TU WIEN Informatics

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

telecos BCN

Implementation of algebraic cryptographic primitives in the Plonk framework

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Engineering

in

Telecommunications Technologies and Services Engineering

by

Àlex Mitjans Llorach

to the Faculty of Informatics

at the TU Wien

Advisor: Asst. Prof. Elena Andreeva
Assistance: Stefano Trevisani

Vienna, June 15, 2024

Àlex Mitjans Llorach

Elena Andreeva

Abstract

As cryptographic technologies evolve, the need for specialized hash functions to operate efficiently over different computational environments become necessary. Traditional symmetric alogrithms like AES and SHA-3 have been optimized for hardware and software implementations, which are designed over binary fields. However, protocols like zero-knowledge proofs require hash functions that are optimised over large prime fields.

This thesis addresses the growing demand for Arithmetization-Oriented (AO) cryptographic hash functions for zero-knowledge applications. The performance and efficiency of many zero-knowledge applications often depends on the efficiency of the hash function used.

In response to this need, this work explores a selection of these hash functions and implements them within two zero-knowledge proving systems: Plonk and Plonky2. The research focuses on benchmarking these hash functions to asses their performance within these sytems. The results obtained provide a fundamental base for future research in the field of zero-knowledge proof technologies, recommending potential areas for further development.

Keywords: Hash functions · Arithmetization-oriented · Plonk · Plonky2 · Zero-knowledge · Arithmetic circuits

Resumen

A medida que las tecnologías criptográficas evolucionan, surge la necesidad de funciones hash especializadas para operar eficientemente en diferentes entornos computacionales. Algoritmos simétricos tradicionales como AES y SHA-3 han sido optimizados para implementaciones en hardware y software, enfocándose principalmente en campos binarios. Sin embargo, protocolos como las pruebas de conocimiento cero requieren funciones hash que estén optimizadas sobre grandes campos primos.

Esta tesis aborda la creciente demanda de funciones hash criptográficas orientadas a la aritmetización adaptadas para aplicaciones de conocimiento cero. El rendimiento y la eficiencia de las aplicaciones de conocimiento cero a menudo dependen de la eficiencia de la función hash utilizada.

En respuesta a esta necesidad, este trabajo explora una selección de estas funciones hash y las implementa dentro de dos sistemas de prueba de conocimiento cero: Plonk y Plonky2. La investigación se centra en la evaluación comparativa de estas funciones hash para evaluar su rendimiento dentro de estos sistemas. Los resultados obtenidos proporcionan una base fundamental para la investigación futura en el campo de las tecnologías de prueba de conocimiento cero, recomendando áreas potenciales para un desarrollo adicional.

Palabras clave: Funciones hash · Orientado a la aritmetización · Plonk · Plonky2 · Conocimiento cero · Circuitos aritméticos

Resum

A mesura que les tecnologies criptogràfiques evolucionen, sorgeix la necessitat de funcions hash especialitzades per operar eficaçment en diferents entorns computacionals. Algorismes simètrics tradicionals com AES i SHA-3 han estat optimitzats per a implementacions en maquinari i programari, centrant-se principalment en camps binaris. No obstant això, protocols com les proves de coneixement zero requereixen funcions hash que estiguin optimitzades sobre grans camps prims.

Aquesta tesi aborda la creixent demanda de funcions hash criptogràfiques orientades a l'aritmetització adaptades per a aplicacions de coneixement zero. El rendiment i l'eficiència de les aplicacions de coneixement zero sovint depenen de l'eficiència de la funció hash utilitzada.

En resposta a aquesta necessitat, aquest treball explora una selecció d'aquestes funcions hash i les implementa dins de dos sistemes de prova de coneixement zero: Plonk i Plonky2. La investigació es centra en l'avaluació comparativa d'aquestes funcions hash per avaluar el seu rendiment dins d'aquests sistemes. Els resultats obtinguts proporcionen una base fonamental per a la recerca futura en el camp de les tecnologies de prova de coneixement zero, recomanant àrees potencials per a un desenvolupament addicional.

Paraules clau: Funcions hash · Orientat a l'aritmetització · Plonk · Plonky2 · Coneixement zero · Circuits aritmètics

Acknowledgements

First of all, I wish to express my appreciation to my advisor, Prof. Elena Andreeva, for providing me with her guidance and the opportunity to conduct my thesis within her research group. I would also like to thank Stefano Trevisani for his advice and for assisting me with technical issues that arose during my thesis.

Finally, I would like to thank my family for their support and belief in me throughout my studies, and for making this incredible experience in Vienna possible.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The interest in zero-knowledge proofs has experienced a notable growth, especially with the increasing interest in blockchain technology and verifiable computation. Although this technology was first described by Goldwasser et al. in 1985, blockchain technology has helped zero-knowledge proofs increase his popularity. Today, this technology is widely used across various industries, from Web3 platforms to supply chains and the Internet of Things (IoT).

In the world of blockchain technology, zero-knowledge proofs play a crucial role in enhancing privacy and security. They enable participants to prove the validity of transactions or computations without revealing sensitive or private information, therefore addressing concerns regarding data privacy and confidentiality.

Moreover, in IoT networks, they can prove and verify the authenticity of exchanged data. Similarly, in supply chains, zero-knowledge proofs are used to validate product authenticity while mantaining privacity.

Cryptographic hash functions play a fundamental role in ensuring the integrity and security of digital data. These functions are widely employed in various cryptographic protocols, digital signatures, messages and passwords verification, for example. Hash functions such as SHA-2 or SHA-3, while widely used and secure, may not meet the efficient requirements of zero-knowledge proofs. This has led to the development of zero-knowledge friendly hash functions such as Poseidon, Arion and Griffin to name a few, which offer compact arithmetic circuits, enabling faster proof generation and verification within zero-knowledge proof systems.

1.2 Statement of purpose

This thesis aims to investigate some of the most used and novel hash functions efficient in zero-knowledge and implement them within a zero-knowledge framework. Specifically, the obejctives of this research include the plain implementation of these hash functions in the Goldilocks field and the BLS12-381 scalar field. Subsequently, these hash functions will also be implemented in two zero-knowledge frameworks, Plonk and Plonky2.
Furthermore, benchmarks have been done to asses the performance of these hash functions in plain performance and within these frameworks.

1.3 Outline

This thesis is organized as follows. We begin by presenting the necessary background knowledge and defining all the concepts used throughout this thesis in Chapter 2. Then, in Chapter 3, we explain proofs systems, define zero-knowledge proofs, SNARKs, and specifically, the proof system that we will use, Plonk. In Chapter 4, we define the hash functions that will be implemented in this thesis and provide all the necessary theoretical knowledge to understand them. Next, in Chapter 5, we explain how we developed these hash functions, including some of the parameters used for their implementation. In Chapter 6, we present and discuss the results obtained from the benchmarks. Next, conclusions are developed togheter with some insights for future work in Chapter 7. Finally, in Chapter 8, we detail a sustainability analysis and ethical implications of ower project.

1.4 Work plan

The work on this thesis has been organized according to the schedule in Figure 1.1.

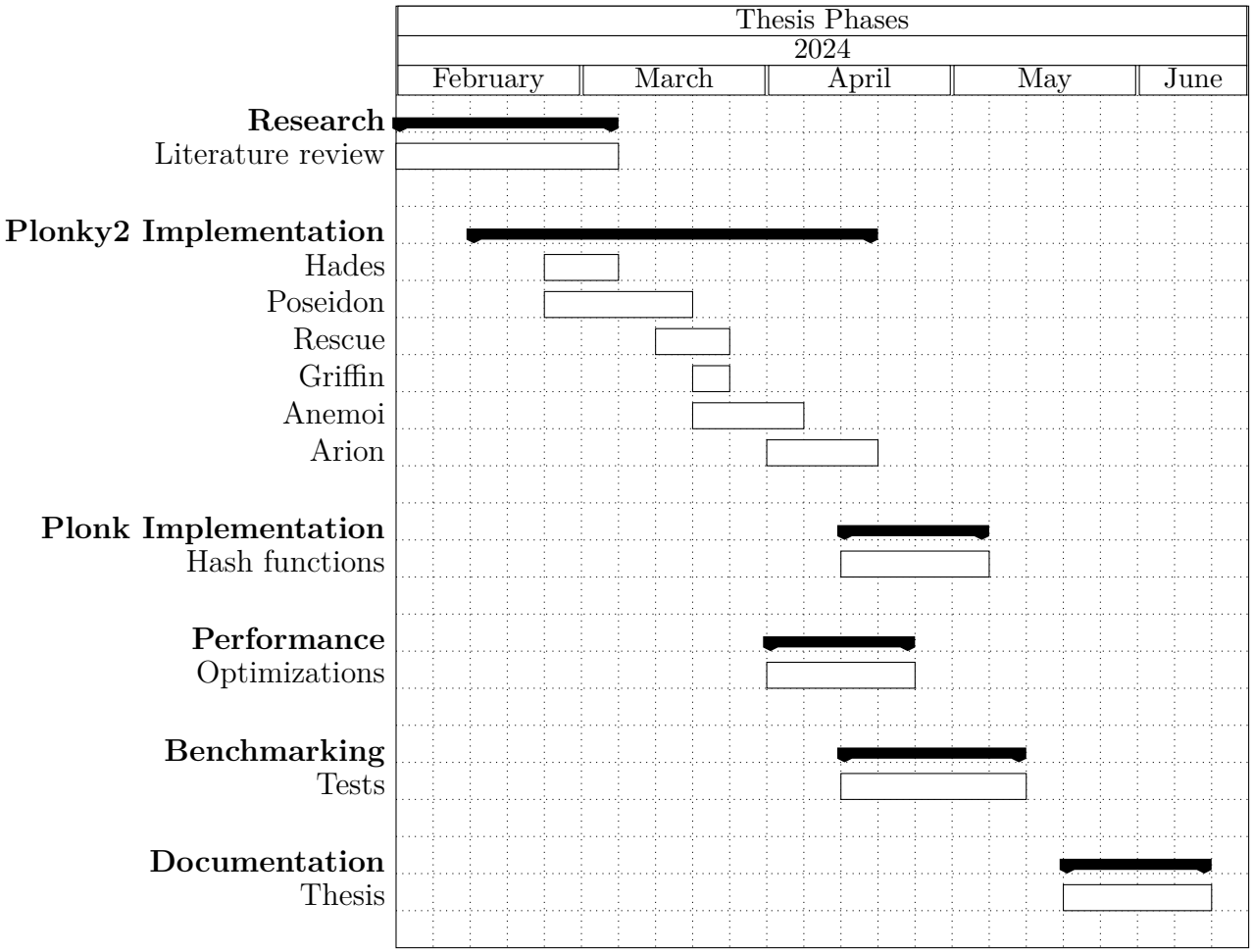

Figure 1.1: Gantt Chart

CHAPTER 2

Preliminaries

In this chapter, we provide some background knowledge that will be required for understanding future parts of the thesis.

2.1 Hash Functions

A bit-oriented hash function [Pre94] is a mathematical function $H : \{0,1\}^* \rightarrow \{0,1\}^n$ with $n \geq 1$ which takes an input (or 'message') of variable length and computes a fixed length output.

A hash function generally starts by dividing the input data into fixed size blocks. This is necessary because hash functions operate on data with a fixed length, and the input could be of any length. This padding process varies depending on the details of the hash function.

Properties

To be an effective cryptographic tool, the hash function needs to posses the following properties [Pre93]:

- **One way function (pre-image resistance):** It should be computationally difficult to reverse a hash funtion.
- **Target collision resistance (Second pre-image resistance):** Given an input and its hash, it should be computationally difficult to find a different input with the same hash.

- **Collision resistance:** It should be computationally difficult to find two different inputs of any arbitrary length that result in the same hash value.
- **Deterministic:** The same input will always produce the same output.
- **Efficiency:** The hash function must compute values quickly to ensure practical usability.

2.1.1 Block ciphers

Block ciphers [Knu99] are symmetric key cryptographic alogrithms that operates on fixed length groups of bits, called blocks. The alogrithm accepts an input block of size n bits and a key of size k bits, and produce an n bits output block. They are designed to be reversible, meaning the ciphertext can be transformed back into the plaintext using the same key.

$$E : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}^n. \tag{2.1}$$

Block ciphers usually use multiple rounds of operations, including permutations and substitutions, until inverting the function becomes hard. They can be used to construct hash functions.

Feistel networks, *Substitution-Permutation Networks* (SPNs), *Horst* schemes and *open Flyestel* components are fundamental structures used to design block ciphers.

Feistel Networks

A Feistel network [Nyb96, Fei73] is a structure used to construct block ciphers. The input block X is divided into two parts X_1 and X_2. One round of a Feistel network is defined as follows:

$$\begin{aligned} Y_1 = X_1 \oplus F(X_2, K) \\ Y_2 = X_2 \\ Y_1' = Y_2 \\ Y_2' = Y_1, \end{aligned}$$

where K is the round key, F is the round function and (Y_1', Y_2') is the data output of the round taken as input to the next round.

The Feistel network provides difussion of the input blocks. After two rounds of the network the dissufion is complete, that is, both output blocks depend on both input blocks.

Substitution-permutation network

Substitution-permutation network (SPN) [Fei73, Sha49] is used to generate block ciphers. It is a repeated series of mathematical operations.

SPN takes an input block of the plaintext and a key as the input, and applies substitution (S-box) and Permutation (P-box) to each round.
An **S-box** substitues each byte of the input block with another byte. This layer of the SPN introduces non-linearity.
A **P-box** is a permutation of every bit: the bits of the S-box output are permuted, and then are passed to the S-box of the next round.

Thus, in SPN consists on three primary operations in each round: addition of a round key, substitution and permutation.

Horst scheme

Horst is another structure that can be used to design block ciphers, it is defined in [GHR+22] and used in the Griffin hash function design. Let $t \geq 2$. For each $i \in 1, 2, \ldots, t-1$, let $G_i : \mathbb{F}_q^i \to \mathbb{F}_q \backslash \{0\}$ and $\mathbb{F}_i : \mathbb{F}_q^i \to \mathbb{F}_q$. Horst is defined over $\mathbb{F}_q^t$ as $x = (x_0, \ldots, x_{t-1}) \mapsto y = (y_0, \ldots, y_{t-1})$, where

$$y_i := \begin{cases} x_{i+1} \cdot G_{i+1}(x_0, x_1, \ldots, x_i) + F_{i+1}(x_0, x_1, \ldots, x_i) & \text{if } i \in 0, 1, \ldots, t-2, \\ x_0 & \text{otherwise } (i = t-1). \end{cases}$$

The final circular shift is crucial for achieving full diffusion (as in the case of any Feistel scheme), but it can be replaced with a different linear diffusion.

Open Flyestel structure

The *open Flyestel* structure is a non-linear component presented and used in the design of Anemoi [BBC+23].

Let $Q_\gamma : \mathbb{F}q \to \mathbb{F}_q$ and $Q_\delta : \mathbb{F}_q \to \mathbb{F}_q$ be two quadratic functions, and let $E : \mathbb{F}_q \to \mathbb{F}_q$ be a permutation. Then, the *Flyestel* is a pair of functions relying on Q_γ, Q_δ and E. The *open Flyestel* is the permutation of $(\mathbb{F}_q)^2$ obtained using a 3-round Feistel network with Q_γ, E^{-1}, and Q_δ as round functions. It is defined as $\mathcal{H}(x, y) = (u, v)$.

$$\begin{cases} x \leftarrow x - Q_\gamma(y), \\ y \leftarrow y - E^{-1}(x), \\ x \leftarrow x + Q_\delta(y), \\ u \leftarrow x, v \leftarrow y. \end{cases} \tag{2.2}$$

2.2 Sponge structure

The sponge construction will be used in the different hash functions that are implemented on this thesis.

Overview

The sponge construction maps an input of arbitrary length to an arbitrary length output. So the input don't have any restriction of his length, and the output length is not determined by the input [GJMG11].

The sponge construction is built over an internal permutation, $\mathcal{P}$, that operates on the $\mathbb{F}_q^{r+c}$. This permutation works on a state with a length of t called the width, where $t = r + c$, r is the rate (outer part of the state) and c the capacity (inner part of the state). The digest consists of h elements.

A sponge can be used to obtain hash function: you use the sponge to absorb the input data, and then squeeze out just enough to form a hash. This approach enables hash functions to handle inputs of varying lengths while producing fixed-length outputs. Figure 2.1 explains the principle of the Sponge construction.

First, the state is initialized to zero, and the following operations are made to the input message m:

1. **Padding:** The padding rule of the sponge works as follows: when the length of the input message is not a multiple of the rate of the sponge, we append $1 \in \mathbb{F}_q$ to the input followed by enough zeros to make it a multiple of r, and split the message into blocks of size r, called chunks.

2. **Absorbing:** When the input is divided into blocks of size r (chunks), they are absorbed into the state of the sponge construction. Each one of these blocks is added to the outer part of the state using vector addition in $\mathbb{F}^r$ (where $\mathbb{F}$ is a finite field), and then the permutation is applied to it. This process is repeated for each block.

3. **Squeezing:** In this phase the output is produced by extracting $\min(h, r)$ elements from the outer part of the state r. If $r < h$, we apply the permutation $\mathcal{P}$ and extract the r elements, this process is repeated until we reach the desired output length h.

The length of the output, h, can be arbitrarily chosen.

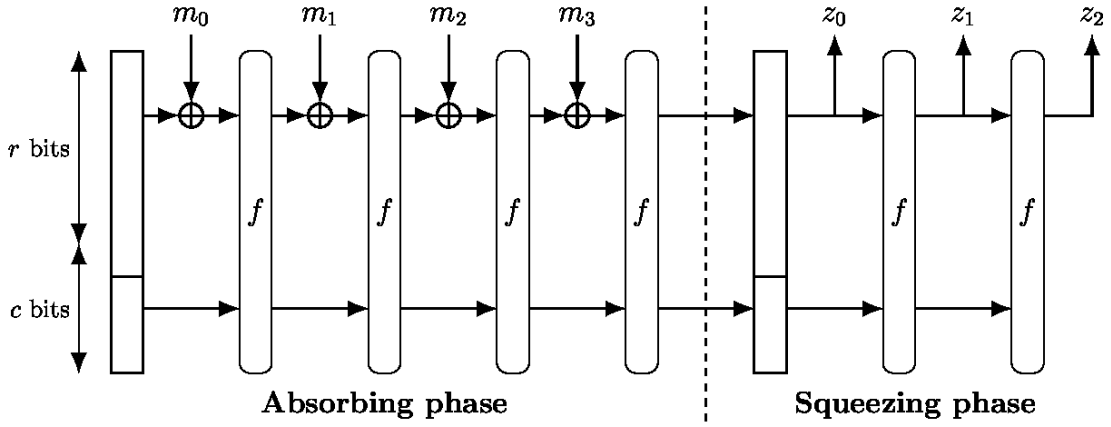

Figure 2.1: Sponge construction with 4 input chunks and a permutation f.

2.2.1 Cryptographic permutation

A cryptographic permutation is a function that maps an input to an unique output of the same length. The function is designed to be bijective, every input has a unique output. It consists in a sequence of operations iterated over R rounds. This sequence of operations are called the round function.

Some of the most typical elements that will be used in the implemented hash functions are:

Non-linear layer (S-box)

In the designs of S-boxes, functions like $f(x) = x^{\alpha}$ over finit fields $\mathbb{F}_p$ are often used. These functions must be bijective, to be suitable for cryptographic purposes. To ensure that it is bijective, *Fermat's Little Theorem* [DGDG11] is used along with specific properties of finite fields.

Fermat's Little Theorem states that if p is a prime number and a is an integer such that a is not divisible by p, then:

$$a^{p-1} \equiv 1 \pmod{p} \tag{2.3}$$

The condition $\gcd(\alpha, p-1) = 1$ ensures that the funciton $f(x) = x^{\alpha}$ is a permutation of the field $\mathbb{F}_p$. Next, a detailed explanation is provided:

- The condition $\gcd(\alpha, p-1) = 1$ guarantees that α has a multiplicative inverse modulo $p-1$. This means there exists an integer β such that:

$$\alpha \cdot \beta \equiv 1 \pmod{p-1}. \tag{2.4}$$

This inverse β allows us to define the inverse function $g(x) = x^\beta$. Consequently, $f(g(x)) = g(f(x)) = x$, proving that $f(x) = x^\alpha$ is bijective.

- For $f(x) = x^\alpha$ to be a permutation, the mapping $x \mapsto x^\alpha$ must cover all elements of $\mathbb{F}_p$. Thus, if $\gcd(\alpha, p-1) \neq 1$, there would exists some $a, b \in \mathbb{F}_p$, such that $a^\alpha = b^\beta$ for $a \neq b$, not satisfying the property.

Constant addition

This component of the permutation consists on the addition of a constant to the state of the sponge, how to add this constant and when depends on the design of the hash function.

$$\text{state} = \text{state} \oplus c_i \tag{2.5}$$

The round constants, $c_0, \ldots, c_{R-1}$, are fixed and public.

Liner Layer

The linear layer consists on the computation of a matrix times vector multiplication. The matrix depends on the design of the hash function, although it is usually a *Maximum Distance Separable* (MDS) matrix [DL18] and the vector is the state of the sponge.

A *Maximum Distance Separable* matrix is a type of matrix that provides diffusion properties. It has the property that every square submatrix is invertible. This means that each output bit depends on all input bits. Thus, it ensures that small changes in input lead to significant changes in the output.

2.3 Finite fields

This section aims to provide and overview of finite fields and offer a brief introduction to the Goldilocks field and the BLS12-381 elliptic curve. As the focus of this thesis does not include elliptic curves and their associated mathematics, we limit it to essentials concepts. The definitions provided in this section will be used through the rest of the document.

Overview

A field $\mathbb{F}$ in mathematics is a set on which operations such as addition, multiplication, division and substraction are defined.

A finite field $\mathbb{F}_q$ is a field that contains a finite number of elements, $|\mathbb{F}_q| \in \mathbb{N}$. The number of elements in the field is also called *order*, and for a finite field of *order* q, it exists if $q = p^k$, where p is a prime number and $k \in \mathbb{N}$ [CD98].

Next we will focus on two types of finite fields:

- **Prime Fields**. They are constructed over a prime number of elements. The finite field $\mathbb{F}_p$, denotes the prime field, where p is a prime number. The elements of $\mathbb{F}_p$ range from 0 to $p-1$.
  There is an important property of prime field that makes them suitable for cryptography: anytime you perform a mathematical operation such as addition, multiplication, division or substraction with another element from the same prime field, it will lead to an element that also belongs to the prime field.
- **Binary Fields**. The elements of this field are binary polynomials with coeficients being 0 or 1. It is defined as $\mathbb{F}_{2^n}$, where n is an arbitrary integer and it constains 2^n elements, represented as n-bits string, where the degree of each polynomial is at most $n-1$.
  In modulo 2 arithmetics, $1+1 \equiv 0 \bmod 2, 1+0 \equiv 1 \bmod 2, 0+0 \equiv 0 \bmod 2$, so addition in $\mathbb{F}_{2^n}$ is a XOR operation.

2.3.1 Goldilocks field

The Goldilocks field is a prime field denoted as $\mathbb{F}_p$, where $p = 2^{64} - 2^{32} + 1$. Due to its specific form, this field allows for efficient arithmetic operations.

2.3.2 BLS12-381

BLS12-381 [BGM17] is a pairing-friendly elliptic curve. The field modulus q is a prime and has 383 bits or fewer, and the subgroup size p is also a prime and has 256 bits or fewer.

$$q = \texttt{0x1a0111ea397fe69a4b1ba7b6434bacd764774b84f38512bf}$$
$$\texttt{6730d2a0f6b0f6241eabfffeb153ffffb9feffffffffaaab}$$

$$p = \texttt{0x73eda753299d7d483339d80809a1d80553bda402f}$$
$$\texttt{ffe5bfeffffffff00000001}$$

CHAPTER 3

Proof Systems

3.1 Zero Knowledge Proofs

Overview

This concept was introduced by Goldwasser et al. [GMR85] in 1985, which provided a widely used definition of zero-knowledge protocols: "A zero-knowledge proof allows one party (the prover) to convince another party (the verifier) of the truth of a statement without disclosing any information other than the fact that the statement is indeed true".
To ensure this the prover constructs a new statement, known as a proof of knowledge, whose truth implies the truth of the original statement, without divulging any private information.

First we will explain Interactive proofs systems, required for explaining SNARKs, and their role in zero-knowledge proofs.

3.1.1 Interactive Proof System

An interactive proof system models an interaction between two parties: the prover and the verifier. The prover wants to convince the verifier of the truth of a statement, and the verifier aims to verify the statement without being deceived.

Interactive zero-knowledge proof systems are a subset of interactive proofs systems. All zero-knowledge proofs must satisfy the following three properties, where $\mathcal{L}$ is a language and x a public statement, and a true public statement is $x \in \mathcal{L}$ [For89]:

- **Completeness**. If the statement is true, and the prover possesses a valid proof, then the the verifier will be convinced of its truth. Thus, $\forall x \in \mathcal{L}$, the verifier will accept the proof.
- **Soundness**. If the statement is false, a dishonest prover P cannot convince an honest verifier except with a small probability. In other words, $\forall x \notin \mathcal{L}$, the honest verifier V cannot be convinced otherwise with probability $> 1/2^{|x|}$.
- **Zero-Knowledge**. The verifier learns only the truth or falsehood of the statement and gains no additional information. Thus, the statement is enough to convince the verifier that the prover knows the necessary knowledge.

The protocol involves a public parameter x belonging to the language $\mathcal{L}$, known to both the prover P and the verifier V, and a private parameter, the witness, w. The protocol proceeds through several steps:

1. The prover P sends a commitment of the private input (witness) to V.
2. The verifier V creates a random challenge and sends it to P.
3. The prover P sends to the verifier V the evaluated proofs of the challenge to V.
4. The verifier verifies these evaluations.

These interactions are conducted a sufficient number of times, decided by the verifier.

The verifier finally accepts the proof if the prover responded correctly to enough challenges, which is possible if the prover knows the private input.

We can see that there is a continuos interaction between the prover and the verifier. Next, non-interactive proof systems will be introduced in which the number of interactions between the prover and the verifier will be only one.

3.1.2 SNARKs

Now, we can define SNARKs, *succinct non-interactive argument of knowledge* [BFM19]. SNARKs can be used in large computations for providing integrity of the results [Nit20].

The correctness of some proofs need to be publicly verified, preferably by multiple parties. A way to overcome this problem is to make proofs non-interactive, thus,

public verifiable. In this solution, the challenges are created by the prover. So, continuous interaction between the prover and verifier is no longer necessary.

The proof system of the SNARK is:

- **succinct**. The size of the proof is small compared to the statement being proved, so we can generate a small proof of a large computation.
- **non-interactive**. There is no interaction between the prover and the verifier.
- **argument.** This property means that the proof convinces the verifier of the truth of the statement, assuming that the prover is computationally bounded (i.e., limited to polynomial-time computations). This allows us to relax some constraints compared to proofs of knowledge, as an exponentially powerful prover could potentially cheat. The key difference between an argument of knowledge and a proof of knowledge is that the former assumes the prover's computational limitations.
- **knowledge-sound.** The proof provided by the prover must be constructed using a witness associated with the statement being proved. So the proof not only demonstrate integrity of the statement but also knowledge of some secret information. In the context of NP statements[1], a polynomial-time prover cannot solve the problem unless it already knows a witness. Knowledge-soundness guarantees that if the prover can generate a valid proof, it must possess knowledge of a valid witness.

A zk-SNARK, *zero-knowledge succinct non-interactive arguments of knowledge*, can be used to generate proofs without revealing nothing more (witnesses) than the proof. A zk-SNARK is defined as follows:

1. Setup. A trusted third party (TTP) or a secure multi-party computation (MPC) [Gol98] generates a public common reference string S that will be used in the proof generation and verification.
2. Proof generation. Given S, the statement x and the witnesses w, the prover generates the proof π.
3. Proof verification. The verifier takes a verification key vk from S, the statement x and the proof π, determines if accept or reject the proof.

[1]A NP statement is a statement about a problem for which we don't know has an easy solution, but if we have a solution, we can easily verify its correctness.

3.1.3 Arithmetic circuits

Most zkSNARK proof systems operate over arithmetic circuits. This means that the computations must be "converted" into arithmetic circuits. These circuits are described using directed acyclic graphs whose vertices and edges represent gates and wires.

There is some similarity to boolean circuits. In boolean circuits, the operations are logical operations like AND, and OR, while on arithmetic circuits, the operations are addition and multiplication. Unlike boolean gates who operates on bits, arithmetic circuit operates on elements of the finite field $\mathbb{F}_p$.

Each gate of the circuit can be an input gate, an output gate, a constant gate, an addition gate or a multiplication gate. An example of an arithmetic circuit is provided in Figure 3.1. Let's consider an arithmetic circuit $\mathcal{G} : \mathbb{F}_p^3 \longrightarrow \mathbb{F}_p$, that computes the polynomial $f(x_1, x_2, x_3) := x_3 \cdot (x_1 + x_2)$.

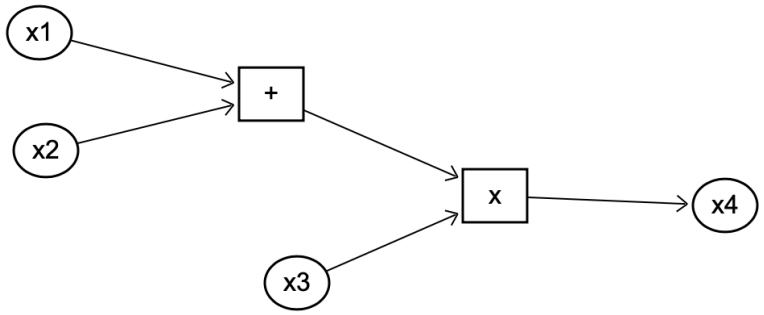

Figure 3.1: Arithmetic circuit for computing $x_4 := x_3 \cdot (x_1 + x_2)$

In this example, x_1, x_2 and x_3 are input gates, x_4 is an output gate and there is a multiplication and an addition gate.

3.2 PLONK

Overview

PLONK, *Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*, was introduced by Gabizon et al. in 2019 [GWC19]. It is a zkSNARK proving system and the one that will be used and explained in this thesis.

3.2.1 Constraint system

In [GWC19], a constraint system $\mathcal{C}$ over $\mathbb{F}_p$ is introduced, allowing the representation of arithmetic circuits. These circuits can contain gates with at most two inputs each, and output gates that can be connected to any number of other gates.
Let m represent the number of wires and n the number of gates, excluding private input/output gates. Let $\mathbf{x} \in \mathbb{F}_p^m$ be the wire assignment[2]. Let $\mathbf{a}, \mathbf{b}, \mathbf{c} \in [m]^n$ be the left, right and output vector sequence of each gate in the circuit $\mathcal{G}$, respectively. Additionally, let $\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C} \in \mathbb{F}_p^n$, where each $\mathbf{q}$ term represents a selector vector[3] in $\mathbb{F}_p$ used to define constraints based on the gate type.
Thus, the PLONK constraint at every gate $i \in [n]$ must satisfy

$$(\mathbf{q_L})_i \cdot \mathbf{x}_{\mathbf{a}_i} + (\mathbf{q_R})_i \cdot \mathbf{x}_{\mathbf{b}_i} + (\mathbf{q_O})_i \cdot \mathbf{x}_{\mathbf{c}_i} + (\mathbf{q_M})_i \cdot (\mathbf{x}_{\mathbf{a}_i}\mathbf{x}_{\mathbf{b}_i}) + (\mathbf{q_C})_i = 0 \tag{3.1}$$

For example, from the vector sequence $\mathbf{a}$, we set $\mathbf{a}_i := j$, where $j \in [m]$ is the index of the value $\mathbf{x}_j$ assigned to the left input of the i gate, in other words, for each gate i in the circuit, we are selecting a specific wire to be its left input.

To build a **multiplation gate** we set $(\mathbf{q_L})_i = 0$, $(\mathbf{q_R})_i = 0$, $(\mathbf{q_M})_i = 1$, $(\mathbf{q_O})_i = -1$, $(\mathbf{q_C})_i = 0$, and we get the following constraint for a gate i

$$\mathbf{x}_{\mathbf{a}_i}\mathbf{x}_{\mathbf{b}_i} - \mathbf{x}_{\mathbf{c}_i} = 0.$$

Similary, for an **addition gate**, we set $(\mathbf{q_L})_i = 1$, $(\mathbf{q_R})_i = 1$, $(\mathbf{q_M})_i = 0$, $(\mathbf{q_O})_i = -1$, $(\mathbf{q_C})_i = 0$, resulting in the constraint:

$$\mathbf{x}_{\mathbf{a}_i} + \mathbf{x}_{\mathbf{b}_i} - \mathbf{x}_{\mathbf{c}_i} = 0.$$

Lastly, for a **constant gate**, we set $(\mathbf{q_L})_i = 1$, $(\mathbf{q_R})_i = 0$, $(\mathbf{q_M})_i = 0$, $(\mathbf{q_O})_i = 0$, $(\mathbf{q_C})_i = -c_i$, and the corresponding constraint is

$$\mathbf{x}_{\mathbf{a}_i} - c_i = 0.$$

To summarize the initial proces of PLONK:

1. We build the arithmetic circuit based on the PLONK constraint of the program, where each operation of the circuit (addition, multiplication) corresponds to a gate in the circuit.

[2] A **wire assignment** referes to the assignment of specific values from the field $\mathbb{F}_p$ to the wires in an arithmetic circuit.

[3] A **selector vector** is a constant used to control the behavior of the arithmetic circuit based on the type of gate being evaluated.

2. The arithmetic circuit is translated into a constraint system, formed by all the contraints used to build the circuit.

Copy constraints

Up until now, we have been focused on how to build gates within the arithmetic circuit. However, we are not connecting them, we are missing the wiring. Copy constraints ensure that the outputs of some gates correctly map to the inputs of subsequent gates.

Example

Let's illustrate the PLONK protocol with an example. Consider the arithmetic circuit from Figure 3.2, which computes $v_4 := v_1 \cdot v_2 + v_3$. In this circuit, v_1, v_2, v_3 are private inputs (the witnesses), while v_4 is the public input. In this arithmetic

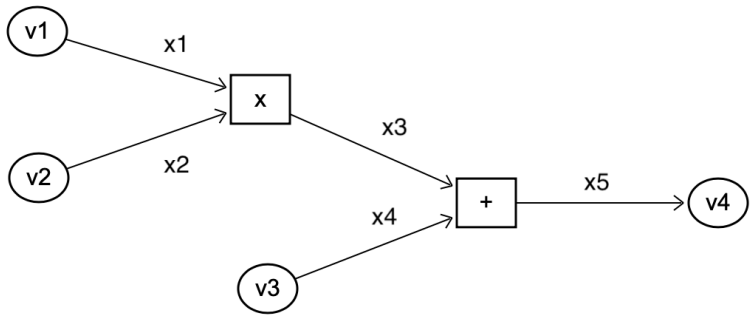

Figure 3.2: Arithmetic circuit for computing $v_4 := v_1 \cdot v_2 + v_3$

circuit we have 3 private input gates (v_1, v_2, v_3), $m = 5$ wires and $n = 3$ gates, let $i = 1$ be the multiplication gate, $i = 2$ the addition gate, and $i = 3$ the output gate, the sequence vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c} \in [5]^3$ are

$$\mathbf{a} := (1, 3, 5),\ \mathbf{b} := (2, 4, 5),\ \mathbf{c} := (3, 5, 5).$$

To make it more clear, let's demonstrate the process for gate $i = 2$ (addition gate). The left wire $x_3 = \mathbf{x}_{\mathbf{a}_2}$, where $\mathbf{a}_2$ denotes the index of the left input wire. Similary, the right input wire $x_4 = \mathbf{x}_{\mathbf{b}_2}$, and the output wire $x_5 = \mathbf{x}_{\mathbf{c}_2}$, and so on for each component of the vector sequence.
By assigning the corresponding values to the selector vectors, depending on the

type of gate as explained earlier, we get the following constraints:

$$\begin{aligned} x_1 \cdot x_2 - x_3 &= 0 \\ x_3 + x_4 - x_5 &= 0 \\ x_5 - v4 &= 0 \end{aligned}$$

Next, we need to establish the connections between different gates. In this case, it consists in connecting the multiplication gate with the addition gate and the addition with the output gate. Thus, we define the copy constraints for these cases:

$$\begin{aligned} a_2 &= c_1 \\ c_2 &= a_3 \end{aligned}$$

3.2.2 Compressing Constraints

There is a property about polynomials, called the *Schwartz-Zippel lemma* [Zip79, BGK+21] widely used in zero-knowledge proving systems, including Plonk. This lemma affirms that if two polynomials P and Q result in the same value when evaluated at some random point a, then it's highly probable that the two polynomials are identical.

To use this property we convert the constraint system into a polynomial expression that is true if the constraint system is satisfied.

Lagrange interpolation

Plonk uses Lagrange interpolation to convert the constraint system to a polynomial. We define polynomials $a(i), b(i), c(i)$ to interpolate the values in $\mathbf{x}$ with the vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $Q_L(i), Q_R(i), Q_O(i), Q_M(i), Q_C(i)$ to interpolate $\mathbf{q_L}, \mathbf{q_R}, \mathbf{q_O}, \mathbf{q_M}, \mathbf{q_C}$ for each i.
Now, using Equation (3.1) we express the constraint as a polynomial:

$$Q_{\mathrm{L}}(i)a(i) + Q_{\mathrm{R}}(i)b(i) + Q_{\mathrm{O}}(i)c(i) + Q_{\mathrm{M}}a(i)b(i) + Q_{\mathrm{C}}(i) = 0 \tag{3.2}$$

3.2.3 Polynomial commitments

A polynomial commitment can be used to verify evaluations of a polynomial without needing access to entire polynomial itself. In other words, let's define a commitment c representing the polynomial $P(x)$. There is a proof that can convince you of the value of $P(z)$ for some point z, without revealing the polynomial $P(x)$ itself.

The mathematical details of the polynomial commitment scheme is out of the scope of this thesis, requiring the exclusion of certain details. For a more detailed explanation look at [KZG10].

Intuitively, there are some equations between the polynomials that need to be checked. In Plonk, the prover need to commit to multiple polynomials. Although, the prover and the verifier could handle all the commitments one by one, in [GWC19], these polynomials are commited within a single step. These commitments are evaluated at some random point z, along with proofs indicating that these evaluations are correct. Finally, the equations are applied to these evaluations rather than the original polynomials.
Thus, the final proof is just some commitments and the evaluations, and can be checked with a few equations.

CHAPTER 4

State of the Art on Zero-Knowledge Friendly Hash Functions

This chapter will provide a theoretical approach of the hash functions that will be implemented.

Overview

Most of the hash functions that work in zero-knowledge operate within a finite field, with their efficiency usually related to the number of multiplications and additions in the circuit. Traditional hash functions such as SHA-256 or AES are designed to operate on bits and perform bitwise operations. However, when these bit-oriented hash functions are used in zero-knowledge proofs, they often result in very large circuit sizes, reducing efficiency. This inefficiency is due to the need for representing bitwise operations of long bit-size numbers, which usually involve numerous multiplications and additions within the circuit.

To address these issues, Arithmetization-Oriented (AO) hash functions have been developed. These hash functions are designed specifically for zero-knowledge proofs and operate within a finite field. By focusing on arithmetic operations rather than bitwise operations, AO hash functions simplify the circuit complexity. Consequently, they enable more efficient computations in zero-knowledge proofs.

Moreover, some techniques will be introduced to minimize the number of multiplications in the zero-knowledge circuits for better efficiency.

4.1 MiMC

The first hash funcion we explain is MiMC [AGR+16], it was publiched by Albrecht et al. in 2016 and designed to provide high performance for the applications of zero-knowledge proofs, Fully-Homomorphic Encryption (FHE) and Multi-Party-Computation (MPC) frameworks. This hash function is very simple, its core non-linear component is the function $F(x) := x^3$ that is iterated with subkey additions. The advantatge of using this function is that only requires two field multiplications, in contrast to using a look-up table S-box, which can be expensive to represent in an arithmetic circuit, for example.
MiMC provides three cryptographic primitives. First, we are going to describe the block cipher and finally a permutation. We consider MiMC just for cipher/permutation, and GMiMC [AGP+19] or Hades [GLR+20] for hashing. All these primitives work in a finite field $\mathbb{F}_q$, where q is a prime p or a power of 2. In this thesis we will focus on $q = p$.

Block cipher

First of all we are going to explain the block cipher, MiMC-b/k is a keyed permutation where each input block is maped to a unique output block with b as block size and k as key size.

MiMC-p/p

The MiMC block cipher takes an input $x \in F_p$ and does r iterations over a round function. The round function consists on a key addition of the k vector, the addition of a round constant $c_i \in F_p$ and the previous discussed non-linear function $F(x) := x^d$ for $x \in F_p$ is applied for updating the current sate. Additionally, we need to assure that $d \geq 3$ is the smallest permutation, so $\gcd(d, p-1) = 1$, explained in section 2.2.1.
The encryption function consists in r iterations over the round functions, adding the key k after r rounds have been performed.
So, the round function is defined as

$$F_i(x) = F\left(x \oplus k \oplus c_i\right) \tag{4.1}$$

and the encryption function is expressed as

$$E_k(x) = \left(F_{r-1} \circ F_{r-2} \circ \ldots F_0\right)(x) \oplus k \tag{4.2}$$

Figure 4.1 represents the encryption process of MiMC for $d = 3$.

The equation for calculating the number of rounds is

$$r = \left\lceil \frac{p}{\log_2 3} \right\rceil \tag{4.3}$$

The values of the $r-1$ round constants can be choosen randomly from the field $\mathbb{F}_p$ except for the first c_0 and last c_r round constant whose values are 0.

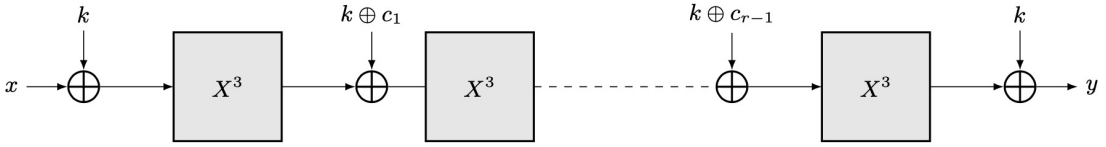

Figure 4.1: Encryption process of MiMC [AGR+16] for $d = 3$

MiMC-2p/p

In addition to the standard version of MiMC, there are some other variants, including those using a Feistel network, with extended key size, and using different round functions. We focus on the **Feistel network** variant over prime fields, MiMC-2p/p. By using the Feistel network in the block cipher, we can process larger blocks, with two blocks being processed in each round.
The new round function of MiMC-2p/p differs from the one in MiMC-p/p, taking the form [AGR+16]:

$$x_L \| x_R \longleftarrow x_R \oplus (x_L \oplus k \oplus c_i)^3 \| x_L \tag{4.4}$$

However, the encryption function is the same as in MiMC-n/n.
In the Feistel network the round constants c_i are also randomly generated from the $\mathbb{F}_p$ field. The cost for processing larger blocks comes with the incress of the number of rounds, where

$$r' = 2 \cdot r = 2 \cdot \left\lceil \frac{p}{\log_2 3} \right\rceil \tag{4.5}$$

Here, r is the number of rounds of MiMC-p/p. Figure 4.2 represents the encryption process of MiMC-2p/p. Because there is one more XOR operation and the number of rounds needs to be doubled, the computation of the encryption will be slightly slower than the regular MiMC.

MiMC permutation

The MiMC permutation, MiMC^P, is obtained by setting all the key k values to 0.

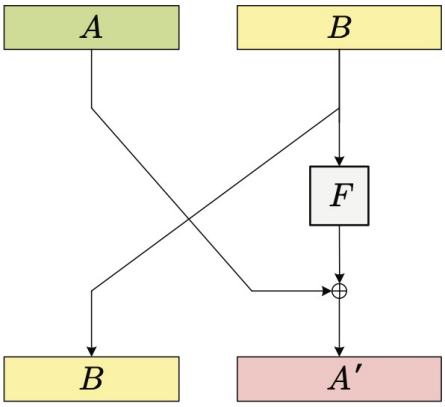

Figure 4.2: Feistel network

4.2 Poseidon

The Poseidon hash function [GKR+21], published by Grassi et al. in 2021 works using the Poseidon$^{\pi}$ permutation within the sponge framework. This permutation is based on the Hades design, which will be explained below. The hash function takes inputs from the $\mathbb{F}_p$ field, where p is a prime number, and maps them to a fixed-length output over the $\mathbb{F}_p$ field, $\mathbb{F}_p^* \longrightarrow \mathbb{F}_p^o$, with o representing the length of the output.

Hades strategy

The Hades strategy doesn't use r number of rounds, instead, it uses r_f rounds initially, with full S-box layers. Subsequently, after these r_f rounds, r_p rounds are applied, but with only a single S-box. Finally r_f rounds are applied again at the end, using full S-box layers.

Figure 4.3 represents this explained aproach.

Poseidon round function

The Poseidon round function is composed of two different rounds: full rounds and partial rounds. The full round $\mathcal{E}$ is defined as

$$\mathcal{E}_i(x) = M \cdot \left(\left(x_0 + c_0^{(i)} \right)^d, \ldots, \left(x_{t-1} + c_{t-1}^{(i)} \right)^d \right) \tag{4.6}$$

for $i \in \{0, 1, \ldots, R_{r_f} - 1\}$. The partial round $\mathcal{I}$ is defined as

$$\mathcal{I}_i(x) = M \cdot \left(\left(x_0 + c_0^{(i)} \right)^d, x_1 + c_1^{(i)}, \ldots, x_{t-1} + c_{t-c}^{(i)} \right) \tag{4.7}$$

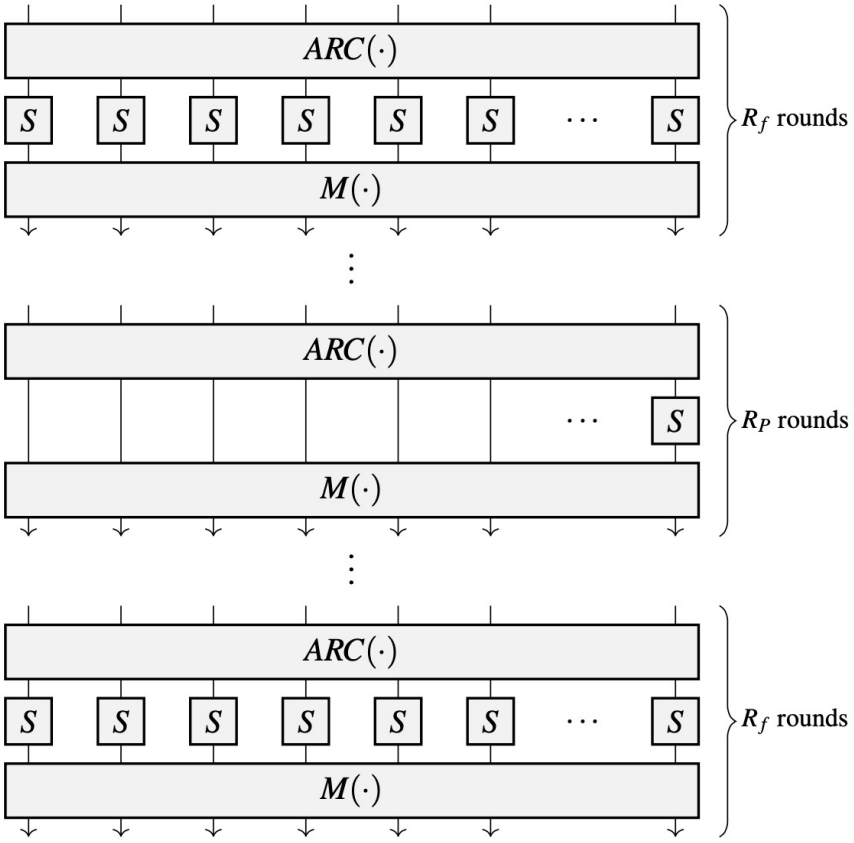

Figure 4.3: Encryption process of Hades [GKR+21]

for $i \in \{0, 1, \ldots, R_{r_p} - 1\}$.

Table 4.1 represents the three steps of one round of Poseidon$^{\pi}$

	Hades round function
1	Add the round constants, denotated by $ARC(\cdot)$
2	Apply substitution i. Full S-box$(\cdot)$ for r_f rounds ii. Partial S-box$(\cdot)$ for r_p rounds
3	Multiply state with the MDS matrix, denotated by $M(\cdot)$

Table 4.1: One round of the Poseion$^{\pi}$

Next, we will describe the components of the Poseidon round function.

Constant layer

The ARC($\cdot$) consists in adding the round constants c_i to the state, where each component of the state will have a round constant added.

S-box layer

The s-box layer is defined as $\text{Sbox}(x) = x^d$, where $d \geq 3$ is the smallest integer that satisfies $gcd(d, p-1) = 1$. For rounds without an S-box (r_p rounds), the state remains unchanged and is forwarded as it is to the next round.

Linear layer

The $t \times t$ Matrix with elements in $\mathbb{F}_p$ is a Maximum Distance Separable (MDS) matrix, which guarantees that each state component depends on each other component in each round. All t components of the state will be multiplied by the $t \times t$ MDS matrix. The result of this multiplications is the input state for the next round layer. Unlike the Hades strategy, the linear layer is omitted in the last round.

Poseidon permutation

The Poseidon$^\pi$ permutation over the vector space $\mathbb{F}_p^t$, where $t \geq 2$ is the length of the sponge state, is defined by

$$\mathcal{P}(x) = \mathcal{E}_{R_F-1} \circ \cdots \circ \mathcal{E}_{R_F/2} \circ \mathcal{I}_{R_P-1} \circ \cdots \circ \mathcal{I}_0 \circ \mathcal{E}_{R_F/2-1} \circ \cdots \circ \mathcal{E}_0(x), \tag{4.8}$$

where $\mathcal{E}$ represents a full round of the R_F rounds and $\mathcal{I}$ is a partial round of r_P rounds, with $R_F = 2 \cdot r_f$.

Poseidon hash

The Poseidon hash operates within the sponge construction, in the vector space $\mathbb{F}_p^{r+c}$, where r is the rate and c is the capacity of the sponge.

4.3 Poseidon2

The Poseidon2 hash function [GKS23] published by Grassi et al. in 2023, works within the sponge construction, being a more efficient version of Poseidon$^\pi$. Below, we explain the Poseidon2 hash, the Poseidon2 permutation and his round function.

Poseidon2 round function

Similar to Poseidon, the Poseidon2 round function consists on two different rounds, a full round and a partial round.
The full round, $\mathcal{E}$, is defined as

$$\mathcal{E}_i(x) = M_{\mathcal{E}} \cdot \left(\left(x_0 + c_0^{(i)}\right)^d, \left(x_1 + c_1^{(i)}\right)^d, \ldots, \left(x_{t-1} + c_{t-c}^{(i)}\right)^d \right) \tag{4.9}$$

where d is the exponent of the S-box, and $c_j^{(i)}$ the j-th round constant in the i-th full round.
The partial round $\mathcal{I}$ is defined as

$$\mathcal{I}_i(x) = M_{\mathcal{I}} \cdot \left(\left(x_0 + \hat{c_0}^{(i)}\right)^d, x_1, \ldots, (x_{t-1}) \right) \tag{4.10}$$

where d is the same exponent as before, and $\hat{c}_0^{(i)}$ is the round constant in the i-th partial round.
A description of the components used in the Poseidon2 round function is provided below.

Constant layer

The round constants are generated as in Poseidon$^\pi$.

Linear layer

Firstly, the case of $t = 4t' \geq 4$ is considered, followed by $t \in \{2, 3\}$. For the **full rounds** and $t = 4t'$ with $t' \in \mathbb{N}$, the matrix is defined as

$$M_{\mathcal{E}} = \begin{cases} M_4 & \text{if } t = 4, \\ \text{circ}\left(2 \cdot M_4, M_4, \ldots, M_4\right) \in \mathbb{F}_p^{t \times t} & \text{if } t \geq 8, \end{cases} \tag{4.11}$$

where M_4 is a 4×4 MDS matrix

$$M_4 = \begin{pmatrix} 5 & 7 & 1 & 3 \\ 4 & 6 & 1 & 1 \\ 1 & 3 & 5 & 7 \\ 1 & 1 & 4 & 6 \end{pmatrix}, \tag{4.12}$$

An efficient way for computing vector per matrix multiplication is defined in [DL18]. Corresponding to the $M_{4,4}^{8,4}$ matrix, with $\alpha = 2$.

For the **partial rounds**, the MDS matrix is no longer necessary. In this case, the matrix $M_{\mathcal{I}}$ is defined as

$$M_{\mathcal{I}} = \begin{pmatrix} \mu_0 & 1 & \dots & 1 \\ 1 & \mu_1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & \mu_{t-1} \end{pmatrix}, \tag{4.13}$$

where $(\mu_0, \mu_1, \dots, \mu_{t-1})$ are random elements from the field $\mathbb{F}_p \backslash \{0, 1\}$.

Plonk Arithmetization

For the partial round vector per matrix multiplication. Let's define a vector $(x_0, x_1, \dots x_{t-1})$, the computation of the multiplication is

$$\begin{cases} s = x_0, x_1, \dots, x_{t-1}, \\ y_i = (\mu_i - 1)\, x_i + s \quad \text{for } i \in \{0, 1, \dots, t-1\}, \end{cases} \tag{4.14}$$

where s represents the sum of the elements of the vector and y is the output vector. Additionally, for $t \in \{2, 3\}$, we set the condition that $M_{\mathcal{I}}$ is MDS, so, first $M_{\mathcal{I}}$ is computed and we set $M_{\mathcal{I}} = M_{\mathcal{E}}$. Appart from this we also check for

- $t = 2$ if $\mu_0\mu_1 - 1 \neq 0$ and $\mu_0, \mu_1 \neq 0$,
- $t = 3$ if $\mu_0\mu_1\mu_2 - \mu_0 - \mu_1 - \mu_2 \neq 0$ and $\mu_0\mu_1\mu_2 \neq 0$, $\mu_0\mu_1 - 1 \neq 0$, $\mu_0\mu_2 - 1 \neq 0$, $\mu_1\mu_2 \neq 0$.

S-box layer

The S-box layer is defined as $\text{Sbox}(x) = x^d$, where $d \geq 3$ is the smallest positive integer such that $\gcd(d, p-1) = 1$.

Poseidon2 permutation

The permutation is defined over the $\mathbb{F}_p^t$ vector space, where p is a prime number and $t \in \{2, 3, 4, \dots, 4 \cdot t', \dots, 24\}$ is the length of the state of the sponge for $t' \in \mathbb{N}$. The Poseidon2$^\pi$ is defined as

$$\mathcal{P}_2(x) = \mathcal{E}_{R_F-1} \circ \cdots \circ \mathcal{E}_{R_F/2} \circ \mathcal{I}_{R_P-1} \circ \cdots \circ \mathcal{I}_0 \circ \mathcal{E}_{R_F/2-1} \circ \cdots \circ \mathcal{E}_0(M_{\mathcal{E}}.x) \tag{4.15}$$

where $\mathcal{E}$ is the full round and $\mathcal{I}$ is the partial round, whith the same number of rounds as in Equation 4.2 of Poseidon$^\pi$.

Poseidon2 hash

Similar to the other hash functions presented in this thesis, the Poseidon2 hash also operates within a sponge construction in the field $\mathbb{F}_p^{r+c}$, where r is the rate and c the capacity of the sponge.

Poseidon$^\pi$ vs Poseidon2$^\pi$

The differences between the permutation of Poseidon2 and Poseidon are provided below. Figure 4.4 shows a visual representation of these difference.

- The linear layer is also applied at the input of the permutation.
- Two diferent linear layers are used for $t \geq 4$.
- In each round only one round function is applied.

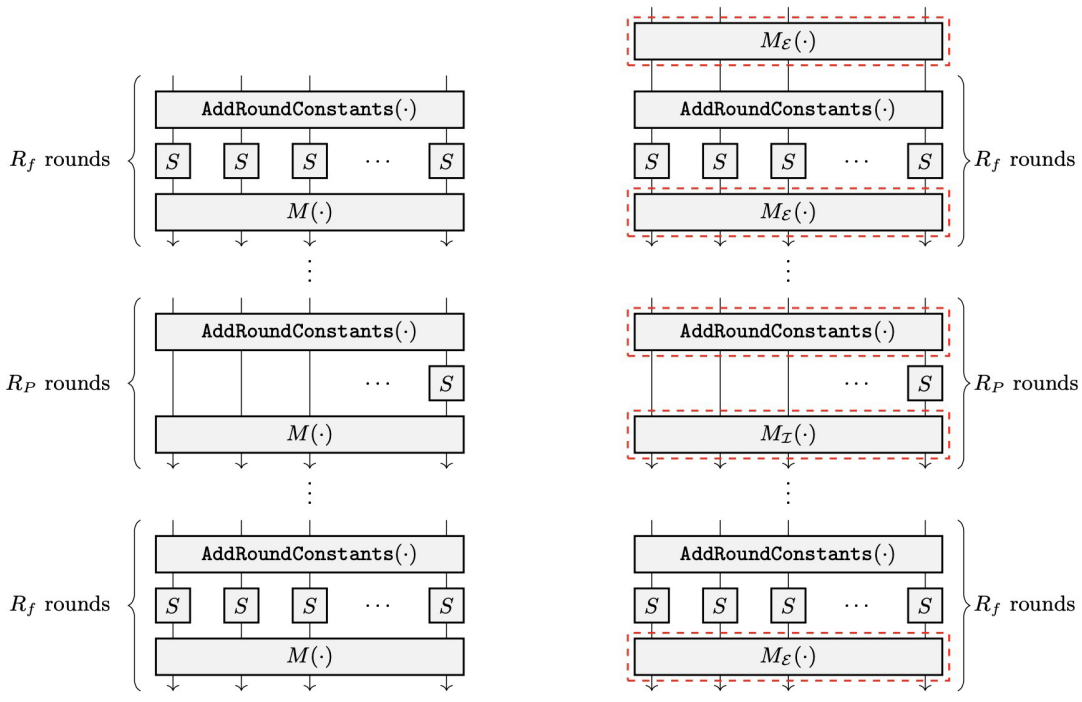

Figure 4.4: Poseidon$^\pi$ vs Poseidon2$^\pi$ [GKS23]

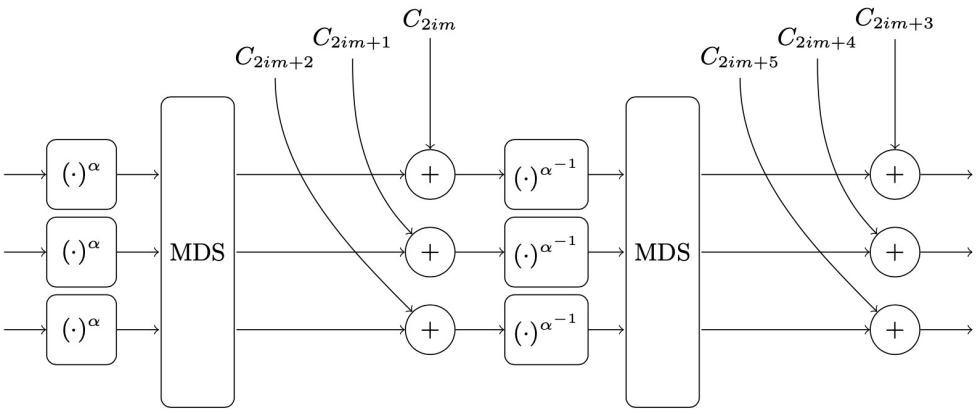

Figure 4.5: Round i of Rescue$^{\pi}$ [SAD20]

4.4 Rescue-prime

Rescue [SAD20] published by Szepieniec et al. in 2020, operates on field elements on $\mathbb{F}_p$, where p is a prime field, and the state is a vector in the vector space $\mathbb{F}_p^m$, where m is the number of elements. Here, we will explain both the standard and the optimized versions of Rescue-prime.

Rescue round function

The Rescue round function [AABS+20] operates over the vector space $\mathbb{F}_p^m$, where p is a prime number. Below, Table 4.2 presents the components and steps of a single round i, while Figure 4.5 provides a diagram of it.

	Rescue round function
1	S-box layer, apply $(\cdot)$ to each element of the state.
2	Linear layer, matrix-vector multiplication of the MDS matrix and the state.
3	Constants layer, add from the list of round constants $\{C_i\}_{i=0}^{2mN-1}$, m constants to the state.
4	Inverse S-box layer, apply $(\cdot)^{\alpha^{-1}}$ to each element of the state.
5	Liner layer, matrix-vector multiplication of the MDS matrix and the state.
6	Constants layer, add from the list of round constants $\{C_i\}_{i=0}^{2mN-1}$, m constants to the state.

Table 4.2: One round components

To build the S-box, we find the smallest prime α such that $gcd(p-1, \alpha) = 1$. Therefore, the S-box is defined as $S : \mathbb{F}_p \longrightarrow \mathbb{F}_p : x \mapsto x^{\alpha}$ and $S^{-1} : \mathbb{F}_p \longrightarrow \mathbb{F}_p :$

$x \mapsto x^{1/\alpha}$, where $1/\alpha$ is the multplicative inverse of a modulo $p-1$, whose existance is guaranteed by the fact that $\gcd(\alpha, p-1) = 1$. The generation of the MDS matrix and the round constants is out of the scope of this thesis, although [SAD20] provides a SageMath implementation with the computation of the number of rounds and the MDS matrix.

Rescue-prime optimized

The optimized version of the Rescue-prime [AKM+22] published in 2022, works only over the Goldilocks prime field, where $p = 2^{64} - 2^{32} + 1$, explained in section 2.3.

It uses the same components of the standard Rescue-prime round function, but the order of the operations differs from the standard version. The new order is presented in Table 4.3.

	Rescue-prime optimized round function
1	MDS matrix
2	Constant layer
3	S-box layer
4	MDS matrix
5	Constant layer
6	Inverse S-box layer

Table 4.3: Rescue-prime optimized single round components

Parameters

In the optimized version, certain parameters are fixed. The exponents of the S-box and the inverse S-box, α and α^{-1}, are set to $\alpha = 7$ and $\alpha^{-1} = 10540996611094048183$. Table 4.4 provides additional parameters.

Prime field p	$2^{64} - 2^{32} + 1$
Number of rounds N	7
State size m	12
Rate r	8
Capacity c	4

Table 4.4: Integer parameters for the Rescue-prime optimization

MDS Matrix

The MDS matrix is a circulant matrix with the first row defined as:

$$[7, 23, 8, 26, 13, 10, 9, 7, 6, 22, 21, 8]$$

Overwrite mode

In contrast to the standard Sponge construction, the elements from the state associated with the rate are added by the matching elements from the input chunk, but in this optimized case, the elements are not added but they are overwritten. In other words, if the $state[j]$ represents the state elements and $input[j]$ the input chunk, it will permorm this absorbing operation $state[j] \leftarrow input[j]$ instead of $state[j] \leftarrow state[j] + input[j]$

Indexing of state elements

In the optimized version, the state is composed by the capacity and the rate, where each one indices from 0 to $c - 1$ and c to $m - 1$, respectively.

Rescue-prime permutation

The Rescue-prime permutation consists in iterating N times the round function.

Rescue-prime hash

Both the standard and optimized versions of Rescue-prime utilize the Sponge construction for hashing.

Sponge construction

There is a modification in the squeezing phase of the sponge construction and the one explained in section 2.2. In the squeezing phase the top r elements of the state are the output, note that we don't apply the permutation in this phase.

Figure 4.6 demonstrate this modification for $N = 2$.

4.5 Griffin

We are going to explain the Griffin hash function [GHR+22], proposed by Grassi et al. in 2022. It is a hash function that operates over the permutation, Griffin$^{\pi}$, and a round function.

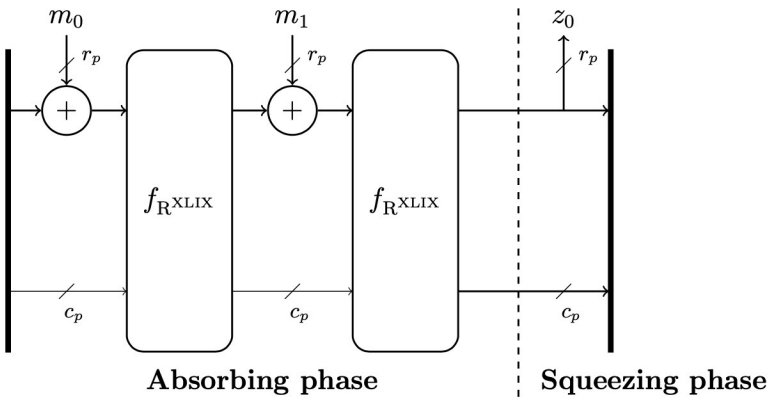

Figure 4.6: Rescue-prime hash function with $N = 2$ [SAD20]

Griffin round function

The Griffin round function, denoted as $\mathcal{F}_i$, is defined as

$$\mathcal{F}_i(\cdot) = c^{(i)} + \mathcal{M} \times \mathcal{S}(\cdot). \tag{4.16}$$

Here, $c^{(i)} \in \mathbb{F}_p^t$ is the round constant, where $c^{R-1} = 0$, $\mathcal{S} : \mathbb{F}_p^t \longrightarrow \mathbb{F}_p^t$ is a nonlinear layer, and $i \in \{0, 1, \ldots, R-1\}$ with R being the number of rounds. The round constants that are added to the state will be different in every round but the matrix M that is applied to the state will be the same in every round.
Each one of the components of the round function will be explained next.

Nonlinear layer *S*

The permutation is limited in $t \leq 24$, with d taking values from $\{3, 5, 7, 11\}$, similar to the previous hash functions. These values are choosen such that $gcd(d, p-1) = 1$. Additionally, α_i and $\beta_i \in \mathbb{F}_p^2 \backslash \{(0,0)\}$ are a pairwise distinct, ensuring that $\alpha_i^2 - 4\beta_i$ is a quadratic nonresidu modulo p for $2 \leq i \leq t-1$.
The nonlinear layer $\mathcal{S}(x_0, \ldots, x_{t-1}) = y_0 \| \cdots \| y_{t-1}$ is composed of two nonlinear sublayers defined via three different nonlinear functions. Two of them are defined via the invertible power maps $x \mapsto x^d$ and $x \mapsto x^{1/d}$, inspired by Rescue. The final one uses the Horst scheme (Section 2.1.1), using the map $(x, y) \mapsto (x, y \cdot G(x))$ for a quadratic function G such that $G(z) \neq 0$ for each z. It is defined as:

$$y_i = \begin{cases} x_0^{1/d} & \text{if i } = 0, \\ x_1^d & \text{if i } = 1, \\ x_2 \cdot \left(\left(L_i \left(y_0, y_1, 0 \right) \right)^2 + \alpha_2 \cdot L_i \left(y_0, y_1, 0 \right) + \beta_2 \right) & \text{if i } = 2, \\ x_i \cdot \left(\left(L_i \left(y_0, y_1, x_{i-1} \right) \right)^2 + \alpha_i \cdot L_i \left(y_0, y_1, x_{i-1} \right) + \beta_i \right) & \text{otherwise,} \end{cases} \tag{4.17}$$

where $L_i : \mathbb{F}_p^3 \longrightarrow \mathbb{F}_p$ represents $L_i(z_0, z_1, z_2) = \gamma_i \cdot z_0 + z_1 + z_2$ for $\gamma_i \in \mathbb{F}_p \backslash \{0\}$.

Linear layer M
For $t \in \{3, 4\}$, the matrix M must be MDS,

$$M_3 = \begin{pmatrix} 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 1 & 2 \end{pmatrix}, M_4 = \begin{pmatrix} 5 & 7 & 1 & 3 \\ 4 & 6 & 1 & 1 \\ 1 & 3 & 5 & 7 \\ 1 & 1 & 4 & 6 \end{pmatrix}, \tag{4.18}$$

we can use the efficient method proposed in [DL18] to compute the matrix per vector multiplication, where the M_4 corresponds to $M_{4,4}^{8,4}$, setting $\alpha = 2$.

When $t \geq 8$, M is

$$M = M'' \times M' \equiv M' \times M'' = \begin{pmatrix} 2 \cdot M_4 & M_4 & \dots & M_4 \\ M_4 & 2 \cdot M_4 & \dots & M_4 \\ \vdots & \vdots & \ddots & \vdots \\ M_4 & M_4 & \dots & 2 \cdot M_4 \end{pmatrix}, \tag{4.19}$$

where $M' = \text{diag}(M_4, M_4, \dots, M_4) \in \mathbb{F}_p^{t \times t}$ and $M'' = \text{circ}\big(2 \cdot I, I, \dots, i \in \mathbb{F}_p^{t \times t}\big)$ and M_4 is a 4×4 MDS matrix and I is the 4×4 identity matrix.

Griffin permutation

The Griffin permutation, denotated as $\mathcal{G}^\pi : \mathbb{F}_p^t \longrightarrow \mathbb{F}_p^t$ is defined as

$$\mathcal{G}^\pi(\cdot) := \mathcal{F}_{\mathcal{R}-1} \circ \mathcal{F}_1 \circ \mathcal{F}_0(\mathcal{M} \times \cdot), \tag{4.20}$$

where $\mathcal{M}$ is a matrix defined as above, and $\mathcal{F}_i : \mathbb{F}_p^t \longrightarrow \mathbb{F}_p^t$ represents the round function of the $\mathcal{G}^\pi$ permutation.
Figure 4.8 shows the $\mathcal{G}^\pi$ permutation, where $\boxplus$ represents the addition opeeration of two vectors in the field $\mathbb{F}_p^t$.

Griffin-Sponge hash function

The Griffin hash function operates over a field $\mathbb{F}_p$, where p is a prime number and $t \in \{3, 4t'\}$ the length of the state for a positive integer $t' \in \{1, 2, \dots, 6\}$, that means that t must be 3 or a multiple of 4. Additionally, due to the use of the sponge construction, the rate r must satisfy $r \geq t/3$.
Figure 4.7 provides the Griffin-Sponge hash function, where $\oplus$ represents the addition operation of two vectors in the field $\mathbb{F}_p^r$ and $\mathcal{G}^\pi$ the Griffin permutation.

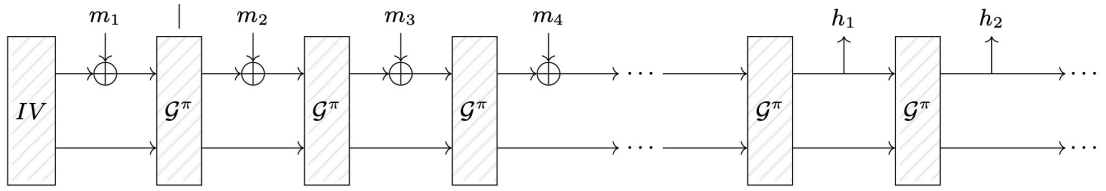

Figure 4.7: Griffin-Sponge hash function [GHR+22]

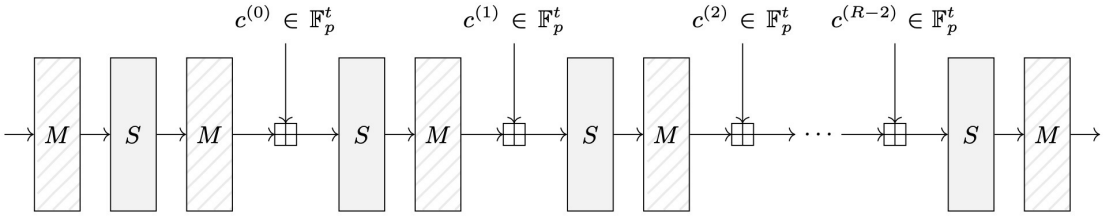

Figure 4.8: $\mathcal{G}^\pi$ permutation [GHR+22]

4.6 Anemoi

The Anemoi hash function [BBC+23] is also a very recent hash function, it was published by Bouvier et al. in 2023. It operates over a permutation, Anemoi$^\pi$, and this permutation works on a round function.
First we will start presenting the round function, next the permutation and finally the hash function.

Anemoi round function

The Anemoi round function operates over the vector space $\mathbb{F}_q^{2l}$, where $l > 0$, and q is a prime number p or a power of 2, in this thesis, we focus on $q = p$.

The length of the state is $2 \times l$. In the Anemoi, the state is divided in two parts, the first part is $(x_0, \ldots, x_{l-1})$ and the second part is defined as $(y_0, \ldots, y_{l-1})$. We define a generator g of the multiplicative subgrup of the field $\mathbb{F}_q$. If q is prime, g is the smallest generator, otherwiese g is one of the roots of $\mathbb{F}_q = \mathbb{F}_{2^n} = \mathbb{F}_2[x]/p(x)$, where p is an irreducible polynomial of degree n.

The structure of the Anemoi round function is composed of the following components: linear layer, S-box layer and constant addition. Figure 4.9 represents this structure.

Next, a description of each component is presented below.

Constant layer $\mathcal{A}$

The constants layer corresponds to the addition of constants to the state vector, $x_j \leftarrow x_j + c_j^i$ and $y_j \leftarrow y_j + d_j^i$ where $c_j^i \in \mathbb{F}_q$ and $d_j^i \in \mathbb{F}_q$ are the round constants that depends on the position i and the round r of the permutation.
For calculating the round constants, we use

$$\pi_0 = 14159265358979323846264338327950288419716939937510582097494459230781640628620899862803482534211706 79,$$

$$\pi_1 = 82148086513282306647093844609550582231725359408128481117450284102701938521105559644622948954930381 96,$$

where π_1 and π_2 are the decimal expansion of π, and the computations for calculating the round constants c_j^i and d_j^i are

$$c_j^i = g\left(\pi_0^i\right)^2 + \left(\pi_0^i + \pi_1^j\right)^\alpha \tag{4.21}$$

$$d_j^i = g\left(\pi_1^j\right)^2 + \left(\pi_0^i + \pi_1^j\right)^\alpha + g^{-1}, \tag{4.22}$$

in the $\mathbb{F}_q$ field.

Linear layer $\mathcal{M}$

In the Anemoi, the matrix M operates on the two vectors of the state $X = (x_0, \ldots, x_{i-1})$ and $Y = (y_0, \ldots, y_{i-1})$, separately. The matrix M is defined as

$$\mathcal{M}(X, Y) = \left(\mathcal{M}_x(X), \mathcal{M}_y(Y)\right), \tag{4.23}$$

$\mathcal{M}_x$ operates on the X elements of the state and $\mathcal{M}_y$ on the Y elements.
We define $\mathcal{M}_x$ as a $l \times l$ matrix of $\mathbb{F}_q$ and $\mathcal{M}_y = \mathcal{M}_x \circ \rho$, where ρ is a permutation over the X vector defined as $\rho(x_0, \ldots, x_{l-1}) = (x_1, \ldots x_{l-1}, x_0)$. The matrix $\mathcal{M}_x$ depends on the value of l. Depending on the l size we separate two possible situations.

- l is small, then the field size needs to be large.
- If l is large, we expect a smallest field.

Similar to the previous hash functions, when $l \leq 4$ we can optimize the matrix and vector multiplications using the approach described in [DL18].
In this cases, $l \in \{2, 3, 4\}$, the matrix $\mathcal{M}_x^l$ is

$$\mathcal{M}_x^2 = \begin{pmatrix} 1 & g \\ g & g^2+1 \end{pmatrix}, \ \mathcal{M}_x^3 = \begin{pmatrix} g+1 & 1 & g+1 \\ 1 & 1 & g \\ g & 1 & 1 \end{pmatrix}, \tag{4.24}$$

$$\mathcal{M}_x^4 = \begin{pmatrix} 1 & g^2 & g^2 & 1+g \\ 1+g & g+g^2 & g^2 & 1+2g \\ g & 1+g & 1 & g \\ g & 1+2g & 1+g & 1+g \end{pmatrix}. \tag{4.25}$$

Pseudo-Hadamard transform $\mathcal{P}$

Anemoi uses a new layer that has not been used in the previous hash functions, it is used to increase the security after computing the S-box layer. It is very simple to implement, only needing two additions: $Y \leftarrow Y + X$ and $X \leftarrow X + Y$.

Nonlinear layer $\mathcal{S}$

The nonlinear or S-box layer of the Anemoi also works different than in the previous hash functions. First we define the S-box as

$$\mathcal{S}(X, Y) = \left(\mathcal{H}(x_0, y_0), \ldots, \mathcal{H}\left(x_{l-1, y_{l-1}}\right)\right), \tag{4.26}$$

where $\mathcal{H}$ is an *open Flystel* (Section 2.1.1) operating over $\mathbb{F}_q^2$. $\mathcal{H}$ uses 4 parameters, the exponent α, the multiplier β, and the two constants γ and δ.
So, we let $\beta = g$, where g has been defined before, $\delta \neq \gamma$, $\gamma = 0$ and $\delta = g^{-1}$, and finally the exponent α, that must satisfy $\gcd(p-1, \alpha) = 1$.

$$\mathcal{H} = \begin{cases} x_i \longleftarrow x_i - gQ(y_i) - g^{-1} \\ y_i \longleftarrow y_i - x_i^{1/\alpha} \\ x_i \longleftarrow x_i + Q(y_i) \end{cases} \tag{4.27}$$

Anemoi permutation

The Anemoi permutation, Anemoi$^\pi$ iterates n_r rounds over the round function previously explained, with a linear layer $\mathcal{M}$ when finishing the last round. It is defined as

$$\text{Anemoi}_{q,\alpha,l}^\pi = \mathcal{M} \circ \text{R}_{n_r-1} \circ \ldots \text{R}_0. \tag{4.28}$$

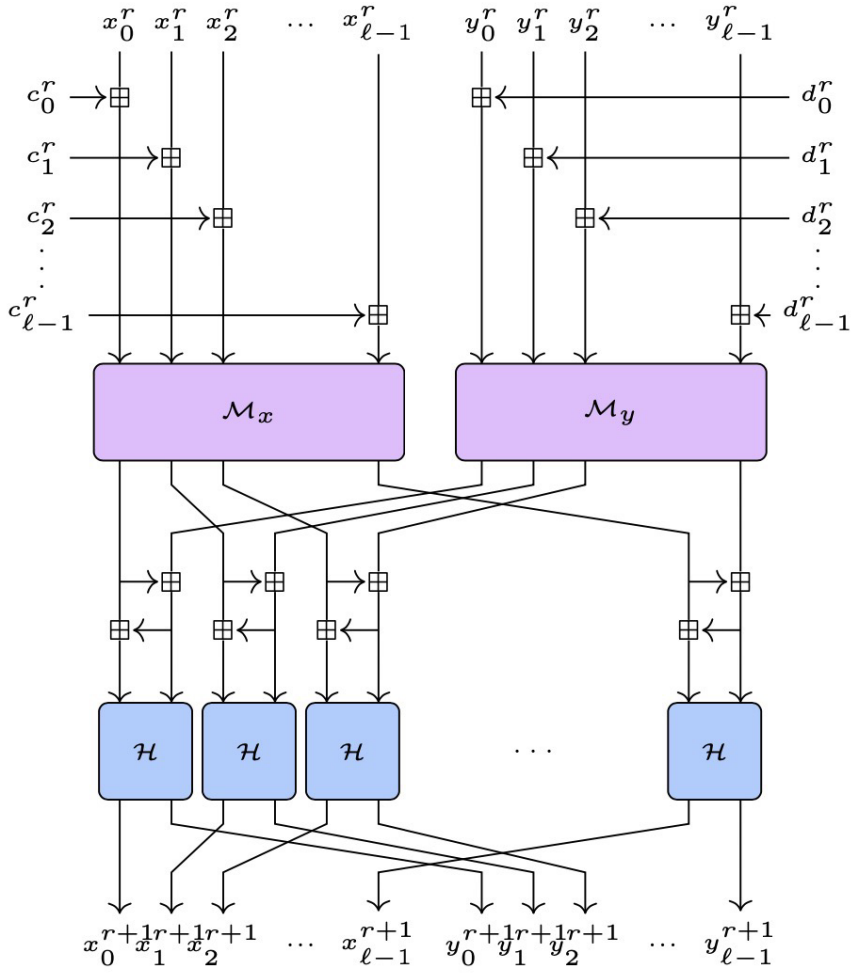

Figure 4.9: Round r of Anemoi [BBC+23]

where R_i is the i round function.
The number of rounds is computed using (q, α, l) with

$$n_r \geq max\{8, \min(5, 1+l) + 2 + \min\{r \in \mathbb{N} | \mathcal{C}_{alg(r)} \geq 2^s\}\}, \tag{4.29}$$

where

$$\begin{cases} \mathcal{C}_{alg(r)} = \left(\binom{4lr+k_\alpha}{2lr}^2\right) & \text{for } q = p, \\ \mathcal{C}_{alg(r)} = lr \cdot 9^{2lr} & \text{for } q = 2^n \end{cases} \tag{4.30}$$

Anemoi Hash

The Anemoi hash function operates over the sponge construction in the field $\mathbb{F}_q^{r+c}$, as all the sponge constructions, r values are used as the rate, and c as the capacity. We must take into consideration that the state has to be separated in X and Y with l elements for each one, so the input must be even.

Sponge construction

There is a modification of the mode of operation of the sponge construction explained in Section 2.2 and the one used in the Anemoi hash.
If we apply the standard sponge construction we may be applying the permutation, Anemoi$^\pi$, one more time when the length of the input is already a multiple of r. To solve this, we don't append more blocks at the end of the message if it is a multiple of r, rather, we add a constants σ before the squeezing phase.
This constant σ is equal to 0 if the message length is not multiple of r, and 1 if it is. Figure 4.10 shows this modification, with P being the Anemoi$^\pi$ permutation.

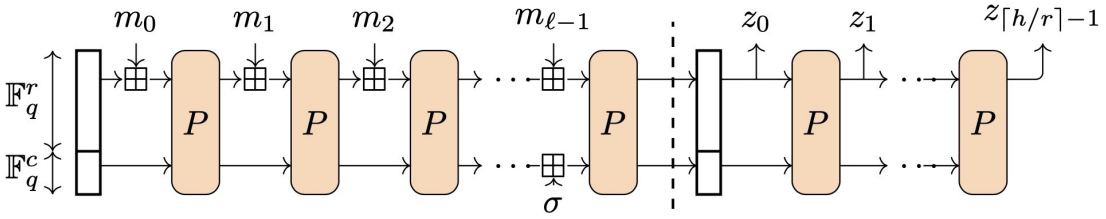

Figure 4.10: Anemoi-sponge [BBC+23]

4.7 Arion

The Arion hash function [RST23] was published by Roy et al. in 2023 and operates over a permutation, and on a lowest level there is the round function.
First we are going to explain the round function, followed by the permutation, and finally the hash function.

Arion round funtion

The Arion round function works on the $\mathbb{F}_p^t$ vector space, and is defined as

$$\mathcal{R}_k^{(i)}(x) = \mathcal{K}_k \circ \mathcal{L}_{c_i} \circ \mathcal{F}_{Arion}^{(i)}(x) \tag{4.31}$$

where $\mathcal{R}_k^{(i)} : \mathbb{F}_p^t \times \mathbb{F}_p^t$.

The structure of the Arion round function differs from the previous hash functions, it is composed of the following components: it starts with a GTDS, *Generalized Triangular Dynamical System* [RS22], followed by the affine layer and a keyed permutation.
Next, a description of each component is provided below.

GTDS Layer

First, we define a field $\mathbb{F}_p$ with p prime elements. Let $n, d_1, d_2, e \in \mathbb{Z}$ that

1. d_1 is the smallest positive integer such that $\gcd(d_1, p-1) = 1$,
2. d_2 is an integer such that $\gcd(d_2, p-1) = 1$,
3. $e \cdot d_2 \equiv 1 \bmod (p-1)$.

The GTDS is defined as $\mathcal{F}_{Arion} = \{f_0, \ldots, f_{t-1}\}$ for $1 \leq i \leq t-1$, and $\alpha_{i,0}, \alpha_{i,1}, \beta_i \in \mathbb{F}_p$ so that $\alpha_{i,0}^2 - 4 \cdot \alpha_{i,1}$ is a quadratic non-residue modulo p.
Thus,

$$\begin{cases} f_i\left(x_0, \ldots, x_{n-1}\right) = x_i^d \cdot g_i\left(\sigma_{i+1,t-1}\right) + h_i\left(\sigma_{i+1,t-1}\right), & 0 \leq i \leq t-2, \\ f_{t-1}\left(x_0, \ldots, x_{t-1}\right) = x_{t-1}^e, & i = t-1, \end{cases} \tag{4.32}$$

where

$$g_i(x) = x^2 + \alpha_{i,0} \cdot x + \alpha_{i,1}, \tag{4.33}$$

$$h_i(x) = x^2 + \beta_i + x, \tag{4.34}$$

and

$$\sigma_{i+1,t-1} = \sum_{j=i}^{t-1} x_j + f_j\left(x_0, \ldots, x_{t-1}\right). \tag{4.35}$$

Affine Layer

The Affine Layer of Arion is defined over the vector space $\mathbb{F}_p^t$ as

$$\mathcal{L}_c(x) = \operatorname{circ}\left(0, \ldots, t-1\right) x + c, \tag{4.36}$$

where $\operatorname{circ}(0, \ldots, t-1)$ is the circulant matrix with the first row being $(1, \ldots, t-1)$ and $c \in \mathbb{F}_p^t$ is the constant vector.

Keyed permutation

The keyed permutation, $\mathcal{K}_k$, works over the field $\mathbb{F}_p^t$. It is defined as

$$\mathcal{K}_k\left(x, k\right) = x + k. \tag{4.37}$$

Arion permutation

The Arion permutation: $\mathbb{F}_p^t \times \mathbb{F}_p^t \times (r+1) \to \mathbb{F}_p^t$ is defined as

$$\text{Arion}^\pi(x, k_0, \ldots, k_{r-1}) = \mathcal{R}_{k_{r-1}}^{(r-1)} \circ \cdots \circ \mathcal{R}_{k_0}^{(0)} \circ \mathcal{L}_0 \circ \mathcal{K}_{k_0}(x) \tag{4.38}$$

for $0 \leq i \leq r-1$, where r is the number of rounds of the permutation.
Furthermore, the Arion$^\pi$ is instantiated with the key $k_0 = \cdots = k_{r-1} = 0$.

Arion hash

The Arion hash function works in the field $\mathbb{F}_p^t$ and over the sponge construction. As a sponge construction, the length of the state is $t = r + c$, where r is the rate and c the capacity.

CHAPTER 5

Implementation

We have developed two libraries[1] to evaluate the performance of the hash functions discussed in Section 4. The first library uses the Plonky2[2] proving system from Polygon Labs, a SNARK proving system based on PLONK and FRI, allowing us to write these functions as arithmetic circuits and generate proofs of knowledge. The second library implements these functions using Dusk Plonk[3] from Dusk Network, a Plonk proving system as descrived in section 3.2.

In addition to zero-knowledge implementations, we also incorporated the plain implementation of such hash functions.

Both libraries are written in the Rust programming language and designed to allow future inclusions of additional hash functions.

Rust was chosen as the programming language due to its compatibility with both Plonky2 and PLONK, given that these frameworks are implemented in Rust.

Both libraries are not stable, meaning that they are under active development, so for using them we can't use the Stable Rust version, we worked on the Nightly version.

These hash functions are constructed based on the concepts provided in section 4. Note that, we used SageMath to compute the constants of the hashes and create binary files to store them. SageMath was selected for his ability in algebra over finit fields and the broader range of mathematical functionalities it provides.

[1] `https://github.com/alexmllo/zk-hashes/tree/main`
[2] `https://github.com/0xPolygonZero/plonky2`
[3] `https://github.com/dusk-network/plonk`

Additionally, we used Criterion, a benchmarking library in Rust, to check the performance of the hashes in both libraries, in the standard and zero-knowledge.

5.1 Implementation using Plonky2

Plonky2 operates on the Goldilocks field (section 2.3.1), $p = 2^{64} - 2^{32} + 1$. We used SageMath for the generation of round constants, MDS matrices, and number of rounds for each hash function.

The implementation is divided into four parts:

1. **Circuit setup**. Firts, we generate the virtual targets[4], such as the private input and the state of the sponge. Subsequently, we check that the input ≥ 0 and is multiple of the rate r. Next, we develop the arithmetic circuit, in our case, it corresponds to the algorithm of each hash function, including the absorbing and the digest part, to generate the output based on the chosen length, and finally assing it as the public input.

2. **Witness generation**. Here, we assing the values of the private inputs (witnesses), such as the inputs and the constants of the hash function.

3. **Proof generation**. Internally, in the Plonky2 framework, the circuit is constructed, all the constraints of the circuit are generated, including copy constraints, forming the constraint system. Afterwards, the witnesses are used to evaluate the constraint system, generating the proof following the procedure explained in section 3.2.

4. **Proof verification**. Finally, the circuit is verified, the verifier using the proof alognside the public inputs decides if he accepts the proof or not.

We only conducted tests for a state length $t = 12$, contrary to the Plonk implementation. Plonky2 is an upgraded version of Plonk, and there are some automatic optimizations to reduce the number of constraints, thus, improving efficiency. Table 5.1 shows the round numbers for the hashes that we implement in the Goldilocks field, where α is the smallest integer value for wich x^α is a permutation, in other words, $gcd(p-1, \alpha) = 1$.

Optimizations
The main goal of this optimization is to improve the efficiency of the circuit by

[4]The term virtual target refers to an intermediate value used during the setup of the circuit but not used in the computation during proof generation.

Rounds					
Hash	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
State size t	$\alpha = 7$				
12	$r_f = 8$, $r_p = 22$	7	8	10	4

Table 5.1: Round numbers for Poseidon [Section 4.2], Rescue-prime [Section 4.4], Griffin [Section 4.5], Anemoi [Section 4.6], Arion [Section 4.7]

reducing the computational complexity and the number of constraints.
We optimized certain computations of the code to minimize the number of constraints. The inverse S-box involves the calculation of $x^{1/d}$. Instead of computing $x^{1/d}$ directly, we first compute y^d within the circuit, where y is a new virtual target representing the inverse operation. This allows us to simplify the exponentiation to a series of multiplications, which are computationally cheaper.
Subsequently, we compute x from y^d during circuit design rather than during proof generation phase. Finally, we connect x and y^d. This is done by establishing a constraint within the circuit that binds the outcome of y^d to the original value x.

In the following code, we implement this optimization.

First of all, we define a trait that implements CircuitBuilder, used to build constraints, add witnesses, generally, to setup the circuit, and we define a function used to compute $x^{1/d}$. Next, we create a virtual target y and with it we compute y^d. Subsequently, we add a generator, it assigns a value to a virtual target during proof generation. In this generator we compute the expensive operation $x^{1/d}$, this operation is computed once during circuit setup and not during proof generation.

Finally we ensure that the result from the operation $x^{1/d}$ is correctly linked to the previously established virtual target y, so $y = x^{1/d}$.

```
1  trait CircuitBuilderExtensions<F: RichField + Extendable<D>, const D:
       usize> {
2      fn exp_inv(&mut self, x: Target) -> Target;
3      fn exp_inv_extension(&mut self, x: ExtensionTarget<D>) ->
       ExtensionTarget<D>;
4  }
5
6  impl<F: RichField + Extendable<D>, const D: usize>
       CircuitBuilderExtensions<F, D>
7    for CircuitBuilder<F, D>
8  {
9    fn exp_inv(&mut self, x: Target) -> Target {
10       let x_ext = self.convert_to_ext(x);
```

```
        Self::exp_inv_extension(self, x_ext).0[0]
    }

    fn exp_inv_extension(&mut self, x: ExtensionTarget<D>) ->
     ExtensionTarget<D> {
        let exp_inv = self.add_virtual_extension_target();
        self.add_simple_generator(ExpGeneratorExtension {
            base: x,
            exp_result: exp_inv,
        });

        // Enforce that y^d = x
        // d = 7 (ALPHA)
        let x2 = self.mul_extension(exp_inv, exp_inv);
        let x4 = self.mul_extension(x2, x2);
        let x6 = self.mul_extension(x4, x2);
        let y_inv = self.mul_extension(x6, exp_inv);
        self.connect_extension(y_inv, x);

        exp_inv
    }
}

#[derive(Debug, Default)]
pub struct ExpGeneratorExtension<const D: usize> {
    base: ExtensionTarget<D>,
    exp_result: ExtensionTarget<D>,
}

impl<F: RichField + Extendable<D>, const D: usize> SimpleGenerator<F,
    D>
    for ExpGeneratorExtension<D>
{
    fn id(&self) -> String {
        "ExpGeneratorExtension".to_string()
    }

    fn dependencies(&self) -> Vec<Target> {
        let deps = self.base.to_target_array().to_vec();
        deps
    }

    fn run_once(
        &self,
        witness: &plonky2::iop::witness::PartitionWitness<F>,
        out_buffer: &mut plonky2::iop::generator::GeneratedValues<F>,
    ) {
        let base = witness.get_extension_target(self.base);
        let mut current_base = base.clone();
```

```
58          let mut exp = <F as Extendable<D>>::Extension::from(F::ONE);
59          let mut power = ALPHA_INV;
60          while power > 0 {
61              if power % 2 == 1 {
62                  exp = exp * current_base;
63              }
64              current_base = current_base * current_base;
65              power /= 2;
66          }
67          out_buffer.set_extension_target(self.exp_result, exp)
68      }
69
70      fn serialize(
71          &self,
72          dst: &mut Vec<u8>,
73          _common_data: &plonky2::plonk::circuit_data::
    CommonCircuitData<F, D>,
74      ) -> plonky2::util::serialization::IoResult<()> {
75          dst.write_target_ext(self.base)?;
76          dst.write_target_ext(self.exp_result)
77      }
78
79      fn deserialize(
80          src: &mut plonky2::util::serialization::Buffer,
81          _common_data: &plonky2::plonk::circuit_data::
    CommonCircuitData<F, D>,
82      ) -> plonky2::util::serialization::IoResult<Self> {
83          let base = src.read_target_ext()?;
84          let exp = src.read_target_ext()?;
85          core::result::Result::Ok(Self {
86              base,
87              exp_result: exp,
88          })
89      }
90  }
```

5.2 Implementation using Dusk Plonk

In the Plonk implementation of the Dusk Network, we operate on the BLS12-381 curve, detailed in section 2.3.2, using r as the modulus of the field.

Plonk modifications. In this Plonk framework, there is a modification in the two input Plonk constraint detailed in Equation 3.1. However, in this framework, the Plonk equation has 3 inputs (3 addition gates)

$$(\mathbf{q_L})_i \cdot \mathbf{x_{a_i}} + (\mathbf{q_R})_i \cdot \mathbf{x_{b_i}} + (\mathbf{q_O})_i \cdot \mathbf{x_{ac_i}} + (\mathbf{q_M})_i \cdot (\mathbf{x_{a_i}} \mathbf{x_{b_i}}) + (\mathbf{q_C})_i + (\mathbf{q_F})_i \cdot \mathbf{x_{d_i}} = 0 \quad (5.1)$$

where d is the "fourth" variable and $\mathbf{q_F}$ the selector coefficient.

We integrated the hash functions directly to the Dusk Network repository[5], with the exception of the Poseidon hash function, which was already available. To acommodate the hash functions from Section 4, we generalized the code.

The implementation is the same as in the Plonky2, due to Plonky2 being based on the PLONK proving system.

For the generation of round constants and MDS matrices, we again used SageMath. However, due to the significant size of the field, reaching up to 256 bits, we can't store the values directly. Therefore, after obtaining the values using SageMath, we converted them into an array of 32 bytes, configuring the byte order to be "little-endian" to ensure the most significant byte is at the end of the byte array. After that, we create a binary file to store the these computed constants. For using them on the code, they were then mapped onto "BlsScalar" in the BLS12-381 scalar field.

In the Plonk implementation we conducted tests for $t \in 4, 5, 6, 8$. Table 5.2 shows the round numbers for each hash function and their state length t.

	Rounds				
	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
t	$\alpha = 5$				
4	$r_f = 8$, $r_p = 56$	11	11	12	5
5	$r_f = 8$, $r_p = 60$	9	-	-	5
6	$r_f = 8$, $r_p = 57$	8	-	10	5
8	$r_f = 8$, $r_p = 57$	8	9	10	4

Table 5.2: Round numbers for Poseidon [Section 4.2], Rescue-prime [Section 4.4], Griffin [Section 4.5], Anemoi [Section 4.6], Arion [Section 4.7]

Note that, the length of the state in Griffin must be 3 or a multiple of 4 as explained in section 4.5 and in Anemoi it must be even, outlined in section 4.6.

To improve efficiency, we perform the same optimization used in Plonky2 to reduce number of constraints in the Plonk framework.

In the following code, the optimization over the Plonk framework is conducted. We can see that the approach to compute $x^{1/d}$ is more simplier than in the Plonky2 framework. However, both of these implementation follow the same idea to optimize this operation.

[5] `https://github.com/dusk-network/Poseidon252`

In Plonk, the Composer is the same as the CircuiBuilder in Plonky2, it handles the arrangement and management of computations and constraints within the circuit.

We start by retrieving the value assigned to a witness variable x which is used to compute $x^{1/d}$. Once the operation on x is computed, this value is allocated within the composer, and its index is stored. This index represents y, and is used to compute $y^{1/d}$. The final step involves ensuring that the original witness x is consistent with the computed index from $y^{1/d}$. This is necesary for ensuring that they conform to the circuit constraints.

```
1   fn inverse_sbox(&mut self, value: &mut Witness) {
2       // Retrieve the value from the composer and compute a power
    operation
3       let tmp = self.composer[*value];
4       let tmp = tmp.pow_vartime(&E_2);
5
6       // Allocate the computed value as a new witness in the composer
7       let wit = self.composer.append_witness(tmp);
8
9       // y^2
10      let constraint = Constraint::new().mult(1).a(wit).b(wit);
11      let tmp_wit = self.composer.gate_mul(constraint);
12      // y^4
13      let constraint = Constraint::new().mult(1).a(tmp_wit).b(tmp_wit
    );
14      let tmp_wit = self.composer.gate_mul(constraint);
15      // Compute y^5 by multiplying y^4 with y, and update the
    original witness value
16      let constraint = Constraint::new().mult(1).a(tmp_wit).b(wit);
17      *value = self.composer.gate_mul(constraint);
18      *value = wit;
19    }
```

CHAPTER 6

Evaluation

Overview

In this section, we provide a comparative analysis of the implemented hash functions from section 4, including Poseidon, Rescue-prime, Griffin, Anemoi and Arion on the Plonky2 and Dusk Plonk proving sytems.

Our objective with these measurement is to provide a realistic perspective and reason about the user decision when using these hash functions.

6.1 Specifications

Measurements were conducted on a system configured with a 1.4 GHz Quad-Core Intel Core i5 processor, 8 GB LPDDR3 RAM with a transfer rate of 2133 MHz, and running macOS 14.5. The system was clocked at 1.4 GHz and utilized Rust's Nightly build dated 2024-02-01. Benchmarking was performed using Criterion 0.5, with Plonky2 version 0.1.4 and dusk-plonk version 0.19.

6.2 Plonky2 Performance

In Table 6.1, the plain performance on the Goldilocks field is compared using the number of rounds specified in Table 5.1.

From this metrics, we can observe that Arion and Griffin shows the best performance with the shortest execution time compared to Rescue and and Anemoi. However, as we show later in Table 6.2, Arion has a worst performance when used in SNARKs

over the Goldilocks field. Rescue and Anemoi have the worst plain performance due to having many $x^{1/\alpha}$ evaluations per round. Griffin and Arion also uses $x^{1/\alpha}$, but only once per round. Poseidon, despite having a high number of rounds, doesn't compute inverse S-box so its performance is as good as Arion and Griffin.

Time (µs)					
	Hash				
	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
State size t	$\alpha = 7$				
12	8.1879	23.293	4.6652	32.266	3.6192

Table 6.1: Plain performance comparasion on the Goldilocks field.

Table 6.2 presents metrics for circuit setup and proof generation of all hash functions in the Plonky2 framework within the Goldilocks field.
Circuit setup and proof generation times reflect the complexity and computational demands of integrating the hash functions into the zero-knowledge framework.

In this table we can see that, Anemoi, Griffin and Arion show the most efficient circuit setup times around 33 ms, meaning that their circuit design are less complex than in the case of Poseidon and Rescue, for the same reasons we explained before. Anemoi presents better metrics than in plain performance, due to being able to apply the optimization to compute $x^{1/\alpha}$ as it is explained in section 5.1.
Poseidon has the longest proof generation time (114,32 ms), which indicate a higher number of constraint in the circuit design, due to the high round number it has. Arion, we can see that it has a moderately high proof generation time, indicating that it is not optimized to work on the Goldilocks field, in contrast, in Table 6.5 it presentes good results when working on the BLS12-381 curve.

Time (ms)										
	Circuit Setup					Proof Generation				
Hash	Poseidon	Rescue-prime	Griffin	Anemoi	Arion	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
State size t	$\alpha = 7$									
12	79.575	42.081	33.891	33.778	33.806	114.32	74.845	75.450	74.802	89.75

Table 6.2: Plonky2 performance comparasion.

6.3 Plonk Performance

In Table 6.3, we evaluate the plain implementation of the hash functions within the BLS12-381 curve using the round numbers from Table 5.2.

As the table shows, Poseidon and Arion are the fastest permutations, reinforcing the results obtained in Table 6.1. Similar to previous observations, Rescue and Anemoi have the lowest performance, fundamentally due to the number of $x^{1/\alpha}$ operations required to compute in one round. In contrast, Griffin maintains comparable results for different state size t.

As we can see, for smaller state size t the performance is better than for a larger t, due to the cost of high computational operations needed to be evaluated per round. If we take a look at Table 5.2, we can observe that we have more rounds for smallest t and less for larger t, with this, we may believe that we will have better performance for bigger t, but looking at the results we can conclude that the state size is more influential than the number of rounds.

Time (µs)					
	Hash				
	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
State size t	$\alpha = 5$				
4	47.89	776.98	206.96	421.56	94.76
5	73.308	795.47	-	-	96.84
6	99.2	856.76	-	525.08	100.25
8	166.06	1146	221.51	704.67	84.649

Table 6.3: Plain performance comparison on the BLS12-381 scalar field.

Next, we will provide metrics for the hash functions implemented in the Dusk Plonk proving system within the BLS12-381 curve.

Table 6.4 compare the efficiency of the different hash functions in Plonk by counting the number of constraints required to compute the proof.
The table indicates that as the state size t increases, the number of constraints typically increases as well for all hash functions. This trend suggests that larger state sizes require more computational resources, which align with the expected increase in the complexity of the computation.

Arion is the most efficient in terms of constraints requirements across all tested state sizes. Making it suitable for systems where performance is important.
Poseidon starts at a resonable constraint count for smaller states, esclates faster

than the other hashes due to his high number of rounds.
Anemoi and Griffin demonstrate a consistent increase of constraints when increasing the state size.

Plonk Constraints					
	Hash				
	Poseidon	Rescue-prime	Griffin	Anemoi	Arion
State size t	$\alpha = 5$				
4	788	536	356	328	241
5	995	550	-	-	305
6	1501	682	-	412	367
8	2461	1034	866	634	406

Table 6.4: Dusk Plonk constraint comparasion. Round numbers are the same as in Table 5.2

The Table 6.5 shows performance metrics for the circuit setup and proof generation times of the hash functions implemented using Dusk Plonk in the BLS12-381 curve. Figure 6.1 represents a graphic for the proof generation times using the values from Table 6.5.

In the circuit setup, the times generally increase with the state size for most hash functions.

Poseidon and Rescue-prime show very similar setup times for smaller state sizes but when $t \geq 6$ Poseidon shows an increase and Anemoi also for $t = 8$.

Arion stands out as the most efficient and scalable in both circuit setup and proof generation. In contrast to the implementation in Plonky2 over the Goldilocks field, in the Plonk framework over the BLS12-381, it presents the best performance over all the hashes making it more suitable when working over the BLS12-381 curve than in the Goldilocks field.
Poseidon, similar to the previous tables, shows increases in both circuit setup and proof generation times as state size increase.
Griffin, Anemoi and Arion, the three of them perform the inverse S-box only one time per round, as previously explained, but Arion having less number of rounds than the previous hashes makes it more efficient in circuit setup and proof generations.

Time (ms)								
	Circuit setup				Proof generation			
	State size t							
	4	5	6	8	4	5	6	8
Hash	$\alpha = 5$							
Poseidon	814.58	811.16	1514.4	2489.1	766.25	761.49	1465.1	2337.9
Rescue-prime	812.88	812.62	813.4	1507.6	770.11	766.81	773.95	1421.8
Griffin	434.83	-	-	809.59	437.83	-	-	835.42
Anemoi	435.18	-	437.03	812.83	422.63	-	414.32	784.38
Arion	303.07	436.53	436.76	437.69	293.58	412.53	418.73	417.21

Table 6.5: Dusk Plonk performance comparasion with Round numbers from Table 5.2

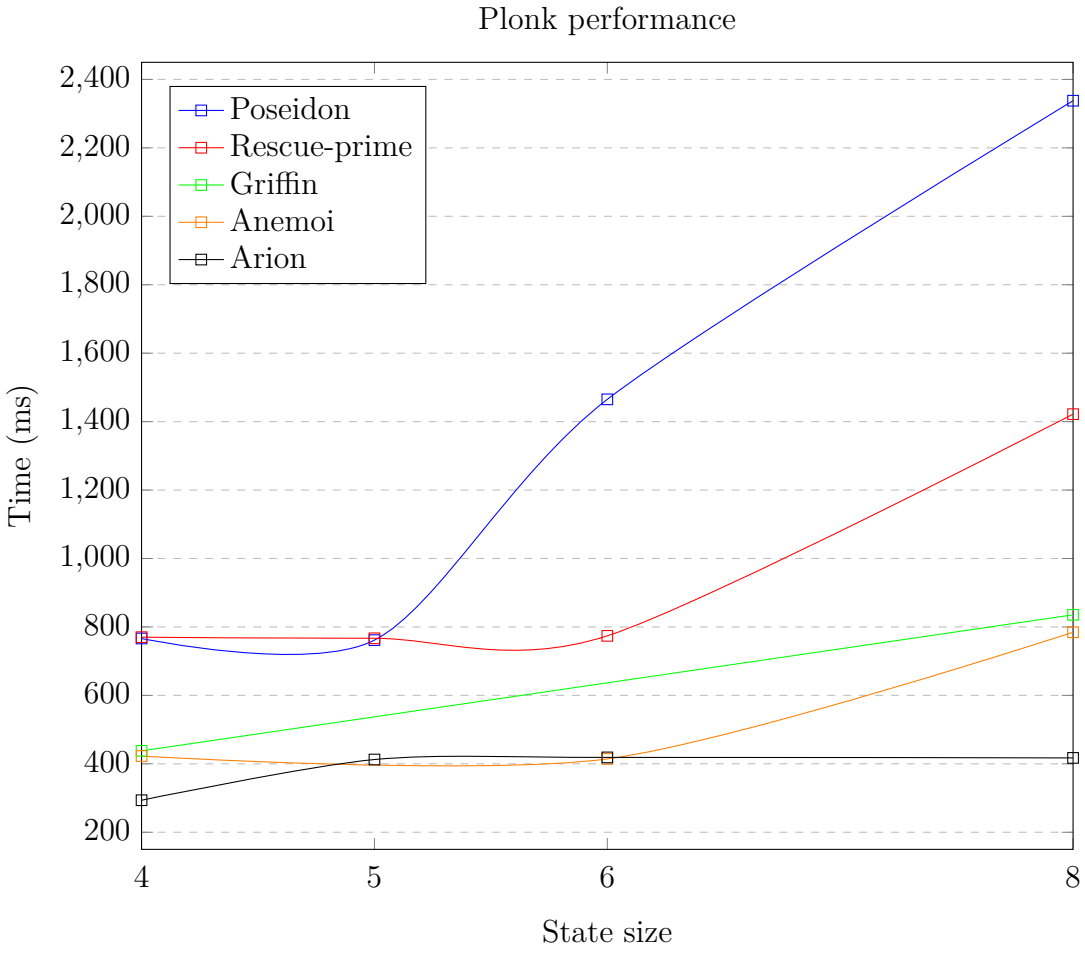

Figure 6.1: Dusk Plonk proof generation performance using values from Table 6.5

CHAPTER 7

Conclusions

7.1 Discussion

This thesis succesfully developed and evaluated a series of hash functions within two zero-knowledge frameworks, namely Plonky2 and Plonk, focusing on the generation of proofs of knowledge for these computed hashes.

In chapter 2, we provide the theoretical knowledge needed to understand the development of the project. This include some background knowledge on hash functions, sponge structure and finit fields, forming a solid foundation for understanding zero-knowledge proofs and the specific proving system we are using in this thesis, Plonk.

Chapter 5 discusses the implementation of these hash functions, highlighting significant optimizations that improve performance, particulary within the zero-knowledge framework. Benchmarking results indicate that the implemented hash functions meet the expected performance regarding their characteristics, as explained in section 4.

So, which is the best hash function? As is often the case, the answer is not that simple and depends of various factors, such as the chosen zero-knowledge proof system and the primary cost metric for the use case. Let's first investigate the best hash function considering prover performance, in other words, minimizing the number of constraints inside the cicuit. Since Poseidon, the objective has been to minimize the number of multiplications within the circuit. This led to using both $y = x^d$ and $y = x^{1/d}$, which results in a secure hash function with a small number of rounds (and thus also a small number of constraints). The first hash functions to implement this is Rescue, followed by new optimizations that let to the development

of Griffin, Anemoi and Arion. We compared these hash functions for performance when used in various configurarions and proof systems. From the results obtained in Chapter 6, Arion shows better performance when used in Plonk proof system compared to all other hash functions. In contrast, in the Plonky2 proving system, Rescue, Griffin and Anemoi, each with similar performance, perform better than Arion.

What makes these hash functions, Griffin, Rescue, Anemoi and Arion so fast inside the zero-knowledge circuit makes them slow for plain hashing. Consequently, for cases where plain performance is equally important as the prover performance, the hash function with best performance differs from the previous results. Here, we need to consider the prime field we are working on. In this thesis, we implemented them in two field, the Goldilocks and the BLS12-381 scalar field. In the case of the BLS12-381, a large field, Poseidon results in the best performance for small state sizes, and Arion for large state sizes. If using the Goldilocks small field, Arion, Griffin and Poseidon perform similary when the state size is large. However, although is has not been explained in this thesis, if our proving system support lookup arguments [BCG+18, EFG22, PK22, ZBK+22, ZGK+22], we can use hash functions designed for native performance using lookups, such as Reinforced Concrete [GKL+21] or Monolith [GKL+23].

In conclusion, choosing the best hash function depends on various factors, which we explored for various needs. If prover performance is needed, use Arion. If plain performance is important, Poseidon is the best option for large fields, while small fields, Arion, Griffin and Poseidon present similar performance.

7.2 Future work

The work completed in this thesis set a baseline for further research and development.

It would be interesting to explore the development of recursive proofs, which is a property of SNARKs allowing a SNARK to verify other SNARKs, this could simplify processes where multiple computations need verification.

Future research could include the addition of new hash functions to our library, expanding the scope for benchmark comparasions.

Another direction could involve testing the implemented hash functions within another zero-knowledge framework. This metrics would allow us to evaluate how the performance of the hash functions varies with different frameworks.

We can remark that the objectives of this thesis have been succesfully archieved, preparing the stage for further work and innovations.

CHAPTER 8

Sustainability Analysis and Ethical Implications

Sustainability

Environmental impact

During development

The main environmental consideration for this project is the computer, GitLab, emails and a printer.

Macbook Pro (2020): Operating at 0.0582 kWh[1] and with an electricity impact of 0.051 $kgCO_2eq/kWh$[2], we can get his carbon footprint doing:

$$0.0582 \text{ kWh} \times 0.051 \text{ kgCO}_2\text{eq/kWh} = 2.968 \text{ gCO}_2\text{eq/hour}.$$

Embodied Energy of the Computer: The MacBook has an embodied energy of 212 kg CO_2e[3] distributed over a 5 year lifespan. Thus, the embodied energy per year is:

$$\frac{212 \text{ kg CO}_2\text{e}}{5 \text{ years}} = 42.4\text{kg CO}_2\text{e/year}$$

[1] https://support.apple.com/en-us/111981

[2] https://app.electricitymaps.com/zone/AT?solar=false&remote=true&wind=false

[3] https://www.apple.com/environment/pdf/products/notebooks/13-inch_MacBookPro_PER_Nov2020.pdf

and per hour:

$$\frac{42.4 \text{ kg } CO_2e/\text{year}}{87600 \text{ hours/year}} = 4.84 \text{ g}CO_2e/\text{hour}.$$

GitLab usage: With corporate emissions at 16.654 tCO_2e/year[4] distributed among 30 millions users. The emissions per user are:

$$\frac{16.654 \text{ t}CO_2e}{30000000 \text{ users}} \approx 0.5551 \text{ kg}CO_2e/\text{year}.$$

Per hour is 0.00633 gCO_2e/hour.

Printer: Assuming an emission of 5 gCO_2e per printed page.

Emails: Based on [BL20], an email without attachments generates about 0.3 gCO2e, and an email with an attachment generates about 50 gCO2e.

This thesis has got a duration of 21 weeks, and 40 hours per week, so, we have spend 840 hours in total. During the project, 18 email were sent without attachments, 21 with attachments, and 13 papers of aproximately 30 pages each one. With this, we can calculate the total footprint of this thesis:

$$\begin{aligned}
\text{Computer} &= 2.968 \text{ g}CO_2\text{eq/hour} \times 840 \text{ hours} = 2493.12 \text{ g}CO_2e \\
\text{Emails} &= 18 \times 0.3 \text{ g}CO_2e + 21 \times 50 \text{ g}CO_2e = 1055.4 \text{ g}CO_2e \\
\text{GitLab} &= 0.00633 \text{ g}CO_2e/\text{hours} \times 840 \text{ hours} = 5.322\text{g}CO_2e \\
\text{Printer} &= 13 \times 30 \times 0.3 \text{ g}CO_2e = 117 \text{ g}CO_2e \\
\text{Total} &= 2493.12 + 1055.4 + 5.322 + 117 = 3670.84\text{g}CO_2e
\end{aligned}$$

Project execution

The primary resources used in this thesis include computational resources (compouters, servers for simulations and software tools), electricity for running the computers and printer.

This thesis focuses on providing metrics of existing cryptographic primitives (hash functions) eficient in zero-knowledge frameworks. Thus, providing help for which one of these efficient primitives works better for the frameworks it is intended to use. And if these results are used in cryptographic processes, it could lead to lower energy consumption.

We could mitigate the environmental impact, we could use cloud computing, in this way we won't have for an individual computer to run all the software.

[4] https://about.gitlab.com/files/esg/GitLab_ESG_FY23_Highlights.pdf

Risks and Limitations

There are some escenarios that could arise the the project's footprint. For example, if the project needs more extensive test it could increase the electricity consumption. Furthermore, if the duration is extended regarding the initial planned, the associated emissions would increase.

This project could have been conducted in a more efficient way by using cloud resources and not rely entirely on a physical computer.

Economic impact

During development

Now, we are going to analize and quantify the human cost and material costs during the development of this thesis (21 weeks).

We used a MacBook Pro (2020), regarding the development of the project, we used Rust and Python, both of them are a free open-source programming language, Criterion, also a free open-source benchmarking library in Rust, SageMath, a free open-source mathematics software system, and LaTeX, for writting the thesis, also a free software. The printer is owned by the research group, so it is also free.

For the student, we will assume the salary of an internship, 10€/hour, and 20€/hour salary for each supervisors.

	Amount	Salary	Dedication	Total
Student	1	10€/hour	40 hours/week	8400€
Supervisors	2	20€/hour	1 hour/week	420€
Computer	1			1000€
			Total:	8820€

Table 8.1: Economic impact

Project execution

This project could benefit other projects by providing knowledge gained from optimizing cryptographic primitives within zero-knowledge frameworks, these results could be applied to other areas of cryptography. Furthermore, the developed software could be reused or adapted for other projects. Thus, reducing their initial development costs and speeding up the development.

Risks and Limitations

Several escenarios could put at risk the viability of the project:

For example, if the developed cryptographic algorithms are compromised or found to have vulnerabilities, the viavility of the project could be reduced. Furthermore, fast advances in cryptography could make them not usefull compared to these new ones.

Social impact

During development

This project provides metrics of the behavior of some cryptographic primitive within two zero-knowledge frameworks. Besides it also provides theoretical knowledge to fully understant all the concepts required for using these metrics. Thus, it presents advances in the field of applied cryptography.

Project execution

The primary beneficiaries of this project are researchers and developers in the field of cryptography, they can use these metrics to develop new cryptographic tools or improve the ones implemented in this thesis.

The field of zero-knowledge is very new, and this project helps to solve the lack of information in this section by providing metrics of cryptographic primitives within zero-knowledge and help researchers with these results.

Risks and Limitations

We could not find a risk that could negatively effect the population.

Ethical implications

This thesis addresses the need for more information regarding these new cryptographic primitives efficient within zero-knowledge frameworks. We provide metrics that can be used for organitzations when deciding which primitive is more efficient in their project or for researchers to continue to develop and improve in this area.

These needs are defined by the advances in this field, it is a very promising field that has been started to develop recently, the implemented primitives are 4 years old maximum, so advances in this area are required.

This project complies with the General Data Protection Regulation (GDPR) to ensure that any data used or generated respects privacy and data protection laws.

Relationship with the Sustainable Development Goals

From the 17 SDGs, we can say that our thesis contributes to three of them.

SDG 9: Industry, Innovation, and Infrastructure
We contribute to this goal, which focus on building resilient infrastructure, promoting sustainable industrialization, and promoting innovation.

By providing metrics of efficient cryptographic primitives in zero-knowledge frameworks, we enchange the security of industries that use this technology, including finance, healtcare, etc. These cryptographic primitives are fundamental to protect data and ensuring integrity of them.

SDG 11: Sustainable Cities and Communities
We don't contribute directly to this goal but, smart cities contribute to the sustainability of the cities, so IoT devices are used for this goal. Cryptographic primitives are esencial to protect exchanged data between devices, and mantain privacy.

SDG 13: Climate Action
We are presenting metrics that could be used for companies to decide which cryptographic primitive is the more efficient for their needs. Besides, the use of this optimizated cryptographic primitives reduce the computational costs, thus, reducing the carbon footprint associated to it.

List of Figures

List of Tables

Bibliography

[AABS+20] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, pages 1–45, 2020.

[AGP+19] Martin R. Albrecht, Lorenzo Grassi, Leo Perrin, Sebastian Ramacher, Christian Rechberger, Dragos Rotaru, Arnab Roy, and Markus Schofnegger. Feistel structures for mpc, and more. Cryptology ePrint Archive, Paper 2019/397, 2019. `https://eprint.iacr.org/2019/397`.

[AGR+16] 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.

[AKM+22] Tomer Ashur, Al Kindi, Willi Meier, Alan Szepieniec, and Bobbin Threadbare. Rescue-prime optimized. *Cryptology ePrint Archive*, 2022.

[BBC+23] Clémence Bouvier, Pierre Briaud, Pyrros Chaidos, Léo Perrin, Robin Salen, Vesselin Velichkov, and Danny Willems. New design techniques for efficient arithmetization-oriented hash functions: anemoi permutations and jive compression mode. In *Annual International Cryptology Conference*, pages 507–539. Springer, 2023.

[BCG+18] Jonathan Bootle, Andrea Cerulli, Jens Groth, Sune Jakobsen, and Mary Maller. Arya: Nearly linear-time zero-knowledge proofs for correct program execution. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018*, pages 595–626, Cham, 2018. Springer International Publishing.

[BFM19] Manuel Blum, Paul Feldman, and Silvio Micali. *Non-interactive zero-knowledge and its applications*, page 329–349. Association for Computing Machinery, New York, NY, USA, 2019.

[BGK+21] Mario Barbara, Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, Roman Walch, and T Graz. Reinforced concrete: Fast hash function for zero knowledge proofs and verifiable computation. *IACR Cryptol. ePrint Arch.*, 2021:1038, 2021.

[BGM17] Sean Bowe, Ariel Gabizon, and Ian Miers. Scalable multi-party computation for zk-snark parameters in the random beacon model. Cryptology ePrint Archive, Paper 2017/1050, 2017. `https://eprint.iacr.org/2017/1050`.

[BL20] Mike Berners-Lee. *How bad are bananas?: the carbon footprint of everything*. Profile Books, 2020.

[CD98] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In *Advances in Cryptology—CRYPTO'98: 18th Annual International Cryptology Conference Santa Barbara, California, USA August 23–27, 1998 Proceedings 18*, pages 424–441. Springer, 1998.

[DGDG11] Ulrich Daepp, Pamela Gorkin, Ulrich Daepp, and Pamela Gorkin. Fermat's little theorem. *Reading, Writing, and Proving: A Closer Look at Mathematics*, pages 315–323, 2011.

[DL18] Sébastien Duval and Gaëtan Leurent. Mds matrices with lightweight circuits. *IACR Transactions on Symmetric Cryptology*, 2018(2):48–78, 2018.

[EFG22] Liam Eagen, Dario Fiore, and Ariel Gabizon. cq: Cached quotients for fast lookups. Cryptology ePrint Archive, Paper 2022/1763, 2022. `https://eprint.iacr.org/2022/1763`.

[Fei73] Horst Feistel. Cryptography and computer privacy. *Scientific american*, 228(5):15–23, 1973.

[For89] Lance Jeremy Fortnow. *Complexity-theoretic aspects of interactive proof systems*. PhD thesis, Citeseer, 1989.

[GHR+22] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. Horst meets fluid-spn: griffin for zero-knowledge applications. *Cryptology ePrint Archive*, 2022.

[GJMG11] Bertoni Guido, Daemen Joan, P Michaël, and VA Gilles. Cryptographic sponge functions, 2011.

[GKL+21] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Reinforced concrete: A fast hash function for verifiable computation. Cryptology ePrint Archive, Paper 2021/1038, 2021. https://eprint.iacr.org/2021/1038.

[GKL+23] Lorenzo Grassi, Dmitry Khovratovich, Reinhard Lüftenegger, Christian Rechberger, Markus Schofnegger, and Roman Walch. Monolith: Circuit-friendly hash functions with new nonlinear layers for fast and constant-time implementations. Cryptology ePrint Archive, Paper 2023/1025, 2023. https://eprint.iacr.org/2023/1025.

[GKR+21] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for {Zero-Knowledge} proof systems. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 519–535, 2021.

[GKS23] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In *International Conference on Cryptology in Africa*, pages 177–203. Springer, 2023.

[GLR+20] Lorenzo Grassi, Reinhard Lüftenegger, Christian Rechberger, Dragos Rotaru, and Markus Schofnegger. On a generalization of substitution-permutation networks: The hades design strategy. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 674–704, Cham, 2020. Springer International Publishing.

[GMR85] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, page 291–304, New York, NY, USA, 1985. Association for Computing Machinery.

[Gol98] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 78(110):1–108, 1998.

[GWC19] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[Knu99] Lars R Knudsen. Block ciphers—a survey. In *State of the Art in Applied Cryptography: Course on Computer Security and Industrial*

Cryptography Leuven, Belgium, June 3–6, 1997 Revised Lectures, pages 18–48. Springer, 1999.

[KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In Masayuki Abe, editor, *Advances in Cryptology - ASIACRYPT 2010*, pages 177–194, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Nit20] Anca Nitulescu. zk-snarks: a gentle introduction. *Ecole Normale Superieure*, 2020.

[Nyb96] Kaisa Nyberg. Generalized feistel networks. In Kwangjo Kim and Tsutomu Matsumoto, editors, *Advances in Cryptology — ASIACRYPT '96*, pages 91–104, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[PK22] Jim Posen and Assimakis A. Kattis. Caulk+: Table-independent lookup arguments. Cryptology ePrint Archive, Paper 2022/957, 2022. `https://eprint.iacr.org/2022/957`.

[Pre93] Bart Preneel. *Analysis and design of cryptographic hash functions*. PhD thesis, Citeseer, 1993.

[Pre94] Bart Preneel. Cryptographic hash functions. *European Transactions on Telecommunications*, 5(4):431–448, 1994.

[RS22] Arnab Roy and Matthias Johann Steiner. Generalized triangular dynamical system: An algebraic system for constructing cryptographic permutations over finite fields. *arXiv preprint arXiv:2204.01802*, 2022.

[RST23] Arnab Roy, Matthias Johann Steiner, and Stefano Trevisani. Arion: Arithmetization-oriented permutation and hashing from generalized triangular dynamical systems. *arXiv preprint arXiv:2303.04639*, 2023.

[SAD20] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: a standard specification (sok). *Cryptology ePrint Archive*, 2020.

[Sha49] Claude E Shannon. Communication theory of secrecy systems. *The Bell system technical journal*, 28(4):656–715, 1949.

[ZBK+22] Arantxa Zapico, Vitalik Buterin, Dmitry Khovratovich, Mary Maller, Anca Nitulescu, and Mark Simkin. Caulk: Lookup arguments in sublinear time. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 3121–3134, New York, NY, USA, 2022. Association for Computing Machinery.

[ZGK+22] Arantxa Zapico, Ariel Gabizon, Dmitry Khovratovich, Mary Maller, and Carla Ràfols. Baloo: Nearly optimal lookup arguments. Cryptology ePrint Archive, Paper 2022/1565, 2022. https://eprint.iacr.org/2022/1565.

[Zip79] Richard Zippel. Probabilistic algorithms for sparse polynomials. In *International symposium on symbolic and algebraic manipulation*, pages 216–226. Springer, 1979.