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 within the constraint-based framework of modern ZK-proof systems like Plonk, which is widely used for privacy-focused applications such as blockchain and secure authentication.

By implementing and testing these primitives in the Rust programming language, using 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, and 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 and scalability of ZK-proof systems.

Chapter 1

Introduction

In recent years, the field of cryptography has seen a surge of interest in Zero-Knowledge (ZK) proofs—a groundbreaking technology that 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 back-and-forth communication between the prover and verifier, making them ideal for decentralized environments like blockchains.

This paper focuses on a specific type of non-interactive ZK proof system called Plonk, which belongs to the SNARK (Succinct Non-interactive ARguments of Knowledge) family. Plonk proofs are based on advanced mathematical structures known as arithmetic circuits, which are translated into constraints and then evaluated as polynomials. This structure makes the proof generation process more efficient while keeping computational complexity low.

However, one challenge with this approach is that many traditional cryptographic primitives are not well-suited to work efficiently within Plonk's constraint-based framework. 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 implemented in the Rust programming language and tested using the Dusk Network's Plonk ZK library. 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.

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 proof, evaluation is important, but 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 (*1*), plus they also implement the primitive Poseidon (*2*), we have decided to implement the 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 in order 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 curve, which is the BLS12-381 one, with $p = 52435875175126190479447740508185965837690552500527637822603658699938581184513$.

2.1 Primitives

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

- `GMiMC`(*3*)
- Poseidon(*4*)
- `Rescue`(*5*)
- `Rescue-Prime`(*6*)
- Griffin(*7*)
- `Anemoi`(*8*)
- `Arion`(*9*)

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 multiplicative inverses, and these primitives are Griffin, `Anemoi` and `Arion`.

2.1.1 GMiMC

The first function implemented in this project 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 $x \to x^3$, which is a design idea by Nyberg and Knudsen, that has been shown to lead to efficient instatiations for SNARKs (*10*). Among the available Feistel network modes, in this project has been implemented only the ERF[2] one, which schema is visible in figure 2.1.

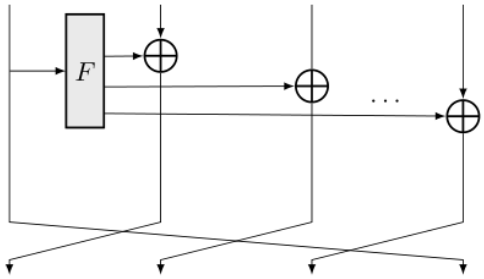


Figure 2.1: Single round of $\texttt{GMiMC}_{\texttt{erf}}$.

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 $y = F(x_0) : x_0 \to x_o^3$, 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:

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

Table 2.1: GMiMC instances.

where the formula used to compute the number of rounds n is:

$$n = \max\left\{2 + 2 \cdot (t + t^2), \lceil 2 \cdot \log_d(p) \rceil + 2t\right\} \tag{2.1}$$

with t the state width, d the exponent of the mapping function F (*i.e.* 3) and p the prime used for our curve.

[1] Generalized MiMC
[2] Expanding Round Function

2.1.2 Poseidon

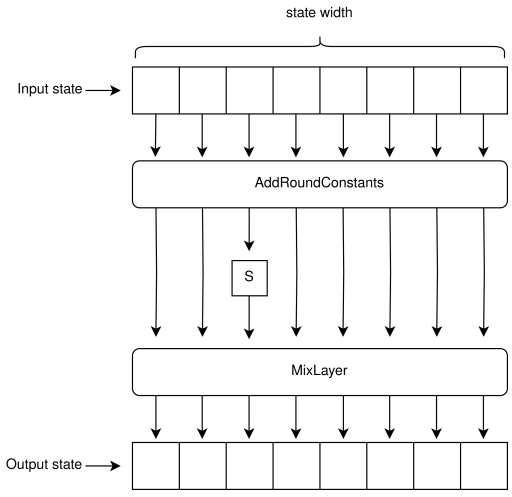
POSEIDON is a **sponge function** based on the hashing designed strategy called HADES, a permutation round function composed of both *full* and *partial* rounds. The decision to opt to this division is to lighten the computational cost of HADES, allowing it to maintain a minimum level of performance even with a high number of rounds. The only difference between a partial and a full round is just that the S-Box[3] is applied only to **one** element of the state, while on the other hand is applied to each element of the state.

NOTE:
It is important to notice that HADES 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 of the state.

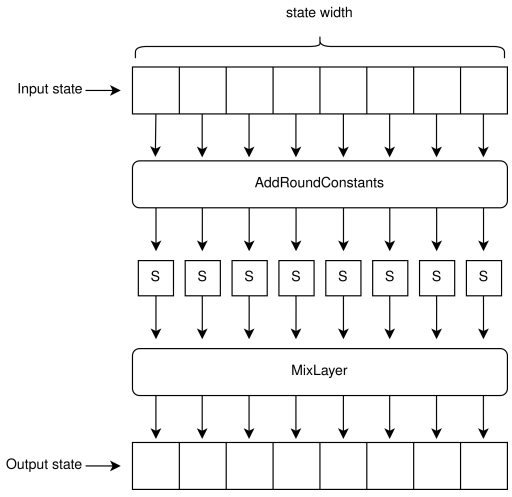
A single round is composed of the following concatenation of functions:

- *AddRoundConstant*: a constant that depends on the round is added to each value of state;
- *S-Box*: the input is exponentiated to an exponent α where $\alpha \geq 3$ and also α is co-prime with $p - 1$. This function is applied singularly to each/one element of the state, depending on the type of round;
- *MixLayer*: 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.

The followings (figure 2.3) are the visual representations of the internal structure of the HADES function:

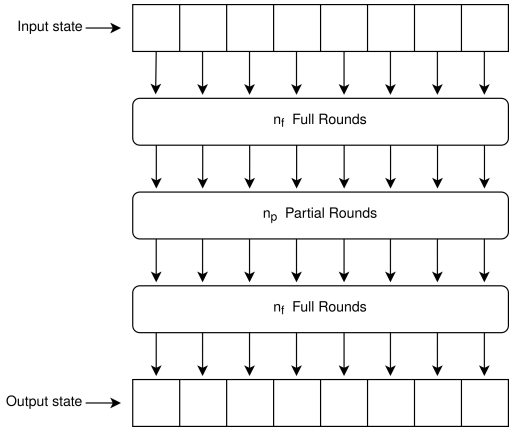


(a) HADES partial round.



(b) HADES full round.

[3]Substitution Box
[4]Maximum Distance Separable



(c) Complete HADES function.

Figure 2.3

Some commonly used instantiations of POSEIDON are shown here in Table 2.2:

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

Table 2.2: POSEIDON instances.

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 on the realization of a secure and robust function, rather than an efficient one, while keeping a simple structure. For this reason is born 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 the different operations.
A single round is organized as follows:

- **First phase**
  - *Inverse S-Box*: application of the inverse power map $x_i \to x_i^{-d}$ to each element of the state;
  - *MixLayer*: matrix-vector multiplication between an MDS matrix and the state;
  - *AddRoundConstant*: addition of a different constant to each element of the state, which also depends on the round and on the phase;
- **Second phase**
  - *S-Box*: application of the power map $x_i \to x_i^d$ to each element of the state;
  - *MixLayer*: matrix-vector multiplication with the same MDS matrix of the first phase;

[5]Substitution-Permutation Network

– *AddRoundConstant*: addition of other different constants to the elements of the state.

The round function is summarized in figure 2.4:

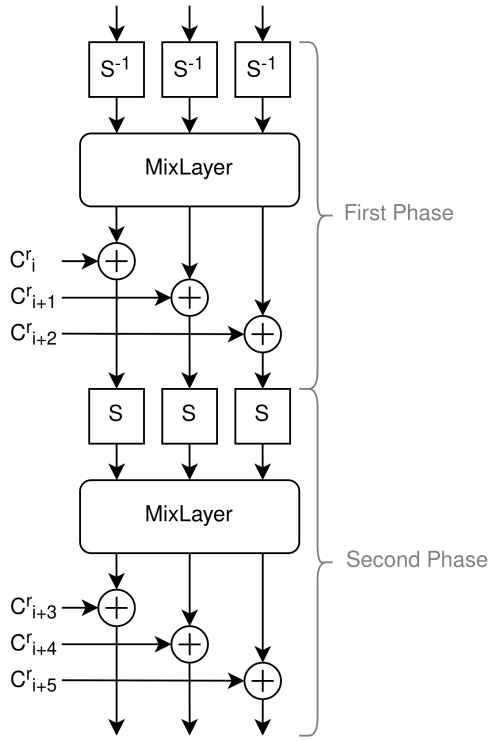


Figure 2.4: Single round of `Marvellous` permutation.

The most common instantiations of `Rescue` are shown in table 2.3:

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

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

2.1.4 `Rescue-Prime`

The hash function `Rescue-Prime` has the same general underlying structure of `Rescue` 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 in 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 it has been 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.* inversion 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.* Gröbner basis attacks.
The internal permutation function implemented for Griffin is called Griffin-π 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.
A single round of the permutation function Griffin-π is organized as follows:

- *S-Box*: depending on the element's position, the transformation of the *S-Box* is different and its formula is

$$y_i = \begin{cases} x_0^{1/d} & \text{if } i = 0; \\ x_1^d & \text{if } i = 1; \\ x_i \cdot ((L_i^2 + \alpha_i \cdot L_i + \beta_i) & \text{otherwise.} \end{cases} \tag{2.2}$$

  where

$$L_i = \begin{cases} (i-1) \cdot y_0 + y_1 & \text{if } i = 2; \\ (i-1) \cdot y_0 + y_1 + x_{i-1} & \text{otherwise.} \end{cases} \tag{2.3}$$

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

- *MixLayer*: matrix-vector multiplication between an MDS fixed matrix and the state;

- *AddRoundConstant*: a round constant that is equal for each element of the state, is added to the state.

The overall round function has been schematized in the following figure 2.5:

[6]Rescue Forty-nine

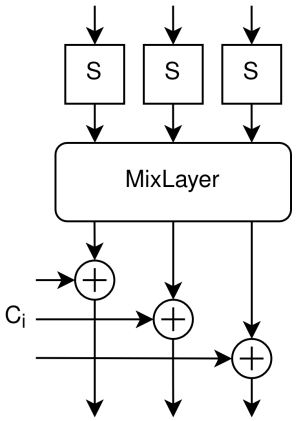


Figure 2.5: Single round of Griffin-π permutation.

Before computing the Griffin-π 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-π can be used with Griffin in sponge as well as in a compression mode, interchangeably.
In the table 2.4 are provided the most common instantiations of Griffin:

t state width	3	4	8
n rounds	14	11	9

Table 2.4: Griffin instances for $d = 5$.

where the minimum number of rounds n is computed with the following formula:

$$n \geq \left\lceil 1.2 \cdot \max\left\{6, \left\lceil \frac{2.5 \cdot \kappa}{\log_2(p) - \log_2(d-1)} \right\rceil, 1 + n_{GB}\right\}\right\rceil \tag{2.4}$$

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

2.1.6 `Anemoi`

To improve the efficiency in terms of **evaluation** and **verification** of zero-knowledge proof's circuits, it has been proposed this new primitive based on **CCZ-equivalence** called `Anemoi`. It is a sponge function which inner permutation doesn't implement the usual S-Box structure, but instead a new design called `Flystel` which allows both a high degree evaluation and a low degree verification. There are two types of `Flystel` gates, which are the open and the closed one and because are CCZ-equivalent, for this project it has been decided to implement only the open one with the documentation's proposed parameters to work in odd prime characteristic (figure 2.6), due to our prime p choice.

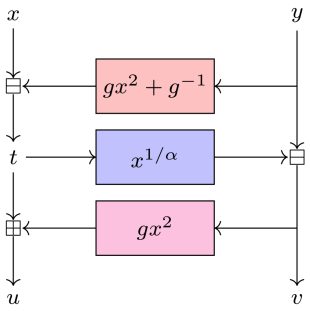


Figure 2.6: $\texttt{Flystel}_p$ in odd prime characteristic.

The management of the state in the `Anemoi` round function is different from the previous primitives, because the state is divided into two halves, respectively called x and y, for which the operations slightly differ.
The round is composed of the following steps:

- *AddRoundConstant*: addition of the round constants to both halves;
- *MixLayer*: 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]: application of the PHT to mix the two halves, which is defined as:

$$Y \leftarrow Y + X \tag{2.5}$$
$$X \leftarrow X + Y \tag{2.6}$$

- *Flystel*: application of previously described open `Flystel` gate to the state.

and the visual representation is the following (figure 2.7):

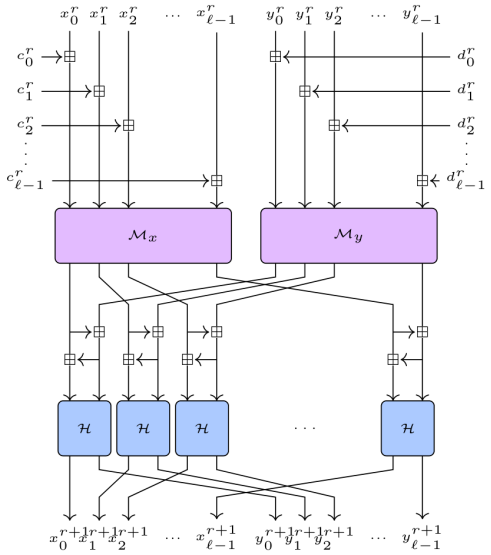


Figure 2.7: Single round of the `Anemoi` permutation.

[7] Pseudo-Hadamard Transformation

The `Anemoi` permutation function works by iterating the round function for the needed number of rounds and finally compute an additional *MixLayer* on the state.
The most common instantiations of this primitive are shown in the following table 2.6:

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

Table 2.5: `Anemoi` instances for $d = 5$.

and the number of rounds n is computed with the following formula:

$$n \geq \max \left\{ 8, \underbrace{\min\{5, l+1\}}_{\text{security margin}} + \underbrace{\min\left\{r \in \mathbb{N} | \mathcal{C}_{alg(r)} \geq 2\right\} + 2}_{\text{security for algebraic attacks}} \right\} \tag{2.7}$$

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

2.1.7 `Arion`

The last primitive implemented in this project is `Arion`, from which has been built the respective hash function named `ArionHash` and although it exists also an aggressive version, called α-`Arion`, this has not been implemented in this project. The permutation function inside `Arion` is different from the previous ones because is used the *GTDS*[8] structure, 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` is computed with the following operations:

- *GTDS*: application of the GTDS structure to the state, which is defined as follows:

$$f_i(x_1, \ldots, x_t) = \begin{cases} x_i^{d_1} \cdot g_i(\sigma_{i+1,n}) + h_i(\sigma_{i+1,n}) & \text{if } i < t; \\ x_i^e & \text{if } i = t. \end{cases} \tag{2.8}$$

  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 modulo $p - 1$, with d_2 coprime with $p - 1$;
  - the function $g_i(x)$ is:

$$g_i(x) = x^2 + \alpha_{i,1} \cdot x + \alpha_{i,2} \tag{2.9}$$

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

$$h_i(x) = x^2 + \beta_i \cdot x \tag{2.10}$$

    with β_i a constant that depends on the round;

[8] Generalized Triangular Dynamical System

– and finally the parameter $\sigma_{i+1,n}$ is equal to:

$$\sigma_{i+1,n} = \sum_{j=i+1}^{t} x_j + f_j(x_1, \ldots, x_t) \tag{2.11}$$

- *MixLayer*: matrix-vector multiplication between a fixed matrix and the state;
- *AddRoundConstant*: addition of the round constant to each element of the state.

where, in the `Arion` documentation([9]), the last two operations are seen as a single layer called *AffineLayer*.
The visual representation of a single round is shown below (figure 2.8):

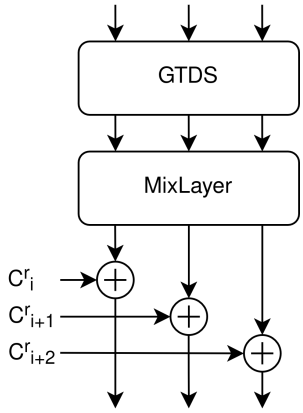


Figure 2.8: Single round of the `Arion` permutation.

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:

t state width	3	4/5/6	8
n rounds	6	5	4

Table 2.6: `Arion` instances for $d_1 = 5$.

2.2 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 hash function: 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.

plus 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.

2.2.1 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 performance*: the time of computation of the final digest working with scalar values in the BLS12-381 curve, measured in μs;
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	16GB
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 following table (3.1) have been collected the average times of the results for this testing category, expressed in μs:

state width	3	4	5	6	8
`Anemoi`		663		428	573
`Arion`	530	606	385	467	508
`GMiMC`	43	45	24	26	30
Griffin	549	440			198
Poseidon	46	77	56	78	133
`Rescue`	1641	1715	879	943	1264
`Rescue-Prime`	1631	1719	878	940	1261

Table 3.1: Plain timing performance in μs with $d = 5$.

And all the above results have been plotted in the following 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.

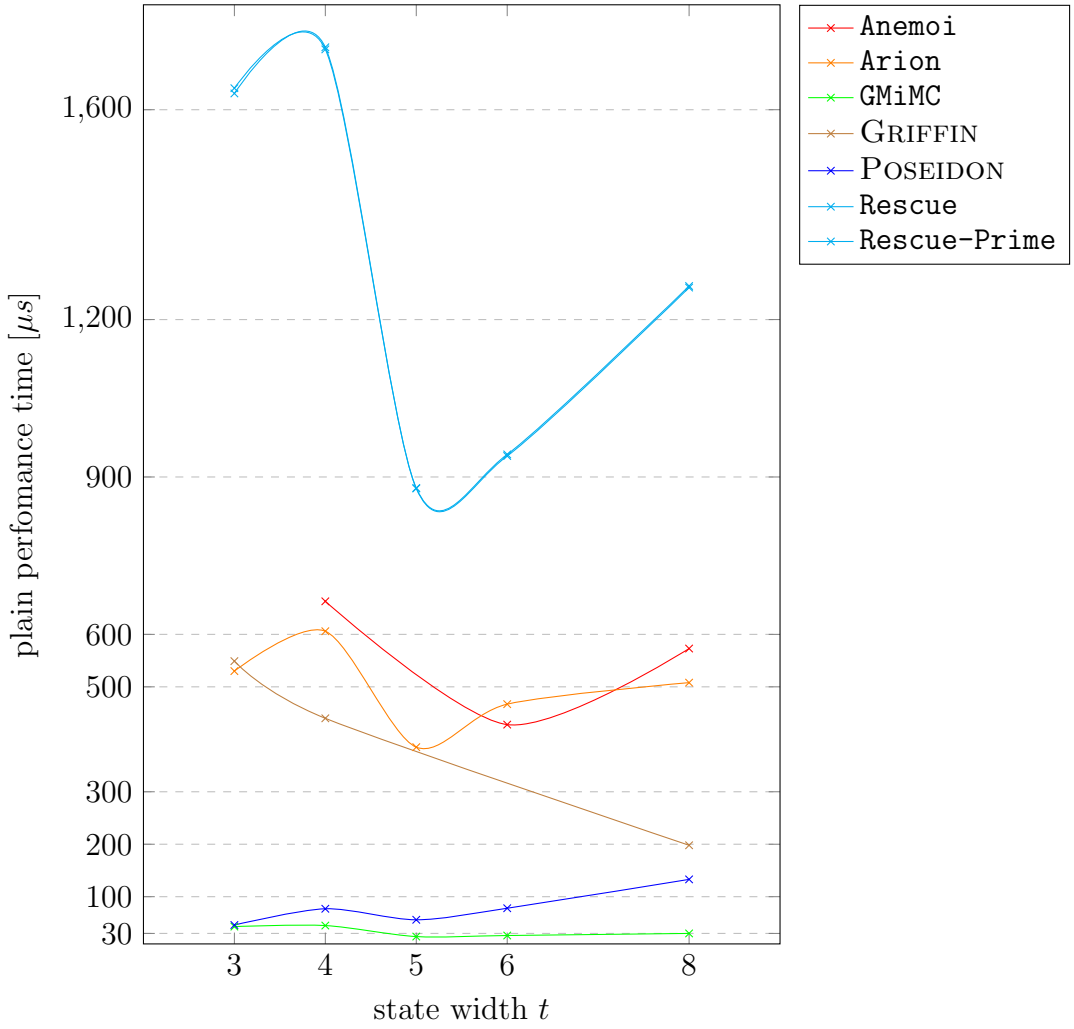


Figure 3.1: Plain timing performance in μs with $d = 5$.

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 sections (3.3) (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}.

state width	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

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

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

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

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

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

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

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

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

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

state width	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

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

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 tests type, we focused on the time needed to generate a zero-knowledge proof with the Plonk constraint system, which we have collected in the following table (3.4):

state width	3	4	5	6	8
Anemoi		657		656	658
Arion	358	660	356	355	657
GMiMC	1959	1956	1955	1966	1966
Griffin	655	657			660
Poseidon	1958	1961	1973	1959	3719
Rescue	1221	1212	659	664	1219
Rescue-Prime	1212	1219	659	659	1213

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

And the following figure (3.2) corresponds to the above data that have been plotted, to have also a visual comparison between the different primitives:

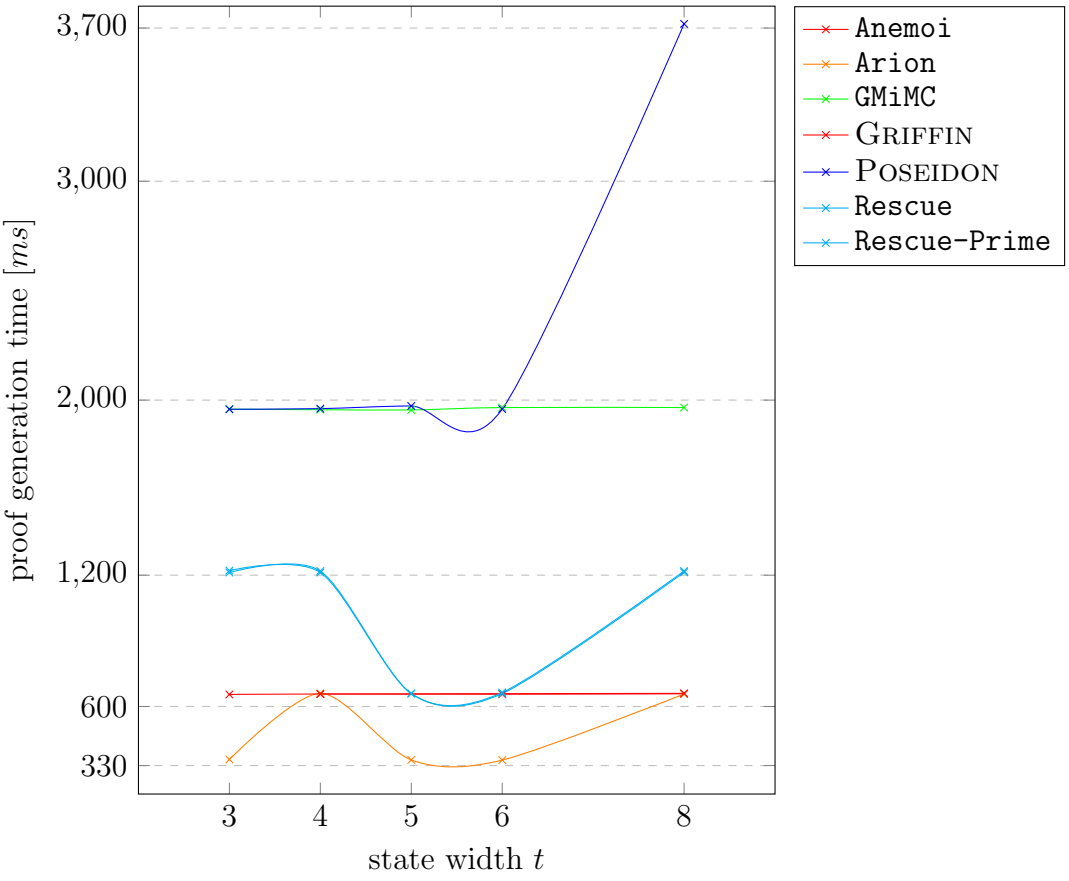


Figure 3.2: Proof generation performance's time in ms with $d = 5$.

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 works 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 system, and the results have been pretty much the same, around 7.26 ± 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

1. Dusk Network Team, *Pure Rust implementation of the PLONK ZKProof System*, GitHub repository https://github.com/dusk-network/plonk (https://dusk-network.github.io/plonk).
2. Dusk Network Team, *Poseidon SNARK-friendly Hash algorithm* (https://github.com/dusk-network/Poseidon252).
3. M. R. Albrecht *et al.*, *Feistel Structures for MPC and More* (https://eprint.iacr.org/2019/397).
4. L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, M. Schofnegger, *Poseidon: A New Hash Function for Zero-Knowledge Proof Systems* (https://eprint.iacr.org/2019/458).
5. A. Aly, T. Ashur, E. Ben-Sasson, S. Dhooghe, A. Szepieniec, *Design of Symmetric-Key Primitives for Advanced Cryptographic Protocols* (https://eprint.iacr.org/2019/426).
6. A. Szepieniec, T. Ashur, S. Dhooghe, *Rescue-Prime: a Standard Specification (SoK)* (https://eprint.iacr.org/2020/1143).
7. L. Grassi *et al.*, *Horst Meets Fluid-SPN: Griffin for Zero-Knowledge Applications* (https://eprint.iacr.org/2022/403).
8. C. Bouvier *et al.*, *New Design Techniques for Efficient Arithmetization-Oriented Hash Functions:Anemoi Permutations and Jive Compression Mode* (https://eprint.iacr.org/2022/840).
9. A. Roy, M. J. Steiner, S. Trevisani, *Arion: Arithmetization-Oriented Permutation and Hashing from Generalized Triangular Dynamical Systems* (https://arxiv.org/abs/2303.04639).
10. M. Albrecht, L. Grassi, C. Rechberger, A. Roy, T. Tiessen, *MiMC: Efficient Encryption and Cryptographic Hashing with Minimal Multiplicative Complexity* (https://eprint.iacr.org/2016/492).
11. *GMiMC homepage* (https://hadesmimc.github.io/gmimc/).
12. M. J. Steiner, S. Trevisani, *Arion rust implementation* (https://github.com/sca-research/Arion).
13. V. Buterin, *Quadratic Arithmetic Programs: from Zero to Hero* (https://medium.com/@VitalikButerin/quadratic-arithmetic-programs-from-zero-to-hero-f6d558cea649).
14. A. Gabizon, Z. J. Williamson, O. Ciobotaru, *PLONK: Permutations over Lagrange-bases for Oecumenical Noninteractive arguments of Knowledge* (https://eprint.iacr.org/2019/953).

15. V. Buterin, *Understanding PLONK* (https://vitalik.eth.limo/general/2019/09/22/plonk.html).
16. Matter Labs, *Curated list of awesome things related to learning Zero-Knowledge Proofs (ZKP)* (https://github.com/matter-labs/awesome-zero-knowledge-proofs).
17. *Plonk Café blog website* (https://www.plonk.cafe/).