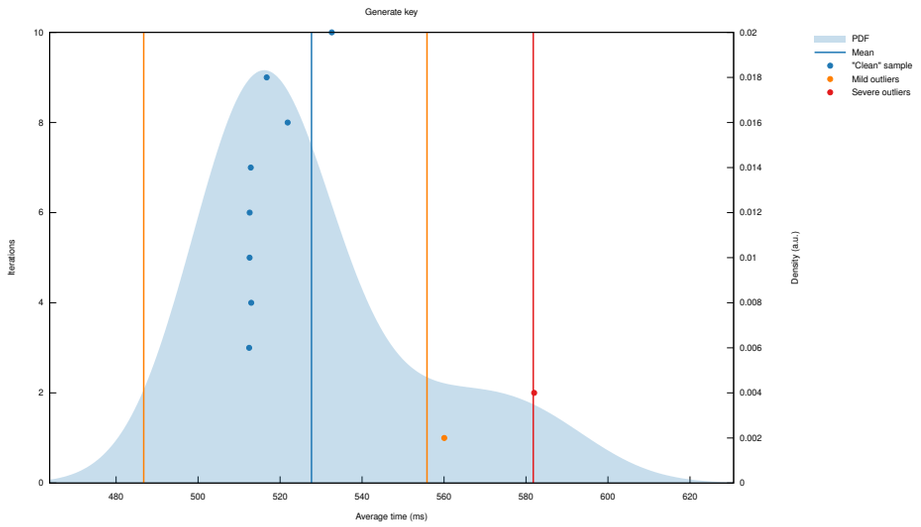


Figure 5.4: Detailed view of the estimated PDF of the key generation procedure. The mean estimate is 527.67 *ms* and the median is 514.85 *ms* with a standard deviation of 24.269 *ms*. The key generation procedure generates both the secret symmetric key, the keyswitching key and the bootstrapping key (which combined result in the *cloud* key).

assume that the outliers are disregardable and that the mean can be used as an estimate.

Comparison Between Optimized and Non-Optimized Implementation

For comparison, we compare the performance differences between our naïve polynomial multiplication procedure, with time complexity of $O(n^2)$, to the FFT-based implementation. Figure 5.6 on page 53 shows that the FFT-based implementation provided a change in execution time of a 74.408% decrease.

One thing to note is that changing to the other parameter set we allow the user to generate keys from, the bootstrapping operation performs $\sim 2\times$ faster while using the FFT-optimized implementation. This result shows that the parameter set used in encryption has a substantial impact on performance. The user should consider using the parameter set guaranteeing 80-bit security for performance increases if they allow the lower security guarantees.

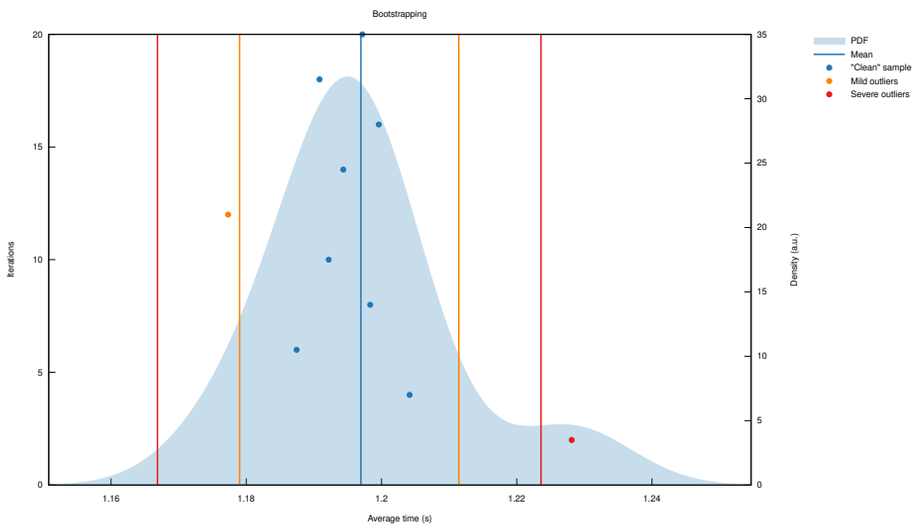


Figure 5.5: Detailed view of the estimated PDF of the bootstrapping procedure. The mean estimate is 1.1937 *s* and the median is 1.1969 *s* with a standard deviation of 13.234 *ms*. The bootstrapping procedure takes an Learning With Errors (LWE) sample as input, along with an output message encoded in the message space, and the bootstrapping and keyswitching keys.

Performance Comparison to the Original Library

We compare our version to the original library implementation on the same hardware using the original library's benchmarks. They perform 50 samples and compute the arithmetic mean as the measure of the bootstrapping operation. They provide several different FFT processors, including FFTW which is remarked as the fastest free FFT implementation available[1]. As noted in the previous chapter, we depend on the RustFFT crate which does not currently use any SIMD instructions, but pure Rust. Thus the most natural comparison would be to compare the original library's implementation when linked with the Nayuki project's portable C implementation[2]. They also provide two benchmarks, where one uses the Lagrange half-complex representation internally, and the other does not. We use the latter benchmark as our implementation does not use the Lagrange representation.

Executing the original library's benchmark using the FFT implementation written in C without SIMD instructions gives us an average of 614.47 *ms* for a single bootstrapping operation. Compared to our implementation, this is only $\sim 2\times$ faster, which is not too bad considering our objective was not to

1. `http://www.fftw.org/`
2. `https://www.nayuki.io/page/fast-fourier-transform-in-x86-assembly`

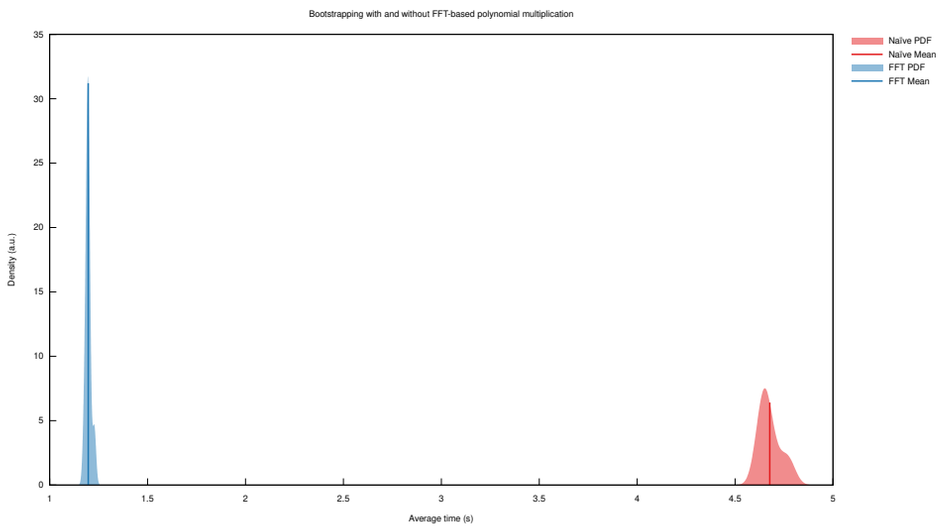


Figure 5.6: The execution time of the bootstrapping operation in our implementation with and without the polynomial multiplication based on FFT. The *Naïve* red region marks the naïve implementation of the polynomial multiplication, while the *FFT* blue region marks the optimized version.

implement a fast implementation, but rather an implementation that was safe in terms of memory safety, easy to use, and would easily allow execution within Software Guard Extensions (SGX).

We also performed the benchmark of the original implementation with all their optimizations included. We use the FFT processor they name spqlios with the FMA instruction set extensions and achieved an execution time of 14.771 *ms*. This number is similar to their findings. This result should not be compared to our's as it implements several more optimizations.

5.3 Evaluation Methodology

To evaluate our hybrid system is complicated. Many factors come into play, and one can measure performance in various ways. Firstly, the parameter set used in the TFHE encryption scheme significantly affects the performance of the homomorphic operations performed on data. Secondly, Fully Homomorphic Encryption (FHE) has a multitude of different use-cases, as one can use it for virtually any situation where secure data processing is needed. This variation means use-cases range from performing simple arithmetic operations on encrypted data, such as in electronic voting systems, to processing-heavy programs such as processing genome data, to more complex programs. How-

ever, more complex programs need to be represented as a logical (boolean) circuit to be processable using the TFHE scheme. Due to the clear limitations of computing circuits on encrypted data, this eliminates circuits which include Data-Dependent Branching (DDB) as the executor cannot and should not be able to determine the result of a comparison resulting in a branch. These limitations make it very difficult to perform general-purpose programs using FHE.

Furthermore, working with SGX is not straightforward, either. An SGX program needs to specify the heap size and stack size before entering an enclave. Additionally, it requires specifying the number of threads available to it. These requirements mean the user has to know the program's memory usage characteristics before executing it in SGX, else resulting in an error. Specifying a too large heap may lead to performance loss.

For our purposes, we wish to discover the relative performance changes when using our hybrid solution to a version which only uses FHE, as covering integrity weaknesses in FHE schemes while retaining performance is our main objective. As such, it is sufficient to evaluate a simple program. This is because performance of other programs can be extrapolated to some degree as we have to know the full circuit and execution flow of a program for it to be encodable for homomorphic evaluation, as long as we know the circuit used in our simple program.

5.3.1 Yao's Millionaires' Problem

Andrew Yao introduced a Secure Multi-Party Computation (SMPC) (computations performed by multiple parties with private inputs) problem in 1982 known as Yao's Millionaires' Problem [73]. The problem is simple and considers two millionaires, Alice and Bob, wishing to figure out which of them is wealthier. However, they wish to find the answer to this problem while at the same time keeping their actual wealth private. That is, Alice or any other party should not learn of Bob's wealth, and vice versa. Essentially, the problem aims to calculate the following: $a \leq b$, where a represents the wealth of Alice in some monetary unit, and b represents Bob's wealth in the same unit, while remaining private for the computing party.

The problem has several solutions, with techniques ranging from oblivious transfer methods [74] to private set intersections with Homomorphic Encryption (HE) [75]. We solve this problem using FHE with our library implementation of the TFHE scheme, by protecting the confidentiality of a and b using encryption.

5.3.2 Socialist Millionaire Problem

The socialist millionaire problem is a modification of Yao's Millionaires' Problem, presented in [76]. In this variant, two millionaires wish to compare their wealth and figure out if it is equal or not, while not disclosing information about their actual wealth to each other. Essentially it aims to calculate the following: $a = b$, where a and b represent the wealth of the two parties, respectively, and are private.

5.3.3 Fused Millionaire Problem

Yao's Millionaires' Problem and the socialist millionaire problem are both problems that seem simple in practice, but they operate under conditions that make them more challenging to solve than one might think. Thus, they are good to use as proofs that a particular system can solve problems in the domain of SMPC. To increase the computational load in our experiments, we consider a fused problem of Yao's Millionaires' Problem and the socialist millionaire problem. In this problem, we aim to figure out the total ordering of two parties' wealth while keeping their actual wealth's private. That is to say when a represents party A's wealth, and b party B's, and both are private, we want to figure out which one of the three cases is true:

- A is wealthier than B
- B is wealthier than A
- A and B have equal wealth

We solve this problem by encrypting the values using the TFHE scheme. Technically, this requires two parties to compute on encrypted data jointly using a multi-key setup. Multi-key HE is possible, as shown in [77], which conveniently turns the TFHE scheme we use into a multi-key scheme, leading to a multi-key FHE scheme. Multi-key FHE may be used for spooky encryption, leading to round-efficient SMPC [78], hence it has several additional use-cases.

Using the multi-key TFHE scheme, the two parties would encode and encrypt their respective amount of wealth, transmit them to a computing node, where their partial evaluation keys are combined, and then the comparisons computed. Our implementation of the TFHE scheme does not support multi-key setups as we based it on an implementation that also did not support it, but implementations exist[3] and the only other operation required for such a setup

3. https://github.com/ilachill/MK-TFHE

is the key combination step, which is not too expensive as it scales linearly with the number of parties, which in our case is two.

Implementation of the Fused Millionaire Problem

In our experiment, we start by producing the binary decomposition of the two values. We use two 32-bit signed integers for this purpose. For each of the values, we decompose them into bytes in big-endian order, then decompose those into the individual bits. We use big-endian as we implemented the circuits we use to work on big-endian values. Then each bit is individually encrypted with our TFHE implementation. We now have two pairs of 32 ciphertexts representing the encryption of the two values. In a multi-key setup, the two parties perform these actions separately after completing a key-exchange protocol.

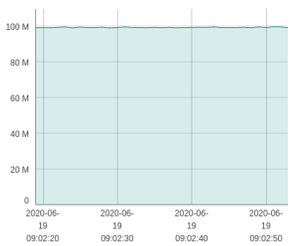
After this, the setup phase is complete. We then perform the comparison circuit equivalent to computing $a \leq b$ and the equality circuit equivalent to $a = b$, both computing on a list of encrypted bits (two pairs of 32-bits) producing encrypted results. These two circuits are independent and are thus evaluable in parallel, although our implementation performs them sequentially.

5.4 Experiment Evaluation

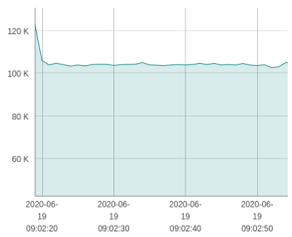
We evaluated our hybrid FHE and SGX implementation and the FHE-only version with 25 runs each, timing only the relevant sections. Running with 80-bit security, the hybrid solution finished with an arithmetic mean of 90.504 s and a standard deviation of 0.60286 s while the FHE-only solution finished in 116.08 s and a standard deviation of 2.3548 s. These results indicate that the hybrid solution is faster than the FHE-version using our implementation of the TFHE library. The FHE-version is around 28% slower. We did not expect this result due to several reasons an SGX enclave should perform slower, such as memory encryption and paging.

To investigate this discrepancy, we looked into possible reasons this might occur. The most significant difference between the two programs is how an SGX enclave handles memory, so we profiled our non-SGX program using the 128-bit security parameter set (as it uses the most memory) to look into its memory usage characteristics. Figure 5.7 on the next page shows the program's memory usage and the number of allocations and deallocations per second. From 5.7a on the facing page we see that the program constantly uses around 100 MiB of memory. Additionally, it has around 100 k allocations per second and the same number of deallocations per second, seen in Figure 5.7b on the next page

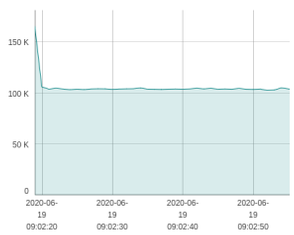
and Figure 5.7c. These allocations are small, usually around a few kibibytes each, correctly relating to the small vectors allocated for numeric processing in our homomorphic library. Memory usage is measured and visualized using an open-source memory profiler[4].



(a) Memory usage during the first part of the program, at a constant usage of around 100 *MiB*.



(b) Memory allocations during the first part of the program, at a constant level around 100 k *allocs*/s. Each allocation is small and always, and sometimes significantly, below 1 *MiB*.



(c) Memory deallocations during the first part of the program, at a constant level around 100 k *deallocs*/s. Each deallocation is small and follows the allocations per second, at around 1 *MiB* or below.

Figure 5.7: Memory usage characteristics of the program using FHE and the default system allocator. Usage is shown in Figure 5.7a, allocations per second in Figure 5.7b and deallocations in 5.7c.

The significant result of memory profiling is that the program does a lot of allocations and deallocations. Regular Rust code relies on the system allocator, which is libc's malloc[5] allocator on Linux, the platform we tested on. However, after consulting with one of the core developers and researcher on the Fortanix Rust EDP platform, we learned that in SGX, Rust uses the dlmalloc[6] allocator, and that it might be the culprit accounting for some of the faster execution time. It emphasizes to minimize memory usage and fragmentation, something that would occur when a program regularly allocates and deallocates memory, which our program does. To see if the allocator could account for the 28% slower execution time, we modified our FHE program to use the dlmalloc allocator and performed the tests again.

The FHE-only program finished faster with the dlmalloc allocator instead of the system allocator. In this test, the average is 93.556 s with a standard deviation of 1.4932 s for the program with 80-bit security and an average of 160.61 s and 3.9251 s. While this result is only around 3% slower than the hybrid solution,

4. `https://github.com/koute/memory-profiler`
5. `https://www.gnu.org/software/libc/manual/html_node/The-GNU-Allocator.html`
6. `http://gee.cs.oswego.edu/dl/html/malloc.html`

it is still slower. All the combined results represented in Table 5.1 and visualized in Figure 5.8.

		80-**bit security**	128-**bit security**
Hybrid solution	*Execution time (s)*	90.504	156.17
	Standard Deviation (s)	0.60286	0.31911
FHE program	*Execution time (s)*	116.08	196.58
	Standard Deviation (s)	2.3548	0.60422
FHE program with dlmalloc	*Execution time (s)*	93.556	160.61
	Standard Deviation (s)	1.4932	3.9251

Table 5.1: Execution time means and standard deviations from the fused millionaire problem experiments. Measurements are calculated from 25 samples.

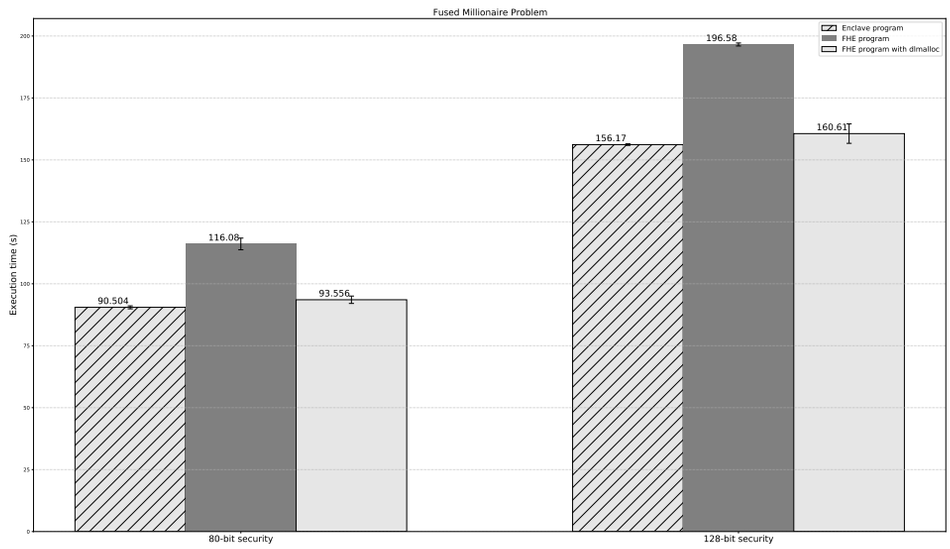


Figure 5.8: Execution times of our fused millionaire problem. Experiments were performed 25 times and respective standard deviations are represented by the vertical error bars.

As is seen from the figure and the table, using 128-bit security significantly impacts performance. The hybrid program executes roughly 72.5% slower, and the FHE-only using dlmalloc for allocation is 71.7% slower than the ones with 80-bit security. As mentioned, this is because ciphertexts in the TFHE scheme grow substantially in size with increased security and thus increases the required computation.

As for the performance difference between the hybrid solution and the solution using only FHE, the execution time difference of only 3% is low. Thus, a user would benefit from using the hybrid solution, which covers the integrity weaknesses of the FHE.

Why the hybrid solution is faster than the program using only FHE is unclear.

However, the standard deviations measured for the FHE-only program are also higher than the hybrid solution, as seen in Table 5.1 on the preceding page, meaning the performances could be more similar than they appear. Another reason the hybrid solution executes faster than the other program could be because of the automatic frequency scaling of the CPU. However, we turned this off, capping the CPU clock frequency at around 4.0 *GHz*, and used a performance setting on the CPU and reran the experiments to receive the same results. We also used strace[7], a diagnostic utility tool, to monitor system calls performed by our program. We witnessed no behavior such as Input/Output (IO) that could imply performance differences between an SGX enclave and standard userspace programs. The system calls mostly consisted of memory allocation calls, which are handled by the enclave memory manager. Our program is single-threaded; it does not use any SIMD instructions, nor include randomness during the measured execution times. If the program performed randomness, it could affect performance as the SGX Software Development Kit (SDK) tries the `RDRAND` instruction up to ten times if it cannot get a random number successfully on the previous tries. However, this is not the case for our program. Thus we are unsure of what causes this behavior. The performance increase in the enclave might lie in implementation details of the SGX SDK optimizing some situations that are normally not possible to optimize, due to the complex nature of an OS and process interactions.

5.5 Summary

In this chapter, we evaluated the performance of our implementation of the TFHE scheme in Rust. We found that the bootstrapping procedure, the most expensive operation of computing on encrypted data in the TFHE scheme, executes at 1.1937 *s* or only about 2× slower than the original library's equivalent implementation. This result is decent, as our main focus was not implementing a fast implementation, but one that would allow execution within SGX while using a memory-safe language and enabling easy use of this system. The original library also has a specialized implementation that utilizes several optimizations such as various internal representations of data and hand-optimized assembly versions of FFT processors, decreasing execution times to around 15 *ms* at best, but at the cost of increased code complexity.

The following sections introduced the millionaires' problem, a SMPC problem, which we combined with another version of the same problem to get a more complex experiment. We implement a solution to this problem using the hybrid approach of performing FHE with our TFHE implementation within

7. `https://strace.io/`

an SGX enclave, and a solution where we only use FHE. We then evaluated these experiments and found that the hybrid solution was, in fact, faster. After investigating reasons for this, we tested the solution using only FHE again with the dlmalloc allocator, and found it was only 3% slower than the hybrid solution, compared to the original 28% slowdown. We are still unsure as to why the hybrid solution is slightly faster, but the error in our measurements might account for some of this difference, and the answer might lie in the SGX SDK implementation. Overall, this is a positive result, as our hybrid solution is both more secure and faster.

6 Discussion

This chapter details discussions on our thesis work. Section 6.1 discusses the reality of our experiment and how real-life examples compare to it. Further, Section 6.2 on page 63 goes on to list several possible improvements we can make to our implementation of the TFHE scheme, both to make it faster in general, but also fit more in line to Intel Software Guard Extensions (SGX) perfomance semantics and limitations. Section 6.3 on page 67 goes on to list some recent attacks on SGX that appeared during our thesis work, and our stance on them in relation to our thesis. Moreover, Section 6.4 on page 68 describes techniques one could use to optimize homomorphic circuits for use in our setup. Eventually, Section 6.5 on page 69 details the possibility for improving our hybrid design by using another platform than the one provided by Fortanix. Lastly, Section 6.6 on page 70 discusses some correctness properties of our implementation of the TFHE scheme and the potential to improve these. Finally, Section 6.7 on page 71 summarizes this chapter.

6.1 Experiment Design Versus Real-World Programs

The fused millionaire problem experiments in our evaluation is an experiment designed to show execution performance characteristics and the relative differences of a hybrid solution to running the program outside SGX. However, this

experiment is not representative of all situations. The experiment manages to stay within the memory limits of the enclave memory bounds. Though, another program might not be able to do this. In the accompanying experiment code to this thesis, we have added a demonstrative program that shows the effects of exceeding the memory limits of the version of SGX we use. It uses around 900 *MiB* Random-Access Memory (RAM) and executes in around 800 *ms*. However, the same program, when executed within an SGX enclave with a preallocated heap of 900 *MiB*, executes in around 4. *s*. This result shows that paging from the Enclave Page Cache (EPC) Protected RAM (PRAM) (currently limited to around 128 *MiB*) to non-protected regions, and back, greatly affects performance. This overhead occurs because the enclave memory manager has to encrypt and decrypt pages and maintain integrity checks to ensure confidentiality for data stored in non-protected RAM. We have not experimented further with varying memory sizes as the way this affects performance varies greatly depending on peculiarities such as memory access patterns, which are highly domain- and program-specific, and compiler implementations may also affect these. Thus it is challenging to draw general conclusions from any experiments we perform on this matter.

The enclave memory manager does not know that data is already encrypted using the TFHE scheme, so data is redundantly double-encrypted when paged out, even though it need not be. Research such as VAULT [79] has shown that this overhead is reducible with more advanced integrity verification structures. However, a more straightforward and probably more performant solution to our hybrid approach of Fully Homomorphic Encryption (FHE) and SGX is to reduce memory overhead and ensure the enclave program does not exceed PRAM limits, if possible. Sometimes this is not possible, as programs require access to many streams of data at once, but in cases where data can fit or rarely exceeds the memory limits of PRAM, a solution where one continuously streams required data to the enclave is preferable. As our experiments showed, a hybrid solution where memory usage falls short of the PRAM limit has similar or negligible performance overhead so that the only real cost would be the transfer to and from an enclave. An interactive system where one continuously transfers data to and from an enclave may also be more viable if the program is particularly memory intensive. However, copying data across enclave boundaries is costly, and one should avoid it if possible, as shown by the *cohesion principle* in [13], an investigation into performance principles when using SGX and the care one must design their system with to ensure efficiency. This overhead might be diminished by the overhead of exceeding enclave memory limits, however. Thus, more experimentation is required to conclude anything more meaningful here.

6.2 Improvements to TFHE Implementation

There are several ways to improve our implementation of the TFHE scheme. In terms of performance and the internal workings of the library, we discuss some of these possible techniques in the sections below.

6.2.1 Internal Representation

The internals of the TFHE scheme relies on structuring data as matrices and polynomials. Most of the internal operations when performing homomorphic operations on ciphertexts are linear transformations of matrices. As such, it would be beneficial to represent data differently than our current implementation does. We store data as nested vectors. Flattening this representation to a one-dimensional vector would be an improvement in terms of memory usage, as a flattened representation implies a single allocation. In contrast, e.g., a two-dimensional vector of size $m \times n$ currently performs at least $1 + m$ allocations in our implementation — one for the top-level vector and one for each of the internal vectors, of which we have m. The number of allocations increases polynomially with an increased number of dimensions (if $n = m = \ldots$). Reducing the number of allocations should theoretically reduce memory fragmentation, which is critical to minimize in SGX due to the performance penalty when having to page out memory to unprotected RAM.

Furthermore, as the internal operations perform calculations on matrices, a linear algebra library would improve the readability of the code. Potentially, it could also improve performance by making optimizations that are easy to miss when manually implementing. Most linear algebra libraries often come with specialized implementations of common operations such as matrix multiplication, where a library can implement complex algorithms developers do not implement due to development costs or lack of knowledge. Rust has a few different and sophisticated libraries for linear algebra already, such as nalgebra[1] and ndarray[2]. Using such a library internally could both improve code clarity and performance.

Of course, the opposite could also possibly be real, where our use-case is so specialized that using a generalized linear algebra library would penalize performance. As evidence against this, we have not encountered implementation details that are complicated enough for a general-purpose linear algebra library not to provide the required functionality. Thus, we stand by the point that using such a library would be an improvement. If one uses strong types for different

1. `https://nalgebra.org/`
2. `https://crates.io/crates/ndarray`

types of matrices (e.g., a type for diagonal matrices, symmetric matrices, or other matrices with special properties), one might even spot situations where one can optimize an operation due to there existing several ways to transform matrices to produce the desired result. A linear algebra library would increase the chances of such a phenomenon.

6.2.2 Memory Management

One way to improve our implementation of the TFHE scheme is to make memory management internally more advanced. As we have seen in our experiments, the way the program uses memory impacts performance in SGX substantially. Rust more easily allows functions to return objects allocated in memory to the caller than C as it moves the object to the caller. It also does not require the caller to explicitly free them, as the ownership and lifetime system alleviates the need for this by dropping (deallocating) the object and its memory when it falls out of scope. This style means that idiomatic Rust may have more allocations and deallocations than in C. This, in turn, affects our system when executing within an SGX enclave, as memory allocations may be more expensive, depending on whether the situation requires paging to untrusted RAM or not. As most of the code in the original TFHE library mostly uses a C-style C++, it uses the style of having functions returning `void` (indicating the function cannot fail) and accepting the result pointer as an argument, and the caller is responsible for both allocation and deallocation. Writing in this interior mutability-style instead of the more functional, immutable style we wrote our implementation in, is more challenging to follow in terms of control-flow but may improve memory management. In Rust, writing much code in this way results in more explicit lifetime management, which will also increase code complexity. The original TFHE library also consistently reuses allocated structures to minimize the need for deallocation and allocation. This reuse along with the fact that C and C++ do not explicitly show which functions mutate raw pointers and which does not makes the code difficult to follow due to implicit mutation of values. Rust excels in readability at this point, due to its compile-time guarantees of no implicit mutation.

6.2.3 SIMD and Parallelization

A benefit of the fact that our current implementation of the TFHE scheme is in a functional style where data is mostly immutable is that it is easily convertible to a parallelized version. Many of the operations we perform are embarrassingly parallelizable. A simple approach would be to spread the necessary work among the available hardware threads, so we looked into where this is most appealing. The bootstrapping process performs what the authors of the TFHE scheme call

the external product between the TGSW and TLWE samples used internally, where the TGSW ciphertext is an intermediate encryption of the encryption key [64]. These multiplications are with the default parameter sets only performed 12 times, closely relating to a typical amount of hardware threads available to processors using Simultaneous Multithreading (SMT) today (logical cores), at around 8. We also perform these multiplications relatively high up in the call stack, something our flamegraphs made easy to identify. That these operations are performed high in the call stack is advantageous as it reduces the potential overhead of thread intercommunication and synchronization compared to spawning threads at a leaf-level function call. Unfortunately, the way we have implemented this specific functionality is through an aggregative function (a functional fold-style), and thus it is a serialized and sequential process. This part will require some work to unfold to allow for safe concurrency. There should be other places to parallelize code from; however, it will require more investigation and testing to determine if it is worth it.

Rust is propitious with regards to parallelization as one of its mottos is to ensure *fearless concurrency*[3]. The ownership and lifetime part of the type system ensures that code that is unsafe to parallelize is unable to compile. This safety has fueled the growth of efficient and easy to use parallelization libraries for Rust such as Rayon[4]. They base their techniques on the results of the Cilk Project[5], and dynamically handles task scheduling through work-stealing. Our implementation of the TFHE scheme does not use any primitives for interior mutability, and all types are thread-safe (implements the `Send` and `Sync` marker traits[6]), so using Rayon in our code should be a simple matter to implement. However, in SGX, thread synchronization is more expensive, and parallelizing our implementation using multithreading will require some testing to ensure we do not impose a greater overhead than the time we cut by parallelization. As our bootstrapping operation takes just over a second to complete, it seems the implementation has much to gain, considering thread synchronization takes a substantially lower amount of time.

Furthermore, considering many of the internal operations are only element-wise binary operations applied to vectors of integers, using Single Instruction, Multiple Data (SIMD) instruction sets should be able to speed up performance quite considerably. Most internal vectors are at least 1024 elements in length, depending on the encryption parameters used, so they are also aligned to the width of the data lanes used in many instructions. In the current implementation, we represent the values (coefficients of matrices) as signed 32-bit integers,

3. https://doc.rust-lang.org/book/ch16-00-concurrency.html
4. https://crates.io/crates/rayon
5. http://supertech.csail.mit.edu/cilk/
6. https://doc.rust-lang.org/nomicon/send-and-sync.html

so using a 512-bit wide instruction should theoretically improve performance by around 16×, but there are several caveats. Not all SIMD instructions that offer processing 512 bits handle 16 32-bit integers, but rather 8 64-bit integers, or some other setup. Furthermore, which SIMD instruction set a Central Processing Unit (CPU) offers depends on which manufacturer one uses (e.g., Intel versus AMD), which SIMD extensions are supported (e.g., if one uses older processors), and more. Implementing SIMD support requires a lot of internal conditional compilation (or runtime intrinsics identification) to ensure the current processor architecture supports the instruction set extensions used. This manual handling of different CPU architectures leaves room for many bugs, as conditional compilation is hard to manage, especially with the various architectures available. If our implementation should be available for general use, it should handle all of those conditions internally, so users need not worry.

Additionally, using SIMD instructions in Rust is unsafe, as it involves directly handling CPU instructions and is hardware-specific, something the compiler does not necessarily have control over as one might compile for a different platform than the one compiles on. When using unsafe Rust, the developer needs to ensure that the code is actually safe, lest they risk introducing Undefined Behavior (UB). Thus, introducing SIMD instructions as means of parallelizing code might be beneficial, but at the cost of considerably increased code complexity, reducing internal code clarity and readability.

6.2.4 Ciphertext Packing

The TFHE scheme has had several improvements since its inception, and the original authors revisit the scheme and improve it in [64]. It brings a way to do circuit bootstrapping in the scheme, which in turn improves the Leveled Homomorphic Encryption (LHE) functionality of the scheme, providing speedups to arithmetic functions. Our current implementation does not support the LHE mode of the scheme. Furthermore, it presents new techniques for packed ciphertexts, which can improve upon a homomorphic Lookup Table (LUT). Ciphertext packing is when a ciphertext encrypts more than one piece of data so that homomorphic operations can operate on packed ciphertexts to, in a sense, provide SIMD operations on encrypted data. They use this ciphertext packing technique to create a LUT over a function's inputs (multiple bits), and use techniques they call horizontal and vertical packing to speed up the evaluation of such a function, evaluating all of its outputs at once. Mainly, using ciphertext packing and encoding functions as a homomorphic LUT can speed up the evaluation of those functions linearly, by up to a factor of n when packing n values. If we were to implement this functionality in our implementation along with support for LHE, we could support larger circuits

and the circuit bootstrapping technique they present. However, implementing this functionality is more complicated than some of the other improvements we can make, as this relies more on the paper's cryptographic reasoning, something the author of this thesis does not have extensive knowledge of. Additionally, the original library does also not implement this functionality yet, a testament to its inherent developmental complexity.

Ciphertext packing would bring more benefits than the performance gain, with regards to our hybrid solution. Packing ciphertexts also implies a smaller memory footprint, which is highly advantageous as exceeding SGX enclave memory limits is expensive. A lower memory usage equates to a more substantial proportion of programs that can efficiently execute in our hybrid solution as they can access more memory in a sense. Thus, it is a compelling long-term improvement even though it requires more effort than some of the other changes we mention.

6.3 Recent Attacks on SGX

Throughout the later parts of when we worked on this thesis, three new attacks to SGX were published, namely the LVI [52], CacheOut [6], and SGAxe [51] attacks. We do not consider these attacks in our design, as both LVI and CacheOut break integrity guarantees that our hybrid construction relies on, and we do not have enough time to investigate the impact this has on our work. The SGAxe attack builds on the weaknesses exploited in CacheOut and allows an untrusted party to remotely (and falsely) attest that they are an authenticated SGX enclave. Our hybrid solution exists only to cover the integrity weaknesses a FHE scheme such as TFHE has, mainly allowing data injection or ciphertext manipulation. Mitigating these attacks is difficult, and some parts require hardware changes to cover them without imposing substantial performance penalties.

Intel's guidance[7] on mitigating the LVI attack relies on compilation changes, inserting `LFENCE` instructions to serialize memory accesses, possibly after every memory load [52]. These changes can significantly affect performance, depending on the program's memory access patterns, imposing overheads of up to potentially 19×. We performed our experiments using a Rust version which does not apply the software mitigations for the LVI attack. However, we reran our hybrid experiment program with 128-bit security using a Rust version

7. `https://software.intel.com/security-software-guidance/insights/deep-dive-load-value-injection`

that applies this patch unconditionally[8], released relatively close to the end of our thesis work. In this situation, the evaluation takes around 3.3× longer than without the mitigations. This result shows that the mitigations impose significant overhead on our use-cases.

Nevertheless, we also consulted a security researcher at Fortanix and the author of the software patches to Rust. We discussed the implications of the mitigations to the LVI attack and concluded that future hardware changes would mitigate these attacks entirely, probably without imposing as substantial performance penalties as the current software patches do. Hence, we pretend their nonexistence in our work.

6.4 Circuit Optimization

As the TFHE scheme requires one to represent homomorphic programs as boolean circuits, a logical step forward would be to look at the optimization techniques known for boolean and logical circuits. Optimizing boolean logic has been researched for a long time, with famous techniques such as Karnaugh maps [80] from the mid 20th century. Karnaugh maps allow easy identification of inefficient circuits and their simplification through pattern recognition. They also enable simple identification and elimination of race hazards. Karnaugh maps are simple for human use, while the Quine-McCluskey algorithm is more friendly for computer use through its tabular form, which allows easy identification of if the circuit optimized to its minimal form [81].

Circuit optimization is highly relevant in Homomorphic Encryption (HE) as circuits are nearly prohibitively expensive in terms of whether it is worth using or not. Thus, performing an optimization before execution will most likely significantly outweigh the costs of the non-optimized circuit's execution costs. HE has the property that the noise growth function of ciphertext multiplication is higher than the noise growth function of ciphertext addition, meaning one can perform more additions than multiplications as the noise grows slower than with multiplication. This noise-growth relationship is one of the reasons homomorphic circuits are measured using the term multiplicative depth to measure the number of multiplication operations permitted on a ciphertext before reaching the noise limit [1]. As such, optimization techniques for optimizing homomorphic circuits to a more efficient form exists, where one notable example is the *cone rewrite operator* [82]. Using a specialized operator, one can focus both on minimizing circuit size and reducing multiplicative depth, which positively affects performance.

8. `https://github.com/rust-lang/rust/pull/72655`

We have not looked into or used circuit optimization techniques in our evaluation, as we only compared relative performance metrics between using pure FHE and our hybrid design, and it is out of scope. As such, the circuits we used in our evaluation experiments are inefficient and unoptimized. As an example, our `circuits` module exposes an adder circuit based on the ripple-carry adder. This circuit is known to be a relatively slow adding circuit, and we could have improved this by using a parallel carry-lookahead circuit such as the Lynch–Swartzlander spanning tree adder [83], or even by only using an improved ripple-carry adder [84]. Needless to say, there are many techniques one can use from the digital logic circuit research to improve homomorphic circuits. We leave this as future work.

6.5 Rust-SGX SDK

We use the Fortanix Rust Enclave Developer Platform (EDP) in our implementation, but another Software Development Kit (SDK) for SGX is available in Rust at the time of writing, the Rust-SGX project, now a part of the Apache Software Foundation[9], but formerly a project by Baidu. The Rust-SGX project has published research on its design and compared it to the Fortanix alternative in [62]. They perform a small set (5 programs) of macrobenchmarks, where the Fortanix version has a normalized overhead compared to the Rust-SGX version ranging from 8% to 54% slower at the most. However, they do note that the example benchmarks are heavily using the `ECALL`/`OCALL` instructions, which the Fortanix Rust EDP replaces with usercalls. Furthermore, they note that Fortanix's design uses Rust's standard library while their design wraps the Intel SDK and optimizes certain parts with more unsafe code and interaction with assembly code [62, Sec. 3.3].

These benchmarks indicate that our hybrid design could, in theory, perform better if we used the Rust-SGX SDK instead of the Fortanix Rust EDP it currently uses. It does seem that this approach would lose some benefits, however. The Fortanix Rust EDP is an easy-to-use platform where the user can run unmodified programs within SGX. Using Fortanix's platform with our library means a user only needs to create a new project, add our library as a dependency and set up the Fortanix platform as a compile target, and they are ready to use FHE through the TFHE scheme within SGX. The Rust-SGX SDK approach is to build upon Intel's SDK and provide bindings to a set of trusted APIs. This means the Rust-SGX approach requires more work to achieve functional software, but it might simultaneously provide better performance. However, they do focus on application layer memory safety, though without providing mechanisms to

9. `https://github.com/apache/incubator-teaclave-sgx-sdk`

mitigate side-channel attacks such as [3] and [85], something Fortanix's Rust EDP does to some extent (such as the LVI attack mitigation). However, with the time we have at hand, we have not investigated and compared these solutions any further and leave it as future work.

6.6 Implementation Correctness

Cryptographic software is different from other types of software in that it often mostly manipulates numbers using arithmetic. In general-purpose software, the programs often handle many different classes of data, which may allow general-purpose programs to use static typing to a greater extent. That is, general-purpose programs may comprehensively utilize the type systems, e.g., using different types for valid data versus unvalidated data. In cryptographic software, the different numbers often have unique intrinsic properties that may be hard to model in the type system without losing performance or increasing code complexity. As an example, numbers may be part of a particular quotient group, such as $\mathbb{Z}/4\mathbb{Z}$ (the set of integers modulo 4), a value of a bounded set such as $\mathbb{N}^{<10}$ (the natural numbers less than 10), or a matrix of shape $m \times n$. These might be difficult to model in type systems that do not have fully dependent types, such as Idris [86] or Agda [87], without losing performance due to runtime checks. As such, cryptographic software is challenging to type statically and provide meaningful, informative types while remaining performant. For example, the original library uses the signed 32-bit integer for most of the computations, but in some cases, the numbers represent a number in a torus representing up to 32-bit sized numbers. However, they are declared using a `typedef`, but this only creates an alias, not a distinct, nominal type (often called a newtype). This use means that implicit conversions from the torus type to the regular signed integer type occur, which is correct according to the C++ specification, leading to a loss of type safety.

Our implementation of the TFHE scheme tries to be strict with typing and to leverage the type system where we can, but falls short in some cases. Rust has, at the time of writing, experimental support for *const generics*[10], or functions and values that are generic over constant values. This functionality is a subset of fully dependent types and could help improve some compile-time guarantees.

We are currently aware of an issue where our implementation is unsound. The bootstrapping operation we have implemented fails in some circumstances.

10. https://github.com/rust-lang/rfcs/blob/master/text/2000-const-generics.md

Encryption in the TFHE scheme is semantically secure, meaning it involves random sampling to encrypt data. In some cases, we experience an incorrect result after performing the bootstrapping operation. Ciphertexts in the TFHE scheme have noise limits, meaning that if a ciphertext's noise exceeds the noise limit specified by the encryption parameters, its decryption may result in an incorrect value. We have spent a significant amount of time investigating this, and have carefully investigated our implementation for algorithmic correctness. We found that our implementation performs the correct operations, so the error must lie in an incorrect noise handling or subtleties such as integer overflow semantics. The original library relies on what is technically UB, so their compiled library might operate differently than the Rust equivalent, as the same operations in Rust are not UB. We have not investigated assembly output differences as this is difficult due to the sheer amount of output and the limited amount of time at hand. However, as we found that our implementation performs the same number of operations as it should, and does not exhibit this incorrect behavior in all situations, we have decided to rely on our evaluation in the preceding chapter, as it tests performance characteristics and not correctness properties of the HE operations. That is, the implementation performs equivalently if the inaccuracy issue is there or not, and as it probabilistically appears most likely due to semantic security properties or the fact that the original library relies on UB and may have this incorrectness property itself, we can still perform our evaluation without inaccurate performance results. Future work and effort will most likely discover the source of this inaccuracy and fix it.

6.7 Summary

In this chapter, we discussed several aspects of our thesis and the potential for improving it. We discussed our experiments and how it relates to real-life usages, where real-life programs might have a higher memory usage. We discussed techniques such as interactivity for minimizing memory usage, although it seems impractical.

Furthermore, we discussed several techniques for improving our implementation of the TFHE scheme. We mentioned techniques such as changing the internal representation to decrease memory allocations and potentially reduce memory fragmentation, using linear algebra libraries to improve performance and readability and using more internal mutability to reduce allocations. We also discussed parallelizing the implementation through SIMD intrinsics and multithreading, and finally using more advanced techniques such as ciphertext packing to enable homomorphic SIMD behavior.

Next, we mentioned the recent LVI, CacheOut, and SGAxe attacks published during our thesis work. We discuss some of their impact and our decision on how we do not consider them in this thesis.

Further, we discussed the possibility of optimizing homomorphic circuits and the potential benefits they bring. They are engaging in terms of the performance benefits they provide and a smaller memory footprint, something which would improve overall performance in our hybrid design.

Eventually, we discussed the potential performance benefits from using the Rust-SGX SDK instead of the Fortanix Rust EDP. However, we note that this requires more development work as it is more complicated to use than the Fortanix Rust EDP.

Lastly, we discussed the correctness properties of our implementation of the TFHE scheme. We noted that we could improve our implementation could by further work investigating in the nondeterministic bootstrapping failure we probabilistically experience, but that it does not affect our evaluation as it is an orthogonal problem regarding performance. We also mention how a dependent type system could improve parts of our program for stronger compile-time guarantees, and that experimental features in Rust are worked on to provide similar functionality.

7

Conclusion

This thesis has combined the techniques of Fully Homomorphic Encryption (FHE) for computing on encrypted data, and performing operations within a Trusted Execution Environment (TEE). This combination was done to alleviate integrity weaknesses of FHE schemes while covering up flawed confidentiality guarantees of an implementation of a TEE, specifically Intel Software Guard Extensions (SGX). As such, the combined solution achieves a greater degree of security, with both data and code confidentiality and integrity guarantees. Additionally, we have used a memory-safe programming language to provide compile-time guarantees against a multitude of potential errors, which could lead to security vulnerabilities.

7.1 Concluding Remarks

For us to achieve the aforementioned goals, this thesis implemented and evaluated a library for performing FHE, specifically the TFHE [11] scheme, written in pure Rust, call it TFHE-rs. Our TFHE-rs implementation was based on an existing library written in a mix of C and C++ [16], call it TFHE-c. Compared to TFHE-c, TFHE-rs is around 2× slower at the worst, although TFHE-rs' performance is still reasonable considering we implemented it with correctness in mind, not performance. The performance of TFHE-rs is acceptable, and could with further work prove to be on par with TFHE-c.

The central aspect of this thesis was to evaluate TFHE-rs' relative performance in conjunction with Intel SGX and without it, running as a standalone executable outside the confines of SGX. Our evaluation in Chapter 5 on page 47 found that one can use TFHE-rs within an SGX enclave feasibly. The evaluation showed that the hybrid solution using TFHE-rs and SGX is 3% faster than the version using only TFHE-rs without SGX. This result is not in line with what we conjectured, which was that the hybrid solution with TFHE-rs and SGX should be slower. However, the measured standard deviation does account for most of the performance difference, and the benchmarks themselves take long enough for this discrepancy to be due to environmental factors in our experimental setup (i.e., due to system load). Thus, we concluded that our hybrid solution with TFHE-rs inside an SGX enclave has a negligible overhead and is a feasible design while simultaneously providing more reliable security guarantees. We did find that the way memory is managed greatly affects performance. The default system allocator on Linux (libc's malloc) was near 28% slower than the dlmalloc allocator we used in the SGX setup. As such, a system with a similar setup to ours should emphasize low memory usage and experiment with different allocators to ensure that they stay within the memory limits imposed by SGX to the utmost extent.

To summarize, our thesis presented TFHE-rs, a library for performing the TFHE FHE scheme, written in Rust, a memory-safe language. It embeds in SGX as a single dependency by using the Fortanix Rust EDP. There is no user-required configuration apart from the minimum required for creating an SGX enclave. TFHE-rs provides pre-made circuits to make it easy for users to create common circuits and built-in serialization and deserialization support for easy transfer to and from enclaves. Further, our thesis evaluated our system by measuring the performance characteristics of TFHE-rs with and without an SGX enclave and found that performance overhead is negligible. Thus, we conclude that using FHE operations within SGX, written in the memory-safe language Rust, is both feasible and provides several additional security guarantees, given that the developer ensures a reasonable memory usage. Moreover, our discussion in Chapter 6 on page 61 provides several possible improvements and optimizations for TFHE-rs and users of the hybrid setup, which could further improve this design.

7.2 Future Work

As we saw in our experimental implementation of the TFHE scheme in Chapter 5 on page 47, our current implementation is up to 2× slower than the original TFHE implementation using the bootstrapping operation. As our implementation was mostly an experimental implementation for use in SGX, it leaves

some room for improvement. As we detailed in the discussion in Chapter 6 on page 61, we already know of a few ways to improve it. The most obvious implementation would be to use the Lagrange half-complex representation described in [64], representing numbers internally as the complex roots of unity. Further, another noticeable improvement would be parallelization using multithreading, as our implementation is conveniently written to be easily parallelized. Other possible improvements include Single Instruction, Multiple Data (SIMD) instructions, linear algebra libraries, managing memory with a higher granularity to reduce allocations, and implement support for ciphertext packing and encoding functions as a Lookup Table (LUT).

In addition to improving our implementation of the TFHE scheme, a step toward building a more easy-to-use framework for FHE is to add support for circuit netlist compilation. A netlist is a description of the connectivity within a circuit. Netlists for large, well-known circuits are available online, and a compiler could compile these into a representation our library could execute. Support for this has some precedence as the authors of the original TFHE library has shown that their implementation can support automatically generated circuits. Supporting compiling circuit netlists and their evaluation in our library could substantially improve the ease-of-use of our library.

Bibliography

[1] C. Gentry, *A fully homomorphic encryption scheme*. PhD thesis, Stanford University, 2009. https://crypto.stanford.edu/craig/.

[2] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms," *Foundations of Secure Computation, Academia Press*, pp. 169–179, 1978.

[3] Y. Xu, W. Cui, and M. Peinado, "Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems," in *2015 IEEE Symposium on Security and Privacy*, pp. 640–656, May 2015.

[4] F. Brasser, U. Müller, A. Dmitrienko, K. Kostiainen, S. Capkun, and A.-R. Sadeghi, "Software Grand Exposure: SGX Cache Attacks Are Practical," in *11th USENIX Workshop on Offensive Technologies (WOOT 17)*, (Vancouver, BC), USENIX Association, Aug. 2017.

[5] J. Wang, Y. Cheng, Q. Li, and Y. Jiang, "Interface-Based Side Channel Attack Against Intel SGX," *CoRR*, vol. abs/1811.05378, 2018.

[6] S. van Schaik, M. Minkin, A. Kwong, D. Genkin, and Y. Yarom, "CacheOut: Leaking Data on Intel CPUs via Cache Evictions." https://cacheoutattack.com/, 2020.

[7] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, "Memory Errors: The Past, the Present, and the Future," in *Research in Attacks, Intrusions, and Defenses* (D. Balzarotti, S. J. Stolfo, and M. Cova, eds.), (Berlin, Heidelberg), pp. 86–106, Springer Berlin Heidelberg, 2012.

[8] M. Miller, "Trends, Challenges, and Strategic Shifts in the Software Vulnerability Mitigation Landscape." Presented at BlueHat IL, Tel Aviv, Israel. [Online; accessed 14-April-2020] https://youtu.be/PjbGojjnBZQ?t=848, 2019.

[9] Y. Shen, Y. Chen, K. Chen, H. Tian, and S. Yan, "To Isolate, or to Share?

That is a Question for Intel SGX," in *Proceedings of the 9th Asia-Pacific Workshop on Systems*, APSys '18, (New York, NY, USA), Association for Computing Machinery, 2018.

[10] W. Wang, Y. Jiang, Q. Shen, W. Huang, H. Chen, S. Wang, X. Wang, H. Tang, K. Chen, K. E. Lauter, and D. Lin, "Toward Scalable Fully Homomorphic Encryption Through Light Trusted Computing Assistance," *CoRR*, vol. abs/1905.07766, 2019.

[11] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds," in *Advances in Cryptology – ASIACRYPT 2016* (J. H. Cheon and T. Takagi, eds.), (Berlin, Heidelberg), pp. 3–33, Springer Berlin Heidelberg, 2016.

[12] A. T. Gjerdrum, H. D. Johansen, L. Brenna, and D. Johansen, "Diggi: A Secure Framework for Hosting Native Cloud Functions with Minimal Trust." Unpublished, 2019.

[13] A. T. Gjerdrum, R. Pettersen, H. D. Johansen, and D. Johansen, "Performance Principles for Trusted Computing with Intel SGX," in *Cloud Computing and Service Science* (D. Ferguson, V. M. Muñoz, J. Cardoso, M. Helfert, and C. Pahl, eds.), (Cham), pp. 1–18, Springer International Publishing, 2018.

[14] P. Denning, D. Comer, D. Gries, M. Mulder, A. Tucker, J. Turner, and P. Young, "Computing as a discipline," *Computer*, vol. 22, pp. 63–70, 03 1989.

[15] I. Anati, S. Gueron, S. P. Johnson, and V. R. Scarlata, *Innovative Technology for CPU Based Attestation and Sealing*. Intel Corporation, Aug. 2013. `https://software.intel.com/en-us/articles/innovative-technology-for-cpu-based-attestation-and-sealing`.

[16] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Library," August 2016. https://tfhe.github.io/tfhe/.

[17] T. Elgamal, "A public key cryptosystem and a signature scheme based on discrete logarithms," *IEEE Transactions on Information Theory*, vol. 31, pp. 469–472, July 1985.

[18] P. Paillier, "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes," in *Advances in Cryptology — EUROCRYPT '99* (J. Stern, ed.), (Berlin, Heidelberg), pp. 223–238, Springer Berlin Heidelberg, 1999.

[19] F. Armknecht, C. Boyd, C. Carr, K. Gjøsteen, A. Jäschke, C. A. Reuter, and M. Strand, "A Guide to Fully Homomorphic Encryption." Cryptology ePrint Archive, Report 2015/1192, 2015. https://eprint.iacr.org/2015/1192.

[20] C. Gentry, "Computing Arbitrary Functions of Encrypted Data," *Commun. ACM*, vol. 53, pp. 97–105, Mar. 2010.

[21] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "(Leveled) Fully Homomorphic Encryption without Bootstrapping," in *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, (New York, NY, USA), p. 309–325, Association for Computing Machinery, 2012.

[22] L. Ducas and D. Micciancio, "FHEW: Bootstrapping Homomorphic Encryption in Less Than a Second," in *Advances in Cryptology – EUROCRYPT 2015* (E. Oswald and M. Fischlin, eds.), (Berlin, Heidelberg), pp. 617–640, Springer Berlin Heidelberg, 2015.

[23] J. H. Cheon, A. Kim, M. Kim, and Y. Song, "Homomorphic Encryption for Arithmetic of Approximate Numbers," in *Advances in Cryptology – ASIACRYPT 2017* (T. Takagi and T. Peyrin, eds.), (Cham), pp. 409–437, Springer International Publishing, 2017.

[24] Z. Brakerski, C. Gentry, and V. Vaikuntanathan, "Fully Homomorphic Encryption without Bootstrapping." Cryptology ePrint Archive, Report 2011/277, 2011. https://eprint.iacr.org/2011/277.

[25] P. W. Shor, "Algorithms for quantum computation: discrete logarithms and factoring," in *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, Nov. 1994.

[26] D. Dolev, C. Dwork, and M. Naor, "Non-Malleable Cryptography," *SIAM Journal of Computing*, vol. 30, Mar. 2001.

[27] S. Goldwasser and S. Micali, "Probabilistic Encryption & How to Play Mental Poker Keeping Secret All Partial Information," in *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, (New York, NY, USA), p. 365–377, Association for Computing Machinery, 1982.

[28] A. Das, S. Dutta, and A. Adhikari, "Indistinguishability against Chosen Ciphertext Verification Attack Revisited: The Complete Picture," in *Provable Security* (W. Susilo and R. Reyhanitabar, eds.), (Berlin, Heidelberg), pp. 104–120, Springer Berlin Heidelberg, 2013.

[29] T. Acar, M. Belenkiy, M. Bellare, and D. Cash, "Cryptographic Agility and its Relation to Circular Encryption," in *EUROCRYPT 2010*, Springer Verlag, May 2010.

[30] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway, "Relations among notions of security for public-key encryption schemes," in *Advances in Cryptology — CRYPTO '98* (H. Krawczyk, ed.), (Berlin, Heidelberg), pp. 26–45, Springer Berlin Heidelberg, 1998.

[31] N. Emmadi, P. Gauravaram, H. Narumanchi, and H. Syed, "Updates on Sorting of Fully Homomorphic Encrypted Data," in *2015 International Conference on Cloud Computing Research and Innovation (ICCCRI)*, pp. 19–24, Oct. 2015.

[32] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh, "Terra: A Virtual Machine-based Platform for Trusted Computing," in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, (New York, NY, USA), pp. 193–206, ACM, 2003.

[33] OMTP limited, *Advanced trusted environment: omtp tr1 v1.1*, May 2009.

[34] "TEE system architecture." `http://www.globalplatform.org/specificationsdevice.asp`, 2011. [Online; accessed 19-May-2020].

[35] A. Vasudevan, J. M. McCune, and J. Newsome, *Trustworthy execution on mobile devices*. Springer Publishing Company, 2013.

[36] J. M. Rushby, "Design and Verification of Secure Systems," in *Proceedings of the Eighth ACM Symposium on Operating Systems Principles*, SOSP '81, (New York, NY, USA), pp. 12–21, ACM, 1981.

[37] T. Alves and D. Felton, "TrustZone: Integrated Hardware and Software Security," Jan. 2004.

[38] ARM Limited, *ARM Security Technology Building a Secure System using TrustZone® Technology*. ARM Limited, Apr. 2009. `http://infocenter.arm.com/help/topic/com.arm.doc.prd29-genc-009492c/PRD29-GENC-009492C_trustzone_security_whitepaper.pdf`.

[39] V. Scarlata, S. Johnson, J. Beaney, and P. Zmijewski, *Supporting Third Party Attestation for Intel® SGX with Intel® Data Center Attestation Primitives*. Intel Corporation, 2018. `https://software.intel.com/sites/default/files/managed/f1/b8/intel-sgx-support-for-third-party-attestation.pdf`.

[40] B. Parno, "Bootstrapping Trust in a "Trusted" Platform," in *Proceedings of the 3rd Conference on Hot Topics in Security*, HOTSEC'08, (Berkeley, CA, USA), pp. 9:1–9:6, USENIX Association, 2008.

[41] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens, "Plundervolt: Software-based Fault Injection Attacks against Intel SGX," in *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[42] D. Genkin, A. Shamir, and E. Tromer, "RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis," in *Advances in Cryptology – CRYPTO 2014* (J. A. Garay and R. Gennaro, eds.), (Berlin, Heidelberg), pp. 444–461, Springer Berlin Heidelberg, 2014.

[43] C. Percival, "Cache missing for fun and profit," Aug. 2009.

[44] S. Chen, X. Zhang, M. K. Reiter, and Y. Zhang, "Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '17, (New York, NY, USA), pp. 7–18, ACM, 2017.

[45] O. Goldreich, "Towards a Theory of Software Protection and Simulation by Oblivious RAMs," in *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, (New York, NY, USA), pp. 182–194, ACM, 1987.

[46] E. Stefanov and E. Shi, "Path O-RAM: An Extremely Simple Oblivious RAM Protocol," *CoRR*, vol. abs/1202.5150, 2012.

[47] X. Wang, H. Chan, and E. Shi, "Circuit ORAM: On Tightness of the Goldreich-Ostrovsky Lower Bound," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, (New York, NY, USA), pp. 850–861, ACM, 2015.

[48] S. Sasy, S. Gorbunov, and C. W. Fletcher, "ZeroTrace : Oblivious Memory Primitives from Intel SGX," in *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, 2018.

[49] P. Mishra, R. Poddar, J. Chen, A. Chiesa, and R. A. Popa, "Oblix: An Efficient Oblivious Search Index," *2018 IEEE Symposium on Security and Privacy (SP)*, pp. 279–296, 2018.

[50] M. Marlinspike, "Technology preview: Private contact discovery for

Signal," Sept. 2017. https://signal.org/blog/private-contact-discovery/.

[51] S. van Schaik, A. Kwong, D. Genkin, and Y. Yarom, "SGAxe: How SGX fails in practice." https://sgaxeattack.com/, 2020.

[52] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens, "LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection," in *41th IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[53] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown: Reading kernel memory from user space," in *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[54] N. Drucker and S. Gueron, "Achieving trustworthy Homomorphic Encryption by combining it with a Trusted Execution Environment," *JoWUA*, vol. 9, pp. 86–99, 2018.

[55] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "CryptDB: Protecting Confidentiality with Encrypted Query Processing," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 85–100, ACM, 2011.

[56] S. D. Tetali, M. Lesani, R. Majumdar, and T. Millstein, "MrCrypt: Static Analysis for Secure Cloud Computations," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, (New York, NY, USA), pp. 271–286, ACM, 2013.

[57] M. N. Sadat, M. M. A. Aziz, N. Mohammed, F. Chen, S. Wang, and X. Jiang, "SAFETY: Secure gwAs in Federated Environment Through a hYbrid solution with Intel SGX and Homomorphic Encryption," *CoRR*, vol. abs/1703.02577, 2017.

[58] "Microsoft SEAL (release 3.4)." https://github.com/Microsoft/SEAL, Oct. 2019. Microsoft Research, Redmond, WA.

[59] S. Halevi and V. Shoup, "Algorithms in HElib," in *Advances in Cryptology – CRYPTO 2014* (J. A. Garay and R. Gennaro, eds.), (Berlin, Heidelberg), pp. 554–571, Springer Berlin Heidelberg, 2014.

[60] C. Gentry, A. Sahai, and B. Waters, "Homomorphic Encryption from Learn-

ing with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based," in *Advances in Cryptology – CRYPTO 2013* (R. Canetti and J. A. Garay, eds.), (Berlin, Heidelberg), pp. 75–92, Springer Berlin Heidelberg, 2013.

[61] R. Silva, P. Barbosa, and A. Brito, "DynSGX: A Privacy Preserving Toolset for Dynamically Loading Functions into Intel(R) SGX Enclaves," *CoRR*, vol. abs/1710.11423, 2017.

[62] H. Wang, P. Wang, Y. Ding, M. Sun, Y. Jing, R. Duan, L. Li, Y. Zhang, T. Wei, and Z. Lin, "Towards Memory Safe Enclave Programming with Rust-SGX," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, (New York, NY, USA), pp. 2333–2350, ACM, 2019.

[63] W. Qiang, Z. Dong, and H. Jin, "Se-lambda: Securing privacy-sensitive serverless applications using sgx enclave," in *Security and Privacy in Communication Networks* (R. Beyah, B. Chang, Y. Li, and S. Zhu, eds.), (Cham), pp. 451–470, Springer International Publishing, 2018.

[64] I. Chillotti, N. Gama, M. Georgieva, and M. Izabachène, "TFHE: Fast Fully Homomorphic Encryption Over the Torus," *Journal of Cryptology*, vol. 33, 04 2019.

[65] M. Carpen-Amarie, P. Marlier, P. Felber, and G. Thomas, "A Performance Study of Java Garbage Collectors on Multicore Architectures," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '15, (New York, NY, USA), p. 20–29, Association for Computing Machinery, 2015.

[66] N. D. Matsakis and F. S. Klock, "The Rust Language," *Ada Lett.*, vol. 34, p. 103–104, Oct. 2014.

[67] A. J. Bernstein, "Analysis of Programs for Parallel Processing," *IEEE Transactions on Electronic Computers*, vol. EC-15, pp. 757–763, Oct 1966.

[68] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, "RustBelt: Securing the Foundations of the Rust Programming Language," *Proc. ACM Program. Lang.*, vol. 2, pp. 66:1–66:34, Dec. 2017.

[69] The Chromium Projects, "Memory safety." [Online; accessed 24-May-2020] `https://www.chromium.org/Home/chromium-security/memory-safety`.

[70] X. Wang, H. Chen, A. Cheung, Z. Jia, N. Zeldovich, and M. F. Kaashoek, "Undefined Behavior: What Happened to My Code?," in *Proceedings of the Asia-Pacific Workshop on Systems*, APSYS '12, (New York, NY, USA), Association for Computing Machinery, 2012. Revision #2 found at `https://people.csail.mit.edu/nickolai/papers/wang-undef-2012-08-21.pdf` [Online; accessed 13-May-2020].

[71] B. Gregg, "Blazing Performance with Flame Graphs," (Washington, D.C.), USENIX Association, Nov. 2013.

[72] A. K. Lenstra, "Key Lengths. Contribution to The Handbook of Information Security," 2010.

[73] A. C. Yao, "Protocols for Secure Computations," in *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, (Washington, DC, USA), pp. 160–164, IEEE Computer Society, 1982.

[74] I. Ioannidis and A. Grama, "An efficient protocol for Yao's millionaires' problem," in *36th Annual Hawaii International Conference on System Sciences, 2003. Proceedings of the*, pp. 6 pp.–, 2003.

[75] H.-y. Lin and W.-g. Tzeng, "An efficient solution to the millionaires' problem based on homomorphic encryption," in *In ACNS 2005, volume 3531 of Lecture*, pp. 456–466, 2005.

[76] M. Jakobsson and M. Yung, "Proving Without Knowing: On Oblivious, Agnostic and Blindfolded Provers," in *Advances in Cryptology — CRYPTO '96* (N. Koblitz, ed.), (Berlin, Heidelberg), pp. 186–200, Springer Berlin Heidelberg, 1996.

[77] H. Chen, I. Chillotti, and Y. Song, "Multi-Key Homomophic Encryption from TFHE." Cryptology ePrint Archive, Report 2019/116, 2019. `https://eprint.iacr.org/2019/116`.

[78] Y. Dodis, S. Halevi, R. D. Rothblum, and D. Wichs, "Spooky Encryption and its Applications." Cryptology ePrint Archive, Report 2016/272, 2016. `https://eprint.iacr.org/2016/272`.

[79] M. Taassori, A. Shafiee, and R. Balasubramonian, "VAULT: Reducing Paging Overheads in SGX with Efficient Integrity Verification Structures," *ACM SIGPLAN Notices*, vol. 53, pp. 665–678, 03 2018.

[80] M. Karnaugh, "The map method for synthesis of combinational logic circuits," *Transactions of the American Institute of Electrical Engineers, Part*

I: Communication and Electronics, vol. 72, pp. 593–599, Nov 1953.

[81] E. J. McCluskey, "Minimization of Boolean functions," *The Bell System Technical Journal*, vol. 35, pp. 1417–1444, Nov 1956.

[82] P. Aubry, S. Carpov, and R. Sirdey, "Faster homomorphic encryption is not enough: improved heuristic for multiplicative depth minimization of Boolean circuits." Cryptology ePrint Archive, Report 2019/963, 2019. `https://eprint.iacr.org/2019/963`.

[83] T. Lynch and E. E. Swartzlander, "A spanning tree carry lookahead adder," *IEEE Transactions on Computers*, vol. 41, pp. 931–939, Aug 1992.

[84] N. Burgess, "Fast Ripple-Carry Adders in Standard-Cell CMOS VLSI," in *2011 IEEE 20th Symposium on Computer Arithmetic*, pp. 103–111, July 2011.

[85] M. Hähnel, W. Cui, and M. Peinado, "High-Resolution Side Channels for Untrusted Operating Systems," in *Proceedings of the 2017 USENIX Conference on Usenix Annual Technical Conference*, USENIX ATC '17, (USA), p. 299–312, USENIX Association, 2017.

[86] E. Brady, "IDRIS —: systems programming meets full dependent types," in *PLPV '11*, 2011.

[87] U. Norell, *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.