TECHNISCHE
UNIVERSITÄT
DRESDEN

Faculty of Computer Science Institute of Computer Engineering, Chair for Compiler Construction

Efficient Cloud-Based Privacy-Preserving Computation via Fully Homomorphic Encryption and Proxy Re-Encryption

Mirko Schäfer

mirko.schaefer@tu-dresden.de
Born on: 30.08.1995 in Sebnitz
Matriculation number: 4052462

Diploma Thesis

Supervisors
Lars Schütze, TU Dresden
Kilian Becher, SAP SE

Supervising professors
Prof. Dr.-Ing. Jerónimo Castrillón, TU Dresden
Prof. Dr.-Ing. Thorsten Strufe, KIT

Submitted on: 25.01.2022

Statement of authorship

I hereby certify that I have authored this document entitled *Efficient Cloud-Based Privacy-Preserving Computation via Fully Homomorphic Encryption and Proxy Re-Encryption* independently and without undue assistance from third parties. No other than the resources and references indicated in this document have been used. I have marked both literal and accordingly adopted quotations as such. There were no additional persons involved in the intellectual preparation of the present document. I am aware that violations of this declaration may lead to subsequent withdrawal of the academic degree.

Dresden, 25.01.2022

[illegible]

Mirko Schäfer

TECHNISCHE UNIVERSITÄT DRESDEN

cfaed CENTER FOR ADVANCING ELECTRONICS DRESDEN

TU Dresden Fakultät Informatik, Institut für Technische Informatik, Professur für Compilerbau

Task Description for Final Thesis (Diplomarbeit)

For:	**Mirko Schäfer**
Degree program:	Diplom Informatik (Studienordnung 2010)
Matriculation number:	4052462
E-mail:	mirko.schaefer@tu-dresden.de
Topic:	**Efficient cloud-based privacy-preserving computation via fully homomorphic encryption and proxy re-encryption**

Fully Homomorphic encryption (FHE) enables computation over encrypted data without intermediate decryption. This allows outsourcing of computations to untrusted parties, such as cloud environments, while preserving privacy of inputs, intermediate results, and outputs. Confidential inputs can be provided by multiple parties. Such scenarios may require the use of different encryption keys to ensure privacy of the different parties' inputs. Proxy re-encryption (PRE) is a technique that enables the privacy-preserving transformation of ciphertexts encrypted under one key into ciphertexts of the same plaintexts which are encrypted under a different key. However, PRE adds substantial computational overhead.

The main research question of this work is: How can PRE be utilized best to enable efficient cloud-based privacy-preserving computation using FHE such that inputs can be encrypted under different encryption keys? This includes an analysis of the effect that introducing PRE has on key management, benchmarking to identify sources of inefficiencies, and a study of graph optimizations to improve the overall efficiency of cloud-based privacy-preserving computations. This study is enabled by an initial DSL and will be built on its corresponding IR and compiler infrastructure.

This thesis addresses the following aspects:

- Design and implementation of efficient methods for combining PRE and FHE to enable cloud-based privacy-preserving computation with support for different encryption keys.
- Analysis of approaches to graph optimization such as parallelization and outsourcing of pre and post computations to the parties' client applications.
- Evaluation of the impact that introducing PRE has on key management and computation runtime.

Start:	24.08.2021
End:	25.01.2022
1st referee:	Prof. Dr.-Ing. Jerónimo Castrillón
2nd referee:	Prof. Dr.-Ing. Thorsten Strufe
Supervisor:	Lars Schütze, Kilian Becher (SAP SE)

[illegible]

Prof. Dr.-Ing. Jerónimo Castrillón
(Professor in charge)

Contents

List of Figures

List of Listings

List of Tables

List of Acronyms

AST	abstract syntax tree
BFV	Brakerski-Fan-Vercauteren
CPU	central processing unit
CST	concrete syntax tree
CKKS	Cheon-Kim-Kim-Song
DAG	directed acyclic graph
DSL	domain-specific language
EVA	encrypted vector arithmetics
FHE	fully homomorphic encryption
HE	homomorphic encryption
IDE	integrated development environment
IR	intermediate representation
LWE	learning with errors
MPC	multi-party computation
PHE	partially homomorphic encryption
PRE	proxy re-encryption
RLWE	ring learning with errors
SHE	somewhat homomorphic encryption
SIMD	single instruction, multiple data
SMC	secure multi-party computation
TTP	trusted third party

1. Introduction

1.1. Motivation

Nowadays, healthcare institutions collect a large amount of data about their patients like medical-treatment protocols and surgical reports. This patient data is highly confidential and strict data-protection requirements apply [20]. Consequently, sharing and aggregation of patient data with research institutions are very complicated due to regulatory reasons. Fully homomorphic encryption (FHE) combined with proxy re-encryption (PRE) can enable researchers to combine their patient-data records without sharing them in plaintext. FHE schemes are special encryption schemes that enable performing operations on encrypted data without intermediate decryption [22]. Proxy re-encryption (PRE) enables the transformation of ciphertexts encrypted under one key into ciphertexts of the same plaintexts but encrypted under a different key without intermediate decryption [6]. In combination, FHE and PRE enable computations on distributed data sets that are encrypted under different encryption keys [30]. Therefore, it allows analyzing distributed patient-data sets and gaining more significant insights. For example, with larger data sets, more accurate information about correlations between genome mutations and properties of tumors can be identified. That can enable researchers to develop more specific treatment methods against the tumors [37, 27].

Developing efficient FHE applications with PRE is challenging and requires cryptographic expertise. Compilers can help developers by automating tasks like parameterization and the efficient use of re-encryptions. A few compilers that aim to improve the development process of FHE programs have been proposed in the past. For FHE exist a variety of compilers that aims to improve the developer experience. However, compiler support for PRE is currently not present.

1.2. Objectives

This work aims to study methods of utilizing proxy re-encryption to enable cloud-based privacy-preserving computations using FHE such that inputs can be encrypted under different encryption keys. Therefore, the main objective of this thesis is to design and implement an extension of the HElium FHE compiler to support PRE. Due to the fact that single proxy re-encryptions can be expected to add a fair amount of computational overhead to FHE-based programs, the main focus of this extension is to enable the efficient use of PRE. Furthermore, the resulting compiler has to be evaluated to analyze the impact of PRE. This includes the effect of PRE on the execution runtime of FHE programs and the identification of opportunities for further optimization. Based on the evaluation results, this thesis will address further optimizations and point out future work.

1.3. Structure

This section provides a brief overview of the structure of this work. First, Chapter 2 presents required preliminary information about fully homomorphic encryption (FHE) and proxy-re-encryption (PRE). Thereafter, related work is discussed in Chapter 3. Chapter 4 elaborates the computation scenario for computations with FHE and PRE and derives requirements. Within the scope of the scenario, Chapter 5 presents design concepts of a PRE integration into the HElium compiler. Subsequently, the concepts are implemented in Chapter 6. The resulting implementation is then evaluated in Chapter 7. This evaluation provides valuable findings for the following elaboration of further optimization opportunities in Chapter 8. The findings and results of this work are concluded in Chapter 9.

2. Preliminaries

This chapter provides the required preliminaries for this work. This includes an introduction to fully homomorphic encryption (FHE), proxy re-encryption (PRE), and the HElium compiler.

2.1. Fully Homomorphic Encryption

Encryption schemes consist of three main components: a key-generation function $KeyGen(\cdot)$, an encryption function $E(\cdot)$, and a decryption function $D(\cdot)$ [33]. The (probabilistic) key generation function $KeyGen(\cdot)$ takes as input a security parameter λ and yields an encryption key, denoted by k_enc, as well as a decryption key, denoted by k_dec, as shown in Equation (2.1). The (probabilistic) encryption function $E(\cdot)$ takes as input the encryption key pk and the message, denoted by $m \in M$, that is to be encrypted, and yields a ciphertext $c \in C$ as depicted in Equation (2.2). M and C denote the plaintext space and ciphertext space, respectively. In contrast, the decryption function $D(\cdot)$ yields the plaintext message $m \in M$. It takes as input the decryption key sk and the ciphertext $c \in C$, like shown in Equation (2.3).

$$(k_dec, k_enc) \leftarrow KeyGen(\lambda) \tag{2.1}$$

$$c \leftarrow E(m, k_enc) \tag{2.2}$$

$$m \leftarrow D(c, k_dec) \tag{2.3}$$

In asymmetric encryption schemes, the encryption key k_enc is public. Therefore, it is typically referred to as public key pk. Whereas the decryption key is secret and therefore referred to as secret key sk [33]. In symmetric schemes, the same secret key sk is used for encryption and decryption. For simplicity, encryption and decryption can be denoted by $c = E(m)$ and $m = D(c)$, respectively.
Homomorphic Encryption (HE) is a special type of encryption scheme that allows computations on ciphertexts [22]. That means that performing an operation $\circledcirc$ on a ciphertext yields a ciphertext of the result of the corresponding homomorphic operation $\circ$ applied to the plaintexts, as shown in Equation (2.4) [1].

$$a \circ b = D(E(a) \circledcirc E(b)) \tag{2.4}$$

HE schemes can have different homomorphic properties and therefore support different operations on encrypted data. Four different types of HE schemes exist [1].

Homomorphic encryption schemes that have only one homomorphic property are referred to as partially homomorphic encryption (PHE) schemes. They support, e.g., either multiplicative or additive homomorphic operations. Somewhat homomorphic encryption (SHE) schemes support two types of operations, typically addition and multiplication. However, SHE schemes only allow a limited number of operations and a particular type of circuits. Leveled fully homomorphic encryption (LFHE) schemes also provide two homomorphic operations. These schemes can evaluate arbitrary circuits with bounded depth depending on certain parameters of the scheme. Fully homomorphic encryption (FHE) schemes can perform an arbitrary number of two different homomorphic operations, e.g., additions and multiplications.

The RSA scheme introduced by Rivest, Shamir, and Adleman in [31] is an example for multiplicative PHE schemes. In RSA, the multiplication of two ciphertexts yields a ciphertext of the multiplication of the corresponding plaintexts, as depicted in Equation (2.5).

$$a \cdot b = D(E(a) \cdot E(b)) \tag{2.5}$$

In contrast, the PHE cryptosystem introduced by Pallier in [28] is additively homomorphic. As shown in Equation (2.6), a multiplication of two ciphertexts yields a ciphertext of the sum.

$$a + b = D(E(a) \cdot E(b)) \tag{2.6}$$

In 2009, Gentry proposed the first FHE scheme [22]. In the following years, a variety of schemes were presented. Similar to Gentry's approach, modern FHE schemes follow a similar pattern: In these schemes, the public key is an element that cancels out to zero when combined with the corresponding secret key [35].

To ensure security, a small error is added to the message that is referred to as noise. If the noise is small enough, the initial message can be reconstructed correctly. Homomorphic operations cause noise growth. If the noise exceeds some limit, the message can not be reconstructed, i.e., the ciphertext cannot be decrypted correctly. Therefore, this construction only supports a limited number of operations. To overcome this problem, a function, called Bootstrapping, is introduced by Gentry [22]. This computationally complex operation reduces the noise of a ciphertext. In the Bootstrapping operation, the decryption function of the encryption scheme is translated into a circuit. The bootstrapping circuit is homomorphically evaluated given the encrypted secret key of the encryption system and the ciphertext. This yields fresh encryption of the plaintext. In this context, *fresh* means that the resulting ciphertext contains less noise than the input ciphertext of the Bootstrapping algorithm. Therefore, further operations can be applied to the fresh ciphertext. Consequently, such a scheme can evaluate an arbitrary number of computation gates. Thus, such construction is an FHE scheme built from a SHE scheme and a corresponding Bootstrapping operation.

However, Gentry's initial FHE scheme is not suitable for practical applications due to high computational complexity [23]. More recent FHE schemes are much more efficient. In [9], Brakerski, Gentry, and Vaikuntanathan presented a more recent SHE scheme based on the ring learning with errors (RLWE) problem. This and following schemes like BFV [21] and CKKS [12] improved the growth of noise during homomorphic operations. This allows performing larger circuits without Bootstrapping. Furthermore, they introduced batching, the packing of multiple plaintexts into one ciphertext [34]. Batching allows computations in a SIMD fashion, i.e., an operation is applied to multiple elements simultaneously. Therefore, this technique can improve the efficiency of a program.

2.1.1. Encryption Schemes and Implementations

Table 2.1.: Overview of FHE Libraries with Supported Schemes and Availability of Bootstrapping & Batching

Implementation Library	Scheme	Type of operations	Supports Bootstrap.	Supports Batching	PRE
HEAAN	CKKS	approx. arithmetic	yes	yes	no
HELib	CKKS	approx. arithmetic	no	yes	no
	BGV	arithmetic	yes	yes	no
SEAL	CKKS	approx. arithmetic	no	yes	no
	BFV	arithmetic	no	yes	no
Palisade	BFV	arithmetic	no	yes	yes
	BGV	arithmetic	no	yes	yes
	CKKS	arithmetic	no	yes	yes
	TFHE	Boolean	yes	no	no
TFHE	TFHE	Boolean	yes	no	no
Concrete	TFHE	Boolean & arithmetic	yes	yes	no

Since Gentry's seminal work [22], various encryption schemes were proposed [9, 12, 14]. These schemes can be divided into two groups. The first group is schemes derived from the BGV [9] scheme. They are optimized for fast arithmetic calculations. Recent schemes support techniques like Batching and provide highly efficient integer or fixed-point operations [8, 12]. Since they allow batching of ciphertexts, computations can have a high throughput which can be beneficial, for example, in machine-learning scenarios. However, the existing bootstrapping algorithms of these schemes have high computational complexity. Fortunately, these schemes can be used in a leveled mode, i.e., with encryption parameters that allow the necessary amount of computations without intermediate bootstrapping. The amount of levels depends on the complexity and the structure of the problem.

The second group is derived from the GSW scheme [24] and is optimized for boolean operations and fast bootstrapping. Due to the existence of efficient algorithms, bootstrapping can be applied after each operation. Therefore, the encryption parameters are independent of the depth and structure of the circuit. This allows computations of circuits of arbitrary depth. However, these schemes do not support batching, since it is incompatible with the fast bootstrapping approach [14]. As GSW-based schemes like TFHE are optimized for Boolean operations, i.e., they provide operations in $GF(2)$, higher-level integer operations must be constructed from Boolean operations.

Table 2.1 provides an overview of FHE libraries and their supported encryption schemes.

2.1.2. Computation Scenario

HE and especially FHE has a wide range of applications. One promising use case is the application of machine learning models. Let there be two participants: *Party A* holds a data set and *Party B* has a data model that provides further analysis of *A*s data. However, the data set of *Party A* may be confidential and the data model of *Party B* a trade secret. HE enables such computations. It allows to prevent the revealing of data of *Party A* and does not require direct access to the model of *Party B*. Figure 2.1 depicts such a computation of a function $f(\cdot)$ in an abstract form. It shows three parties: *Party*

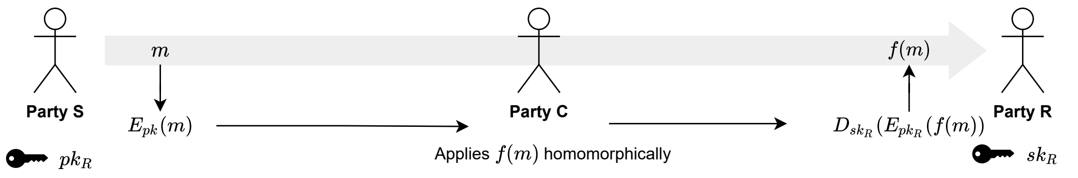

Figure 2.1.: Abstract FHE Scenario

S, *Party R*, and *Party C*. *Party A* represents the provider of data, i.e., the message *m*. It holds a public key pk_R to encrypt its message *m*. *Party B* is the receiving party. It holds the secret key sk_R, gets the computation results, and decrypts the ciphertext to retrieve the result of $f(m)$. *Party C* is placed between the others. It receives the encrypted message from *Party S*. Then, it performs the homomorphic computation of $f(m)$ without intermediate decryption and sends the resulting ciphertext to *Party R*. In practical applications, *Party S* and *Party R* can be represented by one participant. For example, this is the case for the previously presented application. In this use case, *Party A* acts as the sender of the message and receiver of the results too. Similarly, *Party C* represents the computation party.

2.1.3. The BFV Scheme

A promising FHE scheme is the Fan-Vercauteren variant [21] of Brakerski's scale-invariant scheme [9]. It is referred to as Brakerski-Fan-Vercauteren (BFV) scheme. BFV performs operations over polynomial rings and is based on the assumed hardness of the Ring Learning With Errors (RLWE) problem. Its security relies on adding a small error component (noise) during encryption. Taking the notation of [25], textbook BFV can be formalized as follows.

In the following section, operations modulo b are denoted by $[a]_b$. Vectors are highlighted by bold text if not stated otherwise. Furthermore, rounding down, up, and to the nearest integer is denoted by $\lfloor a \rfloor$,$\lceil a \rceil$, and $\lfloor a \rceil$, respectively. Sampling some a from a distribution D is denoted by $a \leftarrow D$.

Let $R = \mathbb{Z}[X]/\langle f(X) \rangle$ be a ring such that $f(X) \in \mathbb{Z}[X]$ is a monic irreducible polynomial of degree n, typically a cyclotomic polynomial. Arithmetic computations in R are performed modulo $f(X)$. The plaintext space is denoted by the quotient ring $R_t = R/tR$. It is referred to the integer $t \geq 2$ as the plaintext modulus. Plaintexts in R_t are polynomials in R with coefficients in $\mathbb{Z}_t$. Similarly, the ciphertext space is denoted by $R_q = R/qR$ with the ciphertext modulus $q \gg t$. Additionally, a decomposition base $w \in \mathbb{Z}$ is defined to represent polynomials in R_q by $l + 1$ polynomials in base w and require $l = \lfloor log_w q \rfloor$. Furthermore, let χ_e be a Gaussian distribution with expected value $\mu = 0$ and χ_k be a uniform distribution over $\{-1, 0, 1\}^n$. The BFV scheme consists of five procedures.

Key generation – $G(\kappa, w)$: Sample a low-norm secret $s \overset{U}{\leftarrow} \chi_k$ and set $sk = (1, s) \in R^2$. Randomly choose $a \overset{U}{\leftarrow} R_q$ and $e \overset{G}{\leftarrow} \chi_e$, compute $b = [-(as + e)]_q \in R_q$, and set $pk = (pk[0], pk[1]) = (b, a)$. Additionally, $G(\cdot)$ outputs a public evaluation key evk, referred to as relinearization gadget. It is a set of $l + 1$ pairs of polynomials and is computed for $0 \leq i \leq l$ as follows. First, sample $a_i \overset{U}{\leftarrow} R_q$ and noise $e_i \overset{G}{\leftarrow} \chi_e$. Then, set $evk[i] = ([w^i s^2 - (a_i \cdot sk + e_i)]_q, a_i)$. The output of $G(\kappa, w)$ is the tuple (sk, pk, evk).

Encryption – $E_{pk}(m)$: Sample $u \overset{U}{\leftarrow} R_2$ and $e_1, e_2 \overset{G}{\leftarrow} \chi_e$. The encryption of $m \in R_t$ is a pair $c = (c[0], c[1])$ of elements in R_q. It is computed as

$$c = ([\Delta m + pk[0]u + e_1]_q, [pk[1]u + e_2]_q)$$

such that $\Delta = \lfloor \frac{q}{t} \rfloor$.

Decryption – $D_{sk}(c)$: To decrypt c and obtain m, compute

$$m = \left[\left\lfloor \frac{t}{q} [c[0] + c[1]sk]_q \right\rceil\right]_t .$$

Homomorphic addition – $EvalAdd(c_1, c_2)$: Given two ciphertexts c_1, c_2, the encrypted sum $c_1 \oplus c_2$ of the underlying plaintexts is computed as

$$c_{add} = ([c_1[0] + c_2[0]]_q, [c_1[1] + c_2[1]]_q).$$

Homomorphic multiplication – $EvalMult_{evk}(c_1, c_2)$: Given two ciphertexts c_1, c_2, the encrypted product $c_1 \odot c_2$ of the underlying plaintexts is computed in two steps: tensoring and relinearization. For tensoring, compute

$$\begin{aligned}
c_{mult_0} &= \left[\left\lfloor \frac{t}{q} c_1[0] \cdot c_2[0] \right\rceil\right]_q , \\
c_{mult_1} &= \left[\left\lfloor \frac{t}{q} (c_1[0] \cdot c_2[1] + c_1[1] \cdot c_2[0]) \right\rceil\right]_q , \\
c_{mult_2} &= \left[\left\lfloor \frac{t}{q} c_1[1] \cdot c_2[1] \right\rceil\right]_q .
\end{aligned}$$

For relinearization, first decompose c_{mult_2} in base w such that $c_{mult_2} = \sum_{i=0}^{l} c_{mult_2}^{(i)} w^i$. Then, compute the result $c_{mult} = (c_{mult}[0], c_{mult}[1])$ as follows for $j \in \{0, 1\}$.

$$c_{mult}[j] = \left[c_{mult_j} + \sum_{i=0}^{l} evk[i][j] c_{mult_2}^{(i)} \right]_q$$

2.2. Proxy Re-Encryption

Re-encryption transforms a ciphertext $c_1 = E_{pk_1}(m)$ encrypted under a key pk_1 into a ciphertext $c_2 = E_{pk_2}(m)$ of the same plaintext, encrypted under a different key pk_2. Proxy re-encryption (PRE) allows an untrusted party to perform this transformation

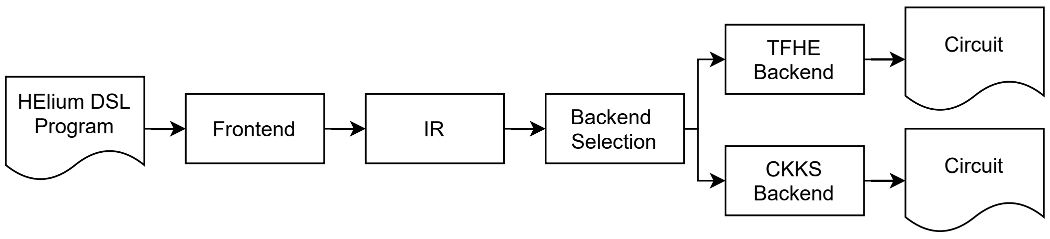

Figure 2.2.: Compiler Pipeline of HElium

without affecting confidentiality [6]. A standard construction to obtain a PRE scheme from an FHE scheme is described in [22].

Following the notation of [30], a PRE scheme is defined as a tuple $\mathcal{PRE} = (PG, KG, ReGK, E, D, RE)$ of six procedures. Parameter generation $PG(\cdot)$ computes a set of public parameters related to the security parameter λ. The key generation algorithm $KG(\cdot)$ outputs a key pair (pk, sk). Re-encryption-key generation $ReKG(\cdot)$ takes a secret key sk_i and a public key $pk_{j \neq i}$ and computes a re-encryption key $rk_{i \to j}$. The re-encryption algorithm $RE(\cdot)$ transforms a ciphertext c_i of m encrypted under pk_i into a ciphertext c_j of m such that c_j encrypts m under $pk_{j \neq i}$. $E(\cdot)$ and $D(\cdot)$ denote encryption and decryption algorithms, respectively.

2.3. HElium Compiler

The HElium compiler is a compiler prototype for FHE. It compiles programs that are written in an external DSL into computation graphs that are compatible to FHE implementations. Its main contribution is an automatic encryption scheme selection. Figure 2.2 depicts the internal compiler stages of HElium. First the program is parsed by the frontend and an abstract syntax tree (AST) is built. Thereafter, the AST is converted to an intermediate representation (IR). A backend-selection stage analyzes the IR and selects a suitable backend for further compilation. Helium comprises two different backends: a backend targeting Boolean circuits and an arithmetic backend. The following subsections provide further information about the stages of the HElium compiler.

2.3.1. HElium Domain-Specific Language

The HElium compiler provides its own domain-specific language (DSL). The type system of HElium consists of three base types: signed integer, unsigned integer, and floating-point numbers. These base types are configurable, i.e., their size could be adjusted to the developer's needs. By using these type parameters other types could be constructed. For example, a Boolean type could be represented through an unsigned integer of length 1, i.e., *int<1>*. The type system further differentiates between encrypted and plaintext data. Each variable is defined as encrypted per default if it is not annotated with the qualifier *plain*. This behavior ensures security by default.
HElium provides different types of operations: arithmetic operations, vector-specific operations, Boolean operations, and comparison operations. These operations are expressed by symbols. For a complete list of operations see Appendix A.1. All operations are operating on vectors, i.e., they perform element-wise operations. Scalar operands are treated as single-element vectors. In addition to operations, HElium

provides built-in functions. For instance, the method *size()* returns the size of a vector.
Due to the properties of HE, control-flow decisions on encrypted conditions cannot be made directly without access to the corresponding decryption key. HElium emulates *if-then-else* constructs and *for*-loops trough *MUX* operations. Furthermore, it supports *for*-loops that operate like a *for-each*-loop in other programming languages. It represents the iteration over the elements of a vector.

2.3.2. Intermediate Representation

HElium implements a term-based intermediate representation (IR). A term represents a node in the computation graph, i.e., an operation on a ciphertext or a plaintext. Depending on the type of operation, each term has a certain number of operands. There are unary terms that are taking one operand as input and binary terms require two operands. Input terms and output terms are special terms with none or one operand, respectively. Constant values are represented by a constant term without an operand. Each operand is a term as well. Together the terms form a computation graph.

Figure 2.3 shows the IR of a program which implements Equation (2.7). The nodes 0 to 5 are terms. The terms 0 and 1 represent the arguments a_0 and x. Both are operands of the multiplication term 3. Term 2 represents a constant of value 5.0. The constant term and the multiplication result are added by term 4. The result of the addition is provided to term 5 which represents the output y of the computation.

$$y = a1 \cdot x + 5.0 \tag{2.7}$$

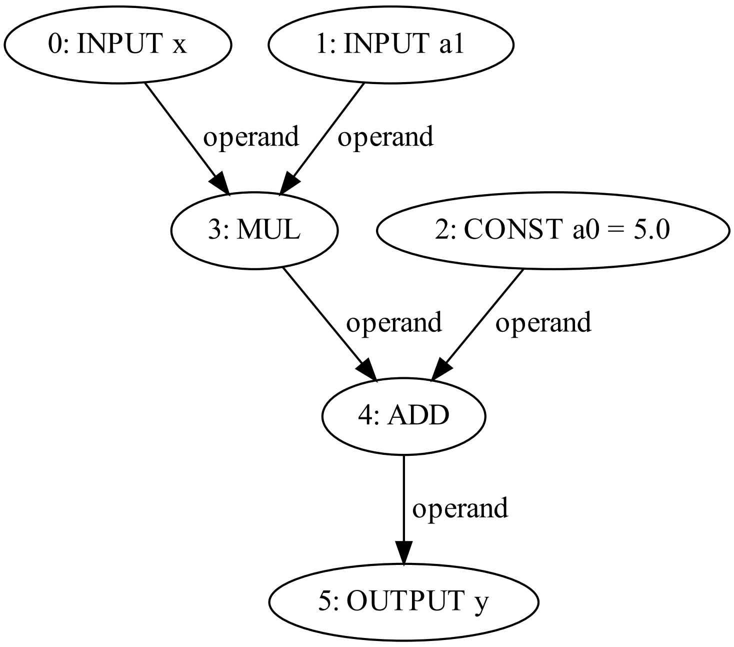

Figure 2.3.: IR Graph Representation

An IR term refers to an operation on encrypted or unencrypted data. A complete list of operations including descriptions can be found in Appendix A.1.

2.3.3. Type System

Each term of HElium has a type assigned to. HElium supports two adjustable base types: n-bit integer and fixed-point numbers. On top of these, it provides an array data type to represent array structures of the base types. Custom types can be constructed by adjusting the base types. For example, a boolean type can be represented by a one-bit integer. This type system is used in the abstract syntax tree (AST) of HElium and in the IR.
For *INPUTS* and *CONST* terms, the type is set during transformation from the AST. The type of other terms is inferred from the operation the term represents and from its operands.

2.3.4. Backends

The HElium compiler is designed to be scheme-independent. It provides backends for different encryption schemes like CKKS, and TFHE. Its design allows extension with backends for further encryption schemes. The TFHE backend of HElium focuses on compiling functions to efficient boolean circuits. Therefore, it utilizes common tools from the subject of hardware design. The arithmetic backend utilizes the low-level compiler EVA for the CKKS scheme. Hence, the backend transforms the IR into the IR of EVA and serializes the resulting optimized circuit. HElium decides automatically which backend is most suitable for a provided program and compiles the program with the selected backend.

3. Related Work

This chapter addresses related work. This comprises libraries that implement fully homomorphic encryption (FHE) schemes as well as compilers that are optimized for secure computation based on FHE.

3.1. Libraries for Fully Homomorphic Encryption

There exist a variety of open-source libraries that implement one or multiple FHE schemes. One of the first libraries is HElib.[1] It was initially developted at IBM by Gentry, Shoup, and Halevi. Now it is developed by an open-source community. HElib provides implementations of the BGV and the CKKS scheme. Furthermore, it implements Bootstrapping for BGV. The SEAL library is developed at Microsoft Research.[2] It implements the BFV and CKKS scheme. Furthermore, it provides support for various platforms like Android, iOS, and embedded systems. The Lattigo library implements the CKKS and the BFV scheme and is written in Go.[3] Similar to Palisade, Lattigo provides threshold variants of the implemented schemes. The TFHE library implements a variant of the GSW scheme.[4] TFHE is optimized for binary operations and fast bootstrapping. The concrete library developed by Zama implements a further improved version.[5] Concrete supports programmable bootstrapping. This technique allows the application of unary functions during bootstrapping via lookup tables. It is a promising technology to enable efficient computation of non-linear functions like square roots or activation functions like ReLU. The ability to compute non-linear functions is a big advantage for machine learning use cases. However, the concrete library does not provide PRE support.

There are further libraries that do not directly implement an FHE scheme. The Intel HEXL library uses 512-bit Advanced Vector Extension (AVX) of Intel processors to provide efficient implementations of integer arithmetic.

[1]https://github.com/homenc/HElib
[2]https://github.com/microsoft/seal/
[3]https://github.com/ldsec/lattigo
[4]https://tfhe.github.io/tfhe/
[5]https://github.com/zama-ai/concrete/

3.2. Compilers for Fully Homomorphic Encryption

In addition to libraries, a variety of compilers for FHE exist. However, none of the public available compilers for FHE has support for PRE builtin. This section gives a brief overview of the available compiler for FHE. See [35] for a more detailed comparison of FHE compiler. Alchemy is a compiler built on top of Haskel [16].[6] It supports a set of operations to define arithmetic functions. Alchemy uses its own implementation of the BFV scheme. The integrated compiler automatically inserts maintenance operations into the program and selects suitable encryption parameters. However, Alchemy is limited to basic arithmetic operations.

There exist a variety of source-to-source compilers that transforms C++ code into FHE programs. For example, Cingulata compiles C++ programs to fully homomorphic encryption (FHE) programs. [7] The resulting programs operate on Boolean circuits and support the TFHE library and a BFV implementation as backends [10]. Similar to HElium, Cingulata uses tools from hardware synthesis to optimize generated boolean circuits. Another tool that compiles C++ code into an FHE program is Encrypt-Everything-Everywhere (E3).[8] E3s input language supports both arithmetic and boolean circuits in BFV, BGV, and TFHE [13]. Furthermore, it provides basic support for SIMD operations of BFV and BGV. However, the expressibility of E3 is limited since it does not support rotation operations. Furthermore, maintenance operations are inserted naively, and encryption parameters must be chosen manually. Marble is an integrated DSL for FHE based on C++ [36].[9] Marble's central entrypoint is the proxy type *M* which represents a ciphertext message. Marble tracks all operations performed with objects of *M* via operator overloading and constructs a computation graph from them. It supports the construction of arithmetic and boolean circuits using the BFV scheme. HE Transpiler is a library that converts C++ code into FHE programs. It is maintained by Google.[10] The HE Transpiler uses Google's XLS library to compile C++ code into boolean circuits. For the execution of the boolean circuits, it provides runtimes based on TFHE or Palisade.

There are specialized compilers for FHE with a strong focus on machine learning applications and tensor operations. EVA is a low-level compiler for vector-arithmetic operations [19, 18]. It is a C++ library with a python interface.[11] EVA is mainly designed for arithmetic functions and vector operations. It targets the CKKS implementation of SEAL. The main contributions of EVA are efficient insertion of maintenance operations to the computation graph and automatic parameter selection. The Compiler and Runtime for Homomorphic Evaluation of Tensor Programs (CHET) project is built on top of EVA and focuses on optimizations for matrix operations. It comes with a high-level language for machine-learning operations. NGraph-HE is an extension of Intel's nGraph machine-learning compiler [7]. It is focused on the inference of machine learning models over encrypted data.[12] NGraph-HE translates Tensorflow computations into FHE circuits for BFV or CKKS. It applies optimizations on the computation graph and supports SIMD-packing, i.e., the efficient packing of multi-dimensional tensors into batched ciphertexts. Furthermore, nGraph-HE supports non-polynomial functions by implementing interactive protocols which compute these functions with the client.

[6]https://github.com/cpeikert/ALCHEMY
[7]https://github.com/CEA-LIST/Cingulata
[8]https://github.com/momalab/e3
[9]https://github.com/MarbleHE/Marble
[10]https://github.com/google/fully-homomorphic-encryption
[11]https://github.com/microsoft/EVA
[12]https://github.com/IntelAI/he-transformer

4. Scenario, Use Cases and Requirements

This chapter gives an introduction to the scenario of privacy-preserving computation via homomorphic encryption combined with proxy re-encryption (PRE). Based on the scenario, Section 4.3 derives requirements for compilers with support for PRE.

4.1. Scenario

Digitalization of processes and evolving capabilities of data analysis results in novel problems about privacy and confidentiality. Homomorphic encryption (HE) is considered to be a promising approach to overcoming some challenges. This section provides an overview of the abstract computation scenario.

Computation using standard FHE has some major limitations. For example, for the result to be encrypted under a common key, the inputs also need to be encrypted under the same key. That makes it challenging for scenarios with more than one data-providing participant. All participants have to agree on a key pair before encryption. Therefore, this model is not well-suited for frequently changing participant sets. There exist approaches like multi-key FHE and threshold FHE that address computations with multiple participants. However, multi-key FHE requires a pre-determined set of encryption keys, i.e., a relatively constant set of participants [26]. Threshold FHE allows distributing operations like the decryption of ciphertexts to a group of participants [3]. Nonetheless, threshold FHE requires active communication of the participants.

PRE can be a flexible solution for these problems. It allows the transformation of ciphertexts that are encrypted under one key to a ciphertext that is encrypted under another key without intermediate decryption. To perform this transformation, a re-encryption key is needed. Such a key can be generated from the source secret key and the target public key. For further information on PRE see Section 2.2.

Figure 4.1 depicts an abstract scenario of a PRE program. It comprises three types of participants: the senders $\{S_1, S_2, \ldots, S_i\}$, the receiving party R, and a computation party in between. The sender parties have data that is encoded in messages $m_1, m_2, \ldots, m_i$. They want to compute a function $f(m_1, m_2, \ldots, m_i)$ by joining their messages. The result gets the receiving party R. All senders and the receiver have own key pairs to encrypt and decrypt messages. The computation party in between enables the joint computation of the function and acts as a re-encryption proxy. It re-encrypts the ciphertexts using corresponding re-encryption keys and homomorphically computes the function $f(m_1, m_2, \ldots, m_i)$. This construction allows use cases

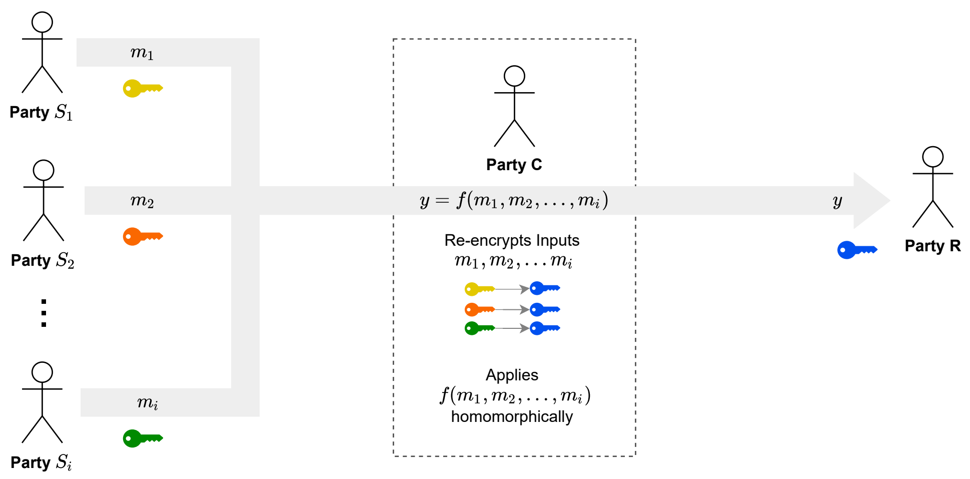

Figure 4.1.: Abstract FHE Scenario with PRE

with frequently-changing participants since new participants only have to provide a corresponding re-encryption key. Furthermore, it enables asynchronicity between encryption and usage of the data. Since data can be encrypted under participant-specific keys, computations can use data that is encrypted at different times. The receiving party or the computation itself does not need to be determined at the time of encryption. Therefore, PRE adds flexibility with regard to key management and the participant setup to FHE programs.

Figure 4.2 shows an instance of the abstract scenario with two senders. The sending parties and *Party R* have key pairs (sk_{S_1}, pk_{S_1}),(sk_{S_2}, pk_{S_2}) and (sk_R, pk_R), respectively. Similar to Figure 2.1, the parties S_1 and S_2 have messages m_1 and m_2. Similarly, *Party C* acts as re-encryption proxy and computation party that computes a function $f(m_1, m_2)$ homomorphically over the two messages. Therefore, *Party C* holds two re-encryption keys allowing to re-encrypt ciphertexts of *Party* S_1 and *Party* S_2 for *Party R*.

4.2. Key Management for Programs using Proxy-Re-Encryption

This section addresses the key management in computations with proxy re-ecryption (PRE). Before elaborating on concepts of key management for multiple encryption keys, the scenario of single encryption can be analyzed. In scenarios with a single key, i.e. without PRE, typically a single party generates a key pair (sk, pk) in cases where an asymmetric encryption scheme is used. Other parties can encrypt their data using the public key pk. Therefore, the key-generating party has to distribute its public key to other participants.

In contrast with PRE, participants can have their own key pair. Each participant generates its own key pair (sk_i, pk_i). Therefore, participants are immediately able to encrypt data using their encryption key pk_i. Furthermore, participants can generate multiple key pairs over time. For example, separate keys can be used for different chunks of data or to implement key-rotation mechanisms. As for other types of en-

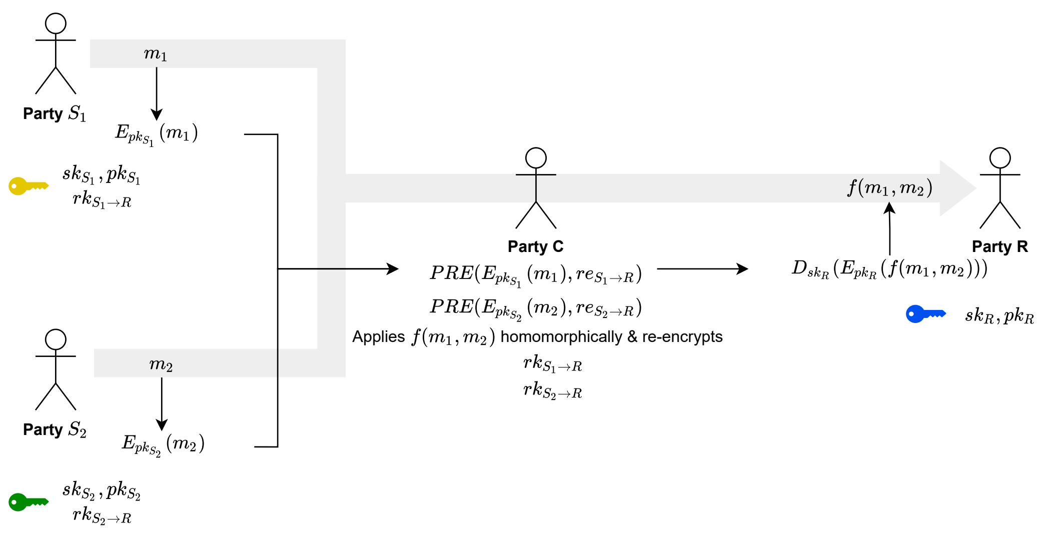

Figure 4.2.: Abstract FHE Scenario with PRE

cryption, it is important to mention that individual keys of a party must be stored properly to allow later access.

From the perspective of the participant, there are different ways to provide its data to an FHE computation.

- The participant can provide the raw data itself via a secure communication channel. However, this requires that the participant has still direct access to the data. Furthermore, it gives control over its data completely to the receiving party.
- The Participant can provide the secret key of the encrypted data directly to the receiving participant. It does not require the participant to have access to its data. However, the control over its data is still passed to the receiving party.
- The Participant can provide the secret key of the encrypted data to a trusted third-party that performs the computation. This moves the trust assumptions from the receiving party to the computation party. Furthermore, it can be difficult to agree on such a trusted third party.
- The participant can provide a re-encryption tag to a third party that allows performing re-encryption to the data from the participant's key to the key of the receiving party. The intermediate party does not have access to the data.

Figure 4.3 depicts a sequence diagram of computation using FHE and PRE. It is a generalized variant of the procedure proposed by Polyakov [30]. The figure comprises two participants *Alice* and *Bob*, and a computation party with a storage. The storage can be implemented in various ways. For example, distributed ledgers can be used as storage [5]. *Alice* and *Bob* want to compute a function $f(m)$ on a message m that is provided by *Bob*. *Bob* encrypts its message m with its own encryption key pk_{Bob}. This can take place completely asynchronous with no time related to the computation. When *Alice* wants to start a computation it sends Bob its public key pk_{Alice}. If *Bob* agrees on the computation it generates the re-encryption key $rk_{Bob \to Alice}$ and sends it to the computation party. This allows the computation party the transformation of the ciphertext and to start with the computation. The re-encryption key does not provide direct access to the data. After the computation finished, the function result can be decrypted by *Alice*. However, it must be assumed that the computation party does not colude with *Alice*. Otherwise, it could retrieve the message by re-encrypting *c* and decrypting it with the private key of *Alice*.

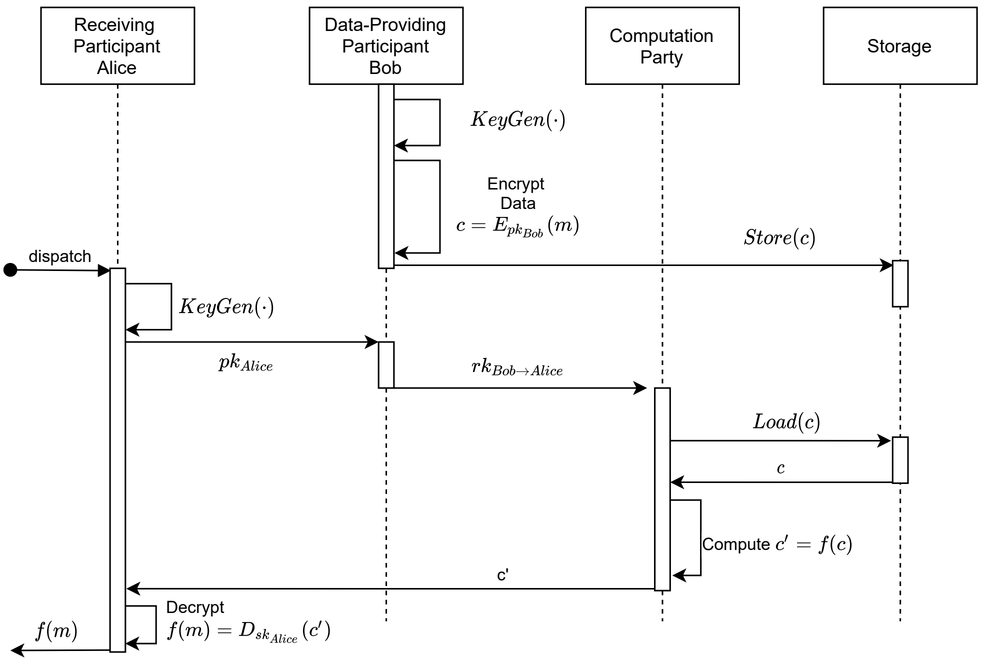

Figure 4.3.: Example Sequence of a Computation with Proxy Re-Encryption

4.3. Requirements for Compilers for Proxy Re-Encryption

This section elaborates requirements on compilers for FHE to support PRE and the previously defined computation scenario. Since this work extends the HElium compiler, these requirements are defined in the context of the available functionalities of HElium.

HElium has its own general-purpose DSL to describe FHE programs. According to the scenario description, the DSL has to model the computation between multiple participants. Each participant can provide inputs to the computation. Participants can provide their inputs under a common key or their own key. Therefore, the DSL must be able to map inputs to participants and to the encryption key that is used to encrypt data.

In addition to a multi-party-enabled frontend, the compiler needs a backend that supports PRE. This backend has to target an encryption scheme that supports PRE. Furthermore, it needs a scheme implementation that provides operations to generate re-encryption keys and provides operations to re-encrypt ciphertexts. In addition, the compiler needs to provide the targeted scheme with the information that is necessary for the execution. First, it is necessary to define at which point of the computation a proxy re-encryption takes place. Therefore, the circuit representation must be extended with a PRE operation. Second, the parameter selection of the backend must be modified. Since PRE operations add additional noise to the ciphertexts, parameters need to be adjusted to ensure correct execution.

Security and confidentiality are the main intentions to use FHE and PRE. According to the scenario, the inputs of the computations are confidential. However, the computation function can be publicly known by all participants. The computation party should not learn anything about the inputs besides their data structure. However, the compiler can assume that the computation party is performing the computation correctly. For example, the computation party can be a cloud provider which charges the participants for the computation resources. Such a cloud provider can lose its reputation if it actively violates the computation protocol. Consequently, a cloud provider has a strong intention to compute the function correctly. Furthermore, there are mechanisms to prove empirically that the computation party performs the intended computation. For example, the participants can randomly prove the computations by inserting test data without the knowledge of the computation party. However, the use of FHE prevents the leakage of input data or results to attackers who listen passively. The compiler has a direct impact on the security of the used FHE scheme. The security of FHE encryption depends on various scheme parameters that are selected by the compiler automatically. The parameters must be selected automatically according to the suggestions of the HE Standard [2].

Depending on the use case, different requirements can apply to the performance of the compiler and its results. On the one hand, performance can relate to different metrics like runtime or memory usage and can contradict each other. For example, applying more complex optimizations during compilation can reduce the runtime of the computation result while increasing the compilation time. On the other hand, the requirements of the actual use case on interactivity or responsiveness impact the compiler requirements as well. Using FHE in a real-time application can have much higher requirements on the execution runtime than a benchmarking solution.

5. Concept of Proxy Re-Encryption Compiler Support

This chapter elaborates on concepts for integrating proxy re-encryption (PRE) into HElium, an existing compiler for FHE. It describes necessary additions to HElium in order to support computations with multiple encryption keys via PRE. Section 5.1 discusses concepts to integrate mechanisms for PRE into HElium's DSL. Based on this, Section 5.2 derives requirements on the type system. Section 5.3 and 4.2 discuss challenges related to the PRE introduction and provide approaches for optimization.

5.1. Domain-Specific-Language Support for Proxy-Re-Encryption

This section derives requirements of PRE on domain-specific languages (DSL). First, on DSLs with support of PRE apply similar data flow restrictions and implications on the type system as for general FHE DSLs. The main difference is that PRE-enabled FHE schemes provide additional operations that allow switching the encryption key of ciphertexts. Consequently, such DSLs can inherit operations and properties of DSLs without PRE support. The additional functionalities of PRE-enabled FHE schemes allow computation on inputs provided by different participants and encrypted under different keys. These operations can be represented either directly or indirectly in the DSL. In DSLs with indirect integration, PRE operations are not part of the DSL. The re-encryption is indirectly determined by the keys of the inputs and outputs. Therefore, developers do not need to determine the insertion location of PRE operation themselves. This lowers the programming complexity for developers.

As presented in the previous section, encryption key pairs can be represented by labels. The DSL of HElium models inputs and outputs of FHE programs via input and output statements, respectively. With regard to PRE, the input statements of HElium can be extended with key labels.

Listing 5.1: Input and Output Definition

```
input a: int<5> @Key1 <= Party1;
input b: int<6> @Key2 <= Party2;
input c: int<1> <= Party3;
output y = (a + b) * c;
```

Listing 5.1 shows an example program in HElium that contains an input and an output definition. In line 1, the input *a* of type *int<5>* is defined. The source identifier shows that this input is provided by *Party1* and encrypted using the key pair *Key1*. Similarly, line 2 contains an input definition with name *b* of type *int<6>* provided by *Party2* and encrypted under *Key2*. In line 3, the input *c* is defined without a key label. If the key label is omitted, the compiler automatically applies a default key label. Line 4 shows an output definition of *y* to which the result of $(a + b) \cdot c$ is assigned. It comes without a key declaration. According to the use-cases defined in Chapter 4, the results of the FHE computations are encrypted under one key. Therefore, the compiler applies the default key labels on output definitions. The data type of outputs is determined by the compiler, too.

The mapping of an input or output definition to its source or target party, respectively, has no technical function for the compiler. However, these mappings can be useful at runtime. For example, they can be the basement for external authorization by the execution runtime. Input-providing, as well as output-retrieving, parties can be identified by the participant identifiers of the HElium program. Furthermore, it may make the program code more readable and can help users and developers to understand data dependencies between different parties.

5.2. Multi-Key Type System

Strong type systems can provide helpful information for compilers and their optimizations. In general-purpose programming languages, commonly used data types like boolean values, signed and unsigned integers, real numbers, and string types exist. On a higher layer, there can be more complex structures like arrays, matrices, and objects. Programs using FHE can add more layers of complexity to the type system. For example, in FHE programs, the type system must distinguish between *encrypted* and plaintext data. Additionally, the encoding of data can be different.

This section discusses an approach to extend the type system of the FHE-compiler HElium to support multiple encryption keys via PRE. In such scenarios, inputs of programs can be provided to the computation under different encryption keys. Listing 5.2 shows a slightly modified version of the previously presented example program. Similar to Listing 5.1, it represents the computation of $y = (a+b) \cdot c$. Whereas, *a* and *b* are provided by *Party1* and encrypted under *Key*1. Input *c* is provided by *Party2* under *Key*2. All inputs are of type *int<8>*, i.e., eight-bit integers.

Listing 5.2: Example Program with Three Inputs and Two Keys

```
input a: int<8> @ Key1 <= Party1;
input b: int<8> @ Key1 <= Party1;
input c: int<8> @ Key2 <= Party2;
output y = (a + b) * c;
```

When compiling this example with HElium, the compiler transforms the DSL code into a corresponding IR form. The nodes of the IR represent operations, inputs, outputs or constant values. Dependent nodes are connected by edges. In programs with multiple encryption keys, the inputs can not be described only by types and names. It needs an additional "key" property to indicate under which key an input is encrypted. For the inputs *a* and *b* this property is *Key*1, for input *c* it is *Key*2, respectively. One approach to integrating such properties to the IR is the introduction into the type system of HElium.

Figure 5.3 depicts a computation graph representing the function $y = (a + b) * c$. The graph consists of six nodes. Three nodes represent the inputs a, b,c, and d, encrypted under two different keys *Key 1* and *Key 2*. The computation result y can be decrypted with *Key 3*. The intermediate nodes 4, 5, 6 and 7 represent the operations *addition* and *multiplication*. Different colors of the nodes stand for the key under which an input or an output is encrypted. Each input and output node has a type and label that identifies the corresponding encryption key. At this point, the key labels of intermediate nodes are undetermined.

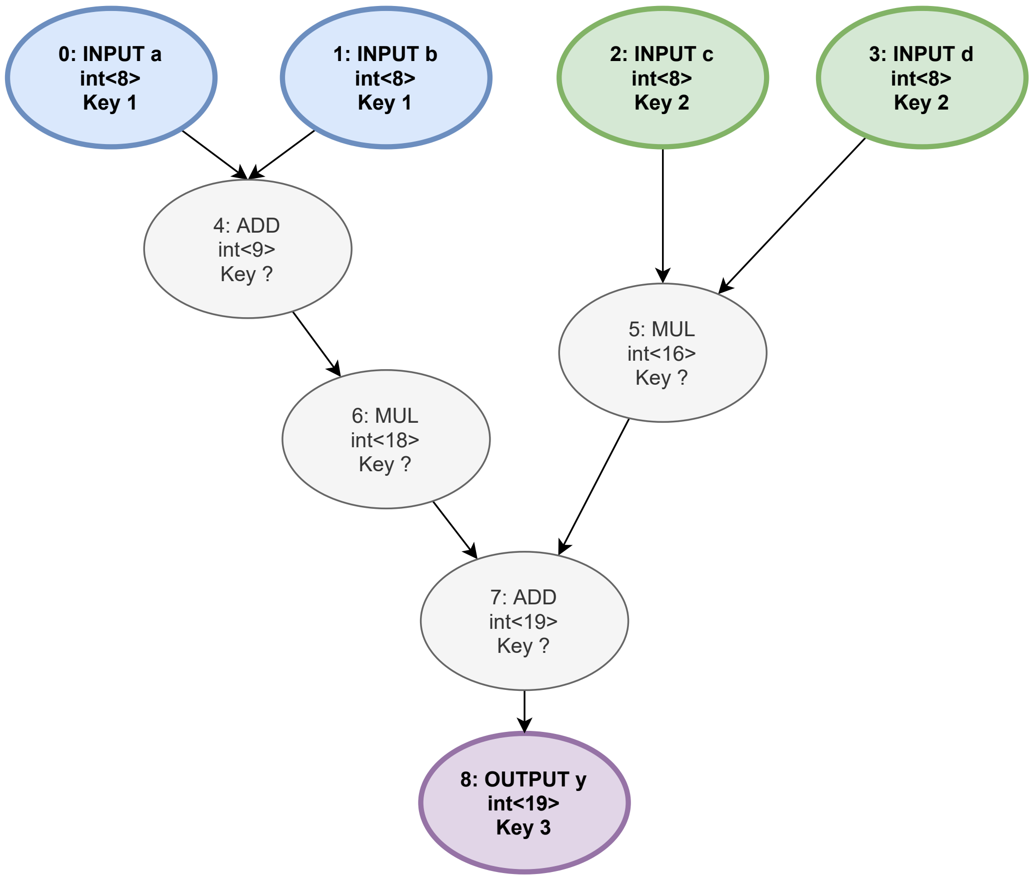

Figure 5.1.: Example Computation Graph with Inputs under Different Encryption Keys

Execution of the example program shown in Figure 5.3 starts at the input nodes. Since the encrypted value of the operands of node 4 is already present, the execution proceeds with node 4. The operands of node 4 are the inputs a and b. Since both are encrypted under *Key1*, the addition can take place without further adjustments and the encrypted value of node 4 can be computed. Thereafter, the execution can proceed with node 6 that represents a "square" operation. At the same time, the encrypted values of all operands of node 5 are present. Both shares the same key *Key 2*. Therefore, the multiplication operation yields a ciphertext that is encrypted under *Key 2*. At this point, the encrypted values of all operands of node 7 are present. However, the values of its operands are encrypted under different keys. While the value of node 6 is encrypted under *Key 1*, the value of node 5 is encrypted under *Key 2*. As stated in Section 2.2, homomorphic operations on ciphertexts require operands that are encrypted under the same key. Operations on ciphertexts encrypted under different keys are not possible without further adjustments. PRE allows transforming a ciphertext encrypted under one key to a ciphertext that is encrypted under another key. In the case of the example program, the values of node 4 and node 3 can be re-encrypted to a common key by performing a PRE operation. Consequently, the introduction of PRE operations can enable the execution of node 5. Hence, PRE is one approach to enable computation with inputs that are encrypted under different keys.

The example program has a small computation graph with only six nodes. For larger computation graphs it becomes challenging to determine where a PRE operation must be inserted. Compilers can automate this process. For this, it is crucial to determine the encryption keys of intermediate results of the computation. Similar to the example shown in Figure 5.3, encryption keys can be represented by key labels identifiers or key labels. Each key label maps uniquely to a key. For the compiler, it is sufficient to use only key labels. Further information like the party who issued the key is unnecessary. Therefore, a mapping from key identifiers to parties is not part of the compiler. One approach of integration of key labels into the IR is the direct storage within the type system of the IR. Figure 5.2 shows an abstract base type class. It holds three main information: the encryption state *encrypted* and the corresponding key label represented by *keyPair*.

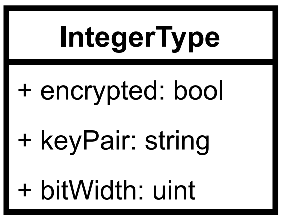

Figure 5.2.: Example Integer Type

The additional type information can be used to solve the original challenge: the determination of key labels of intermediate nodes of the computation graph. The key-label information that is provided with the inputs can be propagated through the computation graph. This process can be assumed to be similar to the type deduction of the compiler. The following section focuses on concepts to determine key labels efficiently.

5.3. Optimization of Proxy Re-Encryption Programs

The advantages of proxy re-encryption come with the cost of additional complexity. In the BFV scheme (see Section 2.1.3), the PRE operation has a similar computational complexity as homomorphic multiplications [30]. Furthermore, PRE operations can introduce additional noise to the ciphertext. Consequently, larger encryption parameters could be necessary to ensure correct computation. The execution performance typically decreases with larger scheme parameters. Therefore, it is important to perform only as few PRE operations as necessary for the execution.

This problem can be reduced to the problem of inserting PRE operation at the right point in the computation graph. As mentioned previously, a PRE operation is necessary where the key-pair label of the IR term differs from the key-pair labels of

the operands.
In terms of key labels, a computation graph can consist of nodes of two types: nodes with an explicitly defined key-par label, for example, inputs and outputs, and intermediate nodes. The key-pair of the intermediate nodes is not defined.

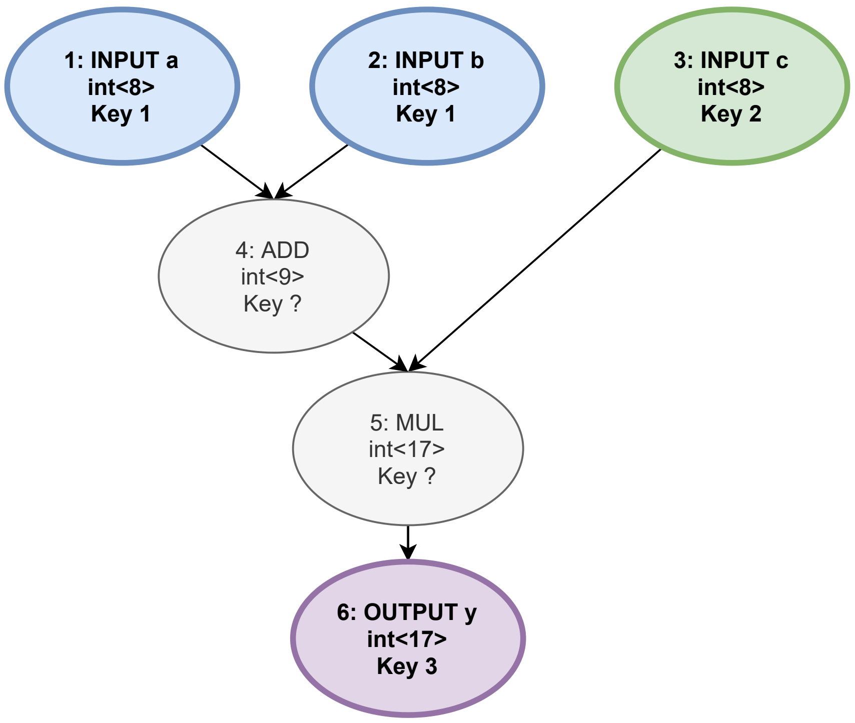

Figure 5.3.: Example Computation Graph with Unknown Key Labels

Figure 5.3 shows a computation graph with six nodes. The key-pair identifiers of the three inputs 1, 2, and 3, as well as, of the output node 6 are defined. However, there is no explicit definition for nodes 4 and 5.
To insert PRE operations properly, the compiler must determine sufficient key-pair labels for node 4 and 5. From a theoretical perspective, these can use either *Key1*, *Key2* or *Key3*. Consequently, there are $3^2 = 9$ possible variants of computation graphs with a different amount of necessary PRE operations and a different depth. These nine variants and their additional inserted PRE operations are presented in Table 5.1. The table lists variants of key selection for the two nodes including the number of inserted PRE operations p, the depth d of the resulting circuit, and a list of necessary re-encryption keys rk. It shows that the first and third variants require the lowest number of PRE operations and have the lowest depth. Therefore, it can be assumed that these variants are most efficient and have the lowest execution runtime.
An evaluation of all possible key-selection variants allows finding the most efficient one. However, their number grows polynomial in the number of keys and exponential in the number of nodes. Consequently, this method is unfeasible for larger computation graphs with many keys.
In practical scenarios, the set of allowed transformations between different keys is limited. Each key transformation from a key k_i to k_j requires a re-encryption key $rk_{i \to j}$. In the presented scenario, the input-providing parties do not interact with each other. Therefore, only re-encryption keys for PRE operations from input keys to output keys are available. In the presented example, this means that the set of re-encryption keys is limited to $\{rk_{1\to3}, rk_{2\to3}\}$. Consequently, the list of key-selection variants depicted

Table 5.1.: Comparison of Selection of Different Key Pairs

#	Node 4	Node 5	PRE Ops between Nodes	p	d	rk
1	Key1	Key1	$(3 \rightarrow 5), (5 \rightarrow 6)$	2	5	$rk_{2\rightarrow 1}, rk_{1\rightarrow 3}$
2	Key1	Key2	$(4 \rightarrow 5), (5 \rightarrow 6)$	2	6	$rk_{1\rightarrow 2}, rk_{2\rightarrow 3}$
3	Key1	Key3	$(4 \rightarrow 5), (3 \rightarrow 5)$	2	5	$rk_{1\rightarrow 3}, rk_{2\rightarrow 3}$
4	Key2	Key1	$(1 \rightarrow 4), (2 \rightarrow 4), (4 \rightarrow 5), (3 \rightarrow 5), (5 \rightarrow 6)$	5	7	$rk_{1\rightarrow 2}, rk_{2\rightarrow 1}, rk_{2\rightarrow 3}$
5	Key2	Key2	$(1 \rightarrow 4), (2 \rightarrow 4), (5 \rightarrow 6)$	3	6	$rk_{1\rightarrow 2}, rk_{2\rightarrow 3}$
6	Key2	Key3	$(1 \rightarrow 4), (2 \rightarrow 4), (4 \rightarrow 5), (3 \rightarrow 5)$	4	6	$rk_{1\rightarrow 2}, rk_{2\rightarrow 3}$
7	Key3	Key1	$(1 \rightarrow 4), (2 \rightarrow 4), (4 \rightarrow 5), (3 \rightarrow 5), (5 \rightarrow 6)$	5	7	$rk_{1\rightarrow 3}, rk_{3\rightarrow 1}, rk_{2\rightarrow 1}$
8	Key3	Key2	$(1 \rightarrow 4), (2 \rightarrow 4), (4 \rightarrow 5), (5 \rightarrow 6)$	4	7	$rk_{1\rightarrow 3}, rk_{3\rightarrow 2}, rk_{2\rightarrow 3}$
9	Key3	Key3	$(1 \rightarrow 4), (2 \rightarrow 4), (3 \rightarrow 5)$	3	5	$rk_{1\rightarrow 3}, rk_{2\rightarrow 3}$

in Table 5.1 can be reduced to rows #3 and #9. This limits the number of possible key selections and limits the complexity.
However, it shows that there is still room for optimization since variant #3 needs fewer PRE operations than variant #9.

Since the use cases of this work require a single output key, a simple yet effective approach can be applied to determine key labels. First, the compiler iterates over all nodes of the computation graph in a breadth-first search (BFS) order beginning from the inputs. For each node, it checks if the operands of the node have the same key label. In that case, the node uses the same key label as its operands. Otherwise, the default output-key label is selected. Figure 5.4 depicts an example computation graph. On the left, the initial state is shown. The right graph shows the result of the key-label propagation. The key labels are propagated through the graph until a node has operands with two different key labels. The edges between differently colored nodes require the insertion of re-encryption operations.

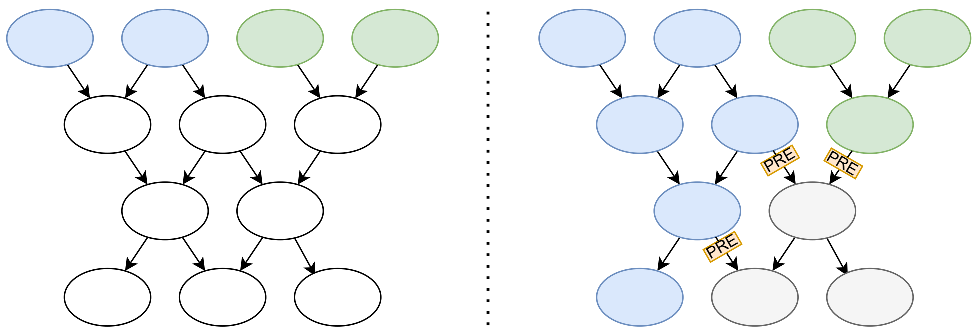

Figure 5.4.: Example Graph Before and After Key-Label Selection

If this naive algorithm is applied to the example from Figure 5.3, it starts with the input nodes. These are skipped by the procedure. In node 4, the key labels of the operands are both *Key1*. Therefore, node 4 is labeled with *Key1*, too. The following node 5 has node 4 and the input node 3 as an operand. Hence, its operands are labeled with *Key1* and *Key2*, respectively. Consequently, the key label of node 5 is set to the default output key. The resulting variant is identical to #3 of Table 5.1.

Based on the key labels, the compiler can insert PRE operations between nodes with different key labels. The insertion can be performed by iterating over all nodes of the graph. An additional PRE node must be inserted in the case that an operand of a node has a different key label than the node itself.
The current work focuses on programs with a single output key. Future versions can extend the approach to multiple keys.

6. Implementation of Proxy Re-Encryption in HElium

This Chapter focuses on the practical integration of proxy re-encryption (PRE) into the HElium compiler by implementing the concepts defined in the previous chapter. In Section 6.2, the extension of HEliums intermediate representation (IR) is discussed. This is followed by an explanation of the implementation of the reduction of PRE operations in Section 6.3. Thereafter, Section 6.4 addresses processes of the backend.

6.1. Key-Label Arguments in HElium's DSL

This section discusses additions that are made to the frontend of HElium to support multiple encryption keys. HElium uses the ANTLR4 parser generator for its frontend. Bases on a grammar description, ANTLR4 generates a lexer, parser, and a concrete-syntax-tree structure. To implement the concepts of Section 5.1, the syntax of type identifiers is extended with a key argument. Listings 6.1 and 6.2 depict excepts of the ANTLR4 grammar of HElium. In lines 48 to 50 of Listing 6.2, it shows that the *type_ident* rule is extended with an optional *key_arg*. This *key_arg* represents the key-label property of the type. The *key_arg* consists of an *IDENTIFIER* and is prefixed by an '@'-sign. For the complete grammar of HElium, see A.2

Listing 6.1: Excerpt of the ANTLR4 Grammar of the Lexer of HElium

```
...
 7 TAT:  '@';
...
74 IDENTIFIER: [a-zA-Z][a-zA-Z0-9]*;
...
```

Listing 6.2: Excerpt of the ANTLR4 Grammar of the Parser of HElium

```
...
46 ident  : IDENTIFIER;
47 type_ident: TYPEINT CLT INTEGERLIT CGT type_args* key_arg?
48      | TYPEAUTO type_args* key_arg?
49      | TYPEFLOAT CLT INTEGERLIT CGT type_args* key_arg? ;
50 key_arg: TAT IDENTIFIER;
51 type_args: LSBRACE INTEGERLIT RSBRACE ;
...
```

6.2. Intermediate-Representation Extension for Proxy Re-Encryption

This work extends the HElium compiler. HElium's type system consists of tree base types: integer, fixed-point numbers, and arrays. Each type can either represent plaintext or ciphertext data. Furthermore, each type has type-specific properties. For example, the integer type has a width property to represent integers of different bit widths. The compiler of this work adds support for PRE and multiple encryption keys to HElium.
According to Section 5.2, it is sufficient to represent encryption keys by identifiers. These are referred to as key labels. The compiler and further processes only need to know under which key the inputs are encrypted. Such a label can be interpreted as a property of types of the type system. Figure 6.1 depicts a simplified version of the extended type system of HElium. The base class of the types now has a *keyLabel* property of type *std::string* that represents the corresponding key. Each type provides a getter and a setter method for the *keyLabel* property.
The key label of input and output nodes is set during translation of the concrete-syntax tree (CST) to the abstract syntax tree (AST) in the frontend of HElium. List-

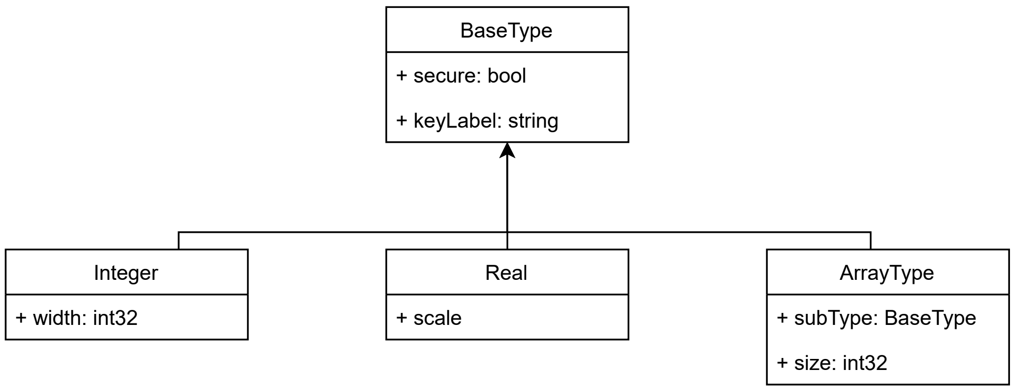

Figure 6.1.: Type System of HElium

ing 6.3 shows parts of the type construction during translation of the CST to an AST. The identifier of the DSL program is parsed as string and directly used as key label. If no identifier is provided for input or output nodes, the label *"default"* is applied. The key labels of intermediate nodes are undetermined. They are determined at later stages of the compilation chain.
The IR itself is then built from the AST. Since the AST and the IR share the same type system, the key labels of the types are not modified during the translation process. Therefore, the type identifier of the DSL are propagated to the IR in the form of key labels. In most computation graphs the key labels of intermediate nodes are undetermined.

Listing 6.3: Excerpt of *BuildASTVisitor.cpp* - AST Type Construction

```
antlrcpp::Any BuildASTVisitor::visitType_ident(
  hedsl::hedslParser::Type_identContext *context) {
 std::shared_ptr<ExprType> type;
 // ...
 if (context->key_arg()) {
  type->setEncryptionKey(context->key_arg()->IDENTIFIER()->
    getText());
 } else {
  type->setEncryptionKey("default");
 }
 // ...
}
```

One important addition to the IR is the PRE operation itself. It is implemented as a unary operation, i.e., an operation with one operand. Each PRE operation represents a re-encryption of a ciphertext from a source key to a target key. The key label of the operand defines the source key. The target key is defined by the key label of the type of operation. PRE operations are inserted by the compiler. Hence, a freshly constructed IR does not contain any PRE operation. Future versions of HElium can support the manual insertion of PRE operations via built-in methods added to the DSL.
One important rule applies to the key labels of all operations of a computation graph: Each node must share the same key label as its operands, except for PRE operations. Consequently, a PRE operation must be introduced into each edge between nodes with different key labels to ensure consistency.

6.3. Efficient Insertion of Proxy Re-Encryption Operations

As described in Section 5.3, the number of inserted PRE operations and their position affects the execution runtime. HElium inserts PRE operations in two steps. First, all key labels of intermediate nodes are determined by the compiler. Second, PRE operations are inserted based on the previously determined key labels. The number of inserted PRE operations is mainly influenced by the first step.
The main objective of the first step is the selection of suitable key labels for intermediate nodes. Thereby, the goal is to reduce the number of edges between nodes with different key labels. For programs with fewer input key-labels than inputs, there can be an opportunity for reducing PRE operations.

Algorithm 1 describes an approach to omit PRE operations. It iterates over graph nodes beginning with the inputs. Constant, input, output, and proxy re-encryption nodes are skipped. For each node, it checks whether the operands share the same key. If they do, the shared key label is applied to the node. Otherwise, the node needs a different key label. The problem of selecting a suitable target key can have high complexity, as previously shown in Section 5.3. The scenarios of this work have two constraints that can reduce the complexity: There is only one output key and only PRE operations from input to output keys are allowed. Since only re-encryptions from input keys to outputs keys are supported, the default output key is used as key label. Consequently, the *selectKey()* function is defined as

$$selectKey(node) = ' default'.$$

However, future versions of HElium can extend this mechanism and can support more than one output key.

Algorithm 1: Key-Label Propagation Algorithm

```
Data: nodes
Result: nodes with key labels
foreach $node$ of $nodes$ do topological sorting
    if $node.type \in \{PRE, INPUT, OUTPUT, CONST\}$ then
        continue;
    else
        $k \leftarrow node.operands[0]$;
        $s \leftarrow true$;
        for $i = 1; i < size(node.operands); i++$ do
            $s \leftarrow s \wedge (k = node.operands[i])$;
        if $s = true$ then
            $node.keyLabel \leftarrow k$;
        else
            $node.keyLabel \leftarrow selectKey(node)$;
```

Furthermore, the separation between key-label determination and insertion of PRE operations allows further optimizations in future versions. The key labels could be used for re-balancing optimizations or parallelizing. If the structure of the graph changes, the key labels can be recalculated easily. This separation prevents inserting or removing PRE operations during ongoing optimizations.

Nonetheless, after all optimizations took place, PRE operations have to be inserted into the graph. At this point, the key labels of all nodes of the computation graph are determined. The PRE operation itself can be inserted based on the operands or usages of a node. Algorithm 2 shows one approach to insert PRE operations. It traverses the computation graph beginning from the leaf nodes, i.e. the output nodes. For each node, it checks whether one of its successor nodes, i.e. its usages, has a different key label than the node itself. In the case that the key labels differ, it inserts a PRE-operation node between it and the successor. The algorithm uses map data structure to cache PRE nodes to prevent that the algorithm inserts an equivalent PRE operation more than one time.

Algorithm 2: Key-Label Propagation Algorithm

```
Data: nodes with key labels
Result: graph with PRE operations
foreach $node$ of $nodes$ do topological sorting
    if $node.type \in \{PRE, INPUT, OUTPUT, CONST\}$ then
        continue;
    else
        $p \leftarrow \{\}$;
        for $i = 1; i < size(node.usages); i++$ do
            if $node.usages[i].keyLabel! = node.keyLabel$ then
                if $!p.has(node.usages[i].keyLabel)$ then
                    $p.add(makePRENode(mode, node.usages[i].keyLabel))$;
                $node.usages[i].replaceOperand(node, p.at(node.usages[i].keyLabel))$;
```

6.4. Implementation of a Backend with Proxy-Re-Encryption Support

This section discusses the implementation of a compiler backend for the BFV FHE scheme. As described in Section 2.3, the main task of a backend of HElium is to prepare the execution of the computation graph. The compiler provides a program in the form of an intermediate representation to the backend. In the case of HElium, the IR is a computation graph. Then, backends may apply runtime or scheme-specific optimizations to the IR. In the last step, the backend typically determines suitable encryption parameters and serializes the program. HElium provides a common interface to serialize programs for various execution runtimes. It uses protocolbuffers, a message-structure-description protocol proposed by Google.[1]

The HElium backend with PRE support utilizes the PALISADE Lattice Cryptography Library.[2] PALISADE, is a library for lattice cryptography, especially for homomorphic encryption (HE), which implements a wide range of FHE schemes. It comprises an implementation of the BFV scheme with PRE support. The backend consists of two loosely coupled parts. The first part is integrated in the compiler. It measures specific metrics like the multiplicative depth and the number of inserted PRE operations. These metrics are necessary to determine suitable encryption parameters. Then, the backend uses the integrated circuit serialization to serialize the computation graph. None of the backends that are currently implemented in HElium does support PRE.

The second part is a parameter generator with uses a parameter-generation function of PALISADE. This function takes the measured graph metrics and derives secure scheme parameters according to the Homomorphic Encryption Standard Initiative [2].

[1]https://github.com/protocolbuffers/protobuf
[2]https://gitlab.com/palisade/palisade-release

7. Evaluation of the Proxy Re-Encryption Implementation

This chapter investigates the effect of HElium's PRE extension on the compile time, execution runtime, and efficiency, where efficiency is defined as introducing as minimum PRE operations as necessary. All experiments are conducted using the example use case of health data aggregation. Therefore, Section 7.1 gives a detailed overview of the use case and the computed function. Based on this example program, Section 7.2 presents the different conducted experiments including the corresponding evaluation results. The results and findings are summarized in Section 7.3

7.1. Use Case: Aggregation of Patient Data for Cancer Research

Hospitals, universities, and private healthcare companies collect a large amount of data about their patients. This includes, for example, protocols from medical treatments, drug applications, and surgical reports. With the introduction of the electronic medical file, more information will be digitalized [15]. This information about patients is highly confidential and high data protection requirements apply [20]. Therefore, sharing and aggregation of patient data is very complicated due to regulatory reasons. FHE combined with PRE can enable researchers to combine their patient-data sets with other researchers without sharing the patient data as plaintext. It allows performing analyses on the shared data sets without disclosing individual patient records.

For example, records of genome mutations can be compared with histology analysis or surgical reports of tumor patients to obtain information about correlations between specific genome mutations and the properties of the tumor [27, 37]. That enables researchers to develop more specific treatment methods for tumors. One measure that is analyzed in the context of cancer research is the recurrence rate of variants of tumors. It represents the percentage of patients for whom particular cancer reappears. This measure indicates the medical treatment of the cancer patient. The particular tumor type can often only be determined by surgery and the following histology analysis. A non-invasive determination of the tumor type and its properties can allow treatment of the tumor without surgery. Therefore, finding correlations between genome mutations and tumor properties is an active field of research.

Equation (7.1) depicts the calculation of the tumor-recurrence rate r' of patients.

It is calculated as an average of the recurrence of a tumor in relation to the absolute number of patients with the same tumor variant.

$$r = \frac{\text{Number of Tumor Patients with Recurrence}}{\text{Number of Tumor Patients}} \tag{7.1}$$

In order to find correlations between genome mutations and tumor recurrence, r can be calculated with the presence of different mutations. The presence of mutations for the i-th patient P_i can be efficiently encoded by bits of a bit vector denoted by $\mathbf{b}_i$. Each element b_j of $\mathbf{b}_i$ represents the presence or absence of a mutation j. Similarly, the tumor recurrence of a patient i can be encoded as a single bit a_i. The recurrence rate r in relation to the presence of mutations is defined as shown in Equation (7.2). Each element of the result vector $\mathbf{r}$ represents the recurrence rate in relation to the j-th mutation. This measure can help to find correlations between the presence of mutations and the recurrence of tumors.

$$\mathrm{r} = \frac{\sum_{i=0}^{n} a_i \cdot \mathrm{b_i}}{\sum_{i=0}^{n} \mathrm{b_i}} \tag{7.2}$$

Listing 7.1, shows the HElium code of an example implementation of the presented use case. Each patient data-set consists of two values a_i and b_i that are provided under an own key Key_i.

Listing 7.1: HElium Program: Recurrence Rate of Tumors

```
1    input a0:int<1>@Key0;
2    input b0:int<1>[1024]@Key0;
3    input a1:int<1>@Key1;
4    input b1:int<1>[1024]@Key1;
5    input a2:int<1>@Key2;
6    input b2:int<1>[1024]@Key2;
7    input a3:int<1>@Key3;
8    input b3:int<1>[1024]@Key3;
9
10   output R = a0*b0+a1*b1+a2*b2+a3*b3
11   output n = b0+b1+b2+b3
```

In lines 1 to 8, the inputs are defined for four datasets of four different participants. The outputs and the function are declared in lines 10 and 11. The division of two unknown ciphertexts is an operation with high computational complexity. Therefore, the example program computes only the two sums with FHE. The division is performed afterward at the client as plaintext operations.

7.2. Evaluation of the HElium Compiler

In this section, the HElium compiler and its PRE integration are evaluated using the previously described use case. The following aspects are evaluated. First, the effect of PRE on the execution runtime is analyzed. Second, the effectiveness of the compiler in terms of the introduction of PRE operations is studied. Third, the scalability of the compiler is examined.

All experiments are conducted on a (virtual) server with 8 Intel(R) Xeon(R) Platinum 8124M CPU cores and 16 GB of RAM that is hosted on Amazon Web Services (AWS). Each experiment is performed 100 times, and the average, as well as the standard deviation, are calculated. After each run, the results are decrypted and compared with the expected result of the corresponding plaintext function to verify correctness of the compiled program. The experiments are conducted for set sizes of up to 1000. This relates to typical data-set sizes of medical research [37]. Often, only a few hundred patient records are available at a research institution depending on the type of tumor. For a complete list of all measurements, see Appendix B.

7.2.1. The Effect of Proxy Re-Encryption on Execution Runtimes

This section discusses the effect of PRE operations on execution time. To analyze the effect, the execution time of the previously described use-case program (Listing 7.1) is measured with PRE and without PRE. The example program is compiled, executed, and its runtime, as well as the number of performed PRE operations, is measured for different patient-data set sizes n. One time with PRE and a different encryption key k for each input n, i.e., $n/k = 1$. A second time without PRE and all inputs and outputs encrypted under the same key pair. It can be expected that the use of PRE results in a higher execution time due to the added computational complexity.

Figure 7.1 shows the execution times of the example program for different data-set sizes $n \in \{4, 10, 250, 500, 750, 1000\}$. For each data set, it depicts a bar for the execution with PRE and one without. Furthermore, the execution time is divided into the raw computation time and "IO" time, i.e., the time used to deserialize keys, inputs, and to serialize the results.

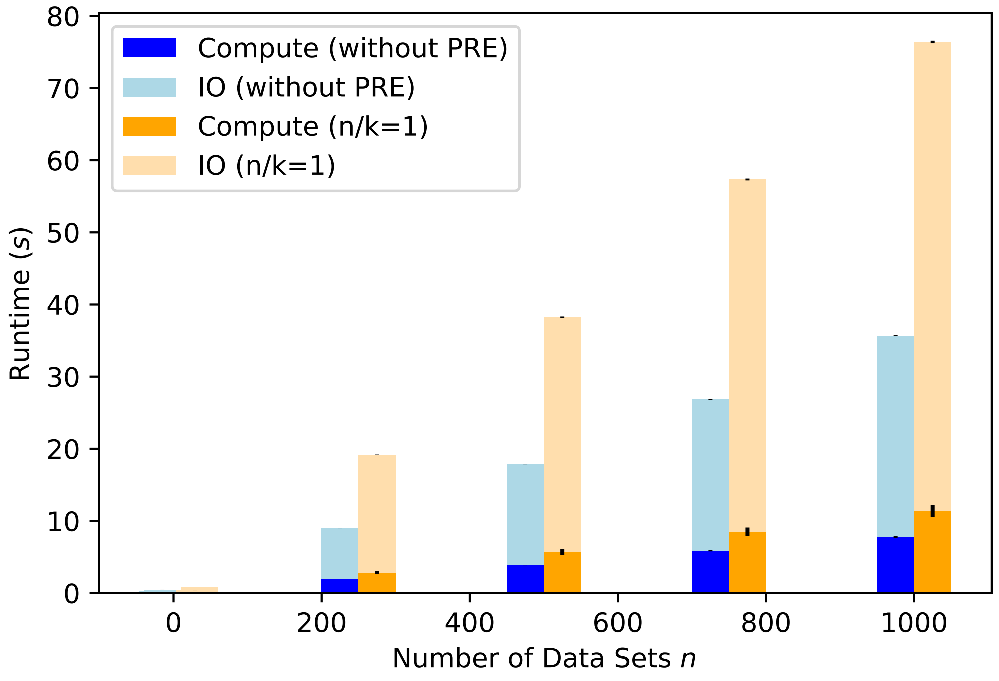

Figure 7.1.: Execution Runtime: Recurrence Rate Use Case (n/k = 1)

Table 7.1 gives a detailed overview of results from the conducted measurements. It shows that the variant with PRE has a higher execution runtime. For 1000 data sets, the total execution time is approximately 114 % higher than for the variant without PRE. However, it shows that the raw computation time is only 46 % higher than without PRE. The time spent for IO operations grows to approximately 133 %. The largest part of the execution runtime is spent on input-output operations like loading keys or inputs. For the PRE version and a set size $n = 1000$, approximately 78 % of the execution time is caused by IO operations. Consequently, only 22 % of the runtime is spent on the evaluation of the function. Furthermore, the experiments indicate that the

Table 7.1.: Execution Runtimes with and without PRE

		IO (s)	Execution (s)	Runtime (s)	IO Percentage (%)
PRE	Sets n				
w/	4	0.214	0.031	0.246	87.271
	10	0.381	0.078	0.459	83.050
	250	7.053	1.935	8.988	78.470
	500	14.007	3.865	17.872	78.375
	750	20.963	5.875	26.838	78.108
	1000	27.904	7.782	35.686	78.193
w/o	4	0.327	0.043	0.369	88.411
	10	0.716	0.108	0.824	86.898
	250	16.317	2.834	19.151	85.201
	500	32.577	5.668	38.245	85.179
	750	48.853	8.479	57.333	85.210
	1000	65.008	11.388	76.396	85.094

runtime is growing linear in the number of data sets n. In addition, the percentage of time spent on IO operations stays relatively constant for sets $n \geq 250$. That could be a result of efficiency gains of the higher number of sequential data-load operations.

To further analyze the effect of PRE on the runtime, the time spent on PRE operation is measured. Table 7.2 depicts the measured runtimes of PRE operations, the number of performed PRE operations, and the percentage of the execution runtime, that is spent on PRE operations. It shows that the percentage of time spent on PRE is relatively constant for $n \geq 250$. Additionally, the table shows the calculated average runtime of a single PRE operation. For $n = 1000$, a PRE operation takes on average 1.803 *ms*.

Table 7.2.: Percentage and Runtime of PRE Operation

Sets	Execution (s)	PRE Time (s)	PRE Time (%)	Time per PRE (ms)
4	0.043	0.012	26.998	1.445
10	0.108	0.030	27.899	1.506
250	2.834	0.899	31.723	1.798
500	5.668	1.804	31.818	1.804
750	8.479	2.604	30.712	1.736
1000	11.388	3.606	31.664	1.803

To summarize, the constant behavior of single PRE operations adds an amount of complexity to the computation that depends linearly on the data-set size n. Furthermore, it showed that the example program can aggregate data sets of 1000 patients in less than one and a half minutes. This is a suitable runtime compared to the typical runtime of preparatory processes that take place before computation. For example, the process of genome sequencing for a single data set can take from a few hours up to a few days depending on the technique that is applied and the size of the genome [29]. With the linear behavior in mind, a much larger number of data sets could be aggregated within one or a few hours.

7.2.2. The Efficiency of the Introduction of Proxy Re-Encryption Operations

This section discusses how effectively the compiler Insertions PRE operations. The compiler aims to insert only as few PRE operations as necessary. A naive approach of inserting PRE operations re-encrypts all inputs before calculation and transforms them to a common key. The number of used PRE operation p_{naive} is determined by the number of inputs i of the program, i.e., $p_{naive} = i$. From a theoretical perspective, the minimal number of PRE operations p_{min} is determined by the keys of the inputs and outputs of the program, as shown in Equation (7.3). K_I and K_O denote the set of input or output keys, respectively. In the best case, only one PRE operation is needed for all keys that are members of K_I but not of K_O, i.e., the difference of K_I and K_O.

$$p_{min} = |K_I \setminus K_O| \tag{7.3}$$

However, not for all programs, it is possible to reach p_{min} due to the structure of the programs. HElium aims to achieve a number of PRE operations p between p_{naive} and p_{min}. To measure the efficiency of HElium in inserting PRE operations, the number of necessary PRE operations is measured for different data-set sizes. Furthermore, the ratio between data sets n and keys k, denoted as n/k, is varied. The number of inserted PRE operations is measured for $n/k = 1$, $n/k = 2$, $n/k = 5$, and $n/k = n$.

Figure 7.2 depicts the number of inserted PRE operations p in relation to the number of data sets for different n/k ratios. The grey-highlighted area represents values for p that are equal or less than p_{naive}. All results within this area are equal or better than the naive approach. The figure shows that for $n/k \neq 1$, the HElium compiler inserts fewer PRE operations than p_{naive}. Consequently, HElium inserts PRE operations more efficiently than the naive approach. For $n/k = 2$, p is reduced by 50 % compared to p_{naive}. For $n/k = 5$, it is reduced by 92 %. The variant $n/k = n$ represents the case in which all inputs are encrypted under a common input key. HElium reduces the necessary PRE operation constantly to $p = 2$. That results in a reduction of up to 99.96 % of the PRE operations for $n = 1000$. However, for $n/k = 1$, the compiler can not reduce p. In that case, each data set is provided under its own key that is different from the output key and needs to be re-encrypted. The results of Figure 7.2 are measured using a program with sorted inputs.

Listing 7.2 and Listing 7.3 show a comparison of sorted and unsorted inputs. In the sorted variant, the inputs and following operations are grouped by their keys. The experiments are re-conducted with unsorted inputs to analyze the impact of the program structure.

Listing 7.2: Sorted Inputs

```
input a0:int<1>@Key0;
input a1:int<1>@Key0;
input a2:int<1>@Key1;
input a3:int<1>@Key1;
...
```

Listing 7.3: Unsorted Inputs

```
input a0:int<1>@Key0;
input a1:int<1>@Key1;
input a2:int<1>@Key0;
input a3:int<1>@Key1;
...
```

Figure 7.3 depicts the number of inserted PRE operations p in relation to the number of data sets n for $n/k = 5$. It compares a variant using sorted inputs with a variant with unsorted inputs. The figure shows that the "unsorted" variant has a higher p than the "sorted" variant. Furthermore, the "unsorted" variant has only an up to 0.4 % lower p than p_{naive}. Consequently, the efficiency of the PRE insertion of the compiler depends on the structure of the program and its inputs.

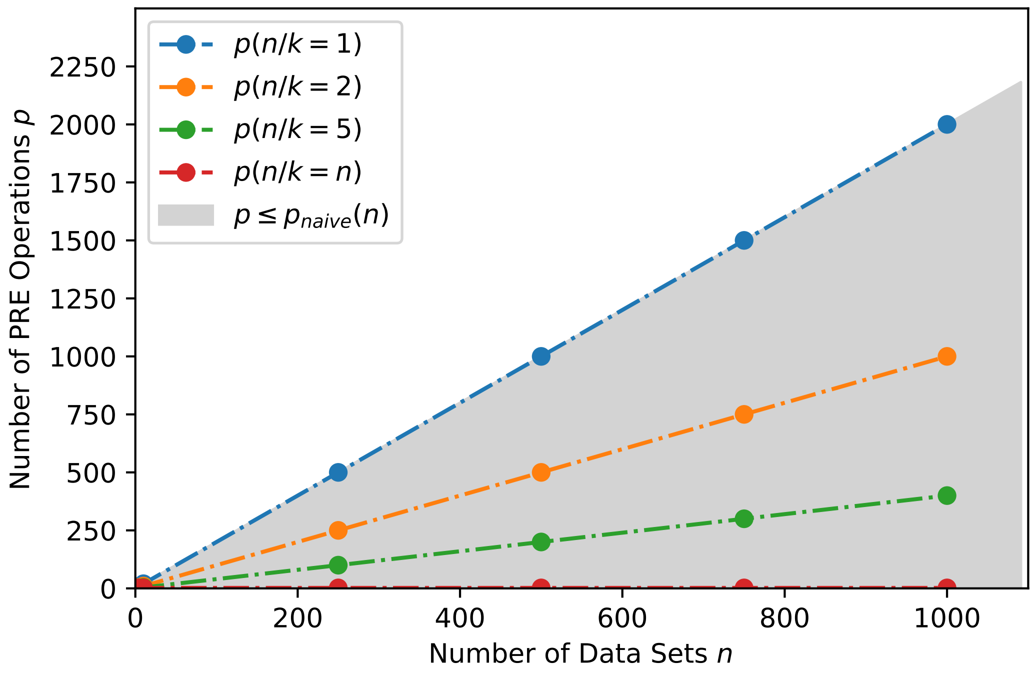

Figure 7.2.: Number of Inserted PRE Operations

To summarize, the evaluation showed that the compiler can reduce PRE operations to a minimum necessary number. Unfortunately, the compiler performs not as efficient in programs with unstructured and unsorted inputs. Hence, there are opportunities for optimization. Future versions of the compiler could automatically restructure the program to lower the number of necessary PRE operations. Additionally, future optimization could not only target a low number of PRE operations. Instead, the parallelism of the computation graph could be taken into account.

7.2.3. Scalability of the Compiler

This section analyzes how the compiler performs for different problem sizes. The compilation time is measured for different data-set sizes n of the use case. Furthermore, the runtime of the internal processes of the compiler is analyzed individually. It is expected that the compilation time grows for programs with a larger data set size. Figure 7.4 depicts the compile time relative to the set size n. In parallel, it shows the number of nodes of the compiled computation graph in relation to the set size n. While the number of nodes is linear in the set size n, the compile time grows quadratically.

In order to analyze the cause of the non-linear growth of the compile time, the runtimes of the compile processes are measured. Figure 7.5 depicts the runtime of the internal compiler processes for the use-case program relative to the data-set size n. Each color represents the runtime of one process in the compilation. For further information about the compiler processes, see Section 6.4. It shows that the "Type Deduction", "Key Selection", and "Metrics" processes cause the non-linear growth of the compile time.

Future versions of the compiler could aim to improve the complexity of these processes. However, the compiler performs the compilation for $n = 1000$ data sets in less than 0.8 s. Compared to the execution runtime of the compilation result the

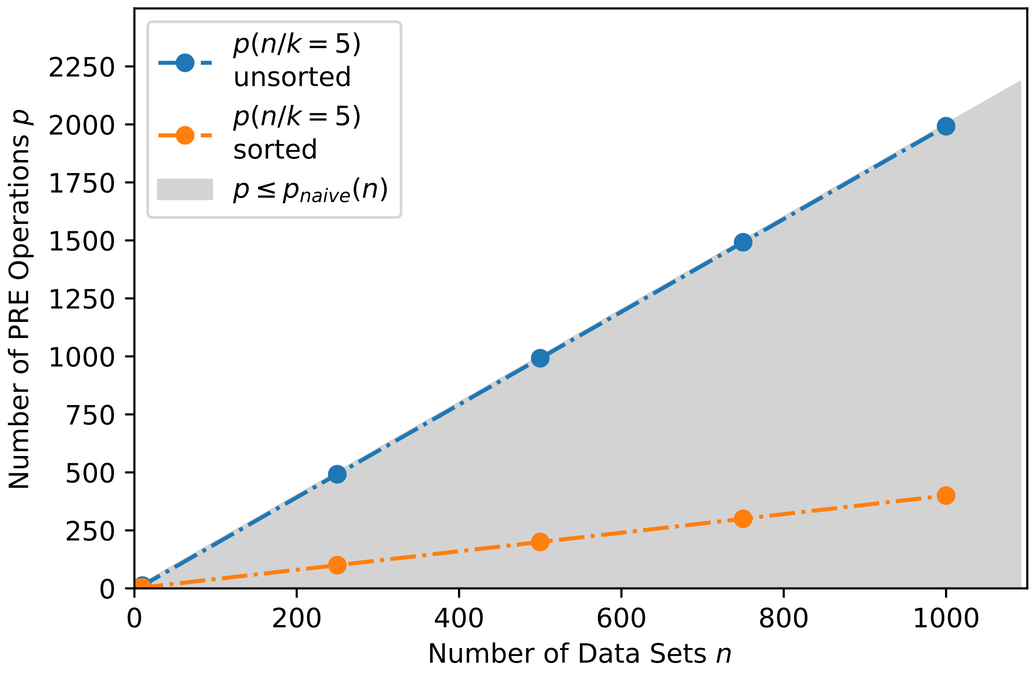

Figure 7.3.: Number of Inserted PRE Operations: Sorted and Unsorted Inputs

compile time can be still sufficient.

7.3. Summary

Within the previous sections the PRE implementation of the HElium compiler is evaluated with regard to effects of PRE on the execution runtime, the efficiency of the PRE implementation and the scalability of the compiler itself. The experiments demonstrated that the HElium compiler can generate efficient results while supporting PRE and an arbitrary number of encryption keys. It showed that PRE can be integrated into FHE programs to acceptable costs in terms of runtime performance. The compiler and the resulting FHE programs scale sufficiently and can solve problems of practically relevant size.

Furthermore, opportunities for further improvements were identified during the experiments. For example, it showed that HElium's DSL can be inconvenient to develop programs with many inputs. To mitigate this problem, HElium could support arrays and other data structures with elements of different keys. That would allow simplifying the previously presented program. Additionally, it showed that the compiler may not reach the optimal value of PRE operations in the case of restructured inputs. Therefore, internal balancing and reordering processes in the compiler would further improve its performance.

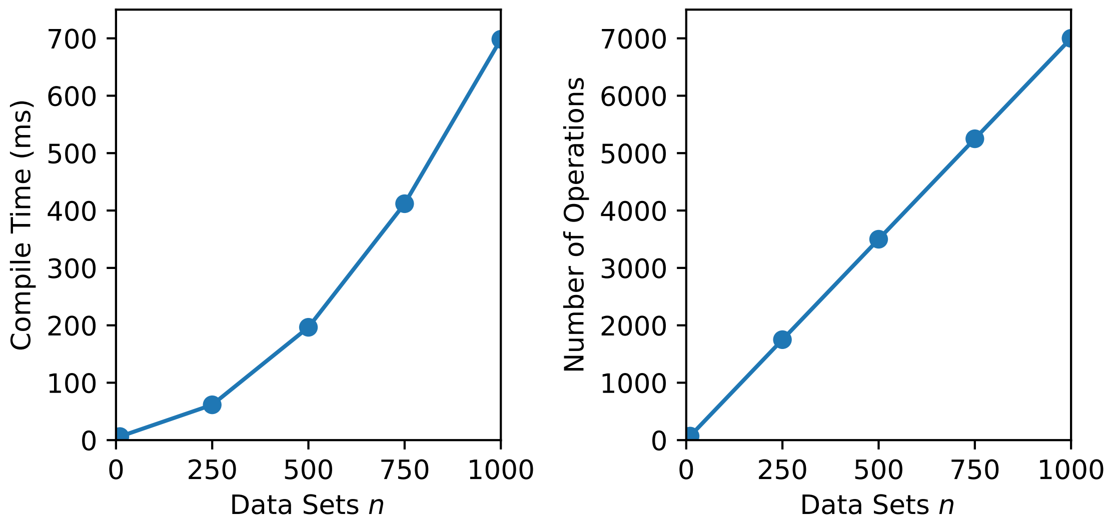

Figure 7.4.: Compile Time and Number of Operations Relative to Data-Set Size *n*

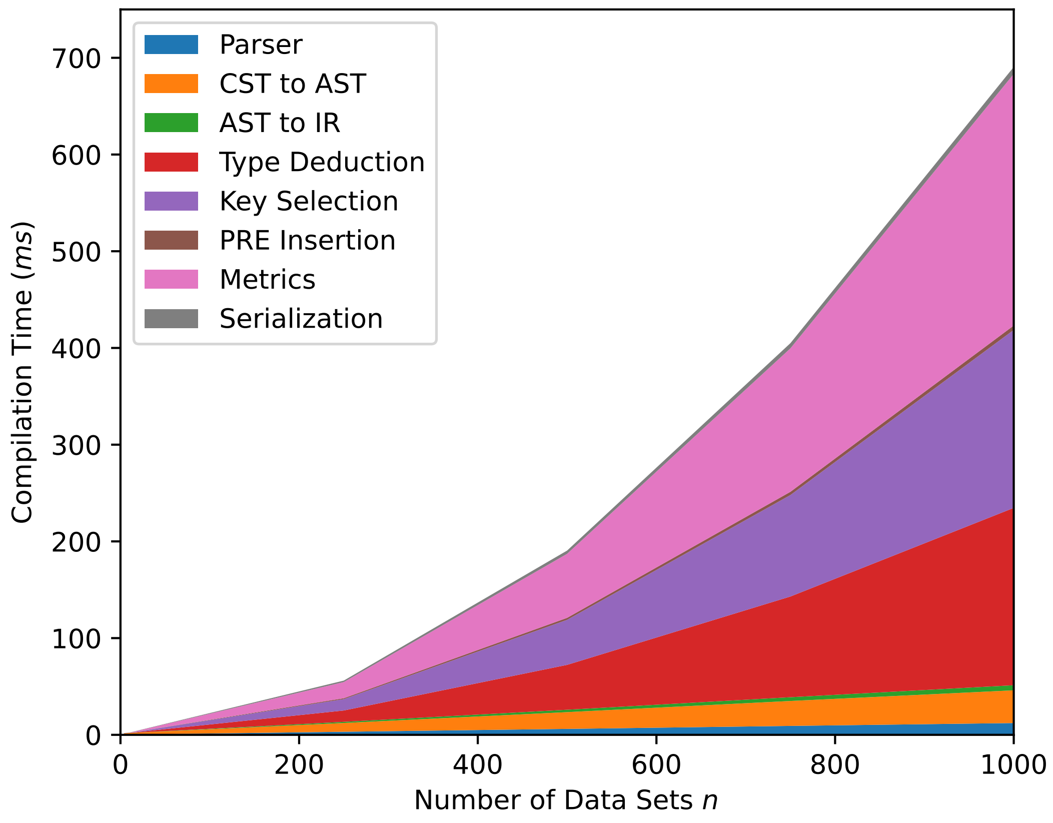

Figure 7.5.: Runtime of Compiler Processes

8. Optimization for Homomorphic Encryption and Proxy Re-Encryption

This chapter discusses different approaches for further optimization of FHE programs with and without Proxy Re-Encryption (PRE). The performance of FHE programs can benefit from parallel-execution capabilities of modern computers. Therefore, Section 8.1 discusses optimizations for operation-level and data-level parallelism. Another important optimization opportunity are client-aided computations. Section 8.2 presents different mechanisms of pre and post- computations performed by the clients of a computation.

8.1. Parallelism

8.1.1. Concurrent Execution

Concurrent execution is an important mechanism to perform multiple independent operations in parallel. Modern CPUs often consist of multiple computation cores that are able to compute multiple threads in parallel. Similarly, FHE applications can benefit from concurrent execution. Nodes of an FHE computation graph can be evaluated in parallel if they do not depend on each other. This parallelism can be implemented by parallel threads or parallel nodes. Figure 8.1 depicts a typical execution architecture for large FHE applications. It shows a management service that distributes the workload to worker nodes. Each worker node spawns a set of threads that execute parts of the program. This architecture has advantages for the execution of large programs. By using multiple worker nodes and multiple threads per worker, the computation can be distributed. Consequently, even complex analytics can be performed in an acceptable time. However, the architecture introduces the main challenge: the partitioning of the computation graph for different workers and threads.

Dependencies between workers require additional communication. Due to the size of ciphertexts, communication is costly and can have a negative impact on performance. Therefore, it is important to avoid dependencies between workers. One approach to this is the use of graph-partitioning algorithms. This is only worthwhile for circuits of a certain size and with opportunities for parallelism.

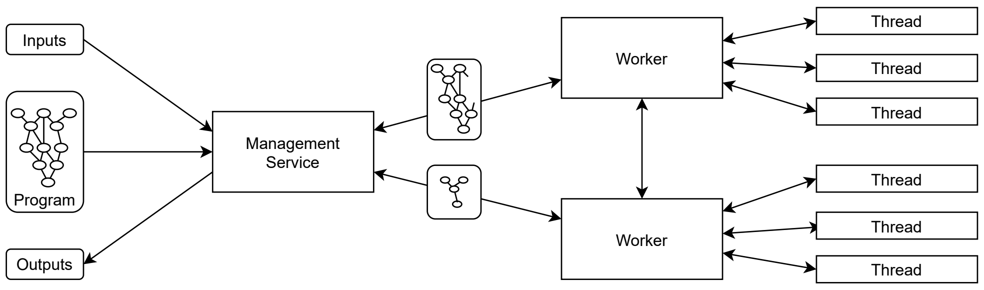

Figure 8.1.: Large-Scale Architecture for Execution of FHE Programs

8.1.2. Data Parallelism

Modern CPUs provide a variety of vector instructions, also referred to as single instruction multiple data (SIMD). These allow performing a single operation on multiple data values in parallel. FHE schemes can provide a similar mechanism that is referred to as batching or ciphertext packing [9]. This applies especially to schemes that rely on the ring variant of the learning with errors problem (RLWE). These allow encrypting multiple messages into one ciphertext. For example, a BFV ciphertext can encrypt $n/2$ elements, referred to as slots. The scheme parameter n denotes the polynomial degree. Typical values for n are between 1024 and 32,768 according to the homomorphic encryption standardization initiative [2].

Homomorphic operations performed on packed ciphertext apply to all elements of the ciphertext in parallel [9]. Therefore, batching allows SIMD-like operations on ciphertexts. It can speed up programs with a large amount of parallel data. For example, a program that uses batching with $n/2 = 1024$ slots can perform an operation on 1024 elements in parallel. That results in a theoretical amortized speedup of 1023 compared to the execution on a single slot. For example, Chen et al. used batching perform fast private set intersection [11]. The following constructions of utilizing batching for parallelization are based on previous work. For example, Dathathri et al. proposed mechanism to encode matrices using batching to perform efficient matrix multiplications. Furthermore, there are a variety of use cases that benefit from efficient encoding, for example [11, 7, 32, 4].

Currently, available schemes do only support performing element-wise operations on batched ciphertexts. This includes addition, multiplication, and rotation. Performing operations on specific elements is not directly possible. However, by using addition, multiplication, and rotation array access can be emulated, as shown in Figure 8.2 For example, access of the i-th element of a vector $\mathbf{a} = (a_0, a_1, \ldots, a_{n/2})$ can be achieved by multiplication with a selection-mask vector $\mathbf{b}$. That vector b is filed with zeros and its i-th element is one. The resulting vector contains the selected element at the i-th position and is filled with zeros. It shows that there exist differ-

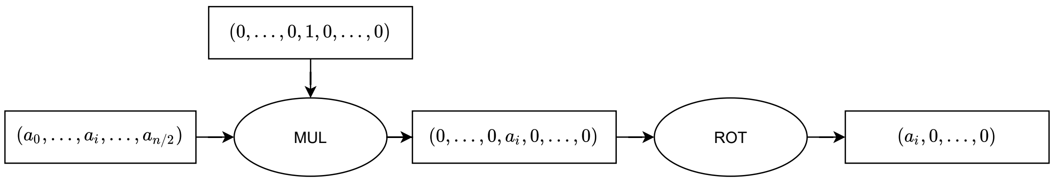
$(0, \ldots, 0, 1, 0, \ldots, 0)$
$(a_0, \ldots, a_i, \ldots, a_{n/2})$
MUL
$(0, \ldots, 0, a_i, 0, \ldots, 0)$
ROT
$(a_i, 0, \ldots, 0)$

Figure 8.2.: Emulating Array Access using Rotation and Addition

ent variants to encode data into the slots of a batched ciphertext. The slots can be used to encrypt vectors or matrices [18, 17]. In such cases, each slot can represent an element of the vector or matrices. Scalars can be either encoded in a particular slot or encoded in all slots in parallel. The second variant is beneficial for following element-wise vector operations.

Figure 8.3 depicts one approach to "copy" a particular element to all slots of a batched ciphertext. It uses $\log \frac{n}{2}$ rotation and addition operations.

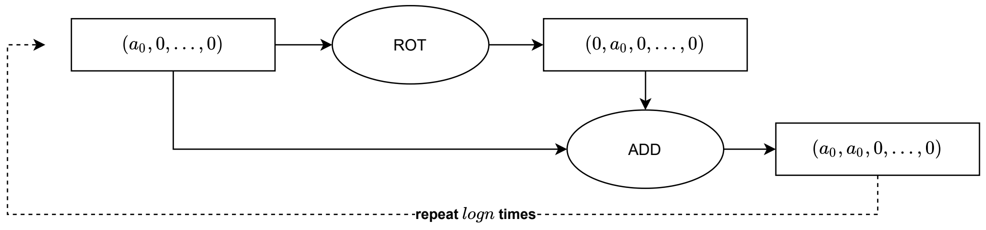

Figure 8.3.: Emulating Array Construction from Scalar using Rotation and Addition

Similarly, arrays can be constructed from multiple batched ciphertexts that encode a single scalar. Figure 8.4 shows a construction that uses rotation and addition operations to combine multiple ciphertexts with a scalar encoded into the first slot. For an array of length l it requires l rotations and $\log l$ addition operations. The presented

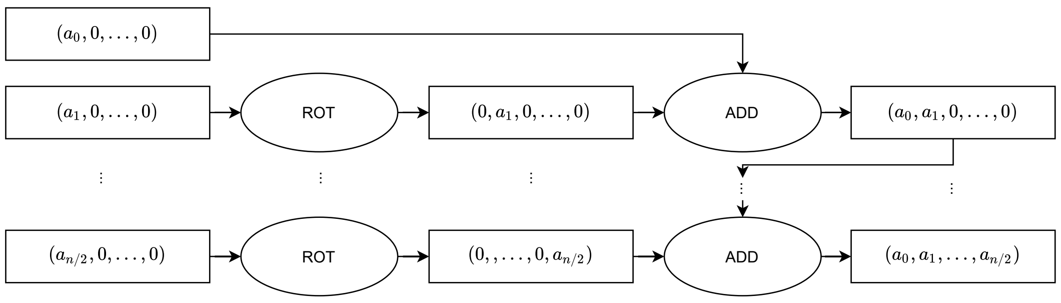
$(a_0, 0, \ldots, 0)$
$(a_1, 0, \ldots, 0)$
ROT
$(0, a_1, 0, \ldots, 0)$
ADD
$(a_0, a_1, 0, \ldots, 0)$
$(a_{n/2}, 0, \ldots, 0)$
ROT
$(0, , \ldots, 0, a_{n/2})$
ADD
$(a_0, a_1, \ldots, a_{n/2})$

Figure 8.4.: Emulating Array Construction from Multiple Scalars using Rotation and Addition

approaches of transformation between different encoding variants of batched ciphertexts require additional operations. Therefore, they add computational complexity. However, there are programs that can benefit from an automatic transformation from a scalar encoding to vector encoding. For example, a transformation of l scalar-encoded ciphertexts to an array encoding and back to a scalar encoding requires additional $2l$ rotations, l multiplications, and $\log l$ additions. Although this requires additional operations and adds complexity, it can speed-up operations that can be performed in parallel by a factor of $l - 1$.

HElium supports batching for FHE schemes that support this technique. The compiler translates element-wise operations on arrays to batched ciphertext operations. For example, the use case presented in Section 7.1 utilizes batching to compute the recurrence rate relative to many mutations in parallel. However, HEliums batching support is limited. It does only support element-wise operations on vectors. Access to single elements of the vector or automatic restructuring of scalar data into vectors is not supported yet. Future versions could automatically detect opportunities for optimizations through batching and automatically insert necessary transformation operations.

8.2. Client-Aided Computation

This section presents mechanisms to utilize the computation power of the clients, i.e., data-providing participants, to improve the execution-runtime performance of FHE programs. Subsections 8.2.1 and 8.2.2, discuss approaches of offloading parts of the computation graph to the clients.

8.2.1. Pre-Computation by the Client

The encoding of inputs and outputs of an FHE program is an important starting point for optimizations. For example, one straightforward optimization can be the selection of suitable bit lengths for the input values. In many use cases, the domain of inputs can be reduced by simple operations of the data-providing party.

The computation of polynomials using FHE is a good example problem to highlight the advantages of pre-computation by the client. The example, a polynomial of $f(x) = ax^9 + bx - 5$ requires computational-complex multiplications if the computation is performed completely using FHE. It would require at least $log_2 9$ multiplications of x to compute x^9. Multiplications using FHE can be computational complex. However, for the data-providing client, the calculation of x^9 is an operation on plaintexts with a low complexity compared to FHE. Therefore, the client can directly provide the result of x^9 as an additional input x' to the computation. This results in the function $f(x', x) = ax' + bc - 5$ whereas $x' = x^9$ is a preliminary function that is calculated by the client that provides x. Consequently, the communication complexity is doubled since the client has to transmit two inputs x and x' instead of one input x. However, the function $f(x', x)$ has a much lower computational complexity because it consists only of additions and subtractions. A similar construction was used by Chen et al. to optimize private set intersection [11]. In this example, it would be even possible to replace the FHE scheme with a partially homomorphic encryption (PHE) scheme. A PHE scheme supports only one type of homomorphic operation, for example, additions and multiplications. Despite this limitation, PHE schemes are much more efficient than FHE schemes.

To summarize, the pre-computation of some operations of the function reduces the complexity effectively. Operations that are computational complex in FHE can be performed in plaintext on the client. This can be accompanied by the cost of additional communication and additional operations on the client. Therefore, in some cases, it is a tradeoff between computational and communication complexity.

For the presented example it is unchallenging to identify opportunities for pre-computation. However, for more complex programs this task can become difficult. A compiler can support developers with the identification of optimization opportunities. From a theoretical perspective, this process follows similar rules as the insertion of PRE operations. Each operation that depends by itself or by its operands only on the inputs of a single participant can be pre-computed by the participant. Figure 8.5 depicts the initial computation graph of $g(x) = ax^2 + bx$ on the left side. Nodes that depend only on inputs, i.e., data of the same single party are colored homogeneous.

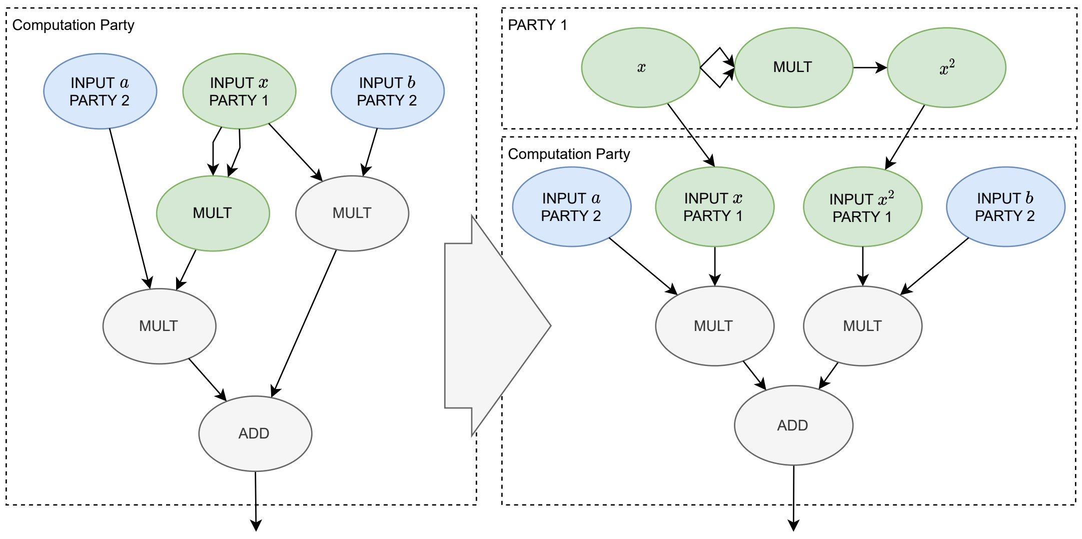

Figure 8.5.: Computation Graph of $g(x) = ax^2 + bx$ with and without Pre-Computation

On the right, it shows a possible computation graph that utilizes pre-computation. This computation graph is partitioned into two parts. One part is executed at the participant *PARTY1*. The other part is still executed by a computation party. Similarly to the previous example, the function x^2 is pre-computed by the client and provided as an additional input.

The pre-computation of parts of the computation circuit has one major limitation. It requires that the function that is to be computed is known at the time of encryption of the inputs. As it may require specific encoded inputs or additional inputs, the function or at least the pre-computed part must be known before encryption.

One approach to determine operations on the computation graph that can be pre-computed is tainting or labeling the operations. Therefore, each party is a label assigned. The labels represent a partition of the computation graph. In the first step, only the input nodes are labeled with the label of the corresponding input-providing party. Then the labels can be propagated through the graph by a greedy algorithm. Beginning from the input nodes can traverse the computation graph from nodes to their usages in a depth-first-search manner. Each visited node can be labeled in the case that all operands of the node share the same label. This process is repeated until no more nodes can be labeled. An example result of a labeled graph can be seen in Figure 8.5. Similar to this example, the graph can now be partitioned into subgraphs according to the applied labels.

However, as previously discussed, this approach can increase the communication complexity, i.e., the amount of data that must be transmitted by the participants. There are maybe scenarios that are limited in terms of communication. For example, mobile applications. Therefore, a heuristic can be applied to the labeled nodes in order to decide whether the nodes should be pre-computed or part of the main computation. This decision is done with the tradeoff between communication and computational complexity in mind. The selected or labeled subgraphs can be separated from the main computation graph. The leaf nodes of the subgraphs must be replaced by additional input nodes of the main graph to maintain correctness. These inputs represent the intermediate results of the pre-computation subgraphs. An example of such a split graph is shown in Figure 8.5. The resulting pre-computation graphs can be evaluated directly by the input-providing participants.

8.2.2. Post-Computations

The same mechanism can be applied to the outputs of computation in order to offload post-computations of the function to the receiving party. Similarly to pre-computation, operations that are dependencies of outputs can be separated from the main computation graph to form a post-computation. The receiving party can perform the post-computation in plaintext. Consequently, the overall-computational complexity decreases compared to a complete FHE-based solution. For example, in the use case presented in Section 7.1, the final division operation is performed by the receiving party to lower the computational complexity of the function. A similar construction is used in [5] to offload a complex division operation to a participant with the decryption key. Chen et al. implemented a construction that offloads final comparison operations to the decrypting participant [11].

Figure 8.6 depicts an example computation graph of the function $h(a,b,x) = \frac{a \cdot x}{x+b}$ on the left. On the right, it shows on possibility to partition the computation graph into a main computation and a post computation. The parts $h_0(a,x) = a \cdot x$ and $h_1(b,x) = x + b$ are computed by the computation party. The receiving party performs the second part of the computation and combines both intermediate results via a division $h(a,b,x) = \frac{h_0(a,x)}{h_1(b,x)}$. This lowers the complexity of the main computation but requires additional communication since two encrypted results must be transmitted to the receiving party.

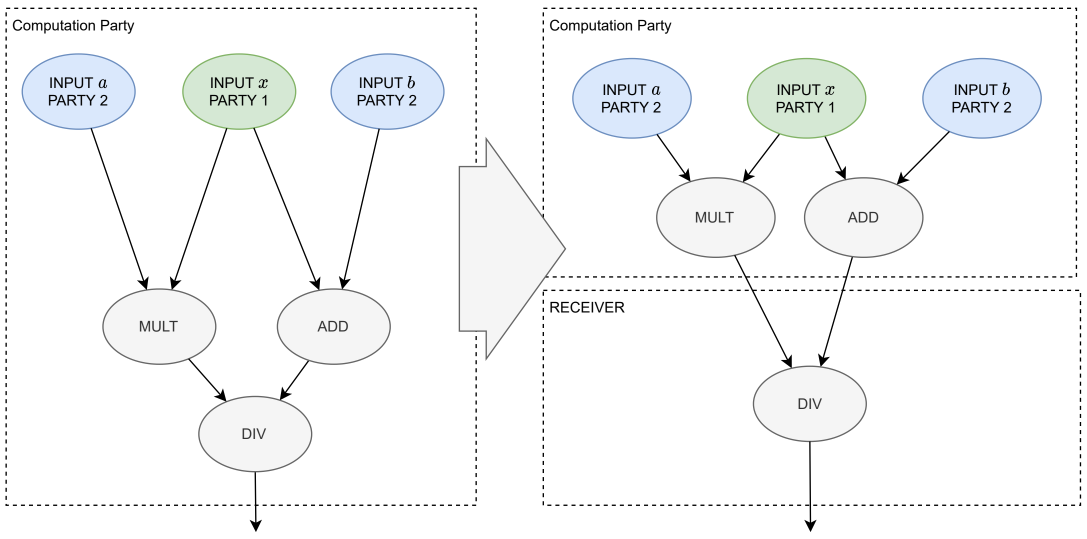

Figure 8.6.: Computation Graph of $h(a,b,x) = \frac{a \cdot x}{x+b}$ with and without Post-Computation

However, an important aim of FHE is to enable computations and analyses of data while preserving privacy and confidentiality. Therefore, it is important to limit post-computations to prevent leakage of the inputs. As one approach, the separation of post-computations can be limited to operations that do not yield a result with lower entropy than its operands. As another approach, the computation party could blind the intermediate results such that the result of the computation does not change and the receiving party does not learn the intermediate results. For example, in the previously presented program showed in Figure 8.6, the intermediate results $h_0(a,x) = a \cdot x$ and $h_1(b,x) = x + b$ could be blinded by a random r. Therefore, the blinding factor r is randomly sampled from a Gaussian distribution and multiplied with the intermediate results, as depicted in Equations (8.1) and (8.2).

$$h'_0(a, x) = a \cdot x \cdot r \tag{8.1}$$

$$h'_1(b, x) = (x + b) \cdot r \tag{8.2}$$

When combining these two blinded intermediate results, the blinding factor r eliminates itself, as shown in Equation (8.3). The intermediate values are secured without changing the result.

$$\begin{aligned} h(a, b, x) &= \frac{h'_0(a, x)}{h'_1(b, x)} \\ &= \frac{a \cdot x \cdot r}{(x + b) \cdot r} \\ &= \frac{a \cdot x}{x + b} \end{aligned} \tag{8.3}$$

Furthermore, if a compiler applies partitioning of pre and post-computations simultaneously without any limitations, FHE can be removed completely from the computation. In this case, the inputs or their pre-computed successors are revealed to the receiving party. This violates the confidentiality requirements. Consequently, the separation of post-computations must be used consciously to prevent data leakage.

8.3. Summary

The previous sections discuss a variety of opportunities for optimizations of FHE programs with support for PRE. Each of them aims to improve the execution runtime of FHE programs towards practical usability. It showed that there are opportunities for optimization at each level of abstraction: On the data level, batching allows performing SIMD-like operations on ciphertexts. This can provide an important efficiency improvement for data-parallel problems. On the operation level, the replacement of complex operations with approximate alternatives can be beneficial. Additionally, operations can be offloaded to participants of the computation and performed in plaintext. On graph level, re-balancing and re-structurings of the computation graph provide many opportunities for optimization. For example, re-ordering of chained operations can reduce the number of necessary PRE operations significantly, as shown in Section 7.2.2. Furthermore, offloading of pre and post-computations is an important mechanism to reduce the complexity of programs. On a larger scale, the computation power of modern cloud environments can be used to execute even complex programs with many data records. Partitioning of computation graphs can help to distribute the computation to many computation nodes. In summary, there have been various opportunities for further optimization of FHE programs identified that are a base for future work.

9. Conclusion and Future Work

This chapter concludes the finding of the thesis. The following section 9.1 provides a conclusion of this thesis by recapitulating the previous chapters. Section 9.2 presents opportunities for further improvement that have been identified, followed by the thesis statement in Section 9.3.

9.1. Conclusion

In this thesis, the efficient integration of proxy re-encryption (PRE) into compilers for fully homomorphic encryption (FHE) is studied.

First, Chapter 2, discusses the preliminaries of this work. This includes information about FHE and PRE, as well as a presentation of the HElium compiler and its parts. Chapter 3 discusses relevant related lines of work and focuses on libraries and compilers for FHE

As a foundation for the following chapters, the computation scenario is elaborated in Chapter 4. It addresses the targeted setting for computation, communication, and key management. Section 4.3 derives general requirements on a PRE integration from the elaborated scenario. This includes requirements on usability, security, and performance.

Based on that, concepts for integration of PRE into the FHE compiler HElium are developed in Chapter 5. This chapter focuses on the abstraction of encryption keys by key labels and its conceptual integration into HElium. Therefore, as a first step, a concept to extend HEliums domain-specific language (DSL) with an encryption-key property, i.e., the key label, is presented in Section 5.1. Similarly, Section 5.2 addresses a concept to integrate the key label as a regular property into HElium's type system. By using the abstraction of key labels, Section 5.3 discusses approaches of insertion of re-encryption operations in the computation graph. The main idea is to propagate the key labels from the roots of the computation graph to the leaf node to perform as few re-Encryptions as necessary.

Chapter 6 addresses the implementation of the previously defined concepts into HElium. Therefore, in Section 6.1, the key labels are integrated into HElium's DSL via an extension of the grammar of HElium. Thereafter, the re-encryption operation is added to the intermediate representation of HElium, and the type system is extended by the key-label property, in Section 6.2. The key labels are propagated through the computation graph. One algorithm is presented and implemented in Section 6.3. Section 6.4 addresses the functionalities and the implementation of the compiler backend for PRE. For example, the backend collects additional metrics for the generation of parameters for the FHE scheme. In the last step, it serializes the computation graph.

The resulting integration of PRE into HEliums is evaluated in Chapter 7. It is evaluated with a program developed for combining medical data from cancer patients to identify correlations between genome mutations and the recurrence of the tumors. This evaluation addresses three main aspects. In the first experiment, the effect of PRE on the execution runtime is investigated. Therefore, Section 7.2.1 compares the execution runtime with PRE with the runtime of a similar program without PRE. The variant that uses PRE has a higher execution runtime that is mainly caused by the additionally required communication of re-encryption keys. In Section 7.2.2, the second experiment analyses the efficiency of the PRE integration in the insertion of PRE operations by measuring the number of introduced PRE operations. The compiler aims to insert only as few as necessary PRE operations. It showed that the compiler can achieve a number of PRE operations that is close to the theoretical minimum. However, the level of efficiency depends on the structure of the program. The scalability of the compiler is investigated in a third experiment, in Section 7.2.3. Compile times are measured for different problem sizes. It shows that the compiler scales well for practical problem sizes.

Chapter 8 addresses opportunities for further optimization. Section 8.1.2 discusses approaches to parallelize the execution on data, operation, and graph level. The Sections 8.2.1 and 8.2.2 present mechanisms to reduce the complexity of FHE programs by offloading computationally complex operations to data-providing or data-receiving participants.

In summary, this thesis develops concepts of integration of PRE into FHE programs and describes the implementation of those concepts into the HElium compiler. The result is the first compiler for FHE that supports PRE and automatically optimizes the use of re-encryption operations.

9.2. Future Work

This section addresses opportunities for further improvement that have been identified.

This thesis already provides a practically usable integration of proxy re-encryption (PRE) into the HElium compiler as shown by the evaluation in Chapter 7. The evaluation helped to identify opportunities for further improvements that are described in Chapter 8. Therefore, future versions of HElium could implement advanced mechanisms to enable parallelism on multiple levels. On the data level, for example, HElium could automatically determine the most efficient encoding of data and introduce required transformations automatically. On the operation level, advanced scheduling algorithms could enable more parallelism. Furthermore, the PRE integration in HEliums is currently limited to a constrained scenario. Additionally, future versions could provide more flexibility by supporting arbitrary settings of input and output keys.

HElium and its DSL provide already a set of critical functionalities. Nonetheless, future versions can further improve the developer experience of the compiler. For example, a mechanism to define higher-level data structures could improve HElium's usability. A module system that allows developers to encapsulate algorithms into separate models could reduce code duplications and could enable the development of function libraries for HElium. Furthermore, more comprehensive support through developer tools like integrated-development environments can ease access to HElium and can improve usability.

9.3. Thesis Statement

This thesis addresses the question of how fully homomorphic encryption (FHE) programs can be efficiently extended by proxy re-encryption (PRE) to allow computations over inputs under different keys. It describes the extension of an existing compiler with PRE functionalities and discusses approaches for further optimization. The practicality of the compiler and the generated FHE programs was demonstrated using the analysis of confidential patient data as a lifelike example.

Bibliography

[1] Abbas Acar et al. “A Survey on Homomorphic Encryption Schemes: Theory and Implementation”. In: *ACM Computing Surveys* 51.4 (July 2018).

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

[3] Gilad Asharov et al. “Multiparty Computation with Low Communication, Computation and Interaction via Threshold FHE”. In: *Advances in Cryptology – EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 483–501.

[4] Gilad Asharov et al. “Privacy-Preserving Search of Similar Patients in Genomic Data”. In: *Proceedings on Privacy Enhancing Technologies* 2018.4 (2018), pp. 104–124.

[5] Kilian Becher, J. A. Gregor Lagodzinski, and Thorsten Strufe. “Privacy-Preserving Public Verification of Ethical Cobalt Sourcing”. In: *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2020, pp. 998–1005.

[6] Matt Blaze, Gerrit Bleumer, and Martin Strauss. “Divertible protocols and atomic proxy cryptography”. In: *Advances in Cryptology — EUROCRYPT’98*. Ed. by Kaisa Nyberg. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 127–144.

[7] Fabian Boemer et al. “nGraph-HE”. In: *Proceedings of the 16th ACM International Conference on Computing Frontiers*. Ed. by Francesca Palumbo et al. New York, NY, USA: ACM, 2019, pp. 3–13.

[8] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) Fully Homomorphic Encryption without Bootstrapping”. In: *ACM Transactions on Computation Theory* 6.3 (2014), pp. 1–36.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. “(Leveled) fully homomorphic encryption without bootstrapping”. In: *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference on - ITCS ’12*. Ed. by Shafi Goldwasser. New York, New York, USA: ACM Press, 2012, pp. 309–325.

[10] Sergiu Carpov, Paul Dubrulle, and Renaud Sirdey. “Armadillo”. In: *Proceedings of the 3rd International Workshop on Security in Cloud Computing*. Ed. by Feng Bao et al. New York, NY: ACM, 2015, pp. 13–19.

[11] Hao Chen, Kim Laine, and Peter Rindal. “Fast Private Set Intersection from Homomorphic Encryption”. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. New York, NY, USA: ACM, 2017.

[12] Jung Hee Cheon et al. “Homomorphic Encryption for Arithmetic of Approximate Numbers”. In: *Advances in Cryptology – ASIACRYPT 2017*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture notes in computer science. Cham: Springer International Publishing, 2017, pp. 409–437.

[13] Eduardo Chielle et al. *E3: A Framework for Compiling C++ Programs with Encrypted Operands*. Cryptology ePrint Archive, Report 2018/1013. `https://ia.cr/2018/1013`. 2018.

[14] Ilaria Chillotti et al. “Faster Fully Homomorphic Encryption: Bootstrapping in Less Than 0.1 Seconds”. In: *Advances in cryptology - ASIACRYPT 2016*. Ed. by Jung Hee Cheon. Lecture notes in computer science. Berlin: Springer, 2016, pp. 3–33.

[15] Martin R. Cowie et al. “Electronic health records to facilitate clinical research”. In: *Clinical research in cardiology : official journal of the German Cardiac Society* 106.1 (2017), pp. 1–9.

[16] Eric Crockett, Chris Peikert, and Chad Sharp. “ALCHEMY: A Language and Compiler for Homomorphic Encryption Made EasY”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1020–1037.

[17] Roshan Dathathri et al. “CHET: An Optimizing Compiler for Fully-Homomorphic Neural-Network Inferencing”. In: *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. Phoenix, AZ, USA: Association for Computing Machinery, 2019, pp. 142–156.

[18] Roshan Dathathri et al. “CHET: Compiler and Runtime for Homomorphic Evaluation of Tensor Programs”. In: *CoRR* abs/1810.00845 (2018).

[19] Roshan Dathathri et al. “EVA: an encrypted vector arithmetic language and compiler for efficient homomorphic computation”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. Ed. by Alastair F. Donaldson. ACM Digital Library. New York,NY,United States: Association for Computing Machinery, 2020, pp. 546–561.

[20] Edward S. Dove and Mark Phillips. “Privacy Law, Data Sharing Policies, and Medical Data: A Comparative Perspective”. In: *Medical Data Privacy Handbook*. Ed. by Aris Gkoulalas-Divanis and Grigorios Loukides. Cham: Springer International Publishing, 2015, pp. 639–678.

[21] Junfeng Fan and Frederik Vercauteren. “Somewhat Practical Fully Homomorphic Encryption”. In: *Cryptology ePrint Archive, Report 2012/144*. 2012.

[22] Craig Gentry. “A fully homomorphic encryption scheme”. PhD thesis. Stanford University, 2009.

[23] Craig Gentry and Shai Halevi. “Implementing Gentry’s Fully-Homomorphic Encryption Scheme”. In: *Advances in Cryptology – EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–148.

[24] Craig Gentry, Amit Sahai, and Brent Waters. “Homomorphic Encryption from Learning with Errors: Conceptually-Simpler, Asymptotically-Faster, Attribute-Based”. In: *Advances in Cryptology – CRYPTO 2013*. Ed. by Ran Canetti and Juan A. Garay. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 75–92.

[25] Shai Halevi, Yuriy Polyakov, and Victor Shoup. “An Improved RNS Variant of the BFV Homomorphic Encryption Scheme”. In: *Topics in Cryptology – CT-RSA 2019*. Ed. by Mitsuru Matsui. Cham: Springer International Publishing, 2019, pp. 83–105.

[26] Adriana López-Alt, Eran Tromer, and Vinod Vaikuntanathan. "On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption". In: *Proceedings of the 44th symposium on Theory of Computing - STOC '12*. Ed. by Howard Karloff and Toniann Pitassi. New York, New York, USA: ACM Press, 2012, p. 1219.

[27] Amy M Braden et al. "Breast cancer biomarkers: risk assessment, diagnosis, prognosis, prediction of treatment efficacy and toxicity, and recurrence". In: *Current pharmaceutical design* 20.30 (2014), pp. 4879–4898.

[28] Pascal Paillier. "Public-Key Cryptosystems Based on Composite Degree Residuosity Classes". In: *Advances in Cryptology — EUROCRYPT '99*. Ed. by Jacques Stern. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 223–238.

[29] Erik Pettersson, Joakim Lundeberg, and Afshin Ahmadian. "Generations of sequencing technologies". In: *Genomics* 93.2 (2009), pp. 105–111.

[30] Yuriy Polyakov et al. "Fast Proxy Re-Encryption for Publish/Subscribe Systems". In: *ACM Transactions on Privacy and Security* 20.4 (2017), pp. 1–31.

[31] R. L. Rivest, A. Shamir, and L. Adleman. "A method for obtaining digital signatures and public-key cryptosystems". In: *Communications of the ACM* 21.2 (1978), pp. 120–126.

[32] Shai Halevi and Victor Shoup. "Faster Homomorphic Linear Transformations in HElib". In: *Advances in Cryptology – CRYPTO 2018 - 38th Annual International Cryptology Conference, 2018* (2018), pp. 93–120.

[33] Nigel P. Smart. *Cryptography: An Introduction, 3rd Edition*. London: McGraw-Hill College, 2013.

[34] Nigel. P. Smart and F. Vercauteren. "Fully homomorphic SIMD operations". In: *Designs, Codes and Cryptography* 71.1 (2014), pp. 57–81.

[35] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. "SoK: Fully Homomorphic Encryption Compilers". In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 1092–1108.

[36] Alexander Viand and Hossein Shafagh. "Marble". In: *Proceedings of the 6th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*. Ed. by Michael Brenner. ACM Conferences. New York, NY: ACM, 2018, pp. 49–60.

[37] Sayaka Yuzawa, Hiroshi Nishihara, and Shinya Tanaka. "Genetic landscape of meningioma". In: *Brain tumor pathology* 33.4 (2016), pp. 237–247.

A. Additional Resources for HElium

A.1. Supported Operations of HEliums Intermediate Representation

Table A.1.: Intermediate-Representation Operations of HElium with Description

Operation	Description	Newly added
ADD(a, b)	Addition	-
SUB(a, b)	Substraction	-
MUL(a, b)	Multiplication	-
DIV(a, b)	Division	-
MOD(a, m)	Modulus	-
POW(a, p)	Power	-
AND(a,b)	Boolean AND	-
OR(a, b)	Boolean OR	-
XOR(a, b)	Boolean XOR	-
NAND(a, b)	Boolean NAND	-
NOT(a)	Boolean NOT	-
MUX(s, a, b)	Boolean MUX	-
EQUAL(a, b)	Equal	-
NEQUAL(a, b)	Unequal	-
GT(a, b)	Greater than	-
LT(a, b)	Lower than	-
GET(a, b)	Greater or equal	-
LET(a, b)	lower than or equal	-
ARRAYACCESS(a, i)	Access the *i*-th element of *a*	-
INPUT(name)	Input with *name*	-
OUTPUT(a, name)	Output *a* with *name*	-
CONST(value)	Const value	-
PRE(value)	Proxy re-encryption	yes

A.2. ANTLR4 Grammar of HElium

The following Listings depict the grammar of the HElium DSL written for ANTLR4. It is divided into a lexer grammar, shown in Listing A.1 and a parser grammar, shown in Listing A.2.

Listing A.1: ANTLR4 Grammar of the Lexer

```
lexer grammar hedslLexer;
/*
Lexer
*/
TVAR : 'var';
TAT: '@';
TPARAMS: '#';
TINPUT : 'input';
OUTPUT: 'output';
RETURN : 'return';
DEFFUNCTION: 'fun';
IF : 'if';
FOR: 'for';
THEN : 'then';
SERVER: 'server';
ELSE: 'else';
FROM: 'from';
PUBLIC: 'plain';
TTO: '=>';
SEMICOLON: ';';
COLON: ':';
TSIF: '?';

TWODOTS: '..';

SHIFTL: '<<';
SHIFTR: '>>';
EQUAL: '=';
CEQ: '==';
CNE: '!=';
CLT: '<';
CLE: '<=';
CGT: '>';
CGE: '>=';
LPAREN: '(';
UNDERSCORE: '_';
RPAREN: ')';
LBRACE: '{';
RBRACE: '}';
LSBRACE: '[';
RSBRACE: ']';
OR: '||';
AND: '&&';
PIPE: '|';
DOT: '.';
COMMA: ',';
PLUS: '+';
MINUS: '-';
MUL: '*';
DIV: '/';
```

```
POW: '**';
NOT: '!';
MODDIV: '%';
/**
    Types
*/
TYPEAUTO: 'auto';
TYPEINT: 'int';
TYPEFLOAT: 'float';

COMMENT: '/*' .*? '*/' -> skip;
LINE_COMMENT: '//' ~[\r\n]* -> skip;

IDENTIFIER: [a-zA-Z][a-zA-Z0-9]*;
INTEGERLIT: [0-9]+;
FLOATLIT: [0-9]+'.'[0-9]+;

NEWLINE     : ('\r'? '\n' | '\r')+  -> skip;
TAB         : ('t' | '        ' | '    ' ) ->skip ;
WHITESPACE  : ' '+  -> skip;
```

Listing A.2: ANTLR4 Grammar of the Parser

```
parser grammar hedslParser;

options {
    tokenVocab = hedslLexer;
}

program : stmts EOF;

stmts : stmt+
/*    | stmts stmt*/
        ;

stmt : TVAR var_decl SEMICOLON #VarDeclStmt
        | input_def SEMICOLON #InputStmt
        | ident (LSBRACE index=expr RSBRACE)* EQUAL value=expr
    SEMICOLON# VarAssignment
        | func_decl #FuncDeclStmt
        | if_stmt  #IfStmt
        | for_decl #ForExpressionStmt
        | RETURN expr SEMICOLON #ReturnStmt
        | OUTPUT ident EQUAL expr (TTO ident)? key_arg?
    SEMICOLON? #OutputStmt
        ;

block : LBRACE stmts RBRACE
        | LBRACE RBRACE
        ;

var_decl : TVAR? PUBLIC? ident COLON type_ident
            | TVAR? PUBLIC? ident (COLON type_ident)? EQUAL
    expr
            ;

input_def : PUBLIC? TINPUT ident COLON type_ident (CLE ident)?
    (TPARAMS input_params)? ;
```

```

input_params : IDENTIFIER EQUAL (IDENTIFIER|numeric)
               | input_params COMMA IDENTIFIER EQUAL (IDENTIFIER|numeric);

func_decl : DEFFUNCTION ident LPAREN func_decl_args RPAREN COLON type_ident block;

for_decl : FOR LPAREN ident (COMMA ident)? COLON expr RPAREN block;

func_decl_args : /*blank*/
             | var_decl
             | func_decl_args COMMA var_decl
             ;

ident : IDENTIFIER
        ;
type_ident: TYPEINT CLT INTEGERLIT CGT type_args* key_arg?
        | TYPEAUTO type_args* key_arg?
        | TYPEFLOAT CLT INTEGERLIT CGT type_args* key_arg? ;

key_arg: TAT IDENTIFIER;

type_args: LSBRACE INTEGERLIT RSBRACE ;

numeric : INTEGERLIT
        | FLOATLIT
        ;

expr : LPAREN expr RPAREN # SubExpression
        | ident LPAREN call_args RPAREN # FunctionCall
        | ident # VarUsage
        | LSBRACE expr TWODOTS expr RSBRACE # RangeExpression
        | ident (LSBRACE expr RSBRACE)+ # ArrayAcces
        | ident DOT ident LPAREN call_args RPAREN # MethodCall
        /* Literals */
        | LSBRACE call_args? RSBRACE # ArrayLiteral
        | numeric # NumericLiteral
        | NOT expr # NotExpression
        | expr DIV expr # DivExpression
        | expr POW expr # PowerExpression
        | expr MUL expr # MultExpression
        | expr PLUS expr # PlusExpression
        | expr MINUS expr # MinusExpression
        | expr MODDIV expr # ModDivExpression
        | expr AND expr # AndExpression
        | expr OR expr # OrExpression
        | expr comparison expr # ComparisonExpression
        | expr TSIF expr COLON expr # IfExpression
        | input_def # InputExpression
        ;


if_stmt : IF LPAREN expr RPAREN block ELSE block;

call_args : /*blank*/
             | expr
```

```
                | call_args COMMA expr
                ;

comparison : CEQ | CNE | CLT | CLE | CGT | CGE;
```

B. Measurement Results of the Evaluation

Table B.1.: Evaluation Results in *s*

Sets	Keys	Compile Time	Evaluation Time	IO Time	Number of Nodes
4	0	0.005	0.031	0.214	20.000
4	1	0.006	0.043	0.327	28.000
4	2	0.006	0.037	0.252	24.000
4	4	0.006	0.034	0.215	22.000
4	5	0.006	0.034	0.215	22.000
10	0	0.006	0.078	0.381	50.000
10	1	0.006	0.108	0.716	70.000
10	2	0.006	0.092	0.531	60.000
10	5	0.006	0.084	0.419	54.000
10	10	0.006	0.081	0.382	52.000
250	0	0.053	1.935	7.053	1250.000
250	1	0.062	2.834	16.317	1750.000
250	2	0.059	2.350	11.660	1500.000
250	5	0.057	2.115	8.876	1350.000
250	250	0.056	1.946	7.057	1252.000
500	0	0.168	3.865	14.007	2500.000
500	1	0.197	5.668	32.577	3500.000
500	2	0.187	4.753	23.249	3000.000
500	5	0.179	4.278	17.701	2700.000
500	500	0.177	3.877	14.008	2502.000
750	0	0.354	5.875	20.963	3750.000
750	1	0.412	8.479	48.853	5250.000
750	2	0.395	7.220	34.851	4500.000
750	5	0.374	6.433	26.473	4050.000
750	750	0.364	5.862	20.958	3752.000
1000	0	0.606	7.782	27.904	5000.000
1000	1	0.698	11.388	65.008	7000.000
1000	2	0.668	9.791	46.455	6000.000
1000	5	0.641	8.672	35.278	5400.000
1000	1000	0.622	7.874	27.904	5002.000

Table B.2.: Evaluation Results: Compiler-Stage Runtimes in *μs*

Sets	Keys	Frontend	Type-Deduction	Key-Selection	PRE-Insert	Metrics-pass
4	0	1454.7	36.0	20.5	7.3	21.1
4	1	1659.4	35.1	20.9	16.9	29.4
4	2	1665.5	34.9	22.9	13.4	25.9
4	4	1662.2	35.7	21.2	12.1	22.6
4	5	1812.5	33.2	22.4	12.1	22.2
10	0	1699.6	71.7	58.8	17.2	59.5
10	1	1976.1	72.5	59.7	39.7	83.0
10	2	1991.9	70.3	60.5	31.1	69.7
10	5	1983.6	70.0	60.5	23.8	64.0
10	10	1973.3	72.7	60.0	22.0	62.1
250	0	10406.0	11760.0	11773.3	427.4	11963.1
250	1	13440.3	11729.8	11752.7	939.0	16697.0
250	2	13812.3	11790.2	11794.8	695.3	14474.0
250	5	13606.2	11650.5	11687.5	543.1	12895.0
250	250	13474.7	11701.1	11709.5	431.5	11924.9
500	0	19956.0	46065.4	46259.7	957.9	46601.2
500	1	25956.1	46392.8	46545.5	2045.8	66204.1
500	2	26575.6	46584.4	46690.5	1511.2	56891.9
500	5	26376.6	45954.4	46177.2	1280.7	50515.1
500	500	25951.9	46885.2	47169.3	967.8	47503.4
750	0	29797.7	103834.0	104299.1	1650.3	104884.7
750	1	38905.2	104112.9	104913.9	3344.2	148322.8
750	2	39926.5	105702.8	106275.5	2562.4	128874.1
750	5	38740.9	104033.0	104619.4	2020.2	113880.7
750	750	38835.1	103738.1	104519.0	1693.4	104946.6
1000	0	39709.7	182995.1	183850.1	2486.8	184692.8
1000	1	51286.9	183344.8	184010.3	4537.6	260199.1
1000	2	53284.6	185355.5	186512.0	3837.6	224603.4
1000	5	51517.0	184941.3	186193.4	3004.3	202083.4
1000	1000	51739.8	184037.3	185076.9	2544.3	185751.5

Table B.3.: Evaluation Results: Key-Generation and Encryption Runtimes in *s*

Sets	Keys	Key-Generation Runtime	Encryption Runtime
4	0	0.110	0.191
4	1	0.280	0.190
4	2	0.195	0.190
4	4	0.156	0.192
4	5	0.152	0.190
10	0	0.109	0.431
10	1	0.538	0.431
10	2	0.320	0.432
10	5	0.193	0.431
10	10	0.153	0.435
250	0	0.110	10.127
250	1	10.656	10.111
250	2	5.391	10.077
250	5	2.210	10.092
250	250	0.153	10.113
500	0	0.110	20.110
500	1	21.309	20.087
500	2	10.807	20.110
500	5	4.356	20.165
500	500	0.154	20.087
750	0	0.111	30.261
750	1	31.891	30.203
750	2	15.924	30.200
750	5	6.514	30.204
750	750	0.153	30.337
1000	0	0.111	40.279
1000	1	43.055	40.479
1000	2	22.121	40.257
1000	5	8.530	40.167
1000	1000	0.153	40.368

Table B.4.: Evaluation Results: Results of the Execution Runtime in *s*

Sets	Keys	Key-Loading	Input-Loading	Evaluation	Storing Ooutputs	Number of PRE
4	0	0.102	0.111	0.031	0.001	0.000
4	1	0.215	0.111	0.043	0.001	8.000
4	2	0.140	0.111	0.037	0.001	4.000
4	4	0.103	0.111	0.034	0.001	2.000
4	5	0.103	0.111	0.034	0.001	2.000
10	0	0.102	0.278	0.078	0.001	0.000
10	1	0.437	0.278	0.108	0.001	20.000
10	2	0.252	0.278	0.092	0.001	10.000
10	5	0.140	0.278	0.084	0.001	4.000
10	10	0.103	0.278	0.081	0.001	2.000
250	0	0.102	6.949	1.935	0.001	0.000
250	1	9.368	6.948	2.834	0.001	500.000
250	2	4.712	6.947	2.350	0.001	250.000
250	5	1.927	6.948	2.115	0.001	100.000
250	250	0.103	6.953	1.946	0.001	2.000
500	0	0.102	13.904	3.865	0.001	0.000
500	1	18.678	13.898	5.668	0.001	1000.000
500	2	9.357	13.892	4.753	0.001	500.000
500	5	3.795	13.905	4.278	0.001	200.000
500	500	0.103	13.904	3.877	0.001	2.000
750	0	0.103	20.859	5.875	0.001	0.000
750	1	28.004	20.848	8.479	0.001	1500.000
750	2	14.011	20.839	7.220	0.001	750.000
750	5	5.636	20.836	6.433	0.001	300.000
750	750	0.103	20.853	5.862	0.001	2.000
1000	0	0.103	27.800	7.782	0.001	0.000
1000	1	37.215	27.792	11.388	0.001	2000.000
1000	2	18.671	27.784	9.791	0.001	1000.000
1000	5	7.491	27.786	8.672	0.001	400.000
1000	1000	0.104	27.800	7.874	0.001	2.000