ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Automated Noise Guided Circuit Analysis & Optimisation for FHE

Master Thesis

Moritz Winger

Monday 13th September, 2021

Advisors: Alexander Viand, Prof. Dr. Kenny Paterson

Applied Cryptography Group
Institute of Information Security
Department of Computer Science, ETH Zürich

Abstract

Fully Homomorphic Encryption (FHE) allows computations on encrypted data, but applying FHE efficiently and securely is a challenging task requiring deep understanding of the underlying cryptographic protocols. Therefore, there is a need to make FHE accessible to the general user. Recently, FHE compilers have been developed aiming to automate homomorphic evaluation of user-written programs. FHE encryption introduces an error term called *noise* which grows during operations. We aim to minimize this error term, since handling larger noise requires larger instances which lead to slower evaluation of FHE computations. In this work, we apply noise heuristics from [18] and [10] to guide automatic rewriting of arithmetic circuits and study the impact of circuit rewriting on computational efficiency of homomorphic evaluation. In particular, a set of algorithms to insert operations that reduce computational complexity of evaluation in an arithmetic circuit based on decisions guided by noise heuristics has been developed and tested. We find that the proposed optimisation strategy's effects on computational runtimes are highly dependent on the nature of the considered arithmetic circuit.

Contents

Chapter 1

Introduction

In recent years the use of cloud computing services has become increasingly prevalent. A significant number of businesses and organisations have moved data storage and services to the cloud, generating a need for security and confidentiality of outsourced data. While standard encryption can protect data during transit and while at rest, data must generally be decrypted before computation is possible. Since this undermines the protection offered, the need for techniques to perform computations on encrypted data has arisen.

Fully Homomorphic Encryption (FHE) allows third parties to perform arbitrary computations on encrypted data, eliminating the need to decrypt the data and expose it to potential security risk while in use. FHE has long been considered impossible to achieve, until a first feasible scheme was proposed by Craig Gentry in 2009 [14]. In the last decade, homomorphic encryption has evolved from a theoretical concept to reality with a variety of practical open-source implementations available.

These technological advances have made a wide selection of applications possible, such as implementations of FHE in privacy-preserving genome analysis [20] and mobile applications [24]. Recently, FHE has been applied in the implementation of a new feature of the Microsoft *edge* browser, the Password Monitor, that notifies users if any of their saved passwords have been found in a third-party breach [26].

However, it remains a challenging task to build secure and efficient FHE calculations. Current FHE schemes are efficient if used correctly, but the gap between the highest possible efficiency achievable and what is practically achieved by the general user remains high. This is due to the fact that operations over encrypted values introduce *noise* into the ciphertext which, if not carefully managed, will grow to a point where decryption fails. Noise growth can be managed by rearranging the computation, as well as by introducing certain *maintenance* operations on encrypted data and choosing appropriate parameters. Maintenance operations consist of operations that are specific to FHE schemes and can be applied to a ciphertext to reduce its noise. It is this, highly application dependent, noise management and selection of parameters that makes developing an FHE application a complex and frequently tedious task. As a result, tools have emerged that aim to simplify this process in the form of FHE libraries and compilers that translate high level programs to low-level FHE programs [30].

Existing work takes one of two approaches to manage noise growth. The first uses heuristic bounds on the noise growth to try and automatically calculate the optimal parameters for a given computation. However, these estimates are currently still considerably too conservative and do not result in optimal performance. The second type of tools instead tries to automatically rearrange or rewrite the computation to achieve better noise growth characteristics. However, these tools either require the developer to manually choose parameters or use simplistic methods that lead to very inefficient choices.

This thesis aims to improve noise management for the inexperienced user, by combining both approaches into a more complete and holistic solution. While the potential for improvement in heuristics used for parameter selection seems to be limited [10], existing solutions consider a fixed computation and do not consider rearrangements. Similarly, while existing rewriting-based tools have limited impact in real-world programs [2, 30], they use either no or overly simplistic heuristics of noise growth.

Intuitively, the interdependence of parameters and applications suggests that there might be further optimization opportunities by considering both holistically. As a first step one could use heuristics to identify and select parts of the computation that would most benefit from rewriting. This leads us to pose our main research question: Can hybrid constructions offer benefits beyond what is possible by applying these approaches independently? The work conducted in this thesis extends the Automated Batching Compiler (ABC) framework for the homomorphic evaluation of arithmetic circuits by a novel automatic circuit rewriting functionality, namely the automatic insertion of so-called *modulus switching* operations based on noise information.

Specifically, noise heuristic calculations are used to identify regions of significant noise growth present in a given arithmetic circuit. Based on these computations, the above strategy has been pursued to rewrite the circuit to improve computational efficiency while ensuring correctness of the computation. Further, based on previous work on rewriting circuits to reduce multiplicative depth, we apply noise heuristics to identify regions, where rewriting the computation would be most beneficial to reduce noise growth or increase computational efficiency.

Goals of this thesis FHE encryption introduces a noise term that increases when performing computations. We aim to use the quantity of noise to aid a general user to design computation circuits optimally and to develop an automatic circuit rewriting framework that, based on noise-guided decisions, produces a semantically equivalent circuit that can be evaluated with greater computational efficiency. Specifically, we aim to implement noise-heuristic calculations to efficiently analyse a circuit in terms of noise properties and use this information on the circuit to automatically rewrite the circuit by performing modulus switching at appropriate points in the computation.

Outline In Chapter 2, we discuss the mathematical background underlying FHE schemes, basic notions and definitions are presented. We continue to discuss the specifics of Simple Encrypted Arithmetic Library (SEAL) and present theoretical bounds on ciphertext noise arising through homomorphic operations. Then, we show the most recent results on noise heuristics, which we shall later use as a metric to study noise growth in given arithmetic circuits. Chapter 3 presents the specifics of the ABC framework and we present algorithms to rewrite circuits to improve performance using noise heuristics, and the additions introduced throughout the current project. In Chapter 4, we measure their effectiveness in terms of noise growth and computational performance.

Chapter 2

Background

In this section we present the mathematical background of a *somewhat* homomorphic encryption scheme based on the Ring Learning With Errors (RLWE) problem [23]. A somewhat homomorphic encryption scheme is a scheme, where addition and multiplication are indeed homomorphic operations but it is only possible to evaluate functions of limited complexity, whereas a Fully Homomorphic Encryption (FHE) scheme alllows the evaluation of arbitrary functions. We describe the Brakerski/Fan-Vercauteren (BFV) scheme presented in [13] and its variant implemented in the Simple Encrypted Arithmetic Library (SEAL) library for homomorphic encryption [22].

The first Fully Homomorphic Encryption (FHE) scheme proposed by Gentry [14] was based on a shortest vector problem of an ideal lattice. The hardness of the RLWE problem can be related to the shortest vector problem on ideal lattices. Gentry first proposed a *somewhat* homomorphic encryption scheme, and then made it fully homomorphic by introducing the *bootstrapping* technique to reduce ciphertext noise and be able to continue computations on said ciphertext. In practice, since the bootstrapping procedure requires significant computational effort, it is rarely used. Instead, *levelled* FHE is used, supporting the evaluation of arbitrary circuits composed of multiple types of gates of bounded (pre-determined) depth.

2.1 Basic Notation and Preliminaries

The mathematical object under consideration is the ring $R = \mathbb{Z}/(f(x))$, with $f(x)$ a monic irreducible polynomial of degree d with coefficients in $\mathbb{Z}$. In most Fully Homomorphic Encryption (FHE) schemes, the polynomial $f(x)$ is chosen to be the m-th cyclotomic polynomial $\Phi_m(x)$. Here, we choose to take $f(x) = x^d + 1$ with $d = 2^n$ for a fixed positive integer n. Let $\mathbf{a} =$

$\sum_{i=0}^{d-1} a_i x^i \in R$. We define the *infinity norm* of the polynomial a as

$$\|a\| := \max_{i \in \{0,\dots,d-1\}} \{a_i\} \tag{2.1}$$

and the *expansion factor* of the ring R as

$$\delta_R := \max_{\mathbf{a},\mathbf{b} \in R} \{\|\mathbf{a} \cdot \mathbf{b}\| / \|\mathbf{a}\| \cdot \|\mathbf{b}\|\}. \tag{2.2}$$

Let $q > 1$ be an integer and denote with $\mathbb{Z}_q$ the finite set of integers $\mathbb{Z}/q\mathbb{Z}$. We denote by R_q the set of polynomials in R with coefficients in $\mathbb{Z}_q$. Further, we denote by $[a]_q$ the image of an integer a under the projection map

$$\pi : \mathbb{Z} \longrightarrow \mathbb{Z}_q; \quad a \mapsto [a]_q := a \mod q. \tag{2.3}$$

For an element $\mathbf{a} \in R$, we denote by $[\mathbf{a}]_q$ as the element in R obtained by reducing all coefficients a_i ($i \in 0,\dots,d-1$) modulo q. Let $x \in \mathbb{R}$. We shall denote rounding x to the nearest integer by $\lfloor x \rceil$, and $\lfloor x \rfloor$, $\lceil x \rceil$ to denote rounding down or up.

From an algebraic point of view, it is often convenient to view the ring R as the ring of integers $\mathcal{O}_{\mathbb{Q}(\zeta_m)}$ of the m-th cyclotomic field $\mathbb{Q}(\zeta_m)$, where ζ_m is a primitive m-th root of unity. This ring has a $\mathbb{Z}$-basis given by $\{1, \zeta, \dots, \zeta^{d-1}\}$ (see [25] for a proof). An *embedding* of a cyclotomic number field $\mathbb{Q}(\zeta_m)$ is a ring homomorphism $\sigma_i : \mathbb{Q}(\zeta_m) \longrightarrow \mathbb{C}$ fixing the elements of $\mathbb{Q}$. The field $\mathbb{Q}(\zeta_m)$ has precisely d embeddings, that come in complex conjugate pairs. The embeddings are defined on their action on the powers of ζ_m. We shall assume that σ_1 is the identity map. We define the *canonical embedding* $\sigma : \mathbb{Q}(\zeta_m) \longrightarrow \mathbb{C}^d$ of an element $\mathbf{a} \in \mathbb{Q}(\zeta_m)$ as the d-dimensional vector given by

$$\sigma(\mathbf{a}) = (\sigma_1(\mathbf{a}), \dots, \sigma_d(\mathbf{a})). \tag{2.4}$$

We define the *canonical embedding norm* of the ring element $\mathbf{a} \in R$ as

$$\|\mathbf{a}\|^{\text{can}} := \max_{i \in \{1,\dots,d\}} \|\sigma_i(\mathbf{a})\|, \tag{2.5}$$

where $\|\cdot\|$ denotes the infinity norm as defined above. The canonical embedding norm has the properties, that for any polynomials $\mathbf{a}, \mathbf{b} \in R$, $\|\mathbf{a}\| \leq \|\mathbf{a}\|^{\text{can}}$ and $\|\mathbf{a} \cdot \mathbf{b}\|^{\text{can}} \leq \|\mathbf{a}\|^{\text{can}} \cdot \|\mathbf{b}\|^{\text{can}}$.

For a probability distribution $\mathcal{D}$, we denote by $x \leftarrow \mathcal{D}$, whenever x is sampled from $\mathcal{D}$. Similarly, given a set S, if x is sampled uniformly from S, we write $x \leftarrow S$. To define a distribution χ on the ring R, we use a discrete Gaussian distribution $D_{\mathbb{Z},\tilde{\sigma}}$, where $\tilde{\sigma}$ denotes the variance.

2.2 The RLWE Problem

We briefly introduce the Ring Learning With Errors (RLWE) problem, which underpins most modern FHE schemes.

Definition 2.1 (Decision-RLWE) *Let $f(x)$ be a m-th cyclotomic polynomial of degree $d = \varphi(m)$, where φ denotes the Euler totient function. Consider the ring $R = \mathbb{Z}[x]/(f(x))$. Let $q \geq 2$ be an integer and let $\mathbf{s} \in R_q$ be a random element. Let χ be a probability distribution over R and denote with $A^{(q)}_{\mathbf{s},\chi}$ the distribution obtained by choosing an element $\mathbf{a} \leftarrow R$ uniformly at random and a noise term $\mathbf{a} \leftarrow \chi$ and outputting $(\mathbf{a}, [\mathbf{a} \cdot \mathbf{s} + \mathbf{e}]_q)$. Then, the problem of distinguishing between the distribution $A^{(q)}_{\mathbf{s},\chi}$ and the uniform distribution $U(R_q^2)$ is called the decision RLWE$_{d,q,\chi}$ problem.*

It has been shown that the RLWE problem can be reduced to the shortest vector problem over ideal lattices, a polynomial-time reduction from the shortest-vector problem to the RLWE problem has been presented in [23].

The decision-RLWE is used to define to modern homomorphic encryption schemes as we shall see in the following.

2.3 The Brakerski/Fan-Vercauteren Scheme

We provide a brief description of the Brakerski/Fan-Vercauteren (BFV) scheme, both in its original "textbook" variant as described in [5, 13] and the slightly modified version used in Microsoft's Simple Encrypted Arithmetic Library.

2.3.1 Textbook BFV

We define an RLWE-based encryption scheme implementing *somewhat* homomorphic encryption scheme following the exposition in [13].

Definition 2.2 (RLWE-encryption scheme) *Let $t > 1$ be an integer and define $\Delta := \lfloor q/t \rfloor$. Denote with $r_t(q) = q \mod t$. Fix $\tilde{\sigma} = 3.2$ as the variance of the distribution χ. We define the RLWE-encryption scheme consisting of:*

- *Secret Key Generation: sample $\mathbf{s} \leftarrow \chi$ and output* $\texttt{sk}= \mathbf{s}$.
- *Public Key Generation: set $\mathbf{s} = \texttt{sk}$, sample $\mathbf{a} \leftarrow R_q$, $\mathbf{e} \leftarrow \chi$ and return*

$$\texttt{pk} = ([-(\mathbf{a} \cdot \mathbf{s} + \mathbf{e})]_q, \mathbf{a}). \tag{2.6}$$

- *Encryption: This procedure takes as input a message $m \in R_t$ and the public key* $\texttt{pk}$. *Let $\mathbf{p}_0 := \texttt{pk}[0]$ and let $\mathbf{p}_1 := \texttt{pk}[1]$. Sample $\mathbf{u}, \mathbf{e}_1, \mathbf{e}_2 \leftarrow \chi$ and output*

$$\texttt{ct} = \left([\mathbf{p}_0 \cdot \mathbf{u} + \mathbf{e}_1 + \Delta \cdot \mathbf{m}]_q, [\mathbf{p}_1 \cdot \mathbf{u} + \mathbf{e}_2]_q\right). \tag{2.7}$$

- *Decryption: Take as an input the secret key* sk *and a ciphertext* ct *and set* $\mathbf{s} = \mathtt{sk}$, *let* $\mathbf{c}_0 := \mathtt{ct}[0]$ *and* $\mathbf{c}_1 := \mathtt{ct}[1]$. *The decryption procedure computes:*

$$\left[\left\lfloor \frac{t \cdot [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_q}{q} \right\rceil\right]_t . \tag{2.8}$$

The parameters for the RLWE encryption scheme are chosen according to a security parameter λ. In the following, we refer to the parameter q as the *coefficient modulus* and the parameter t will be called the *plaintext modulus*.

The RLWE encryption scheme defined above has been shown to be semantically secure ([23]) and serves as the basis for most modern FHE schemes. Correctness follows from the next lemma.

Lemma 2.3 *Considering the above encryption scheme and assuming that there exists an integer B such that* $\|\chi\| < B$, *then*

$$[\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}]_q = \Delta \cdot \mathbf{m} + \mathbf{v} \tag{2.9}$$

with $\|v\| \leq \delta_R \cdot B^2 + B$. *This implies that if* $2 \cdot \delta_R \cdot B^2 + B < \Delta/2$, *the ciphertext decrypts correctly.*

Proof Using the definition of the BFV scheme, we can write

$$\begin{aligned} \mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} &= \mathbf{p}_0 \cdot \mathbf{u} + \mathbf{e}_1 + \Delta \cdot \mathbf{m} + \mathbf{p}_1 \cdot \mathbf{s} \cdot \mathbf{u} + \mathbf{e}_2 \cdot \mathbf{s} \mod q \\ &= \Delta\mathbf{m} + (\mathbf{p}_0 + \mathbf{p}_1 \cdot \mathbf{s}) \cdot \mathbf{u} + \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{s} \mod q \\ &= \Delta\mathbf{m} + \mathbf{e} \cdot \mathbf{u} + \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{s} \mod q. \end{aligned} \tag{2.10}$$

Setting $\mathbf{v} := \mathbf{e} \cdot \mathbf{u} + \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{s}$, and since $\mathbf{e}, \mathbf{e}_1, \mathbf{e}_2, \mathbf{u}, \mathbf{s} \leftarrow \chi$, we obtain the bound

$$\|\mathbf{v}\| \leq 2 \cdot \delta_R \cdot B^2 + B. \tag{2.11}$$

Let $\varepsilon := q/t - \Delta < 1$. Writing

$$\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} = \Delta \cdot \mathbf{m} + \mathbf{v} + q \cdot \mathbf{r} \tag{2.12}$$

for a polynomial $\mathbf{r}$ with integer coefficients and multiplying by the factor t/q we get

$$\frac{t}{q}(\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s}) = \mathbf{m} + \frac{t}{q} \cdot (\mathbf{v} - \varepsilon \cdot \mathbf{m}) + t \cdot \mathbf{r}. \tag{2.13}$$

In order for the rounding to be correct and decryption to yield the correct plaintext $\mathbf{m}$, we require

$$\frac{t}{q} \cdot \|\mathbf{v} - \varepsilon \cdot \mathbf{m}\| < \frac{1}{2}. \tag{2.14}$$

Since $\mathbf{m} \in R_t$, the given bound follows. $\square$

Definition 2.4 (Ciphertext Noise) *The term*

$$\mathbf{v} = \Delta\mathbf{m} + \mathbf{e} \cdot \mathbf{u} + \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{s} \tag{2.15}$$

is called the noise contained in the ciphertext.

To derive the BFV scheme from the encryption scheme in Definition 2.2, we will assume that the polynomials $\mathbf{s}$ and $\mathbf{u}$ are sampled from R_2, thus simplifying the bound from Lemma 2.3 to $\|v\| \leq B \cdot (2 \cdot \delta_R + 1)$.

We note that with Equation (2.9), we can interpret the elements of a ciphertext $\mathtt{ct}$ as the coefficients of a polynomial

$$\mathtt{ct}(x) = \mathbf{c}_0 + \mathbf{c}_1(x). \tag{2.16}$$

Evaluating this polynomial in $\mathbf{s}$ yields

$$[\mathtt{ct}(\mathbf{s})]_q = \Delta \cdot \mathbf{m} + \mathbf{v}, \tag{2.17}$$

from which it is possible to recover the plaintext $\mathbf{m}$. We shall use this representation of the ciphertext to derive the homomorphic operations of addition and multiplication.

Addition: Consider two ciphertexts $\mathtt{ct}_1(\mathbf{s})$ and $\mathtt{ct}_2(\mathbf{s})$. Since $[\mathtt{ct}_i(\mathbf{s})]_q = \Delta \cdot \mathbf{m}_i + \mathbf{v}_i$ for $i \in \{1, 2\}$, the addition of the two ciphertexts evaluates to

$$[\mathtt{ct}_1(\mathbf{s}) + \mathtt{ct}_2(\mathbf{s})]_q = \Delta \cdot [\mathbf{m}_1 + \mathbf{m}_2]_t + \mathbf{v}_1 + \mathbf{v}_2 - \varepsilon \cdot t \cdot \mathbf{r}, \tag{2.18}$$

where $\varepsilon = q/t - \Delta < 1$ and $\mathbf{m}_1 + \mathbf{m}_2 = [\mathbf{m}_1 + \mathbf{m}_2]_t + t \cdot \mathbf{r}$ with $r \in R$ such that $\|r\| \leq 1$. We therefore see, that the noise grows additively by a maximum of t when adding two ciphertexts. We define the addition operation on ciphertexts:

$$\mathtt{Add}(\mathtt{ct}_1, \mathtt{ct}_2) := ([\mathtt{ct}_1[0] + \mathtt{ct}_2[0]]_q, [\mathtt{ct}_1[1] + \mathtt{ct}_2[1]]_q). \tag{2.19}$$

Multiplication: Multiplying two ciphertexts of the form $[\mathtt{ct}_i(\mathbf{s})]_q = \Delta \cdot \mathbf{m}_i + \mathbf{v}_i$ introduces a non-linear term, resulting in a ciphertext that consists of three ring elements instead of two. This issue can be resolved by a *relinearisation* step.

Consider two ciphertexts $\mathtt{ct}_i(x), i \in \{1, 2\}$ and their evaluation in $\mathbf{s}$ in R

$$\mathtt{ct}_i(\mathbf{s}) = \Delta \cdot \mathbf{m}_i + \mathbf{v}_i + q \cdot \mathbf{r}_i. \tag{2.20}$$

Multiplying those expression yields

$$\begin{aligned}(\mathtt{ct}_1 \cdot \mathtt{ct}_2)(\mathbf{s}) =& \Delta^2 \cdot \mathbf{m}_1 \cdot \mathbf{m}_2 + \Delta \cdot (\mathbf{m}_1 \cdot \mathbf{v}_2 + \mathbf{m}_2 \cdot \mathbf{v}_1) \\ &+ q \cdot (\mathbf{v}_1 \cdot \mathbf{r}_2 + \mathbf{v}_2 \cdot \mathbf{r}_1) \\ &+ \mathbf{v}_1 \cdot \mathbf{v}_2 + q \cdot \Delta \cdot (\mathbf{m}_1\mathbf{r}_2 + \mathbf{m}_2 \cdot \mathbf{r}_1) + q^2 \cdot \mathbf{r}_1 \cdot \mathbf{r}_2.\end{aligned} \tag{2.21}$$

This shows that the product of two ciphertexts needs to be scaled by a factor $1/\Delta$ in order to obtain a ciphertext that decrypts to the product of plaintexts $[\mathbf{m}_1 \cdot \mathbf{m}_2]_t$. However, since Δ does not necessarily divide q, scaling by $1/\Delta$ could potentially introduce significant noise due to rounding error. Therefore, in the BFV scheme, it is chosen to scale by t/q and for a product of ciphertexts of the form $\mathtt{ct}_1(x) \cdot \mathtt{ct}_2(x) = \mathbf{c}_0 + \mathbf{c}_1 \cdot x + \mathbf{c}_2 \cdot x^2$, the approximation

$$\frac{t}{q} \cdot (\mathtt{ct}_1 \cdot \mathtt{ct}_2)(\mathbf{s}) = \lfloor t \cdot \mathbf{c}_0/q \rceil + \lfloor t \cdot \mathbf{c}_1/q \rceil \cdot \mathbf{s} + \lfloor t \cdot \mathbf{c}_2/q \rceil \cdot \mathbf{s}^2 + \mathbf{r}_a \tag{2.22}$$

is used, introducing an approximation error $\mathbf{r}_a$ with $\|\mathbf{r}_a\| < (\delta_R \cdot \|\mathbf{s}\| + 1)^2/2$.

The following lemma shows the effect of multiplying two ciphertexts on noise growth:

Lemma 2.5 *Let* $\mathtt{ct}_i$ $(i \in \{1,2\})$ *be two ciphertexts, where* $[\mathtt{ct}_i(\mathbf{s})]_q = \Delta \cdot \mathbf{m}_i + \mathbf{v}_i$ *and suppose there exists an integer E, such that* $\|\mathbf{v}_i\| < E < \Delta/2$. *Let* $\mathtt{ct}_1(x) \cdot \mathtt{ct}_2(x) = \mathbf{c}_0 + \mathbf{c}_1 \cdot x + \mathbf{c}_2 \cdot x^2$. *Then,*

$$[\lfloor t \cdot \mathbf{c}_0/q \rceil + \lfloor t \cdot \mathbf{c}_1/q \rceil \cdot \mathbf{s} + \lfloor t \cdot \mathbf{c}_2/q \rceil \cdot \mathbf{s}^2]_q = \Delta \cdot [\mathbf{m}_1 \cdot \mathbf{m}_2]_t + \mathbf{v}_3, \tag{2.23}$$

where $\|\mathbf{v}_3\| < 2 \cdot \delta_R \cdot t \cdot E \cdot (\delta_R \cdot \|\mathbf{s}\| + 1) + 2 \cdot t^2 \cdot \delta_R^2 \cdot (\|\mathbf{s}\| + 1)^2$.

Proof See [23] for a detailed proof. □

This lemma shows that the noise is multiplied by a factor of approximately $2 \cdot t \cdot \delta_R^2 \cdot \|\mathbf{s}\|$ upon multiplication.

Relinearisation: When multiplying two plaintexts, the number of elements in the ciphertext increases. *Relinearisation* is a procedure that takes a degree 2 ciphertext polynomial and reduces it again to a degree 1 ciphertext polynomial. This step requires a relinearisation key, sometimes also referred to as the evaluation key.

Let $\mathtt{ct} = [\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2]$ denote a degree 2 ciphertext. Then, it is the goal to find a ciphertext $\mathtt{ct'} = [\mathbf{c}'_0, \mathbf{c}'_1]$ such that

$$[\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} + \mathbf{c}_2 \cdot \mathbf{s}^2]_q = [\mathbf{c}'_0 + \mathbf{c}'_1 \cdot \mathbf{s} + \mathbf{r}]_q, \tag{2.24}$$

where $\|r\|$ is small.

Sampling $\mathbf{a}_0 \leftarrow R_q$ and $\mathbf{e}_0 \leftarrow \chi$, we define the relinearisation key

$$\mathtt{rlk} := \left[\left([-(\mathbf{a_0} \cdot \mathbf{s} + \mathbf{e}_0) + \mathbf{s}^2]_q, \mathbf{a}_0\right)\right]. \tag{2.25}$$

Setting

$$\mathbf{c}'_0 = \mathbf{c}_0 + \mathtt{rlk}[0]\mathbf{c}_2 \tag{2.26}$$

and

$$\mathbf{c}_1' = \mathbf{c}_1 + \texttt{rlk}[1]\mathbf{c}_2 \tag{2.27}$$

gives

$$\begin{aligned}\mathbf{c}_0' + \mathbf{c}_1' \cdot \mathbf{s} &= \mathbf{c}_0 + \left(-(\mathbf{a} \cdot \mathbf{s} + \mathbf{e}_0) + \mathbf{s}^2\right) \mathbf{c}_2 + \mathbf{c}_1 \cdot \mathbf{s} + \mathbf{c}_2 \cdot \mathbf{s}^2 - \mathbf{e}_0 \cdot \mathbf{c}_2 \\ &= \mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} + \mathbf{c}_2 \cdot \mathbf{s}^2 - \mathbf{e}_0 \cdot \mathbf{c}_2,\end{aligned} \tag{2.28}$$

which is the desired expression except for the term $\mathbf{e}_0 \cdot \mathbf{c}_2$. This term, however, will very likely cause the decryption process to fail, since $\mathbf{c}_2$ has coefficients up to size q. This issue can be solved by writing $\mathbf{c}_2$ in terms of some small base T:

$$\mathbf{c}_2 = \sum_{i=0}^{\ell} T^i \cdot \mathbf{c}_2^{(i)} \mod q, \tag{2.29}$$

where $\ell = \lfloor \log_T q \rfloor$ and the coefficients of $\mathbf{c}_2^(i)$ are in R_T. Instead of a single relinearisation key pair we get $\ell + 1$ pairs

$$\texttt{rlk} := \left[\left([-(\mathbf{a_i} \cdot \mathbf{s} + \mathbf{e}_i) + T^i \cdot \mathbf{s}^2]_q, \mathbf{a}_i \right) : i \in \{0, \dots, \ell\} \right], \tag{2.30}$$

with $\mathbf{a}_i \leftarrow R_q$ and $\mathbf{e}_i \leftarrow \chi$. Defining

$$\mathbf{c}_0' = \left[\mathbf{c}_0 + \sum_{i=0}^{\ell} \texttt{rlk}[i][0]\mathbf{c}_2^{(i)} \right]_q \tag{2.31}$$

and

$$\mathbf{c}_1' = \left[\mathbf{c}_1 + \sum_{i=0}^{\ell} \texttt{rlk}[i][1]\mathbf{c}_2^{(i)} \right]_q \tag{2.32}$$

yields

$$\mathbf{c}_0' + \mathbf{c}_1' \cdot \mathbf{s} = \mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} + \mathbf{c}_2 \cdot \mathbf{s}^2 - \sum_{i=0}^{\ell} \mathbf{e}_i \cdot \mathbf{c}_2^{(i)} \mod q. \tag{2.33}$$

Hence, we have successfully replaced the large additive term that appeared in the naive approach above with a term of size linear in T. We note that the noise introduced by relinearisation is bounded by $(\ell + 1) \cdot B \cdot T\delta_R/2$.

2.3.2 BFV in Microsoft SEAL

In this section, we describe the differences in the encryption scheme implemented in the Simple Encrypted Arithmetic Library library [22] to BFV. In practice, some operations in SEAL are done differently, or in more generality than in textbook BFV. Further, we will elucidate how noise calculation is performed in SEAL and introduce the *noise budget* of a ciphertext.

Probability distribution In SEAL, the distribution χ is defined as the distribution on R_q obtained by choosing each coefficient of the polynomial from a discrete Gaussian distribution over $\mathbb{Z}$.

Plaintext and Ciphertext space Plaintext elements in SEAL are polynomials in R_t, precisely as in textbook BFV. Ciphertexts in SEAL are $k+1$-tuples of polynomials $\mathtt{ct} = (\mathbf{c}_0, \dots, \mathbf{c}_k)$ in the ring R_q of length at least 2. This is a generalisation of textbook BFV, where the ciphertexts are always of length two.

Encryption Scheme The encryption scheme in SEAL is the same as in textbook BFV, except for decryption, which is adapted to general ciphertexts of length k. A SEAL ciphertext $\mathtt{ct} = (\mathbf{c}_0, \dots, \mathbf{c}_k)$ is decrypted by computing

$$\left[\left\lfloor \frac{t}{q}[\mathtt{ct}(\mathbf{s})]_q \right\rceil\right]_t = \left[\left\lfloor \frac{t \cdot [\mathbf{c}_0 + \mathbf{c}_1 \cdot \mathbf{s} + \ldots + \mathbf{c}_k \cdot \mathbf{s}^k]_q}{q} \right\rceil\right]_t . \tag{2.34}$$

Addition Let $\mathtt{ct}_1 = (\mathbf{c}_0, \dots, \mathbf{c}_j)$ and $\mathtt{ct}_2 = (\mathbf{d}_0, \dots, \mathbf{d}_k)$ be two SEAL ciphertexts encrypting two plaintext polynomials $\mathbf{m}_1$ and $\mathbf{m}_2$, respectively. Suppose $j \leq k$. Then,

$$\mathtt{ct}_{\text{add}} = \left([\mathbf{c}_0 + \mathbf{d}_0]_q, \dots, [\mathbf{c}_j + \mathbf{d}_j]_q, \mathbf{d}_{j+1}, \dots, \mathbf{d}_k\right) \tag{2.35}$$

is an encryption of $[\mathbf{m}_1 + \mathbf{m}_2]_t$.

Multiplication Let $\mathtt{ct}_1 = (\mathbf{c}_0, \dots, \mathbf{c}_j)$ and $\mathtt{ct}_2 = (\mathbf{d}_0, \dots, \mathbf{d}_k)$ be two SEAL ciphertexts. Then, the ouput of the multiplication $\mathtt{ct}_{\text{mult}} = (\mathbf{C}_0, \dots, \mathbf{C}_{j+k})$ is a ciphertext of size $j+k$ where

$$\mathbf{C}_m = \left[\left\lfloor \frac{t}{q} \left(\sum_{r+s=m} \mathbf{c}_r \cdot \mathbf{d}_s \right) \right\rceil\right]_q . \tag{2.36}$$

Negation SEAL provides a function that computes the negation of a given ciphertext.

Plaintext-ciphertext operations SEAL provides functions that, given a ciphertext $\mathtt{ct}$ encrypting a plaintext $\mathbf{m} \in R_t$, and unencrypted plaintexts $\mathbf{m}_{\text{add}}$ and $\mathbf{m}_{\text{mult}}$, compute encryptions of $\mathbf{m} + \mathbf{m}_{\text{add}}$ and $\mathbf{m} \cdot \mathbf{m}_{\text{mult}}$, respectively:

Definition 2.6 (Plaintext-ciphertext addition) *Let $\mathtt{ct} = (\mathbf{x}_0, \dots, \mathbf{x}_j)$ be a ciphertext encrypting a plaintext $\mathbf{m}_1 \in R_t$ and let $\mathbf{m}_2 \in R_t$ be a plaintext polynomial. Then, the ciphertext encrypting the plaintext polynomial $\mathbf{m}_1 + \mathbf{m}_2$ is given by*

$$\mathtt{ct}_{padd} := (\mathbf{x}_0 + \Delta \cdot \mathbf{m}_2, \mathbf{x}_1, \dots, \mathbf{x}_j). \tag{2.37}$$

Definition 2.7 (Plaintext-ciphertext multiplication) *Let* $\texttt{ct} = (\mathbf{x}_0, \ldots, \mathbf{x}_j)$ *be a ciphertext encrypting a plaintext* $\mathbf{m}_1 \in R_t$ *and let* $\mathbf{m}_2 \in R_t$ *be a plaintext polynomial. Then, the ciphertext encrypting encrypting the plaintext polynomial* $\mathbf{m}_1 \cdot \mathbf{m}_2$

$$\texttt{ct}_{pmult} := (\mathbf{m}_2 \cdot \mathbf{x}_0, \ldots, \mathbf{m}_2 \cdot \mathbf{x}_j). \tag{2.38}$$

When one of the operands in either addition or multiplication does not need to be encrypted, these operations can be used to significantly improve performance over first encrypting the plaintexts $\mathbf{m}_{\text{add}}$ and $\mathbf{m}_{\text{mult}}$ and subsequently performing the homomorphic ciphertext addition or multiplication.

Relinearisation Relinearisation in SEAL works in the same way as in textbook BFV and takes as input a ciphertext of length k to produce a ciphertext of length $k-1$. Repeated application of the relinearisation procedure yields a ciphertext of length two.

Homomorphic operations in the Residue Number System (RNS) Operations on ciphertexts in a BFV scheme are performed on the ring R_q for an integer q. We therefore can write q as a product of primes $q = q_1 \cdots q_m$. By the Chinese Remainder Theorem (CRT), we have a ring isomorphism $R_q \cong R_{q_1} \times \cdots \times R_{q_m}$. Therefore, operations can be performed in the factors R_{q_i} separately. In SEAL, all the homomorphic operations over the polynomial coefficients ring is implemented based on the RNS arithmetic. A variety of optimization techniques, as proposed in [3], in low level arithmetic implementation adopted in SEAL improve performance of the FHE scheme significantly.
We recall that in order for the product of two ciphertexts to decrypt correctly, we need to scale the result of homomorphic multiplication by the factor $\Delta = \lfloor q/t \rfloor$. Since q is usually a large parameter, homomorphic multiplications are slow. It is faster to use a RNS to represent the integer q as the product of the prime numbers q_i.
Let $x \in R_q$ be given in CRT representation $(x_1, \ldots, x_k)$ and let $t \in \mathbb{Z}$ be the coefficient modulus. If we want to scale x by the factor Δ, we can use the following equality presented in [16]:

$$y := \left\lceil \frac{t}{q} \cdot x \right\rfloor = \left[\left\lceil \left(\sum_{i=1}^{k} x_i \cdot (\tilde{q}_i \cdot \frac{t}{q_i}) \right) \right\rfloor \right]_t, \tag{2.39}$$

where $\tilde{q}_i = \frac{q}{q_i} \in \mathbb{Z}$. Results presented in [16] suggest that this transformation reduces the runtime of homomomorphic multiplication significantly.

Key Sampling We have seen that in textbook BFV, the secret key is a polynomial uniformly sampled from R_2. In SEAL, however, the secret key is sampled from R_3.

Invariant Noise and Noise Budget Noise analysis in SEAL is similar to textbook BFV, however it must be extended to ciphertexts of arbitrary length.

Definition 2.8 (Invariant Noise: SEAL) *Let* $\mathtt{ct} = (\mathbf{c}_0, \dots, \mathbf{c}_k)$ *be a ciphertext of length* $k+1$ *that encrypts the plaintext* $\mathbf{m} \in R_t$*. The invariant noise* $\mathbf{w}$ *is the polynomial with the smallest infinity norm such that*

$$\frac{t}{q}\mathtt{ct}(\mathbf{s}) = \frac{t}{q}\left(\mathbf{c}_0 + \mathbf{c}_1 + \ldots + \mathbf{c}_k\mathbf{s}^k\right) = \mathbf{m} + \mathbf{w} + \mathbf{r} \cdot t, \tag{2.40}$$

for a polynomial $\mathbf{r}$ *with integer coefficients.*

We note that in the case of a BFV scheme, the polynomial $\mathbf{w}$ is precisely the polynomial $(t/q) \cdot (\mathbf{v} - \varepsilon \cdot \mathbf{m})$ from the proof of Lemma 2.3. The invariant noise describes the notion that if the noise $\mathbf{w}$ is not rounded correctly, decryption fails.

Lemma 2.9 *Decryption (Equation (2.34)) correctly decrypts a ciphertext* $\mathtt{ct}$ *encrypting a plaintext* $\mathbf{m}$*, if the invariant noise* $\mathbf{w}$ *satisfies*

$$\|\mathbf{w}\| < 1/2. \tag{2.41}$$

Proof The result follows directly from Equation 2.14 and the proof of Lemma 2.3. □

A further, sometimes more useful notion is the noise budget, a quantity that captures, how much noise there is left until decryption will fail:

Definition 2.10 (Noise Budget) *Let* $\mathbf{w}$ *be the invariant noise of a ciphertext* $\mathtt{ct}$ *encrypting the plaintext* $\mathbf{m} \in R_t$*. The noise budget of* $\mathtt{ct}$ *is defined as* $-\log_2(2 \cdot \|\mathbf{w}\|)$.

From this definition and Lemma 2.9 the next lemma follows immediately.

Lemma 2.11 *Let* $\mathtt{ct}$ *be a ciphertext decrypting a plaintext message* $\mathbf{m} \in R_t$*. Decryption as presented in Equation (2.34) correctly decrypts the ciphertext, if the noise budget of* $\mathtt{ct}$ *is positive.*

Noise calculation in SEAL: The noise budget is calculated in the function `Decryptor::invariant_noise_budget`. We recall from above that

$$\begin{aligned} \mathbf{c}_0 + \cdots + \mathbf{c}_{k-1} \cdot \mathbf{s}^{k-1} &= \Delta \cdot \mathbf{m} + \mathbf{v} + q \cdot \mathbf{r} \\ &= \left(\frac{q}{t} - \varepsilon\right) \cdot \mathbf{m} + \mathbf{v} + q \cdot \mathbf{r} \\ &= \frac{q}{t} \cdot \mathbf{m} + \mathbf{v} - \varepsilon \cdot \mathbf{m} + q \cdot \mathbf{r} \\ &= \frac{q}{t} \cdot \mathbf{m} + \frac{q}{t} \cdot \mathbf{w} + q \cdot \mathbf{r}. \end{aligned} \tag{2.42}$$

It is the quantity $-\log_2(2 \cdot \|\mathbf{w}\|)$, that is calculated:

- The function takes as input a ciphertext $\texttt{ct} = (\mathbf{c}_0, \dots, \mathbf{c}_{k-1})$.
- Using the secret key, the ciphertext polynomial is evaluated at $\mathbf{s}$ to obtain $\mathbf{c}_0 + \cdots + \mathbf{c}_{k-1} \cdot \mathbf{s}^{k-1}$.
- Then, this quantity is multiplied by the plaintext modulus t to get

$$t \cdot \mathbf{c}_0 + \cdots + \mathbf{c}_{k-1} \cdot \mathbf{s}^{k-1} = q \cdot \mathbf{m} + t \cdot (\mathbf{v} - \varepsilon \cdot \mathbf{m}) + q \cdot t \cdot \mathbf{r}. \tag{2.43}$$

- Reducing modulo q, we get

$$t \cdot \mathbf{c}_0 + \cdots + \mathbf{c}_{k-1} \cdot \mathbf{s}^{k-1} \mod q = t \cdot (\mathbf{v} - \varepsilon \cdot \mathbf{m}), \tag{2.44}$$

  which is precisely the noise $\mathbf{w}$ multiplied by the ciphertext modulus q.
- Then, the infinity norm of $\mathbf{w}$ is calculated and the noise budget $-\log_2(2 \cdot \|\mathbf{w}\|)$ is determined.

Modulus Switching As seen previously, the coefficient modulus q can be written as the product of primes q_i. SEAL automatically creates a *modulus switching chain*, which is a chain of other encryption parameters derived from the original set.

Definition 2.12 (Modulus Switching) *Let* ct *be a ciphertext. Set* $c_0 = \texttt{ct}[0]$ *and* $c_0 = \texttt{ct}[1]$. *Let p and q be arbitrary coprime moduli. The modulus switching operation (*ModSwitch*) on a ciphertext outputs*

$$(c_0', c_1') := \left(\left[\left\lfloor \frac{p}{q} c_0 \right\rceil \right]_p, \left[\left\lfloor \frac{p}{q} c_1 \right\rceil \right]_p \right). \tag{2.45}$$

In a BFV scheme, modulus switching introduces noise into the ciphertext (see Appendix E of [19]), therefore the use of modulus switching in BFV is limited to 'shaving' off primes, hence reducing the bitlength of coefficient modulus, and potentially improve computational efficiency of homomorphic operations.

Given a coefficient modulus $q = \prod_{i=1}^{k} q_i$, in SEAL, the order of the primes q_i is significant, due to the modulus switching chain. The last prime in the chain is called the *special prime*. The first parameter set in the modulus switching chain is the only one that involves the special prime. All key objects, such as the secret key, are created at this highest level. All data objects, such as ciphertexts can be only at lower levels. The special prime should be as large as the largest of the other primes in the coefficient modulus, although this is not a strict requirement. Modulus switching changes the ciphertext parameters down in the modulus chain. When using a BFVscheme, modulus switching has the following benefits: First, the size of the ciphertext is linearly dependent on the number of primes contained in the coefficient modulus. Thus, if there is no intention to perform any further computations on a given ciphertext, it is of computational benefit to switch to the smallest set of parameters in the chain before performing decryption. Also, since computational effort of homomorphic operations potentially decreases with a lower ciphertext modulus, after doing enough computations such that the noise budget reaches a certain threshold, it could be of computational benefit to apply modulus switching. In some cases it can be useful to switch to a lower level slightly earlier, sacrificing some of the noise budget in the process, to gain computational performance from having smaller parameters. In this thesis, we intend to evaluate the computational benefit of the insertion of such operations and use the effects runtime to automatically rewrite computations to insert modulus switching at appropriate points of the computation, guided by noise considerations (see Section 3.3).

We note that in other FHE schemes, such as the Cheon-Kim-Kim-Song (CKKS) [8] and the Brakerski-Gentry-Vaikuntanathan (BGV) [6] schemes, modulus switching has applications that go far beyond the reduction of computational complexity.

2.4 Noise Growth Heuristics

Homomorphic operations increase the noise in a ciphertext. For freshly encrypted ciphertexts, as well as for ciphertexts resulting from homomorphic operations, the theory of BFV dictates upper bounds for the noise determined by the parameters chosen for the encryption scheme, such as the upper bound on the distribution χ, plaintext modulus, ciphertext modulus and the ring R, as well as the encrypted plaintext. Theoretical bounds follow immediately from the definition of the invariant noise and the definition of the homomorphic operations on ciphertexts (see Appendix A.1). However, these theoretical bounds result in poor practical estimates and very often lead to overly conservative parameter choices. Instead, it is useful to apply a heuristic approach outlined in [15, 10]. This approach relies on the average distributional analysis, which estimates the expected size of the invariant noise in the canonical embedding norm by bounding the canonical norm of random polynomials whose coefficients are generated from a discrete Gaussian or uniform distribution, as in the BFV scheme. This uses the fact that given two independent random variables drawn from zero-mean distributions with variances $\tilde{\sigma}_1$ and $\tilde{\sigma}_2$, the variance of their product is equal to $\tilde{\sigma}_1\tilde{\sigma}_2$ and the variance of their sum is equal to $\tilde{\sigma}_1 + \tilde{\sigma}_2$ ([18]).

SEAL uses its own upper bound noise heuristics [22], but there are several additional studies that evaluate noise heuristics in FHE [27, 10]. Expressions for theoretical and heuristic bounds on the canonical norm of the ciphertext noise for homomorphic operations can be found in the Appendix.

Practicality of heuristic bounds Recently, experiments have been conducted in order to validate the heuristic bounds proposed in [18] by Costache *et al.* ([10]). This particular work involved 10000 trials, where the i-th trial consisted of encrypting the integers i and $i+1$ using SEAL, calculating the sum and the product of the resulting ciphertexts homomorphically, and outputting the mean of resulting ciphertext noises for fresh encryption, addition and multiplication, respectively. The obtained results ([10]) did indeed confirm that the heuristic bounds presented above are satisfied with high probability and setting parameters accordingly ensures correctness of the use of the scheme. However, the heuristic bounds from [18] do not seem to be tight, as for encryption it is reported that the heuristic bounds predicts 6 to 8 fewer bits of remaining noise budget than observed in practice. This disparity increases with the number of operations on ciphertexts, for example after a single multiplication the gap between heuristically predicted and measured noise budget reaches 8 to 17 bits.

Parameter Selection: Practical aspects As understood from the preceding sections, parameter selection of a homomorphic encryption scheme is dictated by noise growth which strongly varies with the number and nature of performed homomorphic operations. Parameter selection has significant impact on the performance of a homomorphic encryption scheme and therefore parameter selection is highly dependent on the specific arithmetic circuit to be evaluated. A factor that strongly affects the noise growth is the *multiplicative depth*, the number of consecutively performed multiplications.

Several approaches have been explored to optimise parameter selection given a certain arithmetic circuit, notably the CINGUPARAM suite ([17]), that determines the multiplicative depth of a circuit and selects parameters using a predefined parameter database. There exist algorithms that take arithmetic as an input and rewrite the circuit to minimise its multiplicative depth [2], one of which, the *cone rewriting* operator, we shall discuss later in this text.

Chapter 3

Design

We begin with a brief overview of the challenges of FHE development, specifically from the point of view of a compiler. Then, we present how we can employ existing heuristics about ciphertext degradation to guide users towards optimising computation circuits to minimise said degradation. Finally, we present our automated optimisation which can transform FHE programs into semantically equivalent circuits with optimized performance.

3.1 FHE Compilation

Fully Homomorphic Encryption (FHE) has become increasingly popular in recent years, due to significant advances and performance improvements of FHE schemes such as the introduction of Single Instruction Multiple Data (SIMD) style parallelism [28] allowing to encode multiple plaintext values into single ciphertexts. However, to this day it remains a challenging task to build efficient applications based on FHE. This is due to the computation models that FHE dictates, such as data-independent computations, as well as different performance tradeoffs offered by different FHE schemes. Noise management in FHE computations is also an important factor in achieving optimal performance. Minimising the noise build-up in an FHE evaluation of an arithmetic circuit, allows the choice of smaller parameters, expected to increase performance.

In practice, it is a complex endeavour to realise efficient FHE computations, requiring significant expertise in the underlying cryptographic protocols as well as deep knowledge of high-performance computations. Therefore, FHE libraries such as SEAL [22] offer higher-level APIs to make basic functionalities such as key generation, encryption, decryption, as well as homomorphic addition and multiplication accessible to the general user. However, the libraries still remain relatively low-level and leave issues such as noise management and circuit design to the user. It was therefore a natural development, that in recent years multiple higher-level tools, commonly known as FHE *compilers*, have surfaced in the community, aiming to translate programs given by the user to a set of FHE instructions, while ensuring correctness of the computations and simultaneously achieving a desired level of efficiency.

We recall that FHE encryption introduces noise into ciphertexts that grows during homomorphic computations and can eventually grow to a point where decryption fails. FHE schemes offer ciphertext maintenance functionalities that can be applied to ciphertexts to limit noise growth.

The focus on currently available compilers lies on improving usability for the general inexperienced user, while offering optimisations that have previously been exclusive to the advanced user. There exist a variety of domain specific optimisations, for example optimising matrix-vector operations [12] or applications tailored to suit the needs of machine learning [21], [11], [29]. FHE compilers have made the task of developing FHE computations much more accessible with compiler-optimised computations outperforming hand-crafted FHE circuits in many instances [4]. However, available compiler frameworks come with limitations, some domain specific in their application, being limited to computations that can be expressed by polynomials [1] or offering only limited support of non-polynomial functions ([4]). In many cases, circuit optimisations are not available and ciphertext maintenance operations are not inserted optimally by the compiler [29]. A comprehensive overview over available FHE compilers can be found in [30].

The goal of this thesis is to support developers in achieving state of the art results without having to perform manual tuning by adopting a holistic approach combining rewriting and noise heuristics for FHE applications. Specifically, we aim to further the development of the Automated Batching Compiler (ABC) compiling framework to be able to efficiently identify bottlenecks in a circuit via noise heuristics: we apply existing noise heuristics to guide the developer to manually transform the program to achieve optimisation, requiring extending the existing compiler framework to support a mapping between the high-level input program and the resulting FHE operations in the compiled circuit. Further, we aim to develop a automated rewriting algorithm guided by noise heuristics. We use noise heuristics to guide an automated rewriting algorithm to insert modulus switching operations at appropriate points in the computation.

The ABC functions by translating a high-level input program into a circuit in the form of an Abstract Syntax Tree (AST), that is later translated into the set of instructions needed for the FHE library to execute the circuit. An AST is a tree representation of the abstract syntactic structure of the given high-level code. In the AST, each node corresponds to a construct from the source code.

The ABC compiler translates high-level program descriptions in a C-like language into the circuit-based programming paradigm of FHE. It does so while automating as many aspects of the development as possible. The ABC uses a simple C-like high-level input language, and a parser that translates a circuit given in this language into an AST, which forms the*intermediate representation* (IR). The ABC features a runtime system that can take (circuit-compatible) ASTs and run them against FHE libraries. Currently, the ABC framework exclusively supports SEAL.

Optimisations on circuits are performed on the AST. The compilation of a circuit itself can be divided into three stages:

Program Transformations These AST-to-AST transformations aim to modify the program to make it more suitable for efficient FHE computations. These include optimizations common in standard compilers and FHE-specific optimizations like exploiting opportunities to use the powerful SIMD parallelism ("batching") present in many FHE schemes.

AST-to-Circuit Transformations These transform the AST into a circuit by transforming non-compatible operations (e.g., If- and While-Statements) into their circuit-equivalent using gates. Note that instead of changing to a wires-and-gates IR, circuits are still expressed using (a subset of) the AST IR.

Circuit-to-Circuit Transformations These transformations transform a circuit into a semantically equivalent circuit using certain rewriting rules.

The advantages of translating the program provided by the user into an AST are that every node can be enriched with properties such as additional variables that keep track of certain properties, providing a powerful framework for optimisations.

This thesis aims to use the ABC framework to apply optimisations to given programs in the form of ASTs. We calculate noise heuristics for each node of the AST of a given circuit and use these quantities to improve circuit design automatically.

Noise grows when performing operations on ciphertexts. This means, if the noise is not managed carefully, the computation will not produce the correct result. Several strategies of noise management can be pursued, such as the insertion of relinearisation and modulus switching when appropriately performed. However, noise management is generally considered to be a difficult task, often inaccessible to a non-expert user. It is not trivial when to insert these maintenance operations, since, for instance, in the case of modulus switching operations can cause parameter mismatches in binary operations performed at a later point in the evaluation of the circuit, compromising the correctness of the circuit.

We pursue a set of strategies, that are based on noise heuristics, to transform a given circuit into an equivalent circuit with improved performance. These will be outlined in the following sections.

3.2 Circuit Optimisations and Noise Heuristics

3.2.1 Identifying Areas of Significant Noise Growth

When homomorphically evaluating circuits, the quantity of the noise of a ciphertext plays a significant role. When the noise budget reaches zero, decryption of the result will fail and an incorrect result will be returned to the user. Noise growth can be limited by adjusting parameters, such as the polynomial degree of the elements of the ring R and the ciphertext modulus. However, limiting noise growth this way comes with a significant computational cost that we aim to minimise. It is therefore useful to identify regions that are the source of major noise growth in a given circuit to suggest to the user where there might be potential for circuit redesign. This feature is aimed towards the novice user that have little to no experience in circuit design for FHE computations to provide a starting point of potential circuit optimisation that can be carried out manually.

We view an arithmetic circuit as a directed acyclic graph (DAG) $C = (V, E)$, where each node represents a value available during homomorphic execution. Nodes with one or more incoming edges are called *instructions*, which compute a new value as a function of the children nodes connected to it. Each instruction v has an opcode specifying the operation performed at the node such as homomorphic addition (`Add`), subtraction (`Sub`), multiplication (`Mult`) and modulus switching (`ModSwitch`).

We use the heuristic noise budget and the heuristic relative noise budget decay to identify such regions. The relative noise budget decay is defined as

$$\frac{1}{\min\{E_1, E_2\}} \cdot (\min\{E_1, E_2\} - E), \tag{3.1}$$

where E_1 and E_2 denote the noise budgets of the operands and E denotes the remaining noise budget after a homomorphic evaluation of a binary operation. However, these quantities alone can not be used to identify regions of noise growth: the noise budget will always be the lowest at the very end of a computation, since homomorphic operations in levelled FHE build up noise, but never reduce noise. As a simple illustrative example, we consider the DAG of the arithmetic circuit computing the function $f(x) = (x \cdot x \cdot x \cdot x) \cdot ((x \cdot x) \cdot (x \cdot x))$ (Figure 3.1). We wish to identify the

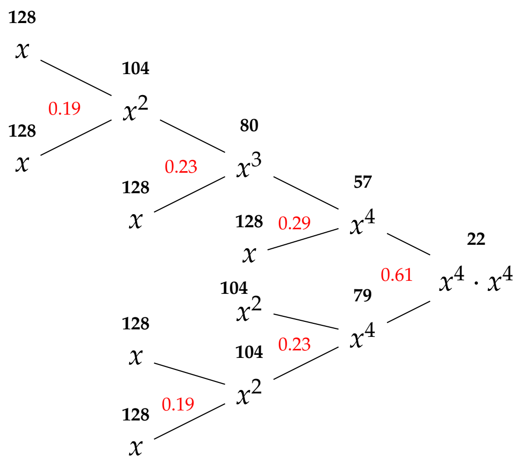

Figure 3.1: Circuit noise budget analysis for the evaluation of $f(x) = (x \cdot x \cdot x \cdot x) \cdot ((x \cdot x) \cdot (x \cdot x))$. The tree shows remaining noise budgets after each operation (bold black) and relative noise budget decay for each binary operation (red). Results are shown for a plaintext modulus of $t = 65537$ and a polynomial degree of $d = 8192$.

subtree evaluating the function $x \cdot x \cdot x \cdot x$. We observe that, indeed, the heuristic noise budget reaches its lowest value at the end of the computation. Also the relative noise budget decay does not point us to the source of most significant noise growth, the subtree evaluating the function $x \cdot x \cdot x \cdot x$.

To identify such subtrees of significant noise growth in an AST, we propose an algorithm, that, starting from a root node (result of the computation), traverses the DAG, recursively visiting the children that exhibit the lowest noise budget. If all children have the same noise budget, all children are visited. Eventually, the algorithm terminates and returns one or more leaf nodes that indicate paths that are the source of significant noise growth. Algorithm 1 uses the functions *getNoiseBudget(node v)* that returns the noise heuristics of a given input node (ciphertext), as well as *calcInitNoiseHeur()* that returns noise heuristics for a freshly encrypted ciphertext using Equation (A.10).

Algorithm 1 visit(node)

```
$left \leftarrow getNoiseBudget(leftChild)$
$right \leftarrow getNoiseBudget(rightChild)$
$initial \leftarrow calculateEncryptionNoiseHeuristic()$
**if** $((right == left) \wedge ((right == initial) \vee (right == NULL)))$ **then**
  **return** node
**else if** $(left < right)$ **then**
  visit(left)
**else if** $(left > right)$ **then**
  visit(right)
**else**
  visit(left)
  visit(right)
**end if**
```

Lemma 3.1 *Algorithm 1 always terminates and returns a node.*

Proof We proceed by induction: for trees of size one, the algorithm returns the single node. Assume the algorithm terminates and returns a node for a tree of size $n - 1$. Considering a tree of size n, in the first step of the algorithm, the algorithm visits a tree of size at most $n - 1$. Applying the induction hypothesis, the algorithm terminates and returns a node.

The algorithm produces one or more paths that are potentially subject to rewriting to limit noise growth during homomorphic evaluation allowing for a selection of more optimal parameters and consequently increasing computational efficiency. However, it is important to note that the returned path is not necessarily a region in the circuit where potential for optimisation exists, it may well be that the algorithm returns a path that is already fully optimised. Therefore, the algorithm can merely serve as a basis for circuit improvement strategies. This poses an additional research question, namely if there is potential for noise heuristics to provide a basis for automatic circuit rewriting. Automatic circuit rewriting strategies indeed exist to improve computational efficiency (*e.g.* cone rewriting), but they are generally based on much simpler *multiplicative depth* heuristics instead of noise heuristics.

In the next section, we will review one such strategy and discuss how noise heuristics compare to simpler heuristics like multiplicative depth.

3.2.2 Cone Rewriting of Arithmetic Circuits

As an example of an optimisation heuristic, we consider the *cone rewriting* approach to reduce the multiplicative depth, *i.e.*, the number of consecutive multiplications, of FHE circuits [2]. While originally proposed for boolean circuits, it can be generalised to arithmetic circuits. It tries to reduce the multiplicative depth which should improve the noise growth behaviour, allowing smaller parameters to be used. As a trade-off, it increases the overall number of multiplications in the circuit. Therefore, this transformation is not guaranteed to improve the circuit and can in fact lead to slower overall run times.

Consider a Boolean circuit represented as a directed acyclic graph (DAG) $C = (V, E)$ consisting of a set of nodes V and a set of edges E. The *multiplicative depth* of a Boolean circuit is defined as the number of successively performed AND operations. The minimisation of the multiplicative depth permits smaller ciphertext sizes and reduces the overall execution time of a homomorphic evaluation of the circuit. Let $d : V \longrightarrow \{0, 1\}$ be a function that returns 1 for AND nodes and 0 otherwise and let $\mathrm{pred} : V \longrightarrow 2^V$ and $\mathrm{succ} : V \longrightarrow 2^V$ be functions giving the set of predecessors, respectively successors, of a node $v \in V$. The *multiplicative depth* of a given node v is defined as the maximimum number of AND gates on any path from an input node of the arithmetic circuit to the node v:

$$l(v) = \begin{cases} 0 & \text{if } |\mathrm{pred}(v)| = 0 \\ \max_{u \in \mathrm{pred}(v)} l(u) + d(v) & \text{otherwise} \end{cases}. \tag{3.2}$$

Similarly, the *reverse multiplicative depth* of a node v is defined as the maximum number of AND gates on any path beginning by a successor of v and ending with an output node. The *reverse multiplicative depth* is hence given by

$$r(v) = \begin{cases} 0 & \text{if } |\mathrm{succ}(v)| = 0 \\ \max_{u \in \mathrm{succ}(v)} r(u) + d(v) & \text{otherwise} \end{cases}. \tag{3.3}$$

Finally, the *multiplicative depth* of a circuit is defined as the maximal multiplicative depth of all of its nodes, given by

$$l^{\max} := \max_{v \in V} l(v) = \max_{v \in V} r(v). \tag{3.4}$$

A node $v \in V$ is called a *critical node* if

$$l(v) + r(v) = l^{\max}. \tag{3.5}$$

A subcircuit C^* of a boolean circuit C containing all the critical nodes of C is called a *critical circuit*. A *critical* path is a path in C^* and a *critical cone* is a subset of connected *critical nodes* with a common descendant. We note that the multiplicative depth of the circuit C is equal to the multiplicative depth of the subcircuit C^*. Hence, decreasing the multiplicative depth of C^* decreases the overall multiplicative depth of the entire circuit.

In [7], two rewrite operators for Boolean circuits to reduce multiplicative depth of circuits have been proposed. First an operator which rewrites simple paths composed of two AND gates only, and second, an operator which allows obtaining a path with two AND gates from any path of multiplicative depth 2. More specifically, a critical path of length 2 can be rewritten using the associativity of an AND operation: $(x \cdot y) \cdot z = x \cdot (y \cdot z)$. When the circuit is rewritten and has the form $x \cdot (y \cdot z)$, multiplicative depth decreases only if the multiplicative depth of nodes y and z are less than the multiplicative depth of node x. If this is the case, then multiplicative depth decreases by one. When critical paths are of size larger than two, *i.e.* also contain inner XOR gates, a second operator is described in [7] that allows to move an AND gate up the critical path by one place. This operator uses the XOR distributivity rule $(x \oplus y) \cdot z = (x \cdot z) \oplus (y \cdot z)$ and is applicable to paths with any number of internal XOR gates. The application of this operator adds an AND gate for each XOR gate on the path. Given a critical path $p = (v_1, v_2, \ldots, v_{|p|})$, where v_1 and $V_{|p|}$ are AND gates, it is possible to first move up the END gate $v_{|p|}$ next to v_1 yielding a critical path of length two. Afterwards, this path can be rewritten using the first operator. However, since there might be multiple parallel critical paths in a circuit, all these paths have to be rewritten to obtain a circuit of overall lower multiplicative depth. A so called multti-start heuristic is described that is a priority based heuristic to rewrite critical paths of multiplicative depth two.
In [2], the two operators described above have been combined into a single one. The multiplicative depth-2 path operator described in [7] is generalised to cones of multiplicative depth two. The resulting operator is called the *cone rewriting* operator. In [2], a heuristic is presented that aims at minimizing the multiplicative depth of a Boolean circuit in a single pass: At each iteration a set Δ_{min} of cones to minimize is computed. If this set is non-empty, the cones from this set are rewritten and subsequently the multiplicative depths of the circuit nodes are updated. If the multiplicative depth of the new circuit is smaller, the circuit is updated. Otherwise, a new set of cones is determined and the the rewriting algorithm is applied again. The algorithm finally terminates when the set Δ_{min} is empty. The goal of the cone selection method is to find a minimal set of cones to rewrite, the reduction of which is likely to lead to a decrease in the overall multiplicative depth. Any cone rewriting operator adds new nodes to the circuit. Therefore, minimizing the set of cones to be rewritten also limits the number of additional nodes created by the rewriting operator.

In [2], the effect of the minimisation of the multiplicative depth of circuits on homomorphic execution has been studied and yields more favourable runtimes of homomorphic execution compared to results from [7]. However, in the context of an homomorphic execution of Boolean circuits, the minimization of multiplicative depth is beneficial only if the number of newly created AND gates is below a threshold. However, circuit rewriting based on multiplicative-depth heuristic to reduce noise growth is of limited use in arithmetic circuits, which tend to feature significantly lower depths to begin with. In general, the effects of heuristics are highly dependent on the nature of the circuit. If a circuit's noise growth is dominated by a path that is not subject to cone rewriting, as for example a large number of consecutive additions, cone rewriting strategies have little to no effect on circuit optimisation.

When considering optimisations based on noise heuristics, there is currently no direct way to translate from high noise growth to circuit improvement in the same way that we can translate high multiplicative depth into the reduction of multiplicative depth through simple heuristics. However, there is indeed an optimisation that immediately follows from understanding noise growth behaviour of a circuit, which we will define and explore in the following to evaluate its effect on computational efficiency of homomorphic circuit evaluation.

3.3 Circuit Rewriting based on noise-guided Modulus Switching

For any given function, one can design a large number of arithmetic circuits evaluating that function. However, when evaluating a function homomorphically, circuit design is of great importance. Some circuits accumulate more ciphertext noise than others caused by different factors, such as the number of consecutively performed multiplications or the placement of ciphertext maintenance operations such as relinearisation and modulus switching. As mentioned before, it is a challenging task for a user to manually optimise circuits in terms of noise growth and computational performance. As we aim improve the computational efficiency of circuit evaluation using FHE by automatically rewriting user-input circuits, we present a pair of algorithms that uses noise heuristics to insert modulus switching operations at appropriate points to reduce the bitlengths of ciphertext moduli to accelerate homomorphic evaluation of the rest of the circuit.

In BFV, ciphertexts are large polynomials with potentially large coefficients. For performance reasons, the coefficients are split into smaller parts, that are easier to handle computationally. The idea of rewriting a circuit based on noise-guided modulus switching is to improve runtimes of the homomorphic evaluation by reducing the bitlengths of ciphertexts involved in homomorphic binary operations. Further, for hardware specifically designed to perform FHE evaluations, the acceleration achieved is limited by memory bandwidth. When reducing bitlengths of ciphertexts, the acceleration can be maximised, making the insertion of modulus switching at appropriate points in a circuit relevant.

In BFV encryption schemes, the coefficient modulus q can be written as the product of primes q_i. We write $q = \prod_{i=1}^{r} q_i = \{q_1, \ldots, q_r\}$. In SEAL, it is possible to specify the bitlengths of the primes $q_1, \ldots, q_r$. As described previously, it is possible to increase computational eficiency by 'dropping' primes in order to reduce the total bitlength of the modulus q using modulus switching (`ModSwitch`). Based on the constraints discussed in Paragraph 2.3.2, the proposed algorithm (Algorithm 2) considers the operands of each multiplication in a given circuit and calculates the noise budget that has been spent during the computation to automatically apply modulus switching operations to the operands if the number of bits in a ciphertext occupied by noise exceed the bitlength of the last prime in the modulus switching chain. If this is the case, the algorithm applies modulus switching to the operands.

We note that, when inserting `ModSwitch` nodes into the DAG, to ensure correctness of the computation, for binary operations involving two ciphertexts $\texttt{ct}_1$ and $\texttt{ct}_2$, it must be satisfied that both have the same ciphertext modulus. As the `ModSwitch` operation changes the coefficient modulus of a ciphertext, it must be ensured that two ciphertexts involved in a homomorphic binary operation have the same coefficient modulus at all times. We have seen that after enough computations have been performed it is reasonable to perform `ModSwitch`, but we must ensure that for both ciphertexts the bitlength of the noise term is larger than the bitlength of the last prime q_s in the product of remaining ciphertext modulus $\{q_1, \ldots, q_s\}$.

We propose an algorithm (Algorithm 2) that uses the quantity of ciphertext noise to decide whether the insertion of a modulus switching operation in a circuit is possible without compromising the computation. Specifically, the proposed algorithm uses noise heuristics calculated for each node of a given circuit. We consider the difference of the noise budget of a fresh ciphertext and the noise budget of a node in the DAG and shall call it the *spent noise budget*. For each binary operation in the circuit, the operands' spent noise budgets are compared to the bitlengths of the last prime factors of the operands' ciphertext moduli. If their difference exceeds the last prime in the modulus switching chain and the binary expression is a multiplication, a `ModSwitch` operation is possible and inserted in the circuit by the function $insertModSwitch(operand)$. This function takes the operand ciphertext as input. In this step of the algorithm, it is ensured, that no parameter mismatches occur in the considered binary operation, *i.e.* both operands of the multiplication have the same ciphertext modulus. Then, the noise heuristics of the new circuit are recalculated. If the noise budget at the root node of the DAG remains greater than zero, the changes are accepted and the circuit is updated. The proposed algorithm makes use of a set of auxiliary functions. The functions $getLeftOperand(op)$ and $getRightOperand(op)$

take a binary operation op as an input and return the left, respectively right, operand. The function $getLastPrimeIndex(v)$ takes a ciphertext node as an input and returns the index of the last prime in the ciphertexts coefficient modulus chain. To calculate the bitlength of specific primes in the coefficient modulus, the function $bitLen(q_i)$ has been defined. Finally, the function $spentNoiseBudget(node)$ takes as input a ciphertext and returns the spent noise budget as defined above.

As mentioned above, we must ensure that coefficient moduli of ciphertexts involved in binary operations match at all times. It is therefore necessary to perform second pass over the circuit to ensure the new circuit fulfills the constraint of matching ciphertext moduli. The procedure, as described in Algorithm 3, starts at the root of the DAG and visits all binary expressions in ascending order towards the root node and compares the ciphertext moduli of the input nodes (children) of the current binary expression. If there is a mismatch, then `ModSwitch` operations on the operand with the larger ciphertext modulus are performed until the operands' ciphertext moduli are equal. This may be done, since the resulting spent noise budget after any binary operation will exceed the size of the primes dropped.

Lemma 3.2 *The Algorithms 2 and 3 terminate and produce a valid arithmetic circuit.*

Proof The main loop of algorithm 2 terminates, since there are finitely many binary instructions in a tree. Also, Algorithm 3 returns a valid tree, since it ensures that the number of primes in the moduli is equal for operands in each binary expression. □

As we will discuss when evaluating the influence of circuit rewriting using the proposed algorithms, there is indeed a difference if a modulus switching operation is inserted before an addition or a multiplication. The homomorphic evaluation of an addition is relatively fast compared to modulus switching and multiplication, since ciphertexts are added component-wise which is a fast operation. The operations underlying modulus switching in BFV schemes underlies more complex mathematical operations and therefore inherently greater computational complexity. This can impact the performance of a circuit evaluation negatively, if the algorithms insert modulus switching before additions. Further, since noise heuristics generally overestimate the spent noise budget of a ciphertext, the position of modulus switches inserted does not necessarily correspond to the insertion site obtained from real noise calculations. However, since Algorithm 2 recalculates the noise heuristics of the entire circuit after each insertion of a modulus switching operation, the correctness of the circuit is always guaranteed.

Algorithm 2 rewrite(*circuit*)

```
for $op : binaryOps(circuit)$ do
    if $(op.opcode == Mult)$ then
        $tempCircuit \leftarrow circuit$
        $left \leftarrow getLeftOperand(op)$
        $right \leftarrow getRightOperand(op)$
        $leftIndex \leftarrow getLastPrimeIndex(left)$
        $rightIndex \leftarrow getLastPrimeIndex(right)$
        $diff \leftarrow indexLeft - indexRight$
        if $(diff > 0)$ then
            $sum \leftarrow 0$
            for $(i = leftIndex; i = leftIndex - (|diff| + 1); i - -)$ do
                $sum \leftarrow sum + bitLen(coeffModulus[i])$
            end for
            if $(sum < spentNoiseBudget(left)) \wedge spentNoiseBudget(right) > bitLen(coeffModulus[rightIndex])$ then
                $tempCircuit.insertModSwitch(right)$
                for $(j = 0; j = |diff| - 1; j + +)$ do
                    $tempCircuit.insertModSwitch(left)$
                end for
            end if
            if $(tempCircuit.getNoiseBudget(root) > 0)$ then
                $circuit \leftarrow tempCircuit$
            end if
        else if $(diff < 0)$ then
            $sum \leftarrow 0$
            for $(i = rightIndex; i = rightIndex - (|diff| + 1); i - -)$ do
                $sum \leftarrow sum + bitLen(coeffModulus[i])$
            end for
            if $(sum < spentNoiseBudget(right)) \wedge spentNoiseBudget(left) > bitLen(coeffModulus[leftIndex])$ then
                $tempCircuit.insertModSwitch(left)$
                for $(j = 0; j = |diff| - 1; j + +)$ do
                    $tempCircuit.insertModSwitch(right)$
                end for
            end if
            if $(tempCircuit.getNoiseBudget(root) > 0)$ then
                $circuit \leftarrow tempCircuit$
            end if
        else
            if $bitLen(coeffModulus[leftIndex]) > spentNoiseBudget(left) \wedge bitLen(coeffModulus[rightIndex]) > spentNoiseBudget(right)$ then
                $tempCircuit.insertModSwitch(left)$
                $tempCircuit.insertModSwitch(right)$
                if $(tempCircuit.getNoiseBudget(root) > 0)$ then
                    $circuit \leftarrow tempCircuit$
                end if
            end if
        end if
    end if
end for
return $circuit$
```

Algorithm 3 visit(*node*)

```
if $isBinaryExpression(node)$ then
    if $(node.getLeft().hasChild() \wedge node.getLeft().visited() == false)$ then
        $visit(node.getLeft())$
    end if
    if $(node.getRight().hasChild() \wedge node.getRight().visited() == false)$ then
        $visit(node.getRight())$
    else if $(getLastPrimeIndex(node.getLeft()) \neq getLastPrimeIndex(node.getRight()))$ then
        $insertModSwitch$
    end if
end if
```

Chapter 4

Implementation and Evaluation

4.1 Implementation

All implementations of functionalities and algorithms relevant for this thesis are included in the ABC Compiler Framework.

4.1.1 Noise Heuristics in the ABC

To efficiently analyse the noise growth present in a given circuit to be evalutated using FHE via the ABC framework, it is useful to have a means of monitoring noise growth without having to evaluate significantly time consuming operations on ciphertexts using actual FHE schemes. Therefore, we have implemented a Runtime System into the ABC compiler that calculates noise heuristics as given in Section A.2 for each node of an AST obtained from a given arithmetic circuit. The calculation of noise heuristics is done in the class `SimulatorCiphertext`, which can be used in complete analogy to the already existing class `SealCiphertext` that evaluates arithmetic circuits given by an AST using the FHE scheme SEAL. The reader is referred to Section A.3 for performance analysis and comparison to previous noise heuristics calculations [10] and evaluations using SEAL.

Recording Noise Heuristics in the AST We recall that the ABC takes a high level program and translates it to an AST which is in turn evaluated in SEAL or as a simulation calculating noise heuristics depending on the user's choice. The AST is evaluated by the `RuntimeVisitor` class of the ABC. We have modified this class to maintain a map from the node id to the current noise budget left in the ciphertexts. This allows efficient analysis of arithmetic circuits in terms of noise growth heuristics and serves as a metric for identification of areas of significant noise growth in a given circuit providing a basis for suggesting optimisation and parameter choice. Noise Heuristics have been implemented in a separate class called `SimulatorCiphertext` that evaluates noise heuristics for encryption, binary operations and modulus switching based on the noise heuristics of operands' ciphertexts. Since noise heuristics calculations require high numerical precision, the class uses the GNU Multiple Precision Arithmetic library (https://gmplib.org) to represent noise values. Evaluation of the implementation and calculated noise heuristics in comparison to results from [10] are shown in the Appendix.

Identifying Areas of Significant Noise Growth The logic of Algorithm 1 can be found in the class `IdentifyNoisySubtreeVisitor` and has been implemented as a visitor that recursively visits binary expression starting at the root node of an arithmetic circuit's AST. The visitor follows the operands with the smaller noise budget, or both, if equal and returns the leaf nodes of the AST corresponding to the subtrees generating the largest noise in the AST.

Noise-based insertion of modulus switching operations in ASTs The identification of binary expressions in an AST that are suitable candidates for an insertion of a modulus switching operation to potentially improve performance of homomorphic execution has been implemented as a visitor in `InsertModSwitchVisitor`. The visitor recursively visits binary expressions and examines their operands' noise budgets, if inserting a modulus switching is feasible. Since the visitor needs to compare the operands' coefficient moduli bitlengths with the spent noise budgets, a map is maintained that contains the ciphertexts' coefficient moduli. The `InsertModSwitchVisitor` produces a vector of potential modulus-switching insertion sites. In the same class, the function `insertModSwitch` has been implemented that inserts the modulus switching operations into the AST based on the visitor's results and updates the coefficent modulus map. We note that the function `insertModSwitch` is not a visitor. After insertion of modulus switches, it is required to pass over the AST a second time to avoid parameter mismatches at binary operations. For this, a second visitor has been implemented in the separate class `FixParamMismatchVisitor` that, based on the maintained map containing the coefficient moduli for each ciphertext, inserts additional modswitches to one of the operands of the binary expression to allow correct homomorphic execution of the AST.

4.1.2 Evaluation Setup

Benchmarking on a variety of arithmetic circuits has been performed to evaluate the effect of suggested circuit improvements on computational runtime. The machine used for performance measurements is an Apple M1 CPU (3.2 *GHz*) with 8 GB RAM. Circuits have been evaluated using SEAL for a variety of polynomial degrees and plaintext and ciphertext moduli calculated automatically from SEAL based on the chosen polynomial degree. To analyse effect on execution time of the developed circuit optimisation tools, a selection of arithmetic circuits from the EPFL Combinatorial Benchmark Suite have been used for experimentation. A parser translating circuits from the Benchmark Suite given in *verilog* format to the domain specific language used by the parser employed by the ABC has been implemented in the class `VerilogToDsl`. Circuits selected for evaluation have been truncated in order to allow their correct homomorphic evaluation using SEAL without the noise budget becoming zero.

4.2 Circuit Rewriting based on noise-guided modulus switching

In this section, we evaluate the effect of automatic circuit rewriting based on noise-guided modulus switching, in particular the effects of consecutively applying Algorithms 2 and 3 to a selection of arithmetic circuits. We show that the application of said algorithms can have different effects on runtime of homomorphic evaluation, dependent on the nature of the arithmetic circuit considered. First, we evaluate the runtimes of single homomorphic binary operations and modulus switching to show the effect of modulus switching insertion on single operations.

We will show that introducing modulus switches is not always beneficial for runtimes. The devised strategy however, is of potential benefit in terms of memory usage, since ciphertexts with smaller coefficient moduli require less space in memory. This has implications for the use of hardware acceleration specifically designed for FHE computations.

Benchmarking. Runtimes in microseconds averaged over 1000 homomorphic evaluations of the binary operations addition and multiplication at different levels in the modulus-switching chain compared to the average runtime of modulus switching at the same level are shown in Tables 4.1 and 4.2 for polynomial degrees $d \in \{16384, 32768\}$. We observe that the average runtime of a modulus switching operation lies above the runtime for an addition, indicating that it will have adverse effects on performance if Algorithms 2 and 3 insert modulus switching instructions before homomorphic additions. However, it can be seen that the cost of performing modulus switching on both operands and then multiplying ciphertexts with resulting smaller coefficient moduli indeed benefits performance. In the following, we construct circuits based on these observations and evaluate their runtimes with and without the insertion of modulus switching to study the effects of modulus switching in a variety of circumstances. We further evaluate the the effects of Algorithms 2 and 3 on runtimes of a subset of circuits of the EPFL Benchmarking Suite.

d = 16384	`Mult`	`Add`	`ModSwitch`
Level 1	30189(31)	220(1)	1167(2)
Level 2	25733(11)	208(2)	1036(2)
Level 3	21786(12)	191(2)	856(1)
Level 4	18030(4)	143(1)	855(1)

Table 4.1: Benchmarking for binary ciphertext-ciphertext operations as well as modulus switching using SEAL for the polynomial degree $d = 16384$. Results are shown in μs and standard errors are given in brackets.

d = 32768	`Mult`	`Add`	`ModSwitch`
Level 1	149681(191)	844(4)	4375(7)
Level 2	135788(87)	740(1)	4038(4)
Level 3	124121(89)	699(1)	3747(3)
Level 4	110709(50)	648(1)	3774(2)

Table 4.2: Benchmarking for binary ciphertext-ciphertext operations as well as modulus switching using SEAL for the polynomial degree $d = 32768$. Results are shown in μs and standard errors are given in brackets.

We analyse the effect of circuit rewriting based on noise-guided modulus switching on a selection of circuits to represent best-case scenarios as well as worst-case scenarios to show that the pursued strategies can have various effect on circuits of varying nature. In addition, we use circuits from the EPFL benchmarking suite to evaluate the effect on a non-constructed circuit. Note that the circuits from the EPFL benchmarking suite are designed to benchmark hardware design tools and are not realistic in terms of what is feasible with homomorphic evaluation. Hence, truncated versions have been chosen to represent realistic and non-biased non-pathological circuits. Evaluated circuits have been selected to clearly show the effect of the number of performed additions in a circuit on the efficacy of the optimisations introduced. The overall effect on runtime of homomorphic circuit evaluation after application of Algorithms 2 and 3 is relatively small, usually in the range of 2 percent. Table 4.3 shows the ratios of observed average runtimes of homomorphic evaluation of the original circuits and the runtimes measured after insertion of modulus switching instructions by Algorithms 2 and 3. Bootstrapping on the benchmarking data has been performed to obtain a distribution of averages, from which the average speedup has been calculated. From the Table 4.3 it can be seen, that circuit rewriting based on modulus switching does not necessarily have a positive impact on computational performance. We will see that the effect of applying Algorithms 2 and 3 is highly dependent on the circuit considered.

Consider the circuit representing the evaluation of the polynomial $f(x, y, z) = (x^4 + y) \cdot z^4$ (Figure 4.1). Noise heuristics calculations are shown for a poly-

Circuit	$d = 8192$	$d = 16384$	$d = 32768$
$(x^4 + y) \cdot z^4$	1.02	1.02	1.01
$(x^4 + y) \cdot z^4 + \sum_{i=1}^{12} a_i$	0.93	0.98	0.98
`Adder`	-	1.00	0.99
`Bar`	-	1.00	1.02
`Max`	-	1.02	1.00

Table 4.3: Average speedup factors for the evaluation of selected circuits. Speedup factors have been calculated as the ratio of runtimes without modulus switching and with modulus switching.

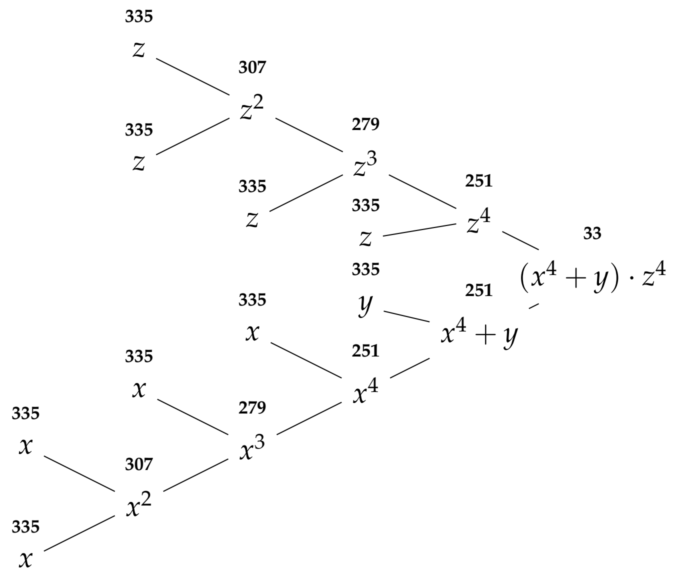

Figure 4.1: Circuit noise budget analysis for the evaluation of $f(x, y, z) = (x \cdot x \cdot x \cdot x + y) \cdot (z \cdot z \cdot z \cdot x)$ using noise heuristics from [18] and for a polynomial degree of $d = 16384$. The tree shows remaining noise budgets after each operation (bold black).

nomial degree of $d = 16384$. By construction, the algorithms insert a modulus switching operation applied to each of the operands $(x^4 + y)$ and z^4 precisely before the calculation of the final result $(x^4 + y) \cdot z^4$, In Table 4.4 and Figure 4.2 we show a comparison of average over 100 runtime evaluations when computing the circuit with and without `ModSwitch` before the last operation using SEAL for polynomial degrees $d \in \{8192, 16384, 32768\}$. Table 4.3 shows a speedup factor greater than 1 when inserting modulus switching. We observe improvement in runtime when including modulus switching into the homomorphic evaluation of the circuit while ensuring the correctness of the computation's result.

This is, however due to the nature of the circuit, since there are no additions present in the circuit, whose operands need to undergo modulus switching as a consequence of the rewriting of the circuit. To illustrate the importance of the nature of the arithmetic circuit on runtime improvement and to show that the application of Algorithms 2 and 3 can indeed have a detrimental effect on the runtimes of homomorphic circuit evaluation we consider the circuit evaluating the function $(x^4 + y) \cdot z^4 + \sum_{i=1}^{12} a_i$. As in the previous example, Algorithm 2 inserts a modulus switching operation before the multiplication $(x^4 + y) \cdot z^4$. In order to compensate for parameter mismatches in the consecutive additions, Algorithm 3 applies modulus switching to the summands $a_i, (i \in \{1, \dots, 12\})$. This, however, has a negative effect on runtimes of homomorphic evaluations as shown in Table 4.5 and Figure 4.3, as well as Table 4.3. We observe slower runtimes after the application of Algorithms 2 and 3 compared to evaluating the original circuit.

d	`ModSwitch`	no `ModSwitch`
8192	75(0.2)	76(0.02)
16384	323(1)	338(8)
32768	1605(4)	1624(5)

Table 4.4: Average runtime in ms for the evaluation of $f(x, y, z) = (x^4 + y) \cdot z^4$ using SEAL with and without performing modulus switching on the operands of the last multiplication. Standard errors are shown in parentheses.

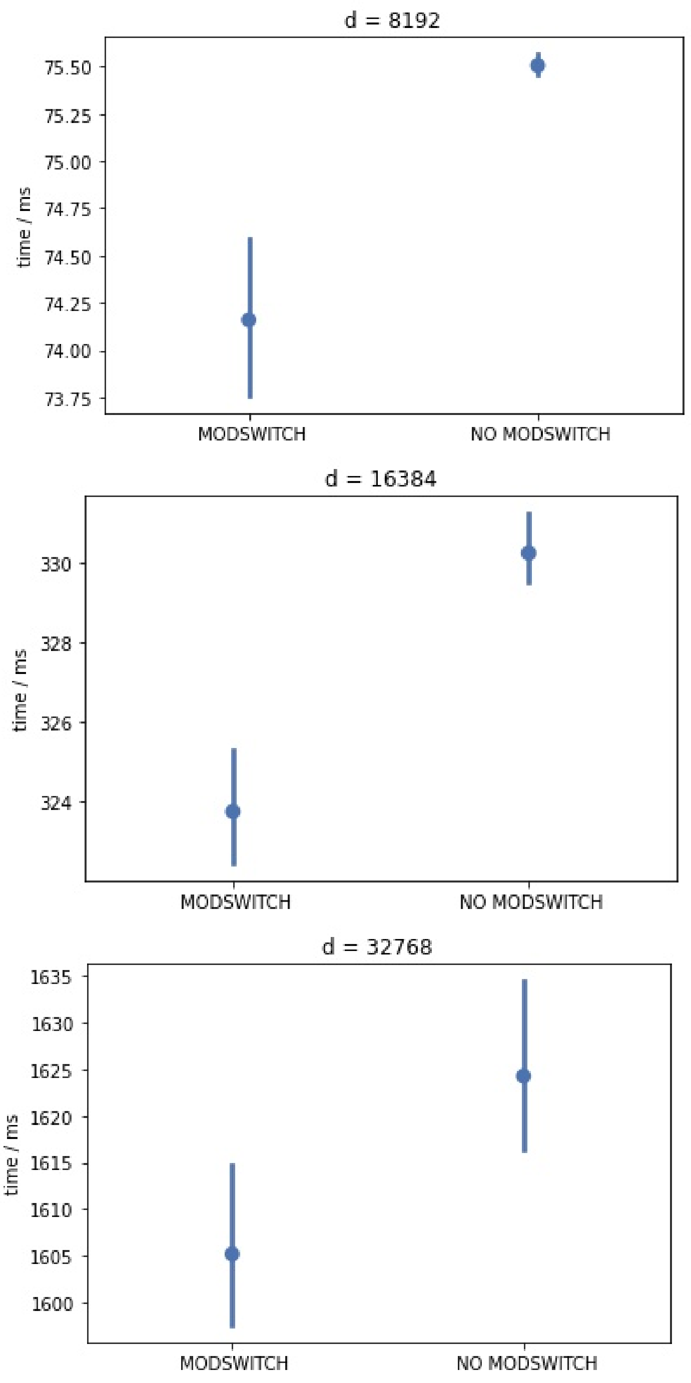

Figure 4.2: Point plots of average runtimes (ms) with error bars of the homomorphic evaluation of the function $f(x,y,z) = (x^4 + y) \cdot z^4$ using SEAL with (left) and without (right) the insertion of modulus switching operations using the Algorithms 2 and 3. Results are shown for polynomial degrees of 8192 (top), 16384 (middle), and 32768 (bottom).

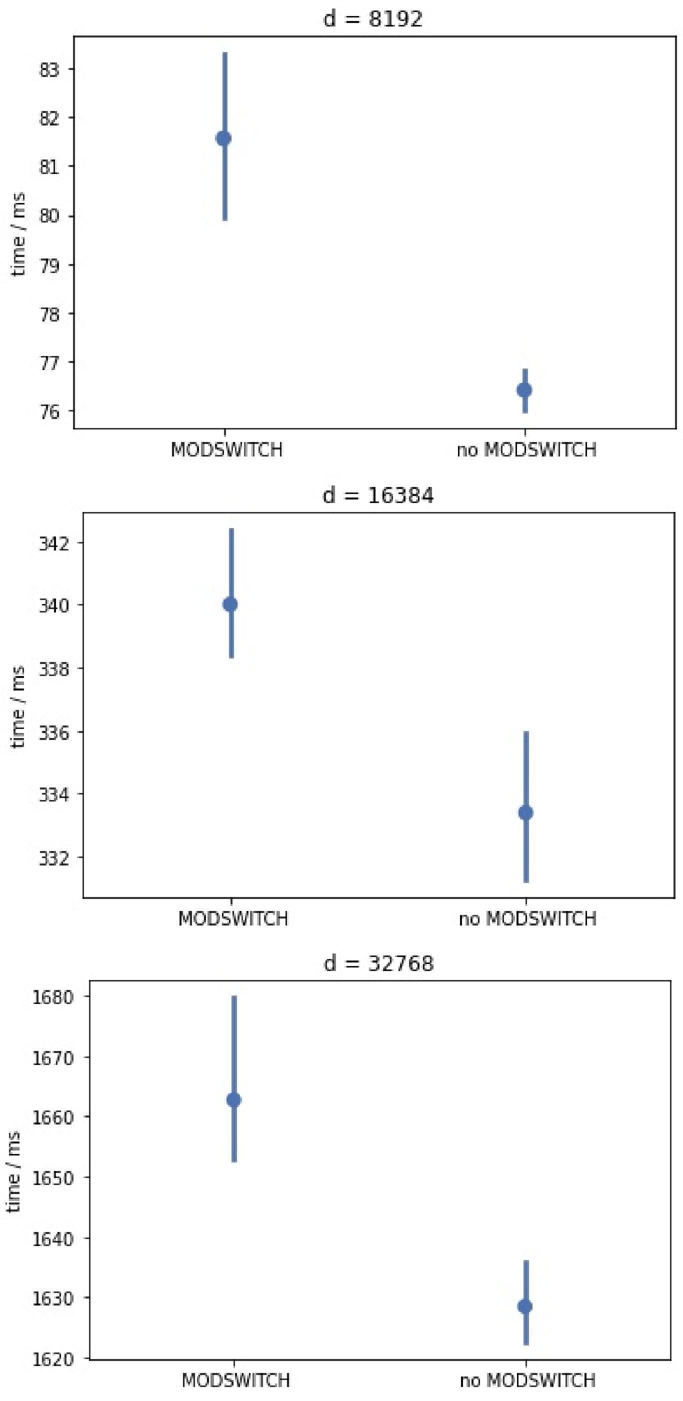

Figure 4.3: Point plots of average runtimes (ms) with error bars of the homomorphic evaluation of the function $(x^4 + y) \cdot z^4 + \sum_{i=1}^{12} a_i$ using SEAL with (left) and without (right) the insertion of modulus switching operations using the Algorithms 2 and 3. Results are shown for polynomial degrees of 8192 (top), 16384 (middle), and 32768 (bottom).

d	ModSwitch	no ModSwitch
8192	82(1)	76(0.2)
16384	340(1)	333(1)
32768	1662(7)	1628(4)

Table 4.5: Benchmarking results in *ms* of the homomorphic evaluation of the arithmetic circuit evaluating the function $(x^4 + y) \cdot z^4 + \sum_{i=1}^{12} a_i$ before (right column) and after (left column) insertion of modulus switching operations using Algorithms 2 and 3. Standard errors are shown in brackets.

To study the effect of automatic circuit rewriting based on noise-guided modulus switching on realistic arithmetic circuits, we further performed benchmarking on a subset of arithmetic circuits from the EPFL Combinatorial Benchmark Suite, a set of circuits that are intentionally left sub-optimal to allow testing of optimisation strategies. The set of arithmetic circuits chosen for testing consist of the circuits *Adder* (Adder), *Barrel Shifter* (Bar), and *Max* (Max) and circuits have been truncated to ensure the correctness of homomorphic evaluation. Runtime evaluation of the circuits with and without inserted modulus switching operations is presented in Tables 4.6 and 4.7, as well as in Figures 4.4, 4.5 and 4.6 for polynomial degrees $d = 16384$ and $d = 32768$, respectively. Speedup factors are presented in Table 4.3. The *Adder* circuit's evaluation time in fact increases upon the insertion of modulus switches by the algorithms, which is likely due to the large number of additions in the circuit. For the other two circuits, we observe some improvement in runtime. These results suggest that we cannot predict a significant runtime improvement in a general circuit's homomorphic evaluation and that the efficacy of the proposed rewriting algorithms is highly dependent on a circuit's number of performed additions.

d = 16384	Adder	Bar	Max
no ModSwitch	4902(14)	2421(3)	8914(46)
ModSwitch	4858(2)	2413(3)	8775(25)

Table 4.6: Benchmarking results in *ms* for the homomorphic evaluation of selected circuits from the EPFL Combinatorial Benchmarking Suite using SEAL for a polynomial degree $d = 16384$. Standard errors are given in brackets.

d = 32768	`Adder`	`Bar`	`Max`
no `ModSwitch`	26074(25)	13130(31)	53479(225)
`ModSwitch`	26322(68)	12830(19)	53520(186)

Table 4.7: Benchmarking results in ms for the homomorphic evaluation of selected circuits from the EPFL Combinatorial Benchmarking Suite using SEAL for a polynomial degree $d = 32768$. Standard errors are given in brackets.

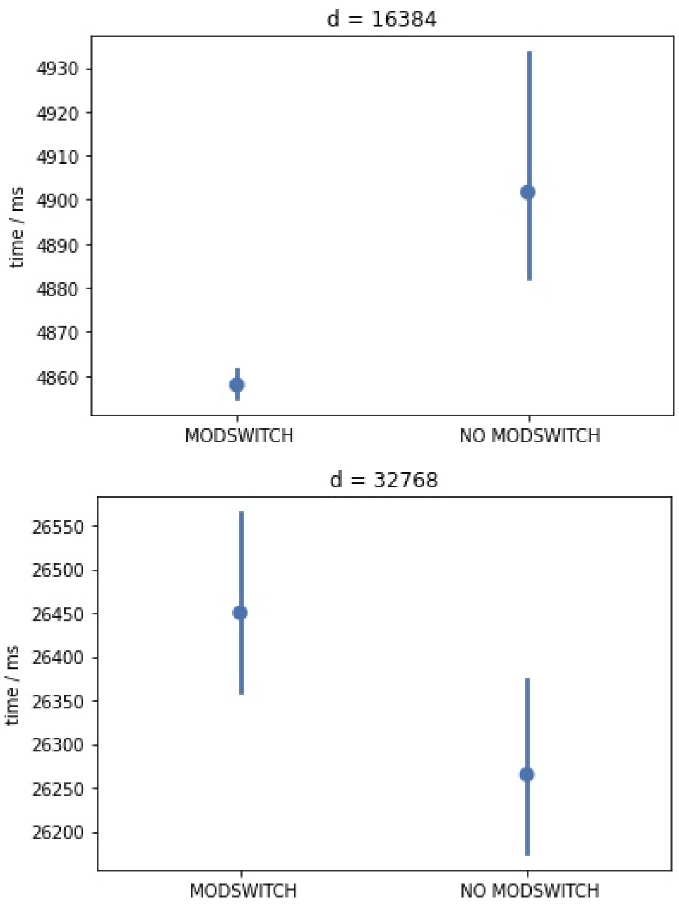

Figure 4.4: Point plots of average runtimes (ms) with error bars of the homomorphic evaluation of the Adder circuit using SEAL with (left) and without (right) the insertion of modulus switching operations using the Algorithms 2 and 3. Results are shown for polynomial degrees 16384 (top), and 32768 (bottom).

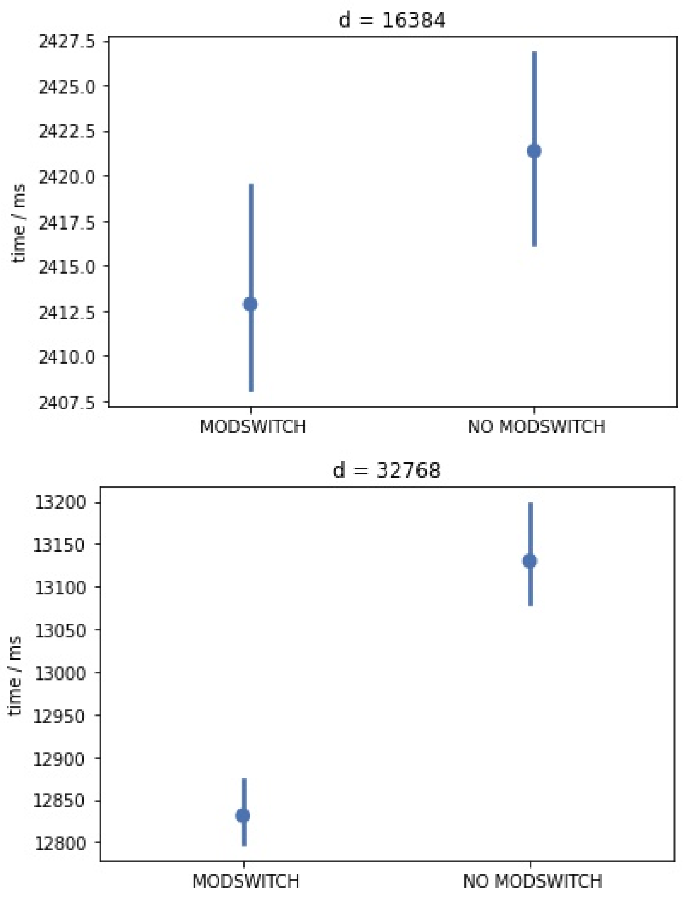

Figure 4.5: Point plots of average runtimes (ms) with error bars of the homomorphic evaluation of the Barrel Shifter circuit using SEAL with (left) and without (right) the insertion of modulus switching operations using the Algorithms 2 and 3. Results are shown for polynomial degrees 16384 (top), and 32768 (bottom).

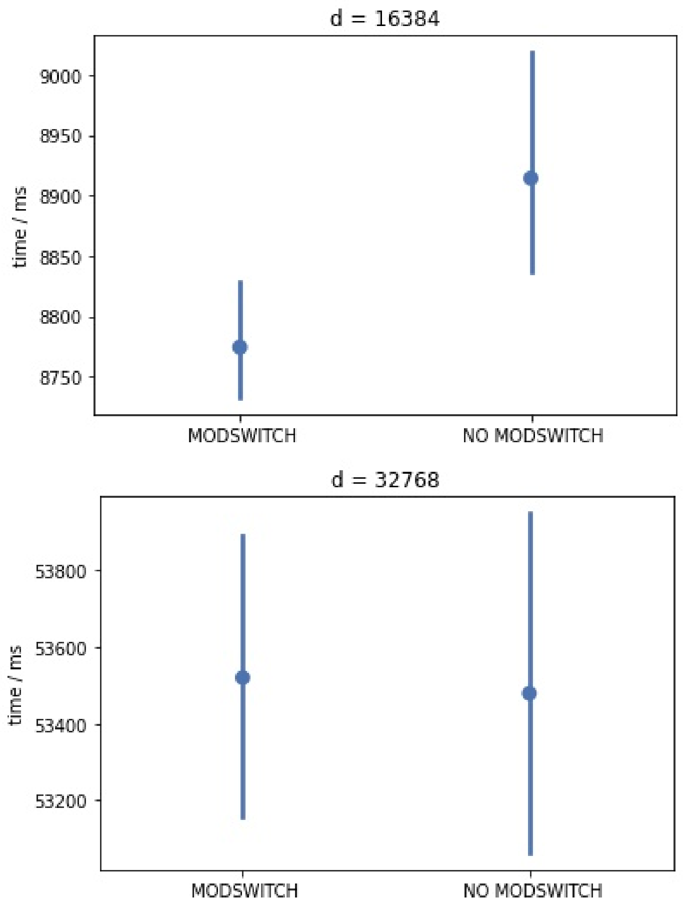

Figure 4.6: Point plots of average runtimes (ms) with error bars of the homomorphic evaluation of the Max circuit using SEAL with (left) and without (right) the insertion of modulus switching operations using the Algorithms 2 and 3. Results are shown for polynomial degrees 16384 (top), and 32768 (bottom).

To further elucidate potential runtime improvement as a function of performed multiplications in an arithmetic circuit, we measure runtimes of homomorphic evaluations of the circuit evaluating the function $\left(\sum_{i=1}^{n} x_i^2\right) \cdot y^8$ for $n = 2^r, (r \in \{0, \ldots, 9\})$. We apply modulus switching to the variables x_i before calculating their respective squares and to y^8 and compare runtimes to the circuits evaluating the same function without modulus switching for polynomial degrees $d = \{16384, 32768\}$. Overall runtimes are shown in Figure 4.7 and average speedup factors calculated from bootstrapping performed on the benchmarking data is shown in Figure 4.8.

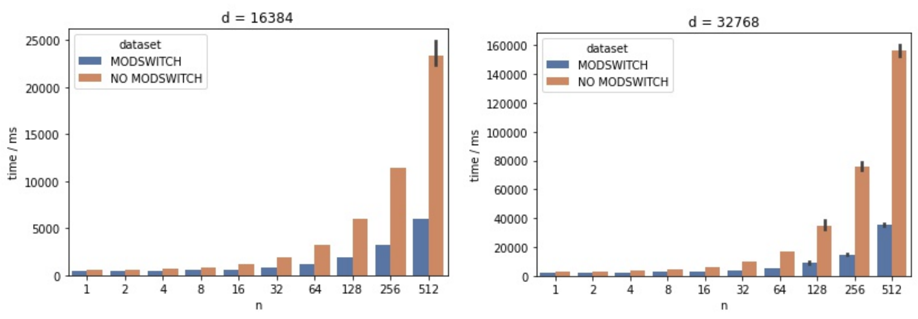

Figure 4.7: Runtime evaluation of the circuit evaluating $\left(\sum_{i=1}^{n} x_i^2\right) \cdot y^8$ for $n = 2^r$ as function of the number of summands x_i^2 for polynomial degrees 16384 (left) and 32768 (right). Runtimes in milliseconds are shown for the evaluation with (blue) and without (brown) inserted modulus switching operations before calculating squares of $x_i, i \in \{1, \ldots, n\}$ and after calculating y^8.

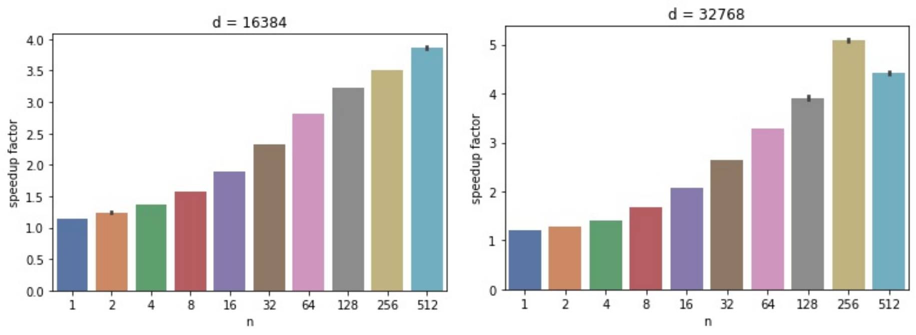

Figure 4.8: Average speedup factor of the homomorphic evaluation of $\left(\sum_{i=1}^{n} x_i^2\right) \cdot y^8$ for $n = 2^r$ as function of the number of summands x_i^2 for polynomial degrees 16384 (left) and 32768 (right). The speedup factor has been calculated as the fraction of the bootstrapped averages over the runtimes without modulus switching divided by the bootstrapped averages over the runtimes with modulus switching.

To illustrate the adverse effect the automatic insertion of modulus switching operations can have on performance, we consider the circuit evaluating the function $(x^4 + y) \cdot z^4 + \sum_{i=1}^{n} a_i$, again for $n = 2^r, (r \in \{0, \ldots, 9\})$. As before, Algorithm 2 inserts a single modulus switching operation that has to be compensated for by applying modulus switching to all operands a_i of the following additions. Runtimes are shown in Figure 4.9 and speedups are presented in Figure 4.10 for for polynomial degrees $d = \{16384, 32768\}$.

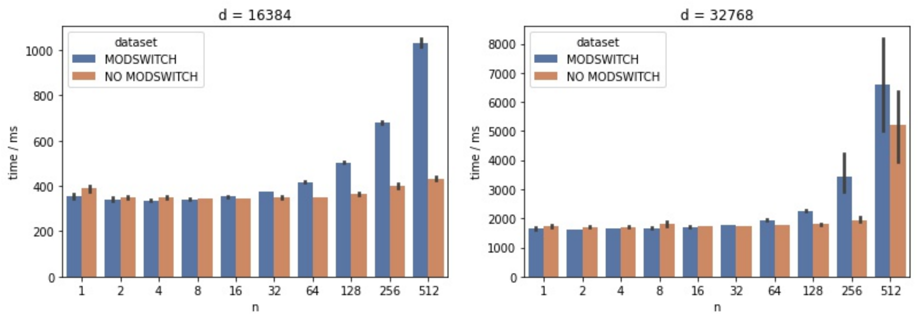

Figure 4.9: Runtime evaluation of the circuit evaluating $(x^4 + y) \cdot z^4 + \sum_{i=1}^{n} a_i$ for $n = 2^r$ as function of the number of summands a_i for polynomial degrees 16384 (left) and 32768 (right). Runtimes in milliseconds are shown for the evaluation with (blue) and without (brown) inserted modulus switching operations.

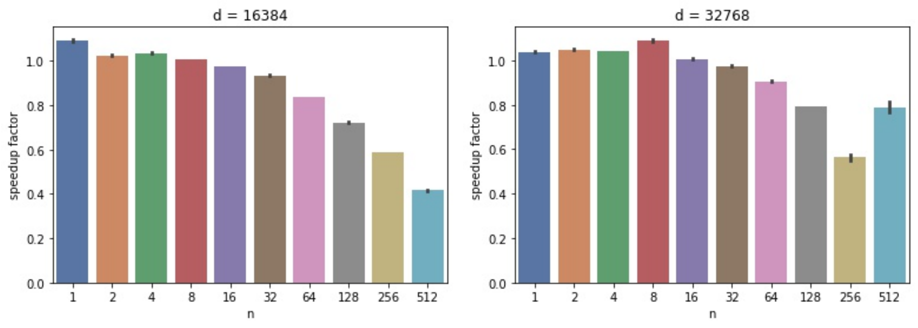

Figure 4.10: Average speedup factor of the homomorphic evaluation of $(x^4 + y) \cdot z^4 + \sum_{i=1}^{n} a_i$ for $n = 2^r$ as function of the number of summands x_i^2 for polynomial degrees 16384 (left) and 32768 (right). The speedup factor has been calculated as the fraction of the bootstrapped averages over the runtimes without modulus switching divided by the bootstrapped averages over the runtimes with modulus switching.

These results show that it is indeed possible to achieve both, performance improvement and deterioration thereof, depending on the nature of the arithmetic circuit, when applying Algorithms 2 and 3. Results indicate a significant increase of performance of a homomorphic evaluation, when applying modulus switching to operands of multiplications early in the computation. However, a negative effect on runtime has been observed when applying modulus switching before additions, highlighting the need of refining Algorithms 2 and 3. We suggest, that the algorithms proposed could be adapted to only accept the insertion of modulus switching when there are few or no additions affected by the changes. As seen in Tables 4.1 and 4.2, modulus switching operations are fast compared to multiplication, but slow compared to additions. Therefore, if one intends to achieve runtime improvements of the homomorphic evaluation of arithmetic circuits, it is advisable to apply modulus switching solely operands of multiplications. However, to avoid parameter mismatches and therefore ensure correctness of the circuit, this is not always feasible, since the automatic insertion of a modulus switching operation can cause the insertion of modulus switches to the operands of an an addition at later points in the computation. This is expected to limit the overall runtime improvement of the automatic insertion of modulus switches in a general circuit.

However, circuit rewriting based on noise-guided modulus switching is expected to have favourable effect on the throughput for hardware accelerated FHE by minimising the use of memory requirements of homomorphic evaluation.

Chapter 5

Discussion

The current work explores automatic circuit rewriting to optimise homomorphic evaluations. In particular, the quantity of ciphertext noise has been used to guide a user to potentially problematic areas in the given circuit, as well as to identify regions of circuits where rewriting is possible and potentially beneficial in terms of evaluation performance.

Noise heuristics Noise heuristics from [10] have been implemented in the compiling framework of the Automated Batching Compiler (ABC). Noise heuristic calculations are fast compared to actual homomorphic circuit evaluations and therefore are a well suited to serve as a base for rewriting algorithms. Noise heuristics generally overestimate actual ciphertext noise, but are indeed a useful quantity characterising the quality of arithmetic circuits.

Cone Rewriting We used the example of the existing Cone Rewriting optimization [2] to explore the state of the art of circuit transformations. While cone rewriting has many opportunities to improve efficiency of the evaluation of large binary circuits (see results in [2]), the method is less beneficial for arithmetic circuits, since many arithmetic circuits do not fulfill the specific requirements for the application of cone rewriting. As an example, one might consider the circuit evaluating the χ^2 test as done in [30], where cone-rewriting is not applicable. Further, binary circuits are usually evaluated using the TFHE scheme [9], which uses per-gate bootstrapping. In this setting, cone rewriting has a negative effect on efficiency because depth is not a factor and runtime is instead correlated only to the number of logical gates, which cone rewriting increases.

Noise-guided modulus switching We present a novel automatic circuit rewriting technique that is capable of rewriting circuits based on insights generated from noise heuristics. The algorithms insert modulus switching operations before homomorphic multiplications at appropriate points while ensuring correctness of decryption of the final result of the computation. When applying modulus switching to ciphertext operands of a multiplication, we observe improvement in runtime of homomorphic evaluation. However, since ciphertext moduli of operands of a binary operations must match, it is possible that in order to retain correctness of the arithmetic circuits, additional modulus switching operations must be inserted in different parts of the circuit, including before additions. Our results show that this can have a detrimental effect on runtime of homomorphic evaluation. Therefore, this optimisation strategy is–like the existing Cone Rewriting optimization–strongly dependent on the nature of the considered circuit.

Considering potential future developments in hardware design specifically for FHE, which will probably be bottle-necked by memory rather than compute, the developed algorithms could be beneficial for limiting ciphertext size from the earliest possible point on in the circuit. This should allow higher throughput and is expected to more significantly impact performance of FHE calculations on such accelerators.

Outlook Future work could entail a refinement of the set of algorithms that insert modulus switching. A strategy could be pursued that avoids the insertion of modulus switching operations before a multiplication if too many operands of additions need to undergo modulus switching as a consequence.

Another promising outlook is the effect of the developed algorithms on memory usage of homomorphic evaluations paving the way for optimisations for high throughput FHE specific hardware.

Appendix A

Noise Growth Heuristics

A.1 Theoretical Bounds

We present theoretical bounds on the noise of freshly encrypted ciphertexts, as well as noise resulting from homomorphic ring operations in SEAL. In this section, we use Definition 2.8 for the noise. In the following we shall use the notation $r_t(q) = q \mod t$ and consider a distribution χ such that $\|\chi\| < B$ for an integer B.

Initial Noise:

Lemma A.1 *Let* $\mathtt{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ *be a freshly encrypted ciphertext of the plaintext* $\mathbf{m} \in R_t$*. Then, the noise* $\mathbf{v}$ *contained in* $\mathtt{ct}$ *satisfies the bound*

$$\|\mathbf{v}\| \leq \frac{r_t(q)}{q}\|\mathbf{m}\| + \frac{tB}{q}(2d+1). \tag{A.1}$$

Proof See the Appendix of [22] for a proof. □

Addition of ciphertexts:

Lemma A.2 *Let* $\mathtt{ct}_1 = (\mathbf{c}_0, \dots, \mathbf{c}_j)$ *and* $\mathtt{ct}_2 = (\mathbf{d}_0, \dots, \mathbf{d}_k)$ *two ciphertexts that decrypt to* $\mathbf{m}_1 \in R_t$ *and* $\mathbf{m}_2 \in R_t$ *and with noises* $\mathbf{v}_1$ *and* $\mathbf{v}_2$*, respectively. Then, the noise in the sum* $\mathtt{ct}_{add} = \mathtt{ct}_1 + \mathtt{ct}_2$ *is* $\mathbf{v}_{add} = \mathbf{v}_1 + \mathbf{v}_2$ *and satisfies* $\|\mathbf{v}_{add}\| \leq \|\mathbf{v}_1\| + \|\mathbf{v}_2\|$*.*

Proof See the Appendix of [22] for a proof. □

Multiplication of ciphertexts:

Lemma A.3 *Let* $\mathtt{ct}_1 = (\mathbf{x}_0, \dots, \mathbf{x}_{j_1})$ *and* $\mathtt{ct}_2 = (\mathbf{y}_0, \dots, \mathbf{y}_{j_2})$ *be two ciphertexts encrypting* $\mathbf{m}_1$ *and* $\mathbf{m}_2$ *with noises* $\mathbf{v}_1$ *and* $\mathbf{v}_2$*, respectively. Let the number of non-zero terms in the polynomials* $\mathbf{m}_1$ *and* $\mathbf{m}_2$ *be bounded by* $N_{\mathbf{m}_1}$ *and* $N_{\mathbf{m}_2}$*, respectively.*

Then, the noise $\mathbf{v}_{mult}$ *in the product* $\mathtt{ct}_{mult} = \mathtt{ct}_1 \cdot \mathtt{ct}_2$ *is given by*

$$\mathbf{v}_{mult} = \mathbf{m}_1 \cdot \mathbf{v}_2 + \mathbf{m}_2 \cdot \mathbf{v}_1 + \mathbf{v}_1 \cdot \mathbf{v}_2 + (\mathbf{v}_1 \cdot \mathbf{a}_2 + \mathbf{v}_2 \cdot \mathbf{a}_1)t + \frac{t}{q} \sum_{i=0}^{j_1+j_2} \varepsilon_i \mathbf{s}^i, \tag{A.2}$$

where $\mathbf{a}_1$ *and* $\mathbf{a}_2$ *are polynomials with integer coefficients. The ciphertext* $\mathtt{ct}_{mult}$ *satisfies the bound*

$$\begin{aligned}\|\mathbf{v}_{mult}\| \leq & \left[(N_{m_1} + d)\|\mathbf{m}_1\| + \frac{dt}{2} \cdot \frac{d^{j_1+1} - 1}{d-1}\right] \|\mathbf{v}_2\| \\ & + \left[(N_{m_2} + d)\|\mathbf{m}_2\| + \frac{dt}{2} \cdot \frac{d^{j_2+1} - 1}{d-1}\right] \|\mathbf{v}_2\| \\ & + 3d\|\mathbf{v}_1\|\|\mathbf{v}_2\| + \frac{t}{2q}\left(\frac{d^{j_1+j_2+1} - 1}{d-1}\right),\end{aligned} \tag{A.3}$$

where d denotes the degree of the cyclotomic polynomial $f(x)$ from Definition 2.1.

Proof See the Appendix of [22] for a proof. □

Relinearisation:

Lemma A.4 *Let* $\mathtt{ct}$ *be a ciphertext of size* $M + 1$ *encrypting the plaintext* $\mathbf{m} \in R_t$*. Denote by* $\mathbf{v}$ *the noise of* $\mathtt{ct}$*. Let* $\mathtt{ct}_{relin}$ *be the ciphertext of size* $N + 1$ *obtained by relinearisation of the ciphertext* $\mathtt{ct}$*. Then,* $2 \leq N + 1 \leq M + 1$ *and the noise* $\mathbf{v}_{relin}$ *is given by*

$$\mathbf{v}_{relin} = \mathbf{v} - \frac{t}{q} \sum_{j=0}^{M-N-1} \sum_{i=0}^{\ell} \mathbf{e}_{(M-j),i} \mathbf{c}_{m-j}^{(i)}. \tag{A.4}$$

Its infinity norm can be bounded by

$$\|\mathbf{v}_{relin}\| \leq \|\mathbf{v}\| + \frac{t}{q}(M - N)nB(\ell + 1)w. \tag{A.5}$$

Proof See the Appendix of [22] for a proof. □

Ciphertext-plaintext addition:

Lemma A.5 *Let* $\mathtt{ct} = (\mathbf{x}_0, \dots, \mathbf{x}_j)$ *be a ciphertext that encrypts a plaintext* $\mathbf{m}_1 \in R_t$ *with noise* $\mathbf{v}$ *and let* $\mathbf{m}_2 \in R_t$ *be a plaintext. Let* $\mathtt{ct}_{\boldsymbol{padd}}$ *denote the ciphertext obtained by ciphertext-plaintext addition of* $\mathtt{ct}$ *and* $\mathbf{m}_2$*. Then the noise* $\mathbf{v}_{padd}$ *in* $\mathtt{ct}_{\boldsymbol{padd}}$ *satisfies*

$$\|\mathbf{v}_{padd}\| \leq \|v\| + \frac{r_t(q)}{q}\|\mathbf{m}_2\|. \tag{A.6}$$

Proof See the Appendix of [22] for a proof. □

Ciphertext-plaintext multiplication:

Lemma A.6 *Let* $\mathtt{ct} = (\mathbf{x}_0, \ldots, \mathbf{x}_j)$ *be a ciphertext that encrypts a plaintext* $\mathbf{m}_1 \in R_t$ *with noise* $\mathbf{v}$ *and let* $\mathbf{m}_2 \in R_t$ *be a plaintext. Let* $N_{\mathbf{m}_2}$ *be an upper bound on the number of non-zero terms of the polynomial* $\mathbf{m}_2$*. Then, the noise* $\mathbf{v}_{pmult}$*of the ciphertext* $\mathtt{ct}_{pmult}$ *resulting from the plaintext-ciphertext multiplication of* $\mathbf{m}_2$ *and* $\mathtt{ct}$ *satisfies*

$$\|\mathbf{v}_{pmult}\| \leq N_{\mathbf{m}_2}\|\mathbf{m}_2\|\|\mathbf{v}\|. \tag{A.7}$$

Proof See the Appendix of [22] for a proof. □

Negation:

Lemma A.7 *Let* $\mathtt{ct}$ *be a ciphertext that encrypts a plaintext* $\mathbf{m} \in R_t$ *with noise* $\mathbf{v}$ *and let* $\mathtt{ct}_{neg}$ *be its negation with noise* $\mathbf{v}_{neg}$*. Then,*

$$\mathbf{v}_{neg} = -\mathbf{v} \tag{A.8}$$

and

$$\|\mathbf{v}_{neq}\| = \|\mathbf{v}\|. \tag{A.9}$$

Proof See the Appendix of [22] for a proof. □

A.2 Heuristic Bounds

We show noise heuristics for encryption and homomorphic operations as shown in [18, 10], as they have been studied and compared with experimental bounds in [10] and therefore are most relevant for the current work. There are also noise heuristics presented in the SEAL manual [22], however noise heuristics presented in [18] present tighter bounds on the noise leading to a better approximation of noise derived from actual homomorphic operations. The noise heuristics discussed in [18] use the notion of invariant noise from Definition 2.8.

Encryption noise heuristics: Let $\mathtt{ct} = (\mathbf{c}_0, \mathbf{c}_1)$ be a freshly encrypted ciphertext, encrypting the plaintext $\mathbf{m} \in R_t$. Denote by $\mathbf{v}$ the noise of the ciphertext. The variance of the Gaussian variable $\|\mathbf{e} \cdot \mathbf{u} + \mathbf{e}_1 + \mathbf{e}_2 \cdot \mathbf{s}\|^{\mathrm{can}}$ is equal to $\tilde{\sigma}\sqrt{4d^2/3 + d}$ (see [18] for a proof). Hence, with high probability,

$$\|\mathbf{v}\|^{\mathrm{can}} \leq \frac{t}{q}\left(\frac{d(t-1)}{2} + 2\tilde{\sigma}\sqrt{12d^2 + 9d}\right). \tag{A.10}$$

With Lemma (2.9) it follows that the parameters $q, t, d, \tilde{\sigma}$ can be chosen such that

$$\frac{t}{q}\left(\frac{d(t-1)}{2} + 2\tilde{\sigma}\sqrt{12d^2 + 9d}\right) < \frac{1}{2}. \tag{A.11}$$

Noise heuristics after addition Let $\texttt{ct}_1$ and $\texttt{ct}_2$ be two ciphertexts encrypting having noises $\mathbf{v}_1$ and $\mathbf{v}_2$, respectively. Then the noise $\mathbf{v}_{\text{add}}$ in their sum satisfies

$$\|\mathbf{v}_{\text{add}}\|^{\text{can}} \leq \|\mathbf{v}_1\|^{\text{can}} + \|\mathbf{v}_2\|^{\text{can}}. \tag{A.12}$$

Noise heuristics after multiplication With very high probability, the invariant noise after the multiplication of two ciphertexts is bounded by

$$\begin{aligned}\|\mathbf{v}_{\text{mult}}\|^{\text{can}} &\leq t\sqrt{3d+2d^2}\,(\|\mathbf{v}_1\|^{\text{can}} + \|\mathbf{v}_2\|^{\text{can}}) \\ &\quad + 3\|\mathbf{v}_1\|^{\text{can}}\|\mathbf{v}_2\|^{\text{can}} + \frac{t}{q}\sqrt{3d+2d^2+4d^3/3}.\end{aligned} \tag{A.13}$$

Noise heuristics after relinearisation The canonical norm of the invariant noise after relinearisation can be bounded by

$$\|\mathbf{v}_{\text{relin}}\|^{\text{can}} \leq \|\mathbf{v}\|^{\text{can}} + \frac{t}{q}T\tilde{\sigma}d\sqrt{3(\ell+1)}. \tag{A.14}$$

Noise heuristics after multiplication with subsequent relinearisation The total invariant noise growth after multiplication with subsequent relinearisation is obtained by combining the above noise heuristics for multiplication and relinearisation:

$$\begin{aligned}\|\mathbf{v}_{\text{mult,relin}}\|^{\text{can}} &\leq t\sqrt{3d+2d^2}\,(\|\mathbf{v}_1\|^{\text{can}} + \|\mathbf{v}_2\|^{\text{can}}) + 3\|\mathbf{v}_1\|^{\text{can}}\|\mathbf{v}_2\|^{\text{can}} \\ &\quad + \frac{t}{q}\left(\sqrt{3d+2d^2+4d^3/3} + T\tilde{\sigma}d\sqrt{3(\ell+1)}\right).\end{aligned} \tag{A.15}$$

Modulus Switching noise heuristics Noise heuristics for modulus switching is presented in [10]. Let $\texttt{ct}$ be a ciphertext encrypting a plaintext $\mathbf{m}$ with invariant noise $\mathbf{v}$ with respect to a ciphertext modulus q. Let $\texttt{ct}_{\text{mod}}$ be the ciphertext encrypting $\mathbf{m}$ obtained by modulus switching to the modulus t. Then, with high probability, the invariant noise $\mathbf{v}_{\text{mod}}$ of $\texttt{ct}_{\text{mod}}$ satisfies

$$\|\mathbf{v}_{\text{mod}}\|^{\text{can}} \leq \|\mathbf{v}\|^{\text{can}} + \frac{t}{p}\cdot\sqrt{3d+2d^2}. \tag{A.16}$$

Plaintext-Ciphertext operation heuristics For plaintext-ciphertext operations, one takes the theoretical bounds as seen in Lemmas A.5 and A.6 as noise heuristics.

A.3 Noise Heuristics in ABC

In an effort to improve circuit design using noise heuristics, we have implemented the noise heuristics from [18] for ciphertext-ciphertext operations and from [22] for ciphertext-plaintext operations, as well as the `ModSwitch` heuristic from [10].
As in [10], we calculate noise heuristics for $d \in \{4096, 8192, 16384\}$ and a plaintext modulus of $t = 65537$, a prime congruent to $1 \mod d$, which is required to enable batching in SEAL. All other parameters were kept as the default parameters from SEAL. We note that SEAL selects a suitable, but not necessarily optimal, coefficient modulus q based on the polynomial degree d. Tables A.1 and A.2 show the noise heuristics for ciphertext encryption and ciphertext-ciphertext operations, ciphertext-plaintext operations, as well as modulus switching, taken from [10] as well as the heuristics calculated using the ABC framework. As in [10], we observe an overestimation of the noise value in comparison to SEAL. We also observe discrepancies when comparing with results from [10], which is caused by a different coefficient modulus obtained by different versions of SEAL. However, for the analysis of arithmetic circuits, the noise heuristics as implemented in the ABC have suitable properties for identifying regions of noise growth.

	Enc		Add		Mult		`ModSwitch`	
d	[10]	E	[10]	E	[10]	E	[10]	E
4096	71	27	70	26	41	3	25	27
8192	179	128	178	128	148	104	133	128
16384	398	342	397	341	366	317	352	342

Table A.1: Calculated noise heuristics for ciphertext-ciphertext homomorphic operations, as well as encryption of a fresh ciphertext. Column E gives the noise budget calculated using noise heuristics from [18] and [10]. The column denoted by [10] shows the result from the noise budget estimate by Costache *et al.* using the same noise heuristics ([18]).

	Add	Mult
d	E	E
4096	26	0
8192	127	99
16384	341	312

Table A.2: Calculated noise heuristics for ciphertext-plaintext homomorphic operations, as well as encryption of a fresh ciphertext. Column E gives the noise budget calculated using noise heuristics from [22].

Computational Performance of evaluating Arithmetic Circuits using Noise Heuristics Calculating only noise heuristics is significantly faster than actual evaluations using SEAL, making the noise heuristics an efficient means to analyse noise behaviour in given arithmetic circuits. To see this, the total runtime of the evaluation of an array of arithmetic circuits using the ABC has been collected according to specifications in Section 4.1.2. Evaluations have been performed on single binary ciphertext-ciphertext and ciphertext-plaintext operations, as well as for a selection of other arithmetic circuits. The resulting runtimes of evaluating arithmetic circuits as ASTs using SEAL and heuristics from [18] are shown in milliseconds in Figure A.3.

	SEAL	E
Encryption	320	118
Ctxt-Ctxt Add	317	125
Ctxt-Ctxt Mult	320	118
Ctxt-Ptxt Add	294	128
Ctxt-Ptxt Mult	295	125
$x \cdot x \cdot x \cdot x \cdot x$	330	119

Table A.3: Runtimes (ms) of evaluating single binary operations and the function $x \cdot x \cdot x \cdot x$ using SEAL (column denoted by SEAL) and noise heuristics from [18] (column denoted by E).

Bibliography

[1] David W. Archer, José Manuel Calderón Trilla, Jason Dagit, Alex Malozemoff, Yuriy Polyakov, Kurt Rohloff, and Gerard Ryan. RAMPARTS: A Programmer-Friendly System for Building Homomorphic Encryption Applications. In *Proceedings of the 7th ACM Workshop on Encrypted Computing & Applied Homomorphic Cryptography - WAHC'19*, pages 57–68, London, United Kingdom, 2019. ACM Press.

[2] Pascal Aubry, Sergiu Carpov, and Renaud Sirdey. Faster homomorphic encryption is not enough: improved heuristic for multiplicative depth minimization of Boolean circuits. Technical Report 963, 2019.

[3] Jean-Claude Bajard, Julien Eynard, Anwar Hasan, and Vincent Zucca. A Full RNS Variant of FV like Somewhat Homomorphic Encryption Schemes. Technical Report 510, 2016.

[4] Fabian Boemer, Anamaria Costache, Rosario Cammarota, and Casimir Wierzynski. nGraph-HE2: A High-Throughput Framework for Neural Network Inference on Encrypted Data. *arXiv:1908.04172 [cs]*, August 2019. arXiv: 1908.04172.

[5] Zvika Brakerski. Fully Homomorphic Encryption without Modulus Switching from Classical GapSVP. Technical Report 078, 2012.

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) Fully Homomorphic Encryption without Bootstrapping. *ACM Transactions on Computation Theory*, 6(3):1–36, July 2014.

[7] Sergiu Carpov, Pascal Aubry, and Renaud Sirdey. A multi-start heuristic for multiplicative depth minimization of boolean circuits. Technical Report 483, 2017.

[8] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic Encryption for Arithmetic of Approximate Numbers. Technical Report 421, 2016.

[9] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. Faster Fully Homomorphic Encryption: Bootstrapping in less than 0.1 Seconds. Technical Report 870, 2016.

[10] Anamaria Costache, Kim Laine, and Rachel Player. Evaluating the effectiveness of heuristic worst-case noise analysis in FHE. Technical Report 493, 2019.

[11] Roshan Dathathri, Blagovesta Kostova, Olli Saarikivi, Wei Dai, Kim Laine, and Madanlal Musuvathi. EVA: An Encrypted Vector Arithmetic Language and Compiler for Efficient Homomorphic Computation. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 546–561, June 2020. arXiv: 1912.11951.

[12] Roshan Dathathri, Olli Saarikivi, Hao Chen, Kim Laine, Kristin Lauter, Saeed Maleki, Madanlal Musuvathi, and Todd Mytkowicz. CHET: an optimizing compiler for fully-homomorphic neural-network inferencing. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 142–156, Phoenix AZ USA, June 2019. ACM.

[13] Junfeng Fan and Frederik Vercauteren. Somewhat practical fully homomorphic encryption. *IACR Cryptology ePrint Archive*, page 2012.

[14] Craig Gentry. A Fully Homomorphic Encryption Scheme. *PhD. Dissertation, Stanford University*, page 209, 2009.

[15] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic Evaluation of the AES Circuit. Technical Report 099, 2012.

[16] Shai Halevi, Yuriy Polyakov, and Victor Shoup. An Improved RNS Variant of the BFV Homomorphic Encryption Scheme. In Mitsuru Matsui, editor, *Topics in Cryptology – CT-RSA 2019*, Lecture Notes in Computer Science, pages 83–105, Cham, 2019. Springer International Publishing.

[17] Vincent HERBERT. Automatize parameter tuning in Ring-Learning-With-Errors-based leveled homomorphic cryptosystem implementations. Technical Report 1402, 2019.

[18] I. Iliashenko. *Optimisations of Fully Homomorphic Encryption*. PhD thesis, 2019.

[19] Andrey Kim, Yuriy Polyakov, and Vincent Zucca. Revisiting Homomorphic Encryption Schemes for Finite Fields. Technical Report 204, 2021.

[20] Miran Kim, Arif Harmanci, Jean-Philippe Bossuat, Sergiu Carpov, Jung Hee Cheon, Ilaria Chillotti, Wonhee Cho, David Froelicher, Nicolas Gama, Mariya Georgieva, Seungwan Hong, Jean-Pierre Hubaux, Duhyeong Kim, Kristin Lauter, Yiping Ma, Lucila Ohno-Machado, Heidi Sofia, Yongha Son, Yongsoo Song, Juan Troncoso-Pastoriza, and Xiaoqian Jiang. Ultra-Fast Homomorphic Encryption Models enable Secure Outsourcing of Genotype Imputation. July 2020.

[21] Miran Kim, Yongsoo Song, Baiyu Li, and Daniele Micciancio. Semi-Parallel logistic regression for GWAS on encrypted data. *BMC Medical Genomics*, 13(S7):99, July 2020.

[22] Kim Laine. Simple Encrypted Arithmetic Library 2.3.1. *Microsoft Research.*

[23] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On Ideal Lattices and Learning with Errors Over Rings. Technical Report 230, 2012.

[24] Microsoft. Asure Run. *https://github.com/microsoft/SEAL-Demo/tree/master/AsureRun*, 2019.

[25] Jürgen Neukirch. *Algebraic Number Theory*. Grundlehren der mathematischen Wissenschaften. Springer-Verlag, Berlin Heidelberg, 1999.

[26] Mohammad Saidur Rahman, Ibrahim Khalil, Mohammed Atiquzzaman, and Xun Yi. Towards privacy preserving AI based composition framework in edge networks using fully homomorphic encryption. *Engineering Applications of Artificial Intelligence*, 94:103737, September 2020.

[27] Kazue Sako, editor. *Topics in Cryptology - CT-RSA 2016: The Cryptographers' Track at the RSA Conference 2016, San Francisco, CA, USA, February 29 - March 4, 2016, Proceedings*, volume 9610 of *Lecture Notes in Computer Science.* Springer International Publishing, Cham, 2016.

[28] N. P. Smart and F. Vercauteren. Fully Homomorphic SIMD Operations. Technical Report 133, 2011.

[29] Tim van Elsloo, Giorgio Patrini, and Hamish Ivey-Law. SEALion: a Framework for Neural Network Inference on Encrypted Data. *arXiv:1904.12840 [cs, stat]*, April 2019. arXiv: 1904.12840.

[30] Alexander Viand, Patrick Jattke, and Anwar Hithnawi. SoK: Fully Homomorphic Encryption Compilers. *arXiv:2101.07078 [cs]*, January 2021. arXiv: 2101.07078.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**

With my signature I confirm that
- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.