Behaviour of Algebraic Ciphers in Fully Homomorphic Encryption

Toprakhisar, Dilara

Award date:
2021

Technische Universiteit Eindhoven University of Technology

Department of Mathematics and Computer Science
Coding Theory and Cryptology Group

Behaviour of Algebraic Ciphers in Fully Homomorphic Encryption

Master Thesis

Dilara Toprakhisar

Supervisors:
Capt. (r.) Dr. Tomer Ashur
Dr. Ir. L. A. M. (Berry) Schoenmakers
Dr. Boris Škorić

Eindhoven, August 2021

Preface

I was 15 years old when I got to know the place that introduced the fascination of ”thinking” to me: Nesin Mathematics Village. Then, every summer I found myself there enchanted by thinking, learning, and the wisdom of Ali Nesin who is the founder of the Village. I always admired his boundless knowledge, the way he values thinking and learning, and that he spreads his knowledge. This is how I decided that I want to be in a lifelong thinking and learning experience, and spread my knowledge like him.

Now, with this thesis, I am closing a chapter in my life which I reached a milestone on the way to my goal. In this challenging way, I was lucky enough to have the support of many people. First and foremost, I would like to express my gratitude to my supervisor Tomer Ashur who is the best supervisor a student would like to work with. He invested a big amount of time not only guiding me in this project, but also helping me to take the first steps into the academic life. Apart from that, he contributed my personal development with his friendly advises. I am also thankful to Berry Schoenmakers and Boris Škorić for being in the assessment committee. Moreover, I thank Mairon for all the cryptanalysis of our work.

I appreciate the support I received from the Amandus H. Lundqvist Scholarship Program for making it possible for me to attend TU/e for my studies.

I would like to thank my friends who were always there to help me when I get stressed, and celebrate when I achieve something. But most importantly, for making the Netherlands a new home for me with all those lovely memories.

Also, I would like to express my gratitude to my parents and best friends for always having faith in me, and supporting me despite the huge distance.

But most of all, my utmost appreciation goes to my brother Burak who was always there being my whole family (together with Aline, a huge thanks to her as well) while we were both away from home.

Abstract

Traditional block ciphers like AES and 3DES are used in a wide range of application areas in practice: wireless security, IoT devices, processor security, SSL/TLS, *etc*. What these application areas have in common is the need for efficient software and hardware implementations. Consequently, traditional block ciphers are optimized for speed and resource consumption. However, the rapidly increasing number of applications that employ advanced cryptographic protocols such as multi-party computation or zero-knowledge proofs shifts the optimization focus to a different metric, *arithmetic complexity*, determined by the number of non-linear operations. Ciphers that are optimized with respect to arithmetic complexity are called algebraic ciphers.

Although *fully homomorphic encryption* (FHE) is a widely used advanced cryptographic protocol, designing algebraic ciphers employed by an FHE application is still an open problem in modern cryptography. This thesis stands as a contribution to this research problem.

In this thesis I evaluate the behavior of algebraic ciphers when implemented as a circuit in an FHE protocol. To this end, I present a state-of-the-art comparison of AES as a traditional block cipher, and VISION and RESCUE as algebraic ciphers implemented using HElib. The preliminary results of this comparative study was published in SiTB [35]. Next, I establish the properties affecting the efficiency of an FHE implementation that evaluates an algebraic cipher as a circuit. Then, I identify the bottlenecks of the FHE implementations that evaluate the VISION and RESCUE circuits, and work to improve them. This results in two new ciphers: SELJUK and CHAGHRI. Our first proposal SELJUK was accepted to CFail [36]. The FHE implementations of the SELJUK and CHAGHRI circuits are benchmarked. These benchmarks confirm that SELJUK is more efficient than both VISION and RESCUE when implemented in FHE. Moreover, CHAGHRI achieves a compact algebraic description optimized to FHE and by far it is more efficient than any other cipher, including the next best candidate, AES. This work is being prepared for submission to IEEE Symposium on Security and Privacy (IEEE S&P; Oakland). In addition, we are working to merge our code implementing these algebraic ciphers back to HElib.

Contents

Notation

$\mathbb{Z}$	ring of integers
$\mathbb{Z}_m$	quotient ring $\mathbb{Z}/m\mathbb{Z}$, ring of integers modulo m, m integer
$\mathbb{Z}_m^*$	group of units in $\mathbb{Z}_m$ (*i.e.*, elements with multiplicative inverses)
$\Phi_m(X)$	m^{th} cyclotomic polynomial
$\mathbb{A} = \mathbb{Z}[X]/\Phi_m(X)$	polynomial ring defined by $\Phi_m(X)$
$\phi(\cdot)$	Euler's totient function
$\mathbb{A}_q = \mathbb{A}/q\mathbb{A}$	$\mathbb{Z}[X]/(\Phi_m(X), q)$, the set of integer polynomials of degree up to $\phi(m) - 1$ reduced modulo q
$\mathbb{F}_{p^n}$	the finite field of order p^n
$\odot$	the Hadamard product (element-wise product)
$\lvert\mathbb{G}\rvert$	the order of the group $\mathbb{G}$

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

Block ciphers are fundamental components in modern cryptography. They can be found in a broad range of application domains. An IoT device that has a small processor and/or a limited hardware area, and a router that manages high speed data transfers are two examples of these application domains. As there are many potential application domains, there are many constraints in terms of security and efficiency when designing a block cipher.

Traditional block ciphers such as AES consists of linear and non-linear layers. When these layers are carefully chosen, ciphers resist well studied attacks. In addition to being secure, traditional block ciphers are designed to be efficient in hardware and software implementations. Their design focuses on running time, gate count, or memory/power consumption depending on the target application domain. For instance, while gate count and memory/power consumption are the main constraints for an implementation targeting an IoT device, latency is the main constraint for an implementation targeting a router that handles high speed communication. However, these efficiency constraints are different than the ones that make block ciphers efficient in advanced cryptographic protocol applications. *Multi-Party Computation* (MPC), *Zero-Knowledge* (ZK) *proofs*, and *Fully Homomorphic Encryption* (FHE) are examples of such advanced cryptographic protocols.

Consider the following scenario in which an advanced cryptographic protocol employs a block cipher: a client sends its data encrypted under an FHE scheme to a cloud server that operates on encrypted data. As we will see in Section 2.3, depending on the complexity of the function that is performed by the cloud server, the parameters of the scheme might be eminently large. This would consequently increase the size of the ciphertext, adding unwanted overhead to the communication. One solution to this problem is *transciphering* meaning all the private data sent by a client can be encrypted under a block cipher. Then, the server decrypts homomorphically and consequentially, they are able to operate on encrypted data without additional overhead to the communication [26].

MPC, ZK proofs, and FHE coincide in being described via algebraic operations. These operations can be translated into arithmetic computations and vice

versa. Converting computations into a sequence of algebraic operations over a finite field is called *arithmetization* and it was first applied to cryptographic protocols by Lund *et al*. [25].

As advanced cryptographic protocols increase in popularity, different design constraints are emerging. These in turn give rise to new designs such as MIMC [1], JARVIS [3], VISION, and RESCUE [2]. Unlike traditional block ciphers, the design rationale of these algorithms is to minimize the number of non-linear arithmetic operations, so called arithmetic complexity. Consequently, they improve the efficiency of the protocol employing them. Such ciphers are collectively known as algebraic ciphers.[1] The relevant attacks and security of algebraic ciphers are thus also different.

Designing an algebraic cipher is a research area that is still evolving. Besides novel cipher proposals, there are several design strategies introduced that stand as a framework for designing algebraic ciphers. The *Marvellous* design strategy [2] and the *Hades* design strategy [18] are examples of such design strategies. The ciphers that are proposed following these design strategies are shown to be efficient in ZK and MPC applications.

Although numerous algebraic ciphers were proposed for ZK and MPC applications, little to no attention was given to algebraic ciphers in the context of FHE. As an incontrovertible fact, FHE is an effective tool to remove privacy barriers obstructing data sharing. Therefore, an FHE-optimized algebraic cipher still stands as a research area that needs to be improved.

This thesis provides an exploratory study of algebraic ciphers when implemented as a circuit in an FHE protocol. Three ciphers are implemented: AES [16], VISION, and RESCUE; and their performance is adequately compared. This is an extended version of the comparison provided in SiTB [35] for AES and VISION. Then, by addressing the key factors affecting the performance I suggest two novel FHE-optimized ciphers: SELJUK and CHAGHRI. The former was accepted to CFail [36] and the latter is currently being prepared for submission to IEEE S&P 2022. Furthermore, implemented using the HElib software library CHAGHRI is 2.3 times faster than AES, making it, to the best of my knowledge, the most efficient block cipher in this setting.

In this thesis, I aim to answer the following research questions:

- How do algebraic ciphers behave in FHE when compared to their traditional counterparts?
- What are the specific efficiency metrics that make algebraic ciphers efficient in FHE?

[1] Some older works use the term arithmetization-oriented ciphers which is now considered obsolete.

1.1 The Structure of This Thesis

This thesis is structured as follows: in *Chapter 2*, some mathematical background, relevant features of AES, and an introduction to FHE including the FHE scheme used in this work are presented. *Chapter 3* deals with algebraic ciphers: general design considerations of algebraic ciphers, the *Marvellous* design strategy, and a description of VISION and RESCUE. Then, in *Chapter 4*, working packed implementations of leveled homomorphic encryption that can evaluate the AES-128 [16], VISION, and RESCUE circuits are described. Subsequently, in *Chapter 5*, the findings of the exploratory study that compares VISION and RESCUE with AES are presented. Additionally, the efficiency metrics that improves the performance of the algebraic ciphers are identified. Then, in *Chapter 7*, our first algebraic cipher design SELJUK is described. This design serves to the purpose of exploring the efficiency metrics further and designing our second algebraic cipher, CHAGHRI. Next, in *Chapter 8*, CHAGHRI is introduced and the specifications of the cipher are given. Finally, in *Chapter 9*, a conclusion given.

Chapter 2

Preliminaries

In this chapter I recall some prior knowledge required for this thesis. Section 2.1 provides the necessary mathematical background. Then, Section 2.2 gives an overview of AES. Finally, Section 2.3 introduces the basics of FHE and the FHE scheme used in this thesis.

2.1 Mathematical Background

Definition 2.1.1 (Group)**.** A group $< G, + >$ is a set G together with an operation, denoted by '$+$', defined on elements of G:

$$+ \colon G \times G \to G \colon (a, b) \mapsto a + b. \tag{2.1}$$

Then, $< G, + >$ satisfies the following properties:

- Closure: $\forall a, b \in G \colon a + b \in G$.
- Identity: $\exists \mathbf{0} \in G, \forall x \in G \colon \mathbf{0} + x = x + \mathbf{0} = x$.
- Associativity: $\forall a, b, c \in G \colon (a + b) + c = a + (b + c)$.
- Inverse: $\forall a \in G, \exists b \in G \colon a + b = \mathbf{0}$.

Definition 2.1.2 (Subgroup)**.** Let $< G, + >$ be a group, and H be a subset of G. If H forms a group under the same operation '$+$', then $< H, + >$ is said to be a subgroup of $< G, + >$.

Definition 2.1.3 (Ring)**.** A ring $< R, +, \cdot >$ is a set R together with two operations denoted by '$+$' and '$\cdot$', defined on elements of R:

$$\begin{aligned} + &: R \times R \to R : (a, b) \mapsto a + b, \\ \cdot &: R \times R \to R : (a, b) \mapsto a \cdot b. \end{aligned} \tag{2.2}$$

Then, $< R, +, \cdot >$ satisfies the following properties:

- $< R, + >$ is an Abelian group (commutative): $\forall a, b \in R : a + b = b + a$.
- '$\cdot$' is closed and associative over R, and there is an identity element for '$\cdot$' in R.
- '$+$' and '$\cdot$' are related by the law of *distributivity*: $\forall a, b, c \in R : (a+b) \cdot c = (a \cdot c) + (b \cdot c)$.

Definition 2.1.4 (Field)**.** A field $< F, +, \cdot >$ is a set F together with two operations denoted by '$+$' and '$\cdot$', defined on elements of F:

$$\begin{aligned} + &: F \times F \to F : (a, b) \mapsto a + b, \\ \cdot &: F \times F \to F : (a, b) \mapsto a \cdot b. \end{aligned} \tag{2.3}$$

Then, $< F, +, \cdot >$ satisfies the following properties:

- $< F, +, \cdot >$ is a commutative ring (the operation '$\cdot$' is commutative).
- $F \setminus \{\mathbf{0}\}$ (the set F without the additive identity **0**) is an abelian group under '$\cdot$'.

A field is denoted by $\mathbb{F}$.

Definition 2.1.5 (Finite Field)**.** A finite field is a field with a finite number of elements. The number of elements in the set is said to be the order of the field. A field of order q exists if and only if q is of the form p^n for a positive integer n and a prime p. Then, p is said to be the characteristic of the field. A finite field is denoted by $\mathbb{F}_{p^n}$.

Definition 2.1.6 (Subfield)**.** Let $\mathbb{F}$ be a field, and S be a subset of $\mathbb{F}$. If S forms a field under the same operations '$+$' and '$\cdot$', then S is said to be a subfield of $\mathbb{F}$. $\mathbb{F}$ is also said to be an *extension* of S.

Definition 2.1.7 (Polynomial over a Field)**.** A polynomial over a field $\mathbb{F}$ is an expression of the form

$$b(x) = b_{n-1}x^{n-1} + b_{n-2}x^{n-2} + ... + b_2x^2 + b_1x + b_0,$$

where x is the indeterminate of the polynomial and the coefficients $b_i \in \mathbb{F}$. The *degree* of the polynomial is the highest of the degrees of the monomials with non-zero coefficients.
If $b_i = 0$ for all i, then the degree of the polynomial is said to be $-\infty$. If the degree of the polynomial is zero or $-\infty$, then the polynomial is said to be *trivial*. Otherwise, the polynomial is said to be *nontrivial*.

Definition 2.1.8 (Irreducible Polynomial)**.** A polynomial over a field $\mathbb{F}$ is said to be irreducible if it cannot be factored into nontrivial polynomials over $\mathbb{F}$.

Definition 2.1.9 (Linearized Polynomial)**.** A polynomial $L(x)$ over a finite field $\mathbb{F}_{q^n}$ is said to be a linearized polynomial if the exponents of all the monomials are powers of q:

$$L(x) = \sum_{i=0}^{n-1} a_i x^{q^i}, \text{where } a_i \in \mathbb{F}_{q^n}.$$

Definition 2.1.10 (Linearized Affine Polynomial)**.** Let $L(x)$ be a linearized polynomial over a finite field $\mathbb{F}_{q^n}$. A polynomial $A(x)$ of the form:

$$A(x) = L(x) + a_{-1} = \sum_{i=0}^{n-1} a_i x^{q^i} + a_{-1}, \text{where } a_i \in \mathbb{F}_{q^n}$$

is said to be an $\mathbb{F}_q$-linearized affine polynomial over $\mathbb{F}_{q^n}$.

Definition 2.1.11 (Permutation Polynomial)**.** A polynomial $L(x)$ over a field $\mathbb{F}$ is a permutation polynomial if the map $x \mapsto L(x)$ where $x \in \mathbb{F}$ is a bijection. In other words, $L(x)$ maps distinct elements in $\mathbb{F}$ to distinct elements in $\mathbb{F}$ (*i.e.,* one-to-one) and for every y in $\mathbb{F}$, there is an x in $\mathbb{F}$ such that $L(x) = y$ (*i.e.,* onto).

Theorem 2.1.1 (Lagrange's Theorem)**.** If H is a subgroup of a group G, then the order of H divides the order of G.

2.2 Advanced Encryption Standard (AES)

AES is a key-iterated block cipher that repeatedly applies a round function R to its state S. The state is an element in the vector space $\mathbb{F}_{2^8}^{4\times 4}$ at a specific point in the execution [9]. The elements of the AES state are defined by the Rijndael polynomial $x^8 + x^4 + x^3 + x + 1$, and viewed as bytes. AES encrypts 128-bit plaintext blocks, and supports 128-, 192-, and 256-bit keys. The number of rounds is a function in the key size: 10 rounds for 128-bit keys; 12 rounds for 192-bit keys; and 14 rounds for 256-bit keys.

An AES round consists of four steps with a slight exception in the last round:

- SubBytes: a fixed permutation is applied to each byte of the state. This permutation consists of an inversion ($g : a \mapsto b = a^{-1}$ with 0 mapped to 0) followed by an affine transformation $f : \mathbb{F}_2^8 \mapsto \mathbb{F}_2^8, b = f(a)$:

$$f(a) = \begin{bmatrix} b_7 \\ b_6 \\ b_5 \\ b_4 \\ b_3 \\ b_2 \\ b_1 \\ b_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} a_7 \\ a_6 \\ a_5 \\ a_4 \\ a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \oplus \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \end{bmatrix}$$

The mappings from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^8}$ can be represented by a polynomial over $\mathbb{F}_{2^8}$, and this polynomial can be derived by means of Lagrange interpolation. The polynomial representation of the affine transformation $b = f(a)$ is the following $\mathbb{F}_2$-linearized affine polynomial:

$$f(a) = \texttt{63}+\texttt{05}a^{2^0}+\texttt{09}a^{2^1}+\texttt{f9}a^{2^2}+\texttt{25}a^{2^3}+\texttt{f4}a^{2^4}+\texttt{01}a^{2^5}+\texttt{b5}a^{2^6}+\texttt{8f}a^{2^7}.$$

$\mathbb{F}_2$-linearized affine polynomials are immensely efficient to compute with respect to the normal basis. There exists an element $\alpha \in \mathbb{F}_{2^n}$ such that the set $\{\alpha, \alpha^2, ..., \alpha^{2^{n-1}}\}$ constitutes a basis of $\mathbb{F}_{2^n}$ over $\mathbb{F}_2$ which is said to be a normal basis. Then, any element $x \in \mathbb{F}_{2^n}$ can be represented as a vector $(a_0, a_1, ..., a_{n-1})$ as follows:

$$x = \sum_{i=0}^{n-1} a_i \alpha^{2^i}$$

Then, a squaring is simply a right cyclic shift in this representation;

- ShiftRows: row_i $(0 \leq i < 4)$ of the state is cyclically shifted to the left by i positions:

$$\begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix} \mapsto \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{10} \\ \alpha_{22} & \alpha_{23} & \alpha_{20} & \alpha_{21} \\ \alpha_{33} & \alpha_{30} & \alpha_{31} & \alpha_{32} \end{bmatrix};$$

- MixColumns: a linear bijection is applied to the state. To apply this operation, the columns of the state are considered as polynomials over $\mathbb{F}_{2^8}$ which are then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x)$:

$$c(x) = \texttt{03}x^3 + \texttt{01}x^2 + \texttt{01}x + \texttt{02};$$

- AddRoundKey: the round key is added to the state using a bit-wise XOR. The round keys are derived from the master key by means of a key schedule algorithm. As the AES key schedule algorithm is out of the scope of this thesis, the interested reader is referred to [9] for a detailed description.

The MixColumns operation is removed in the last round of AES. Additionally, prior to the first round, an AddRoundKey operation is applied. Figure 2.1 depicts a schematic description of the encryption and decryption operations where Nr is the number of rounds.

AES is a natural choice for a benchmark of advanced cryptographic protocol implementations [10, 22, 29] due to several reasons. Firstly, AES is used in a wide range of applications. Secondly, the AES circuit is not complicated, and on the other hand, it is not trivial. Thirdly, AES is immensely efficient and suitable for

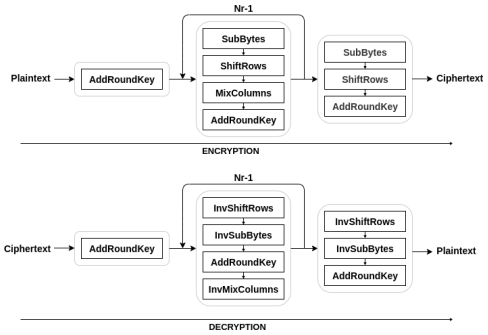

Figure 2.1: AES encryption and decryption

optimizations. For instance, it can improve its efficiency using low level processor support (*i.e.,* AES-NI). Moreover, its quite algebraic structure being unique among traditional block ciphers allows it to be easily implemented in advanced cryptographic protocols.

2.3 Fully Homomorphic Encryption (FHE)

Homomorphic encryption is an advanced cryptographic primitive that allows users to evaluate a permitted set of circuits on encrypted data without first decrypting it. In addition to three algorithms $KeyGen_{\mathcal{E}}$, $Encrypt_{\mathcal{E}}$ and $Decrypt_{\mathcal{E}}$ that any public key encryption scheme $\mathcal{E}$ employs, homomorphic encryption also employs $Evaluate_{\mathcal{E}}$. $KeyGen_{\mathcal{E}}$ is a randomized algorithm that takes a security parameter λ; outputs a secret key sk and a public key pk. pk defines a ciphertext space $\mathcal{C}$ and a plaintext space $\mathcal{P}$. $Encrypt_{\mathcal{E}}$ is a randomized algorithm that takes a plaintext $\pi \in \mathcal{P}$ and sk; outputs a ciphertext $\psi \in \mathcal{C}$. $Decrypt_{\mathcal{E}}$ takes a ciphertext ψ and a secret key sk; outputs the corresponding plaintext π. $Evaluate_{\mathcal{E}}$ is an efficient algorithm that takes a public key pk, a circuit C from the set of permitted circuits $\mathcal{C}_{\mathcal{E}}$, and a tuple of ciphertexts $\Psi = \langle \psi_1, ..., \psi_t \rangle$ of size equal to the circuit input where ψ_i encrypts π_i under pk; outputs a ciphertext encrypting $C(\pi_1, ..., \pi_t)$ [12]. Following definitions state the correctness and the compactness of a homomorphic encryption scheme:

Definition 2.3.1. A homomorphic encryption scheme $\mathcal{E}$ is said to be correct for the circuits in $\mathcal{C}_{\mathcal{E}}$ if for any (sk, pk) pair generated by $KeyGen_{\mathcal{E}}$, any circuit $C \in \mathcal{C}_{\mathcal{E}}$, any plaintexts $\pi_1, ..., \pi_t$, and any ciphertexts $\Psi = \langle \psi_1, ..., \psi_t \rangle$ with $\psi_i \leftarrow Encrypt_{\mathcal{E}}(pk, \pi_i)$, it holds that:

$$\text{if } \psi \leftarrow Evaluate_{\mathcal{E}}(pk, C, \Psi), \text{then } Decrypt_{\mathcal{E}}(sk, \psi) = C(\pi_1, ..., \pi_t).$$

Definition 2.3.2. A homomorphic encryption scheme $\mathcal{E}$ is compact if there exists a polynomial f such that, for every value of the security parameter λ, $Decrypt_{\mathcal{E}}$ can be expressed as a circuit of size at most $f(\lambda)$.

Definition 2.3 defines the minimal requirement of the functionality of $Encrypt_{\mathcal{E}}$. Consider a scheme that $Evaluate_{\mathcal{E}}(pk, C, \Psi)$ outputs C and Ψ without processing the circuit, decrypts Ψ and applies C to the results. Although this scheme is correct, we are interested in the schemes evaluating circuits on the ciphertexts. Definition 2.3 ensures this by setting an upper-bound for the length of the ciphertexts output by $Evaluate_{\mathcal{E}}$. Then, these definitions lead to the definition of FHE:

Definition 2.3.3. A homomorphic encryption scheme $\mathcal{E}$ is said to be an FHE scheme if it is correct, compact and it can evaluate any circuit.

In other words, a homomorphic encryption scheme that allows users to evaluate any circuit on encrypted data without first decrypting it is said to be FHE. The idea behind FHE was introduced by Rivest, Adleman and Dertouzos [30] as *privacy homomorphism* shortly after the invention of RSA [31], in itself the first homomorphic encryption scheme. Since then, several pre-FHE schemes were proposed that are either *partially* or *somewhat* homomorphic encryption schemes.

Partially homomorphic encryption schemes employ only a single type of operation (*i.e.,* addition or multiplication) and possess no limitation on the number of

operations performed. RSA, Paillier [28], and Goldwasser–Micali [17] are a few instances of partially homomorphic encryption schemes. Somewhat homomorphic encryption (SWHE) schemes preserve both operations (*i.e.,* addition and multiplication), but restricts their number. That is, a correct decryption is no longer possible when the number of operations exceeds a certain limit. Boneh-Goh-Nissim [5] and Sander-Young-Yung [32] are two instances of somewhat homomorphic encryption schemes.

2.3.1 From pre-FHE to FHE

Pre-FHE schemes served as intermediate steps that eventually led to an FHE scheme. In 2009, Gentry introduced the first practical FHE scheme [12]. Gentry's pioneering FHE construction gave a lead to many FHE schemes (*e.g.,* [37, 33, 14]) using the same design structure.

The first step in Gentry's FHE construction is to construct a SWHE scheme limited to computing low degree polynomials homomorphically. The encryption process of this scheme adds a small amount of noise to the output ciphertext. Then, each operation on ciphertexts increases this noise. Moreover, the correctness of this scheme is retained only if the noise level remains below a certain threshold.

The second step of Gentry's construction is to transform the SWHE scheme to an FHE scheme using *bootstrapping*. Bootstrapping is a method to reduce the noise of a ciphertext by applying a recryption function to obtain a ciphertext encrypting the same plaintext with less noise. Recryption applies the decryption function to the ciphertext homomorphically using the encrypted secret key. If a SWHE scheme is able to evaluate circuits that are deeper than its decryption circuit, this scheme is said to be *bootstrappable*. Let $\mathcal{D}_\mathcal{E}$ be the decryption circuit of the FHE scheme $\mathcal{E}$, (sk_1, pk_1) and (sk_2, pk_2) be two $\mathcal{E}$ key pairs, and Ψ be the encryption of π under pk_1. Then, the recryption algorithm is listed as following:

Algorithm 1 *Recryption*

Input : pk_2, $\mathcal{D}_\mathcal{E}$, $Encrypt_\mathcal{E}(pk_2, sk_1)$, Ψ

1: $\overline{\Psi} \leftarrow Encrypt_\mathcal{E}(pk_2, \Psi)$

2: **return** $Evaluate_\mathcal{E}(pk_2, \mathcal{D}_\mathcal{E}, \langle Encrypt_\mathcal{E}(pk_2, sk_1), \overline{\Psi}\rangle)$

The output of the recryption algorithm is thus a fresh ciphertext that is the encryption of π under pk_2.

A *squashing* step is needed when the depth of the decryption algorithm is more than the scheme can handle. This step decreases the complexity of the decryption algorithm. The interested reader is referred to [12] for a detailed description of the squashing step as it is out of the scope of this thesis. Unfortunately, both bootstrapping and squashing cause the FHE schemes that use Gentry's construction to have a poor performance in terms of efficiency.

The first major deviations from Gentry's original FHE construction were proposed by Gentry and Halevi [13], and Brakerski and Vaikuntanathan [7]. They

independently proposed methods to construct an FHE scheme without using the squashing step. However, these schemes did not eliminate the performance penalty resulting from bootstrapping.

2.3.2 Brakerski-Gentry-Vaikuntanathan (BGV) Scheme

BGV is a leveled FHE scheme proposed by Brakerski, Gentry and Vaikuntanathan [6]. It deviates from Gentry's construction in how it handles the noise. Leveled FHE is more restricted than FHE in that the depth of circuits it can evaluate is bounded by the parameters of the scheme. BGV uses *modulus-switching* introduced by Brakerski and Vaikuntanathan [7] to keep the noise under the threshold. Modulus switching is originally applied one time to obtain a ciphertext with less noise, whereas it is iteratively applied in BGV to control the noise growth.

In this work we use a BGV variant proposed by Gentry, Halevi and Smart [15]. In this scheme both ciphertexts and secret keys are represented as vectors over the polynomial ring $\mathbb{A}$, and the plaintext space is all polynomials over $\mathbb{A}_p$ for $p \geq 2$. Additionally, at any point during the homomorphic evaluation, there are *current integer modulus q* and *current secret key s* that evolve as the homomorphic operations are applied. Decryption is done by taking the inner product of the ciphertext c and the current secret key s over $\mathbb{A}_q$. Then the result is reduced modulo p:

$$a \leftarrow [\ \underbrace{[\ \langle c, s\rangle \mod \Phi_m(X)]_q\ ]}_{\text{noise}}{}_p. \tag{2.4}$$

In this context, keeping the noise in check means ensuring that it does not exceed q. Five different operations are defined in this scheme. `Addition`, `multiplication` and `automorphism` are used to evaluate circuits and therefore, alter the plaintexts encrypted under these ciphertexts. `Key-switching` and `modulus-switching` are used to control the complexity of the evaluation and therefore, do not affect the underlying plaintext.

Addition. Homomorphic `addition` is simply performed by means of a vector addition over $\mathbb{A}_q$ (with respect to the same secret key and modulus q). This operation slightly increases the noise of the ciphertext, and does not change the current secret key and the current modulus.

Multiplication. Homomorphic `multiplication` is performed by means of a tensor product over $\mathbb{A}_q$. If the two arguments of this operation have dimension n over $\mathbb{A}_q$, the output then has dimension n^2. The change in the dimension of the ciphertext consequently results in a change in the dimension of the secret key. This is because the output ciphertext would then be valid with respect to the secret key s' of dimension n^2. Therefore, the operation changes the current secret key, but not the current modulus. Homomorphic `multiplication` remarkably increases the noise of the ciphertext.

Automorphism. `Automorphism` maps a polynomial $a(X) \in \mathbb{A}$ to $a^{(i)}(X) = a(X^i) \bmod \Phi_m(X)$. The set of transformations $\{a \mapsto a^i : i \in (\mathbb{Z}/m\mathbb{Z})^*\}$ forms a group under the composition operation, and this group is isomorphic to $(\mathbb{Z}/m\mathbb{Z})^*$. Let c be a valid ciphertext encrypting a with respect to s and q. Then the output of the `automorphism` operation $c^{(i)}$ is a valid ciphertext encrypting $a^{(i)}$ with respect to $s^{(i)}$ and q. Different than the `addition` and the `multiplication`, this operation does not increase the noise of the ciphertext.[1]

Key-switching. `Key-switching` is used after the operations increasing the dimension of the secret key. It converts a valid ciphertext with respect to s' to a valid ciphertext capturing the same plaintext with respect to s such that the dimension of s is lower than s'. It does not change the current modulus, but increases the noise of the ciphertext.

Modulus-switching. `Modulus-switching` is applied to reduce the noise of the ciphertext. The output ciphertext captures the same plaintext with respect to the same key, but different modulus q'. If the q' is sufficiently smaller than q, then the output ciphertext has less noise. A property of BGV-type schemes is that they have a chain of moduli $q_0 < q_1 < ... < q_{L-1}$, and fresh ciphertexts are encrypted with respect to q_{L-1}. `Modulus-switching` is applied when the noise gets uncontrollably large to decrease the modulus q from q_i to q_{i-1}. When the output ciphertexts reach modulus q_0, it is no longer possible to operate on them, and bootstrapping is required.

Packed Ciphertexts. This FHE scheme allows performing operations on packed ciphertexts. Smart and Vercauteren [33] proposed using the Chinese Remainder Theorem to represent the plaintext space $\mathbb{A}_p$ as a vector of plaintext slots. This applies when $\Phi_m(X)$ factors modulo p into l irreducible polynomials such that $\Phi_m(X) = \prod_{j=1}^{l} F_j(X) \bmod p$. Then, a plaintext polynomial $a(X) \in \mathbb{A}_p$ can be represented as encoding l different plaintext polynomials with $a_j = a \bmod F_j$. `Addition` and `multiplication` operations are then performed slot-wise. However, this is not the case for `automorphism`. If i is a power of two, then the transformation $a \mapsto a^{(i)}$ can be realized for each slot separately, and this transformation is called a `Frobenius automorphism`[2]. Conversely, if i is not a power of two, then the transformation acts as a shift operation between the different slot elements.

The security level of this scheme is directly proportional to $\phi(m)$, and inversely proportional to the depth of the circuit that is evaluated.

[1] Each automorphism requires a key-switching which increases the noise. However, this is not a significant increase and we ignore it in this work.

[2] The properties of the Frobenius automorphism are exploited extensively throughout this thesis.

Chapter 3

Algebraic Ciphers

In this chapter, I focus on the *Marvellous* design strategy and its two families: VISION and RESCUE. In Section 3.1 I discuss the general design considerations of the algebraic ciphers. Then, in Section 3.2 I introduce the *Marvellous* design strategy [2]. Finally, in Section 3.3 and Section 3.4, the *Marvellous* families VISION and RESCUE are described, respectively.

3.1 Algebraic Cipher Design

Many methods have been proposed for block cipher design to reduce the effort in designing efficient block ciphers that are resistant to attacks. Some of these methods employ a specific set of operations to design block ciphers (*e.g.,* ARX ciphers), whereas some employ general-purpose structures (*e.g.,* Feistel, SPN). Additionaly, many of the proposed methods aim to determine the number of rounds providing desired security properties. Nevertheless, algebraic ciphers deviate from their traditional counterparts in terms of their relevant attacks and security analysis in addition to efficiency metrics. Therefore, algebraic cipher design calls for different considerations.

Non-Procedural Computation. Procedures simply consist of a series of computational steps. In procedural computation, the system's state at any point in time is a function of the system's state at the previous point in time. However, advanced cryptographic protocols tend to violate this procedural model of computation. Even though the participants in the protocols are procedural computers, some phenomena are better interpreted with respect to an alternative timeline. These phenomena are called non-procedural computations [2]. For instance, masked operations in MPC offer non-procedural properties by referring certain computations to an offline phase. Likewise, Frobenius automorphism in the FHE scheme used in this work offer non-procedural properties by computing an exponentiation of the form X^{p^i} over $\mathbb{F}_{p^n}$. The running time of this operation is independent of the exponent.

Non-procedural computations allow constant time execution in operations that would otherwise had resulted in variable running time. Therefore, exploiting non-procedural computation improves the efficiency in advanced cryptographic protocols that support it.

Efficiency Metrics. Recall that running time, number of gates, and energy/memory consumption are not the main efficiency metrics that are considered in algebraic ciphers. Instead, efficiency metrics are defined specific to the advanced cryptographic protocol. For ZK proofs, AIR or R1CS constraints; for MPC, number of multiplications and number communication rounds are driving factors. Interestingly, the cost of the operations do not depend on the size of the field that the operation is defined over in ZK proofs and MPC. However, we will see that this does not hold for FHE. The number of required operations to perform certain computations depend on the field size in FHE.

Cryptanalytic Focus. The design of the algebraic ciphers thwarts statistical attacks in a small number of rounds by means of flexible field sizes [2]. Therefore, traditional security arguments are not the main considerations in algebraic ciphers. Algebraic ciphers operate on finite fields. Therefore, every function that the cipher employs can be represented by a polynomial. Optimizing these polynomials for efficiency enables attacks that manipulate simple polynomials. These attacks include the interpolation attack [23], higher order differentials [24], the GCD attack [1], and Gröbner basis attacks [8]. The interpolation and the GCD attacks exploit the univariate polynomial relation between the plaintext and the ciphertext. In order to prevent these attacks, the univariate polynomial that describes the cipher should be dense and have a high degree. Conversely, Gröbner basis attacks exploit the multivariate polynomial description of the cipher. Gröbner basis attacks are conceptually similar to Gaussian elimination. Their objective is to solve a system of polynomials instead of a system of linear equations. Hence, low non-linear complexity of a cipher poses a weakness to Gröbner basis attacks. This brings the following contradiction: the design goal of algebraic ciphers to have a low non-linear complexity conflicts with their security against Gröbner basis attacks.

3.2 The Marvellous Design Strategy

The *Marvellous* design strategy [2] introduces a set of decisions to be taken when designing a secure and efficient algebraic cipher. A *Marvellous* design is a substitution-permutation (SP) network that repeatedly applies rounds of substitution boxes (S-box) and permutation boxes (P-box). The design principles of the *Marvellous* design strategy prioritize security, simplicity, robustness, and efficiency.

The state of a *Marvellous* design is an element in the vector space $\mathbb{F}_q^\ell$, with q either a power of 2 or a prime number and $\ell > 1$. A *Marvellous* design repeatedly

applies its round function to its state for N iterations. Figure 3.1 depicts a schematic description of the encryption operation of a *Marvellous* design. A plaintext and a master key are the inputs to the first round. Each round consists of two steps and each step employs three layers: S-box, linear, and subkey injection. The subkeys used in subkey injection are derived from the master key by means of a key schedule algorithm.

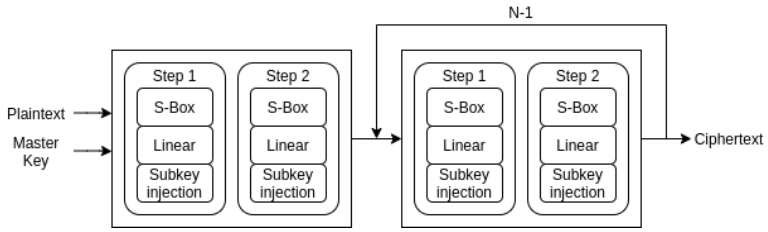

Figure 3.1: The *Marvellous* encryption operation

The S-box layer of a *Marvellous* round applies an S-box to each of the ℓ state elements. Each S-box consists of a power map $g : x^{\alpha}$ possibly followed by an affine transformation. The two steps of the *Marvellous* round employ different S-boxes in terms of their degrees. The S-box employed in the first step is denoted with θ_0, and the S-box employed in the second step is denoted with θ_1. θ_0 is chosen such that it has a high degree when the encryption is performed and a low degree when the decryption is performed. θ_1 is chosen such that it serves the opposite goal: it has a low degree when the encryption is performed and a high degree when the decryption is performed. This construction provides a high degree in both encryption and decryption, and consequently results in the same cost for both. The motivation behind employing power map S-boxes is their cryptanalytic properties [27].

For *Marvellous* ciphers operating over binary fields, the power map is followed by an invertible affine transformation. Owing to its efficiency, an $\mathbb{F}_2$-linearized affine polynomial is recommended to be used which is of the form:

$$B(x) = b_{-1} + \sum_{i=1}^{n-1} b_i x^{2^i} \in \mathbb{F}_{2^n}[x].$$

The coefficients of this polynomial are randomly generated such that they constitute a permutation polynomial. An $\mathbb{F}_2$-linearized affine polynomial is a permutation if and only if its linear part (*i.e.,* $B(x) - b_{-1}$) only has the root 0 in $\mathbb{F}_{2^n}$. While the affine transformation has no effect on the non-linearity of the S-box, it increases the algebraic degree, which provides security against algebraic attacks.

The linear layer diffuses local properties to the entire state. This is realized by multiplying the *Marvellous* state vector by a maximum distance separable (MDS) matrix. An MDS matrix is defined as follows:

Definition 3.2.1. Let $x \mapsto Mx$ be a linear transformation from $\mathbb{F}_q^{\ell}$ to $\mathbb{F}_q^{s}$ defined

by the $s \times \ell$ matrix M. M is then said to be an MDS matrix if the set of all pairs (x, Mx) is a linear code of dimension ℓ, length $\ell + s$, and minimum distance $s + 1$ [19].

The linear layer of the *Marvellous* design does not change the size of the state vector. Therefore, the MDS matrix used in the *Marvellous* round is an $\ell \times \ell$ square matrix.

Using a key scheduling algorithm and different subkeys for each subkey injection increases the complexity of the system of polynomials describing the cipher. Therefore, this strategy increases the security of the cipher against Gröbner basis attacks. The key schedule algorithm of the *Marvellous* design is indeed the iteratively applied *Marvellous* round function. In order to generate the subkeys, the round function takes the master key instead of the plaintext input, and takes additional round constants instead of the subkeys injected. The intermediate state after the round constant injection is provided as a subkey. Round constants are used to prevent possible symmetries and similarities in the algorithm to thwart certain type of attacks (*e.g.,* rotational cryptanalysis). The round constants should be chosen such that they are not rotational-invariant and they do not belong to any subfield of $\mathbb{F}_q$.

The number of rounds in a *Marvellous* design is set to be

$$2 \cdot \max(r_0, r_1, 5),$$

where r_0 is set to be the maximum number of *Marvellous* rounds that can be attacked by differential and linear cryptanalysis, higher order differentials and interpolation attacks; r_1 is said to be the instance-specific number of rounds that can be attacked by a Gröbner basis attack, and five is the sanity factor that protects the cipher against redundant optimization attempts weakening it.

Security of a *Marvellous* design. The *Marvellous* design strategy includes countermeasures against certain type of attacks. Against differential and linear cryptanalysis, the wide trail strategy [9] is followed. The wide trail strategy is an approach for designing secure and efficient block ciphers that are resistant to differential and linear cryptanalysis.

Differential cryptanalysis exploits the propagation of differences in plaintext pairs to the corresponding ciphertext pairs [4]. Therefore, achieving low difference propagation probabilities is crucial for security against differential cryptanalysis. Linear cryptanalysis exploits a linear approximation between plaintext bits and ciphertext bits. Therefore, achieving low correlation amplitudes is crucial for security against linear cryptanalysis.

Self-similarity attacks exploit the possible self-similarity of an algorithm. The working principle of these attacks is to divide an algorithm into sub-algorithms such that they are similar to one another with respect to a similarity definition. Round constants that break the self-similarity are used to thwart such attacks.

Invariant subfield attacks exploit the existence of two subfields $\mathbb{F}_{q_1} \subset \mathbb{F}_q$ and $\mathbb{F}_{q_2} \subset \mathbb{F}_q$ such that for any $x \in \mathbb{F}_{q_1}$ given to the round function as input, the output y satisfies $y \in \mathbb{F}_{q_2}$. These attacks are only relevant for binary fields as prime fields do not have non-trivial subfields. To thwart such attacks, the coefficients of the $\mathbb{F}_2$-linearized affine polynomial B and round constants are chosen such that they do not lie in any subfield of $\mathbb{F}_{2^n}$.

Higher order differential cryptanalysis exploits the low algebraic degree of an algorithm. The resistance of *Marvellous* designs operating on binary fields is provided by the S-box. An $\mathbb{F}_2$-linearized affine polynomial is used in constructing the S-box to increase the algebraic degree.

Interpolation attacks also exploit the low degree of an algorithm by reconstructing the polynomial description of the algorithm from input/output pairs. In a *Marvellous* design, at least one of the S-boxes employ a high degree power map to thwart interpolation attacks.

Gröbner basis attacks attract further attention due to the nature of algebraic ciphers that minimizes the multiplicative complexity as a design goal. Rather than a generic security argument, a novel framework to determine the resistance of a given algorithm against Gröbner basis attacks is proposed in [2]. Interested reader is referred to Appendix A in [2] for a comprehensive discussion.

Table 3.1 summarizes the maximal number of rounds that can be covered by the attacks described above with respect to a security parameter s.

Table 3.1: Resistance of a *Marvellous* design to cryptanalytic attacks

Type of Attack	Binary Fields	Prime Fields
Differential Cryptanalysis	$\frac{2s}{\log_2(q^{\ell+1})-2\cdot(\ell+1)}$	$\frac{2s}{\log_2(q^{\ell+1})-\log_2((\alpha-1)^{\ell+1})}$
Linear Cryptanalysis	$\frac{s}{\log_4(q^{\ell+1})-2\cdot(\ell+1)}$	-
Higher Order Differentials	$\frac{\log_2(s)}{\log_2(n-1)}$	-
Interpolation	3	3

3.3 Vision

VISION is a *Marvellous* family operating on binary fields with its native field $\mathbb{F}_{2^n}$. Most aspects of VISION are directly derived from the *Marvellous* design strategy. Therefore, VISION-specific design decisions are limited to the S-box layer. The state is an element of $\mathbb{F}_{2^n}^{\ell}$ which has ℓ field elements. The S-boxes consist of an inversion (with 0 mapped to 0) followed by an affine transformation. They are

constructed by first choosing a 4^{th} degree $\mathbb{F}_2$-linearized affine polynomial $B(x)$. Then,

$$\theta_1 : \mathbb{F}_{2^n} \mapsto \mathbb{F}_{2^n} : x \mapsto B(x^{-1}),$$

and

$$\theta_0 : \mathbb{F}_{2^n} \mapsto \mathbb{F}_{2^n} : x \mapsto B^{-1}(x^{-1}).$$

The design decisions regarding choosing the $\mathbb{F}_2$-linearized affine polynomial and constructing the linear layer were provided in Section 3.2.

Figure 3.2 depicts a schematic description of the VISION round function. To generate the ciphertext from a given plaintext, the round function is iterated N times. A key injection with a subkey derived from the master key takes place before the first round, between every two steps, and after the last round. Pseudo-code of VISION is listed in Algorithm 2.

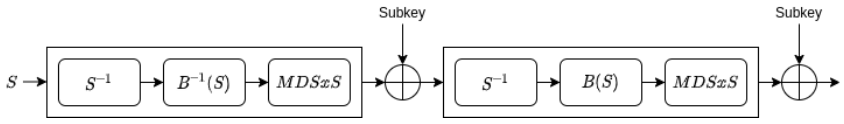

Figure 3.2: A VISION round function

Algorithm 2 *Vision*

Input : Plaintext P, subkeys K_s for $0 \leq s \leq 2N$
Output: *Vision*(K, P)

$S_0 = P + K_0$
for $j \leftarrow 1$ **to** N **do**
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = (S_{j-1}[i])^{-1}$
    $Inter_j[i] = B^{-1}(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{t=1}^{\ell} M[i,t]Inter_j[t] + K_{2j-1}[i]$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = (S_j[i])^{-1}$
    $Inter_j[i] = B(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{t=1}^{\ell} M[i,t]Inter_j[t] + K_{2j}[i]$
  end for
end for
return S_N

To obtain a security level of s, in bits, the required number of rounds is set to $N = 2 \cdot \max(r_0, r_1, 5)$. r_0 is the number of maximum rounds that can be covered by the attacks listed in Table 3.1, and $r_1 = \lceil \frac{s+\ell+8}{8\ell} \rceil$ is the number of rounds that can be attacked by Gröbner basis attack. Interested reader is referred to Section 5.1 in [2] for a detailed description of determining r_1.

3.4 Rescue

RESCUE is another *Marvellous* family this time operating on $\mathbb{F}_p$ where p is an odd prime instead of a power of 2. Same as VISION, most aspects of RESCUE are directly derived from the *Marvellous* design strategy. A RESCUE state is an element of $\mathbb{F}_p^\ell$ which has ℓ field elements. The S-boxes consist of a power map. They are constructed by first finding the smallest prime α such that $\gcd(p-1, \alpha) = 1$. Then,

$$\theta_0 : \mathbb{F}_p \mapsto \mathbb{F}_p : x \mapsto x^{1/\alpha},$$

and

$$\theta_1 : \mathbb{F}_p \mapsto \mathbb{F}_p : x \mapsto x^{\alpha}.$$

The design decisions regarding constructing the linear layer were provided in Section 3.2.

Figure 3.3 depicts a schematic description of the RESCUE round function. To generate the ciphertext from a given plaintext, the round function is iterated N times. A key injection with a subkey derived from the master key takes place before the first round, between every two steps, and after the last round. Pseudo-code of RESCUE is listed in Algorithm 3.

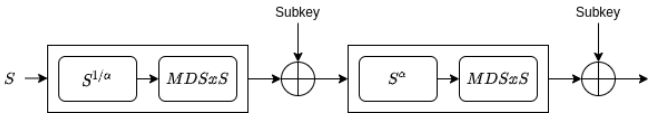

Figure 3.3: A RESCUE round function

To obtain a security level of s, in bits, the required number of rounds is set to $N = 2 \cdot \max(r_0, r_1, 5)$. r_0 is the number of maximal rounds that can be covered by the attacks listed in Table 3.1, and $r_1 = \begin{cases} \lceil \frac{s+2}{4\ell} \rceil & \text{for } \alpha = 3 \\ \lceil \frac{s+3}{5.5\ell} \rceil & \text{for } \alpha = 5 \end{cases}$ is the number of rounds that can be attacked by Gröbner basis attack. Interested reader is referred to Section 6.1 in [2] for a detailed description of determining r_1.

Algorithm 3 *Rescue*

Input : Plaintext P, subkeys K_s for $0 \leq s \leq 2N$
Output: $Rescue(K, P)$

$S_0 = P + K_0$
for $j \leftarrow 1$ **to** N **do**
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = K_{2j-1}[i] + \sum_{t=1}^{\ell} M[i,t](S_{j-1}[t])^{1/\alpha}$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = K_{2j}[i] + \sum_{t=1}^{\ell} M[i,t](Inter_j[t])^{\alpha}$
  end for
end for
return S_N

Chapter 4

Homomorphic Evaluation of the AES, Vision, and Rescue Circuits

In this chapter, I describe the working packed implementations of leveled homomorphic encryption evaluating the AES-128, VISION and RESCUE circuits. All implementations are built using HElib, a software library implementing the BGV variant described in Subsection 2.3.2. The library is written in C++ and uses the NTL mathematical library. An AES-128 circuit is implemented by Gentry *et al.* [16], and used in this work without any modification. Novel VISION and RESCUE circuits are completely implemented. The implementations of these ciphers are solely used for running time comparison featured in Chapter 5. Hence, all the VISION and RESCUE related parameters (*e.g.*, state size in bits, field size, *etc.*) are chosen to support this comparison. Note that throughout this chapter, *ciphertext* refers to an FHE ciphertext occurring during homomorphic computations.

4.1 Parameters and Input Encoding

This section deals with setting an appropriate FHE scheme to evaluate the AES, VISION, and RESCUE circuits.

4.1.1 Native Plaintext Space

The native plaintext space of the FHE scheme is chosen to support the implemented cipher's native field, *i.e.,* addition and multiplication over $\mathbb{F}_{p^n}$. The native plaintext space is defined by the ring polynomial induced by the mth cyclotomic polynomial $\Phi_m(X)$.

4.1.2 Ring Polynomial

Recalling Subsection 2.3.2 paragraph *Packed Ciphertexts*, the ring polynomial $\Phi_m(X)$ is chosen such that it factors modulo p into degree-d irreducible polynomials and

$n|d$. This structure provides the *layout* that holds elements of $\mathbb{F}_{p^d}$ in its *plaintext slots*. As $\mathbb{F}_{p^n}$ is a sub-field of $\mathbb{F}_{p^d}$, the elements of the cipher's state $\mathbb{F}_{p^n}^{\ell}$ can be embedded into this layout. Besides defining the plaintext space, m sets an upper bound for the multiplicative level L that the FHE scheme can evaluate with respect to $\phi(m)$. Finding a suitable m to evaluate a given circuit having multiplicative depth L is not deterministic, but chosen empirically.

4.1.3 Rotation Over the State Elements

Each of AES, VISION, and RESCUE employ a matrix multiplication in their linear layer. Multiplying the cipher's state by a matrix requires computing the sum of all state elements. This requires that the elements be rotated to align them for summation. Therefore, rotation is an essential operation for our implementations.

For the layout structured by $\Phi_m(X)$ to provide rotation of ℓ plaintext slots, m is chosen such that there exists an element $g \in \mathbb{Z}_m^*$ whose order is ℓ in both $\mathbb{Z}_m^*$ and $\mathbb{Z}_m^*/\langle p \rangle$. This condition allows representing ℓ plaintext slots as $t, tg, tg^2, ..., tg^{\ell-1}$ for some $t \in \mathbb{Z}_m^*$. Then, the operation $X \mapsto X^g$ shifts the state cyclically.

4.1.4 Choosing *m*

The required steps to find a suitable m are:

- Finding an m' value such that p's order in $\mathbb{Z}_{m'}^*$ is a multiple of n (*i.e.,* d is the smallest positive integer satisfying $p^d \equiv 1 \bmod m'$ such that $n|d$);
- Checking if there exists an element $g \in \mathbb{Z}_{m'}^*$ that has order ℓ in both $\mathbb{Z}_{m'}^*$ and $\mathbb{Z}_{m'}^*/\langle p \rangle$.

If m' satisfies both conditions, then $m \leftarrow m'$.

4.1.5 Packed State Representation

AES. The AES state in plaintext

$$A = (\alpha_{ij})_{i,j} = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix}$$

is encoded using column-first ordering:

$$a \approx [\, \alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}] \,.$$

Vision and Rescue. For a fair latency comparison, instances of VISION and RESCUE having 128-bit states are used to fix the throughput. To this end, we implemented VISION instances operating on $\mathbb{F}_{2^n}$ where $n \in \{8, 16, 32, 64\}$. This can be interpreted as VISION instances having n bits of data in their plaintext slots. Therefore, the respective number of elements in the cipher's state is $\ell = 128/n$.

Due to the hardness of parameter selection, only one RESCUE instance having state elements over F_{317} is used. Then, the respective number of elements is $\ell = 128/\lfloor \log_2 317 \rfloor = 16$. The VISION and RESCUE states are represented as

$$[\alpha_1, \alpha_2, ..., \alpha_\ell], \quad \text{where } \alpha_i \in \mathbb{F}_{p^n}.$$

4.2 Homomorphic Evaluation of the AES, Vision, and Rescue Operations

As mentioned in Subsection 2.3.2, circuits having a deeper multiplicative level require larger $\phi(m)$ to provide a fixed security level. Increasing $\phi(m)$ consequently causes the primitive operations (*i.e.,* additions, multiplications, automorphisms) to be more expensive. Therefore, operations consisting of numerous multiplications arranged in a deep circuit will have considerably longer running times. For binary fields, employing a Frobenius automorphism, see Section 2.3.2, would be beneficial in decreasing the depth of the circuit and allowing lower $\phi(m)$ parameter. The fact that the running time of a Frobenius automorphism $X \mapsto X^{2^i}$ for $i \in \{1, 2, ..., n-1\}$ does not depend on i, but only on $\phi(m)$ motivates a wide use. Furthermore, the Frobenius automorphism will be instrumental in optimizing our cipher in Chapter 7.

4.2.1 Power Maps

A power map is a function consisting of a single term in the form of $X \mapsto X^e$. Power maps are widely used in S-box layers owing to their cryptanalytic properties [27]. Carefully chosen power maps provide high nonlinearity, high nonlinear order, and resistance against differential cryptanalysis. There are several classes of power maps that are well studied: $x \mapsto x^{-1}$ (inversion), $x \mapsto x^{2^k+1}$ (Gold exponents) [27], $x \mapsto x^{2^{2m}-2^m+1}$ (Kasami exponents) [21], $x \mapsto x^{2^m-2^{m/2}-1}$ (Niho exponents) [11], *etc.* In particular, inversion is used in AES and VISION to construct the S-box.

For a power map to be employed in an S-box, the exponent should be chosen such that the function is a permutation polynomial (Definition 2.1.11). A power map $x \mapsto x^e$ permutes $\mathbb{F}_{p^n}$ if and only if $\gcd(e, p^n - 1) = 1$.

Packed representation of the cipher's state enables the parallelization of computing the power maps on each element. Therefore, the running time of power maps does not depend on the number of elements in the cipher's state. However, many of the above mentioned power maps are not suitable for an FHE implementation. The Frobenius automorphism is not solely enough to compute these power

maps which in turn results in deep circuits. If the exponent depends on the field size and cannot be simply represented in form X^{2^i+c}, then the required number of operations to implement it increases rapidly. In the following paragraph we will see that inversion heavily depends on the field size.

AES and Vision. The SubBytes operation of AES and the S-box layer of VISION employ inversion. This power map over $\mathbb{F}_{2^n}$ is equivalent to computing $x \mapsto x^{2^n-2}$, due to *Lagrange's theorem* (Theorem 2.1.1). The non-zero elements of $\mathbb{F}_{2^n}$ form a finite group $\mathbb{G}$ of order $2^n - 1$ under multiplication. The order of any element $x \in \mathbb{G}$ is equal to the order of the cyclic subgroup generated by itself, $\langle x \rangle$. Then, by *Lagrange's theorem*, $|\langle x \rangle| \mid (2^n - 1)$. Therefore, $x^{2^n-1} \equiv \mathbf{1}$, where $\mathbf{1}$ is the identity element of $\mathbb{G}$. Then, the following statement can be concluded for $\mathbb{F}_{2^n}$ (with 0 mapped 0):

$$x^{-1} \equiv x^{-1} \cdot 1 \equiv x^{-1} \cdot x^{2^n-1} \equiv x^{2^n-2}.$$

Algorithm 4 lists the pseudo-code of the implementation for computing the map $x \mapsto x^{2^n-2}$.

Algorithm 4 *Inversion over* $\mathbb{F}_{2^n}$

Input : x, n

1: $exp \leftarrow 2$
2: $tmp1 \leftarrow x^2$ // Frobenius automorphism
3: $tmp2 \leftarrow x^2 \cdot x = x^3$ // Multiplication (-1 level)
4: **for** $i \leftarrow 2$ **to** $\log n$ **do**
5: $x = (tmp2)^{2^{exp}}$ // Frobenius automorphism
6: $tmp1 = x \cdot tmp1$ // Multiplication (-1 level)
7: **if** $i == \log n$ **then**
8: **return** $tmp1$
9: **end if**
10: $tmp2 = x \cdot tmp2$ // Multiplication[1]
11: $exp = 2 \cdot exp$
12: **end for**

Computing inversion using Algorithm 4 requires $\log n$ Frobenius automorphisms and $2(\log n - 1)$ multiplications arranged in a depth-$\log n$ circuit.

Rescue. RESCUE's S-box employs two different power maps: $x \mapsto x^{1/\alpha}$ and $x \mapsto x^{\alpha}$. In our implementation, the prime field is chosen such that $\alpha = 3$. In the rest of this thesis, the power functions will always be written as $x \mapsto x^{1/3}$ and $x \mapsto x^3$.

[1] This multiplication is parallel to the one in line 6. Therefore, it does not increase the circuit depth.

$x \mapsto x^3$ is computed as a simple exponentiation using square and multiply. On the other hand, computing $x \mapsto x^{1/3}$ requires an exponent manipulation such that $x^{1/3} \equiv x^e \bmod p$, $e \in \mathbb{N}$. Similar to $\mathbb{F}_{2^n}$, the non-zero elements of $\mathbb{F}_p$ form a finite group $\mathbb{G}$ under multiplication and $|\mathbb{G}| = p - 1$. Similarly, the order of any element $x \in \mathbb{G}$ is equal to the order of the cyclic subgroup generated by itself, $\langle x \rangle$. Then, by *Lagrange's theorem* (Theorem 2.1.1), $|\langle x \rangle| \mid (p-1)$. Therefore, $x^{p-1} \equiv \mathbf{1}$, where $\mathbf{1}$ is the identity element of $\mathbb{G}$. Then, the following statement can be concluded for $\mathbb{F}_p$ (with 0 mapped 0):

$$x \mapsto x^{1/3} \equiv x^{1/3} \cdot x^{(2p-2)/3} \equiv x^{(2p-1)/3}.$$

Algorithm 5 lists the pseudo-code of the implementation for computing exponentiation recursively using square and multiply.

Algorithm 5 *Exponentiation(x, e)*

Input : State: x, exponent: e
Output: x^e

1: **if** $e == 1$ **then**
2: **return** x
3: **else if** $e \,\%\, 2 == 0$ **then** // exponent is even
4: $x = x^2$
5: **return** $Exponentiation(x, e/2)$
6: **else** // exponent is odd
7: **return** $x \cdot Exponentiatation(x^2, (e-1)/2)$
8: **end if**

Let $v = (v_0, v_1, ..., v_t)$ be the binary representation of the exponent where v_0 is the most significant bit. Then, $\Gamma = (t + HW(v'))$ is the required number of multiplications, where $v' = (v_1, ..., v_t)$ and $HW(\cdot)$ is a function that outputs the Hamming weight of the input. The multiplications are arranged in a depth-Γ circuit.

4.2.2 Affine Transformation

Both AES and VISION employ an affine transformation in their S-Boxes. As stated in Section 2.2, each $\mathbb{F}_2$ affine transformation can be represented as an $\mathbb{F}_{2^n}$ affine transformation over the conjugates in the form of $A(X) = a_0 + \sum_{i=0}^{n-1} a_i \cdot X^{2^i}$ over $\mathbb{F}_{2^n}$. Then, $A(X)$ is said to be an $\mathbb{F}_2$-linearized affine polynomial (Definition 2.1.10).

Recall that the goal in employing an affine transformation following an inversion is the simple algebraic expression of the inversion (see Section 3.2). As stated by Daemen *et al.* [9], properly chosen $\mathbb{F}_2$-linearized affine polynomials increase the algebraic complexity of the S-box.

Similar to power maps, for an $\mathbb{F}_2$-linearized affine polynomial to be employed in an S-box, it should be a permutation polynomial (Definition 2.1.11). An $\mathbb{F}_2$-linearized affine polynomial $A(x) = L(x) + a_{-1}$ over $\mathbb{F}_{2^n}$ where $L(x) = \sum_{i=0}^{n-1} a_i x^{2^i}$

is a permutation if and only if 0 is the only root of $L(x)$ in $\mathbb{F}_{2^n}$. In other words, $A(x)$ is a permutation if and only if $\det(a_{i-j}^{2^j}) \neq 0$ for $i, j = 0, 1, ..., n-1$.

There are two types of affine transformations. Both types can be represented as an $\mathbb{F}_2$-linearized affine polynomials, either sparse or dense. The $\mathbb{F}_2$-linearized affine polynomial $A(X) = a_0 + \sum_{i=0}^{n-1} a_i X^{2^i}$ over $\mathbb{F}_{2^n}$ is said to be sparse if most of the coefficients of $A(X)$ are 0, and dense otherwise. Both AES and VISION employ a dense $\mathbb{F}_2$-linearized affine polynomial.

Considering the implementation of $\mathbb{F}_2$-linearized affine polynomials as a circuit in FHE, dense polynomials require more operations which in turn would result in longer running times.

Specifically, the $\mathbb{F}_2$-linearized affine polynomial of the AES SubBytes operation is a polynomial of degree 2^7. Recall that the two $\mathbb{F}_2$-linearized affine polynomials in VISION are different in terms of their degrees, depending if the step is even or odd. The $\mathbb{F}_2$-linearized affine polynomial B of θ_1 is a polynomial of degree 4, and B^{-1} of θ_0 is a polynomial of degree 2^{n-1}. Algorithm 6 lists pseudocode for computing the $\mathbb{F}_2$-linearized affine polynomials in AES and VISION. This computation requires two Frobenius automorphisms for B, $n-1$ for B^{-1}, and seven for AES. $\mathbb{F}_2$-linearized polynomials can be arranged in a depth-0.5 circuit; the consumption of 0.5 levels is due to the constant multiplication.

Algorithm 6 *Affine Transformation*

Input : x

if θ_0 **then**
  $deg = n - 1$
else if θ_1 **then**
  $deg = 2$
else if *AES* **then**
  $deg = 7$
end if
$sum \leftarrow \gamma_0 \cdot x$
for $i \leftarrow 1$ **to** deg **do** // Frobenius automorphism
  $sum \mathrel{+}= \gamma_i \cdot x^{2^i}$ // constant multiplication (-0.5 levels)
end for
$sum \mathrel{+}= \delta$
return sum

4.2.3 Linear Layers

The purpose of a linear layer is to diffuse local properties to the entire state. The linear layer of AES is obtained through a composition of ShiftRows and MixColumns. ShiftRows is simply a byte-permutation and MixColumns is a multiplication of the cipher's state with an MDS matrix. Conversantly, linear layers of VISION and RESCUE consist of an MDS matrix multiplication.

AES. In the implementation of AES ShiftRows and MixColumns are lumped together as a single linear transformation over the vectorized state $v \in (\mathbb{F}_{2^8})^{16}$. To implement these operations l-SELECT [15] is used. The l-SELECT operation is implemented as follows: given a set of indices I that will be selected out of the l plaintext slots, a vector v of *select bits* $(v_0, v_1, ..., v_{l-1})$ is constructed such that $v_i = 1$ if $i \in I$ and $v_i = 0$ otherwise. The multiplication of the ciphertext input with the select vector v yields a ciphertext capturing the plaintext elements corresponding to the indices in I, and zero elsewhere.

To yield ShiftRows and MixColumns, four ciphertexts are obtained by using rotation and l-SELECT operations. Then, these four ciphertexts are linearly combined with the appropriate coefficients (1, X, and (X+1)). The constant multiplication with the select vector v can be embedded into the constant multiplications of the linear combination. As a result, due to the parallelized arrangement of the constant multiplications, ShiftRows and MixColumns consume only half a level in terms of added noise.

Recall the vectorized AES state representation:

$$A = \begin{bmatrix} \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \\ \alpha_{30} & \alpha_{31} & \alpha_{32} & \alpha_{33} \end{bmatrix}$$

$$a \approx [\, \alpha_{00}\alpha_{10}\alpha_{20}\alpha_{30}\alpha_{01}\alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}\alpha_{12}\alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}\alpha_{23}\alpha_{33}] \,.$$

Three right rotations by 11, 6, and 1 slots are applied to the state vector a to get the vectors a_{11}, a_6 and a_1 representing the state matrices A_{11}, A_6, and A_1, respectively:

$$A_{11} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \alpha_{10} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \alpha_{20} \\ \alpha_{31} & \alpha_{32} & \alpha_{33} & \alpha_{30} \\ \alpha_{02} & \alpha_{03} & \alpha_{00} & \alpha_{01} \end{bmatrix} \qquad A_6 = \begin{bmatrix} \alpha_{22} & \alpha_{23} & \alpha_{20} & \alpha_{21} \\ \alpha_{32} & \alpha_{33} & \alpha_{30} & \alpha_{31} \\ \alpha_{03} & \alpha_{00} & \alpha_{01} & \alpha_{02} \\ \alpha_{13} & \alpha_{10} & \alpha_{11} & \alpha_{12} \end{bmatrix}$$

$$a_{11} \approx [\, \alpha_{11}\alpha_{21}\alpha_{31}\alpha_{02}...\alpha_{20}\alpha_{30}\alpha_{01}] \qquad a_6 \approx [\, \alpha_{22}\alpha_{32}\alpha_{03}\alpha_{13}...\alpha_{31}\alpha_{02}\alpha_{12}]$$

$$A_1 = \begin{bmatrix} \alpha_{33} & \alpha_{30} & \alpha_{31} & \alpha_{32} \\ \alpha_{00} & \alpha_{01} & \alpha_{02} & \alpha_{03} \\ \alpha_{10} & \alpha_{11} & \alpha_{12} & \alpha_{13} \\ \alpha_{20} & \alpha_{21} & \alpha_{22} & \alpha_{23} \end{bmatrix}$$

$$a_1 \approx [\, \alpha_{33}\alpha_{00}\alpha_{10}\alpha_{20}...\alpha_{03}\alpha_{13}\alpha_{23}]$$

The top rows of A, A_{11}, A_6, and A_1 describe exactly the output of ShiftRows. Before we move to show how to compute MixColumns, note that the top row elements

also correspond to positions 0, 4, 8 and 12. These are the elements required for computing MixColumns, and selected by means of an l-SELECT operation. The MDS matrix of MixColumns consists of the coefficients 1, X and $X + 1$. Therefore, they are encoded in three plaintexts such that they contain the coefficients in positions 0, 4, 8 and 12, and zero elsewhere. Then, the four ciphertexts encrypting a, a_{11}, a_6, and a_1 are multiplied by these constants. Following equations describe the output of MixColumns:

$$m'_0 = a \cdot (X) + (a_1 + a_6) \cdot (1) + a_{11} \cdot (X + 1);$$
$$m'_1 = (a + a_1) \cdot (1) + a_6 \cdot (X + 1) + a_{11} \cdot (X);$$
$$m'_2 = (a + a_{11}) \cdot (1) + a_1 \cdot (X + 1) + a_6 \cdot (X);$$
$$m'_3 = a \cdot (X + 1) + a_1 \cdot (X) + (a_6 + a_{11}) \cdot (1).$$

Now, the top rows of the corresponding matrix representations of m_i's capture exactly the output of MixColumns. Moreover, the other rows contain only zeros, which simplifies the subsequent computation. Finally, three rotations are required to arrange the elements in the correct order and form the state after the linear layer:

$$m = m'_0 + (m'_1 \gg 1) + (m'_2 \gg 2) + (m'_3 \gg 3),$$

where $\gg$ denotes a right rotation. Algorithm 7 lists the implementation of ShiftRows and MixColumns.

Algorithm 7 *AES ShiftRows and MixColumns*

Input : x, constants 1, X, $(X + 1)$: C_1, C_X, C_{X+1}

1: $x_0 \leftarrow x$
$x_1 \leftarrow x \gg 1$
$x_6 \leftarrow x \gg 6$
$x_{11} \leftarrow x \gg 11$
2: //constant multiplication (-0.5 levels)
$x'_0 \leftarrow x_0 \cdot C_X + (x_1 + x_6) \cdot C_1 + c_{11} \cdot C_{X+1}$
$x'_1 \leftarrow (x_0 + x_1) \cdot C_1 + x_6 \cdot C_{X+1} + c_{11} \cdot C_X$
$x'_2 \leftarrow (x_0 + x_{11}) \cdot C_1 + x_1 \cdot C_{X+1} + c_6 \cdot C_X$
$x'_3 \leftarrow x_0 \cdot C_{X+1} + x_1 \cdot C_X + (c_6 + c_{11}) \cdot C_1$
3: **return** $x'_0 + (x'_1 \gg 1) + (x'_2 \gg 2) + (x'_3 \gg 3)$

Vision and Rescue. The linear layer of the VISION and RESCUE rounds consist of a multiplication of the state vector with an MDS matrix. The elements of the output state are linear combinations of input state elements. Computing a linear combination requires input state elements to be aligned, which is realized by data movement techniques, *e.g.,* rotation. The MDS matrix is not in encrypted form; thus data movement is restricted to the ciphertext capturing the cipher's state.

To implement the matrix multiplication, a similar approach to the total-sums algorithm presented in [20] is used. The total-sums algorithm takes a vector v of length l_v as input, and outputs a vector w such that $w[j] = \sum_{i=0}^{l_v-1} v[i]$ for $j \in \{0, 1, ..., l_v - 1\}$. We modify the algorithm to yield the output vector w such that $w[j] = \textit{total-sums}(MDS[j] \odot v)$ for $j \in \{0, 1, ..., l_v - 1\}$ given a vector v of length l_v. Figure 4.1 illustrates the modified algorithm for the matrix multiplication of a vector of size four with a 4×4 matrix where *Rotation-x* denotes a cyclic right shift by x slots:

$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} = \begin{bmatrix} a1 + b2 + c3 + d4 \\ e1 + f2 + g3 + h4 \\ i1 + j2 + k3 + l4 \\ m1 + n2 + o3 + p4 \end{bmatrix} \tag{4.1}$$

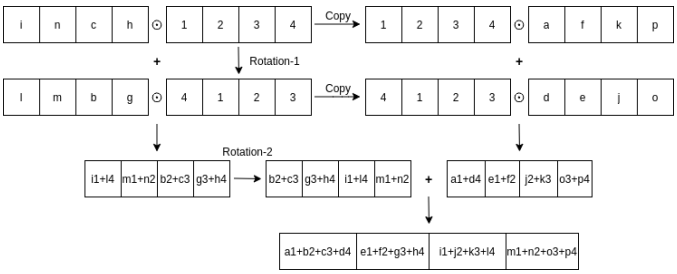

Figure 4.1: Illustration of the matrix multiplication algorithm for Equation 4.1

As depicted in Figure 4.1, the re-positioning of the matrix elements is required so that the vector elements are multiplied by the respective matrix elements. The pseudo-code for computing the matrix multiplication for VISION and RESCUE is listed in Algorithm 8. This operation requires ℓ constant multiplications that consume half a level in terms of added noise, and 1, 2, 4, 6, and 10 rotations for $\ell = 2, 4, 8, 16, 32$, respectively.

4.2.4 Subkey Injection

Subkey injection is simply an addition of the ciphertexts encrypting the state and the round key.

Algorithm 8 *Matrix Multiplication*

Input : preprocessed MDS rows: MDS, vector state: x, number of slots: ℓ

$vec = [],\ vec2 = []$ // Place holders
if $l == 2$ **then**
  // Constant multiplication (-0.5 levels)
  $x' = MDS[1] \cdot (x \gg 1)$ // Rotation
  $x = MDS[0] \cdot x$
  return $x + x'$
else if $\ell == 4$ **then** // Initialization for variable number of slots
  $iter \leftarrow 1,\ rot\_amount \leftarrow 1$
else if $\ell == 8$ **or** $\ell == 16$ **or** $\ell == 32$ **then**
  $iter \leftarrow \log \ell - 2,\ rot\_amount \leftarrow 3$
end if
$groups \leftarrow \ell/(rot\_amount + 1)$
$vec.push\_back(x)$
for $i \leftarrow 1$ **to** rot_amount **do**
  $vec.push\_back(vec[i-1] \gg 1)$ // Rotation
end for
for $i \leftarrow 0$ **to** $groups - 1$ **do**
  $sum \leftarrow 0$
  for $j \leftarrow 0$ **to** rot_amount **do**
    // Constant multiplication (-0.5 levels)
    $sum \mathrel{+}= MDS[j + i * (rot\_amount + 1)] \cdot vec[j]$
  end for
  $vec2.push\_back(sum)$
end for
$rot\_amount \mathrel{+}= 1$
for $i \leftarrow 1$ **to** $iter$ **do**
  $groups \mathrel{/}= 2$
  for $j \leftarrow 0$ **to** $groups - 1$ **do**
    // Rotation
    $vec2[j] = (vec2[j] \gg rot\_amount)$
    $vec2[j] \mathrel{+}= vec2[j + groups]$
  end for
  $rot\_amount \mathrel{*}= 2$
end for
return $vec2[0]$

4.3 Expected Costs

In this section the expected costs of the three ciphers are presented considering different metrics: number of key-switching calls, multiplications, automorphisms, and circuit depth. The number of key-switching calls is a dominating factor in terms of latency. Key-switching is called after each multiplication and automorphism. Note that the expected numbers for the circuit depth are underestimates. The actual numbers depend on m and the accumulation of noise caused by the primitive operations.

AES. The number of elements in an AES state and the degree of the field extension are constant. Table 4.1 presents the cost of one AES round in terms of the different cost metrics.

Table 4.1: The cost of an AES round

Circuit Depth	4
Number of Multiplications	4
Number of Frobenius Automorphisms	11
Number of Rotations	6

Vision. Unlike AES, the cost of implementing the VISION round is a function of n and ℓ. Table 4.2 presents the cost of one VISION round in terms of the different cost metrics for variable ℓ values.

Table 4.2: The cost of a VISION round

Cost Metric / ℓ	Circuit Depth	No. Multiplication	No. Frobenius Automorphisms	No. Rotations
2	$2\log n + 2$	$4\log n - 4$	$2\log n + n + 1$	2
4				4
8				8
16				12
32				20

Rescue. Similar to Vision, the cost of implementing the RESCUE round is a function of p and ℓ. Table 4.3 presents the cost of one RESCUE round in terms of different cost metrics for variable ℓ values.

Table 4.3: The cost of a RESCUE round

Cost Metric / ℓ	Circuit Depth	No. Multi-plication	No. Rotations
2	$\Gamma + 3$	$\Gamma + 2$	2
4			4
8			8
16			12
32			20

Chapter 5

Benchmarks

In this chapter, I evaluate the behavior of VISION and RESCUE in FHE. To this end, a comparative analysis of VISION, RESCUE and AES for 128-bit state, and 128-bit security is performed in terms of latency. In this context, latency refers to the time it takes the encryption function to finish. Preliminary results from comparative study were published in [35].

For a fixed state size, we can choose the degree of the field extension for VISION, and the field size for RESCUE. A VISION state with 16 elements over $\mathbb{F}_{2^8}$ and a RESCUE state with 16 elements over $\mathbb{F}_p$, $\log p \approx 8$ have the same state size as the AES state, *i.e.,* the same throughput. In the benchmarks we used instances of VISION operating on $\mathbb{F}_{2^8}$, $\mathbb{F}_{2^{16}}$, $\mathbb{F}_{2^{32}}$ and $\mathbb{F}_{2^{64}}$, and an instance of RESCUE operating on $\mathbb{F}_{317}$. Especially for larger fields, parameter selection is challenging in order to satisfy the constraints listed in Subsection 4.1.2 and Subsection 4.1.3.

For VISION, the increase in the degree of the field extension d requires larger m values to satisfy the constraint $n|d$. If $\phi(m)$ also gets larger, the latency would increase in addition to abundant security level that is provided by the parameter m. If $\phi(m)$ gets smaller, the provided security level will be insufficient again decreasing the number of eligible m values. Moreover, further candidates are eliminated as $\mathbb{Z}_m^*$ might not contain a suitable generator to provide desired cyclic shift properties. This results in several m's in Table 5.1 providing more than 128 bits of security; we ignore the excess in the benchmarks. Due to the larger depth of RESCUE circuit for large field sizes, there is no benchmark for $\mathbb{F}_p$ where $\log p \geq 16$ as it induces impractical $\Phi(m)$. This is because large m values require excessive amount of memory besides tremendously increasing the latency. Note that the required number of rounds for each cipher is derived from the Sage code that generates VISION and RESCUE instances [34].

Test Environment. We benchmarked the AES implementation, and our VISION and RESCUE implementations in an environment that runs Ubuntu 18.04.5 LTS with 32 GB RAM and AMD EPYC 7452 32-Core Processor @ 2.3 GHz, using the g++ compiler version 7.5.0.

5.1 Results

Table 5.1 shows that VISION and RESCUE fall behind AES in terms of latency. AES performs 88% faster than the most efficient instance of VISION ($\mathbb{F}_{2^8}^{16}$), and 96% faster than RESCUE. The difference in the running times of AES and VISION/RESCUE instances is graphically depicted in Figure 5.1. The reason VISION and RESCUE are slower than AES is that they require deeper circuits which in turn require a larger $\phi(m)$ to evaluate. Therefore, apart from requiring more primitive operations (*i.e.,* multiplications, additions, and automorphisms), the running time of each primitive operation is longer due to the larger $\Phi(m)$.

Table 5.1: Running times of AES, VISION, and RESCUE **in minutes** for 128-bit state

AES				
State	m	Security (bits)	No. rounds	Running Time (minutes)
$\mathbb{F}_{2^8}^{16}$	53261	141.924	10	1.4
VISION				
State	m	Security (bits)	No. rounds	Running Time (minutes)
$\mathbb{F}_{2^8}^{16}$	116885	148.144	10	11.62
$\mathbb{F}_{2^{16}}^{8}$	124645	127.444	10	17.83
$\mathbb{F}_{2^{32}}^{4}$	164737	187.327	10	43.36
$\mathbb{F}_{2^{64}}^{2}$	164737	128.08	13	120.78
RESCUE				
State	m	Security (bits)	No. rounds	Running Time (minutes)
$\mathbb{F}_{317}^{16}$	255219	124.582	10	37.14

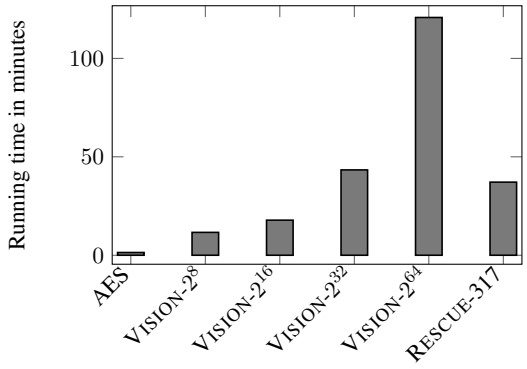

Figure 5.1: Running time comparison of AES, VISION, and RESCUE for 128-bit state

5.2 Understanding the Results

At this point, it is important to determine which operations substantially increase the latency. This investigation reveals some crucial findings that lead to a clearer overview of the behavior of VISION and RESCUE, *i.e.*, the first research question of this thesis. To this end, individual running times of the operations for a 128-bit state are isolated.

Table 5.2 summarizes these running times of each individual operation in a VISION round for $\mathbb{F}_{2^8}^{16}, \mathbb{F}_{2^{16}}^{8}, \mathbb{F}_{2^{32}}^{4}$ and $\mathbb{F}_{2^{64}}^{2}$. For each instance, the same m parameter is used in order to standardize the running times of the primitive operations. Figure 5.2 depicts the running times of the individual round operations for VISION. Similarly, Table 5.3 summarizes the running times of each individual operation in a RESCUE round for $\mathbb{F}_{317}^{16}$ and $\mathbb{F}_{65777}^{8}$. Likewise Vision, in order to standardize the running times of the FHE operations, the same m is used. Figure 5.3 depicts the running times of the individual round operations for RESCUE.

Table 5.2: Running times **in seconds** for each operation in a VISION round for variable state elements using toy parameters

VISION						
State	Inversion	B^{-1}	Matrix Multiplication	Key Addition	B	Total Round
$\mathbb{F}_{2^8}^{16}$	1.6754	1.0988	1.2851	0.0005	0.2442	7.0283
$\mathbb{F}_{2^{16}}^{8}$	2.6905	2.7774	0.8276	0.0005	0.2338	9.5705
$\mathbb{F}_{2^{32}}^{4}$	3.2286	4.7403	0.4171	0.0004	0.2338	12.0184
$\mathbb{F}_{2^{64}}^{2}$	3.6814	9.1155	0.1820	0.0005	0.2436	17.1853

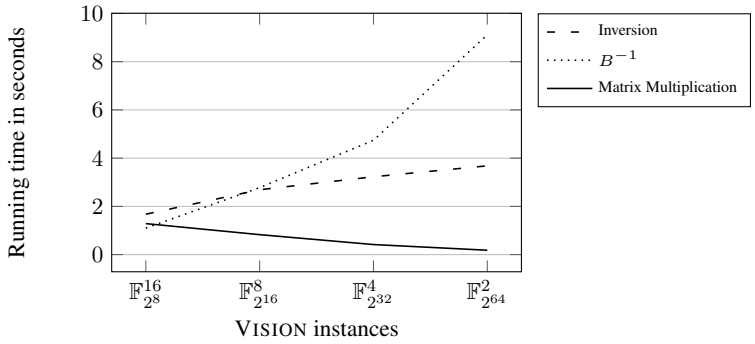

Figure 5.2: Running time comparison for each VISION round operation for different VISION instances. We see that the running time of B^{-1} grows exponentially, that of inversion linearly, and that of the matrix multiplication decreases linearly.

Table 5.3: Running times **in seconds** for each operation in a RESCUE round for variable state elements using toy parameters

RESCUE					
State	$X^{1/3}$	Matrix Multi-plication	Key Addition	X^3	Total Round
$\mathbb{F}_{317}^{16}$	8.7470	2.9274	0.0006	1.5326	16.0093
$\mathbb{F}_{65777}^{8}$	16.4416	0.9803	0.000402	0.936324	19.2206

5.2.1 Discussion

Vision. As the dimension of the state decreases (*i.e.*, the degree of the field extension increases), the running times of the inversion operation and the computation of B^{-1} increase. Recall that the number of operations required to implement inversion is proportional to $\log n$ (Algorithm 4). Therefore, the running time of the inversion operation grows logarithmically as the degree of the field extension increases. On the other hand, recall that the number of operations to implement B^{-1} is proportional to n. As a result, the running time of B^{-1} linearly increases as the degree of the field extension increases. Conversely, the decrease in the dimension of the state decreases the running time for the matrix multiplication. This is because a smaller state dimension requires fewer operations. The running times of the key addition and B do not depend on the dimension of the state but on m, and remain the same for all VISION instances.

Rescue. As the dimension of the state decreases (*i.e.*, the order of the prime field increases), the running time of computing $X^{1/3}$ increases. Recall that this is equivalent to computing $X^{(2p-1)/3}$ whose exponent is a function of p. Therefore, the

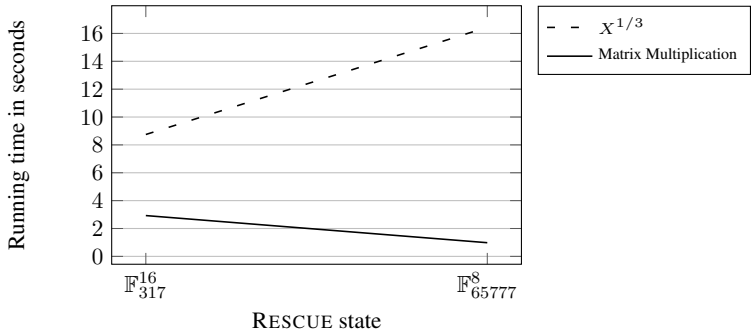

Figure 5.3: Running time comparison for each RESCUE round operation for different RESCUE instances. We see that the running time of $X^{1/3}$ grows rapidly and that of the matrix multiplication decreases.

Table 5.4: Running times **in seconds** for each operation in an AES round using toy parameters

AES					
State	Inversion	Affine Polynomial	ShiftRows/ MixColumns	Key Addition	Total Round
$\mathbb{F}_{2^8}^{16}$	1.71	0.303	0.592	0.0006	2.605

number of operations to implement the power map increases as well as the circuit depth as the order of the field increases. Similar to VISION, the decrease in the dimension of the state decreases the running time of the matrix multiplication. Even though computing X^3 and the key addition consist of same operations for all the RESCUE instances, their running times are different. This is due to the consumed levels during the computation of $X^{1/3}$. Recall that to keep the noise of the FHE ciphertexts in check modulus-switching is applied. This results in a smaller modulus. As the current modulus gets smaller, the running time of the primitive operations gets shorter.

5.2.2 Intermediate Conclusion

As a conclusion, especially for the larger degrees of the field extension in VISION, the inversion and the computation of B^{-1} are the most expensive operations in terms of latency. Similarly, especially for large fields in RESCUE, the computation of $X^{1/3}$ is the most expensive operation. If AES is compared to *Marvellous* designs, individual running times of the operations are minimal except the inversion. Still, since AES is operating on a fixed field, the running time of inversion is reasonable.

Even though VISION and RESCUE achieve a compact algebraic description in

ZK and MPC, they have a poor performance in FHE. This is because VISION and RESCUE heavily depend on non-procedural computation. For instance, inversion is efficiently computed in MPC by means of masking and offloading the heavy operations to the offline phase. However, in FHE this being unavailable; the number of operations required to compute inversion increases as the degree of the field extension increases. Therefore, different efficiency metrics are subject to FHE such as number of Frobenius autormorphisms and the circuit depth.

5.3 Benchmarking With Larger State Sizes

We reproduced our benchmarks for state sizes larger than 128 bits using VISION and RESCUE instances operating on the same fields as in Section 5.1. The motivation of this is that by increasing the number of state elements we can increase the throughput while keeping the running times of the expensive operations constant. There are two drawbacks in increasing the number of state elements. Firstly, it complicates the parameter selection. Recall that there should be an element g whose order is the number of elements in the state. Therefore, increasing the number of elements in the state might necessitate a change to m. This change will be in favor of a larger $\phi(m)$ that will in turn increase the running time of the primitive operations. Secondly, the running time of the matrix multiplication will increase.

However, we hypothesized that the fold increase in the running time for VISION would be significantly less than the fold increase in the state sizes, whereas the running time of AES has the same fold increase as the state sizes. Table 5.5 presents the results of these benchmarks for 256, 512, 1024, and 2048 bit states. Figure 5.4 illustrates the difference between running times and Figure 5.5 illustrates the fold increase in the running times of AES and VISION. The fierce drop in the running time of the VISION instance for $\mathbb{F}_{2^{64}}^2$ is due to the decrease in its number of rounds. Note that RESCUE is not included in this reproduced benchmarking. This is because increasing the number of elements in the state compels the parameter m to be increased in order to find the a suitable generator. Consequently, the m candidates for the new benchmark require excessive amounts of memory.

Hereby, as Figure 5.4 depicts, these benchmarks confirm our hypothesis that the fold increase in the running times of VISION is indeed significantly less than the fold increase in the state sizes.

To conclude, benchmarking VISION with AES for more than 128-bit throughput yields more promising results in favor of VISION. However, as Figure 5.4 depicts VISION still falls behind AES, in spite of benefiting the flexible state size property. AES performs 45% faster than VISION instance for $\mathbb{F}_{2^{16}}^{128}$ yielding 2048-bit throughput.

[1]The relevant attacks (see Section 3.2) are carried out by Mohammad Mahzoun to determine the maximum number of rounds covered by these attacks.

Table 5.5: Running times of AES and VISION **in minutes** for variable state sizes

AES					
State	Throughput (bits)	m	Security (bits)	No. rounds	Running Time (minutes)
$\mathbb{F}_{2^8}^{16}$	128	53261	141.924	10	1.4
VISION					
State	Throughput (bits)	m	Security (bits)	No. rounds [1]	Running Time (minutes)
$\mathbb{F}_{2^{16}}^{16}$	256	124645	121.681	10	20.9
$\mathbb{F}_{2^{16}}^{32}$	512	124645	121.681	10	24.17
$\mathbb{F}_{2^{16}}^{64}$	1024	124645	121.681	10	30.07
$\mathbb{F}_{2^{16}}^{128}$	2048	124645	121.681	10	41.08
$\mathbb{F}_{2^{32}}^{8}$	256	164737	187.327	10	44.92
$\mathbb{F}_{2^{32}}^{16}$	512	164737	187.327	10	48.35
$\mathbb{F}_{2^{32}}^{32}$	1024	164737	187.327	10	54.53
$\mathbb{F}_{2^{32}}^{64}$	2048	164737	183.266	10	66.43
$\mathbb{F}_{2^{64}}^{4}$	256	164737	200.128	8	54.81
$\mathbb{F}_{2^{64}}^{8}$	512	164737	200.128	8	57.30
$\mathbb{F}_{2^{64}}^{16}$	1024	164737	200.128	8	59.54
$\mathbb{F}_{2^{64}}^{32}$	2048	164737	200.128	8	65.60

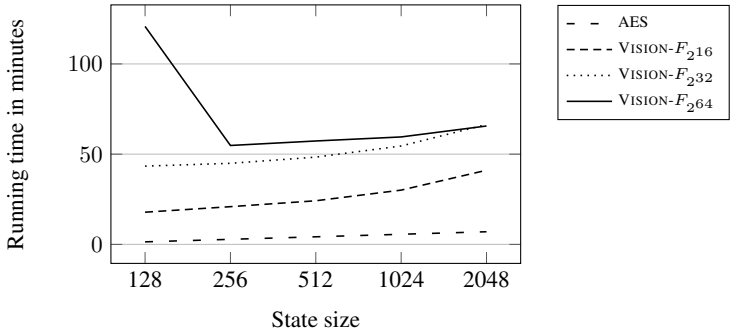

Figure 5.4: Comparison of the running times of AES and VISION for states that are larger than 128 bits

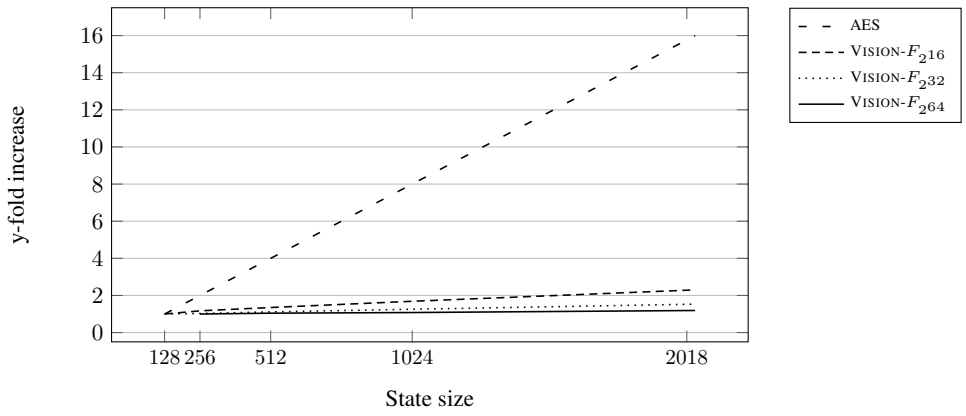

Figure 5.5: Fold increases in the running times of AES and VISION for states that are larger than 128 bits

Chapter 6

Seljuk

Our findings in Chapter 5 convey that in order to reduce the performance difference between AES and VISION, one can instantiate a novel *Marvellous* cipher optimizing for the number of operations and circuit depth. However, it is important to ensure that the improvements do not jeopardize security. In this chapter I present SELJUK. [1] This work was accepted to CFail [36].

In Section 6.1 I present the motivation underlying SELJUK. In Section 6.2 I explain and motivate the design choices of SELJUK. Then, in Section 6.3 I describe a working packed implementation of leveled homomorphic encryption that can evaluate the SELJUK circuit. Similar to previous chapters, this implementation is built on top of the HElib library. Finally, in Section 6.4 I present the performance numbers and a discussion.

6.1 Motivation of Seljuk

According to Table 5.2, the increase in the degree of the field extension most notably increases the running time of B^{-1}. Therefore, in designing SELJUK, we focused on improving the affine polynomial B^{-1} used in even steps. Improving B^{-1} will not contribute to the circuit depth. However, we hypothesized that there would be a significant improvement in the latency.

In an SP-network, the S-box must be invertible to ensure that it is possible to decrypt the ciphertexts. This condition is fulfilled by designing the S-box as a permutation, *i.e.,* a bijective function from a set to itself. Recall that the goal of a properly chosen $\mathbb{F}_2$-linearized affine polynomials is to increase the algebraic complexity. Therefore, one way to keep the algebraic complexity of the S-box high while improving the efficiency is to use a sparse $\mathbb{F}_2$-linearized affine polynomial. Doing so, one must ensure that the modified $\mathbb{F}_2$-linearized affine polynomial is

[1] The warlord that gives the name of our cipher was the eponymous founder of the Seljuk dynasty. Similary, the cipher SELJUK constitutes the *Seljuk* dynasty of algebraic ciphers. *Seljuk* was the son of Tuqaq, also known as *Iron Bow* due to his skills. *Seljuk* provides the transition from the *Marvellous* family (*Iron Man*) to *Seljuk* dynasty.

still a permutation. The requirement for a polynomial to be a permutation was described in Subsection 4.2.2.

6.2 Cipher Description

SELJUK operates on binary fields with its native field $\mathbb{F}_{2^n}$. Most aspects of SELJUK are directly derived from the *Marvellous* design strategy. For the sake of completeness, the state is an element of $\mathbb{F}_{2^n}^{\ell}$. Unlike Vision, the two steps of a SELJUK round are identical. To construct the S-box π, a sparse 8^{th} degree affine polynomial $(B(X))$ over $\mathbb{F}_{2^n}$ is chosen:

$$B(x) = x^8 + x \ \ \in \mathbb{F}_{2^n}[x].$$

Then,

$$\pi : \mathbb{F}_{2^n} \mapsto \mathbb{F}_{2^n} : x \mapsto B(x^{-1}).$$

The linear layer follows the *Marvellous* rationale Section 3.2. Figure 6.1 depicts a schematic description of the SELJUK round function. To generate the ciphertext from a given plaintext, the round function is iterated N times. A key injection with a subkey derived from the master key takes place before the first round, between every two steps, and after the last round. Pseudo-code of SELJUK is listed in Algorithm 9.

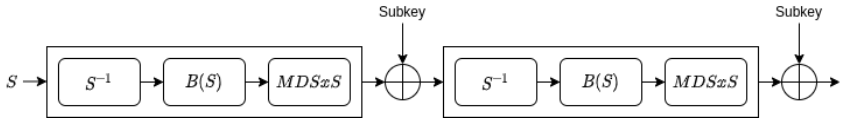

Figure 6.1: A SELJUK round function

Key Schedule. The key scheduling algorithm follows the *Marvellous* rationale. In order to derive the subkeys, the SELJUK round function is applied iteratively. The master key is fed through the plaintext interface, and the round constants are injected as the subkeys. Then, the intermediate state after the round constant injection is provided as a subkey. Round constants are generated by the means of the exact same method used in the *Marvellous* design.

Algorithm 9 *Seljuk*

Input : Plaintext P, subkeys K_s for $0 \leq s \leq 2N$
Output: *Seljuk*(K, P)

$S_0 = P + K_0$
for $j \leftarrow 1$ **to** N **do**
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = (S_{j-1}[i])^{-1}$
    $Inter_j[i] = B(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{k=1}^{\ell} M[i,k] Inter_j[k] + K_{2j-1}[i]$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = (S_j[i])^{-1}$
    $Inter_j[i] = B(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{k=1}^{\ell} M[i,k] Inter_j[k] + K_{2j}[i]$
  end for
end for
return S_N

6.3 Implementation Details

The implementation details of SELJUK follows the implementation of Vision except the computation of the $\mathbb{F}_2$-linearized affine polynomial. To compute $B(x) = x^8 + x$, only a single Frobenius automorphism is required for obtaining x^{2^3}. This computation does not apply a constant multiplication; the coefficients being one. Therefore, it does not consume levels at all.

Expected Cost of a SELJUK Round Likewise Vision, the cost of the implementation of the SELJUK round is a function in n and ℓ. Table 6.1 presents the cost of one SELJUK round in terms of different cost metrics for variable ℓ values.

Table 6.1: The cost of a SELJUK round for variable ℓ values and cost metrics

Cost Metric / ℓ	Circuit Depth	No. Multiplication	No. Frobenius Automorphisms	No. Rotations
2	$2\log n + 1$	$4\log n - 4$	$2\log n + 2$	2
4				4
8				8
16				12
32				20

6.4 Running Times

Table 6.2 summarizes that there is still a difference between SELJUK and AES. For a 128-bit state, AES performs 89% faster than the most efficient SELJUK instance (*i.e.,* SELJUK-$\mathbb{F}_{2^{16}}^{8}$). Notice that SELJUK has a poorer performance than VISION when compared to AES for 128-bit state. The reason is that due to the degree of $B(X)$, there is no SELJUK-$\mathbb{F}_{2^8}^{16}$ instance which would have been more efficient than VISION-$\mathbb{F}_{2^8}^{16}$. Figure 6.2 depicts the performance difference between AES and SELJUK visually. However, as evident from Figure 7.2, SELJUK achieves a big improvement especially for the higher degrees of the field extension. For example, SELJUK performs 58% faster than Vision for $\mathbb{F}_{2^{64}}^{2}$.

Table 6.2: Running times of AES and SELJUK **in minutes** for 128-bit state

AES				
State	m	Security (bits)	No. rounds	Running Time (minutes)
$\mathbb{F}_{2^8}^{16}$	53261	141.924	10	1.4
SELJUK				
State	m	Security (bits)	No. rounds [2]	Running Time (minutes)
$\mathbb{F}_{2^{16}}^{8}$	116885	132.476	10	13.07
$\mathbb{F}_{2^{32}}^{4}$	164737	206.157	10	27.58
$\mathbb{F}_{2^{64}}^{2}$	164737	142.462	13	50.94

[2]The relevant attacks (see Section 3.2) are carried out by Mohammad Mahzoun to determine the maximum number of rounds covered by these attacks.

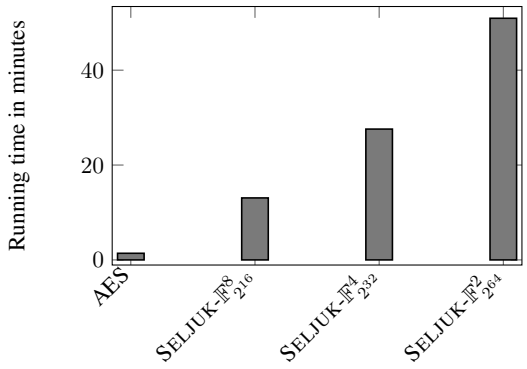

Figure 6.2: Running time comparison of AES and SELJUK for fixed state size of 128 bits

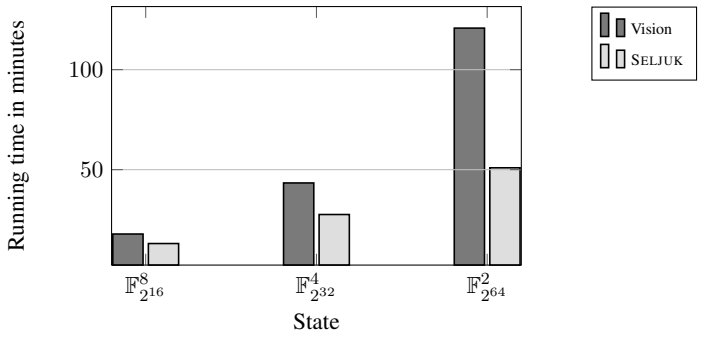

Figure 6.3: Running time comparison of Vision and SELJUK for fixed state size of 128 bits

6.5 Running Times with Larger State Sizes

Similar to VISION, we reproduced our benchmarks for state sizes larger than 128 bits using SELJUK instances operating on the same fields as in Section 6.4. Table 6.3 presents the results of these benchmarks for 256, 512, 1024, and 2048 bit states. AES performs 31.6% faster than SELJUK for $\mathbb{F}_{2^{64}}^{32}$ yielding 2048-bit throughput. Figure 6.4 illustrates the difference between running times and Figure 6.5 illustrates the fold increase in the running times of AES and SELJUK. It can be concluded that SELJUK achieves a more efficient design than Vision. However, there is still a significant difference between AES and SELJUK, in spite of benefiting the flexible state size property. It is evident that the second efficiency barrier is due to inversion (see Table 5.2).

Table 6.3: Running times of AES and SELJUK **in minutes** for variable state sizes

AES						
State	Throughput (bits)	m	Security (bits)	No. Rounds	Running Time (minutes)	Seconds Per Bit
$\mathbb{F}_{2^8}^{16}$	128	53261	141.924	10	1.4	0.66
SELJUK						
State	Throughput (bits)	m	Security (bits)	No. rounds [3]	Running Time (minutes)	Seconds Per Bit
$\mathbb{F}_{2^{16}}^{16}$	256	116885	132.476	10	14.78	3.464
$\mathbb{F}_{2^{16}}^{32}$	512	124645	142.643	10	18.98	2.224
$\mathbb{F}_{2^{16}}^{64}$	1024	124645	142.643	10	24.28	1.422
$\mathbb{F}_{2^{16}}^{128}$	2048	124645	142.643	10	34.52	1.011
$\mathbb{F}_{2^{32}}^{8}$	256	164737	206.157	10	28.05	6.574
$\mathbb{F}_{2^{32}}^{16}$	512	164737	206.157	10	31.16	3.651
$\mathbb{F}_{2^{32}}^{32}$	1024	164737	206.157	10	37.57	2.201
$\mathbb{F}_{2^{32}}^{64}$	2048	164737	206.157	10	46.19	1.353
$\mathbb{F}_{2^{64}}^{4}$	256	164737	216.086	8	23.39	5.482
$\mathbb{F}_{2^{64}}^{8}$	512	164737	216.086	8	25.3	2.964
$\mathbb{F}_{2^{64}}^{16}$	1024	164737	216.086	8	27.72	1.624
$\mathbb{F}_{2^{64}}^{32}$	2048	164737	216.086	8	32.96	0.965

[3] The relevant attacks (see Section 3.2) are carried out by Mohammad Mahzoun to determine the maximum number of rounds covered by these attacks.

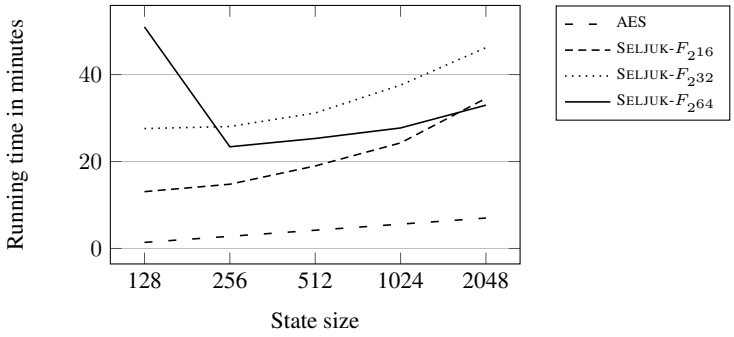

Figure 6.4: Comparison of the running times of AES and SELJUK for states that are larger than 128 bits

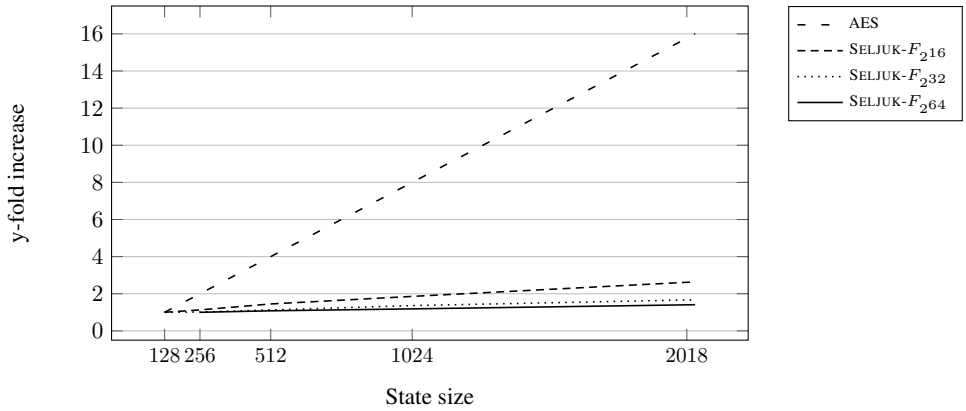

Figure 6.5: Fold increases in the running times of AES and SELJUK for states that are larger than 128 bits

Chapter 7

Chaghri

Our findings in Chapter 6 convey that in spite of the well understood cryptanalytic properties, the inversion is an efficiency barrier in FHE. In this chapter I present the successor of SELJUK: CHAGHRI. [1] CHAGHRI is a novel algebraic cipher further improving over VISION and SELJUK. This work is currently being prepared for submission to IEEE Symposium on Security and Privacy (IEEE S&P; Oakland).

In Section 7.1 I explain and motivate the design choices of CHAGHRI. Then, in Section 7.2 I describe a working packed implementation of leveled homomorphic encryption that can evalue the CHAGHRI circuit. As before, this implementation is built on top of the HElib library. Finally, in Section 7.3 I present the performance numbers and a discussion.

7.1 Cipher Description

CHAGHRI is meant to operate over $\mathbb{F}_{2^n}$. Its state consists of ℓ field elements and is an element of the vector space $\mathbb{F}_{2^n}^{\ell}$. In essence, most aspects of CHAGHRI are derived from VISION (and SELJUK) with the exception of the inversion.

A CHAGHRI round consists of two identical steps and each step employs three layers: S-box, linear and subkey injection. Like a VISION round, the S-box of a CHAGHRI round is a power map x^{α} composed with an affine transformation. To construct the S-box, the Gold exponent $F(x) = x^{2^k+1}$ is used. Owing to the seminal work of Nyberg [27], the cryptanalytic properties of the Gold exponents are well understood. They are shown to be highly nonlinear and safe against differential- and linear-cryptanalysis. Furthermore, the Gold exponents are very close to the form 2^i allowing to compute them via a Frobenius automorphism. Nevertheless, they have a low algebraic degree which pose a weakness against algebraic attacks. That being the case and similar to the cases of AES, VISION, and SELJUK, we employ a carefully chosen $\mathbb{F}_2$-linearized affine polynomial to increase

[1] Chaghri giving the name of this algebraic cipher was the grandson of Seljuk. He was the co-ruler of the early Seljuk Empire.

the algebraic degree. Let $s = \gcd(k, n)$ where n is the degree of the field extension. Then, $F(x)$ is a permutation if and only if n/s is odd. Moreover, if n is odd, $1 < k < n$ and $\gcd(n, k) = 1$, the Gold exponent is a differentially 2-uniform permutation which motivates our choice of n. Emphasizing that computing the Gold exponents does not depend on n, choosing a maximal n increases the throughput for a fixed number of elements. $n = 64$ does not satisfy that n/s is odd. Therefore, in CHAGHRI n is set to 63, and we employed a Gold exponent (denoted by $F(X)$) with $k = 62$ (*i.e.,* $F(x) = x^{2^{62}+1}$). To construct the S-box we first select an $\mathbb{F}_2$-linearized polynomial $B(x)$ of degree 8:

$$B(x) = b_{-1} + \sum_{i=0}^{3} b_i x^{2^i} \quad \in \mathbb{F}_{2^{63}}[x].$$

The requirements on how to choose an $\mathbb{F}_2$-linearized affine polynomial were provided in Chapter 6. Then, the S-box π is described as

$$\text{S-box} : \mathbb{F}_{2^{63}} \mapsto \mathbb{F}_{2^{63}} : x \mapsto B(F(x)).$$

The linear and the subkey injection layers follow the *Marvellous* design strategy as explained in Section 3.2.

To encrypt a plaintext, the CHAGHRI round function is iteratively applied N times. A key injection takes place before the first round, between every two steps and after the last round. Figure 7.1 depicts the round function of CHAGHRI. The plaintext and the master key are the inputs to the first round function, and the ciphertext is the output of the last round function. Pseudo-code of CHAGHRI is listed in Algorithm 10. The key scheduling algorithm again follows the *Marvellous* design strategy.

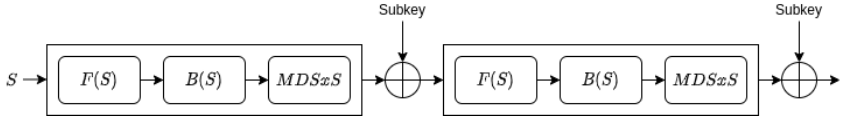

Figure 7.1: A CHAGHRI round function

Algorithm 10 *Chaghri*

Input : Plaintext P, subkeys K_s for $0 \leq s \leq 2N$
Output: *Chaghri*(K, P)

$S_0 = P + K_0$
for $j \leftarrow 1$ **to** N **do**
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = F(S_{j-1}[i])$
    $Inter_j[i] = B(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{k=1}^{\ell} M[i,k] Inter_j[k] + K_{2j-1}[i]$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $Inter_j[i] = F(S_j[i])$
    $Inter_j[i] = B(Inter_j[i])$
  end for
  for $i \leftarrow 1$ **to** ℓ **do**
    $S_j[i] = \sum_{k=1}^{\ell} M[i,k] Inter_j[k] + K_{2j}[i]$
  end for
end for
return S_N

7.2 Implementation Details

In the implementation of the CHAGHRI circuit we set $\ell = 3$. The packed representation of CHAGHRI is the same as the packed representation of VISION with $n = 63$ and $\ell = 3$:

$$[\alpha_1, \alpha_2, \alpha_3], \quad \text{where } \alpha_i \in \mathbb{F}_{2^{63}}.$$

The power map of the CHAGHRI S-Box requires one Frobenius automorphism and one multiplication. The $\mathbb{F}_2$-linearized affine polynomial is likewise simple and requires three Frobenius automorphisms and four constant multiplications. The S-Box is implemented via a depth-1.5 circuit and listed in Algorithm 11. This algorithm requires one multiplication and four Frobenius automorphisms in total.

The implementation of the linear layer resemble that of VISION for $\ell = 2$ (*i.e.,* Algorithm 8) and the pseudo-code is listed in Algorithm 12.

Subkey injection is simply an addition of the ciphertexts encrypting the state and the subkey.

Expected Cost of a CHAGHRI Round. Similar to VISION and SELJUK, the cost of the implementation of the CHAGHRI round is a function in n and ℓ. Table 7.1 presents the cost of one CHAGHRI round in terms of different cost metrics for variable ℓ values. We restricted this implementation to $\ell = 3$. However, to give the

Algorithm 11 *Chaghri S-Box*

Input : Chaghri state: x

1: $y = x^{2^{62}}$ // Frobenius automorphism
2: $x = yx$ // Multiplication (-1 level)
3: $sum \leftarrow b_0 \cdot x$
4: **for** $i \leftarrow 1$ **to** 3 **do**
5: // Frobenius automorphism + constant multiplication (-0.5 levels)
6: $sum \mathrel{+}= b_i \cdot x^{2^i}$
7: **end for**
8: $sum \mathrel{+}= b_{-1}$
9: **return** sum

Algorithm 12 *Chaghri Matrix Multiplication*

Input : Preprocessed MDS Matrix Rows: MDS, Chaghri state: x

1: $x' = MDS[1] \cdot (x \gg 1)$ // Constant multiplication -0.5 levels
2: $x'' = MDS[2] \cdot (x \gg 1)$
3: $x = MDS[0] \cdot x$
4: **return** $x + x' + x''$

complete picture of the cost, I present the cost for variable ℓ values.

Table 7.1: The cost of a CHAGHRI round for variable ℓ values and cost metrics. The circuit depth, number of multiplications and frobenius automorphisms are constant being independent of ℓ.

Cost Metric / ℓ	Circuit Depth	No. Multiplication	No. Frobenius Automorphisms	No. Rotations
2	4	2	8	2
3				4
4				4
8				8
16				12

7.3 Performance

Table 7.2 presents the running times of AES and CHAGHRI. When their running times per bit are compared, CHAGHRI achieves a more efficient design than SELJUK where CHAGHRI performs 71% faster. Moreover, with CHAGHRI we achieve a compact algebraic description performing 57% faster than AES.

Table 7.2: Running times of AES and CHAGHRI **in seconds**

AES						
State	Throughput (bits)	m	Security (bits)	No. Rounds	Running Time (minutes)	Seconds Per Bit
$\mathbb{F}_{2^8}^{16}$	128	53261	141.924	10	84.5	0.66
CHAGHRI						
State	Throughput (bits)	m	Security (bits)	No. rounds [2]	Running Time (minutes)	Seconds Per Bit
$\mathbb{F}_{2^{63}}^{3}$	189	48133	150.525	10	53.738	0.28

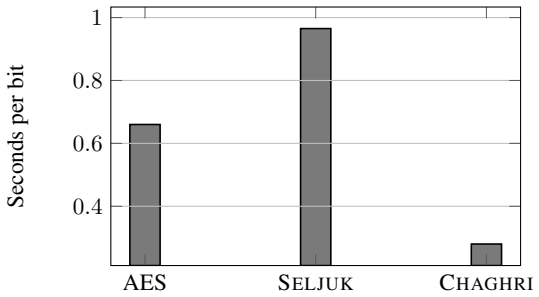

Figure 7.2: Running time comparison of AES, SELJUK, and CHAGHRI (seconds per bit)

[2]The relevant attacks (see Section 3.2) are carried out by Mohammad Mahzoun to determine the maximum number of rounds covered by these attacks.

Chapter 8

Conclusions

The object of this thesis was achieving a noteworthy contribution to the research area designing FHE-optimized algebraic ciphers. This object was motivated by the lack of attention on designing such ciphers. The existing algebraic ciphers are shown to be remarkably more efficient than their traditional counterparts when employed in ZK and MPC applications. Yet, I showed that they have a poor performance when employed by an FHE application. The reason is that FHE applications call for different efficiency metrics. In the cause of this object, I answered the following two research questions in this thesis:

- How do the algebraic ciphers behave in FHE when compared to their traditional counterparts?
- What are the specific efficiency metrics that make algebraic ciphers efficient in FHE?

Then, I presented two novel FHE-optimized algebraic ciphers designed in the light of the answers to these questions.

In *Chapter 1* I introduced the algebraic ciphers, and the current situation of the research focused on FHE-optimized algebraic ciphers to the reader. In *Chapter 2* I presented the required background on mathematical foundation of this work, AES, and FHE to fully understand the rest of the thesis. Then in *Chapter 3* I focused on algebraic ciphers and introduced the *Marvellous* design strategy, and its two families VISION and RESCUE. My contribution starts with *Chapter 4*. In *Chapter 4*, I presented the details of the FHE implementations evaluating AES, VISION, and RESCUE circuits. Then, in *Chapter 5* I evaluated the behaviour of algebraic ciphers when implemented as a circuit in an FHE protocol. To this end, I provided a state-of-the-art comparison of AES as a traditional block cipher, and VISION and RESCUE as algebraic ciphers. This comparative study stands as a pillar upon which future FHE-optimized algebraic ciphers can be built. Following this, in *Chapter 6* I discussed the design considerations of algebraic ciphers in FHE. This discussion led us to focus on algebraic ciphers operating on binary fields. Motivated by the

identified design considerations, we designed two novel algebraic ciphers: SELJUK and CHAGHRI. In *Chapter 6* I presented SELJUK and its performance numbers. I showed that with SELJUK, we achieve a more efficient *Marvellous* design than its *Marvellous* counterparts. Finally, in *Chapter 7* I presented CHAGHRI and its performance numbers. I showed that with CHAGHRI, we achieve a compact algebraic description optimized to FHE and by far it is more efficient than any other cipher, including the next best candidate, AES.

In this work, we focused on FHE-optimized algebraic ciphers operating on binary fields. A possible future work would focus on FHE-optimized algebraic ciphers operating on prime fields and exploring the possibility of achieving a compact algebraic description that is more efficient than its counterparts.

Bibliography

[1] Martin R. Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*, volume 10031 of *Lecture Notes in Computer Science*, pages 191–219, 2016. 2, 14

[2] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols. *IACR Trans. Symmetric Cryptol.*, 2020(3):1–45, 2020. 2, 13, 14, 17, 19

[3] Tomer Ashur and Siemen Dhooghe. MARVELlous: a STARK-Friendly Family of Cryptographic Primitives. *IACR Cryptol. ePrint Arch.*, 2018:1098, 2018. 2

[4] Eli Biham and Adi Shamir. Differential Cryptanalysis of DES-like Cryptosystems. *J. Cryptol.*, 4(1):3–72, 1991. 16

[5] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-DNF Formulas on Ciphertexts. In Joe Kilian, editor, *Theory of Cryptography, Second Theory of Cryptography Conference, TCC 2005, Cambridge, MA, USA, February 10-12, 2005, Proceedings*, volume 3378 of *Lecture Notes in Computer Science*, pages 325–341. Springer, 2005. 10

[6] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (Leveled) fully homomorphic encryption without bootstrapping. In Shafi Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 309–325. ACM, 2012. 11

[7] Zvika Brakerski and Vinod Vaikuntanathan. Efficient Fully Homomorphic Encryption from (Standard) LWE. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 97–106. IEEE Computer Society, 2011. 10, 11

[8] Johannes Buchmann, Andrei Pyshkin, and Ralf-Philipp Weinmann. Block Ciphers Sensitive to Gröbner Basis Attacks. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 313–331. Springer, 2006. 14

[9] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, 2002. 6, 7, 16, 25

[10] Ivan Damgård and Marcel Keller. Secure Multiparty AES. In Radu Sion, editor, *Financial Cryptography and Data Security, 14th International Conference, FC 2010, Tenerife, Canary Islands, Spain, January 25-28, 2010, Revised Selected Papers*, volume 6052 of *Lecture Notes in Computer Science*, pages 367–374. Springer, 2010. 7

[11] Hans Dobbertin. Almost perfect nonlinear power functions on gf(2n): The niho case. *Inf. Comput.*, 151(1-2):57–72, 1999. 23

[12] Craig Gentry. Fully homomorphic encryption using ideal lattices. In Michael Mitzenmacher, editor, *Proceedings of the 41st Annual ACM Symposium on Theory of Computing, STOC 2009, Bethesda, MD, USA, May 31 - June 2, 2009*, pages 169–178. ACM, 2009. 9, 10

[13] Craig Gentry and Shai Halevi. Fully homomorphic encryption without squashing using depth-3 arithmetic circuits. In Rafail Ostrovsky, editor, *IEEE 52nd Annual Symposium on Foundations of Computer Science, FOCS 2011, Palm Springs, CA, USA, October 22-25, 2011*, pages 107–109. IEEE Computer Society, 2011. 10

[14] Craig Gentry and Shai Halevi. Implementing Gentry's Fully-Homomorphic Encryption Scheme. In Kenneth G. Paterson, editor, *Advances in Cryptology - EUROCRYPT 2011 - 30th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Tallinn, Estonia, May 15-19, 2011. Proceedings*, volume 6632 of *Lecture Notes in Computer Science*, pages 129–148. Springer, 2011. 10

[15] Craig Gentry, Shai Halevi, and Nigel P. Smart. Fully Homomorphic Encryption with Polylog Overhead. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 465–482. Springer, 2012. 11, 27

[16] Craig Gentry, Shai Halevi, and Nigel P. Smart. Homomorphic Evaluation of the AES Circuit. *IACR Cryptol. ePrint Arch.*, 2012:99, 2012. 2, 3, 21

[17] Shafi Goldwasser and Silvio Micali. Probabilistic Encryption and How to Play Mental Poker Keeping Secret All Partial Information. In Harry R. Lewis, Barbara B. Simons, Walter A. Burkhard, and Lawrence H. Landweber, editors, *Proceedings of the 14th Annual ACM Symposium on Theory of Computing, May 5-7, 1982, San Francisco, California, USA*, pages 365–377. ACM, 1982. 10

[18] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The HADES design strategy. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 674–704. Springer, 2020. 2

[19] Kishan Chand Gupta and Indranil Ghosh Ray. On Constructions of Involutory MDS Matrices. In Amr M. Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *Progress in Cryptology - AFRICACRYPT 2013, 6th International Conference on Cryptology in Africa, Cairo, Egypt, June 22-24, 2013. Proceedings*, volume 7918 of *Lecture Notes in Computer Science*, pages 43–60. Springer, 2013. 16

[20] Shai Halevi and Victor Shoup. Algorithms in HElib. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 554–571. Springer, 2014. 29

[21] Doreen Hertel. A note on the kasami power function. *IACR Cryptol. ePrint Arch.*, 2005:436, 2005. 23

[22] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster Secure Two-Party Computation Using Garbled Circuits. In *20th USENIX Security Symposium, San Francisco, CA, USA, August 8-12, 2011, Proceedings*. USENIX Association, 2011. 7

[23] Thomas Jakobsen and Lars R. Knudsen. The Interpolation Attack on Block Ciphers. In Eli Biham, editor, *Fast Software Encryption, 4th International Workshop, FSE '97, Haifa, Israel, January 20-22, 1997, Proceedings*, volume 1267 of *Lecture Notes in Computer Science*, pages 28–40. Springer, 1997. 14

[24] Lars R. Knudsen. Truncated and Higher Order Differentials. In Bart Preneel, editor, *Fast Software Encryption: Second International Workshop. Leuven, Belgium, 14-16 December 1994, Proceedings*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994. 14

[25] Carsten Lund, Lance Fortnow, Howard J. Karloff, and Noam Nisan. Algebraic Methods for Interactive Proof Systems. In *31st Annual Symposium on Foundations of Computer Science, St. Louis, Missouri, USA, October 22-24, 1990, Volume I*, pages 2–10. IEEE Computer Society, 1990. 2

[26] Michael Naehrig, Kristin E. Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In Christian Cachin and Thomas Ristenpart, editors, *Proceedings of the 3rd ACM Cloud Computing Security Workshop, CCSW 2011, Chicago, IL, USA, October 21, 2011*, pages 113–124. ACM, 2011. 1

[27] Kaisa Nyberg. Differentially Uniform Mappings for Cryptography. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Workshop on the Theory and Application of of Cryptographic Techniques, Lofthus, Norway, May 23-27, 1993, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 55–64. Springer, 1993. 15, 23, 48

[28] Pascal Paillier. Public-Key Cryptosystems Based on Composite Degree Residuosity Classes. In Jacques Stern, editor, *Advances in Cryptology - EUROCRYPT '99, International Conference on the Theory and Application of Cryptographic Techniques, Prague, Czech Republic, May 2-6, 1999, Proceeding*, volume 1592 of *Lecture Notes in Computer Science*, pages 223–238. Springer, 1999. 10

[29] Benny Pinkas, Thomas Schneider, Nigel P. Smart, and Stephen C. Williams. Secure Two-Party Computation Is Practical. In Mitsuru Matsui, editor, *Advances in Cryptology - ASIACRYPT 2009, 15th International Conference on the Theory and Application of Cryptology and Information Security, Tokyo, Japan, December 6-10, 2009. Proceedings*, volume 5912 of *Lecture Notes in Computer Science*, pages 250–267. Springer, 2009. 7

[30] R L Rivest, L Adleman, and M L Dertouzos. On Data Banks and Privacy Homomorphisms. *Foundations of Secure Computation, Academia Press*, pages 169–179, 1978. 9

[31] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, 1978. 9

[32] Tomas Sander, Adam L. Young, and Moti Yung. Non-Interactive CryptoComputing For NC^1. In *40th Annual Symposium on Foundations of Computer Science, FOCS '99, 17-18 October, 1999, New York, NY, USA*, pages 554–567. IEEE Computer Society, 1999. 10

[33] Nigel P. Smart and Frederik Vercauteren. Fully Homomorphic Encryption with Relatively Small Key and Ciphertext Sizes. In Phong Q. Nguyen and

David Pointcheval, editors, *Public Key Cryptography - PKC 2010, 13th International Conference on Practice and Theory in Public Key Cryptography, Paris, France, May 26-28, 2010. Proceedings*, volume 6056 of *Lecture Notes in Computer Science*, pages 420–443. Springer, 2010. 10, 12

[34] Alan Szepieniec, Siemen Dhooghe, and Ferdinand Sauer. Marvellous (instance generator), 2019. 33

[35] Dilara Toprakhisar and Tomer Ashur. A Comparative Study of Vision and AES in FHE Setting. May 2021. WIC symposium on Information Theory and Signal Processing in the Benelux, SiTB; Conference date: 20-05-2021 Through 21-05-2021. ii, 2, 33

[36] Dilara Toprakhisar, Mohammad Mahzoun, and Tomer Ashur. A Comparative Study of Vision and AES in FHE Setting. August 2021. The Conference for Failed Approaches and Insightful Losses in Cryptology, CFail; Conference date: 14-08-2021. ii, 2, 41

[37] Marten van Dijk, Craig Gentry, Shai Halevi, and Vinod Vaikuntanathan. Fully Homomorphic Encryption over the Integers. In Henri Gilbert, editor, *Advances in Cryptology - EUROCRYPT 2010, 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Monaco / French Riviera, May 30 - June 3, 2010. Proceedings*, volume 6110 of *Lecture Notes in Computer Science*, pages 24–43. Springer, 2010. 10