Karlsruhe Institute of Technology

KU LEUVEN

Interpolation Cryptanalysis of Unbalanced Feistel Networks with Low Degree Round Functions

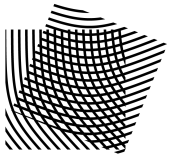
Master's Thesis
in partial fulfillment of the requirements
for the Degree of
Master of Science

Ferdinand Sauer

Competence Center for Applied Security Technology
Karlsruhe Institute of Technology

in cooperation with the

Computer Security and Industrial Cryptography group (COSIC)
KU Leuven



COSIC

Reviewer:	Prof. Dr. Jörn Müller-Quade
2nd Reviewer:	Prof. Dr. Dennis Hofheinz
Supervising Professor:	Prof. Dr. Bart Preneel
Supervisor (COSIC and DTU):	Prof. Dr. Elena Andreeva
Supervisor (University of Bristol):	Dr. Arnab Roy
Supervisor (KIT):	Dr. Willi Geiselmann

2019-07-01 – 2019-12-31

Ich versichere wahrheitsgemäß, die Arbeit selbstständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde.

Karlsruhe, den 2019-12-31

I assure truthfully that I wrote this thesis independently, that I have indicated all used aids completely and exactly and that I have marked everything that has been taken from the work of others unchanged or with changes.

Karlsruhe, the 2019-12-31

Abstract

Arithmetization Oriented Symmetric Primitives (AOSPs) address optimization potential of evaluating block ciphers and hash functions as part of Secure Multi-Party Computations, in Fully Homomorphic Encryption, or in Zero Knowledge Proof Systems. Their design differs from traditional primitives by using algebraically simple building blocks. Many proposals natively work over prime fields as opposed to bits. Because the proposals have only recently emerged, an improved understanding and analysis is required to demonstrate their security. Algebraic cryptanalysis like interpolation attacks are among the most powerful attack vectors on AOSPs. In this thesis, we generalize a prior analysis using interpolation attacks with low memory complexity to the design paradigm of the recently proposed GMiMC and its associated Sponge-based hash function GMiMCHash. More concretely, we propose a novel key recovery technique using root finding, demonstrate complexity improvements by combining multiple outputs, and apply some of the developed techniques in an algebraic Correcting-Last-Block Attack for Sponge constructions. Earlier work has posed the open question whether low memory interpolation analysis is applicable in more general settings, which we answer positively. We give concrete recommended lower bounds for the parameters of the different scenarios considered. We conclude that GMiMC and GMiMCHash are secure against the interpolation attacks developed in this thesis. Further cryptanalytic efforts considering additional attack vectors are required to firmly establish the security of AOSPs.

Zusammenfassung

Arithmetisierungs-Orientierte Symmetrische Primitive (AOSPs) sprechen das bestehende Optimierungspotential bei der Auswertung von Blockchiffren und Hashfunktionen als Bestandteil von sicherer Mehrparteienberechnung, voll-homomorpher Verschlüsselung und Zero-Knowledge-Beweisen an. Die Konstruktionsweise von AOSPs unterscheidet sich von traditionellen Primitiven durch die Verwendung von algebraisch simplen Elementen. Zusätzlich sind viele Entwürfe über Primkörpern statt über Bits definiert. Aufgrund der Neuheit der Vorschläge sind eingehendes Verständnis und ausgiebige Analyse erforderlich um ihre Sicherheit zu etablieren. Algebraische Analysetechniken wie zum Beispiel Interpolationsangriffe sind die erfolgreichsten Angriffsvektoren gegen AOSPs. In dieser Arbeit generalisieren wir eine existierende Analyse, die einen Interpolationsangriff mit geringer Speicherkomplexität verwendet, um das Entwurfsmuster der neuen Chiffre GMiMC und ihrer zugehörigen Hashfunktion GMiMCHash zu untersuchen. Wir stellen eine neue Methode zur Berechnung des Schlüssels basierend auf Nullstellen eines Polynoms vor, demonstrieren Verbesserungen für die Komplexität des Angriffs durch Kombinierung mehrere Ausgaben, und wenden manche der entwickelten Techniken in einem algebraischen Korrigierender-Letzter-Block Angriff der Schwamm-Konstruktion an. Wir beantworten die offene Frage einer früheren Arbeit, ob die verwendete Art von Interpolationsangriffen generalisierbar ist, positiv. Wir nennen konkrete empfohlene untere Schranken für Parameter in den betrachteten Szenarien. Außerdem kommen wir zu dem Schluss dass GMiMC und GMiMCHash gegen die in dieser Arbeit betrachteten Interpolationsangriffe sicher sind. Weitere kryptanalytische Anstrengungen sind erforderlich um die Sicherheitsgarantien von AOSPs zu festigen.

Acknowledgments

This thesis resulted from a cooperation between the Competence Center for Applied Security Technology (KASTEL) of the Karlsruhe Institute of Technology and the Computer Security and Industrial Cryptography group (COSIC) of KU Leuven. I would like to thank Marcel Tiepelt for bringing the possibility of doing my master's thesis in Leuven to my attention and clearing many organizational hurdles by being a forerunner. My gratitude goes to Péla Noë and Willi Geiselmann for helping with any and all formalities required for my stay in COSIC, and to Elena Andreeva and Bart Preneel for granting me the opportunity of doing my master's thesis abroad.

I am grateful for the guidance of my supervisors Elena Andreeva and Arnab Roy, who taught me many aspects of cryptography I had not been aware of before.

My gratitude goes to my wonderful colleagues Siemen Dhooghe, Liliya Kraleva, Adrián Ranea, Tim Beyne, Sayon Duttagupta, Enrique Rúa, and really everyone in COSIC, for an incredibly warm welcome, for plenty of interesting and funny discussions, for board game nights and mountain biking, and making me feel at home.

Contents

Glossary

$\mathbb{F}_p$ finite field with $p \in \mathbb{P}$

$\mathbb{F}_q$ finite field with $q = p^n$ elements, $p \in \mathbb{P}, n \in \mathbb{N}$

$\mathbb{P}$ the set of prime numbers

AES Advanced Encryption Standard

AIR Algebraic Intermediate Representation

AOC Arithmetization Oriented Cipher

AOHF Arithmetization Oriented Hash Function

AOSP Arithmetization Oriented Symmetric Primitive

CRF Contracting Round Function

DES Data Encryption Standard

ERF Expanding Round Function

FHE Fully Homomorphic Encryption

GCD Greatest Common Divisor

GMiMC Generalized MiMC

GMiMCHash Sponge-based hash function using GMiMC

MPC Multi Party Computation

SHA-3 Secure Hash Algorithm 3

SPN Substitution Permutation Network

UFN Unbalanced Feistel Network

ZK Zero Knowledge

List of Figures

List of Tables

1. Introduction

An individual's ability to selectively share information has positively contributed to its reproductive success ever since higher life forms evolved. With the emergence of language, and consequently the possibility to have messages carried by third parties that are neither sender nor receiver of the information, the problem of hiding information has taken on new forms. Consider a case where sender and receiver know each other well, or are even the same entity, sharing some knowledge a third party does not. Any information to be transmitted by that third party might somehow be encoded based on the shared knowledge, for example "I vote as I did at the great convention five years ago." This prevents the third party from learning the information. In this naïve approach, the encoding depends on the message to be delivered, making the process both cumbersome and error prone, and ultimately insecure. The fundamental study that emerged from this struggle is called *cryptography*.

Throughout most of human history, the need to selectively share information was the greatest when lives depended on it, namely during times of conflict. Many of the early developments in cryptography are thus mainly military technology, like the Caesar Cipher or the infamous Enigma used in the second World War. With the advent of modern and electronic communication, the field of cryptography was no longer confined to military usage and experienced a surge of research in academia. With today's added ubiquity of computing devices like phones or the Internet of Things, the problems of selectively hiding information have only increased. In order to address these challenges while still benefiting from the data multitude, new cryptographic protocol types like Multi Party Computation (MPC) or Fully Homomorphic Encryption (FHE) have emerged. They are greatly contributing to the motivation of this thesis and further outlined below.

The design of cryptographic primitives is based on a central conjecture: If there is no known way to circumvent a method, it is secure. In order to establish security in this way, a primitive needs to withstand rigorous cryptanalysis, demonstrating that a circumvention of the technique is impossible or infeasible. Possible attack vectors have accumulated over the last decades and centuries, and new methods of analysis are frequently proposed. Furthermore, proposals for new primitives are published to allow their analysis by anyone,

increasing the probability that potential flaws are found. In part, this cryptanalysis is usually done by the "cryptographic community," and this thesis is part of that effort.

The rest of this section is structured as follows. In Section 1.1, we motivate and summarize the questions answered in this thesis. The contributions of this work as well as an outline of the rest of the document are given in Section 1.2.

1.1 Motivation

The fields of MPC [22, 28, 40, 65, 80], FHE [23, 31, 35, 36, 71], and Zero Knowledge (ZK) proof systems [9–11, 17, 20, 21, 41] fundamentally address privacy preserving computations on data. A more ubiquitous availability of devices like phones or the Internet of things, the associated increasing threat to the privacy of users, and practicality improvements in all three fields leads to their gaining popularity. Symmetric primitives like hash functions or block ciphers are part of various applications in the three protocol types [1, 3, 10, 11, 49]. For example, they can be used to realize *oblivious pseudo random functions*, allowing privacy preserving access to databases by keyword searches [34], or *private set intersection* enabling privacy preserving data mining [46]. Another example are zero knowledge proofs of correct computation using *algebraic intermediate representations* [11, 21]. The evaluation of such symmetric primitives is orders of magnitudes slower when evaluated as part of, for example, an MPC, compared to traditional execution [4, 37, 65]. Furthermore, most symmetric primitives work over bits, while many of the protocols use arithmetic modulo p, making conversions necessary [3, 6, 10, 11]. The field of Arithmetization Oriented Symmetric Primitives (AOSPs) has recently emerged to address these two issues and thus speed up the evaluation of symmetric primitives when part of an MPC, FHE, or ZK proof system.

The potential for optimization arises from a similarity shared by all three protocol types: A non-linear operation, i. e. multiplication, is far less efficient than a linear operation, i. e. addition. This is not the case when considering traditional cost models for computation, where additions and multiplications take roughly the same time to execute in software implementations, and (linear) XOR and (non-linear) AND gates use about the same amount of space on a chip in hardware implementations [4]. In contrast, the cost models for MPC, FHE, and ZK proof systems are of asymmetric nature: Additions can be performed locally while multiplications require time consuming communication between parties in MPC when using *additively homomorphic secret sharing*, which allows easy distribution of linear operations; evaluation of an XOR gate requires only local xor-ing of labels while AND gates need to be sent by time consuming Oblivious Transfers in MPC when using garbled circuits [48]; additions add only a little noise to an FHE ciphertext while multiplications increase the noise a lot; and additions do not increase the size of a polynomial representing a computational trace for a ZK proof, while multiplications do [10, 11]. Even though there are subtle differences in the exact metrics for the different protocol types [5], we use "number of multiplications" as an sufficiently approximate metric for this thesis.

In the last few years, several AOSPs have been proposed. These primitives minimize the use of multiplications while aiming to provide the same levels of security as their traditional counterparts, like the AES [26] or SHA-3 [14]. The initial proposition in this line, LowMC [4], is defined over binary extension fields $\mathbb{F}_2^n$, which we will not regarded further in this thesis. Currently, four major designs for Arithmetization Oriented Ciphers (AOCs) using modulo-p arithmetic are proposed: The Hades framework [42], the MARVELlous suite [5, 8],

MiMC [1], and Generalized MiMC (GMiMC) [3]. The latter two works also propose the Arithmetization Oriented Hash Functions (AOHFs) MiMChash and GMiMCHash based on the Sponge construction [12]. All AOSP proposals primarily use algebraic operations optimized for the respective efficiency metric as their building blocks, thus differing from traditional cipher design. Particular attention needs to be put on *algebraic attacks* due to the algebraically simple nature of the proposals. One such attack vector are *interpolation attacks*, successfully used against a version of MiMC [54]. The question whether a similar type of attack works against GMiMC was left open.

GMiMC and derived GMiMCHash are competitors in a recent challenge about AOSPs [72] and have very competitive evaluation times for large amounts of data. This thesis is an effort to improve the understanding and knowledge about the design principle of GMiMC, namely Unbalanced Feistel Networks (UFNs). More specifically, the following two questions are considered:

- What is the upper bound on the number of rounds for which UFNs are vulnerable to interpolation attacks?
- What is the upper bound on the number of rounds for which Sponge constructions with UFNs are vulnerable to collision, preimage, and second preimage attacks?

1.2 Outline and Contributions

In this thesis, a form of *Lagrange interpolation* using only constant amount of memory is applied to analyze block ciphers based on the UFN paradigm in both variants Expanding Round Function (ERF) and Contracting Round Function (CRF) over finite fields $\mathbb{F}_p$ of prime order. This analysis is directly applicable to GMiMC in both variants ERF and CRF. It positively answers the open question whether the interpolation analysis of MiMC in its Feistel variant is applicable to GMiMC [54]. Albeit more intricate, the methodology is similar to the analysis of Feistel MiMC: First, a key-dependent coefficient of the interpolation polynomial is computed. Then, after fixing some of the inputs, the actual coefficient of the polynomial is recovered through interpolation. Lastly, the key is recovered using the previously introduced Greatest Common Divisor (GCD) approach or through a root finding technique, which is introduced for the first time. Also for the first time, multiple branches are taken into account when analyzing UFNs with ERF, improving run time complexities of the analysis. Summarizing the results, key recovery is more efficient than brute force for UFNs with ERF with a number of rounds smaller than $\lceil \log_d p \rceil + 2t - 2$, while for UFNs with CRF it is more efficient if the number of rounds is smaller than $\lceil \log_d p \rceil + t - 1$, where d is the algebraic degree of the round function, p is the size of the finite field, and t is the number of branches of the UFN.

An analysis of hash functions based on the Sponge paradigm using UFNs is also undertaken by extending the methods used in the block cipher analysis. More specifically, Lagrange interpolation or symbolic evaluation combined with the root finding technique allows mounting an algebraic correcting-last-block attack. The proposed method is applicable for finding collisions, preimages, and second preimages. Summarizing the result, for hash functions generating exactly one field element collisions can be found more efficiently than the birthday bound if the UFN has fewer rounds than $\lceil \log_d p \rceil + t - 1$.

All the discussed methods are demonstrated through application in proof-of-concept experiments. The results are presented in Sections 3.3.5 and 3.6.2 and Table 3.4.

The rest of this document is structured as follows. In Chapter 2, the mathematical and technical background is discussed. This includes an overview of some symmetric primitives, with a focus on UFNs. Furthermore, a form of Lagrange interpolation using constant memory is introduced. Chapter 3 analyzes UFNs algebraically and proposes several attack vectors for key recovery on the different UFNs variants. The analysis is extended to UFN-based Sponge constructions. Our results are summarized in Chapter 4.

2. Preliminaries

Modern cryptography has evolved to create a fundamental understanding of how to selectively share information. This is a non-trivial challenge when considering the presence of potentially malicious third parties. A cryptographic protocol is thus one which can fulfill *only* its intended functionality, even when actively attacked [39]. Complex protocols usually rely on so called *cryptographic primitives* as building blocks. These primitives include *block ciphers*, *hash functions*, *message authentication codes*, and many more. A cryptographic protocol can have completely different security goals than the primitives it uses, but rely on their security to achieve these goals. This makes the primitive's resilience to attacks detrimental. Even though formal security models exist, they are not applicable to any concrete block cipher, requiring rigorous cryptanalysis of a primitive instead to demonstrate its resilience to attacks.

In order to provide one such cryptanalytic effort, we first give some mathematical background and basic notation in Section 2.1. The focus is then shifted to block ciphers in general as well as the cipher motivating this thesis, Generalized MiMC (GMiMC), and its design paradigm of Unbalanced Feistel Networks (UFNs) in particular, in Section 2.2. We introduce the cryptographic primitive of hash functions and the Sponge construction, a popular approach also used to realize GMiMCHash, in Section 2.3. Lastly, the concept of and techniques for polynomial interpolation are introduced in Section 2.5.

2.1 Mathematical Background

This section covers the algebraic concepts as well as some notions from the field of computer science this thesis builds on. More concretely, we recapitulate finite fields, polynomials over finite fields, the factor theorem, and Fermat's little theorem.

Finite Fields

A finite field $\mathbb{F}_q$ is a set of size q for which the operations "addition" and "multiplication" satisfy the field axioms. These entail associativity of addition and multiplication; commutativity of addition and multiplication; neutral elements for addition and for multiplication (usually referred to as "0" and "1", respectively); inverse elements under addition

+	**0**	**1**
0	0	1
1	1	0

(a) Addition

·	**0**	**1**
0	0	0
1	0	1

(b) Multiplication

Table 2.1: Addition and multiplication tables for $\mathbb{F}_2$, corresponding to operations XOR and AND on bits.

(i. e. subtraction); inverse elements under multiplication except for 0 (i. e. division); and distributivity of multiplication over addition.

The *order* of a finite field $\mathbb{F}_q$ is its size, i. e. the number of elements q. All finite fields of the same order q are isomorphic [60] and therefore simply identified as $\mathbb{F}_q$. A finite field of order q exists if and only if q is a prime power $q = p^n$, $p \in \mathbb{P}$, $n \in \mathbb{N}$.

Example 2.1. Examples for finite fields are the objects $\mathbb{Z}/p\mathbb{Z}$ where $p \in \mathbb{P}$ is prime. The addition and multiplication tables of field $\mathbb{F}_2 \simeq \mathbb{Z}/2\mathbb{Z}$ are given in Table 2.1. Field $\mathbb{F}_2$ and its vector space extensions $\mathbb{F}_2^n$ are of particular interest in computer science because of their natural correspondence to bits and bit strings, respectively.

The *multiplicative group* $\mathbb{F}_p^*$ of a finite field $\mathbb{F}_p$ consists of all elements invertible under multiplication, i. e. $\mathbb{F}_p^* = (\mathbb{F}_p \backslash \{0\}, \cdot)$. A generator α of $\mathbb{F}_p^*$, i. e. $\langle \alpha \rangle = \mathbb{F}_p^*$, is called a *primitive element* of $\mathbb{F}_p$. Thus, each non-zero element in $\mathbb{F}_p$ can be written as a power of α.

Polynomials over Finite Fields A polynomial is an expression combining variables with constants from an underlying algebraic object (e. g. $\mathbb{F}_p$) using only addition, multiplication, and exponentiation to a non-negative power. The variables are commonly referred to as *indeterminates*. A polynomial with a single indeterminate is *univariate*. Combining more than one variable leads to *multivariate* polynomials. Any polynomial f univariate in x can be represented as

$$f = \sum_{i=0}^{d} a_i x^i$$

where each summand $a_i x^i$ is called *term* and the constant a_i is that term's *coefficient*. The coefficients are elements from some underlying algebraic object, e. g. $\mathbb{F}_p$. This can be expressed by writing $f \in \mathbb{F}_p[x]$, i. e. "the polynomial f in indeterminate x with coefficients in $\mathbb{F}_p$."

The exponent of the indeterminate of a term is the *degree* of that term, e. g. $a_i x^i$ has degree i. The *degree of the polynomial* is the largest degree of its terms with non-zero coefficient, i. e. $\deg(f) = d$ assuming $a_d \neq 0$. If the coefficient of that term with the largest exponent is 1, the polynomial is called *monic*, i. e. $f = x^d + \sum_{i=0}^{d-1} a_i x^i$.

Note. For ease of writing, we informally refer to the coefficient of the term with the highest degree as the *highest coefficient* in this thesis. The same holds for the coefficient of the term with second highest degree being referred to as *second highest coefficient*.

The Factor Theorem Polynomial f can be seen as a function with argument x, i. e. $f : x \mapsto f(x)$. With this perspective, equation

$$f(x) = 0$$

is called the *polynomial equation* associated with f. The solutions of the polynomial equation are called the *roots* of f.

The *Factor Theorem* states that r is a root of f if and only if $(x - r)$ is a linear factor of f. In other words, there exists a polynomial f' such that $f(x) = f'(x)(x - r)$.

Fermat's Little Theorem Let $\mathbb{F}_p$ be a finite field of prime order and a an element of $\mathbb{F}_p$. Then, *Fermat's Little Theorem* states the following:

$$a^p \equiv a$$

This has direct implications for the maximum degree of a polynomial $f \in \mathbb{F}_p[x]$. More precisely, over domain $\mathbb{F}_p$, the function of any polynomial f of degree p or greater is equivalent to the function of a polynomial g of degree at most $(p - 1)$.

Example 2.2. Consider the function of polynomial $f = x^7 + 2x^5 + 4x^3 \in \mathbb{F}_5[x]$, namely $\mathbb{F}_5 \to \mathbb{F}_5, x \mapsto f(x)$. From Fermat's little theorem follows that $x^5 \equiv x$, and $x^7 = x^2x^5 \equiv x^3$, thus $f(x) \equiv 5x^3 + 2x \equiv 2x = g(x)$. The example demonstrates that the function of polynomial f, which has degree $\mathsf{deg}(f) = 7$, is equivalent to the function of polynomial g with degree $\mathsf{deg}(g) = 1$.

Note. For the remainder of this thesis, we don't distinguish between a polynomial and its associated function.

Big-O notation

In computer science, and specifically in algorithmics, the amount of resources required by an algorithm is of great importance for its classification. The two resources most commonly considered in this context are *run time* and *memory* or *space requirements* In cryptography, an additional resource commonly considered is *number of executions of a function f*. A uniform way to talk about the resource requirements is the algorithm's *asymptotic behavior* on inputs of size n. The required amount of a resource can be described by a function g. For asymptotic behavior, the "details" of g do not matter. This notion of "ignoring details" is formally captured in *Big-O notation*, which describes the order of the growth rate, denoted $\mathcal{O}(g)$.

Note. For more concise summaries, we also use *soft-O* in this document. Notated as $\tilde{\mathcal{O}}(g(x))$, soft-O ignores any logarithmic factors $\log^i x$ in $g(x)$, behaving like Big-O otherwise.

Note. We use mathematically correct "$f(x) \in \mathcal{O}(g(x))$", not colloquial "$f(x) = \mathcal{O}(g(x))$."

Example 2.3. An example can be found in Figure 2.1. Even though f is sometimes bigger than g, the growth rate of g is ultimately faster, starting from n_0.

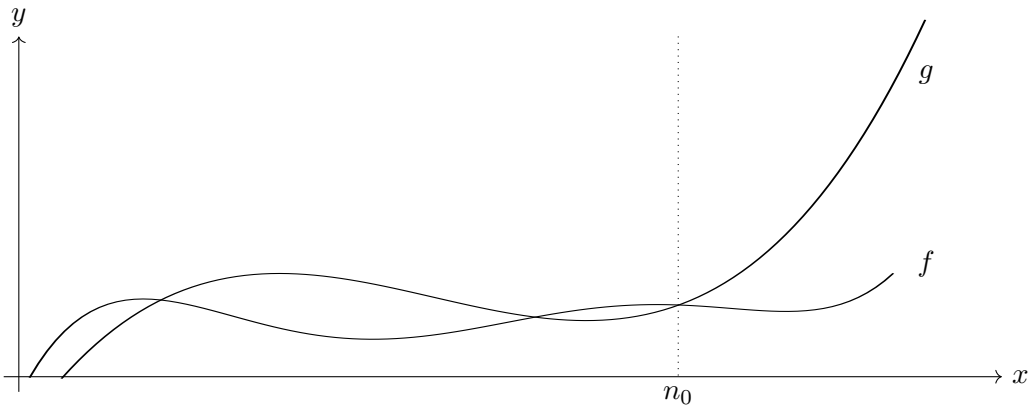


Figure 2.1: Big-O notation: $f \in \mathcal{O}(g)$.

2.2 Block Ciphers

A *block cipher* is a pair of deterministic algorithms $(\mathsf{Enc}, \mathsf{Dec})$ both acting as permutation on some set $M = \{0, \ldots, q-1\}^n$ with $q \in \mathbb{Z}^+$ and traditionally $q = 2$. The algorithms perform operations commonly referred to as *encryption* and *decryption*, respectively. Both algorithms depend on an additional value $k \in K$, the *key* from key space K. For any k, algorithm Dec_k is the inverse of Enc_k:

$$\begin{aligned} \mathsf{Enc} &: K \times M \to M \\ \mathsf{Dec} &: K \times M \to M \\ \forall k \in K : \forall m \in M &: \mathsf{Dec}_k(\mathsf{Enc}_k(m)) = m \end{aligned}$$

For a *message*, or *cleartext*, $m \in M$, the value $c = \mathsf{Enc}_k(m)$ is the associated *ciphertext* under key k.

Security In order to argue about a block cipher's security, an attacker trying to break the cipher can be assumed. We further need to specify the attacker's capabilities and her goal to define what "security" means. For capabilities, consider the *known plaintext* scenario, where it is assumed that the attacker knows the decryption of some ciphertexts. The *chosen plaintext* notion allows the attacker to additionally choose which ciphertexts she learns. A possible goal is recovery of the message for a given ciphertext. Another option is *ciphertext predictability*, where security of the cipher is considered broken if the attacker successfully predicts the ciphertext for a given message. If one can reasonably argue that no attacker with specific capabilities can achieve the stated goal for a certain cipher, it is considered secure in that scenario. More notions for both capabilities and goals exist, the discussion of which is beyond the scope of this thesis. Any analysis is only considered to be a successful attack if the computational effort is less than brute forcing the key, further discussed in Section 2.2.2.

In this thesis we argue about the security of a cipher by considering its resilience to cryptographic analysis, or *cryptanalysis*. Over the past decades, many powerful attack strategies have been developed: Linear cryptanalysis [58, 59, 61], differential cryptanalysis [16, 51, 52], slide attacks, meet-in-the-middle attacks [29, 30], boomerang attacks [15, 79] and many

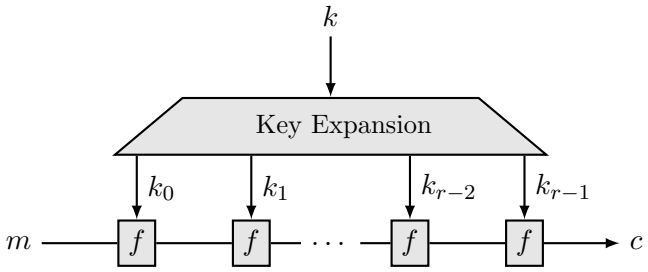


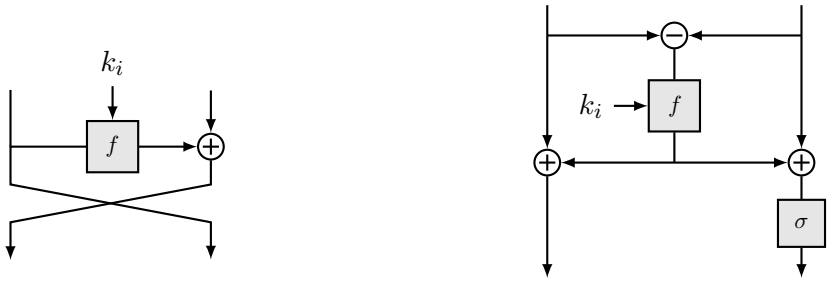
Figure 2.2: Generic iterated cipher.

more have been successfully applied to break ciphers. A more detailed discussion of some common attack vectors can be found in Section 2.2.2.

Note. The notion of *provable security* exists but is, to the best of our knowledge, not applicable to the block ciphers we consider in this thesis.

Design principles The definition and requirements above do not indicate *how* to build a secure block cipher. Over the last decades, several "best practices" have emerged. One of those principles, prevalent to the point of being universal, is that of the *iterated* block cipher. In essence, instead of constructing the entire cipher in one piece, a so called *round function* is designed. The block cipher's key is incorporated in the round function, often in the form of *round keys* generated by applying a *key expansion* function to the key. The round function by itself might only provide weak security properties, but is iterated over several *rounds*. This process amplifies the security properties if the round function is designed "correctly." Thus, designing a secure block cipher is essentially reduced to designing a secure round function and choosing the required number of rounds. A high-level view of a generic iterated cipher can be found in Figure 2.2. Constructing a block cipher in an iterated fashion eases (a) construction, (b) analysis, and (c) implementation of the cipher, explaining the popularity of the approach.

Principles on how to construct a round function producing a secure block cipher when iterated sufficiently many times also exist. For example, in a *Substitution Permutation Network (SPN)* the round function consists of two layers. The first applies non-linear *substitution boxes* (or *S-Boxes*), while the second applies a linear *permutation* (or *P-Box*). Orthogonal approaches to SPNs are *Feistel* [33] and *Lai-Massey* networks [50, 77]. Both these constructions turn any *function* into a *permutation*. This allows a wider design space for the round function at the cost of additional rounds. In fact, there are theoretical results for both Feistel and Lai-Massey networks stating that they behave like random permutations if instantiated with random functions [56, 57, 64]. Another advantage of both constructions is re-usability of an implementation for encryption and decryption: The inverse of a Feistel or Lai-Massey network with key $(k_0, \ldots, k_{r-1})$ is the same network with reversed key $(k_{r-1}, \ldots, k_0)$. One round of a general Feistel network is depicted in Figure 2.3a, while Figure 2.3b shows one round of a general Lai-Massey network, where σ denotes an orthomorphism. Due to our focus on Feistel networks, the interested reader is referred to the available literature for the Lai-Massey construction [47, 50, 77].



(a) Feistel (b) Lai-Massey

Figure 2.3: One round of networks often used to create block ciphers.

Examples Over the last decades, numerous block ciphers have been developed and standardized. Two of the most prominent examples are the Data Encryption Standard (DES) [33] and the Advanced Encryption Standard (AES) [27]. Both standards operate on bits, i. e. $q = 2$, although on different block sizes: The DES has message space $\{0,1\}^{64}$, the AES operates over $\{0,1\}^{128}$. Their design principles also differ vastly. Although both standards have an iterative design, the DES uses an SPN in a Feistel construction while the AES interprets the input as a matrix, successively permuting first elements, then rows, and lastly columns in each round. From today's viewpoint, both the DES and the AES are *traditional* block ciphers.

2.2.1 Unbalanced Generalized Feistel Networks

The original Feistel construction [33] outlined above is using two *branches*, visualized in Figure 2.3a. This can be generalized in one of two ways: (a) by using more than one (not necessarily distinct) round function distributing each function's output to exactly one branch [62], or (b) by still using only one round function and "bundling" many branches for either input or output [70]. The *balanced* generalized networks of approach (a) are not considered in this thesis. Approaches (b) are called Unbalanced Feistel Networks (UFNs). A UFN is usually constructed in one of two variants, Expanding Round Function (ERF) or Contracting Round Function (CRF), which are defined in the following and depicted in Figure 2.4.

Just like a Feistel network with two branches, a UFN is a permutation on the input space independent of whether or not the round function is a permutation itself. Again, this allows for more freedom when choosing the round function, at the expense of more rounds for the cipher. A security result like that for two-branch Feistel networks mentioned above also exists for UFNs [44]. Namely, if the round function is a (truly) random function, then the UFN is asymptotically indistinguishable from a (truly) random permutation after enough rounds. Again, this result is of great theoretical interest with limited applicability for a concrete round function operating on a set of fixed size.

Expanding Round Function A $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$ is a UFN with ERF over $\mathbb{F}_p$ with r rounds and t branches. Let $P_j^{(i)}$ be the value of branch j before round i. For example, $P_0^{(0)}$ is the leftmost input, and $P_{t-1}^{(r)}$ is the rightmost output. Let f be a monic polynomial over $\mathbb{F}_p$.

$$f = \sum_{i=0}^{d-1} a_i x^i + x^d$$

Applying f to the sum of the leftmost branch, the round key, and the round constant of round i gives the round function for that round, the output of which is denoted σ_i.

$$\sigma_i = f\left(P_0^{(i)} + k_i + c_i\right)$$

The output of the round function is added to the rightmost $t-1$ branches. After that, all the branches get rotated to the left by one position, with the leftmost branch going the rightmost position.

$$(P_0^{(i+1)}, \ldots, P_{t-1}^{(i+1)}) = (P_1^{(i)} + \sigma_i, \ldots, P_{t-1}^{(i)} + \sigma_i, P_0^{(i)}) \tag{2.1}$$

One round of a $\mathrm{UFN}_{\mathsf{erf}}$ is shown in Figure 2.4a.

Contracting Round Function A $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$ is a UFN with CRF over $\mathbb{F}_p$ with r rounds and t branches. Applying f to the sum of the rightmost $t-1$ branches, the round key, and the round constant of round i gives the round function for that round, the output of which is denoted σ_i.

$$\sigma_i = f\left(\sum_{j=1}^{t-1} P_j^{(i)} + k_i + c_i\right).$$

The output of the round function is added to the leftmost branch. After that, all the branches get rotated to the left by one position, with the leftmost branch going the rightmost position.

$$(P_0^{(i+1)}, \ldots, P_{t-1}^{(i+1)}) = (P_1^{(i)}, \ldots, P_{t-1}^{(i)}, P_0^{(i)} + \sigma_i) \tag{2.2}$$

One round of a $\mathrm{UFN}_{\mathsf{crf}}$ is shown in Figure 2.4b.

Note. We use the same nomenclature for both $\mathrm{UFN}_{\mathsf{erf}}$ and $\mathrm{UFN}_{\mathsf{crf}}$. In the remainder of the thesis, the variant being referred to is clear from context.

Generalized MiMC

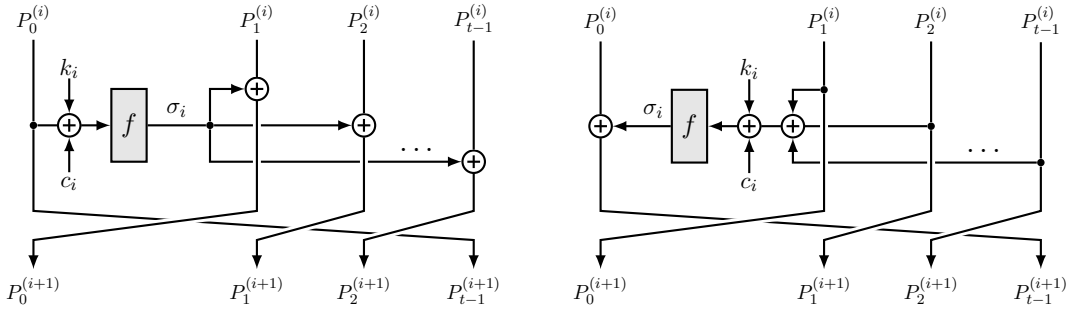
GMiMC [3] is an instantiation of UFNs. The round function used is cubing.

$$f(x) = x^3$$

The cubic function is an *Almost Perfect Nonlinear (APN) Function* providing excellent resistance against linear and differential cryptanalysis when used as round function in a block cipher [1, 63]. Additionally, cubing is algebraically simple, requiring only two multiplications.

GMiMC exists in both variants $\mathrm{GMiMC}_{\mathsf{erf}}$ and $\mathrm{GMiMC}_{\mathsf{crf}}$. Independent of the variant, there is *univariate* and *multivariate* GMiMC. In univariate GMiMC, only one round key $\bar{k}$ exists and is being used after applying the following key schedule [55].

$$k_i = (i+1)\bar{k}$$



(a) Expanding Round Function (b) Contracting Round Function

Figure 2.4: Unbalanced Feistel Network.

Note. The described key schedule corresponds to an update [55] of the proposal unpublished at the point of this thesis's writing. It prevents a recent attack on many variants of univariate GMiMC leveraging the previous key schedule $k_i = \bar{k}$ into a one-round characteristic [18].

In multivariate GMiMC, t keys $\bar{k}_i$ are used, where t is the number of branches of the UFN and r the number of rounds. The round key k_i for round i is discerned as follows:

$$(k_{j\cdot t}, k_{j\cdot t+1}, \ldots, k_{(j+1)\cdot t-1}) = \begin{cases} (\bar{k}_0, \bar{k}_1, \ldots, \bar{k}_{t-1}) & j = 0 \\ (k_{(j-1)\cdot t}, k_{(j-1)\cdot t+1}, \ldots, k_{j\cdot t-1}) \times M^\top & 1 \leqslant j \leqslant \lceil r/t \rceil \end{cases}$$

Matrix M is invertible and raising M to a power $1 \leqslant j \leqslant \lceil r/t \rceil$ must not introduce a zero at any position.

Note. Above key schedules are introduced for the sake of completeness. For univariate UFNs, i. e. with one round key, our analysis considers any linear key schedule. The analysis of multivariate UFNs, i. e. when using distinct round keys, is not limited to any specific key schedule.

2.2.2 Attack Vectors

Traditionally, block ciphers were almost always over bits strings, i. e. $M = \{0,1\}^n$. Thus, many attacks for this scenario have emerged, including *differential cryptanalysis*, *linear cryptanalysis*, and many more. These attacks are generally classified as *statistical* attacks, since they exploit the non-uniform distribution of certain bits or combination of bits. These attacks generally cannot be extended to work beyond bits in a straightforward manner. Because the analysis of this thesis is on UFNs over $\mathbb{F}_p$, the introduction of attacks limited to bits is omitted. The *interpolation attack* and the *Gröbner basis analysis* introduced below fall in the broader category of *algebraic attacks*, while the *brute force attack* is a generic attack against any keyed primitive.

Brute Force Attack Any analytic technique is considered an "attack" only if its computational complexity is lower than that of a brute force attack. A brute force attack is possible for any block cipher, and in fact for any keyed cryptographic primitive. It is the simplest attack conceivable: For a given pair of cleartext & ciphertext (m, c), the attacker tries all possible keys k in the key space K. If $\mathsf{Enc}_k(m) = c$, the correct key has been

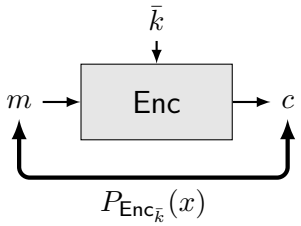


Figure 2.5: Interpolating a block cipher.

found. The only way to defend against brute force attacks is with a big enough key space. For example, with a key space of size $|K| = 2^{128}$ and one evaluation of Enc taking one femtosecond, the search would go on for orders of magnitude longer than the observable universe exists.

Interpolation Attacks Interpolation attacks were introduced to attack block ciphers constructed from algebraically simple components [1, 45, 54]. Using pairs of cleartext & ciphertext as *control points* for polynomial interpolation, a representation of the cipher in form of a polynomial is constructed. This representation exhibits the same behavior as the cipher. The technique of polynomial interpolation is explained in greater detail in Section 2.5. We give a high-level overview of the concept in Figure 2.5, where $P_{\mathsf{Enc}_{\bar{k}}}(x)$ is the polynomial recovered through interpolation. The computational complexity as well as the data complexity of the attack grows with the number of coefficients of the polynomial that is to be recovered, i. e. the degree, assuming *non-sparseness* of the polynomial. A cipher is secure against this attack if the polynomial is of maximum possible degree, which depends on the underlying field, making the attack more expensive than brute forcing the key.

Gröbner Bases One of the more advanced algebraic attack vectors are Gröbner basis analyses. A Gröbner basis is a generating set of an ideal over a polynomial ring with additional properties that ease computation [19]. By expressing a cipher as a compact and low degree set of multivariate polynomials with the key as a variable, the resulting variety can be described using the associated Gröbner basis. The multivariate system describes a zero dimensional ideal, and factorization of the basis allows to recover potential key candidates. Computing the Gröbner basis is computationally the most expensive step. Several different algorithms for this purpose exist, most notably F4 [32]. The run time complexity of these algorithms for general polynomial sets is not particularly well understood, making a cipher's resistance difficult to estimate.

The cipher *Jarvis* from the MARVELlous suite [8] was recently attacked successfully using a Gröbner basis analysis [2]. We don't further consider this attack vector in the remainder of this thesis.

2.3 Hash Functions

Cryptographic hash functions serve a multitude of purposes as cryptographic primitives. For example, they can be used for *message authentication codes* and in *digital signatures*.

The output of a hash function f is deterministic and called the message's *hash*. The input, i. e. the message, can be of arbitrary size while the hash is of fixed length.

$$f : \{0, \ldots, q-1\}^* \to \{0, \ldots, q-1\}^n$$

with $q \in \mathbb{Z}^+$ and traditionally $q = 2$, i. e. f operates on bits.

Security A hash function f needs to satisfy some security criteria in order to be considered a *cryptographic* hash function. The weakest property is *preimage resistance*: An attacker given hash h and function f should not be able to find the inverse of h, i. e. a message m with $f(m) = h$, faster than by brute forcing the message. If this is the case, f is called *one way*. Preimage resistance is implied[1] by *second preimage resistance*: An attacker given a message m_0 and f should not be able to find a second message m_1 with the same hash, i. e. $f(m_0) = f(m_1)$, faster than through brute force. Second preimage resistance, in turn, is implied by *collision resistance*: For any attacker, it should be infeasible to find two messages m_0, m_1 resulting in the same hash, i. e. $f(m_0) = f(m_1)$, faster than through brute force.

A generic collision attack on hash functions makes use of the *birthday paradox*. Roughly summarizing the result, brute forcing a collision by evaluating possible input values takes time in $\mathcal{O}(\sqrt{q^n})$ and not, as might be first suspected, in $\mathcal{O}(q^n)$. Intuitively, this is due to the fact that the hash of any new input value can be compared to *all* the previous hashes. Much like the brute force attack on keyed primitives described in Section 2.2.2, this attack is always possible. A collision attack is only considered successful if it outperforms the birthday attack.

Correcting Block Attacks A *correcting block attack* [66] is a way to find collisions, preimages, or second preimages. A hash function f that iteratively works on message blocks, i. e. messages of the form $M = (m_0, \ldots, m_n)$, can be susceptible to this type of attack. For second preimage attacks with message M, the general approach is as follows: The attacker chooses an arbitrary message $(m'_0, \ldots, m'_i)$ and then finds one or more correcting blocks $(m'_{i+1}, \ldots, m'_\ell)$ such that $f((m_0, \ldots, m_n)) = f((m'_0, \ldots, m'_\ell))$. For preimage and collision attacks, the approach is similar, where the correcting blocks are chosen to influence the hash value in the desired way. If only the last block m_{i+1} is required , the attack is a *correcting-last-block attack* [38, 66]. Correcting block attacks have been used to successfully attack the Message Digest Algorithm MD5 [67, 73, 74], among others [25, 38].

2.3.1 The Sponge Construction

The *Sponge construction* [12] is a way to turn a random permutation into a compressing random function. The length of the output can be fixed, allowing the construction of cryptographic hash functions. Sponges are defined over inputs and outputs from some *group*, for which we will consider $\mathbb{F}_p$ throughout this thesis. Internally, a state $\mathcal{S} = (\mathcal{S}_r, \mathcal{S}_c) \in \mathbb{F}_p \times \mathbb{F}_p^{t-1}$ is used,[2] where $r = \log_2 p$ is the so-called *rate* and $c = (t-1)\log_2 p$ the *capacity*. When evaluating a Sponge for message $(m_0, \ldots, m_\ell) \in \mathbb{F}_p^\ell$, the state is initiated to $\mathcal{S} = (0, \ldots, 0)$.

[1] Special care has to be taken with this implication. For some functions, it does not hold at all. For the others, a statistical argument depending on the relative size of domain and range is necessary [68].

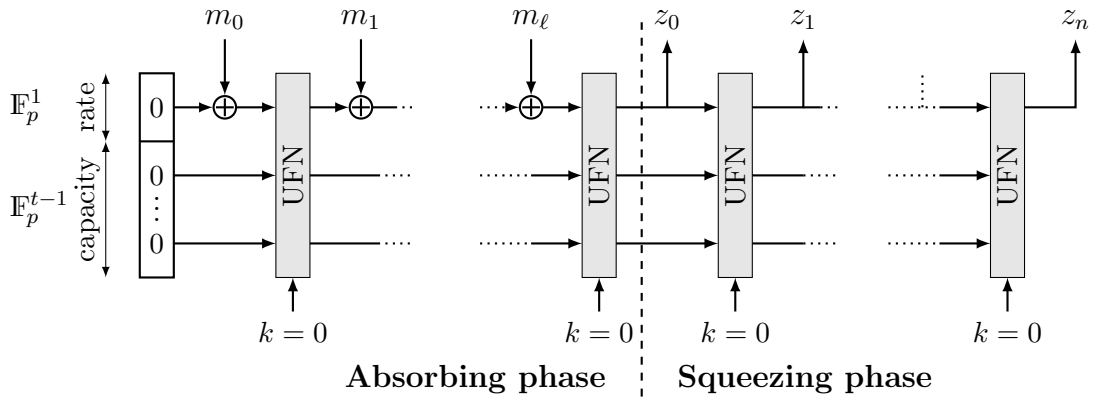


Figure 2.6: Sponge construction over $\mathbb{F}_p$ using a UFN with t branches. The rate is $r = \log_2 p$ and capacity is $c = (t-1)\log_2 p$.

At each iteration, a message block m_i is added to $\mathcal{S}_r$, after which the permutation f is applied to the current state, thus updating it. Once all message blocks are consumed, the *absorbing phase* ends and the *squeezing phase* starts. The current value of $\mathcal{S}_r$ is output, after which the state $\mathcal{S}$ is updated by applying permutation f. This is repeated until the output length has reached the desired length, which depends on the design choice and the use case of the Sponge.

A Sponge construction can be instantiated with a block cipher by fixing the key, turning the cipher into a permutation. In this thesis, we consider Sponges instantiated with UFNs and key $k = 0$. A visualization of such a Sponge construction can be found in Figure 2.6.

One of the most important theoretical results on Sponges states that the Sponge construction is indifferentiable from a random function if instantiated with a random permutation [13]. However, when a concrete permutation, like a block cipher with fixed key, is being used, the security of that instantiation needs to be established through cryptanalysis.

2.4 Block Ciphers & Hash Functions in Secure Computation

Both block ciphers and hash functions are often used as primitives in some bigger cryptographic protocol. The security of a specific cipher or hash function can influence the security, performance, and other properties of the protocol. If the protocol is executed in a Multi Party Computation (MPC) setting, using Fully Homomorphic Encryption (FHE), or employs Zero Knowledge (ZK) proof systems, performance is especially important since many of the current techniques are generally slow. Section 2.4.1 introduces the three settings mentioned above. In Section 2.4.2, generic and concrete propositions to increase performance are reviewed.

2.4.1 MPC, FHE, and ZK

The three cryptographic protocol types MPC, FHE, and ZK proof systems have gotten a lot of attention in the last few decades. Albeit efficiency is measured by respectively different metrics, as summarized in Section 1.1, all fields share the common trait that

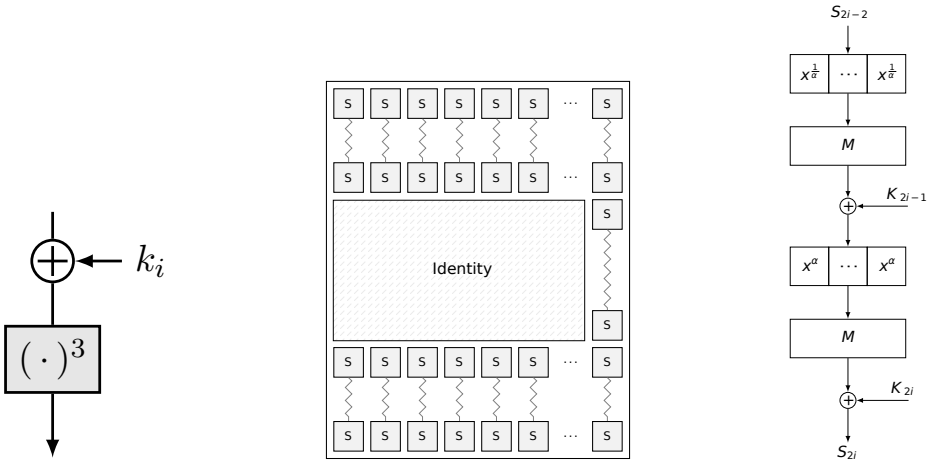
[2]The definition of Sponges considers more general states not relevant for this thesis.

non-linear operations are a lot more expensive than linear operations. This section roughly outlines the goals of the different areas as well as the challenges involved when evaluating block ciphers or hash functions in the different settings.

Multi Party Computation In MPC [22, 28, 40, 65, 80], as the name suggests, multiple parties $P_0, \ldots, P_n$ cooperate to jointly compute the output of a predefined function f. The inputs $i_0, \ldots, i_n$ to f held by the parties are private and should stay that way: Informally, no information on the inputs should leak, other than what can already be inferred from the computation's output $f(i_0, \ldots, i_n)$. The two main techniques used to achieve this are *garbled circuits* [81] and *verifiable secret sharing* [24], the explanations of which are beyond the scope of this document. Both techniques share a common trait: Linear operations, i. e. additions, are very easy to perform, while non-linear operations, i. e. multiplications, are more complicated, requiring time consuming communication between the parties. More concretely, when using linear verifiable secret sharing, additions can be performed by each party locally, while the parties need to cooperate for multiplications. When using garbled circuits, evaluation of the linear XOR gate requires only local addition, while the non-linear AND gates are evaluated by performing multiple Oblivious Transfers [48].

Fully Homomorphic Encryption A different field that recently has seen tremendous advancements is FHE [23, 31, 35, 36, 71]. This form of encryption allows computation on encrypted data. To be more precise, there are operations on ciphertexts which allow addition and multiplication of the underlying messages. Among other things, this allows to offload computations to a more powerful computer without having to trust the correct use of the shared data since the data is never accessible. Both, additions and multiplications introduce noise to the ciphertext. However, linear operations, i. e. additions, introduce little noise, while non-linear operations, i. e. multiplications, introduce a lot of noise. If the noise grows too big, the message cannot be recovered from its encrypted form. To prevent this, an operation called *boot strapping* is performed, reducing the noise. Boot strapping is computationally heavy and thus time consuming, and minimizing its need optimizes running time of the protocol.

Zero Knowledge Proof Systems A third protocol type sharing the similarity of cheaper linear operations are certain ZK proof systems [9, 41]. Research on ZK proof systems recently experienced a surge due to their applicability in privacy preserving blockchains [10, 11, 17, 20, 21]. A proof system is a protocol between two parties (P, V) in which P tries to convince V that a certain statement is true. P should be able to convince V only if the statement actually *is* true. A proof system has the ZK property if V learns nothing in the interaction apart from the fact that the statement is true. One kind of statements provable with ZK proof systems are the correctness of a computation. For this, a *computation's execution trace* is used, i. e. a step-by-step summary of the computation. For Scalable Transparent Arguments-of-Knowledge (STARKs) [10], the type of computation are often evaluations of a hash functions [69]. The execution trace of the hash function's evaluation is expressed as an Algebraic Intermediate Representation (AIR), i. e. a set of polynomials. Verifying the correctness of an AIR requires less computation if the polynomials are of low algebraic degree, e. g. the evaluated hash function is algebraically simple. Because addition of two polynomials does not raise the degree but multiplications



(a) MiMC [1] (b) Hades [42] (c) Rescue [5]

Figure 2.7: Central idea of three currently proposed AOC designs.

do, linear operations do not increase the computational complexity of ZK-STARKs, while non-linear operations do.

2.4.2 Arithmetization Oriented Symmetric Primitives

Traditional block ciphers like the AES [26] can be evaluated using MPC or FHE and hash functions like the Secure Hash Algorithm 3 (SHA-3) [14] can be used for ZK proof systems built on AIRs described above. However, being designed with different optimization metrics in mind, traditional primitives don't usually try to minimize the use of non-linear operations. This results in relatively poor performance in any of the three protocol types. Arithmetization Oriented Symmetric Primitives (AOSPs), on the other hand, are designed with the cost discrepancy between linear and non-linear operations in mind and thus address the potential for performance optimization.

Even though we treat the optimization metrics for MPC, FHE, and ZK proof systems summarized in Section 1.1 as the same throughout this thesis, differences do exist. The interested reader is referred to the introductory section of the MARVELlous cipher suite for an excellent treatment of the involved subtleties [5].

Current Designs The first cipher proposal optimized for a low number of non-linear operations was *LowMC* [4], followed by *MiMC* [1], the *MARVELlous suite* [5, 8] including *Rescue* and *Jarvis*, recently attacked using Gröbner basis analysis [2], *GMiMC* [3] as introduced in Section 2.2.1, and the *Hades* framework [42]. A very rough outline of some of these ciphers can be found in Figure 2.7.

The Arithmetization Oriented Cipher (AOC) proposals for MiMC, GMiMC, and the MARVELlous suite explicitly consider Arithmetization Oriented Hash Functions (AOHFs). These are *MiMChash*, *GMiMCHash*, and *Rescue hash*,[3] respectively. In all cases the AOHF is achieved by instantiating the Sponge construction as introduced in Section 2.3.1. The authors of LowMC also mention the possibility of using the cipher in a Sponge construction but omit a detailed discussion.

[3]The MARVELlous suite contains more hash functions, like *Vision hash*, defined over $\mathbb{F}_2^n$.

Challenges Like most cryptographic primitives, both AOCs and AOHFs strike a delicate balance between performance and security. Some attacks, especially polynomial interpolation attacks which we consider here, can be thwarted by raising the number of rounds, which negatively impacts performance. Adding more rounds increases the algebraic degree of the polynomial, increasing the computation time necessary for its recovery. This is further elaborated on in Section 2.5. A detailed analysis to find the minimum number of rounds necessary to defend against any given attack and taking the maximum of the results guarantees the best performance given maximum security. Naturally, this only works if all attack vectors can be anticipated. In practice, additional rounds are added as a security margin.

2.5 Polynomial Interpolation

The concept of interpolation refers to the construction of new data points given a set of known data points. This can be done in a multitude of ways, e. g. linear interpolation, or using splines. For the remainder of this thesis, we consider only *polynomial interpolation* over $\mathbb{F}_p$, albeit the introduced methods are not necessarily depending on the field. This approach to interpolation constructs a polynomial $f(x)$ of lowest possible degree for a set of known data points $((x_0, y_0), \ldots, (x_d, y_d))$. Under these constraints, f can have a maximal degree of $d-1$ for d independent points. Furthermore, f is unique.

The most straightforward way to interpolate a polynomial $L(x) = \sum_{j=0}^{d} m_j x^j$ is by solving $L(x_i) = y_i$ for the coefficients m_j. This amounts to inversion of the Vandermonde matrix (x_i^j), which has computational complexity of $\mathcal{O}(d^{2.8})$ and needs space in $\mathcal{O}(d^2)$ [78].

Lagrange Interpolation Another approach is Lagrange interpolation. From a linear algebra perspective, this amounts to a basis change such that recovering the coefficients requires inversion of the identity matrix, which is trivial. The new basis is composed of Lagrange basis polynomials

$$\ell_j(x) = \prod_{\substack{0 \leqslant i \leqslant d \\ i \neq j}} \frac{x - x_i}{x_j - x_i}$$

which depend on the known data points. Linearly combining the basis polynomial with y_j as the weights, i. e.

$$L(x) = \sum_{j=0}^{d} y_j \ell_j(x)$$

directly gives the interpolation polynomial. The computational complexity is in $\mathcal{O}(d \log d)$ with space requirements linear in d [78].

Low Memory Polynomial Interpolation For recovering the full polynomial, the computational and space boundaries of Lagrange interpolation are optimal [75]. Indeed, even if only *one* coefficient is to be interpolated, the computational complexity cannot be improved beyond $\mathcal{O}(d \log d)$. However, when interpolating only one coefficient, a method with constant space complexity exists. For polynomials of large degree, this allows cryptanalytic interpolation attacks otherwise prevented by the required amount of memory. In the following, a way to interpolate the coefficient of the second highest term is presented.

The analysis of this thesis substantially relies on the resulting Algorithm 1. The full derivation is given by Li and Preneel [54].

One key insight is that storing the full set of points $((x_0, y_0), \ldots, (x_d, y_d))$ already requires memory in $\mathcal{O}(d)$. For low memory interpolation, the data points are instead generated in an online manner, i. e. "on the fly" during the execution of the algorithm. For the x-values, powers of a primitive element $\alpha \in \mathbb{F}_p$ are used, ensuring their distinctness. The y-values are given by an evaluation oracle $\mathcal{O}^{\mathsf{f}}$ which evaluates the polynomial f at a desired point, i. e. $\mathcal{O}^{\mathsf{f}}(x_j) = f(x_j)$.

Solving Lagrange's interpolation formula for the coefficient of the second highest term m_{d-1} gives

$$m_{d-1} = \sum_{j=0}^{d} f(\alpha^j) \frac{\beta_j}{\gamma_j}$$

where

$$\beta_j = \alpha^j - \sum_{i=0}^{d} \alpha^i,$$

$$\gamma_j = \prod_{\substack{0 \leqslant i \leqslant d \\ i \neq j}} (\alpha^j - \alpha^i).$$

Both β_j and γ_j can be computed recursively, allowing to reuse memory, thus further lowering the memory requirement. For β_j this is relatively straightforward. The more intricate recursive γ_j is given in the following:

$$\gamma_{j+1} = \gamma_j \cdot \alpha^d \cdot \frac{\alpha^j - \alpha^{-1}}{\alpha^j - \alpha^d}$$

Combining these insights results in Algorithm 1, which we replicate from [54] for convenience of the reader. The notation is adapted to be coherent with the rest of this document. Reiterating, computational complexity of Algorithm 1 is in $\mathcal{O}(d \log d)$, space complexity is in $\mathcal{O}(1)$, and data complexity is d.

Algorithm 1: Algorithm for Low Memory Interpolation of the coefficient of the second highest term of a polynomial over $\mathbb{F}_p$ [54].

Input: Algebraic degree d of the polynomial, primitive element $\alpha \in \mathbb{F}_p$, polynomial evaluation oracle $\mathcal{O}^{\mathsf{f}}$

Output: The coefficient z of the second highest term

$z \leftarrow 0$
$\beta \leftarrow -\sum_{i=0}^{d} \alpha^i$
$\gamma \leftarrow \prod_{i=0}^{d}(1-\alpha^i)$
$x \leftarrow 1$
for $i \in \{0, \ldots, d\}$ **do**
    $z \leftarrow z + \mathcal{O}^{\mathsf{f}}(x) \cdot \frac{\beta+x}{\gamma}$
    if $i < d$ **then**
        $\gamma \leftarrow \gamma \cdot \alpha^d \cdot \frac{x-\alpha^{-1}}{x-\alpha^d}$
        $x \leftarrow x \cdot \alpha$
return z

3. Low Memory Interpolation Cryptanalysis of UFNs

In this chapter, we take a look at Unbalanced Feistel Networks (UFNs) over $\mathbb{F}_p$ from an algebraic perspective. The insights gained are then applied in their analysis. We describe several key recovery attack vectors, all of which use polynomial interpolation as a core step. Some of the techniques are re-used to mount a *correcting-last-block attack* against Sponge constructions instantiated with UFNs. Parts of this chapter have been submitted for publication [7].

The remainder of this chapter is structured as follows. In Section 3.1, the algebraic foundations are set, regarding the output of a UFN as a vector of polynomials. We give a high-level overview of the attack vectors in Section 3.2. The details of the different approaches for $\mathrm{UFN}_{\mathsf{erf}}$, including optimizations of the involved complexities and experimental results, are given in Section 3.3. In Section 3.4, we point out the differences of applying the same approaches to $\mathrm{UFN}_{\mathsf{crf}}$. The results of applying our analysis to Generalized MiMC (GMiMC) can be found in Section 3.5. Finally, in Section 3.6, we propose algebraic correcting-last-block attacks against constructions based on UFNs and validate them experimentally.

3.1 Analysis of Output Polynomials

In this section, we analyze common properties of UFNs when seen as polynomials of the input and key variables. For example, take output branch j of a $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$ where the key values $K = (k_0, \ldots, k_{r-1})$ are regarded as indeterminates. Given indeterminate input $(x_0, x_1, \ldots, x_{t-1})$, output branch j can be interpreted as a multivariate polynomial in $\mathbb{F}_p[x_0, \ldots, x_{t-1}, k_0, \ldots, k_{r-1}]$. Fixing all but one of the input variables to an arbitrary constant will give a polynomial $\mathbb{F}_p[x, k_0, \ldots, k_{r-1}]$ instead. We extensively use this technique in this chapter.

3.1.1 Expanding Round Function Analysis

Our analysis starts with output polynomials corresponding to different branches after r rounds of a $\mathrm{UFN}_{\mathsf{erf}}$. In the beginning, only $\mathrm{UFN}_{\mathsf{erf}}[p, 4, 3]$ are considered, i. e. 4 rounds and

3 branches. Throughout the section, we progressively lift these limitations in order to give a clear picture of the analysis. The first generalization is for the number of rounds r, in Proposition 2. We generalize for the number of branches t in Proposition 3.

Throughout this chapter, we simplify presentation of the analysis by "ignoring" the round constants without loss of generality. More concretely, let k'_i and c_i be round key and round constant of round i, respectively. Our analysis then takes into account their sum $k_i = k'_i + c_i$. As will become clear throughout the chapter, this does not affect the analysis in any way while simplifying notation. Additionally, a round function of degree $\mathsf{deg}(f) \geqslant 3$ is assumed.

Proposition 1. *Given an input of the form (b, b, x) to a $\mathrm{UFN}_{\mathsf{erf}}[p, 4, 3]$, the output polynomials $P_0^{(4)}, P_1^{(4)}, P_2^{(4)} \in \mathbb{F}_p[x, k_0, k_1, k_2, k_3]$ for the 3 branches after 4 rounds have the following properties:*

1. $\mathsf{deg}\left(P_0^{(4)}\right) = \mathsf{deg}\left(P_1^{(4)}\right) = d^2$ *and* $\mathsf{deg}\left(P_2^{(4)}\right) = d$.
2. $\mathsf{coeff}\left(P_0^{(4)}, x^{d^2}\right) = \mathsf{coeff}\left(P_1^{(4)}, x^{d^2}\right) = 1$,
3. $\mathsf{coeff}\left(P_0^{(4)}, x^{d^2-1}\right) = \mathsf{coeff}\left(P_1^{(4)}, x^{d^2-1}\right) = d(a_{d-1} + d\beta)$

where $\beta = f\left(b + f(b + k_0) + k_1\right) + f(b + k_0) + k_2 = \sigma_0 + \sigma_1 + k_2$.

Proof Intuition. In the proof, we develop the output polynomials after 4 rounds in the 3 branches, then expand their form according to the binomial theorem. Finally, the coefficients of the term with second highest degree are collected. For an illustration of the branch development, see Figure 3.1.

Proof. After 3 rounds, the branches are:

$$\left(P_0^{(3)}, P_1^{(3)}, P_2^{(3)}\right) = (b + \sigma_1 + \sigma_2, b + \sigma_0 + \sigma_2, x + \sigma_0 + \sigma_1)$$

where

$$\sigma_0 = f(b + k_0),\ \sigma_1 = f(b + \sigma_0 + k_1),\ \text{and}\ \sigma_2 = f(x + \sigma_0 + \sigma_1 + k_2).$$

Finally, after 4 rounds, the output of the round function and then the $\mathrm{UFN}_{\mathsf{erf}}[p, 4, 3]$ is:

$$\sigma_3 = f(b + \sigma_1 + \sigma_2 + k_3)$$
$$\left(P_0^{(4)}, P_1^{(4)}, P_2^{(4)}\right) = (b + \sigma_0 + \sigma_2 + \sigma_3, x + \sigma_0 + \sigma_1 + \sigma_3, b + \sigma_1 + \sigma_2)$$

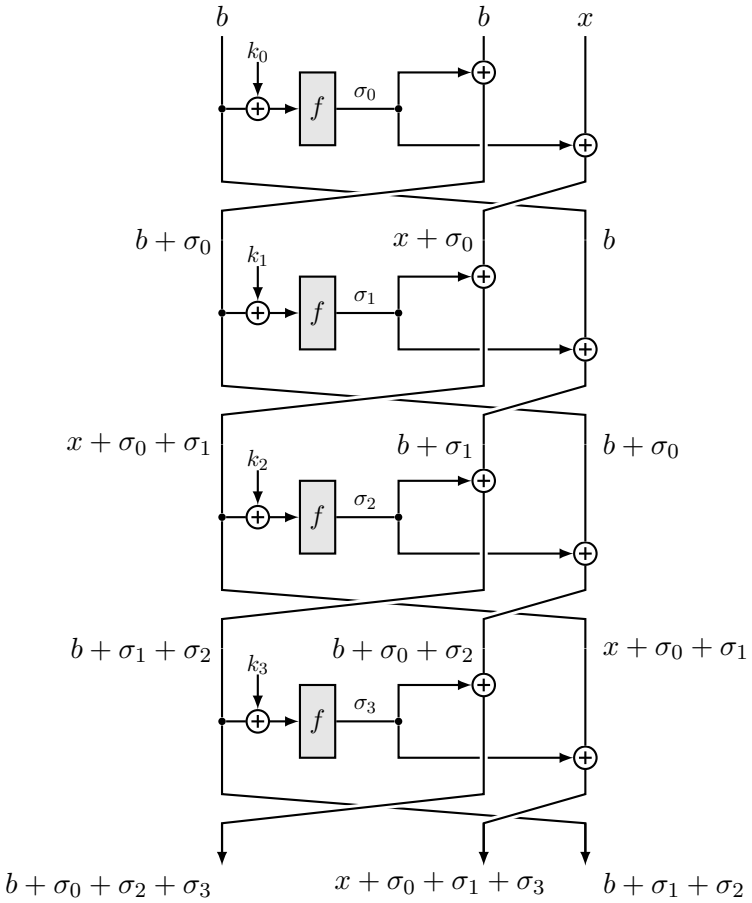


Figure 3.1: Branch development in a $\mathrm{UFN}_{\mathsf{erf}}[p, 4, 3]$.

The first observation is that σ_0 and σ_1 are independent of x. Further developing σ_2 and σ_3 using the binomial theorem yields the following:

$$\begin{aligned}
\sigma_2 &= f(x+\sigma_0+\sigma_1+k_2)\\
&= (x+\sigma_0+\sigma_1+k_2)^d + a_{d-1}(x+\sigma_0+\sigma_1+k_2)^{d-1} + \cdots + a_0\\
&= x^d + d(\sigma_0+\sigma_1+k_2)x^{d-1} + \binom{d}{2}(\sigma_0+\sigma_1+k_2)^2x^{d-2} + \cdots + (\sigma_0+\sigma_1+k_2)^d\\
&\quad + a_{d-1}x^{d-1} + a_{d-1}(d-1)(\sigma_0+\sigma_1+k_2)x^{d-2} + \cdots + a_{d-1}(\sigma_0+\sigma_1+k_2)^{d-2}\\
&\quad + \cdots + a_0\\
&= x^d + (a_{d-1} + d(\sigma_0+\sigma_1+k_2))x^{d-1} + \cdots + a_0\\
\sigma_3 &= f(b+\sigma_1+\sigma_2+k_3)\\
&= (b+\sigma_1+\sigma_2+k_3)^d + a_{d-1}(b+\sigma_1+\sigma_2+k_3)^{d-1} + \cdots + a_0\\
&= \sigma_2^d + d(b+\sigma_1+k_3)\sigma_2^{d-1} + \cdots + (b+\sigma_1+k_3)^d\\
&\quad + a_{d-1}(b+\sigma_1+\sigma_2+k_3)^{d-1} + \cdots + a_0\\
&= (x^d + (a_{d-1} + d(\sigma_0+\sigma_1+k_2))x^{d-1} + \cdots + a_0)^d\\
&\quad + d(b+\sigma_1+k_3)\sigma_2^{d-1} + \cdots + (b+\sigma_1+k_3)^d\\
&\quad + a_{d-1}(b+\sigma_1+\sigma_2+k_3)^{d-1} + \cdots + a_0\\
&= x^{d^2} + d(a_{d-1} + d(\sigma_0+\sigma_1+k_2))x^{d-1}(x^d)^{d-1} + \ldots a_0\\
&= x^{d^2} + d(a_{d-1} + d\beta)x^{d^2-1} + \ldots a_0
\end{aligned}$$

Since $\mathsf{deg}(\sigma_2) = d$ and $\mathsf{deg}(\sigma_3) = d^2$ and, by assumption, $d \geqslant 3$, this concludes the proof. □

Corollary 1. *In a* $\mathrm{UFN}_{\mathsf{erf}}[p,5,3]$*, it holds that* $P_2^{(5)} = P_0^{(4)}$ *in accordance with Equation (2.1). The following properties are a direct consequence of Proposition 1:*

1. $\mathsf{deg}\left(P_2^{(5)}\right) = d^2$
2. $\mathsf{coeff}\left(P_2^{(5)}, x^{d^2-1}\right) = d(a_{d-1} + d\beta)$.

The next proposition is a generalization of Proposition 1 for $r \geqslant 4$ rounds.

Proposition 2. *Given an input of the form* (b,b,x) *to a* $\mathrm{UFN}_{\mathsf{erf}}[p,r,3]$*, after* $r \geqslant 4$ *rounds, the output polynomials* $P_0^{(r)}, P_1^{(r)}, P_2^{(r)} \in \mathbb{F}_p[x,k_0,\ldots,k_{r-1}]$ *for the* 3 *branches have the following properties:*

1. $\mathsf{deg}\left(P_0^{(r)}\right) = \mathsf{deg}\left(P_1^{(r)}\right) = d^{r-2}$ *and* $\mathsf{deg}\left(P_2^{(r)}\right) = d^{r-3}$
2. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-2}}\right) = \mathsf{coeff}\left(P_1^{(r)}, x^{d^{r-2}}\right) = 1$
3. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-2}-1}\right) = \mathsf{coeff}\left(P_1^{(r)}, x^{d^{r-2}-1}\right) = d^{r-3}(a_{d-1} + d\beta)$
   where $\beta = f(b+k_0) + f\left(b + f(b+k_0) + k_1\right) + k_2 = \sigma_0 + \sigma_1 + k_2$.

Proof Intuition. We prove Proposition 2 by induction over r. The special case of $r = 4$ is shown in Proposition 1, giving the beginning of the induction. In the induction step, we apply the round function and developed the polynomials according to the binomial theorem.

Proof. Suppose that Proposition 2 holds for a fix r. Applying one more round $r+1$ yields the following, according to Equation (2.1):

$$\sigma_r = f\left(P_0^{(r)} + k_r\right)$$
$$\left(P_0^{(r+1)}, P_1^{(r+1)}, P_2^{(r+1)}\right) = \left(P_1^{(r)} + \sigma_r, P_2^{(r)} + \sigma_r, P_0^{(r)}\right)$$

Developing σ_r by the binomial theorem results in the following:

$$\begin{aligned}
\sigma_r &= (P_0^{(r)} + k_r)^d + \sum_{i=0}^{d-1} a_i (P_0^{(r)} + k_r)^i \\
&= \left(x^{d^{r-2}} + d^{r-3}(a_{d-1} + d\beta)x^{d^{r-2}-1} + \cdots + a_0 + k_r\right)^d + \sum_{i=0}^{d-1} a_i (P_0^{(r)} + k_r)^i \\
&= (x^{d^{r-2}})^d + d \cdot d^{r-3}(a_{d-1} + d\beta)x^{d^{r-2}-1}(x^{d^{r-2}})^{d-1} + \cdots + (d^{r-3}(a_{d-1} + d\beta)x^{d^{r-2}-1})^d \\
&\quad + \cdots + a_0^d + \cdots + k_r^d + \sum_{i=0}^{d-1} a_i (P_0^{(r)} + k_r)^i \\
&= x^{d^{r-1}} + d^{r-2}(a_{d-1} + d\beta)x^{d^{r-1}-1} + \cdots + a_0
\end{aligned}$$

By the assumption of the induction, $\mathsf{deg}(P_2^{(r)}) \leqslant \mathsf{deg}(P_1^{(r)}) \leqslant d^{r-2}$. Thus the degree of σ_r dominates, leading to the proof's first conclusions.

$$\mathsf{deg}\left(P_0^{(r+1)}\right) = \mathsf{deg}\left(P_1^{(r+1)}\right) = \mathsf{deg}\left(\sigma_r\right) = x^{d^{r-1}}$$
$$\mathsf{coeff}\left(P_0^{(r+1)}, x^{d^{r-1}}\right) = \mathsf{coeff}\left(P_1^{(r+1)}, x^{d^{r-1}}\right) = \mathsf{coeff}\left(\sigma_r, x^{d^{r-1}}\right) = 1$$

By assumption, $d \geqslant 3$ and $r \geqslant 4$, hence it holds that $d^{r-1} - 1 > d^{r-2}$. The coefficients of the second highest term in $P_0^{(r+1)}$ and $P_1^{(r+1)}$ are thus solely contributed by σ_r. This leads to the proof's last conclusion.

$$\mathsf{coeff}\left(P_0^{(r+1)}, x^{d^{r-1}-1}\right) = \mathsf{coeff}\left(P_1^{(r+1)}, x^{d^{r-1}-1}\right) = \mathsf{coeff}\left(\sigma_r, x^{d^{r-1}-1}\right) = d^{r-2}(a_{d-1} + d\beta)$$

□

The next generalization is for the number of branches t in the following proposition.

Proposition 3. *Given an input of the form $(b, \ldots, b, x)$ to a $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$, let $r > t \geqslant 3$, then after r rounds, the output polynomials $P_0^{(r)}, P_1^{(r)}, \ldots, P_{t-1}^{(r)} \in \mathbb{F}_p[x, k_0, \ldots, k_{r-1}]$ have the following properties:*

1. $\mathsf{deg}\left(P_0^{(r)}\right) = \ldots = \mathsf{deg}\left(P_{t-2}^{(r)}\right) = d^{r-(t-1)}$ *and* $\mathsf{deg}\left(P_{t-1}^{(r)}\right) = d^{r-t}$

2. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-(t-1)}}\right) = \ldots = \mathsf{coeff}\left(P_{t_2}^{(r)}, x^{d^{r-(t-1)}}\right) = 1$

3. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-(t-1)}-1}\right) = \ldots = \mathsf{coeff}\left(P_{t-2}^{(r)}, x^{d^{r-(t-1)}-1}\right) = d^{r-t-1}(a_{d-1} + d\beta)$
   where $\beta = \sum_{i=0}^{t-2} \sigma_i + k_{t-1}$.

Proof Intuition. The variable x is not part of the round function's input for the first $t-2$ rounds. Thus, the round function's output σ_{t-1} in round $(t-1)$ is comparable to σ_2 of Proposition 1, albeit differing in the respective β. This is the beginning of an induction over r like in Proposition 2.

Proof. Because of the position of the variable x, $(t-1)$ many "swappings" of branches need to be performed before x becomes part of the input to a round function. Each round of the UFN performs exactly one such swap. Thus, x does not contribute to σ_i for the first $(t-2)$ rounds, i. e. $\mathsf{deg}(\sigma_i) = 0$ for $i < t-1$. This leads to the following observations in the $(t-1)$-st round, much like in Proposition 1:

$$\begin{aligned}
\sigma_{t-1} &= f(x + \sigma_0 + \cdots + \sigma_{t-2} + k_{t-1}) \\
&= f(x + \beta) \\
&= (x+\beta)^d + a_{d-1}(x+\beta)^{d-1} + \sum_{i=0}^{d-2} a_i (x+\beta)^i \\
&= x^d + d\beta x^{d-1} + \cdots + d\beta^{d-1}x + \beta^d \\
&\quad + a_{d-1}x^{d-1} + a_{d-1}(d-1)\beta x^{d-2} + \cdots + a_{d-1}(d-1)\beta^{d-2}x + a_{d-1}\beta^{d-1} \\
&\quad + \sum_{i=0}^{d-2} a_i (x+\beta)^i \\
&= x^d + (a_{d-1} + d\beta)x^{d-1} + \cdots + a_0
\end{aligned}$$

An induction over r in the same way as in Proposition 2 concludes the proof. □

Corollary 2. *The property* $P_t^{(r)} = P_1^{(r-1)}$ *holds in accordance with Equation (2.1). From Proposition 3 we can thus summarize and further conclude:*

1. $\mathsf{deg}\left(P_t^{(r)}\right) = \mathsf{deg}\left(P_1^{(r-1)}\right) = d^{r-t}$

2. $\mathsf{coeff}\left(P_t^{(r)}, x^{d^{r-t}-1}\right) = \mathsf{coeff}\left(P_1^{(r-1)}, x^{d^{r-t}-1}\right) = d^{r-t-1}(a_{d-1} + d\beta)$

Corollary 2 states the algebraic expression of the coefficient of the term with second highest degree in the output polynomial $P_{t-1}^{(r)}$. In the remainder of this thesis, we informally refer to this coefficient as the *second highest coefficient*. Lastly, we generalize our result for the position of indeterminate x.

Proposition 4. *Given an input of the form* $(b, \ldots, b, x, b, \ldots, b)$ *to a* $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$*, where the position of* x *is* $\ell \in \{0, \ldots, t-1\}$*, after* $r > \ell$ *rounds, the network's output polynomials* $P_0^{(r)}, P_1^{(r)}, \ldots, P_{t-1}^{(r)} \in \mathbb{F}_p[x, k_0, \ldots, k_{r-1}]$ *have the following properties:*

1. $\mathsf{deg}\left(P_0^{(r)}\right) = \ldots = \mathsf{deg}\left(P_{t-2}^{(r)}\right) = d^{r-\ell}$ *and* $\mathsf{deg}\left(P_{t-1}^{(r)}\right) = d^{r-\ell-1}$
2. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-\ell}}\right) = \ldots = \mathsf{coeff}\left(P_{t-2}^{(r)}, x^{d^{r-\ell}}\right) = 1$
3. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-\ell}-1}\right) = \ldots = \mathsf{coeff}\left(P_{t-2}^{(r)}, x^{d^{r-\ell}-1}\right) = d^{r-\ell-1}(a_{d-1} + d\beta)$
   where $\beta = \sum_{i=0}^{\ell-1} \sigma_i + k_\ell$

Proof Intuition. Intuitively, the described situation is almost the same as regarding input $(b, \ldots, b, x)$ to a $\mathrm{UFN}_{\mathsf{erf}}[p, r, \ell + 1]$, making Proposition 3 applicable.

Proof. Using the same argumentation as in the proof of Proposition 3, we observe that $\mathsf{deg}(\sigma_i) = 0$ for $i < \ell$. The same expansion as in the proof of Proposition 3 results in the following expanded form for σ_ℓ:

$$\begin{aligned}\sigma_\ell &= f(x + \sigma_0 + \cdots + \sigma_{\ell-1} + k_\ell) \\ &= f(x + \beta) \\ &= x^d + (a_{d-1} + d\beta)x^{d-1} + \cdots + a_0\end{aligned}$$

An induction over r in the same way as in Proposition 2 concludes the proof. □

Corollary 3. *Let σ_i be the output of the round function in round i of a $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$ with input of the form $(b, \ldots, b, x, b, \ldots, b)$, where indeterminate x is at position ℓ, and $d \geqslant 3$. The proof of Proposition 4 allows us to make the following statements:*

$$\mathsf{deg}(\sigma_i) = \begin{cases} 0, & 0 \leqslant i < \ell \\ d^{i-\ell+1}, & \ell \leqslant i < r \end{cases}$$

Note. Generally, the output polynomials in Proposition 4 are of higher degree than those in Proposition 3, unless $\ell = t - 1$. In this case, Proposition 3 and Proposition 4 coincide.

Summarizing the results of above propositions, the output polynomial of lowest degree in a $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$ is $P_{t-1}^{(r)}$ assuming "optimal" input $(b, \ldots, b, x)$, and its second highest coefficient is described in Proposition 3.

3.1.2 Contracting Round Function Analysis

In this section, the degree as well as the coefficients of the highest and second highest terms of the output polynomials of $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$ are analyzed. As in Section 3.1.1, we notationally simplify the analysis by combining actual round key k'_i and round constant c_i to $k_i := k'_i + c_i$. Again, this does not impact the generality of our analysis. Furthermore, we continue to assume round functions of degree $\mathsf{deg}(f) \geqslant 3$.

Proposition 5. *Given an input of the form $(x, b, \ldots, b)$ to a $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$ with $t \geqslant 3$ branches, after $r \geqslant 2$ rounds, the rightmost output polynomial $P_{t-1}^{(r)} \in \mathbb{F}_p[x, k_0, \ldots, k_{r-1}]$ has the following properties:*

1. $\mathsf{deg}\left(P_{t-1}^{(r)}\right) = d^{r-1}$

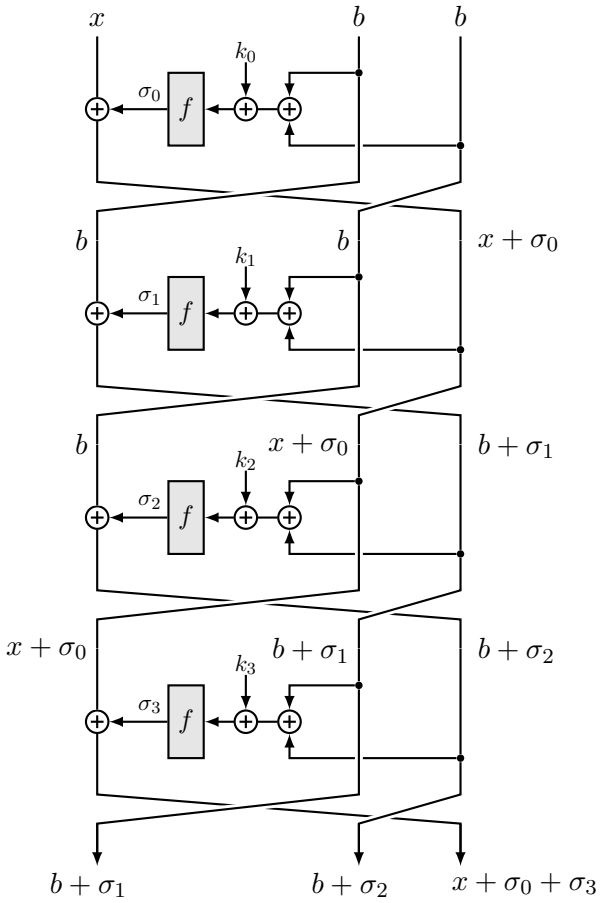


Figure 3.2: Branch development in a $\mathrm{UFN}_{\mathsf{crf}}[p, 4, 3]$.

2. $\mathsf{coeff}\left(P_{t-1}^{(r)}, x^{d^{r-1}}\right) = 1,$

3. $\mathsf{coeff}\left(P_{t-1}^{(r)}, x^{d^{r-1}-1}\right) = d^{r-1}(a_{d-1} + d\beta)$
   where $\beta = (t-2)b + f((t-1)b + k_0) + k_1$

Proof Intuition. Note that σ_0 is not dependent on x, but σ_1 is. Thus, an induction like in Proposition 2 with σ_1 as the beginning concludes the proof. For a visualizing example limited to 3 branches and 4 rounds, see Figure 3.2.

Proof. After 1 round, the round function's output and subsequently the branches are, in accordance with Equation (2.2):

$$\sigma_0 = f\left(\sum_{j=1}^{t-1} b + k_0\right) = f\left((t-1)b + k_0\right)$$

$$\left(P_0^{(1)}, \ldots, P_{t-2}^{(1)}, P_{t-1}^{(1)}\right) = (b, \ldots, b, x + \sigma_0)$$

After 2 rounds:

$$\sigma_1 = f\left(\sum_{j=1}^{t-2} b + x + \sigma_0 + k_1\right) = f(x+\beta)$$
$$\left(P_0^{(2)}, \ldots, P_{t-2}^{(2)}, P_{t-1}^{(2)}\right) = (b, \ldots, x+\sigma_0, b+\sigma_1)$$

Expanding σ_1 by the binomial theorem yields the following:

$$\begin{aligned}
\sigma_1 &= f(x+\beta) \\
&= (x+\beta)^d + a_{d-1}(x+\beta)^{d-1} + \sum_{i=0}^{d-2} a_i(x+\beta)^i \\
&= x^d + d\beta x^{d-1} + \cdots + \beta^d \\
&\quad + a_{d-1}x^{d-1} + a_{d-1}(d-1)\beta x^{d-2} + \cdots + a_{d-1}\beta^{d-1} + \sum_{i=0}^{d-2} a_i(x+\beta)^i \\
&= x^d + (a_{d-1} + d\beta)x^{d-1} + \cdots + a_0
\end{aligned}$$

After expanding σ_1, an induction over r like in Proposition 2 concludes the proof. □

Corollary 4. *Proposition 5 allows us to conclude for $r \geqslant t$:*

1. $\mathsf{deg}\left(P_0^{(r)}\right) = \mathsf{deg}\left(P_{t-1}^{(r-(t-1))}\right) = d^{r-t}$
2. $\mathsf{coeff}\left(P_0^{(r)}, x^{d^{r-t}-1}\right) = \mathsf{coeff}\left(P_{t-1}^{(r-(t-1))}, x^{d^{r-t}-1}\right) = d^{r-t-1}(a_{d-1} + d\beta)$

Corollary 4 gives the algebraic expression of the coefficient of the second term of second highest degree in the output polynomial $P_0^{(r)}$, the *second highest coefficient*. The same corollary shows that $P_0^{(r)}$ is the output polynomial of lowest degree in any $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$. The insight of Proposition 5 allows us to algebraically express the second highest coefficient in indeterminates k_i, specifically as a polynomial in $\mathbb{F}_p[k_0, k_1]$.

3.2 Attack Outline

In this section, we analyze $\mathrm{UFN}_{\mathsf{erf}}$ and $\mathrm{UFN}_{\mathsf{crf}}$ using the results from Section 3.1. The UFNs are instantiated with uniform randomly fixed but unknown key $\bar{K} \in \mathbb{F}_p^r$, i. e. round keys $(\bar{k}_0, \ldots, \bar{k}_{r-1})$. Let $(E_{\bar{K}}, D_{\bar{K}})$ denote the resulting cipher. When the keys are concrete values as opposed to indeterminates, the output polynomials developed in Sections 3.1.1 and 3.1.2, specifically in Corollaries 2 and 4, are elements of $\mathbb{F}_p[x]$ and not of $\mathbb{F}_p[x, k_0, \ldots, k_{r-1}]$. Since the interpolation of a single coefficient requires only constant memory, as outlined in Section 2.5, the second highest coefficient can be recovered to mount a low memory attack.

We describe the general idea of the cryptanalysis in the following steps:

1. Obtain the algebraic expression of the second highest coefficient $Q(k)$ of the output polynomial corresponding to the branch with the lowest algebraic degree. For $\mathrm{UFN}_{\mathsf{erf}}$ and $\mathrm{UFN}_{\mathsf{crf}}$ these are the rightmost and leftmost branch respectively.
2. Find value z of second highest coefficient of $E_{\bar{K}}$ of the same branch as in step 1 by applying the low memory interpolation technique recapitulated in Section 2.5.
3. Recover the key by evaluating relation $Q(K) = z$ by solving for K. Some of the key recovering techniques require multiple equations $Q_i(K) = z_i$.

Independent of the UFN variant, we explore two scenarios: (1) *identical round key* and (2) *distinct round keys*. For an identical, or single, round key it holds that $k_i = g_i(k)$, where g_i is a *linear* function of degree one over $\mathbb{F}_p$ and k can take values in $\mathbb{F}_p$. For single round keys, two different techniques are used: (a) the *Greatest Common Divisor (GCD)* technique previously used by [54] and (b) a novel *root finding* technique.

For $\mathrm{UFN}_{\mathsf{erf}}$, we further reduce the complexity of the key recovery through a technique called *branch subtraction*, introduced in Section 3.3.3. An overview is given in the respective sections Section 3.3.4 and Section 3.4.

3.3 Cryptanalysis of $\mathrm{UFN}_{\mathsf{erf}}$

In this section, we describe attack vectors on UFNs with Expanding Round Function (ERF). More concretely, details for the three steps outlined in the previous section are given. We describe UFNs with Contracting Round Function (CRF) in the next section, Section 3.4.

Algebraic Expression of the Second Highest Coefficient

As in Propositions 1 to 4, we consider the network's input to be fixed for all but one input branch. By arranging the terms, the output polynomial of any branch has the form $x^{d^y} + Q(K)x^{d^y-1} + \cdots + a_0$, where y and $Q(K)$ depend on the number of rounds r, the number of branches t, and the position ℓ of indeterminate x in the input. The coefficient $Q(K)$ is the polynomial we refer to as the *second highest coefficient*. This coefficient is computable by applying the results from Section 3.1, as described below.

In a $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$, the polynomial representing the rightmost output branch has the lowest degree, as shown in Proposition 3. For this polynomial, the coefficient of the second highest degree term has form $Q(K) = d^{r-t-1}(a_{d-1} + d\beta)$, with $\beta = \sum_{i=0}^{t-2} \sigma_i + k_{t-1}$. Algorithm 2 describes the method to obtain the polynomial $q(K)$, representing the second highest coefficient.

Complexity The computation of the polynomial representing the second highest coefficient requires multiplications of polynomials over $\mathbb{F}_p$. The complexity of multiplying two polynomials of degree at most D over $\mathbb{F}_p$ is $\mathcal{O}(d \log p \log(d \log p))$ [78]. Hence, this step has complexity in

$$\mathcal{O}\big(d^{t-1} \log p \log(d^{t-1} \log p)\big) = \mathcal{O}\big(d^{t-1}(t \log d + \log\log p) \log p\big) .$$

Space complexity is in $\mathcal{O}\big(d^{t-1}\big)$ since only one polynomial of degree at most d^{t-1} has to be stored at any given time.

Algorithm 2: Second highest coefficient of rightmost branch in $\mathrm{UFN}_{\mathsf{erf}}[p, r, t]$ on input $(b, \ldots, b, x)$.

Input: r, t, f, branch constant b, round constants $c_0, \ldots, c_{t-1}$
Output: polynomial $Q(K)$ for second highest coefficient of rightmost branch

$s := 0$
for $i \in (0, \ldots, t-2)$ **do**
  $\sigma_i := f(s + b + c_i + k_i)$
  $s := s + \sigma_i$
$\beta := s + k_{t-1} + c_{t-1}$
return $d^{r-t-1}(a_{d-1} + d\beta)$

Value of the Second Highest Coefficient

As outlined at the beginning of Section 3.2, the second step of the analysis consists of recovering the value of the second highest coefficient of the rightmost output polynomial branch of a $\mathrm{UFN}_{\mathsf{erf}}$ $E_{\bar{K}}$ with fixed but unknown key $\bar{K} = (\bar{k}_0, \ldots, \bar{k}_{r-1})$. For this step, we use the low memory interpolation technique described in Section 2.5. In general, inputs of form α^j are required, where $\alpha \in \mathbb{F}_p$ is a primitive element, $0 \leqslant j \leqslant D$, and D is the degree of the underlying polynomial that is to be interpolated. In the current analysis this means using inputs of the form $(b, \ldots, b, \alpha^j)$, in accordance with Proposition 3. The evaluation points y_j for the interpolation are the values of the rightmost output branch. The degree of this polynomial is $D = d^{r-t}$. More concretely, the inputs to Algorithm 1 are (1) degree D, (2) primitive element α, and (3) polynomial evaluation oracle $\mathcal{O}^{\mathsf{erf}}(x) = \mathsf{last\_component}\big(E_{\bar{K}}\big((b, \ldots, b, x)\big)\big)$. The result of the low memory interpolation is the value z.

Note. We clarify: $\mathsf{last\_component}\,((x_0, \ldots, x_n)) = x_n$.

Complexity The time complexity of finding the value of the second highest coefficient using low memory interpolation is in $\mathcal{O}(D \log D)$ for polynomials of degree D. Its memory complexity is in $\mathcal{O}(1)$, and data complexity is $D+1$. For $\mathrm{UFN}_{\mathsf{erf}}$, the degree of the rightmost output polynomial is $D = d^{r-t}$, resulting in time complexity in $\mathcal{O}\big((r-t)d^{r-t} \log d\big)$. The approach requires $d^{r-t} + 1$ pairs of plaintext & ciphertext and uses constant space, i. e. in $\mathcal{O}(1)$. We achieve better time and data complexities by combining branches, as described in Section 3.3.3.

3.3.1 Key Recovery with Single Round Key

We first consider the case of a single round key $k_i = g_i(\bar{k})$ for linear functions g_i. The round keys k_i are derived from the one secret key $\bar{k} \in \mathbb{F}_p$ by key schedule g_i. The polynomial $Q(k)$ representing the second highest coefficient and the value z of the second highest coefficient are recovered as described in the previous two sections. For finding the value of the secret key two different techniques can be employed: (a) finding the GCD and (b) finding roots.

Finding the GCD

The GCD technique was introduced in [1] and used in [54] to analyze two branch Feistel networks. The procedure is as follows. First, select two different input constants b, b' for

the $\mathrm{UFN}_{\mathsf{erf}}$. Using these, two different polynomials $Q(k), Q'(k)$ are obtained, as described in Algorithm 2. Polynomial $Q(k)$ uses b as its branch constant, while $Q'(k)$ uses b'. Next, the value of the second highest coefficient is interpolated twice: First using b yielding z, then using b' yielding z'.

By construction of $Q(k)$ it holds that $Q(\bar{k}) - z = 0$, where $\bar{k}$ is the secret key. From the factor theorem, recapitulated in Section 2.1, it follows that $(k - \bar{k})$ is a factor in $Q(k) - z$. By the same argument, $(k - \bar{k})$ is also a factor of $Q'(k) - z'$. With a high probability, this is the *greatest* common factor. Thus it holds that

$$\bar{k} = k - \gcd\big(q(k) - z, q'(k) - z'\big)$$

with high probability, where k is the indeterminate.

Complexity Finding the GCD of two polynomials of degree at most D over $\mathbb{F}_p$ has time complexity $\mathcal{O}\big(D \log^2 D \log\log D\big)$ [78]. For $\mathrm{UFN}_{\mathsf{erf}}$, the degree (in k) of the algebraic second highest coefficient is $D = d^{t-1}$. Hence the key recovery using the GCD method has time complexity in $\mathcal{O}\big(td^{t-1} \log^2 d \log\log d\big)$. The space complexity is in $\mathcal{O}\big(d^{t-1}\big)$.

Finding Roots

By construction, $Q(k)$ satisfies $Q(\bar{k}) - z = 0$, i. e. secret key $\bar{k}$ is a root of $Q(k) - z$. Identifying all roots of that polynomial equation thus raises a list of key candidates. With an additional pair of cleartext & ciphertext that had not been used during the interpolation, the correct key $\bar{k}$ can be identified from the list by trying decryption with all key candidates.

Note. The degree (in k) of the algebraic second highest coefficient $Q(k)$ is d^{t-1}. Over an algebraically closed field, the list of key candidates would thus be of size d^{t-1}. However, $\mathbb{F}_p$ is not algebraically closed, meaning that the list of key candidates might be significantly shorter. For example, the list in our experiments has an average length of less than 2 for polynomials of degree 27. More details can be found in Section 3.3.5. For a fundamental treatment of roots of random polynomials over finite fields, see [53], parts of which we summarize in Section 3.6.

Complexity Finding all roots without multiplicities of a polynomial with degree D over $\mathbb{F}_p$ has time complexity $\mathcal{O}\big(D \log^2 D \log(Dp) \log\log D\big)$ [78]. The list of key candidates has length at most D for a polynomial of degree D, allowing a check in $\mathcal{O}(D)$ time and space. This is dominated by finding the roots. For $\mathrm{UFN}_{\mathsf{erf}}$, the degree (in k) of the algebraic second highest coefficient $Q(k)$ is $D = d^{t-1}$. Hence the key recovery using the root finding method has time complexity in $\mathcal{O}\big(td^{t-1} \log^2 d \log(dp) \log\log d\big)$ and data complexity of 1.

GCD versus Finding Roots

Theoretically, the complexity of the root finding method is not better than of the GCD technique since the complexity of the root finding method depends on the size of the field $\mathbb{F}_p$. However, for realistic target constructions like GMiMC, the size of the field is bounded and the complexities thus roughly the same. Furthermore, the data complexity of the GCD approach is almost twice as large compared the root finding method, since the interpolation step has to be performed twice. More importantly, the root finding method can be used to find collisions in Sponge constructions, as elaborated on in Section 3.6. We present a comprehensive comparison of the different complexities in Section 3.3.4.

3.3.2 Key Recovery with Multiple Round Keys

In this section, we consider UFNs with general multiple keys $(k_0, \ldots, k_{r-1})$. As opposed to the variant with a single key from Section 3.3.1, the methods building on Proposition 3 are not directly applicable. Instead, we use the results of Proposition 4. Furthermore, multiple instances of the equation $Q_i(K) = z_i$ for different constants b_i, $i \in \{0, 1, 2\}$ are used. This is an adaptation of the approach used in [54] where the authors analyzed "traditional" Feistel networks with 2 branches.

In Proposition 4, let $\ell = 1$, which corresponds to inputs of the form $(b_i, x, b_i, \ldots, b_i)$. Then the second highest coefficient of the rightmost branch of the $\mathrm{UFN}_{\mathsf{erf}}$ is of the form $Q_i(K) = d^{r-2}(a_{d-1} + d\beta_i)$ where $\beta_i = \sigma_0 + k_1 = f(b_i + k_0) + k_1$. The second highest coefficient thus depends on only the first two sub keys. Recall that $E_{\bar{K}}$ is a concrete $\mathrm{UFN}_{\mathsf{erf}}$ with secret but fixed key $\bar{K} = (\bar{k}_0, \ldots, \bar{k}_{r-1})$. After obtaining the three equations $Q_i(K) = z_i$, they are first rearranged:

$$\begin{aligned} f(b_0 + k_0) + k_1 - \frac{z_0}{d^{r-1}} + \frac{a_{d-1}}{d} &= 0 \\ f(b_1 + k_0) + k_1 - \frac{z_1}{d^{r-1}} + \frac{a_{d-1}}{d} &= 0 \\ f(b_2 + k_0) + k_1 - \frac{z_2}{d^{r-1}} + \frac{a_{d-1}}{d} &= 0 \end{aligned} \tag{3.1}$$

For $0 \leqslant i, j \leqslant 2$, subtraction of above equations results in:

$$\Delta_{(i,j)} := f(b_i + k_0) - f(b_j + k_0) - \frac{z_i - z_j}{d^{r-1}} = 0 \tag{3.2}$$

As in Section 3.3.1, it holds by the factor theorem that $(k_0 - \bar{k}_0)$ is a factor of $\Delta_{(i,j)}$ due to the construction of $Q_i(K)$. Thus

$$\bar{k}_0 = k_0 - \gcd\left(\Delta_{(0,1)}, \Delta_{(0,2)}\right) \tag{3.3}$$

Substituting k_0 with $\bar{k}_0$ in any of Equation (3.1) yields $\bar{k}_1$.

$$\bar{k}_1 = \frac{z_0}{d^{r-1}} - \frac{a_{d-1}}{d} - f\left(b_0 + \bar{k}_0\right) \tag{3.4}$$

After recovering $\bar{k}_0$ and $\bar{k}_1$, a partial decryption of any ciphertext – or partial encryption of any cleartext – is possible, allowing to "peel off" two rounds. The method described in this section is then applicable again, allowing an iterative recovery of the entire key $\bar{K}$.

Complexity For the two sub keys $(\bar{k}_0, \bar{k}_1)$, computing the algebraic form of the second highest coefficient can be done in constant time and space. The computational complexity of getting the second highest coefficient's value through interpolation is in $\mathcal{O}(D \log D)$ for polynomials of degree D, with data complexity being $D + 1$. From Corollary 3 it follows that for the current scenario, $D = d^{r-1}$. The computational complexity is thus in $\mathcal{O}\big(rd^{r-1} \log d\big)$. Data complexity is $3d^{r-1} + 3$. The algebraic second highest coefficient $Q_i(K)$ has degree d (in k_0). Computing the GCD of polynomials of degree d has computational complexity in $\mathcal{O}\big(d \log^2 d \log\log d\big)$ [78]. This is dominated by the polynomial interpolation.

To recover the entire key $\bar{K}$, the three steps above have to be repeated $\lceil r/2 \rceil$ many times. This results in an overall computational complexity of

$$\mathcal{O}\big(r^2 d^{r-1} \log d\big)\,.$$

3.3.3 Complexity Improvements via Branch Subtraction

When analyzing a $\mathrm{UFN}_{\mathsf{erf}}[p,r,t]$ $E_{\bar{K}}$ with $\bar{K}=(\bar{k}_0,\ldots,\bar{k}_{r-1})$, improvements on the complexities discussed above are possible. From Corollary 3 it follows that $\mathsf{deg}(\sigma_i)=d^{i-\ell-1}$ for $i \geqslant \ell$ and inputs of the form $(b,\ldots,b,x,b,\ldots,b)$, where $b\in\mathbb{F}_p$ is a constant and indeterminate x is at position ℓ. After round i, by construction of $\mathrm{UFN}_{\mathsf{erf}}$, σ_i has been added to all branches except the rightmost one. As we extensively used in the proofs of Propositions 1 to 4, the degree of the output polynomial of any branch is dominated by the largest σ_i. Thus, somehow removing one or more of the highest σ_i from an output branch reduces the degree of the corresponding polynomial. A lower degree in turn allows interpolation with reduced time and data complexity. Since we use low memory interpolation, space complexity cannot be lowered further.

Example 3.1. As a crude example for the branch subtraction effect, consider the output branches in Figure 3.3, i. e. $P_0^{(5)}(x),\ldots,P_3^{(5)}(x)$. The output branch with the lowest degree, i. e. $P_3^{(5)}(x)$, has degree $\mathsf{deg}(P_3^{(5)}(x))=\mathsf{deg}(\sigma_3)=d$. Let $P'(x):=P_1^{(5)}(x)-P_0^{(5)}(x)=\sigma_1-\sigma_2$. Then, the degree of $P'(x)$ is $\mathsf{deg}(P'(x))=0$ since σ_3 is not a summand. This elimination of high degree σ_i is the basic idea behind branch subtraction.

Let output of $\mathrm{UFN}_{\mathsf{erf}}[p,r,t]$ $E_{\bar{K}}$ with input $(x_0,\ldots,x_{t-1})$ be the vector $\vec{o}$. We represent $\vec{o}$ using the matrix notation described in the following. Intuitively, matrix A permutes the inputs like the last operation in any one round of a $\mathrm{UFN}_{\mathsf{erf}}$. The matrix B accumulates the necessary σ_i, following the definition of a $\mathrm{UFN}_{\mathsf{erf}}$. Figure 3.3 is annotated accordingly.

$$\vec{o} := A^r\cdot\vec{x}+(\underbrace{B_{r \bmod t}|\overbrace{B|\ldots|B}^{\lfloor \frac{r}{t} \rfloor \text{ times}}}_{r \text{ columns}})\cdot\vec{\sigma} \tag{3.5}$$

where

$$A=\begin{pmatrix}— & e_2 & —\\ — & e_3 & —\\ & \vdots & \\ — & e_t & —\\ — & e_1 & —\end{pmatrix},\ \vec{x}=\begin{pmatrix}x_0\\ x_1\\ \vdots\\ x_{t-1}\end{pmatrix},\ B=\begin{pmatrix}0 & & & 1\\ & 0 & & \\ & & \ddots & \\ 1 & & & 0\end{pmatrix},\ \vec{\sigma}=\begin{pmatrix}\sigma_0\\ \sigma_1\\ \vdots\\ \sigma_{r-1}\end{pmatrix}$$

and $B_{r \bmod t}$ are the right $(r \bmod t)$ columns of B. Summarizing the dimensions, $A,B\in\mathbb{F}_p^{t\times t}$, $\vec{x}\in\mathbb{F}_p^t$, and $\vec{\sigma}\in\mathbb{F}_p^r$.

Note. Equation (3.5) is not recursive. Increasing r to $(r+1)$ leads to different dimensions in the composite matrix on the right hand side as well as in $\vec{\sigma}$.

Note. Output branches $\vec{o}$ are nonlinear in variables $\vec{x}$ despite the seemingly linear representation above, since the σ_i are nonlinear in $\vec{x}$.

Example 3.2. Consider a $\mathrm{UFN}_{\mathsf{erf}}[p,5,4]$ with inputs (b,b,b,x) like in Figure 3.3. This instance presents the following scenario:

$$\vec{o}=\begin{pmatrix}0&1&0&0\\0&0&1&0\\0&0&0&1\\1&0&0&0\end{pmatrix}^5\cdot\begin{pmatrix}b\\b\\b\\x\end{pmatrix}+\left(\begin{array}{c|ccc}1&0&1&1\\1&1&0&1\\1&1&1&1\\0&1&1&1\end{array}\right)\cdot\begin{pmatrix}\sigma_0\\\sigma_1\\\sigma_2\\\sigma_3\\\sigma_4\end{pmatrix}=\begin{pmatrix}b\\b\\x\\b\end{pmatrix}+\begin{pmatrix}\sigma_0 & +\sigma_2+\sigma_3+\sigma_4\\ \sigma_0+\sigma_1 & +\sigma_3+\sigma_4\\ \sigma_0+\sigma_1+\sigma_2 & +\sigma_4\\ \sigma_1+\sigma_2+\sigma_3 & \end{pmatrix}$$

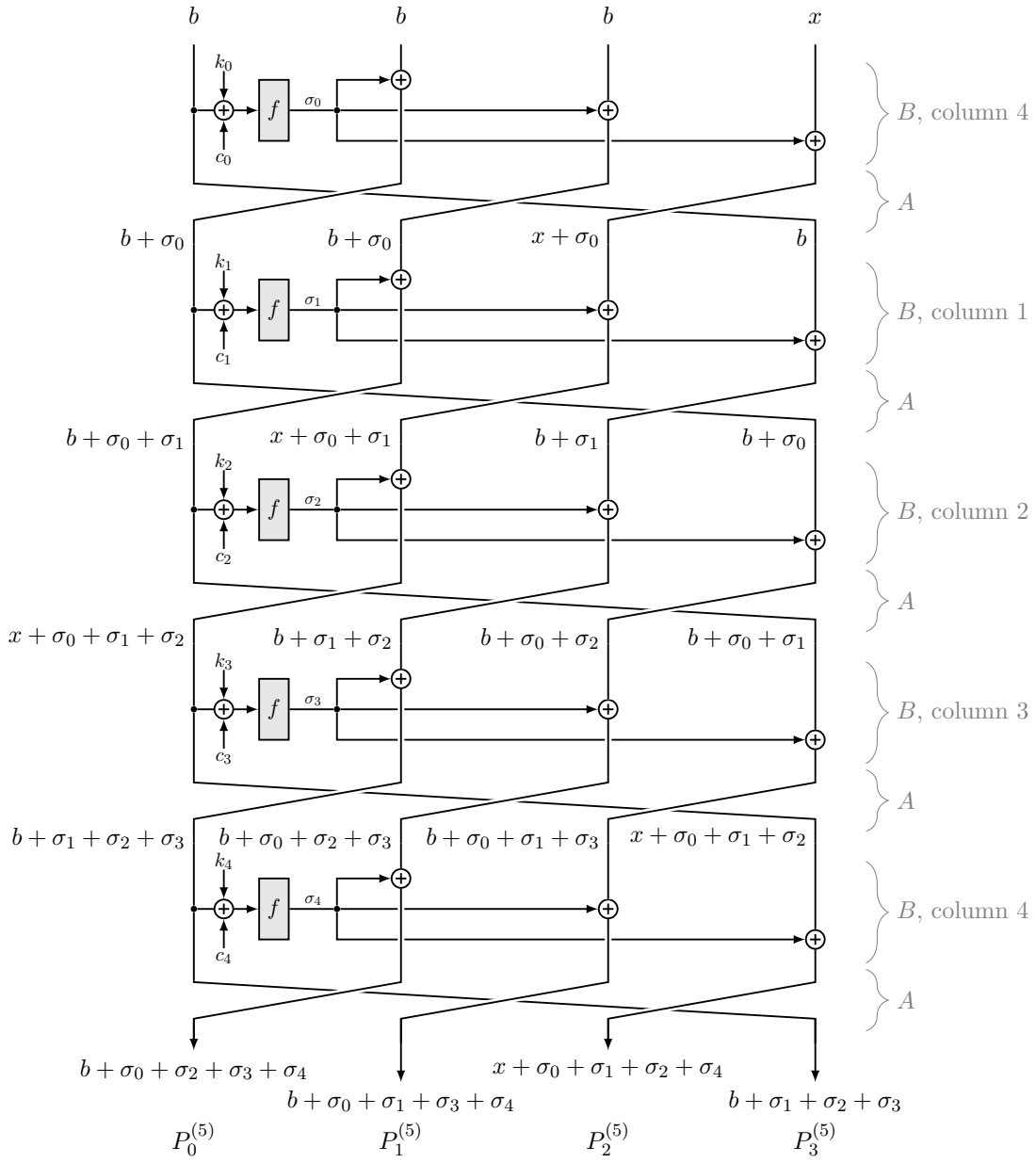


Figure 3.3: Example of summands being added in a UFN$_{\mathsf{erf}}$

Thus, $\vec{o}$ is an alternative representation of $P_0^{(5)}(x), \ldots, P_3^{(5)}(x)$.

Given this representation of the output branches $\vec{o}$ of $E_{\bar{K}}$, we apply some linear algebra in the following way. First, we observe the inverse of matrix B.

$$B^{-1} = (t-1)^{-1} \begin{pmatrix} 2-t & & & 1 \\ & 2-t & & \\ & & \ddots & \\ & 1 & & 2-t \end{pmatrix}$$

Multiplying the vector of output branches $\vec{o}$ by B^{-1} limits occurrence of any σ_i in any one component of $\vec{o}$ to exactly once. This corresponds to every σ_i occurring on only one "combined output branch," or more specifically:

$$B^{-1} \cdot \vec{o} = B^{-1} \cdot A^r \cdot \vec{x} + (\underbrace{I_{r \bmod t} | \overbrace{I_t | \ldots | I_t}^{\lfloor \frac{r}{t} \rfloor \text{ times}}}_{r \text{ columns}}) \cdot \vec{\sigma} \tag{3.6}$$

where $I_t \in \mathbb{F}_p^{t \times t}$ is the identity matrix and $I_{r \bmod t}$ are the right ($r \bmod t$) columns of I_t. Combining Equation (3.6) with Corollary 3 and inputs of the form $(b, \ldots, b, x)$ results in:

$$\begin{aligned} & \mathsf{deg}(\mathsf{first\_component}(B^{-1} \cdot \vec{o})) \\ = \; & \mathsf{deg}(\sigma_{r-t} + \sigma_{r-2t} + \cdots + \mathsf{first\_component}(B^{-1} A^r \vec{x})) \\ = \; & \mathsf{deg}(\sigma_{r-t}) \\ = \; & d^{r-2t+2} \end{aligned} \tag{3.7}$$

Note. We clarify: $\mathsf{first\_component}\,((x_0, \ldots, x_n)) = x_0$.

Example 3.3. *(Continued from Example 3.2.)* Applying the described operations results in the following:

$$3^{-1} \begin{pmatrix} -2 & 1 & 1 & 1 \\ 1 & -2 & 1 & 1 \\ 1 & 1 & -2 & 1 \\ 1 & 1 & 1 & -2 \end{pmatrix} \cdot \begin{pmatrix} b + \sigma_0 & + \sigma_2 + \sigma_3 + \sigma_4 \\ b + \sigma_0 + \sigma_1 & + \sigma_3 + \sigma_4 \\ x + \sigma_0 + \sigma_1 + \sigma_2 & + \sigma_4 \\ b & + \sigma_1 + \sigma_2 + \sigma_3 \end{pmatrix} = \begin{pmatrix} x/3 + \sigma_1 \\ x/3 + \sigma_2 \\ b - 2x/3 + \sigma_3 \\ x/3 + \sigma_0 + \sigma_4 \end{pmatrix}$$

Complexity Improvements First, we consider the scenario with identical round keys from Section 3.3.1. Using the polynomial $\mathsf{first\_component}(B^{-1} \cdot \vec{o})$ of Equation (3.7) instead of the rightmost branch in the analysis of Sections 3.3 and 3.3.1 lowers the complexities involved. Steps 1 and 3 are unaffected by branch subtraction since the complexities do not depend on the number of rounds r. For step 2, i. e. interpolating the value of the second highest coefficient, analysis of the new computational complexity requires a little more care because alongside complexity improvements, we introduced some overhead. Since only the first component of vector $B^{-1} \cdot \vec{o}$ is needed, the overhead consists of t polynomial additions, each of which has run time in $\mathcal{O}(D)$ for polynomials of degree at most D [78]. The components of $\vec{o}$ for the regarded input format are of maximum degree d^{r-t+1}. The total overhead is thus in $\mathcal{O}\left(t d^{r-t+1}\right)$. In total, computational complexity

	Time	Space	Data
GCD	$\tilde{\mathcal{O}}\big((r-t)d^{r-t}\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$2d^{r-t}$
GCD (bs)	$\tilde{\mathcal{O}}\big((r-2t)d^{r-2t+2}+td^{r-t+1}\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$2d^{r-2t+2}$
Root	$\tilde{\mathcal{O}}\big((r-t)d^{r-t}+td^{t-1}\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$d^{r-t}+2$
Root (bs)	$\tilde{\mathcal{O}}\big((r-2t)d^{r-2t+2}+td^{r-t+1}\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$d^{r-2t+2}+2$
Multiple Keys (bs)	$\tilde{\mathcal{O}}\big(r(r-t)d^{r-t}+rtd^{r-t+1}\big)$	$\mathcal{O}(r)$	$\mathcal{O}\big((r-t)d^{r-t}\big)$

Table 3.1: Complexities of the low memory interpolation cryptanalysis for $\mathrm{UFN}_{\mathsf{erf}}[p,r,t]$ assuming $r>2t$, where $\tilde{\mathcal{O}}(\,\cdot\,)$ is ignoring logarithmic factors as defined in Section 2.1. The branch subtraction technique of Section 3.3.3 is abbreviated as "bs."

with branch subtraction is in $\tilde{\mathcal{O}}\big((r-2t)d^{r-2t+2}+td^{r-t+1}\big)$ as opposed to $\tilde{\mathcal{O}}\big((r-t)d^{r-t}\big)$ without, where $\tilde{\mathcal{O}}(\,\cdot\,)$ is ignoring logarithmic factors as defined in Section 2.1. For $r\geqslant 2t$, polynomial interpolation with branch subtraction has smaller computational complexity than without. Data complexity with branch subtraction is $d^{r-2t+2}+1$ as opposed to $d^{r-t}+1$ without. Space complexity stays in $\mathcal{O}(1)$ since the same low memory algorithm for recovery is being used.

When considering distinct round keys as in Section 3.3.2, the branch subtraction technique is applicable as well, lowering the degree of the underlying polynomial by d^{t-1}. As a result, the total computational complexity for this scenario is in $\mathcal{O}\big(r(r-t)d^{r-t}\log d\big)$. A summary of all the complexities with and without branch subtraction can be found in Section 3.3.4 and Table 3.1.

Note. Branch subtraction only works for $\mathrm{UFN}_{\mathsf{erf}}$ and not $\mathrm{UFN}_{\mathsf{crf}}$, since every σ_i is added to only one branch in a $\mathrm{UFN}_{\mathsf{crf}}$.

3.3.4 Summary of Complexities

In the sections above, we proposed a few approaches to recover keys in a $\mathrm{UFN}_{\mathsf{erf}}$. In the case of a single key $\bar{k}$, i. e. $\bar{K}=(g_0(\bar{k}),\dots,g_{r-1}(\bar{k}))$ for linear functions g_i, we applied an existing method using the GCD and pointed out a novel method using root finding. In the general case $\bar{K}=(\bar{k}_0,\dots,\bar{k}_{r-1})$, a slightly different GCD approach accounts for the differing sub keys. The time, space and data complexities of the different approaches are summarized in Table 3.1. In general, the interpolation step dominates both computational and data complexity.

The memory requirements are dominated by the second highest coefficient, which requires memory in $\mathcal{O}\big(d^{t-1}\big)$. If we were to use standard Lagrange interpolation for the interpolation step, its memory requirements would be in $\mathcal{O}(d^r)$. Under the sensible assumption $r\geqslant 2t$, which we further discuss in the next paragraph, the interpolation step would also dominate the memory complexity. Thus, in using the low memory interpolation technique from Section 2.5 we lower the memory requirements of the entire attack.

Recommended Minimum Number of Rounds A UFN has maximum possible security against interpolation attacks if the output branches when seen as polynomials have maximum achievable degree. In $\mathbb{F}_p$ this is $(p-1)$, as outlined in Section 2.1. The conclusion of our results is that for the presented single key attack scenarios, this is achieved if the UFN

	root	GCD
number of roots	1.89	—
algebraic coefficient	0.09	0.11
coefficient value	1 468.51	3 132.22
key recovery	0.81	0.04
total	1 469.40	3 132.36

Table 3.2: Observed average number of roots and running times in milliseconds for key recovery of $\mathrm{UFN}_{\mathsf{erf}}[p, 17, 4]$ using root finding and GCD. The degree of the interpolated polynomial was 3^{11}. ($n = 100$)

has a number of rounds $r \geqslant \lceil \log_d p \rceil + 2t - 2$. For the distinct key scenario, we conclude $r \geqslant \lceil \log_d p \rceil + 1$.

3.3.5 Experimental Verification

We validate our analysis by running small scale experiments. The UFN instances use randomized key, round constants, and coefficients of the round function. Since the analysis of Section 3.2 is for monic polynomials, the highest coefficient of the round function is always 1. The fix parameters of the experiments are $p = 99\,999\,989$, $r = 17$, $t = 4$, chosen because of hardware limitations. The round function is of degree 3. For these parameters p and t, we recommended a minimum number of rounds r of 23. Both proposed methods of key recovery are used, namely root finding and GCD. The branch subtraction technique of Section 3.3.3 is applied in order to lower the involved complexities. Given above parameters, the degree of the combined output polynomial for $\mathrm{UFN}_{\mathsf{erf}}$ is 3^{11}.

Note. As an example for realistic parameters, consider a 128-bit prime p, i. e. $\log_2 p \approx 128$, a round function of degree $d = 3$, and $t = 4$ branches. The recommended minimum number of rounds r is then 87.

The experiments are implemented in python using sagemath [76]. All random values are generated using python's built-in "random" module. Measurements were taken on a machine with standard Intel Core i5-6300U CPU and 7.22 GiB of RAM. Each experiment is run $n = 100$ times. The full code is given in Listings A.1 and A.2. A summary of the observed average running times can be found in Table 3.2.

The key recovery step with the root finding method takes about one order of magnitude longer than in the GCD approach, reflecting the theoretical results of Section 3.3.4. However, run time is dominated by the interpolation step, dwarfing recovery of the algebraic coefficient and subsequent key recovery by about four orders of magnitude. It is interesting to observe the average number of roots. Although theoretically, up to $3^{4-1} = 27$ roots could occur, the experiments show that in practice, this number is significantly lower, with an average of less than 2 roots.

3.4 Cryptanalysis of $\mathrm{UFN}_{\mathsf{crf}}$

In this section, we analyze UFNs in the CRF variant according to the steps outlined in Section 3.2. Since the analysis is quite similar to the ERF variant of section Section 3.3, we only point out significant differences. Notably, key recovery with only one round key $\bar{k}$, i. e. $\bar{K} = (g_0(\bar{k}), \ldots, g_{r-1}(\bar{k}))$, is not reiterated.

Algorithm 3: Second highest coefficient of leftmost branch in $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$ on input $(x, b, \ldots, b)$.

Input: r, t, f, branch constant b, round constants c_0, c_1
Output: polynomial $Q(K)$ for second highest coefficient of leftmost branch

1 $\beta := f\left((t-1)b + k_0 + c_0\right) + (t-2)b + k_1 + c_1$
2 **return** $d^{r-t-1}(a_{d-1} + d\beta)$

Algebraic Expression of Second Highest Coefficient

In a $\mathrm{UFN}_{\mathsf{crf}}[p, r, t]$, the polynomial representing the leftmost output branch has the lowest degree, as shown in Proposition 5. For this branch, as shown in Corollary 4, the second highest coefficient $Q(K)$ has form $d^{r-t-1}(a_{d-1} + d\beta)$, with $\beta = (t-2)b + f((t-1)b + k_0) + k_1$. This coefficient is simpler when compared to $\mathrm{UFN}_{\mathsf{erf}}$ as it depends only on k_0 and k_1. Consequently, computing $Q(K)$ is simpler, as described in Algorithm 3.

Complexity Calculating the algebraic form of the second highest coefficient requires a constant number of addition and multiplication of scalars. Thus, the complexity is in $\mathcal{O}(1)$.

Value of Second Highest Coefficient

Unlike for $\mathrm{UFN}_{\mathsf{erf}}$, the second highest coefficient of the *leftmost* branch is recovered for $\mathrm{UFN}_{\mathsf{crf}}$. Thus, evaluation points y_j for the interpolation are the values of the leftmost output branch and inputs $(\alpha^j, b, \ldots, b)$ are used for the low memory interpolation, where $\alpha \in \mathbb{F}_p$ is a primitive element as before. These changes allow application of Corollary 4. The degree of the polynomial is $D = d^{r-t}$. Summarizing, the inputs to Algorithm 1 are (1) degree D, (2) primitive element α, and (3) polynomial evaluation oracle $\mathcal{O}^{\mathsf{crf}}(x) = \mathsf{first\_component}\big(E_{\bar{K}}\big((x, b, \ldots, b)\big)\big)$. The result of the low memory interpolation is the value z.

Complexity None of the complexities change from those of $\mathrm{UFN}_{\mathsf{erf}}$.

Key Recovery with Multiple Round Keys

Consider $\mathrm{UFN}_{\mathsf{crf}}$ $E_{\bar{K}}$ with general $\bar{K} = (\bar{k}_0, \ldots, \bar{k}_{r-1})$ where $\bar{k}_i$ is key of round i. The strategy to recover the key is extremely similar to the ERF variant of Section 3.3.2. The algebraic form of the second highest coefficient $Q_i(K)$ is slightly different, as proved in Proposition 5. For a $\mathrm{UFN}_{\mathsf{crf}}$, $Q_i(K) = d^{r-1}(a_{d-1} + d\beta_i)$ where $\beta_i = (t-2)b_i + f((t-1)b_i + k_0) + k_1$. Combining the three equations $q_i(K) = z_i$ works the same way as before. Due to the different form of β_i, the equations change slightly:

$$(t-2)b_i + f((t-1)b_i + k_0) + k_1 - \frac{z_i}{d^r} + \frac{a_{d-1}}{d} = 0 \tag{3.8}$$

For $0 \leqslant i, j \leqslant 2$, subtraction of above equations results in:

$$\Delta_{(i,j)} := f((t-1)b_i + k_0) - \frac{z_i - z_j}{d^r} + \frac{2a_{d-1}}{d} = 0 \tag{3.9}$$

	Time	Space	Data
GCD	$\tilde{\mathcal{O}}\big((r-t)d^{r-t}\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$2d^{r-t}$
Root	$\tilde{\mathcal{O}}\big((r-t)d^{r-t}+d\big)$	$\mathcal{O}\big(d^{t-1}\big)$	$d^{r-t}+2$
Multiple Keys	$\tilde{\mathcal{O}}\big(r^2d^{r-1}\big)$	$\mathcal{O}(r)$	$\mathcal{O}\big((r-t)d^{r-t}\big)$

Table 3.3: Complexities of the low memory interpolation cryptanalysis for $\mathrm{UFN}_{\mathsf{crf}}[p,r,t]$ assuming $r>t$, where $\tilde{\mathcal{O}}(\,\cdot\,)$ is ignoring logarithmic factors as defined in Section 2.1. Note that branch subtraction is not applicable to $\mathrm{UFN}_{\mathsf{crf}}$.

As in the ERF case, the first two sub keys can be recovered.

$$\bar{k}_0 = k_0 - \gcd\left(\Delta_{(0,1)}, \Delta_{(0,2)}\right) \tag{3.10}$$

$$\bar{k}_1 = \frac{z_0}{d^{r-1}} - \frac{a_{d-1}}{d} - (t-2)b_i - f((t-1)b_0 + \bar{k}_0) \tag{3.11}$$

Again, the knowledge of $\bar{k}_0$ and $\bar{k}_1$ allows a partial decryption of any ciphertext. Thus, "peeling off" two rounds and iteratively applying the described method allows recovery of the entire key $\bar{K}$.

Complexity Although the form of $Q(k)$ and β are slightly different from those in $\mathrm{UFN}_{\mathsf{erf}}$, the steps are fundamentally the same as for the ERF variant. Thus, the complexities are also as in Section 3.3.2

Summary of Complexities Two scenarios with a total of three key recovery approaches are outlined for $\mathrm{UFN}_{\mathsf{crf}}$: For the single key scenario, the root finding method and the GCD approach are used, while we use a slightly altered GCD method for the distinct round key scenario. A summary of the time, space, and data complexities for all three approaches can be found in Table 3.3. As before, the interpolation dominates time and data complexity.

Note. The branch subtraction technique is not applicable to $\mathrm{UFN}_{\mathsf{crf}}$.

Recommended Minimum Number of Rounds Based on our analysis, a $\mathrm{UFN}_{\mathsf{crf}}$ with a single round key needs at least $r \geqslant \lceil \log_d p \rceil + t$ rounds in order to reach the maximum possible degree. When using distinct round keys, the minimum number of rounds to achieve the maximum degree possible is $r \geqslant \lceil \log_d p \rceil + 1$.

Experimental Verification

We performed small scale experiments for $\mathrm{UFN}_{\mathsf{crf}}$ to validate the different approaches. The parameters are the same as for $\mathrm{UFN}_{\mathsf{erf}}$ in Section 3.3.5, namely $p = 99\,999\,989$, $r = 17$, $t = 4$, with round function of degree 3. The degree of the polynomial that is to be interpolated is 3^{13}. For the parameters p and t we used, the recommended minimum number of rounds is 21. Run times of the experiments can be found in Table 3.4. The used code is replicated in Listings A.1 and A.2.

	root	GCD
number of roots	2.10	—
algebraic coefficient	0.03	0.05
coefficient value	11 832.86	23 482.13
key recovery	0.49	0.03
total	11 833.38	23 482.21

Table 3.4: Observed average number of roots and running times in milliseconds for key recovery of $\mathrm{UFN}_{\mathsf{crf}}[p, 17, 4]$ using root finding and GCD. The degree of the interpolated polynomial was 3^{13}. ($n = 100$)

	$\mathrm{GMiMC}_{\mathsf{erf}}[p, r, t]$	$\mathrm{GMiMC}_{\mathsf{crf}}[p, r, t]$
Low Memory Interpolation	$\lceil \log_3 p \rceil + t$	$\lceil \log_3 p \rceil + t$
Additional Branch Subtraction	$\lceil \log_3 p \rceil + 2t - 2$	—
GMiMC (interpolation) [3]	$\lceil \log_3 p \rceil + 4t - 3$	$\lceil \log_3 p \rceil + 2t$

Table 3.5: Recommended lower limits of r for security level $\log_2 p$ against different attacks for univariate GMiMC.

3.5 Application of the Analysis to GMiMC

Our analysis is mainly motivated by the Arithmetization Oriented Cipher (AOC) GMiMC, introduced in Section 2.2.1. The GMiMC family has two members that are based on UFN: $\mathrm{GMiMC}_{\mathsf{erf}}$ and $\mathrm{GMiMC}_{\mathsf{crf}}$. Independent of the variant, *univariate* and *multivariate* GMiMC, i. e. identical round key or distinct round key versions are proposed. The round function used in all variants is $f(x) = x^3$, i. e. $d = 3$. Our respective analyses of this chapter are directly applicable to the different versions.

Our analysis suggests that the minimum number rounds to thwart the attack vectors proposed herein are $r_{\mathsf{erf}} \geqslant \lceil \log_3 p \rceil + 2t - 2$ for $\mathrm{GMiMC}_{\mathsf{erf}}$, taking branch subtraction of Section 3.3.3 into account. For $\mathrm{GMiMC}_{\mathsf{crf}}$, branch subtraction is not applicable, resulting in $r_{\mathsf{crf}} \geqslant \lceil \log_3 p \rceil + t$. Our recommended minimum number of rounds are based on a more concrete foundation than originally given, providing detailed attack vectors to justify the required respective minima. A comparison of minimum recommended number of rounds between the different scenarios can be found in Table 3.5.

Note. Our analysis does not contradict the recommendations of the authors of GMiMC, who consider many different attack vectors. Reiterated in Table 3.5 is only the recommended minimum against *interpolation* attacks.

3.6 Correcting Block Attacks against UFN-Based Sponges

Motivated by the recently proposed GMiMCHash [3], we consider Sponge constructions instantiated by UFNs. Both variants ERF and CRF can be used, resulting in a hash function over $\mathbb{F}_p$, as described in Section 2.3.1. We re-use the root finding technique of Section 3.3.1 to describe potential attacks on these kinds of hash functions.

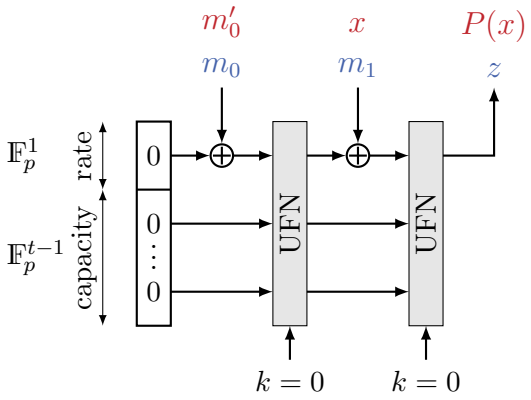


Figure 3.4: Sponge attack setup.

We analyze Sponge constructions of the following form. The rate is one field element of $\mathbb{F}_p$, i. e. $r = \log_2 p$, while all other branches of the UFN make up the capacity, i. e. $c = (t-1)\log_2 p$. The input to the Sponge construction, i. e. the message, is a list of field elements $\mathbb{F}_p^s$ for $s \geqslant 1$. The hash size is one field element, i. e. one squeeze is performed. A visualization of the Sponge construction can be found in Figure 2.6 in Section 2.3.1.

Note. As will become clear in the remainder of this section, our proposed attack vector crucially relies on the Sponge construction using only one squeeze phase. This assumption, while by no means universal, is justified by the hash being exactly *one* field element, as opposed to a tuple.

3.6.1 Attack Setup

The attack vector described in the following is, in some sense, fundamentally different from the rest of this chapter, while reusing some ideas and techniques. One of the reasons is that the key for a UFN when used in a Sponge construction is fixed, while the attacks of Sections 3.3 and 3.4 recover the unknown key. Generally, the attacks to the three security goals of a cryptographic hash function described below are all based on the same principle: We recover the hash value as a polynomial with the last message block as a variable, subtract the hash value of a different message, then calculate the roots. A figure of this high-level description can be found in Figure 3.4. Our described attack vector falls in the category of *correcting-last-block attacks* [66] outlined in Section 2.3, where the root is the correcting block.

For hash functions as opposed to block ciphers, no secret values like keys exist.[1] This opens up a new approach to compute the polynomial described above. One option, much in line with the previous sections, is interpolation between the last message block and the hash value. Since the full polynomial is required for our attack vector, low memory interpolation is not sufficient. The second approach to recover the polynomial is *symbolic evaluation*, which is not possible in the sections above. Since all parameters of the UFN are known, an attacker can choose the last message block as indeterminate x, then evaluate the UFN.

[1] *Keyed Hash Function* is synonymous with *Message Authentication Code* [43], a primitive we don't consider in this thesis.

Second Preimages Consider message $M = (m_0, m_1) \in \mathbb{F}_p^2$. Let f denote the fixed-key UFN used to instantiate the Sponge construction. The branch size of f in our attack setup is p. The number of branches t and rounds r do not change the attack vector but do influence its computational complexity. Let the rightmost branch of f correspond to the rate of the Sponge construction. For hash value z of message M, we find a second preimage in the following way:

1. Choose an arbitrary message block $m_0' \in \mathbb{F}_p$ with $m_0' \neq m_0$. Let $(h_0, \ldots, h_{t-1})$ denote the output branches of $f(m_0')$.
2. Compute the polynomial $P(x)$ corresponding to the rightmost output branch of f for input $(h_0, \ldots, h_{t-1} + x)$. Thus, $P(x)$ is the polynomial corresponding to the hash value of (m_0', x).
3. Find the roots of $P(x) - z$.

Any root m_1' of $P(x) - z$ gives a second preimage attack in form of a *correcting block*. More concretely, message $M' = (m_0', m_1')$ is a second preimage of M.

Preimages When the given value is not a message M but a hash value z, above steps can be executed to mount a preimage attack.

Collisions Finding a collision builds on the same principle with the following slight alterations.

1. Choose any two message blocks $m_0, m_0' \in \mathbb{F}_p$ with $m_0 \neq m_0'$.
2. Compute the polynomials $P(x)$ representing the hash value of a message of the form (m_0, x) and $P'(x)$ corresponding to the hash value of the message (m_0', x).
3. Compute the roots of $P(x) - P'(x)$.

Any root m_1 of $P(x) - P'(x)$ results in a collision. Namely, messages $M = (m_0, m_1)$ and $M' = (m_0', m_1)$ have the same hash value.

Complexity The complexity of finding all roots without multiplicities of a polynomial of degree D over $\mathbb{F}_p$ is in $\mathcal{O}\big(D \log^2 D \log(Dp) \log\log D\big)$. For $\mathrm{UFN}_{\mathsf{erf}}$, the degree of polynomial $P(x)$ after r rounds is d^{r-t}. Hence, the complexity of the root finding step is in

$$\mathcal{O}\big((r-t)^2 d^{r-t} \log^2 d \log((r-t)\log d)((r-t)\log d + \log p)\big)\,.$$

For the collision attack, the degree of the polynomial $P(x) - P'(x)$ is d^{r-t-1} since the terms of highest degree are the same in both $P(x)$ and $P'(x)$. The minimum number of rounds required to be secure against the proposed attack vector is $r \geqslant \lceil \log_d p \rceil + t$.

For $\mathrm{UFN}_{\mathsf{crf}}$, the degree of polynomial $P(x)$ after r rounds is d^r, leading to a recommended minimum number of rounds of $r \geqslant \lceil \log_d p \rceil$. Choosing the rate to be the leftmost branch lowers the degree to d^{r-t+1}, raising the recommended minimum number of rounds to $r \geqslant \lceil \log_d p \rceil + t - 1$.

Note. Because the full, unaltered polynomial of the rightmost output branch is required for the proposed attack vector, the branch subtraction technique of Section 3.3.3 cannot be applied advantageously.

Note. Adding more squeeze rounds and setting the hash value to the last derived field element z_n, discarding all other elements, does not conceptually protect against our proposed attack vector.

GMiMC$_{\mathsf{erf}}$Hash The minimum number of rounds proposed for any instantiation of GMiMC$_{\mathsf{erf}}$Hash is $r = \log_3 p + 4t - 3$ with $d = 3$. The proposal does not explicitly justify the given recommended minimum number of rounds by a security analysis of the hash function. Our analysis shows that GMiMC$_{\mathsf{erf}}$Hash is secure against the proposed attack vector. Moreover, this algebraic analysis justifies the proposed number of rounds in GMiMC$_{\mathsf{erf}}$Hash.

Note. While the GMiMC proposal does not exclude constructions like GMiMC$_{\mathsf{crf}}$Hash, the ERF variant is explicitly chosen for efficiency reasons.

3.6.2 Experimental Verification

The results of Section 3.6.1 are validated by running small scale experiments. The Sponge construction is instantiated with GMiMC$_{\mathsf{erf}}[p, r, t]$ with $p = 99\,999\,989$, number of rounds between $24 \leqslant r \leqslant 29$, number of branches between $3 \leqslant t \leqslant 6$, and fix key $k = 0$. Two different sets of experiments are run: Finding (1) second preimages and (2) collisions. The round constants are randomly chosen and fixed across all experiments. For each combination of (r, t) in the given intervals, 1 000 experiments are performed. The messages are re-randomized for every experiment. All the measurements were taken on a machine with a standard Intel Core i5-6300U CPU and 7.22 GiB of RAM. The experiments are implemented in python using sagemath. To generate the random values, python's "random" module is used. The code for the experiments can be found in Listing A.3.

Second Preimages For the experiments on second preimages, across all 24 000 experiments a total of 8 743 iterations with no second preimage are observed. These experiments are considered failed. This puts the success probability of finding at least one preimage to 63.6%. Of secondary interest is the average number of second preimages found given that the attack is successful, i. e. at least one second preimage is found. Over all the 15 257 successful experiments, an average of 1.58 second preimages are observed.

Collisions For the experiments on collisions, no collision is found in 8 842 of the 24 000 experiments. These are considered failed. The success probability is thus 63.2%. Again of secondary interest is the average number of collisions found in the successful experiments, i. e. at least one collision is found. Over all 15 158 successful experiments, an average of 1.58 collisions are observed.

Note. The failure rates of 36.4% and 36.8% respectively are supported by the fact that for the chosen parameters, about 36.8% of all possible polynomials do not have a root in $\mathbb{F}_p$ [53]. This is further elaborated on below.

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0003	0.0003	0.0003
root finding	0.0005	0.0005	0.0005
total	0.0009	0.0009	0.0009

(a) $t = 3$

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0019	0.0019	0.0019
root finding	0.0028	0.0029	0.0032
total	0.0049	0.0049	0.0052

(b) $t = 4$

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0238	0.0249	0.0237
root finding	0.0279	0.0296	0.0281
total	0.0519	0.0547	0.0520

(c) $t = 5$

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.3144	0.3239	0.3155
root finding	0.3243	0.3288	0.3229
total	0.6407	0.6548	0.6404

(d) $t = 6$

Table 3.6: Observed average running times in milliseconds for finding second preimages of GMiMC$_{\mathsf{erf}}$Hash. ($n = 1\,000$ per column)

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0007	0.0006	0.0006
root finding	0.0005	0.0005	0.0005
total	0.0013	0.0012	0.0011

(a) $t = 3$

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0038	0.0038	0.0038
root finding	0.0029	0.0029	0.0028
total	0.0067	0.0067	0.0067

(b) $t = 4$

	$r = 27$	$r = 28$	$r = 29$
construct poly	0.0485	0.0469	0.0478
root finding	0.0292	0.0271	0.0281
total	0.0778	0.0741	0.0760

(c) $t = 5$

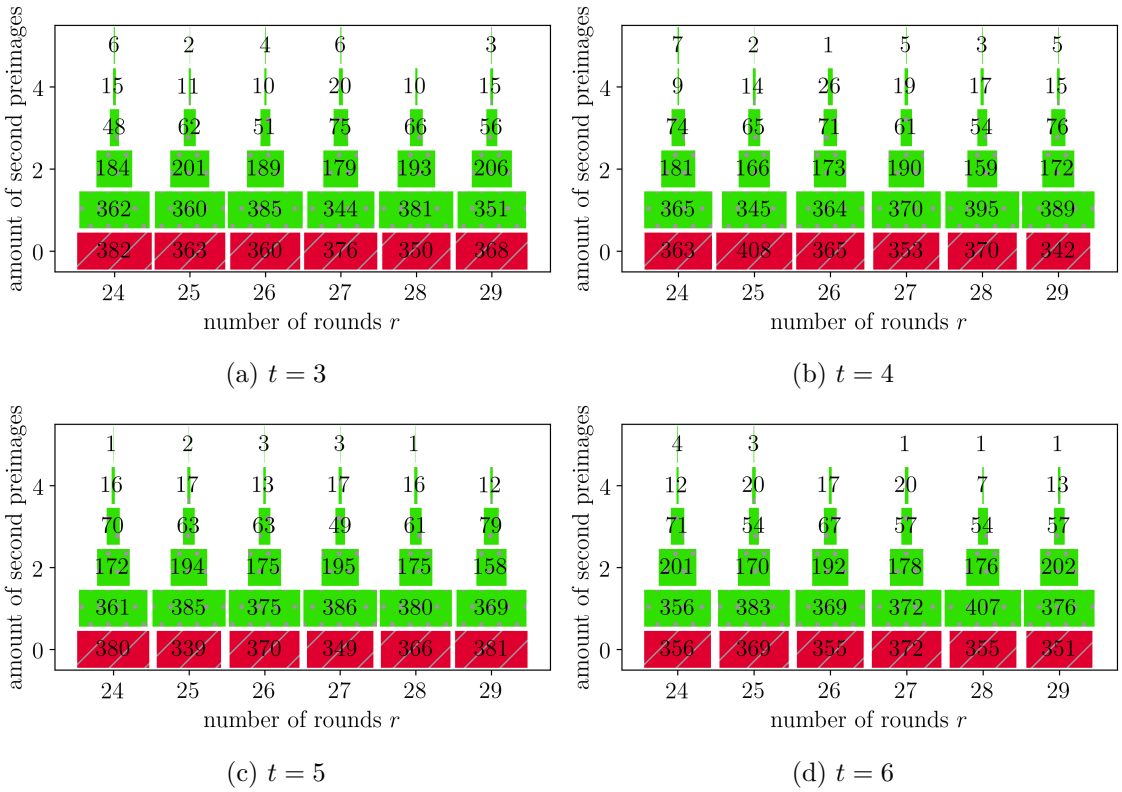
	$r = 27$	$r = 28$	$r = 29$
construct poly	0.6359	0.6036	0.6048
root finding	0.3211	0.3020	0.3006
total	0.9571	0.9058	0.9056

(d) $t = 6$

Table 3.7: Observed average running times in milliseconds for collision finding of GMiMC$_{\mathsf{erf}}$Hash. ($n = 1\,000$ per column)

Run Times In Tables 3.6 and 3.7, the running times for the experiments with $24 \leqslant r \leqslant 29$ and $3 \leqslant t \leqslant 6$ are reported. The running times for symbolic evaluation of the UFN and for root finding are reported alongside the total running times.

Success Probabilities In Figure 3.5, we plot the number of second preimages found in the experiments. Similarly, Figure 3.6 visualizes the number of collisions found. In the subfigures, different numbers of branches t are depicted. Each subfigure shows, for different numbers of rounds r on the x-axis, the number of additional preimages or collisions on the y-axis found over the 1 000 randomized experiments. For example, when regarding $t = 3$ branches in Figure 3.5a, for the GMiMCHash instance instantiated with GMiMC$_{\mathsf{erf}}$ with $r = 28$ rounds, there are 193 of the 1 000 experiments in which 2 preimages were found, and 66 in which 3 preimages were found. A red (striped) bar signifies that no root was found, while the green (dotted) bars indicate a successful attack.



(a) $t = 3$ (b) $t = 4$

(c) $t = 5$ (d) $t = 6$

Figure 3.5: Number of second preimages found in GMiMCHash using $\mathrm{GMiMC}_{\mathsf{erf}}[p, r, t]$ for various numbers of rounds. ($n = 1\,000$ per given (r, t))

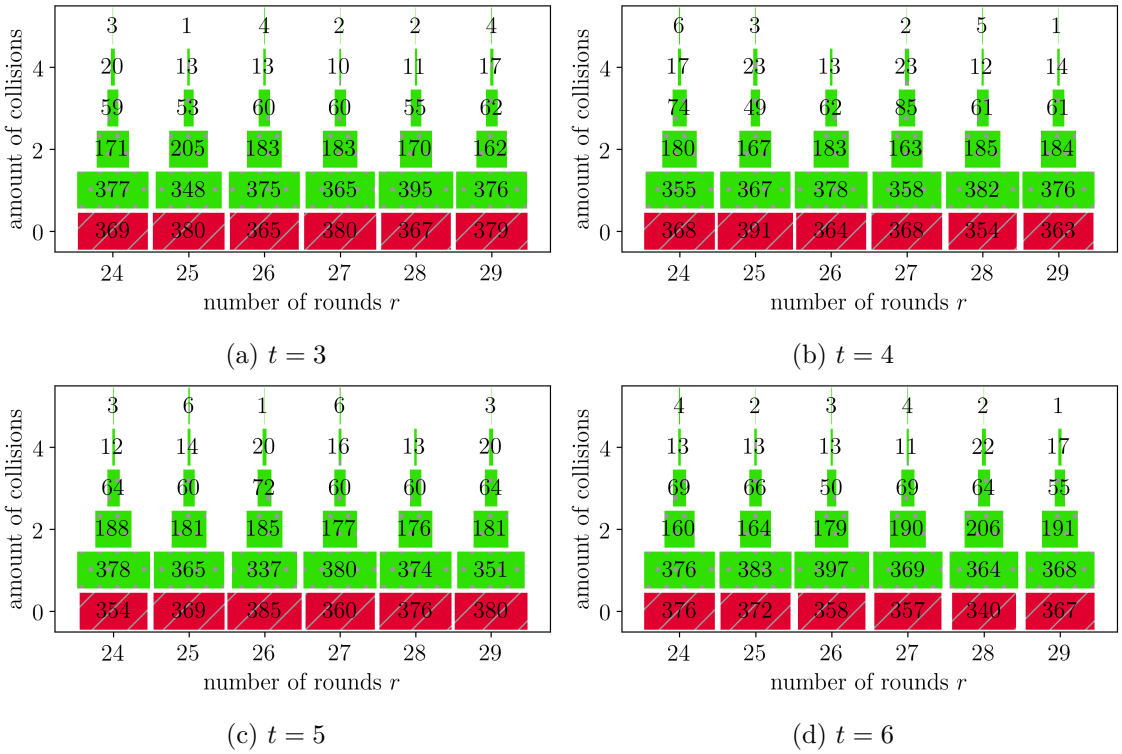


Figure 3.6: Number of collisions found in GMiMCHash using $\text{GMiMC}_{\mathsf{erf}}[p, r, t]$ for various numbers of rounds. ($n = 1\,000$ per given (r, t))

Roots of Random Polynomials over Finite Fields

In order to validate the failure rate of Section 3.6.2, we calculate the probability that a random polynomial of degree d has no roots in a specific finite field. A formula t_d for the number of polynomials of degree d over finite field $\mathbb{F}_q$ that have no root is given in lemma 1 in [53] and reproduced in Equation (3.12).

$$t_d = \sum_{i=0}^{d} (-1)^i \binom{q}{i} q^{d-i} \tag{3.12}$$

The total number of polynomials of degree d over $\mathbb{F}_q$ is q^d. The experiment's parameters are $q = 99\,999\,989$ and $d = 3^i$ with $24 \leqslant i \leqslant 29$. This results in a probability of a random polynomial not having any root in $\mathbb{F}_q$ of $t_d/q^d \approx 36.8\%$ for any i in the given interval, supporting the experimental results.

Note. Investigating whether the polynomials $P(x)$ of Section 3.6.2 are, in fact, randomly and uniformly distributed over the set of polynomials of degree $\mathsf{deg}(P(x))$ over $\mathbb{F}_p$ is beyond the scope of this thesis. For the sake of the validation in this section, we conjecture a distribution "pseudo-random close to uniform" and consider it a close enough approximation.

4. Conclusion

The goal of this thesis was to apply interpolation attacks with low memory complexity to Unbalanced Feistel Networks (UFNs), the design principle of recent Arithmetization Oriented Cipher (AOC) Generalized MiMC (GMiMC). This was motivated by (1) the recently left open question about the generalization of an analysis to Feistel-MiMC [54] and (2) the need for more cryptanalysis on AOCs in order to firmly demonstrate their security.

We applied an existing variant of Lagrange interpolation as the central step for key recovery in multiple scenarios: UFN in variant Expanding Round Function (ERF) in Section 3.3 and Contracting Round Function (CRF) in Section 3.4, each with both identical and distinct round keys. For identical round keys, two approaches were considered, namely a GCD method introduced in prior work [54] and a novel root finding technique. For distinct round keys, a slightly modified GCD approach was used to recover the key. This positively answered the open question mentioned above. Complexity improvements by combining multiple output branches were demonstrated for all scenarios and key recovery approaches of the ERF variant. Based on these, we gave recommendations on the minimum number of rounds required to thwart the considered attacks. Small-scale experiments demonstrated our proposals and supplemented the theoretical run time analysis with measurements.

We applied some of the developed techniques to Sponge constructions with one Squeeze phase and instantiated with UFNs in Section 3.6. A correcting-last-block attack to find collisions, second preimages, and preimages was proposed. Small-scale experiments gave run times for various sets of parameters and granted insight about the attack's success probabilities. Parts of this thesis have been submitted for publication [7].

Future Work Some questions are left open for future work: The applicability of low memory interpolation attacks to other AOCs like the Hades framework or the MARVELlous suite should be determined. An extension of the analysis to UFNs with general, non-monic polynomials as round function might be of further theoretical interest, although to the best of our knowledge, no currently proposed AOC uses such a design paradigm. Algebraically analyzing Sponge constructions with more than one Squeeze phase is another question we leave open.

A. Code

In this appendix the code for this thesis's different experiments on Unbalanced Feistel Networks (UFNs) is listed. It is written in python and using the computer algebra system sagemath [76].

In Appendix A.1, we give general definitions for UFNs. We used the code listed in Appendix A.2 for the key recovery experiments of Sections 3.3.5 and 3.4. Appendix A.3 contains the code for the correcting-last-block attack on Sponge constructions from Section 3.6.

A.1 Unbalanced Feistel Networks

The functions in Listing A.1 provide general definitions of $\mathrm{UFN}_{\mathsf{erf}}$ and $\mathrm{UFN}_{\mathsf{crf}}$ using both identical and distinct round keys for use in Listings A.2 and A.3.

Listing A.1: Functions for UFNs.

```
# -*- coding: utf-8 -*-
import random

def ufn_erf(roundfunction, cleartext, key, round_constants):
    ciphertext = cleartext
    for r in range(len(round_constants)):
        sigma = roundfunction(ciphertext[0] + (key*(r+1)) + round_constants[r])
        temp = ciphertext[0]
        for b in range(len(ciphertext) - 1):
            ciphertext[b] = ciphertext[b+1] + sigma
        ciphertext[-1] = temp
    return ciphertext

def ufn_erf_multikey(roundfunction, cleartext, multikey, round_constants):
    ciphertext = cleartext
    for r in range(len(round_constants)):
        sigma = roundfunction(ciphertext[0] + multikey[r] + round_constants[r])
        temp = ciphertext[0]
        for b in range(len(ciphertext) - 1):
            ciphertext[b] = ciphertext[b+1] + sigma
```

```
21          ciphertext[-1] = temp
22      return ciphertext
23
24  def ufn_crf(roundfunction, cleartext, key, round_constants):
25      ciphertext = cleartext
26      for r in range(len(round_constants)):
27          sigma = roundfunction(sum(ciphertext[1:]) + (key*(r+1)) +
        round_constants[r])
28          ciphertext = ciphertext[1:] + [ciphertext[0] + sigma]
29      return ciphertext
30
31  def ufn_crf_multikey(roundfunction, cleartext, multikey, round_constants):
32      ciphertext = cleartext
33      for r in range(len(round_constants)):
34          sigma = roundfunction(sum(ciphertext[1:]) + multikey[r] +
        round_constants[r])
35          ciphertext = ciphertext[1:] + [ciphertext[0] + sigma]
36      return ciphertext
37
38  def randomize_variables(field, num_rounds, num_branches, f_degree):
39      field_size = len(field)
40      round_constants = []
41      branch_constants = []
42      branch_constants_2 = []
43      branch_constants_3 = []
44      f_coefficients = []
45      key = field(random.randint(0, field_size))
46      multikey = []
47      for _ in range(num_rounds):
48          round_constants += [field(random.randint(0, field_size))]
49          multikey += [field(random.randint(0, field_size))]
50      for _ in range(num_branches):
51          branch_constants += [field(random.randint(0, field_size))]
52          branch_constants_2 += [field(random.randint(0, field_size))]
53          branch_constants_3 += [field(random.randint(0, field_size))]
54      for _ in range(f_degree):
55          f_coefficients += [field(random.randint(0, field_size))]
56      f_coefficients += [field(1)]
57      f = lambda x : sum([f_coefficients[i]*x**i for i in range(len(
        f_coefficients))])
58      return round_constants, branch_constants, branch_constants_2,
        branch_constants_3, f_coefficients, f, key, multikey
```

A.2 Key Recovery of UFNs

Listing A.2 provides functions to measure run times of the interpolation attacks described in Sections 3.3.5 and 3.4. Example results can be found in Tables 3.2 and 3.4.

Listing A.2: Code for timing of key recovery attacks on UFNs.

```
1  # -*- coding: utf-8 -*-
2  import operator, random
3  from time import clock
4  from datetime import datetime
5  from unbalanced_feistel_networks import *
6
7  field_size = 99999989
8  num_rounds = 17
```

```
num_branches = 4
f_degree = 3

field = GF(field_size)

b_inverse_row_i = lambda i : [1]*(i) + [-num_branches+2] + [1]*(num_branches-i-1)
b_inverse = matrix(field, [b_inverse_row_i(i) for i in range(num_branches)])/(num_branches-1)

round_constants = branch_constants = branch_constants_2 = f_coefficients = f = key = 0

def secondsToStr(t):
    return "%d:%02d:%02d.%03d" % reduce(lambda ll,b : divmod(ll[0],b) + ll[1:], [(t*1000,),1000,60,60])

def algo_one(degree, primitive_element, oracle):
    oracle_time = 0
    z = 0
    s = -sum([primitive_element**j for j in range(degree+1)])
    a = reduce(operator.mul, [1-primitive_element**j for j in range(1, degree+1)], 1)
    b = 1
    for i in range(degree+1):
        t = clock()
        o = oracle(b)
        oracle_time += clock() - t
        z += o * (s+b)/a
        if i < degree: # i == degree means b == primitive_element**degree leading to division by 0
            a *= primitive_element**degree * (b - primitive_element**(-1)) / (b - primitive_element**degree)
            b *= primitive_element
    return z, oracle_time

def symbolic_second_coeff_erf(symbol, round_function, round_constants, branch_constants):
    assert(len(round_constants) >= len(branch_constants))
    num_branches = len(branch_constants)
    sigmas = 0
    for i in range(num_branches-1):
        sigmas += round_function(sigmas + (symbol*(i+1)) + round_constants[i] + branch_constants[i])
    return sigmas + (symbol*num_branches) + round_constants[num_branches-1]

def symbolic_second_coeff_crf(symbol, round_function, round_constants, branch_constants):
    beta = round_function(sum(branch_constants[1:]) + symbol + round_constants[0])
    beta += sum(branch_constants[2:]) + 2*symbol + round_constants[1] # anti-Xavier-fix
    return beta

def symbolic_second_coeff_erf_multikey(multisymbol, round_function, round_constants, branch_constants, cleartext_pos):
    sigmas = 0
    for i in range(cleartext_pos):
```

```
54          sigmas += round_function(sigmas + multisymbol[i] + round_constants[i] +
        branch_constants[i])
55      return sigmas + multisymbol[cleartext_pos] + round_constants[cleartext_pos]
56
57  def branch_substract_oracle(cleartext, key, round_constants, branch_constants):
58      output_polys = matrix(field, ufn_erf(f, branch_constants[:num_branches-1] +
        [cleartext], key, round_constants))
59      return (output_polys * b_inverse)[0][0]
60
61  def branch_substract_oracle_multikey(cleartext, cleartext_pos, multikey,
        round_constants, branch_constants):
62      cleartext = branch_constants[:cleartext_pos] + [cleartext] +
        branch_constants[cleartext_pos + 1:]
63      output_polys = matrix(field, ufn_erf_multikey(f, cleartext, multikey,
        round_constants))
64      return (output_polys * b_inverse)[0][0]
65
66  def timing_interpolate_root_erf():
67      k = polygen(field, 'k')
68      assert num_rounds - 2*num_branches + 2 > 0
69      output_degree = f_degree**(num_rounds - 2*num_branches + 2)
70      lowest_branch = lambda x : branch_substract_oracle(x, key, round_constants,
        branch_constants)
71
72      time_symbolic = clock()
73      beta = symbolic_second_coeff_erf(k, f, round_constants, branch_constants)
74      q_k = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta)
75      time_symbolic = clock() - time_symbolic
76
77      time_numeric = clock()
78      z, oracle_time = algo_one(output_degree, field.primitive_element(),
        lowest_branch)
79      time_numeric = clock() - time_numeric - oracle_time
80
81      time_root = clock()
82      key_candidates = (q_k - z).roots()
83      time_root = clock() - time_root
84
85      time_key_search = clock()
86      # just for performance measuring, I don't care about the result
87      for (candidate, _) in key_candidates:
88          ufn_erf(f, [0]*num_branches, candidate, round_constants)
89      success = (key in [x[0] for x in key_candidates])
90      time_key_search = clock() - time_key_search
91
92      return success, len(key_candidates), time_symbolic, time_numeric, time_root
        , time_key_search
93
94  def timing_interpolate_root_crf():
95      k = polygen(field, 'k')
96      assert num_rounds - num_branches > 0
97      output_degree = f_degree**(num_rounds - num_branches)
98      lowest_branch = lambda x : ufn_crf(f, [x] + branch_constants[1:], key,
        round_constants)[0]
99
100     time_symbolic = clock()
101     beta = symbolic_second_coeff_crf(k, f, round_constants, branch_constants)
102     q_k = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta)
```

```
103     time_symbolic = clock() - time_symbolic
104
105     time_numeric = clock()
106     z, oracle_time = algo_one(output_degree, field.primitive_element(),
        lowest_branch)
107     time_numeric = clock() - time_numeric - oracle_time
108
109     time_root = clock()
110     key_candidates = (q_k - z).roots()
111     time_root = clock() - time_root
112
113     time_key_search = clock()
114     # just for performance measuring, I don't care about the result
115     for (candidate, _) in key_candidates:
116         ufn_erf(f, [0]*num_branches, candidate, round_constants)
117     success = (key in [x[0] for x in key_candidates])
118     time_key_search = clock() - time_key_search
119
120     return success, len(key_candidates), time_symbolic, time_numeric, time_root
        , time_key_search
121
122 def timing_interpolate_gcd_erf():
123     k = polygen(field, 'k')
124     assert num_rounds - 2*num_branches + 2 > 0
125     output_degree = f_degree**(num_rounds - 2*num_branches + 2)
126     lowest_branch = lambda x : branch_substract_oracle(x, key, round_constants,
         branch_constants)
127     lowest_branch_2 = lambda x : branch_substract_oracle(x, key,
        round_constants, branch_constants_2)
128
129     time_symbolic = clock()
130     beta = symbolic_second_coeff_erf(k, f, round_constants, branch_constants)
131     beta_2 = symbolic_second_coeff_erf(k, f, round_constants,
        branch_constants_2)
132     q_k = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta)
133     q_k_2 = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta_2
        )
134     time_symbolic = clock() - time_symbolic
135
136     time_numeric = clock()
137     z, oracle_time = algo_one(output_degree, field.primitive_element(),
        lowest_branch)
138     z_2, oracle_time_2 = algo_one(output_degree, field.primitive_element(),
        lowest_branch_2)
139     time_numeric = clock() - time_numeric - oracle_time - oracle_time_2
140
141     time_gcd = clock()
142     g = k - gcd(q_k - z, q_k_2 - z_2)
143     time_gcd = clock() - time_gcd
144
145     success = (g == key)
146     return success, time_symbolic, time_numeric, time_gcd
147
148 def timing_interpolate_gcd_crf():
149     k = polygen(field, 'k')
150     assert num_rounds - num_branches > 0
151     output_degree = f_degree**(num_rounds - num_branches)
```

```
152     lowest_branch = lambda x : ufn_crf(f, [x] + branch_constants[1:], key,
        round_constants)[0]
153     lowest_branch_2 = lambda x : ufn_crf(f, [x] + branch_constants_2[1:], key,
        round_constants)[0]
154
155     time_symbolic = clock()
156     beta = symbolic_second_coeff_crf(k, f, round_constants, branch_constants)
157     beta_2 = symbolic_second_coeff_crf(k, f, round_constants,
        branch_constants_2)
158     q_k = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta)
159     q_k_2 = (output_degree // f_degree) * (f_coefficients[-2] + f_degree*beta_2
        )
160     time_symbolic = clock() - time_symbolic
161
162     time_numeric = clock()
163     z, oracle_time = algo_one(output_degree, field.primitive_element(),
        lowest_branch)
164     z_2, oracle_time_2 = algo_one(output_degree, field.primitive_element(),
        lowest_branch_2)
165     time_numeric = clock() - time_numeric - oracle_time - oracle_time_2
166
167     time_gcd = clock()
168     g = k - gcd(q_k - z, q_k_2 - z_2)
169     time_gcd = clock() - time_gcd
170
171     success = (g == key)
172     return success, time_symbolic, time_numeric, time_gcd
173
174 def timing_interpolate_multikey_erf():
175     assert num_rounds - num_branches > 0
176     recovered_key = []
177     time_symbolic = time_numeric = time_gcd = 0
178     k_0, k_1 = polygen(field, 'k_0, k_1')
179     last_position = []
180     if num_branches % 2 != 0:
181         last_position = [num_branches-1]
182     for cleartext_pos in list(range(1,num_branches,2)) + last_position:
183         output_degree = f_degree**(num_rounds - num_branches - cleartext_pos +
        1)
184         lowest_branch = lambda x : branch_substract_oracle_multikey(x,
        cleartext_pos, multikey, round_constants, branch_constants)
185         lowest_branch_2 = lambda x : branch_substract_oracle_multikey(x,
        cleartext_pos, multikey, round_constants, branch_constants_2)
186         lowest_branch_3 = lambda x : branch_substract_oracle_multikey(x,
        cleartext_pos, multikey, round_constants, branch_constants_3)
187
188         time = clock()
189         if cleartext_pos in last_position:
190             multisymbol = recovered_key + [k_1]
191         else:
192             multisymbol = recovered_key + [k_0, k_1]
193         beta = symbolic_second_coeff_erf_multikey(multisymbol, f,
        round_constants, branch_constants, cleartext_pos)
194         beta_2 = symbolic_second_coeff_erf_multikey(multisymbol, f,
        round_constants, branch_constants_2, cleartext_pos)
195         beta_3 = symbolic_second_coeff_erf_multikey(multisymbol, f,
        round_constants, branch_constants_3, cleartext_pos)
196         time_symbolic += clock() - time
```

```
197
198         time = clock()
199         z, oracle_time = algo_one(output_degree, field.primitive_element(),
    lowest_branch)
200         z_2, oracle_time_2 = algo_one(output_degree, field.primitive_element(),
     lowest_branch_2)
201         z_3, oracle_time_3 = algo_one(output_degree, field.primitive_element(),
     lowest_branch_3)
202         z = (z / (output_degree // f_degree) - f_coefficients[-2]) / f_degree #
     Remove all known, fixed parts except beta
203         z_2 = (z_2 / (output_degree // f_degree) - f_coefficients[-2]) /
    f_degree
204         z_3 = (z_3 / (output_degree // f_degree) - f_coefficients[-2]) /
    f_degree
205         time_numeric += clock() - time - oracle_time - oracle_time_2 -
    oracle_time_3
206
207         time = clock()
208         subtr_1 = beta - beta_2 - z + z_2
209         subtr_2 = beta - beta_3 - z + z_3
210         key_0 = k_0 - gcd(subtr_1, subtr_2)
211         key_1 = z - beta(k_0 = key_0) + k_1
212         if cleartext_pos in last_position:
213             recovered_key += [key_1]
214         else:
215             recovered_key += [key_0, key_1]
216         time_gcd += clock() - time
217
218     success = (recovered_key == multikey[:num_branches])
219     return success, time_symbolic, time_numeric, time_gcd
220
221 if __name__ == "__main__":
222     num_runs = 100
223     success_root_erf = num_roots_erf = time_symbolic_root_erf =
    time_numeric_root_erf = time_root_erf = time_key_search_erf = 0
224     success_root_crf = num_roots_crf = time_symbolic_root_crf =
    time_numeric_root_crf = time_root_crf = time_key_search_crf = 0
225     success_gcd_erf = time_symbolic_gcd_erf = time_numeric_gcd_erf =
    time_gcd_erf = 0
226     success_gcd_crf = time_symbolic_gcd_crf = time_numeric_gcd_crf =
    time_gcd_crf = 0
227     success_mult_erf = time_symbolic_mult_erf = time_numeric_mult_erf =
    time_gcd_mult_erf = 0
228
229     for i in range(num_runs):
230         print("{:>.19} - Starting run {}".format(str(datetime.now()), i))
231         round_constants, branch_constants, branch_constants_2,
    branch_constants_3, f_coefficients, f, key, multikey = randomize_variables(
    field, num_rounds, num_branches, f_degree)
232
233         ret_root_erf = timing_interpolate_root_erf()
234         success_root_erf += ret_root_erf[0]
235         num_roots_erf += ret_root_erf[1]
236         time_symbolic_root_erf += ret_root_erf[2]
237         time_numeric_root_erf += ret_root_erf[3]
238         time_root_erf += ret_root_erf[4]
239         time_key_search_erf += ret_root_erf[5]
240
```

```
241         ret_root_crf = timing_interpolate_root_crf()
242         success_root_crf += ret_root_crf[0]
243         num_roots_crf += ret_root_crf[1]
244         time_symbolic_root_crf += ret_root_crf[2]
245         time_numeric_root_crf += ret_root_crf[3]
246         time_root_crf += ret_root_crf[4]
247         time_key_search_crf += ret_root_crf[5]
248 
249         ret_gcd_erf = timing_interpolate_gcd_erf()
250         success_gcd_erf += ret_gcd_erf[0]
251         time_symbolic_gcd_erf += ret_gcd_erf[1]
252         time_numeric_gcd_erf += ret_gcd_erf[2]
253         time_gcd_erf += ret_gcd_erf[3]
254 
255         ret_gcd_crf = timing_interpolate_gcd_crf()
256         success_gcd_crf += ret_gcd_crf[0]
257         time_symbolic_gcd_crf += ret_gcd_crf[1]
258         time_numeric_gcd_crf += ret_gcd_crf[2]
259         time_gcd_crf += ret_gcd_crf[3]
260 
261         ret_mult_erf = timing_interpolate_multikey_erf()
262         success_mult_erf += ret_mult_erf[0]
263         time_symbolic_mult_erf += ret_mult_erf[1]
264         time_numeric_mult_erf += ret_mult_erf[2]
265         time_gcd_mult_erf += ret_mult_erf[3]
266 
267 
268     print("Field␣size:␣␣␣{}".format(field_size))
269     print("Num␣rounds:␣␣␣{}".format(num_rounds))
270     print("Num␣branches:␣{}".format(num_branches))
271     print("Degree␣round:␣{}".format(f_degree))
272     print("Average␣over␣␣{}␣randomized␣runs".format(num_runs))
273     print("")
274     print("--␣ERF␣Root␣--")
275     print("Polynomial␣degree:␣␣␣␣␣␣␣{}".format(f_degree**(num_rounds - 2*
        num_branches + 2)))
276     print("success␣rate:␣␣␣␣␣␣␣␣␣␣␣{}".format(success_root_erf/num_runs))
277     print("avg␣num␣roots:␣␣␣␣␣␣␣␣␣␣{}".format(num_roots_erf/num_runs))
278     print("avg␣time␣symbolic␣coeff:␣{}".format(time_symbolic_root_erf/num_runs)
        )
279     print("avg␣time␣numeric␣coeff:␣␣{}".format(time_numeric_root_erf/num_runs))
280     print("avg␣time␣root␣finding:␣␣␣{}".format(time_root_erf/num_runs))
281     print("avg␣time␣key␣search:␣␣␣␣␣{}".format(time_key_search_erf/num_runs))
282     print("avg␣total␣running␣time:␣␣{}".format((time_symbolic_root_erf +
        time_numeric_root_erf + time_root_erf + time_key_search_erf)/num_runs))
283     print("")
284     print("--␣CRF␣Root␣--")
285     print("Polynomial␣degree:␣␣␣␣␣␣␣{}".format(f_degree**(num_rounds -
        num_branches)))
286     print("success␣rate:␣␣␣␣␣␣␣␣␣␣␣{}".format(success_root_crf/num_runs))
287     print("avg␣num␣roots:␣␣␣␣␣␣␣␣␣␣{}".format(num_roots_crf/num_runs))
288     print("avg␣time␣symbolic␣coeff:␣{}".format(time_symbolic_root_crf/num_runs)
        )
289     print("avg␣time␣numeric␣coeff:␣␣{}".format(time_numeric_root_crf/num_runs))
290     print("avg␣time␣root␣finding:␣␣␣{}".format(time_root_crf/num_runs))
291     print("avg␣time␣key␣search:␣␣␣␣␣{}".format(time_key_search_crf/num_runs))
292     print("avg␣total␣running␣time:␣␣{}".format((time_symbolic_root_crf +
        time_numeric_root_crf + time_root_crf + time_key_search_crf)/num_runs))
```

```
293     print("")
294     print("--␣ERF␣GCD␣--")
295     print("Polynomial␣degree:␣␣␣␣␣␣␣{}".format(f_degree**(num_rounds - 2*
        num_branches + 2)))
296     print("success␣rate:␣␣␣␣␣␣␣␣␣␣␣␣{}".format(success_gcd_erf/num_runs))
297     print("avg␣time␣symbolic␣coeff:␣{}".format(time_symbolic_gcd_erf/num_runs))
298     print("avg␣time␣numeric␣coeff:␣␣{}".format(time_numeric_gcd_erf/num_runs))
299     print("avg␣time␣gcd␣finding:␣␣␣␣{}".format(time_gcd_erf/num_runs))
300     print("avg␣total␣running␣time:␣␣{}".format((time_symbolic_gcd_erf +
        time_numeric_gcd_erf + time_gcd_erf)/num_runs))
301     print("")
302     print("--␣CRF␣GCD␣--")
303     print("Polynomial␣degree:␣␣␣␣␣␣␣{}".format(f_degree**(num_rounds -
        num_branches)))
304     print("success␣rate:␣␣␣␣␣␣␣␣␣␣␣␣{}".format(success_gcd_crf/num_runs))
305     print("avg␣time␣symbolic␣coeff:␣{}".format(time_symbolic_gcd_crf/num_runs))
306     print("avg␣time␣numeric␣coeff:␣␣{}".format(time_numeric_gcd_crf/num_runs))
307     print("avg␣time␣gcd␣finding:␣␣␣␣{}".format(time_gcd_crf/num_runs))
308     print("avg␣total␣running␣time:␣␣{}".format((time_symbolic_gcd_crf +
        time_numeric_gcd_crf + time_gcd_crf)/num_runs))
309     print("")
310     print("--␣ERF␣MULTI␣--")
311     print("Max␣Polynomial␣degree:␣␣␣{}".format(f_degree**(num_rounds -
        num_branches)))
312     print("success␣rate:␣␣␣␣␣␣␣␣␣␣␣␣{}".format(success_mult_erf/num_runs))
313     print("avg␣time␣symbolic␣coeff:␣{}".format(time_symbolic_mult_erf/num_runs)
        )
314     print("avg␣time␣numeric␣coeff:␣␣{}".format(time_numeric_mult_erf/num_runs))
315     print("avg␣time␣gcd␣finding:␣␣␣␣{}".format(time_gcd_mult_erf/num_runs))
316     print("avg␣total␣running␣time:␣␣{}".format((time_symbolic_mult_erf +
        time_numeric_mult_erf + time_gcd_mult_erf)/num_runs))
```

A.3 Collisions and Second Preimages for Sponges

Functions to generate statistics for collision attacks and second preimage attacks as described in Section 3.6 can be found in Listing A.3. Example results can be found in Section 3.6.2.

Listing A.3: Code for timing and statistics on collision and second preimage finding.

```
# -*- coding: utf-8 -*-
from unbalanced_feistel_networks import *
import random
from time import clock
from datetime import datetime
import pylab as plt
import numpy as np

field_size = 99999989
num_rounds_min = 24
num_rounds_max = 29
num_branches_min = 3
num_branches_max = 6
f_degree = 3 # don't touch, function is hardcoded as x**3
message_length = 2
hash_length = 1
num_runs = 10**3
plot_max_root_count = 6
parallelize = 0 # 0 -> second preimage. 1 -> collision.
```

```
random.seed(0) # determinism

def sponge(message, round_function, key, round_constants, status=None):
    if not status:
        status = [0]*num_branches
    hash_output = []
    # absorb
    for i in range(len(message)):
        last_status = list(status) # not needed for sponge, just avoids reconstruction when interpolating
        status[0] = status[0] + message[i]
        status = ufn_erf(round_function, status, key, round_constants)
    # squeeze
    for _ in range(hash_length):
        hash_output += [status[0]]
        status = ufn_erf(round_function, status, key, round_constants)
    return hash_output, last_status

def random_message(message_length):
    message = []
    for _ in range(message_length):
        message += [random.randint(0, field_size)]
    return message

def polynomial_reduction(polynomial):
    if isinstance(polynomial, (int, sage.rings.integer.Integer, sage.rings.finite_rings.integer_mod.IntegerMod_int)):
        return polynomial
    if polynomial.is_constant():
        return polynomial.constant_coefficient()
    exp_mod = euler_phi(len(polynomial.base_ring()))
    var = polynomial.args()
    new_poly = 0
    for (exp, coeff) in polynomial.dict().iteritems():
        if not isinstance(exp, (tuple, sage.rings.polynomial.polydict.ETuple)):
            exp = (exp,)
        term = 1
        for e,v in zip(exp,var):
            new_e = (e%exp_mod)
            if new_e == 0 and e >= exp_mod:
                new_e = exp_mod
            term *= v**new_e
        new_poly += coeff*term
    return new_poly

def print_basic_stats(data):
    total_runs = sum(sum(len(rounds) for rounds in branches) for branches in data)
    total_roots = sum(sum(sum(rounds) for rounds in branches) for branches in data)
    failed_runs = sum(sum(rounds.count(0) for rounds in branches) for branches in data)
    sccss_runs = total_runs - failed_runs
    print("Total␣runs:␣␣␣␣␣␣␣␣␣␣{}".format(total_runs))
    print("Failed␣runs:␣␣␣␣␣␣␣␣{}".format(failed_runs))
    print("Successful␣runs:␣␣␣␣{}".format(sccss_runs))
    print("Fail␣prob:␣␣␣␣␣␣␣␣␣␣{:02.2f}".format(float(100*failed_runs/total_runs)))
```

```
72      print("Success␣prob:␣␣␣␣␣␣␣{:02.2f}".format(float(100*sccss_runs/total_runs
        )))
73      print("Total␣roots:␣␣␣␣␣␣␣␣{}".format(total_roots))
74      print("Avg␣roots␣if␣sccss:␣{:1.3f}".format(float(total_roots/sccss_runs)))
75
76  def save_distribution_plot(data, num_branches, identifier_string):
77      plt.figure(figsize=(num_rounds_max - num_rounds_min + 1,
        plot_max_root_count / 2), dpi=300)
78      plt.rc('text', usetex=True)
79      plt.rc('font', family='serif', size=16)
80      plt.rc('hatch', color='#999999')
81      # Compile histogram-like
82      all_histo = []
83      for i in range(len(data)):
84          histo = []
85          for j in range(plot_max_root_count):
86              histo += [data[i].count(j)]
87          all_histo += [histo]
88      # Compare neighbouring bars of same y-value to check for overlap
89      biggest_neighbour_sum = 0
90      for i in range(len(all_histo) - 1): # neighboring bar charts
91          for j in range(len(all_histo[i])): # going up the stack
92              biggest_neighbour_sum = max(all_histo[i][j] + all_histo[i+1][j],
        biggest_neighbour_sum)
93      min_gap = 0.03*num_runs # minimum horizontal gap between bars
94      max_stretch = biggest_neighbour_sum + min_gap
95      # Plot the histograms
96      for i in range(len(all_histo)):
97          bars_right = plt.barh(range(plot_max_root_count), [x/max_stretch for x
        in all_histo[i]], left=i, color=["#e00030"] + ["#30e000"]*(
        plot_max_root_count-1), height=0.9, hatch='.')
98          bars_left = plt.barh(range(plot_max_root_count), [-x/max_stretch for x
        in all_histo[i]], left=i, color=["#e00030"] + ["#30e000"]*(
        plot_max_root_count-1), height=0.9, hatch='.')
99          bars_right[0].set_hatch('/')
100         bars_left[0].set_hatch('/')
101     # Add the values of the histograms
102     for i in range(len(all_histo)):
103         for j in range(len(all_histo[i])):
104             if all_histo[i][j] != 0:
105                 plt.text(i, j, all_histo[i][j], horizontalalignment='center',
        verticalalignment='center', color='black')
106     plt.xticks(list(range(num_rounds_max - num_rounds_min + 1)), [
        num_rounds_min + i for i in range(num_rounds_max - num_rounds_min + 2)])
107     plt.xlabel('number␣of␣rounds␣$r$')
108     plt.ylabel('amount␣of␣{}'.format(identifier_string))
109     plt.ylim(ymin = -0.5, ymax = plot_max_root_count - 0.5)
110     plt.savefig('/tmp/sponge/sponge_{}_plot_r=({}-{})_t={}.png'.format(
        identifier_string.replace("␣","_") , num_rounds_min, num_rounds_max,
        num_branches),  bbox_inches='tight')
111
112 def second_preimage(num_rounds, round_constants, print_progress=False):
113     # Setup
114     field = GF(field_size)
115     key = field(0)
116     x = polygen(field, 'x')
117     full_degree = min(f_degree ** num_rounds, field_size - 1)
118     round_function = lambda x : x**3 # gmimc specific
```

```
119
120     num_scnd_preimage = []
121     num_no_scnd_preimage_run = 0
122     time_poly_evaluation = 0
123     time_root = 0
124     time_total = clock()
125     for i in range(num_runs):
126         if print_progress and i % 20 == 0:
127             print("{:>.19} - Starting run {}/{}".format(str(datetime.now()), i,
        num_runs))
128         message_0 = random_message(message_length)
129         hash_output, _ = sponge(message_0, round_function, key, round_constants
        )
130         hash_output = hash_output[0]  # Only works with hash length == 1
131         message_1 = random_message(message_length)
132         _, last_status = sponge(message_1, round_function, key, round_constants
        )
133         # symbolic evaluation
134         time = clock()
135         poly, _ = sponge([x], round_function, key, round_constants, status=
        last_status)
136         poly  = polynomial_reduction(poly[0])
137         time_poly_evaluation += clock() - time
138         # find second preimages
139         time = clock()
140         all_roots = (poly - hash_output).roots()
141         time_root += clock() - time
142         all_roots = [r[0] for r in all_roots] # disregard multiplicativity
143         # check our solutions
144         num_found_second_preimages = 0
145         for root in all_roots:
146             hash_output_0, _ = sponge(message_0, round_function, key,
        round_constants)
147             hash_output_1, _ = sponge(message_1[:-1] + [root], round_function,
        key, round_constants)
148             if hash_output_0 == hash_output_1 and message_0 != message_1[:-1] +
        [root]:
149                 num_found_second_preimages += 1
150         num_scnd_preimage += [num_found_second_preimages]
151         if num_found_second_preimages <= 0:
152             num_no_scnd_preimage_run += 1
153     time_total = clock() - time_total
154     return num_scnd_preimage, num_no_scnd_preimage_run, time_poly_evaluation,
        time_root, time_total
155
156 def collision(num_rounds, round_constants, print_progress=False):
157     # Setup
158     field = GF(field_size)
159     key = field(0)
160     x = polygen(field, 'x')
161     full_degree = min(f_degree ** num_rounds, field_size - 1)
162     round_function = lambda x : x**3 # gmimc specific
163
164     num_collisions = []
165     num_no_collision_run = 0
166     time_poly_evaluation = 0
167     time_root = 0
168     time_total = clock()
```

```
169     for i in range(num_runs):
170         if print_progress and i % 20 == 0:
171             print("{:>.19} - Starting run {}/{}".format(str(datetime.now()), i,
        num_runs))
172         # symbolic evaluation
173         message_0 = random_message(message_length)
174         message_1 = random_message(message_length)
175         message_0[-1] = x
176         message_1[-1] = x
177         time = clock()
178         poly_0, _ = sponge(message_0, round_function, key, round_constants)
179         poly_1, _ = sponge(message_1, round_function, key, round_constants)
180         poly_0  = polynomial_reduction(poly_0[0]) # Only works with hash length
        == 1
181         poly_1  = polynomial_reduction(poly_1[0])
182         time_poly_evaluation += clock() - time
183         # find collisions
184         time = clock()
185         all_roots = (poly_0 - poly_1).roots()
186         time_root += clock() - time
187         all_roots = [r[0] for r in all_roots] # disregard multiplicativity
188         # check our solutions
189         num_found_collisions = 0
190         for root in all_roots:
191             hash_output_0, _ = sponge(message_0[:-1] + [root], round_function,
        key, round_constants)
192             hash_output_1, _ = sponge(message_1[:-1] + [root], round_function,
        key, round_constants)
193             if hash_output_0 == hash_output_1 and message_0[:-1] != message_1
        [:-1]:
194                 num_found_collisions += 1
195         num_collisions += [num_found_collisions]
196         if num_found_collisions <= 0:
197             num_no_collision_run += 1
198     time_total = clock() - time_total
199     return num_collisions, num_no_collision_run, time_poly_evaluation,
        time_root, time_total
200
201 round_constants = []
202 all_data = []
203 for _ in range(num_rounds_max):
204     round_constants += [random.randint(0, field_size)]
205
206 if parallelize == 0:
207     for num_branches in range(num_branches_min, num_branches_max + 1):
208         second_preimage_data = []
209         for num_rounds in range(num_rounds_min, num_rounds_max + 1):
210             num_scnd_preimage, num_no_scnd_preimage_run, time_poly_evaluation,
        time_root, time_total = second_preimage(num_rounds, round_constants[:
        num_branches])
211             print("Field size:   {}".format(field_size))
212             print("Num rounds:   {}".format(num_rounds))
213             print("Num branches: {}".format(num_branches))
214             print("Degree round: {}".format(f_degree))
215             print("Average over  {} randomized runs".format(num_runs))
216             print("avg num scnd primg:          {}".format(sum(
        num_scnd_preimage)/num_runs))
```

```
217             print("runs␣with␣no␣scnd␣primg:␣␣␣␣␣{}".format(
    num_no_scnd_preimage_run))
218             if num_no_scnd_preimage_run < num_runs:
219                 print("avg␣num␣scnd␣primg␣(if␣>=1):␣{}".format(sum(
    num_scnd_preimage)/(num_runs - num_no_scnd_preimage_run)))
220             print("avg␣time␣evaluating:␣␣␣␣␣␣␣␣␣{}".format(time_poly_evaluation
    /num_runs))
221             print("avg␣time␣root␣finding:␣␣␣␣␣␣␣{}".format(time_root/num_runs))
222             print("avg␣total␣runtime:␣␣␣␣␣␣␣␣␣␣{}".format(time_total/num_runs)
    )
223             print("")
224             second_preimage_data += [num_scnd_preimage]
225         print("Second␣Preimage␣Data␣as␣backup:␣(t={})".format(num_branches))
226         print(second_preimage_data)
227         print("")
228         all_data += [second_preimage_data]
229         save_distribution_plot(second_preimage_data, num_branches, 'second␣
    preimages')
230
231 if parallelize == 1:
232     for num_branches in range(num_branches_min, num_branches_max + 1):
233         collision_data = []
234         for num_rounds in range(num_rounds_min, num_rounds_max + 1):
235             num_collisions, num_no_collision_run, time_poly_evaluation,
    time_root, time_total = collision(num_rounds, round_constants[:num_branches
    ])
236             print("Field␣size:␣␣␣{}".format(field_size))
237             print("Num␣rounds:␣␣␣{}".format(num_rounds))
238             print("Num␣branches:␣{}".format(num_branches))
239             print("Degree␣round:␣{}".format(f_degree))
240             print("Average␣over␣␣{}␣randomized␣runs".format(num_runs))
241             print("avg␣num␣collisions:␣␣␣␣␣␣␣␣␣{}".format(sum(num_collisions)/
    num_runs))
242             print("runs␣with␣no␣collision:␣␣␣␣␣{}".format(num_no_collision_run
    ))
243             if num_no_collision_run < num_runs:
244                 print("avg␣num␣collision␣(if␣>=1):␣␣{}".format(sum(
    num_collisions)/(num_runs - num_no_collision_run)))
245             print("avg␣time␣evaluating:␣␣␣␣␣␣␣␣␣{}".format(time_poly_evaluation
    /num_runs))
246             print("avg␣time␣root␣finding:␣␣␣␣␣␣␣{}".format(time_root/num_runs))
247             print("avg␣total␣runtime:␣␣␣␣␣␣␣␣␣␣{}".format(time_total/num_runs)
    )
248             print("")
249             collision_data += [num_collisions]
250         print("Collision␣Data␣as␣backup:␣(t={})".format(num_branches))
251         print(collision_data)
252         print("")
253         all_data += [collision_data]
254         save_distribution_plot(collision_data, num_branches, 'collisions')
255 print_basic_stats(all_data)
```

Bibliography

[1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 191–219. Springer, 2016.

[2] Martin R Albrecht, Carlos Cid, Lorenzo Grassi, Reinhard Khovratovich, Dmitry an Lüftenegger, Christian Rechberger, and Markus Schofnegger. Algebraic cryptanalysis of stark-friendly designs: Application to marvellous and mimc. *IACR Cryptology ePrint Archive*, 2019:419, 2019.

[3] Martin R. Albrecht, Lorenzo Grassi, Léo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. Feistel structures for mpc, and more. In Kazue Sako, Steve Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part II*, volume 11736 of *Lecture Notes in Computer Science*, pages 151–171. Springer, 2019.

[4] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. Ciphers for MPC and FHE. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 430–454. Springer, 2015.

[5] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. Cryptology ePrint Archive, Report 2019/426, 2019. https://eprint.iacr.org/2019/426.

[6] Abdelrahaman Aly, Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Nigel P Smart, and Tim Wood. Scale–mamba v1. 3: Documentation. Technical report, Technical Report, 2019.

[7] Elena Andreva, Arnab Roy, and Ferdinand Sauer. Interpolation Cryptanalysis of Unbalanced Feistel Networks with Low Degree Round Functions. Under Submission, 11 2019.

[8] Tomer Ashur and Siemen Dhooghe. Marvellous: a stark-friendly family of cryptographic primitives. Cryptology ePrint Archive, Report 2018/1098, 2018. https://eprint.iacr.org/2018/1098.

[9] Mihir Bellare, Sha Goldwasser, Carsten Lund, and Alexander Russell. E cient probabilistically checkable proofs and applications to approximation. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 294–304, 1993.

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Report 2018/046, 2018. https://eprint.iacr.org/2018/046.

[11] Eli Ben-Sasson, Lior Goldberg, Swastik Kopparty, and Shubhangi Saraf. Deep-fri: Sampling outside the box improves soundness. Cryptology ePrint Archive, Report 2019/336, 2019. https://eprint.iacr.org/2019/336.

[12] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, number 9. Citeseer, 2007.

[13] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 181–197. Springer, 2008.

[14] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keccak. In *Annual international conference on the theory and applications of cryptographic techniques*, pages 313–314. Springer, 2013.

[15] Eli Biham, Orr Dunkelman, and Nathan Keller. New results on boomerang and rectangle attacks. In *International Workshop on Fast Software Encryption*, pages 1–16. Springer, 2002.

[16] Eli Biham and Adi Shamir. *Differential cryptanalysis of the data encryption standard*. Springer Science & Business Media, 2012.

[17] Nir Bitansky, Ran Canetti, Alessandro Chiesa, and Eran Tromer. From extractable collision resistance to succinct non-interactive arguments of knowledge, and back again. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 326–349, New York, NY, USA, 2012. ACM.

[18] Xavier Bonnetain. Collisions on Feistel-MiMC and univariate GMiMC. Cryptology ePrint Archive, Report 2019/951, 2019. https://eprint.iacr.org/2019/951.

[19] Bruno Buchberger. Bruno buchberger's phd thesis 1965: An algorithm for finding the basis elements of the residue class ring of a zero dimensional polynomial ideal. *Journal of symbolic computation*, 41(3-4):475–511, 2006.

[20] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 315–334. IEEE, 2018.

[21] Benedikt Bünz, Ben Fisch, and Alan Szepieniec. Transparent snarks from dark compilers. Technical report, Cryptology ePrint Archive, Report 2019/1229, 2019, https://eprint. iacr. org . . . , 2019.

[22] David Chaum, Claude Crépeau, and Ivan Damgard. Multiparty unconditionally secure protocols. In *Proceedings of the twentieth annual ACM symposium on Theory of computing*, pages 11–19. ACM, 1988.

[23] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachene. Faster fully homomorphic encryption: Bootstrapping in less than 0.1 seconds. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 3–33. Springer, 2016.

[24] Benny Chor, Shafi Goldwasser, Silvio Micali, and Baruch Awerbuch. Verifiable secret sharing and achieving simultaneity in the presence of faults. In *26th Annual Symposium on Foundations of Computer Science (sfcs 1985)*, pages 383–395. IEEE, 1985.

[25] Don Coppersmith. Analysis of iso/ccitt document x. 509 annex d. *IBM TJ Watson Center, Yorktown Heights, NY*, 10598, 1989.

[26] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[27] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard*. Springer Science & Business Media, 2013.

[28] Ivan Damgård, Valerio Pastro, Nigel Smart, and Sarah Zakarias. Multiparty computation from somewhat homomorphic encryption. In *Annual Cryptology Conference*, pages 643–662. Springer, 2012.

[29] Hüseyin Demirci and Ali Aydın Selçuk. A meet-in-the-middle attack on 8-round aes. In *International Workshop on Fast Software Encryption*, pages 116–126. Springer, 2008.

[30] Whitfield Diffie and Martin E Hellman. Special feature exhaustive cryptanalysis of the nbs data encryption standard. *Computer*, 10(6):74–84, 1977.

[31] Léo Ducas and Daniele Micciancio. Fhew: bootstrapping homomorphic encryption in less than a second. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 617–640. Springer, 2015.

[32] Jean-Charles Faugere. A new efficient algorithm for computing gröbner bases (f4). *Journal of pure and applied algebra*, 139(1-3):61–88, 1999.

[33] PUB FIPS. 46-3. data encryption standard (des). *National Institute of Standards and Technology*, 25(10):1–22, 1999.

[34] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography Conference*, pages 303–324. Springer, 2005.

[35] Craig Gentry et al. Fully homomorphic encryption using ideal lattices. In *Stoc*, volume 9, pages 169–178, 2009.

[36] Craig Gentry, Shai Halevi, and Nigel P Smart. Fully homomorphic encryption with polylog overhead. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 465–482. Springer, 2012.

[37] Craig Gentry, Shai Halevi, and Nigel P Smart. Homomorphic evaluation of the aes circuit. In *Annual Cryptology Conference*, pages 850–867. Springer, 2012.

[38] Marc Girault. Hash-functions using modulo-n operations. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 217–226. Springer, 1987.

[39] Oded Goldreich. *Foundations of cryptography: volume 1, basic tools*. Cambridge university press, 2007.

[40] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 218–229. ACM, 1987.

[41] Oded Goldreich and Yair Oren. Definitions and properties of zero-knowledge proof systems. *Journal of Cryptology*, 7(1):1–32, 1994.

[42] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The hades design strategy. Cryptology ePrint Archive, Report 2019/1107, 2019. https://eprint.iacr.org/2019/1107.

[43] Network Working Group et al. Rfc4949: Internet security glossary, version 2, 2007.

[44] Viet Tung Hoang and Phillip Rogaway. On generalized feistel networks. In *Annual Cryptology Conference*, pages 613–630. Springer, 2010.

[45] Thomas Jakobsen and Lars R. Knudsen. The Interpolation Attack on Block Ciphers. In *In Fast Software Encryption*, pages 28–40. Springer-Verlag, 1997.

[46] Stanisław Jarecki and Xiaomin Liu. Efficient oblivious pseudorandom function with applications to adaptive ot and secure computation of set intersection. In *Theory of Cryptography Conference*, pages 577–594. Springer, 2009.

[47] Lars R Knudsen and Matthew Robshaw. *The block cipher companion*. Springer Science & Business Media, 2011.

[48] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[49] Michael Kraitsberg, Yehuda Lindell, Valery Osheter, Nigel P Smart, and Younes Talibi Alaoui. Adding distributed decryption and key generation to a ring-lwe based cca encryption scheme. In *Australasian Conference on Information Security and Privacy*, pages 192–210. Springer, 2019.

[50] Xuejia Lai and James L Massey. A proposal for a new block encryption standard. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 389–404. Springer, 1990.

[51] Xuejia Lai, James L Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 17–38. Springer, 1991.

[52] Susan K Langford and Martin E Hellman. Differential-linear cryptanalysis. In *Annual International Cryptology Conference*, pages 17–25. Springer, 1994.

[53] V. K. Leont'ev. Roots of random polynomials over a finite field. *Mathematical Notes*, 80(1):300–304, Jul 2006.

[54] Chaoyun Li and Bart Preneel. Improved Interpolation Attacks on Cryptographic Primitives of Low Algebraic Degree. Cryptology ePrint Archive, Report 2019/812, 2019. https://eprint.iacr.org/2019/812.

[55] Arnab Roy Lorenzo Grassi, Christian Rechberger. Gmimcs new key schedule. personal communication, 8 2019.

[56] Michael Luby and Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17(2):373–386, 1988.

[57] Yiyuan Luo, Xuejia Lai, and Zheng Gong. Pseudorandomness analysis of the (extended) lai–massey scheme. *Information processing letters*, 111(2):90–96, 2010.

[58] Mitsuru Matsui. Linear cryptanalysis method for des cipher. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 386–397. Springer, 1993.

[59] Mitsuru Matsui. The first experimental cryptanalysis of the data encryption standard. In *Annual International Cryptology Conference*, pages 1–11. Springer, 1994.

[60] E Hastings Moore. A doubly-infinite system of simple groups. *Bulletin of the American Mathematical Society*, 3(3):73–78, 1893.

[61] Kaisa Nyberg. Linear approximation of block ciphers. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 439–444. Springer, 1994.

[62] Kaisa Nyberg. Generalized feistel networks. In *International conference on the theory and application of cryptology and information security*, pages 91–104. Springer, 1996.

[63] Kaisa Nyberg and Lars Ramkilde Knudsen. Provable security against a differential attack. *Journal of Cryptology*, 8(1):27–37, 1995.

[64] Jacques Patarin. Security of random feistel schemes with 5 or more rounds. In *Annual International Cryptology Conference*, pages 106–122. Springer, 2004.

[65] Benny Pinkas, Thomas Schneider, Nigel P Smart, and Stephen C Williams. Secure two-party computation is practical. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 250–267. Springer, 2009.

[66] Bart Preneel. *Correcting-Block Attack*, pages 259–260. Springer US, Boston, MA, 2011.

[67] Ronald Rivest. The md5 message-digest algorithm. 1992.

[68] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In *International Workshop on Fast Software Encryption*, pages 371–388. Springer, 2004.

[69] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474. IEEE, 2014.

[70] Bruce Schneier and John Kelsey. Unbalanced feistel networks and block cipher design. In *International Workshop on Fast Software Encryption*, pages 121–144. Springer, 1996.

[71] Nigel P Smart and Frederik Vercauteren. Fully homomorphic encryption with relatively small key and ciphertext sizes. In *International Workshop on Public Key Cryptography*, pages 420–443. Springer, 2010.

[72] Starkware. STARK-Friendly Hash Challenge. Website, 8 2019. https://starkware.co/hash-challenge/.

[73] Marc Stevens, Arjen K Lenstra, and Benne De Weger. Chosen-prefix collisions for md5 and applications. *International Journal of Applied Cryptography*, 2(ARTICLE):322–359, 2012.

[74] Marc Stevens, Alexander Sotirov, Jacob Appelbaum, Arjen Lenstra, David Molnar, Dag Arne Osvik, and Benne De Weger. Short chosen-prefix collisions for md5 and the creation of a rogue ca certificate. In *Annual International Cryptology Conference*, pages 55–69. Springer, 2009.

[75] H.-J. Stoss. The complexity of evaluating interpolation polynomials. *Theoretical Computer Science*, 41:319–323, 1985.

[76] The Sage Developers. *SageMath, the Sage Mathematics Software System (Version 8.9)*, 2019. `https://www.sagemath.org`.

[77] Serge Vaudenay. On the lai-massey scheme. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 8–19. Springer, 1999.

[78] Joachim Von Zur Gathen and Jürgen Gerhard. *Modern Computer Algebra*. Cambridge university press, 2013.

[79] David Wagner. The boomerang attack. In *International Workshop on Fast Software Encryption*, pages 156–170. Springer, 1999.

[80] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[81] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.