# AOHashes

*AO privacy-friendly primitives testing over Dusk-Plonk ZK library*

Filippo Merlo

*March 2025*

# Contents

**Abstract**

This paper presents a comparative analysis of several Arithmetization-Oriented (AO) cryptographic primitives, designed to enhance the efficiency of Zero-Knowledge (ZK) proofs. AO primitives are optimized to work with constraint-based ZK proof systems like Plonk, which are widely used for privacy-focused applications such as blockchain and secure authentication. By implementing these primitives in the Rust programming language and testing them with the Dusk Network's Plonk library, we aim to assess their performance in terms of computational speed, constraint complexity and proof generation efficiency. The primitives evaluated include `GMiMC`, Poseidon, `Rescue`, `Rescue-Prime`, Griffin, `Anemoi` and `Arion`, each with unique design strategies that balance the trade-offs between security, efficiency and polynomial degree.

Our analysis provides a clear comparison of how these primitives perform under various conditions, highlighting their strengths and limitations. We also discuss which primitives may be best suited for specific applications based on their stability, computational overhead or adaptability to different field sizes and security levels. By shedding light on these performance characteristics, we hope to guide future optimizations and inspire new cryptographic designs that can further enhance the efficiency ZK proof systems.

# Chapter 1

# Introduction

In recent years, the field of cryptography has seen a surge of interest in Zero-Knowledge (ZK) proofs, which allows one party (the prover) to prove the validity of a computation to another party (the verifier) without revealing any underlying data. This unique property makes ZK proofs particularly valuable in applications where privacy and data security are crucial, such as blockchain technology, authentication systems and secure financial transactions.
Among the various types of ZK proofs, a special category known as non-interactive proofs has gained popularity. Unlike interactive ZK proofs, non-interactive proofs do not require communication between the prover and the verifier, making them ideal for decentralized environments like blockchains.
This paper focuses on a specific type of non-interactive ZK proof system called Plonk [11], which belongs to the SNARK[1] family. Plonk proofs are based on arithmetic circuits, which are translated into constraints and finally represented as polynomials that will be evaluated by the verifier. This structure, even if it at first glance seems complex and computationally heavy, in reality is very efficient and powerful for solving ZK proof problems.
However, one challenge with ZK proof systems including Plonnk is that traditional cryptographic primitives are not well-suited to work within these terms and structures, losing all their efficiency from a performance point of view. To address this, researchers have developed a new class of cryptographic primitives designed specifically for this purpose, known as Arithmetization-Oriented (AO) primitives. These AO primitives aim to maximize the performance of ZK proofs by minimizing the computational overhead associated with constraints, particularly when performing basic operations like addition and multiplication.
In this paper, we explore and evaluate several of the latest AO primitives that have been proposed and published in these last years through a Rust programming language implementation and the testing with the Dusk Network's Plonk ZK library [10]. Our goal is to compare these primitives based on their performance, efficiency and suitability for different use cases. By highlighting their strengths and weaknesses, we aim to provide valuable insights for developers and researchers interested in optimizing ZK-proof systems and designing new cryptographic solutions.

[1] Succinct Non-interactive ARguments of Knowledge

# Chapter 2

# Overview

As mentioned in the introductory chapter 1, the primitives that have been tested in this project are Arithmetization-Oriented and this is because for Zero-Knowledge proofs plain evaluation is important, but proofs generation and verification is even more critical, and this design provides an efficient solution for performing both operations. Another important aspect of AO primitives is that they are suitable for a vast number of field sizes and security levels, making them very elastic and adaptable to different scenarios and applications. However, as we will evidence from this paper's results, the design of these primitives can severely affect the computational time of generating and verifying proofs even though the same environment is provided and the same instances are used.
Because the Dusk Network team provides a library that implements the Plonk system [10], plus they also implement the primitive Poseidon [9], we have decided to implement all the remaining primitives using the same programming language used for these two implementations (*i.e.* Rust) in a single project, in order to have the same baseline for all the primitives and being able to conduct a fair comparison between them, minimizing potential sources of errors in our final results.
For this reason we have chosen to work on only one the BLS12-381 elliptic curve, with $p = 52435875175126190479447740508185965837690552500527637822603658699938581184513$.

## 2.1 Primitives

The primitives that have been implemented and tested in this project are:

- `GMiMC` [1]
- Poseidon [14]
- `Rescue` [3]
- `Rescue-Prime` [19]
- Griffin [12]
- `Anemoi` [6]
- `Arion` [17]

We can split them into two categories: in the *first category* we found those that have been designed for maintaining the **degree of polynomials** as **low** as possible, while having a high number of rounds to achieve a minimum level of bits security, which are `GMiMC`, Poseidon, `Rescue` and `Rescue-Prime`, while in the *second* one has been used the opposite strategy, *i.e.* achieving efficiency maintaining a **low number of rounds**, but increasing exponentially the polynomial degree with the introduction of inverse power permutations, and these primitives are Griffin, `Anemoi` and `Arion`.

### 2.1.1 GMiMC

The first function implemented in this project it has been GMiMC[1]: a hash function based on unbalanced Feistel networks with a low multiplicative complexity that suits well ZK-SNARK applications. The mapping used in the Feistel networks is $x \to x^d$, which is a design idea by Nyberg and Knudsen, that has been shown it leads to efficient instatiations for SNARKs [2]. Among the available Feistel network modes, for this project we implemented the ERF[2] variant, which schema is depicted in Figure 2.1.

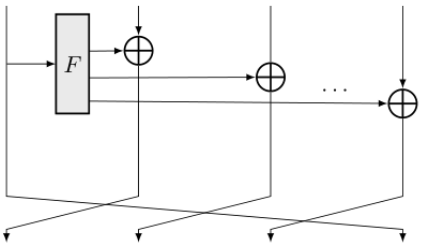


Figure 2.1: Single round of $\texttt{GMiMC}_{\texttt{ERF}}$ [1, Fig. 2].

As it is possible to deduct from the above schema, a single $\texttt{GMiMC}_{\texttt{ERF}}$ round is composed of the following operations:

- Computation of $y = F(x_0) : x_0 \to x_0^d$, where $x_0$ is the first element of the state;
- XOR addition between each element of the state except the first one, and the output $y$ of $F$;
- Left-rotation of the state by one position.

Some common instatiations of GMiMC to achieve the minimum security of 128 bits over the curve BLS12-381 are shown in Table 2.1, while the formula used to compute the minimum number of rounds $n$ [1, Tab. 2, Tab. 3] against some common cryptographic attacks is:

$$n = \max\left\{n_{\text{Int}}, n_{\text{HighOrd}}, n_{\text{RTDif}}\right\}, \tag{2.1}$$

where

$$n_{\text{Int}} = \lceil 2 \cdot \log_d(2) \cdot \log_2(p) \rceil + 2t, \tag{2.2}$$

$$n_{\text{HighOrd}} = 2 + 2t + \lceil 2 \cdot \log_d(t) \rceil, \tag{2.3}$$

$$n_{\text{RTDif}} = 2 + \left\lceil (t^2 + t) \cdot \frac{\log_2(p)}{2 \cdot (\log_2(p) - 1)} \right\rceil, \tag{2.4}$$

where $t$ is the state width, $d$ is the exponent of the mapping function $F$ (in our chase 5) and $p$ is the prime used for our curve.

[1] Generalized MiMC
[2] Expanding Round Function

Table 2.1: $\texttt{GMiMC}_{\texttt{ERF}}$ instances for $d = 5$.

| $t$ state width | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|
| $n$ rounds | 328 | 330 | 332 | 334 | 338 |

### 2.1.2 Poseidon

Poseidon is a **sponge function** [4] whose inner permutation function named Poseidon$^{\pi}$ is an unkeyed version of the Hades strategy [13]. The Poseidon$^{\pi}$ permutation exploits two round functions, respectively denoted as *full* and *partial*, and their difference resides in how the S-Box[3] is applied. Precisely, in the full round version the S-Box is applied to each element of the state, while in the partial one only to the last element of the state. The decision to opt for this design strategy in the permutation function is attributable to the goal of improving Poseidon's performance even with a high number of rounds, by lightening the final computational cost with the removal of some S-Boxes.

**NOTE:**
It is important to notice that Poseidon$^{\pi}$ can be adjusted to use only full rounds and achieving in this way a greater level of security. However, the opposite, *i.e.* using only partial rounds, is strongly discouraged because to achieve a minimum level of security a minimal amount of full rounds are still required, where the exact value depends on the size $t$ of the state.

A single round is composed of the following concatenation of functions:

- *AddRoundConstant* (ARC): a constant $c_r$, where $r$ is the value of the round, is added to each value of state;
- *S-Box* (S): the input is exponentiated to an exponent $d$ where $d \geq 3$ and also $d$ is co-prime with $p - 1$. This function is applied singularly to each/last element of the state, depending on the type of round;
- *MixLayer* (M): matrix-vector multiplication between a fixed matrix and the state, where the matrix is a square MDS[4] matrix chosen such that no subspace trail with inactive/active S-Boxes can be set up for more than $t - 1$ rounds, with $t$ the size of the state.

In Figure 2.2 is shown the complete structure of the Poseidon$^{\pi}$ function, while in Table 2.2 are collected the most commonly used instantiations for Poseidon [14, Tab.2].

[3] Substitution Box
[4] Maximum Distance Separable

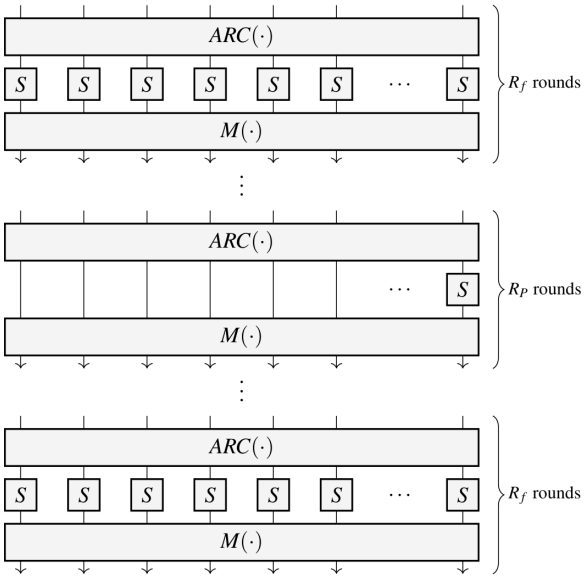


Figure 2.2: POSEIDON$^{\pi}$ permutation function [14, Fig. 2].

Table 2.2: POSEIDON instances for $d = 5$ [14, Tab. 2].

| $t$ state width | 2/3 | 4/5/6/8 |
|---|---|---|
| $n_f$ full rounds | 8 | 8 |
| $n_p$ partial rounds | 57 | 60 |

### 2.1.3 `Rescue`

The `Rescue` primitive is a sponge function based on the `Marvellous` design strategy and the focus of the authors of this primitive was realizing a secure and robust function, rather than an efficient one, while keeping a simple structure. For this purpose, they proposed the `Marvellous` strategy which is a SPN[5] round function, where each round is split into two phases and each phase is the composition of different operations.
A single round is organized as follows:

- **First phase**
  - *Inverse S-Box* ($S^{-1}$): application of the inverse power map $x_i \rightarrow x_i^{\frac{1}{d}}$ to each element of the state;
  - *MixLayer* (M): matrix-vector multiplication between an MDS matrix and the state;
  - *AddRoundConstant* (ARC): addition of a constant $c_i^r$ to each element of the state $x_i$ where $0 \leq i < t$ is the index of the elements of the states and $r$ is the round value and $t$ the state width;

[5] Substitution-Permutation Network

- **Second phase**
  - *S-Box* (S): application of the power map $x_i \to x_i^d$ to each element of the state;
  - *MixLayer* (M): matrix-vector multiplication with the same MDS matrix of the first phase;
  - *AddRoundConstant* (ARC): addition of another set of constants $c_{t+i}^r$ to each element of the state (it has been used the same notation of the first phase).

The round function of `Rescue` is summarized in Figure 2.3:

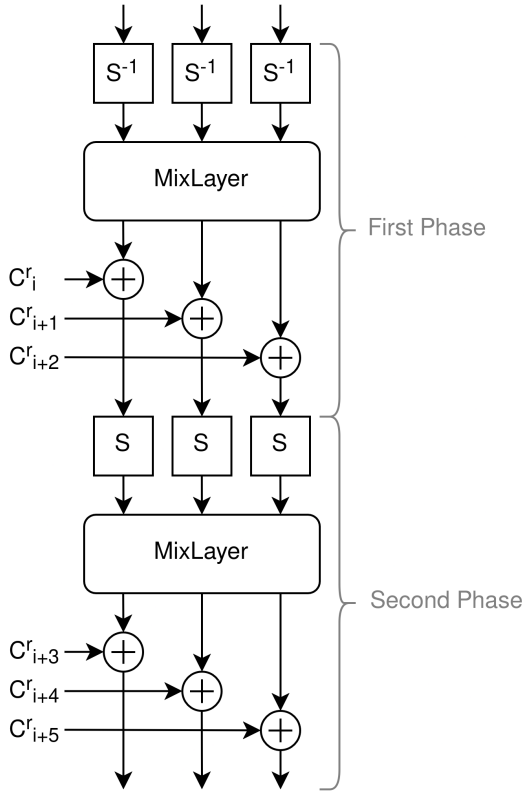


Figure 2.3: $r$-th round of `Marvellous` permutation.

The minimum number of rounds $n$ needed to achieve 128 bits of security is computed with the following formula [3, Tab. 1]:

$$n = 2 \cdot \max\{l_0, l_1, 5\}, \tag{2.5}$$

where

$$l_0 = \max\{n_{\text{Dif}}, n_{\text{HighOrd}}, n_{\text{Int}}\}, \tag{2.6}$$

$$l_1 = \min_N \text{ subject to } \begin{pmatrix} [(16N-1)\cdot t + 1]/4 \\ 2t \cdot N \end{pmatrix}^2, \tag{2.7}$$

which respectively are

- $l_0$ the maximum number of rounds that can be generically attacked via Differential or Linear Cryptanalysis, High Order Differentials or Interpolation;

- $l_1$ the minimum number of rounds to be secure against a Gröebner basis cryptanalysis attack.

Therefore, the most common instantiations of `Rescue` computed using the just explained formula are shown here in Table 2.3:

Table 2.3: `Rescue` instances for $d = 5$.

| $t$ state width | 3 | 4 | 5 | 6/8 |
|---|---|---|---|---|
| $n$ rounds | 14 | 11 | 9 | 8 |

### 2.1.4 `Rescue-Prime`

The hash function `Rescue-Prime` has the same general underlying structure of `Rescue` (subsection 2.1.3), but with minor adjustments to simplify the implementation, which are:

- Simplification of the round constants' derivation function;
- Reduction of the security margin to its half (*i.e.* from 100% to 50%);
- Flipping the order of the S-Boxes.

The authors decided to not override the previous `Rescue` hash function, but to deploy these changes in a new version and to do so they decided to give it a new name to maintain a clear distinction between the two versions. In fact, even the inner permutation function changed name to `Rescue-XLIX`[6].
The round function described for `Rescue` is still valid for `Rescue-Prime`, but with the changes mentioned above, *i.e.* swap of the S-Boxes order.
Because the general structure is the same, also the number of rounds and state width are the same shown in Table 2.3.

### 2.1.5 Griffin

This primitive, born from the results collected by the designs of `GMiMC` and `Rescue`, is the union of both SPN and Feistel networks schemes. The structure on which Griffin has been built upon is called *Horst* and is a revised version of Feistel networks, that grants a more robust defense against algebraic attacks *e.g.* Gröebner basis attacks.
The internal permutation function implemented for Griffin is called Griffin-$\pi$ and has been designed to lower the minimum number of rounds while maintain a certain security level; this is achieved by the introduction of an exponentiation of very high degree on only one element of the state. This choice allows reducing the number of rounds without increasing too much the computational complexity and the number of constraints in the ZK proof.

[6] Rescue Forty-nine

A single round of the permutation function Griffin-$\pi$ is organized as follows:

- *S-Box* (S): depending on the element's position, the transformation of the *S-Box* is different and its formula [12, Eq. 6] is

$$y_i = \begin{cases} x_0^{1/d} & \text{if } i = 0; \\ x_1^d & \text{if } i = 1; \\ x_i \cdot ((L_i^2 + \alpha_i \cdot L_i + \beta_i) & \text{otherwise.} \end{cases}, \tag{2.8}$$

  where

$$L_i = \begin{cases} (i-1) \cdot y_0 + y_1 & \text{if } i = 2; \\ (i-1) \cdot y_0 + y_1 + x_{i-1} & \text{otherwise.} \end{cases}, \tag{2.9}$$

  and $\alpha_i$,$\beta_i$ are constants generated *s.t.* $\alpha_i^2 - 4 \cdot \beta_i$ is a quadratic non-residue modulo $p$;

- *MixLayer* (M): matrix-vector multiplication between an MDS matrix and the state;

- *AddRoundConstant* (ARC): addition of the round constant $c^r$, where $r$ is the round value, to each element of the state. In the last round (*i.e.* $n-1$), the round constant is equal to 0, thus this operation can be omitted in the implementation.

Before computing the Griffin-$\pi$ permutation function on the needed rounds, the state needs to be initialized with the application of the *MixLayer* matrix-vector multiplication for an increased diffusion and better security. Furthermore, the permutation Griffin-$\pi$ can be used with Griffin in sponge as well as in a compression mode, interchangeably.
The complete Griffin-$\pi$ function has been schematized in the following Figure 2.4:

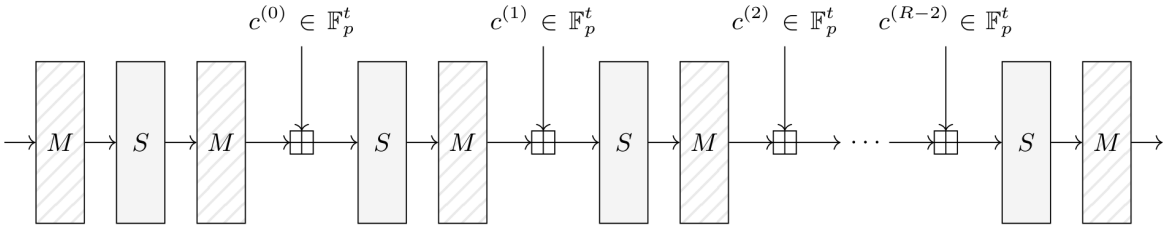


Figure 2.4: Griffin-$\pi$ permutation function [12, Fig. 2].

In the Table 2.4 are provided the most common instantiations of Griffin, where the minimum number of rounds $n$ is computed with the following formula:

$$n \geq \left\lceil 1.2 \cdot \max\left\{6, \left\lceil \frac{2.5 \cdot \kappa}{\log_2(p) - \log_2(d-1)} \right\rceil, 1 + n_{GB} \right\} \right\rceil, \tag{2.10}$$

where $\kappa$ is the bits security level (in our case 128), $p$ is the prime used for the curve, $d$ is the exponent of the S-Box and $n_{GB}$ is the number of rounds needed to defend against Gröebner basis attacks.

Table 2.4: Griffin instances for $d = 5$ [12, Tab. 2].

| $t$ state width | 3 | 4 | 8 |
|---|---|---|---|
| $n$ rounds | 14 | 11 | 9 |

### 2.1.6 `Anemoi`

To improve the efficiency in terms of **evaluation** and **verification** of zero-knowledge proof circuits, the primitive `Anemoi` has been designed exploiting the strength of **CCZ-equivalence**. `Anemoi` is a sponge function whose inner permutation doesn't implement the usual S-Box structure, but instead a new design called `Flystel`.
There are two types of `Flystel` functions:

- Open `Flystel` $\mathcal{H}$: high degree permutation used in plain evaluation (Figure 2.5a);
- Closed `Flystel` $\mathcal{V}$: used for the proof generation because is a low degree function (Figure 2.5b).

The crucial property that these two function have is that they are **CCZ-equivalent**, *i.e.* it is possible to encode the verification of the evaluation of the open `Flystel` using the polynomial representation of the closed `Flystel`. These functions have been implemented in this project using the parameters suggested in the `Anemoi` documentation [6, Sec. 4.4] for odd prime characteristic, which is our case due to our choice of the prime $p$.


(a) Open-`Flystel`.

(b) Closed-`Flystel`.

Figure 2.5: `Flystel` structure [6, Fig. 3].

The management of the state in the `Anemoi` round function is different from the previous primitives, because the state is divided into two halves of length $l = \frac{t}{2}$ (where $t$ is the state size), respectively called $X$ and $Y$, for which the operations slightly differ.
A single round, depicted in Figure 2.6, is composed of the following steps:

- *AddRoundConstant* (ARC): addition of the round constants $c_i^r$ to the half $X$ and of $d_i^r$ to the half $Y$, where $0 \leq i < l$ is the index of the elements of the states and $r$ is the round value;
- *MixLayer* (M): matrix-vector multiplication between a matrix $M_X$ and the $X$ half and also between matrix $M_Y$ and the $Y$ half, where $M_Y$ is the row-permuted version of $M_X$;
- *PHT* [7] (P): application of the PHT to mix the two halves, which is defined as:

$$Y \leftarrow Y + X, \tag{2.11}$$
$$X \leftarrow X + Y\ . \tag{2.12}$$

- *Flystel* (H): application of previously described `Flystel` function to the state.

[7] Pseudo-Hadamard Transformation

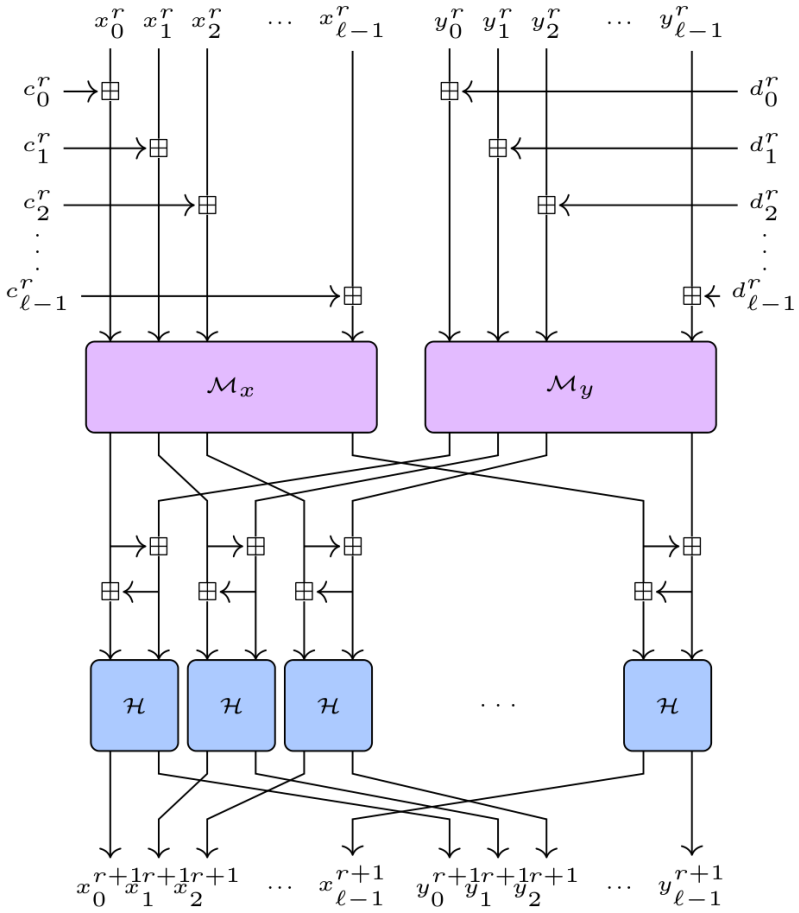


Figure 2.6: $r$-th round of the `Anemoi` permutation [6, Fig. 6].

The `Anemoi` permutation function works by iterating the round function for the needed number of rounds and finally completing the permutation with an additional application of the *MixLayer* operation onto the state.
The most common instantiations of this primitive are shown in the following Table 2.6 and the number of rounds $n$ is computed with the following formula [6, Eq. (2)]:

$$n \geq \max\left\{8, \underbrace{\min\{5, l+1\}}_{\text{security margin}} + \underbrace{\min\left\{r \in \mathbb{N} | \mathcal{C}_{alg(r)} \geq 2\right\} + 2}_{\text{security for algebraic attacks}}\right\}, \tag{2.13}$$

where $l = \frac{t}{2}$ is the width of the halves and $t$ the total size of the state.

Table 2.5: `Anemoi` instances for $d_1 = 5$ [6, Tab. 1].

| $t$ state width | 2 | 4 | 6/8 |
|---|---|---|---|
| $n$ rounds | 21 | 14 | 12 |

### 2.1.7 `Arion`

The last primitive implemented in this project is `Arion`, from which has been built the respective hash function named `ArionHash`. The permutation function inside `Arion` is different from the previous ones because is used the *GTDS*[8] structure [16], which is an alternative solution to the same approach used in the primitive GRIFFIN, where the aim was to maintain a low multiplicative complexity and a low number of rounds, without affecting the security level.
A single round through `Arion`, depicted in Figure 2.8, is computed with the following operations:

- *GTDS*: application of the GTDS function to the state, which is defined as follows [17, Def. 1] (depicted in Figure 2.7):

$$f_i(x_1, \ldots, x_t) = \begin{cases} x_i^{d_1} \cdot g_i(\sigma_{i+1,n}) + h_i(\sigma_{i+1,n}) & \text{if } i < t; \\ x_i^e & \text{if } i = t. \end{cases} , \tag{2.14}$$

  where

  - $t$ is the state width;
  - $d_1$ the smallest positive integer co-prime with $p - 1$;
  - $e$ is the multiplicative inverse of $d_2$[9] modulo $p - 1$, with $d_2$ co-prime with $p - 1$;
  - the function $g_i(x)$ is:

$$g_i(x) = x^2 + \alpha_{i,1} \cdot x + \alpha_{i,2}, \tag{2.15}$$

    where $\alpha_{i,1}$ and $\alpha_{i,2}$ are constants that depend on the round;
  - the function $h_i(x)$ is:

$$h_i(x) = x^2 + \beta_i \cdot x, \tag{2.16}$$

    where $\beta_i$ is a constant that depends on the round;
  - and finally the parameter $\sigma_{i+1,n}$ is equal to:

$$\sigma_{i+1,n} = \sum_{j=i+1}^{t} x_j + f_j(x_1, \ldots, x_t) \ . \tag{2.17}$$

- *MixLayer* (M): matrix-vector multiplication between a fixed matrix and the state;
- *AddRoundConstant* (ARC): addition of the round constant $c_i^r$ to each element $x_i$ of the state, where $0 \leq i < t$ is the element index, $t$ the state wisth and $r$ the number of the round.

**NOTE:**
Note that in the `Arion` documentation [17], the last two operations are seen as a single layer called *AffineLayer* [17, Def. 3].

[8] Generalized Triangular Dynamical System
[9] In our case we decided to use $d_2 = 257$ as default value [17, Tab. 2].

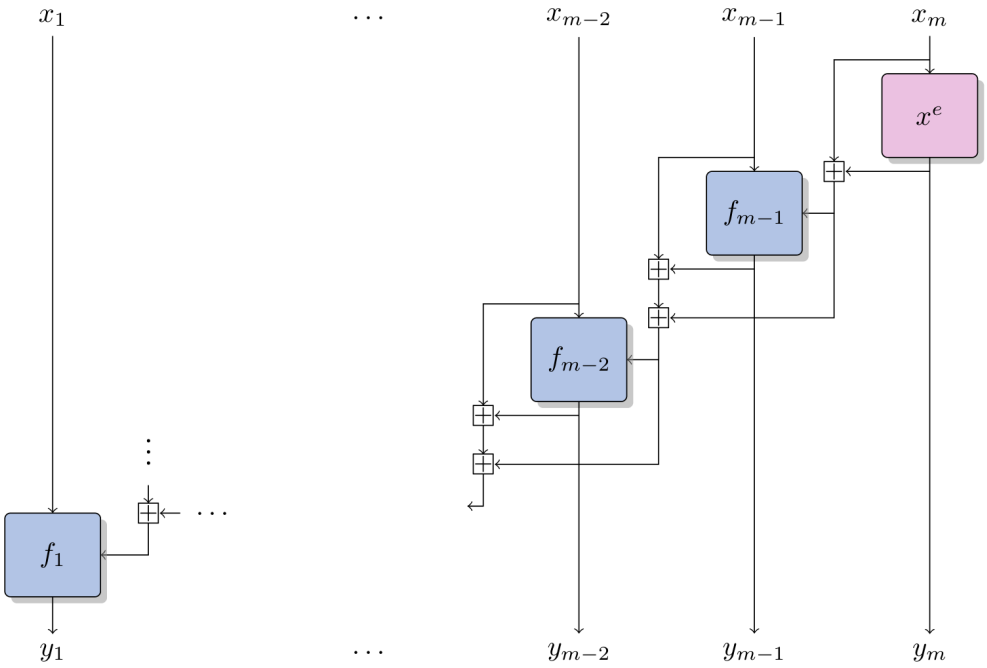


Figure 2.7: `Arion` GTDS function [5, Fig. 1.21].

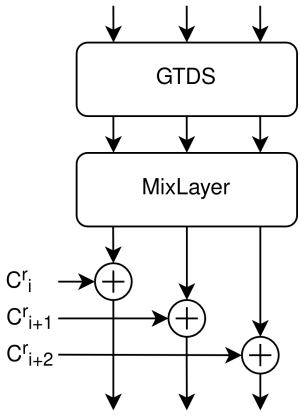


Figure 2.8: $r$-th round of the `Arion` permutation.

Furthermore, a complete permutation in `Arion` is done with an initial application of the *MixLayer* to the state and then the iteration of the round function for the number of rounds needed. The most common instantiations of `Arion` are shown in Table 2.6:

Table 2.6: `Arion` instances for $d_1 = 5$ [17, Tab. 3].

| $t$ state width | 3 | 4/5/6 | 8 |
|---|---|---|---|
| $n$ rounds | 6 | 5 | 4 |

## 2.2 Plonk constraint system

Plonk[10] is a ZK-SNARK constraint proof system based on polynomial commitments. The mechanism of Plonk is based on the representation of a problem $P$ as a logical circuit, which is converted into a system of polynomial equations where the variables correspond to the possible values of the wires of the circuit. There are two types of Plonk constraints:

- 2-wire-input constraints, which are of the form

$$(a \cdot b) \cdot q_M + a \cdot q_L + b \cdot q_R + c \cdot q_O + q_C = 0, \tag{2.18}$$

- 3-wire-input constraints, which are of the form

$$(a \cdot b) \cdot q_M + a \cdot q_L + b \cdot q_R + c \cdot q_O + d \cdot q_F + q_C = 0, \tag{2.19}$$

where

- $a$ is the input value of the left variable and $b$ of the right variable;
- $c$ is the value of the output variable;
- $d$ is the fourth variable;
- $q_M$, $q_L$, $q_R$, $q_O$, $q_F$ and $q_C$ are coefficients that select the operations to be performed, *i.e.* they enables the corresponding wires of the circuit. In the given order they are respectively the selectors for the multiplication, the left wire, the right wire, the output wire, the fourth wire and the constant.

In this project have been used both 2-wire-input and 3-wire-input constraints to build the ZK proofs of the implemented primitives and because most of the permutation functions have some operations in common (e.g. *AddRoundConstant*), we will provide here the explanations of how each implemented operation is represented in the Plonk constraint system.

### 2.2.1 AddRoundConstant

Because the *AddRoundConstant* operation is a simple addition of a constant to an element the state iterated for all the elements of the state, the corresponding constraint is a 2-wire-input constraint where the left wire $q_L$ is enabled (*i.e.* set to 1) and the corresponding $a$ variable is equal to state element, plus the constant selector is set to be equal to the round constant.

### 2.2.2 MixLayer

The mix layer being a sequence of additions in the same row iterated for all the rows, it can naively be represented as a sequence of 2-wire-input constraints, where both selectors for left and right are enabled ($q_L = 1$ and $q_R = \text{matrix}_{i,j}$), the left wire input ($a$) is equal to the temporary addition result and the right wire input ($b$) is equal to the state element. The equation will have the following form

$$1 \cdot \text{result}_i + \text{matrix}_{i,j} \cdot \text{state}_j = 0\ . \tag{2.20}$$

This operation can obviously be optimized by using a 3-wire-input constraint and exploiting the fourth wire input ($d$) to do 3 additions per constraints, but this optimization is not implemented in this project for the reasons explained in the chapter 3.

[10] Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge

### 2.2.3 Power

This operation is the most constraint consuming, because it requires several multiplications depending on the degree of the exponentiation. The optimal way to reduce the number of multiplications is to use the **exponentiation by squaring** technique to reach the desired degree, precisely we will have $\lceil \log_2(d) \rceil$ multiplications, depending on the exponent $d$.

### 2.2.4 Inverse Power

Using the same approach also for the inverse power operation is infeasible due to our choice of prime $p$, thus another approach has been used in this project. The equation that the verifier needs to check is

$$y = x^e, \tag{2.21}$$

where $e$ is the multiplicative inverse of the exponent $d$ modulo $p - 1$. However, if we raise both sides of the equation to the power $d$ we will end with

$$y^d = (x^e)^d = x, \tag{2.22}$$

which for the prover is an evaluation equivalent to the previous equation (2.21). In this way, we just need to compute $y$ and then create a new constraint like we did for the power operation (subsection 2.2.3) and finally add it to the constraints of the proof.

### 2.2.5 S-Box

Being each S-Box different for each permutation function, they have been crafted ad hoc for each primitive via a combination of additions, multiplications, power and inverse power as needed.

### 2.2.6 Permutation functions

As the S-Box, also the permutation functions are all different, thus they have been ad hoc implemented for each primitive by stacking the operations of *AddRoundConstant* (subsection 2.2.1), *MixLayer* (subsection 2.2.2) and *S-Box* (subsection 2.2.5) in the right order to build the round function, and then iterating it for the needed number of rounds.
As an example, we will show in algorithm 1 the pseudocode used for the $\text{Poseidon}_\pi$ full round function.

**Algorithm 1** Proof generation inside $\text{POSEIDON}_\pi$ full round function

$r \leftarrow 0$
**while** $r \leq \frac{n_{full}}{2} - 1$ **do** ▷ Iteration for $\frac{n_{full}}{2}$ rounds
  $i \leftarrow 0$
  **while** $i \leq t - 1$ **do** ▷ AddRoundConstant
    $constraint \leftarrow \text{Constraint}(q_L(1), \text{a}(witness_i), q_C(constant_r))$;
    $witness_i \leftarrow constraint2.\text{c}()$; ▷ Recall that $c$ is the value of output wire
  **end while**

  $i \leftarrow 0$
  **while** $i \leq t - 1$ **do** ▷ S-Box
    $constraint2 \leftarrow \text{Constraint}(q_M(1), \text{a}(witness_i), \text{b}(witness_i))$;
    $power2 \leftarrow constraint2.\text{c}()$;

    $constraint4 \leftarrow \text{Constraint}(q_M(1), \text{a}(power2), \text{b}(power2))$;
    $power4 \leftarrow constraint4.\text{c}()$;

    $constraint5 \leftarrow \text{Constraint}(q_M(1), \text{a}(power4), \text{b}(power4))$;
    $witness_i \leftarrow constraint5.\text{c}()$;

    $i \leftarrow i + 1$
  **end while**

  $result \leftarrow [0, 0, ..., 0]$ ▷ result has size $t$
  $i \leftarrow 0$
  **while** $i \leq t - 1$ **do** ▷ MixLayer
    $j \leftarrow 0$
    **while** $i \leq t - 1$ **do**
      $constraint \leftarrow \text{Constraint}(q_L(1), \text{a}(result_i), q_R(matrix_{i,j}), \text{b}(witness_j))$;
      $result_i \leftarrow constraint.\text{c}()$;
    **end while**
  **end while**
  $witnes \leftarrow result$;
**end while**

## 2.3 Implementation

All the code that has been used to perform the tests over the implemented primitives is hosted the following GitHub repository https://github.com/Crisis82/AOHashes.
The structure of the repository follows the standard hierarchy structure of a Rust project managed with **Cargo**, thus the folders are organized as follows:

- *benches*: it contains the tests used to measure the performances of the primitives, whose results are collected in this paper (see chapter 3);
- *docs*: it contains the official documentation used to implement the primitives and this paper;

- *examples*: it contains a simple usage example to deploy the primitives;
- *src*: it contains almost all the code of the project, especially the implementation of the primitives which are contained under the *permutation* subdirectory;
- *tests*: it contains the code used to perform the tests for the verification of the correctness of the primitives.

For each primitive it has been decided to create two separate modes:

1. **plain evaluation**: this has been implemented in the files called `scalar.rs` and works with values in the BLS12-381 curve;
2. **zero-knowledge proof generation and verification**: inside the `gadget.rs` files have been implemented the operations of the permutation's functions that work with the Plonk constraint system.

Additionally, for each function there is a file `hash_name.rs` (*e.g.* `gmimc.rs`) that defines the permutation and a `params.rs` which dynamically generates the parameters needed, like exponents, constants or matrices, by the permutation function.
It has been decided for a dynamical approach for the parameters' generation for two reasons:

1. follows the elastic nature of AO primitives;
2. better reliability in the parameters generated values, removing the constant worry of having not updated the parameters for the given instances.

**NOTE:**
During the parameters' implementation process, it has been found out that the method provided and suggested by the Dusk Network team for passing the constants and the MDS matrix was faulty, because the generated values were not the same of the ones written on the file.

### 2.3.1 Execution

Thanks to Cargo, the execution of tests or benchmarks is straightforward, via these commands:

```
# for executing tests
cargo test --features={hash_name},zk,encryption

# for executing benchmarks
cargo bench --features={hash_name},zk,encryption
```

The above shell commands execute respectively a test or a benchmark on the *hash_name* primitive provided on all possible evaluations: plain computation, encryption/decryption and zero-knowledge proof. However, is possible to change the evaluation types just by passing as feature only `zk` or `encryption` along with the hash function name. Additionally, for the benchmarks is even possible to evaluate just 1 of the 3 benchmarks available (*encrypt*, *decrypt* or *hash*), for example

```
cargo bench hash --features=gmimc,zk
```

runs the test on the `GMiMC` hash function over *plain* and *zero-knowledge* evaluations, but it does not test the encryption or decryption functions.

# Chapter 3

# Tests

The tests that will be presented in this chapter are divided into four categories, in order to have a complete comparison between the primitives and highlight the point of strength and weakness of each one:

1. *Plain evaluation performance*: the time of computation of the final digest working with scalar values in the BLS12-381 elliptic curve, measured in $\mu s$;
2. Number of *constraints*: the number of constraints needed by the primitive to generate a complete Plonk proof, which gives an idea of the computational cost of the proof;
3. Zero-Knowledge *proof generation*: the time needed to generate a proof measured in $ms$;
4. Zero-Knowledge *proof verification*: the time needed to verify the generated proof measured in $ms$;

All the tests have been performed on the same machine, with the following specifications:

| | |
|---|---|
| CPU | AMD Ryzen 5 3600 |
| GPU | NVIDIA GeForce RTX 2060 |
| RAM | 16 GB |
| OS | Void Linux x86_64 |
| Rustup | 1.27.1 |
| Rustc | 1.85.0-nightly |
| Cargo | 1.85.0-nightly |

All the other cargo dependencies used with their versions are listed in the `Cargo.toml` file inside the project.
Furthermore, to have a fair comparison, any possible optimization proposed by the authors of the primitives has **not** been implemented and for this reason has been removed also the one implemented in the Poseidon primitive for state width $t = 5$. In this way, equal operations have the same computational complexity and an equal number of constraints, allowing a reliable comparison in the tests results.
The exponent used in the power map by the S-Box for all primitives is $d = 5$, which is the smallest integer co-prime with $p - 1$ for our choice of $p$ (see chapter 2). Additionally, each test has been performed for only one primitive at a time without having other programs running in the machine to avoid any possible interference.
Finally, to achieve a certain precision in the results, for each input sample and choice of parameters for the primitive, have been performed 1000 tests. This process has been automated with the adoption of the crate (cargo library) `criterion` and the shell command used to execute each test is:

```
cargo bench hash --features={hash_name},zk
```

It is important to notice that if the reader wants to reproduce the tests, this command alone isn't enough because it has to pay attention to the choice of the maximum evaluation time for `criterion`, otherwise if a primitive is much slower that the others, the tests will be cut off before iterating all the 1000 tests, possibly leading to wrong results.

**NOTE:**
This note is here to remind the reader that not all primitives are built to work on all the tested state widths, that's why there will be some empty cells in the following presented tables.

## 3.1 Plain performance

The first test that we wanted to analyze was the raw performance of the primitives to compute a hash digest, thus working with values in our chosen curve, rather than polynomial constraints.
In the Table 3.1 have been collected the average times of the results for this testing category, expressed in $\mu s$, which have been subsequently plotted in the Figure 3.1, to have also a visual comparison between the timings. Note that for the sake of clarity of the plot, the color of `Rescue` and `Rescue-Prime` is the same, due to the fact that their timings are very close.

Table 3.1: Plain timing performance in $\mu s$ with $d = 5$.

| State width $t$ | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|
| `Anemoi` | | 664 | | 428 | 573 |
| `Arion` | 530 | 606 | 385 | 467 | 508 |
| `GMiMC` | 61 | 63 | 33 | 35 | 41 |
| Griffin | 549 | 440 | | | 198 |
| Poseidon | 46 | 77 | 56 | 78 | 133 |
| `Rescue` | 1641 | 1715 | 879 | 943 | 1264 |
| `Rescue-Prime` | 1631 | 1719 | 878 | 940 | 1261 |

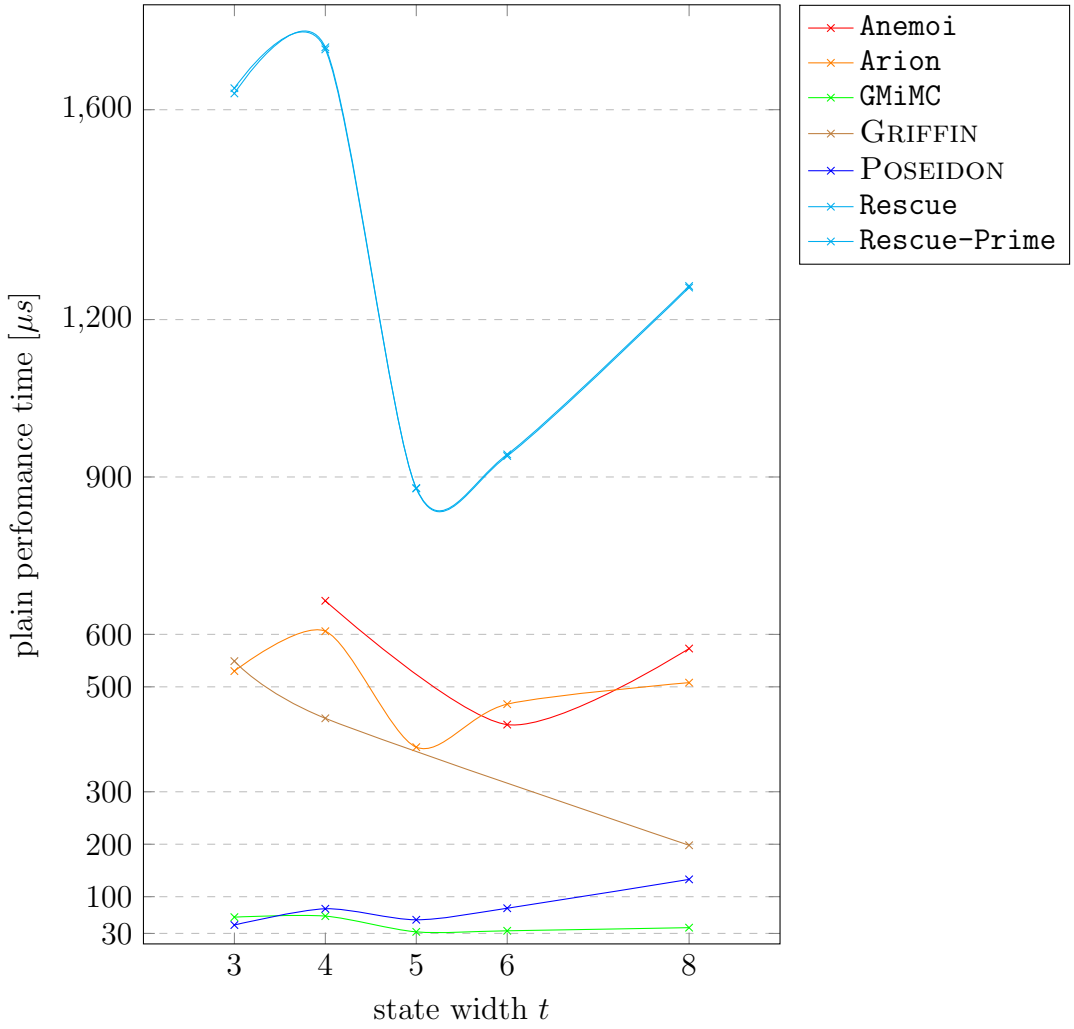


Figure 3.1: Plain timing performance in $\mu s$ with $d = 5$.

As it is possible to notice by the result's table, but even more in the plot, the fastest primitive is `GMiMC`, immediately followed by Poseidon and this is due to the fact that the design of these two primitives is based on keeping the computational cost of each round very low, using only low degree exponentiation in the S-Box, while having a high number of rounds. However, the high number of rounds doesn't affect that much the performance, especially in the case of `GMiMC`, where the timings are pretty much the same for each state width.
It is also interesting the behavior of Griffin, in which the bigger is the state the faster is the computation and the reason behind this is that it's only done one high degree permutation per round, thus having a bigger state doesn't really affect the management of the state, while on the other hand the number of rounds is slightly lower, reducing the overall computational cost.
The remaining primitives (`Anemoi`, `Arion`, `Rescue` and `Rescue-Prime`) have an inconsistent and swinging behavior, especially `Rescue` and `Rescue-Prime`.

## 3.2 Constraints

For the second testing phase it has been analyzed the number of constraints needed both for a single round and for an entire permutation. We analyzed also this aspect for the

zero-knowledge evaluation, in order to have another point of view on the performance of the primitive other than the timings, that are exposed in the following tests section 3.3 section 3.4. The general formulas to calculate the number of constraints for a single round of each primitive are the following:

$$\begin{aligned}\texttt{Anemoi}_{c_r} &= c_{\text{AddRoundConstant}} + c_{\text{MixLayer}} + c_{\text{PHT}} + c_{\text{S-Box}} \\ &= 2l + 2l^2 + 2l + 6l = 2l^2 + 10l \\ &= \frac{t^2}{2} + 5t\end{aligned}$$

$$\begin{aligned}\texttt{Arion}_{c_r} &= c_{\text{GTDS}} + c_{\text{MixLayer}} + c_{\text{AddRoundConstant}} \\ &= (t-1)\cdot 7 + 16 + t + t^2 \\ &= t^2 + 8t + 9\end{aligned}$$

$$\texttt{GMiMC}_{c_r} = c_{\text{ERF}} = t + 2$$

$$\begin{aligned}\textsc{Griffin}_{c_r} &= c_{\text{S-Box}} + c_{\text{MixLayer}} + c_{\text{AddRoundConstant}} \\ &= 6 + (t-2) * 3 + t^2 + t \\ &= t^2 + 4t\end{aligned}$$

$$\begin{aligned}\textsc{Poseidon}_{c_{partial}} &= c_{\text{AddRoundConstant}} + c_{\text{S-Box}} + c_{\text{MixLayer}} \\ &= t + 3 + t^2\end{aligned}$$

$$\begin{aligned}\textsc{Poseidon}_{c_{full}} &= c_{\text{AddRoundConstant}} + c_{\text{S-Box}} + c_{\text{MixLayer}} \\ &= t + 3t + t^2 \\ &= t^2 + 4t\end{aligned}$$

$$\begin{aligned}\texttt{Rescue}_{c_r} &= 2 \cdot (c_{\text{S-Box}} + c_{\text{MixLayer}} + c_{\text{AddRoundConstant}}) \\ &= 2 \cdot (3t + t^2 + t) \\ &= 2t^2 + 8t\end{aligned}$$

$$\begin{aligned}\texttt{Rescue-Prime}_{c_r} &= 2 \cdot (c_{\text{S-Box}} + c_{\text{MixLayer}} + c_{\text{AddRoundConstant}}) \\ &= 2 \cdot (3t + t^2 + t) \\ &= 2t^2 + 8t\end{aligned}$$

We used the variable $c_r$ to denote the constraints per round.
Additionally, because of Poseidon's design, where a round can be full or partial, we decided to show the number of constraints for both, round types and that's why for each cell there are two values, presented with the following notation {partial}/{full}, as you can see in Table 3.2.

Table 3.2: Number of constraints needed for a single round of the permutation with $d = 5$.

| State width $t$ | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|
| Anemoi | | 28 | | 48 | 72 |
| Arion | 42 | 57 | 74 | 93 | 137 |
| GMiMC | 5 | 6 | 7 | 8 | 10 |
| Griffin | 21 | 32 | | | 96 |
| Poseidon | 15/21 | 23/32 | 33/45 | 45/60 | 75/96 |
| Rescue | 42 | 64 | 90 | 120 | 192 |
| Rescue-Prime | 42 | 64 | 90 | 120 | 192 |

Now we extend the above results to an entire permutation, thus we define $c_p$ as constraints per permutation and define the general formulas:

$$\texttt{Anemoi}_{c_p} = \texttt{Anemoi}_{c_r} * n + c_{\text{MixLayer}} = \texttt{Anemoi}_{c_r} * n + 2l^2$$

$$\texttt{Arion}_{c_p} = \texttt{Arion}_{c_r} * n + c_{\text{MixLayer}} = \texttt{Arion}_{c_r} * n + t^2$$

$$\texttt{GMiMC}_{c_p} = \texttt{GMiMC}_{c_r} * n$$

$$\text{Griffin}_{c_p} = \text{Griffin}_{c_r} * n + c_{\text{MixLayer}} = \text{Griffin}_{c_r} * n + t^2$$

$$\text{Poseidon}_{c_p} = \text{Poseidon}_{c_{partial}} * n_p + \text{Poseidon}_{c_{full}} * n_f$$

$$\texttt{Rescue}_{c_p} = \texttt{Rescue}_{c_r} * n$$

$$\texttt{Rescue-Prime}_{c_p} = \texttt{Rescue-Prime}_{c_r} * n$$

Table 3.3: Number of constraints needed for a complete permutation with $d = 5$.

| State width $t$ | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|
| Anemoi | | 400 | | 594 | 896 |
| Arion | 261 | 301 | 395 | 501 | 612 |
| GMiMC | 1640 | 1980 | 2324 | 2672 | 3380 |
| Griffin | 303 | 368 | | | 928 |
| Poseidon | 1317 | 2104 | 2964 | 3960 | 6360 |
| Rescue | 588 | 704 | 810 | 960 | 1536 |
| Rescue-Prime | 588 | 704 | 810 | 960 | 1536 |

Even though Rescue and Rescue-Prime were the two primitives with the highest number of constraints per round, if we consider the entire permutation, which is the most important

aspect, these two are still in the top 3 for the highest number of constraints, but not as much as GMiMC and Poseidon. Is immediately clear that Poseidon is worst in terms of number of constraints per permutation because it was already among the worst primitives considering only a round, but if we take into account its design strategy, it's obvious that the number of constraints per permutations increase exponentially due to its high number of rounds.
Furthermore, even though GMiMC is the second worst's primitive, it's not a bad result considering that the number of rounds needed by GMiMC is almost double of those needed by Poseidon.
On the other hand, thanks to its design that allow to keep the number of rounds very low, Arion is the best primitive in terms of constraints per permutation, even though it wasn't the best in terms of constraints per round.

## 3.3 Proof generation

In the third test type, we focused on the time needed to generate a zero-knowledge proof with the Plonk constraint system, which we have collected in the Table 3.4 and then plotted in the Figure 3.2 for having a visual comparison.

Table 3.4: Proof generation performance's time in $ms$ with $d = 5$.

| State width $t$ | 3 | 4 | 5 | 6 | 8 |
|---|---|---|---|---|---|
| Anemoi | | 659 | | 658 | 659 |
| Arion | 358 | 660 | 356 | 355 | 657 |
| GMiMC | 1962 | 3719 | 2160 | 1972 | 1982 |
| Griffin | 655 | 657 | | | 660 |
| Poseidon | 1958 | 1961 | 1973 | 1959 | 3719 |
| Rescue | 1221 | 1212 | 659 | 664 | 1219 |
| Rescue-Prime | 1212 | 1219 | 659 | 659 | 1213 |

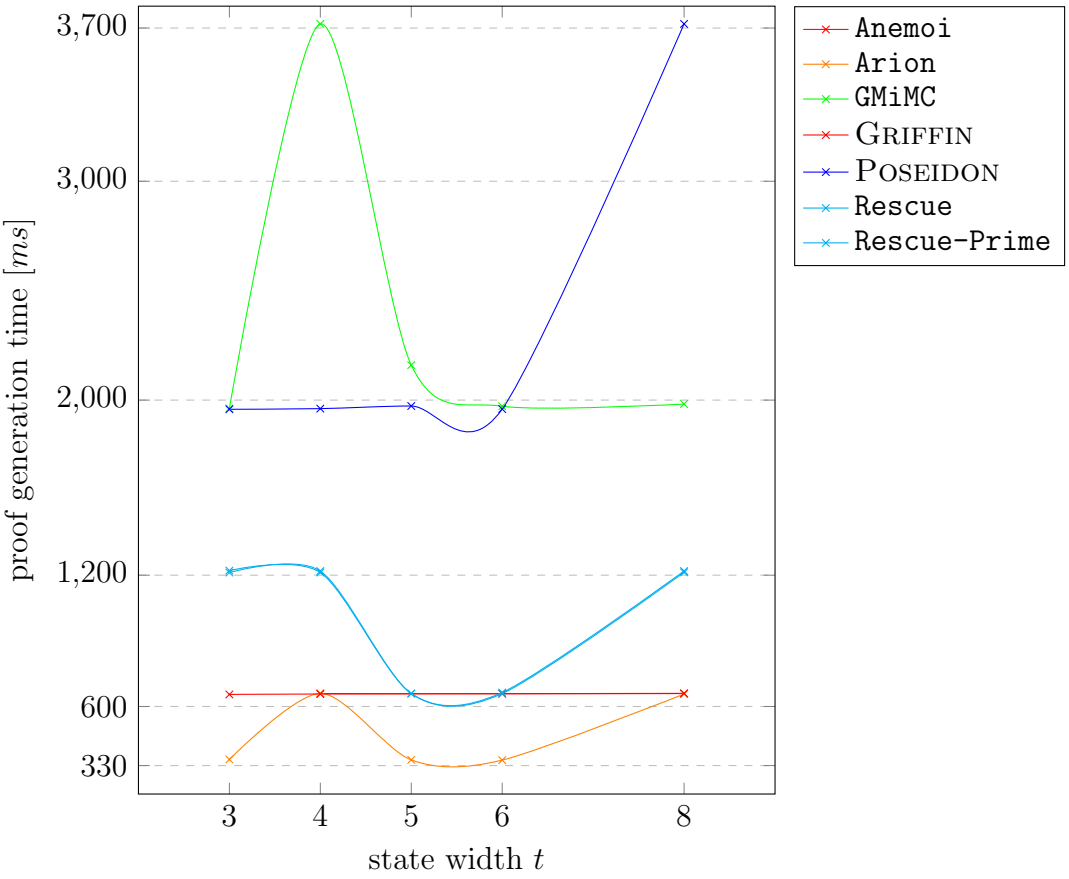


Figure 3.2: Proof generation performance's time in $ms$ with $d = 5$.

For a better understanding of the results, we decided to assign the same color to `Anemoi` and Griffin and also to `Rescue` and `Rescue-Prime`, because their timings are so close that their curves are overlapping.
As it is possible to evince from these results, the fastest primitive for this category of tests is `Arion`, immediately followed by `Anemoi` and Griffin. Nonetheless, considering that the latest two primitives work only with 3 out 5 possible state widths, their results is almost as good as `Arion`. If we have to take also in consideration the stability of the results, `Arion` is a little unstable and whose behavior is closely related to the one of the plain performance test. On the other spectrum of the tests' results, Poseidon is the worst primitive in terms of proof generation as expected by the high number of constraints per permutation highlighted in the previous section 3.2, almost doubling the time in the case of state width $t = 8$ respectively to average time of the other state widths' values.

## 3.4 Proof verification

In the last category of tests we focused on the verification process of the constraint generated in the Plonk syste, and the results have been pretty much the same, around $7.26 \pm 0.03$ $[ms]$ independently of the state width, for all the primitives. Due to these results, it has been decided to not show any table or plot like in the previous sections.

# Chapter 4

# Conclusion

In conclusion, as it was possible to deduct from the tests, the design strategy for primitives plays a key role in how the function behaves with different parameters, highlighting the strengths and weaknesses of each one. This is a crucial step before proceeding with the optimization of these permutation functions, because it's beneficial to decide which applications can fit better the behavior of the primitive, not only from a performance standpoint, but even more from a security one. For example, even though `Arion` is the fastest for a certain choice of parameters, it leads to the same results of `Anemoi` and Griffin for others, thus if an application needs among the requirements also a certain level of stability in the performances while providing a broader choice of parameters, going with `Arion` would not be the best choice. On the other hand, while `GMiMC` has been one of the first primitives that animated some interest in AO cryptography, thus one may think that its performance will not be as good as the newly proposed primitives, if the area of application needs a higher number of hashes computations without the constant need of zero-knowledge proofs, `GMiMC` would be the best choice.
We hope that with this paper we have provided a good overview of the current state of the AO primitives and that with these results new optimizations may come out, or even better that this will spark new ideas for innovative design strategies that will enrich the cryptographic community.

# Chapter 5

# References

## 5.1 Bibliography


[1] Martin R. Albrecht et al. "Feistel Structures for MPC, and More". In: 2019, pp. 151–171. DOI: 10.1007/978-3-030-29962-0_8.

[2] Martin R. Albrecht et al. "MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity". In: 2016, pp. 191–219. DOI: 10.1007/978-3-662-53887-6_7.

[3] Abdelrahaman Aly et al. "Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols". In: 2020.3 (2020), pp. 1–45. DOI: 10.13154/tosc.v2020.i3.1-45.

[4] Guido Bertoni et al. "On the Indifferentiability of the Sponge Construction". In: 2008, pp. 181–197. DOI: 10.1007/978-3-540-78967-3_11.

[5] Clémence Bouvier. "Cryptanalysis and design of symmetric primitives defined over large finite fields". Theses. Sorbonne Université, Nov. 2023. URL: https://inria.hal.science/tel-04327955.

[6] Clémence Bouvier et al. "New Design Techniques for Efficient Arithmetization-Oriented Hash Functions: Anemoi Permutations and Jive Compression Mode". In: 2023, pp. 507–539. DOI: 10.1007/978-3-031-38548-3_17.

[7] Vitalik Buterin. "Quadratic Arithmetic Programs: from Zero to Hero". In: 2016. URL: https://web.archive.org/web/20250325074652/https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649 (visited on 03/25/2025).

[8] Vitalik Buterin. "Understanding PLONK". In: 2019. URL: https://web.archive.org/web/20250325075435/https://vitalik.eth.limo/general/2019/09/22/plonk.html (visited on 03/25/2025).

[9] Dusk Network Team. "Poseidon SNARK-friendly Hash algorithm". In: 2020. URL: https://web.archive.org/web/20250325080300/https://github.com/dusk-network/Poseidon252 (visited on 03/25/2025).

[10] Dusk Network Team. "Pure Rust implementation of the PLONK ZK Proof System". In: 2019. URL: https://web.archive.org/web/20250325080039/https://github.com/dusk-network/plonk (visited on 03/25/2025).

[11] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge*. URL: https://eprint.iacr.org/2019/953.

[12] Lorenzo Grassi et al. "Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications". In: 2023, pp. 573–606. DOI: 10.1007/978-3-031-38548-3_19.

[13] Lorenzo Grassi et al. “On a Generalization of Substitution-Permutation Networks: The HADES Design Strategy”. In: 2020, pp. 674–704. DOI: 10.1007/978-3-030-45724-2_23.

[14] Lorenzo Grassi et al. “Poseidon: A New Hash Function for Zero-Knowledge Proof Systems”. In: 2021, pp. 519–535. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/grassi.

[15] Matter Labs. “Curated list of awesome things related to learning Zero-Knowledge Proofs (ZKP)”. In: 2019. URL: https://web.archive.org/web/20250325075730/https://github.com/matter-labs/awesome-zero-knowledge-proofs (visited on 03/25/2025).

[16] Arnab Roy and Matthias Johann Steiner. “Generalized Triangular Dynamical System: An Algebraic System for Constructing Cryptographic Permutations over Finite Fields”. In: 2024. DOI: 10.1007/978-3-031-82841-6_6.

[17] Arnab Roy, Matthias Johann Steiner, and Stefano Trevisani. “Arion: Arithmetization-Oriented Permutation and Hashing from Generalized Triangular Dynamical Systems”. In: 2023. DOI: 10.48550/arXiv.2303.04639.

[18] Matthias Johann Steiner and Stefano Trevisani. “Arion rust implementation”. In: 2022. URL: https://web.archive.org/web/20250325074033/https://github.com/sca-research/Arion (visited on 03/25/2025).

[19] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. *Rescue-Prime: a Standard Specification (SoK)*. Cryptology ePrint Archive, Report 2020/1143. 2020. URL: https://eprint.iacr.org/2020/1143.

[20] Aztec Network Team. “Plonk Café blog website”. In: 2020. URL: https://web.archive.org/web/20250325075935/https://www.plonk.cafe/ (visited on 03/25/2025).

## 5.2 List of Figures

## 5.3 List of Tables

# List of Algorithms