Boosting Efficiency and Security in Arithmetization-Oriented Hashing for Zero-Knowledge Proof Systems

Elena Andreeva
TU Wien

Rishiraj Bhattacharyya
University of Birmingham

Arnab Roy
University of Innsbruck

Stefano Trevisani
TU Wien

Abstract

Cryptographic compression functions are a core component of vector commitment schemes, including Merkle tree commitments, which are widely used in modern ZK-SNARK and STARK frameworks. Arithmetization-Oriented (AO) compression functions minimize multiplicative complexity over the framework's native field $\mathbb{F}_p$, making them significantly more efficient than bit-oriented designs in algebraic circuits. To date, AO compression functions have been almost exclusively constructed by applying the Sponge mode to an AO permutation.

In this work, we introduce two novel approaches for building permutation-based AO compression modes: the PA family, based on a Permutation with feedforward Addition, and PAX, as an eXtension of the PA family. We formally establish that, in contrast to the Sponge construction, our modes achieve optimal collision and preimage resistance. We also prove that PAX is indifferentiable from a random oracle, further strengthening its security and composability guarantees. We further show that variable-input-length hash functions can be safely instantiated from the PA(X) modes by applying appropriate domain extenders.

Beyond their strong security guarantees, our modes provide a framework that unifies and extends the description of several recently proposed modes that have been studied via cryptanalysis but do not come with provable security guarantees, including Jive and Trunc, as used in the AO designs `Anemoi` and POSEIDON2.

Finally, through extensive experimental evaluation, we compare the concrete efficiency improvement that our modes offer compared to the Sponge approach over two popular AO permutation designs, POSEIDON-π and *Rescue*. For 128 bits of collision resistance, our modes can achieve up to a 2x speed-up over Sponge for equivalent compression rates in a software implementation. When considering R1CS arithmetization in the Groth16 framework, the PA(X) preimage-verification circuit can be 10% faster than Sponge. In the Plonky2 framework, PA(X) can achieve up to a 60% speed-up.

1 Introduction

Cryptographic hash functions play a critical role in both the security and efficiency of zero-knowledge proof (ZKP) systems such as ZK-SNARKs and STARKs. They are central to ZK computations in blockchain roll-ups and within the proof systems [11, 24]. Hash functions are also used in the Fiat-Shamir heuristic [32] to transform interactive protocols into non-interactive ones, and also serve as a core component in commitment schemes, particularly those based on Merkle trees [51]. The efficiency of these hash computations primarily relies on the efficiency of an underlying *fixed-input-length* (FIL) hash functions, a.k.a. *compression functions*. Furthermore, Merkle Tree evaluations represent a central computational bottleneck for both recursive proofs [24, Figure 9] and Reed-Solomon proximity tests [6]. Consequently, optimizing the efficiency of their underlying cryptographic compression function is a major optimization target.

To put things in perspective, ZK-SNARK and STARK protocols [11, 39] have been instrumental to the wide adoption of ZK proofs in broadly deployed applications, ranging from anonymous blockchains such as ZCash [42], distributed cloud storage systems like FileCoin [62], and to ZK virtual machines such as RiscZero [22]. The growing adoption of these privacy-oriented applications amplifies the need to reduce the overhead of the underlying cryptography and to thwart potential attack vectors. This is paramount to ensuring usability, scalability, and long-term security.

AO compression functions. Since the appearance of the first ZK-friendly compression function MiMChash [1] in 2016, a number of other so-called arithmetization-oriented (AO) compression functions have appeared in the literature. Two of the most widely deployed ones include *Rescue* [3], which is deployed for example in Polygon's Knowledge layer [59] and in Meta's Winterfell library [54]), and POSEIDON [36], which is undergoing integration in the Ethereum blockchain [8] and has been integrated in Cairo [34], Filecoin [61] or *Plonky2* [70], to name a few. Due to their popularity and their modular design, we will consider *Rescue* and

POSEIDON as a comparison baseline throughout our work.

Usually such AO compression function designs are defined over a large prime field $\mathbb{F}_p$, such that their description in terms of $\mathbb{F}_p$-arithmetic can be especially 'simple'. The efficiency advantages of AO compression functions over classical (bit-oriented) ones (e.g. SHA-256) in ZK-based protocols are already widely established in existing literature. We refer the reader to [13, Figure 1] or [23, Table 2] for a comparison of different frameworks and cost metrics, and to [31] for a benchmark of SHA-256 across various ZK-SNARK frameworks.

Regardless of the concrete design strategy, the approach towards building a secure AO compression function has persistently been modular. In particular, most AO compression functions are *permutation-based*: a permutation π is first devised, and then it is used as a main component in some compression *mode* of operation. The research in developing efficient (ZK-friendly) AO compression functions has mainly focused on the efficiency of the core permutation. Surprisingly, the effect of different compression modes on efficiency and security has remained largely unaddressed until recently. The recent AO compression functions `Anemoi` [21] and GRIFFIN [35] propose two permutation-based feed-forward modes explicitly designed to surpass the efficiency of the Sponge mode [15], which has been the predominant, and effectively the only, approach for this task.

Security and efficiency evaluation of AO compression functions. Cryptographic compression functions are traditionally designed to satisfy some given security properties, such as *collision resistance* and *(second) preimage resistance* [64]. While such properties are often sufficient when considering a specific target application, the stronger random-oracle (RO) *indifferentiability* property [25, 49] is also desirable. Indeed, a construction based on a given primitive (say, a permutation) that is proven indifferentiable from a random oracle is secure against *all* generic attacks (i.e. those that don't rely on internal weaknesses of the underlying primitive), and can safely replace the random oracle [9] in any higher-level protocols whose security has been proven in the Random Oracle Model (ROM), such as variants of the Fiat-Shamir transform [32] commonly used to convert interactive protocols into non-interactive ones.

For permutation-based compression modes, their provable security guarantees may be given in the Ideal Permutation Model (IPM). Similarly to the closely related ROM and Ideal Cipher Model (ICM [18]), in the IPM we assume an adversary is given black-box access to a permutation uniformly at random from all possible permutations over a given domain. Yet, even under this common foundational assumption, different modes can exhibit significantly different security bounds. Ultimately, these theoretical guarantees must be realized by a specific compression function algorithm; while the proof provides a guarantee for the generic security under the mode transformation, the security of any instantiated compression function will quite clearly still depend on the cryptanalytic strength of the chosen permutation.

Concerning efficiency, in addition to the usual performance metrics on standard hardware, for AO cryptographic primitives, their *arithmetization cost* is of utter importance. In principle, this cost can be traced back to the *multiplicative complexity* of the associated arithmetic circuit, i.e. the directed acyclic graph representation of the computation in terms of wires, addition and multiplication gates, over the system's native algebraic field (usually $\mathbb{F}_p$). Therefore, formally establishing the efficiency of a given compression function could be done through an estimation of its multiplicative complexity. However, due to the different arithmetization techniques employed by different ZK frameworks (e.g. Rank-1 Constraint Satisfaction [13] or $\mathcal{P}\mathfrak{lon}\mathcal{K}$ [33]), as well as further transformations and optimizations potentially applied to the user-provided description of the computation, it is quite difficult to give an exact comparison between two different designs without relying on benchmarks and experiments over concrete implementations[1].

Concluding, to provide a comprehensive and fair assessment of the security and efficiency of AO compression functions, under distinct modes of operation, one must integrate both formal, provable security analysis, and empirical experiments in support of any efficiency claim.

Sponge-based AO compression functions. The Sponge [15] mode (see Figure 2) is a permutation-based mode to construct cryptographic hash functions as extendable-output functions (XOF). Its security has been studied in the literature, both in the Boolean setting [16, 46, 48, 55] and in the prime field (AO) setting [7, 43, 47].

The two main parameters of the Sponge mode are its *rate* r and its *capacity* c. A compression function can be obtained from a Sponge by fixing the input length to some n multiple of r, and truncating the output stream to some ℓ. In order to achieve $\approx \lambda$ bits of RO indifferentiability, in the ideal permutation model, one must choose a capacity parameter $c \geq \lceil 2\lambda/\log_2(p) \rceil$.

To derive a Sponge compression function (over fixed-length inputs) from a general Sponge (over arbitrary sized inputs), the most established and popular approach is to evaluate the underlying permutation once in a so-called single-iteration Sponge, or 'Sponge-1' for short, which occurs when $r = n$. Concretely, this is the way via which *Rescue* and POSEIDON-π are used to instantiate a compression function. We should note that, for Sponge-1, it was shown that, in the ideal permutation model, the capacity bound can be cut in half [7] (i.e. $c \geq \lceil \lambda/\log_2(p) \rceil$) without affecting the indifferentiability of the construction, although this optimization is not always applicable, as in the common case of ≈ 256-bit prime fields where the bound would entail $c = 1$ for the usual 128-bit security level.

[1] Although experience shows that the 'pen-and-paper' approach can work very well for some arithmetization techniques and frameworks, as will become clear in Section 5.

Table 1: Upper bounds on the collision (COL), preimage (PRE), and RO-indifferentiability (INDIF) advantages against the target m-to-ℓ compression modes, over the alphabet $\mathbb{F}_p$, for adversaries making at most $q < p^\ell$ queries to the underlying primitive. For ease of presentation, the bounds reported here highlight the asymptotically dominant terms. Sponge-1 indicates a single iteration of the Sponge construction, which implies a rate $r = m$, and the capacity c is some arbitrary constant.

Mode	Results	Primitive	COL	PRE	INDIF
FIL-RO	—	$\mathsf{Func}(m,\ell)$	q^2/p^ℓ	q/p^ℓ	0
Sponge	[16,43]	$\mathsf{Perm}(m+c)$	$q^2/p^{\min\{\ell,c\}}$	$q/p^{\min\{\ell,c\}}$	q^2/p^c
Sponge-1	[7,46]	$\mathsf{Perm}(m+c)$	$\min\{q^2/p^\ell, q/p^c\}$	$q/p^{\min\{\ell,c\}}$	q/p^c
PA	**this work**	$\mathsf{Perm}(m)$	q^2/p^ℓ	q/p^ℓ	1
PAX	**this work**	$\mathsf{Perm}(m+c)$	q^2/p^ℓ	q/p^ℓ	q/p^c

Despite the widespread adoption of the Spongemode, optimality of its efficiency and security for AO compression have remained mostly unquestioned. More surprisingly, to the best of our knowledge, a systematic exploration of alternative, *permutation-based AO compression designs* is entirely absent from the literature. We hence set out to explore alternative permutation-based compression modes designs for the AO setting.

Other permutation-based AO compression modes. To the best of our knowledge, no previous permutation-based compression mode, other than the Sponge, has been shown to simultaneously offer formal security guarantees and demonstrable efficiency advantages, in the AO setting. While two other permutation-based feed-forward AO compression modes have been proposed, namely Trunc [35] and `Jive` [21], they lack both a formal security proof, and a direct comparison with Sponge-based compression, making it hard to discern the generic advantages of applying such modes to any given AO permutation.

1.1 Contributions

In this work, we propose a generalized design principle for AO permutation-based compression functions based on feed-forward and compressing/expanding linear transformations. Our main contributions are twofold: first, we prove the security of our designs in the ideal permutation model, by simultaneously establishing a sufficient characterization of the associated linear transformations. Second, we demonstrate a significant efficiency advantage of our modes over the Sponge based AO compression function designs both in terms of evaluation efficiency and arithmetization complexity in popular SNARK frameworks. More precisely, our contributions are summarized as follows:

- **Generalized AO compression functions families.** We propose two families of AO compression modes: the 'Linear transformation-based Compression from a Permutation' modes (PA for short), and the Extended PA modes (PAX). Our modes resemble the classical feed-forward addition approach [74], coupled with specific *pre- and post-processing linear transformations*. Additionally, the PA family (see Figure 1, left) is general enough to fully characterize and encompass the recently proposed truncation (henceforth Trunc) mode [35,37,45] and the `Jive` mode [21].

  The PAX family extends the PA family by allowing for more flexibility (see Figure 1, right), by including additional input expansion and feedforward transformations, effectively providing a mathematical generalization of the concept of 'capacity' used in Sponge terminology.

- **Security proofs.** We prove that the PA family offers asymptotically *optimal* collision and preimage resistance (i.e. matching the random oracle) in the ideal permutation model, when the compressing transformation $\boldsymbol{R}$ is right-invertible. Additionally, we prove that the PAX family is RO-indifferentiable when the feedforward compressing transformation $\boldsymbol{F}$ is right-invertible, and the input expansion transformation $\boldsymbol{L}$ is left-invertible.

  Table 1 gives an overview of the security bounds for both our proposed constructions, and the Sponge mode, illustrating that PA and PAX families can achieve a given collision (COL) and preimage resistance (PRE) security level with a smaller 'capacity' c than required by the Spongeconstruction. Further, PAX also offer improved indifferentiability bounds compared to the Spongeconstruction, and match the single iteration case ('Sponge-1').

  Finally, we discuss and highlight how our modes can be used to securely instantiate Merkle Tree commitment schemes, as well as variable-input-length hash functions.

- **Experiments and comparisons.** We carry out extensive benchmarking to substantiate the practical efficiency gains achieved by our modes, when targeting$\approx$ 128 bits of collision resistance. In particular, we instantiate our modes with either of two of the most popular and deployed AO permutation designs, POSEIDON-π [38] and *Rescue* [3], and compare them against the respective

Sponge-based designs POSEIDON and *Rescue*[2].

In our evaluation, we consider both plain software (x86) performance, and SNARK performance within either the Groth16 [40] or *Plonky2* [70] ZK-SNARK frameworks. The former works natively over large prime fields such as the 254-bit scalar field of the BN254 elliptic curve, and is typically paired with R1CS arithmetization [12], while the latter works natively over the 64-bit Goldilocks field, and employs $\mathcal{P}\mathfrak{lon}\mathcal{K}$ arithmetization [33] instead. In plain execution, PA-based compression functions can be up to $2\times$ faster than their Sponge-based counterparts. For a *Plonky2* preimage verification circuit, we need up to half as many arithmetic constraints, resulting in up to $\approx 60\%$ faster proof generation times. Finally, in a similar setting in the Groth16 framework, proof generation for PA-based compression functions requires up to 10% fewer constraints.

As both standard instances of POSEIDON and *Rescue* are in the single-iteration (Sponge-1) setting, for the sake of completeness, we also compared the plain performance of multi-iteration sponges. For a baseline input size $n = r$, where r is the Spongerate, this approach is clearly equivalent to Sponge-1. For $n > r$, then efficiency is heavily tied to the chosen underlying permutation design. As an example, for a large $8:1$ compression ratio, our PA instance is about $2\times$ slower than multi-iteration Spongewhen considering POSEIDON-π in the Goldilocks field, while on the other hand our mode is $3\times$ faster when considering *Rescue* over the BN254 field.

To get an idea of the potential impact that even relatively small performance improvements can have on a real-world system, we can look at the ZK proofs market predictions [2]. They estimated that, by 2030, web3 applications alone will require nearly 90 billion ZKPs annually, translating to approximately 246 million (approx. 2^{27}) proofs per day. Even assuming, for simplicity, that just one compression call per proof is needed, an improvement of $2ms$ would lead to a saving of about 2^{24} seconds of computations per day. Given that ZK proofs over large arithmetic circuits can easily require many thousands of compression function evaluations as part of Merkle Tree commitments, together with the fact that such computations are often one of the major bottlenecks in the proof systems [24, Figure 9], it becomes evident that the impact is significant in both absolute and relative terms.

Related Work

The study of provably secure compression functions in the ideal permutation and ideal cipher models constitutes an important research direction within cryptography. In what follows, we review selected prior works that, to the best of our knowledge, are most relevant to the present contribution and serve to illustrate the context of this study.

[2]The name of the permutation and the corresponding Sponge-based hash function is the same, but the distinction should be clear from the context.

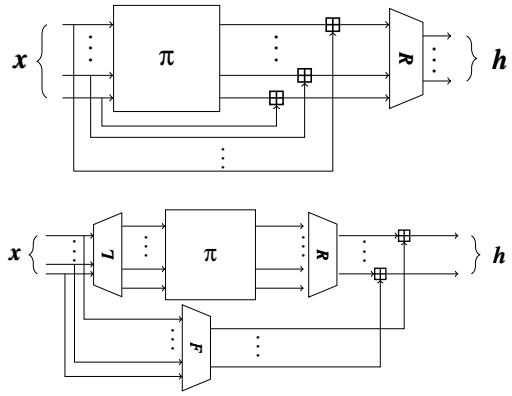

Figure 1: The proposed PA (top) and PAX (bottom) families: $\boldsymbol{x} \in \mathbb{F}_p^m$, $\boldsymbol{h} \in \mathbb{F}_p^\ell$, $\pi \in \mathsf{Perm}\left(\mathbb{F}_p^n\right)$, $\boldsymbol{L} \in \mathbb{F}_p^{n\times m}$, $\boldsymbol{F} \in \mathbb{F}_p^{\ell\times m}$, and $\boldsymbol{R} \in \mathbb{F}_p^{\ell\times n}$. Note that $m = n$ in PA, while $m \leq n$ in PAX.

A prominent line of compression function designs with provable guarantees in the ideal cipher (keyed permutation) research follows the PGV approach [5, 17, 18, 60, 68], which relies on the use of block ciphers as an internal primitive (instead of keyless permutations). In the context of permutation-based compression, the LP modes [65] by Rogaway and Steinberger, largely differ from our proposed modes in two main aspects. First, their compression functions are defined and analyzed exclusively over $\mathbb{F}_2$, whereas our work considers a generic prime field $\mathbb{F}_p$. Second, and more importantly, in their framework, all input, output, and intermediate vectors are treated as 'atomic' entities, with linear operations defined over the field isomorphic to the respective vector space, rather than over the base field. In contrast, in our proposed modes the atomic elements are strictly over the base field $\mathbb{F}_p$, and all associated linear combinations are defined accordingly within that field.

2 Preliminaries

2.1 Background and Notation

We begin by introducing the notation that will be used throughout the work. We generally write numeric variables in lowercase, sets in uppercase, tuples and vectors in lowercase bold, matrices in uppercase bold, functions either in lowercase or uppercase, permutations in lowercase Greek, algebraic structures in blackboard, algorithms and data structures in calligraphic.

Given two integers x and y, $x \mid y$ denotes that x divides y. Given a set S, we denote with $|S|$ its cardinality, with S^n its n-fold Cartesian product, where $S^0 = \emptyset$, with $S^* = \bigcup_{n\in\mathbb{N}} S^n$ its Kleene's closure, and with S^∞ the set of all infinitely-long tuples over S. For a finite set S, we write $x \xleftarrow{\$} S$ to intend the

process of sampling uniformly at random an element of S and naming it x.

Given a tuple $\boldsymbol{x}$, we denote with $|\boldsymbol{x}|$ its length, with $\lfloor \boldsymbol{x} \rfloor_n$ the sub-tuple consisting of its first n elements, with $\lceil \boldsymbol{x} \rceil_n$ the sub-tuple consisting of its last n elements, and with $\boldsymbol{x} \,\|\, \boldsymbol{y}$ its concatenation with the tuple $\boldsymbol{y}$.

$\mathsf{Func}(S,T)$ is the set of all functions from S to T, $\mathsf{Func}(S)$ is the set of all functions over S, and $\mathsf{Perm}(S)$ the set of all the bijections (permutations) over S. For a tuple $\boldsymbol{t}$, t_i denotes its ith element, while $\boldsymbol{t} \lll i$ and $\boldsymbol{t} \ggg i$ respectively denote a left and right rotation by i positions of $\boldsymbol{t}$'s elements. Given a matrix $\boldsymbol{M}$, $m_{i,j}$ denotes the element located at its ith row and jth column, and $\boldsymbol{M}^\mathsf{T}$ denotes the transpose of $\boldsymbol{M}$.

Given a prime number p, $\mathbb{F}_p \cong \mathbb{Z}/p\mathbb{Z}$ denotes the finite field of order p. Note that in this work, $\mathbb{F}_p^*$ denotes the Kleene's closure of $\mathbb{F}_p$, not its underlying multiplicative group.

The n-dimensional vector space over $\mathbb{F}_p$ is interpreted as $\mathbb{F}_p^n$ in the canonical basis, hence we employ tuple notation for $\mathbb{F}_p$-vectors; vector addition and scalar multiplication are defined in the standard way. For convenience, we let $\mathbf{0}_n$, $\mathbf{1}_n$, and $\boldsymbol{e}_{i,n}$ denote respectively the all-zeros, all-ones and the i-th canonical basis vectors over $\mathbb{F}_p^n$; we may drop the dimension from the subscript when clear from the context. We interpret vectors as columns, so that $\boldsymbol{v}^\mathsf{T}$ is the corresponding transposed row (co-)vector. When the choice of p is clear from the context, we use $\mathsf{Func}(m,n)$, $\mathsf{Func}(n)$ and $\mathsf{Perm}(n)$ to intend, respectively, $\mathsf{Func}\left(\mathbb{F}_p^m, \mathbb{F}_p^n\right)$, $\mathsf{Func}\left(\mathbb{F}_p^n\right)$ and $\mathsf{Perm}\left(\mathbb{F}_p^n\right)$.

We interpret linear transformations from $\mathbb{F}_p^m$ to $\mathbb{F}_p^n$ naturally as matrices over $\mathbb{F}_p^{n\times m}$, with $\mathbf{0}_{n\times m}$, $\mathbf{1}_{n\times m}$, and $\boldsymbol{I}_{n\times m}$ denoting respectively the all-zero, all-ones, and (pseudo-)identity matrices over $\mathbb{F}_p^{n\times m}$; again, we drop the dimensionality from the subscripts when clear from the context. Given some matrix $\boldsymbol{M}$, we denote with $\boldsymbol{M}^+$ its (Moore-Penrose [58]) pseudo-inverse. In particular, we recall that $\boldsymbol{M}^+$ is unique, $\left(\boldsymbol{M}^+\right)^+ = \boldsymbol{M}$, and when $\boldsymbol{M}$ has full column (resp. row) rank, $\boldsymbol{M}^+$ is the left (resp. right) inverse of $\boldsymbol{M}$. If $\boldsymbol{M}$ is both square and full rank, then $\boldsymbol{M}^+ = \boldsymbol{M}^{-1}$.

Let $\mathcal{A}$ be an algorithm with n input parameters, we denote with $y \leftarrow \mathcal{A}(x_1,\ldots,x_n)$ the process of running $\mathcal{A}$ over arguments $x_1,\ldots,x_n$ and naming the result y. We generally prefer to use $\xleftarrow{\$}$ rather than $\leftarrow$ to emphasize that $\mathcal{A}$ is a randomized algorithm[3]. Algorithms are assumed to terminate, with decisional ones outputting $\top$ when accepting and $\bot$ when rejecting. We write $\mathcal{A}^{\mathcal{F}_1,\ldots,\mathcal{F}_n}$ to intend that $\mathcal{A}$ is equipped with interfaces $\mathcal{F}_1,\ldots,\mathcal{F}_n$, which can be instantiated by any set of algorithms/oracles adhering to the corresponding interface.

[3]This overloads the meaning of $\xleftarrow{\$}$ depending on whether the right-hand side is a set or an algorithm, which shall be clear from the context.

2.2 AO Cryptography

Arithmetization-Oriented (AO) cryptography deals with the design of cryptographic primitives that have 'simple' arithmetic circuits over some *native* finite field $\mathbb{F}_{p^k}$. Most often, $p \gg 2$ and $k = 1$, although how large p actually is can widely vary depending on the application at hand. For example, for ZK-SNARK systems where the native field is the scalar field of some elliptic curve [33, 40], then usually $p \approx 2^{256}$ [30, 73], but in other settings [11], $p \approx 2^{64}$ or even less [54, 70].

A κ-n-elements AO blockcipher is a family of permutations $E = (E_{\boldsymbol{k}} \in \mathsf{Perm}(n))_{\boldsymbol{k}\in\mathbb{F}_p^\kappa}$. We can equivalently write $E(\boldsymbol{k},\boldsymbol{x})$ to intend $E_{\boldsymbol{k}}(\boldsymbol{x})$, and, abusing notation, $E^{-1}(\boldsymbol{k},\boldsymbol{x})$ to intend $E_{\boldsymbol{k}}^{-1}(\boldsymbol{x})$. Then, $\mathsf{Block}(\kappa,n) = \mathsf{Block}\left(\mathbb{F}_p^\kappa, \mathbb{F}_p^n\right)$ is the set of all κ-n-elements AO blockciphers, and $\mathsf{Block}(n) = \mathsf{Block}(n,n)$ for short.

A function $C\colon \mathbb{F}_p^n \to \mathbb{F}_p^\ell$ such that $n > \ell$ is an n-to-ℓ-elements AO compression function; a function $H\colon \mathbb{F}_p^* \to \mathbb{F}_p^n$ is an n-elements AO hash function; and a function $X\colon \mathbb{F}_p^* \to \mathbb{F}_p^\infty$ is an AO expandable output function (XOF). We generally refer to any of the aforementioned objects as *AO cryptographic primitives*.

As for traditional (bit-oriented) primitives, more complex primitives can be obtained by composing one or more simpler primitives by applying some *mode* $\mathcal{M}$. In this work, we are interested in *compression modes*, that is where the resulting primitive will be a compression function. Traditionally, there have been two main classes of compression modes: permutation-based ones, and blockcipher-based ones. In the context of AO cryptography, the most popular approach towards the design of compression functions has arguably been the (single-)permutation-based one [1, 21, 35, 36, 69]. A (single-)permutation-based m-to-ℓ-elements compression mode can usually be described mathematically as a family $\mathcal{C} = (C_\pi)_{\pi\in\mathsf{Perm}(n)}$. Although other mode-specific parameters may be needed to obtain a full specification of the resulting function, we may assume them as having already being fixed. In this context, we remark that two of the most popular *design strategies* to construct AO cryptographic permutations for compression/hashing are HADES (of which POSEIDON-π is a refined specification) and *Rescue*.

2.3 Provable security

To formally prove the security of a permutation-based compression mode (or generally any cryptosystem), we must start by formalizing a set of assumptions on the permutation itself and on the adversarial capabilities. In this work, we will consider the well-established information-theoretical black box setting: the underlying primitive (in our case, a permutation) is modelled as *ideal*, i.e. randomly sampled from all possible instances, and the adversary is assumed to be a (possibly randomized) algorithm $\mathcal{A}$ that can query an oracle $\mathcal{P}$ implementing such ideal primitive. Since permutations are

invertible, we will assume that the adversary is able to make both *forward* and *backward* queries to $\mathcal{P}$. Proofs in the ideal setting [4, 18, 50, 67, 75]), allow one to ensure that any attack on a concrete instance of the mode must be related to some non-ideal property within the underlying primitive, rather than on an inherent weakness of the mode itself. Additionally, this model is the same adopted to prove the security of the Sponge mode [16], with which we will be comparing the modes that we will propose in Section 3. For ease of exposition, we will usually overload the notation used to refer to some primitive and the oracle that provides the interface required to interact with it.

We express the success probability of $\mathcal{A}$ to break a security notion SEC over a *cryptosystem*[4] $\mathcal{S}_1, \ldots, \mathcal{S}_n$ in function of the number of queries q_i it makes to component $\mathcal{S}_i$, via an *advantage function* $\mathbf{Adv}^{\text{SEC}}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}(\mathcal{A}, q_1, \ldots, q_n)$, and we let $\mathbf{Adv}^{\text{SEC}}_{\mathcal{S}_1,\ldots,\mathcal{S}_n}(q_1, \ldots, q_n) = \max_{\mathcal{A}}\{\mathbf{Adv}^{\text{SEC}}_{\mathcal{C}}(\mathcal{A}, q_1, \ldots, q_n)\}$.

2.3.1 Collision and Preimage Resistance

Two of the most fundamental security notions that a good cryptographic compression function is expected to satisfy are collision and preimage resistance. We give the definitions for permutation-based compression modes:

Definition 1 (Collision advantage)**.** Let $\mathcal{C}$ be an m-ℓ-elements compression mode based on an n-elements permutation. Then, the collision advantage against $\mathcal{C}$ of an adversary $\mathcal{A}$ making at most q queries is:

$$\mathbf{Adv}^{\text{COL}}_{\mathcal{C}}(\mathcal{A}, q) = \Pr\left[\boldsymbol{x} \neq \boldsymbol{x}' \wedge C_\pi(\boldsymbol{x}) = C_\pi(\boldsymbol{x}')\right]$$

Where $\pi \xleftarrow{\$} \mathsf{Perm}(n)$ and $(\boldsymbol{x}, \boldsymbol{x}') \xleftarrow{\$} \mathcal{A}^\pi()$.

Definition 2 (Preimage advantage)**.** Given $\mathcal{C}$, $\mathcal{A}$, and q as per Definition 1, the preimage advantage of $\mathcal{A}$ against $\mathcal{C}$ is:

$$\mathbf{Adv}^{\text{PRE}}_{\mathcal{C}}(\mathcal{A}, q) = \Pr\left[C_\pi(\boldsymbol{x}') = C_\pi(\boldsymbol{x})\right]$$

Where $\pi \xleftarrow{\$} \mathsf{Perm}(n)$, $\boldsymbol{x} \xleftarrow{\$} \mathbb{F}_p^m$ and $\boldsymbol{x}' \xleftarrow{\$} \mathcal{A}^\pi(C_\pi(\boldsymbol{x}))$.

2.3.2 Indifferentiability

The behaviour of an ideal compression function is captured by the fixed-input-length (FIL) random oracle [9] (RO), that is, an oracle implementing a compression function $H \xleftarrow{\$} \mathsf{Func}(m, \ell)$. Then, we could define a compression mode to be secure if it is indistinguishable from a random oracle. However, in an *indistinguishability* game, the adversary has oracle access to C_π rather than π, making it often not a particularly meaningful notion.

The *indifferentiability* notion, introduced by Maurer et al. [49], addresses this issue: in particular, a compression mode that is indifferentiable from a (FIL) random oracle can be used in place of the random oracle even if we assume that the adversary has oracle access to the underlying primitive [25]. Again, for compactness we only give the RO-indifferentiability definition for permutation-based compression functions:

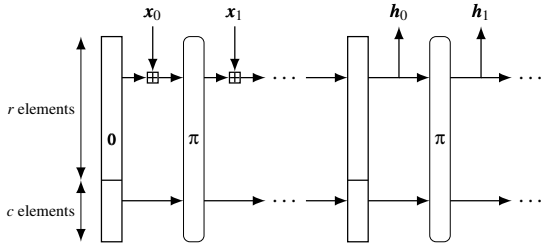

Figure 2: Depiction of the Sponge mode over $\mathbb{F}_p$: π is a permutation over $\mathbb{F}_p^{c+r}$, and $\boxplus$ represents addition over $\mathbb{F}_p^r$.

Definition 3 (Random oracle indifferentiability advantage)**.** Let $\mathcal{C}$ be an m-ℓ-elements AO compression mode based on an n-elements permutation, and $\mathcal{D}^{\mathcal{P},\mathcal{H}}$ be a differentiator adversary that makes at most $q_\mathcal{P}$ queries to the permutation oracle $\mathcal{P}$ and at most $q_\mathcal{H}$ queries to the compression oracle $\mathcal{H}$. Then, given a simulator algorithm $\mathcal{S}^\mathcal{H}$ making at most $q_\mathcal{S}$ queries to the compression oracle $\mathcal{H}$, the random-oracle indifferentiability advantage of $\mathcal{D}$ against $\mathcal{C}$ and $\mathcal{S}$ is:

$$\mathbf{Adv}^{\text{INDIF}}_{\mathcal{C},\mathcal{S}}(\mathcal{D}, q_\mathcal{P}, q_\mathcal{H}) = \Big|\Pr\left[\pi \xleftarrow{\$} \mathsf{Perm}(n); \mathcal{D}^{\pi, C_\pi}() = \top\right] - \Pr\left[H \xleftarrow{\$} \mathsf{Func}(m, \ell); \mathcal{D}^{\mathcal{S}^H, H}() = \top\right]\Big|$$

Note that we omit specifying $q_\mathcal{S}$ in the signature of the advantage function as the notion is in general interesting only when $q_\mathcal{S}$ is 'reasonably' (say, polynomially or better) bounded in $q_\mathcal{P}$, which is the number of queries the simulator would be receiving from the differentiator.

3 New AO Compression Modes

Designing a secure cryptographic scheme is not a trivial task, and the most historically effective way to manage the complexity, has been to adopt modular paradigms and design strategies. For example, in SHA-2 [27] (and older designs like SHA-1 and MD5 [63]) an n-bit hash function is obtained by applying the Merkle-Damgård hashing mode [26, 53] over a $2n$-to-n-bit compression function, which in turn is a variation[5] of the Davies-Meyer [74] compression mode applied to an implicit internal $2n$-bit block cipher (SHACAL [66]).

Another example is the SHA-3 standard (see FIPS 202 [29]), where a family of eXtendable Output Function

[4] We use the term in the sense of [49], hence a compression mode is a cryptosystem.

[5] Addition over $\mathbb{Z}^{16}_{2^{n/8}}$ is employed for the feed-forward instead of the canonical 'XOR'.

(XOFs) are obtained by applying the Sponge mode [15] on some $r+c$-bit permutation. Due to its versatility, the Sponge mode has been widely adopted in the AO setting, and an API specification for Sponge functions over prime fields, along with security proofs, was recently proposed in [43], so it is worth briefly recalling its specification.

3.1 AO Sponges

An AO Sponge function starts from the initial state $\mathbf{0} \in \mathbb{F}^{r+c}$, where r is called the *rate* and c is called the *capacity*. Let $\pi \in \mathsf{Perm}(r+c)$: input blocks $\boldsymbol{x}_i \in \mathbb{F}_p^r$ are *absorbed* in the Sponge by adding them to the rate part of the state, and then applying the permutation π to the entire state. Then, output elements are then *squeezed* out by repeatedly concatenating the rate part of the state while applying in-between π to the state. The Sponge mode can be used for hashing/compression by simply truncating the output to its first ℓ squeezed elements. We denote a fixed-length Sponge construction making a total of a absorbing steps and s squeezing steps with $\mathrm{Sponge}_{a,s}$: this is a compression function whenever $a > s$, and in particular we simply write Sponge_a when $s = 0$.

The capacity parameter c is crucial for the security of the Sponge: to achieve λ bits of DIF security in the ideal permutation model, one should choose $c \geq 2\lambda/\log_2(p)$, as shown in [43]. The standard choice in practice is then to fix $c = \lceil 2\lambda/\log_2(p) \rceil$. More refined analyses [7,46,55] show that, for the first absorbing step, the relaxed bound $c \geq \lambda/\log_2(p)$ is actually sufficient to reach the desired security level, although in this case two different permutations must be employed (one for the initial absorbing step, and another one for the remaining ones).

3.2 The PA Family

We now introduce the PA family of compression modes:

Definition 4 (PA compression modes)**.** Let $\ell < m$ be two positive integers, and let $\boldsymbol{R} \in \mathbb{F}_p^{\ell \times m}$ be a right-invertible matrix. The PA compression mode over $\boldsymbol{R}$ is given by:

$$\mathcal{C}_{\boldsymbol{R}} = \left(C_\pi \colon \mathbb{F}_p^m \to \mathbb{F}_p^\ell ; \boldsymbol{x} \mapsto \boldsymbol{R}(\pi(\boldsymbol{x}) + \boldsymbol{x}) \right)_{\pi \in \mathsf{Perm}(m)}$$

It can be seen that PA offers an optimal compression rate, since the full state of the underlying permutation is used for data compression. In Section 4, we will see that the right-invertibility constraint is key for the security of the mode.

3.3 The PAX Family

While the PA modes achieve optimal rate, as we will see in Section 4, they cannot guarantee indifferentiability. For this reason, we introduce a natural extension of the PA family, called PAX, that can additionally provide indifferentiability (with the appropriate parametrization). PAX includes to additional pre-processing linear transformations:

Definition 5 (PAX compression modes)**.** Let $\ell \leq m \leq n$ be positive integers such that $\ell < n$. Further, let $\boldsymbol{F} \in \mathbb{F}_p^{\ell \times m}$ and $\boldsymbol{R} \in \mathbb{F}_p^{\ell \times n}$ be right-invertible matrices, and $\boldsymbol{L} \in \mathbb{F}_p^{n \times m}$ be a left-invertible matrix. The PAX compression mode over $\boldsymbol{L}$, $\boldsymbol{R}$ and $\boldsymbol{F}$ is then:

$$\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}} = \left(C_\pi \colon \mathbb{F}_p^m \to \mathbb{F}_p^\ell ; \boldsymbol{x} \mapsto \boldsymbol{R}\pi(\boldsymbol{L}\boldsymbol{x}) + \boldsymbol{F}\boldsymbol{x} \right)_{\pi \in \mathsf{Perm}(n)}$$

A note on having separate definitions for PA and PAX: it is immediate to see that PA is a subset of PAX, and hence any result on PAX will also follow for the PA modes. However, in light of the separation results that will be proven in Section 4, together with the fact that PA is sufficient to characterize various compression modes in the literature, we believe that keeping two separate definitions is the better choice.

PA **instances in the wild** The PA (and, consequently, PAX) family of compression modes covers several modes that have appeared in the recent literature about AO designs: by fixing $\boldsymbol{R} = \boldsymbol{I}_{\ell \times m}$, we obtain the Trunc mode [35, 37, 45], usually defined as $C(\boldsymbol{x}) = \lfloor \pi(\boldsymbol{x}) + \boldsymbol{x} \rfloor_\ell$, where $\boldsymbol{x} \in \mathbb{F}_p^m$.

Again, assuming that $\ell \mid m$, and letting $d = m/\ell$, if we fix $\boldsymbol{R}$ to the matrix whose ith row consists of d concatenated copies of $\boldsymbol{e}_{\ell,i}$, we obtain the `Jive` compression mode [21], originally defined as $C(\boldsymbol{x}) = \sum_i \boldsymbol{x}_i + \sum_i \pi(\boldsymbol{x})_i$, with $\boldsymbol{x} \in \left(\mathbb{F}_p^\ell\right)^d$, up to the trivial isomorphism between $\left(\mathbb{F}_p^\ell\right)^d$ and $\mathbb{F}_p^{\ell d} = \mathbb{F}_p^m$.

4 Security Proofs

Below we prove the collision resistance, preimage resistance and indifferentiability of the PA and PAX modes. We show that both PA and PAX are *optimally* collision and preimage resistant, meaning that no attack can achieve a better advantage than the generic attacks (i.e. birthday collisions and exhaustive search respectively), equivalently, security w.r.t. both notions matches the expected security of a random oracle. Additionally, we show that the PAX modes can also achieve random-oracle indifferentiability.

4.1 Collision and Preimage Resistance

Theorem 1 (Collision resistance of PA)**.** *Given ℓ, m, and $\boldsymbol{R}$ as per Definition 4, let $\mathcal{C}_{\boldsymbol{R}}$ be the m-to-ℓ-elements* PA *compression mode. Then, given $q < p^\ell$, it holds that:*

$$\mathbf{Adv}^{\mathrm{COL}}_{\mathcal{C}_{\boldsymbol{R}}}(q) \leq \frac{q^2 - q}{p^\ell - q}$$

Proof. Consider an adversary $\mathcal{A}^\pi$ with π the ideal permutation. Let each query made by $\mathcal{A}$ be collected in the sequence $Q = (Q_1, \ldots, Q_q)$, where $Q_i = (b_i, \boldsymbol{x}_i, \boldsymbol{y}_i)$. When $b_i = 1$, we

say that Q_i is a forward query, meaning that $\boldsymbol{x}_i$ is chosen by $\mathcal{A}$ and $\boldsymbol{y}_i \leftarrow \pi(\boldsymbol{x}_i)$; when $b = -1$, then Q_i is a backward query, i.e. $\boldsymbol{y}_i$ is chosen by $\mathcal{A}$ and $\boldsymbol{x}_i \leftarrow \pi^{-1}(\boldsymbol{y}_i)$. Without loss of generality, we may assume that $\mathcal{A}$ makes only *meaningful* queries, that is, for every Q_i, Q_j, with $i \neq j$, it holds that $\boldsymbol{x}_i \neq \boldsymbol{x}_j \wedge \boldsymbol{y}_i \neq \boldsymbol{y}_j$. We can also assume, at the cost of at most two additional queries ($\mathcal{A}$ could output a guess without consulting the oracle for confirmation), that whenever $\mathcal{A}$ finds a collision, Q contains two queries Q_i and Q_j such that $\boldsymbol{h}_i = \boldsymbol{h}_j$, where $\boldsymbol{h}_i = \boldsymbol{R}(\boldsymbol{y}_i + \boldsymbol{x}_i)$ and $\boldsymbol{h}_j = \boldsymbol{R}(\boldsymbol{y}_j + \boldsymbol{x}_j)$.

Since $\boldsymbol{R}$ is right-invertible, it induces a balanced partition over $\mathbb{F}_p^m$ into p^ℓ equivalence classes $[\boldsymbol{h}]_{\boldsymbol{R}}$ (hence $|[\boldsymbol{h}]_{\boldsymbol{R}}| = p^{m-\ell}$). If $\boldsymbol{R}$ was not right-invertible, the induced partition would have been unbalanced. Given $\boldsymbol{v}, \boldsymbol{w} \in \mathbb{F}_p^m$, and $\boldsymbol{h} \in \mathbb{F}_p^\ell$, we say that $\boldsymbol{w}$ is $\boldsymbol{v}$-$\boldsymbol{h}$-linking (via $\boldsymbol{R}$) when $\boldsymbol{v} + \boldsymbol{w} \in [\boldsymbol{h}]_{\boldsymbol{R}}$ (by commutativity, then $\boldsymbol{v}$ is also $\boldsymbol{w}$-$\boldsymbol{h}$-linking). Let $L_{\boldsymbol{v},\boldsymbol{h}}$ be the set of all $\boldsymbol{v}$-$\boldsymbol{h}$-linking vectors: clearly, $\left|L_{\boldsymbol{v},\boldsymbol{h}}\right| = p^{m-\ell}$ for every choice of $\boldsymbol{v}$ and $\boldsymbol{h}$.

Since π is a random permutation, and $\mathcal{A}$ only makes meaningful queries, either $\boldsymbol{y}_i$ or $\boldsymbol{x}_i$ are selected uniformly at random among exactly $p^m - i \leq p^m - q$ possible choices. Note that, as $\mathcal{A}$ keeps making queries, some choices of $\boldsymbol{x}_i$ (or $\boldsymbol{y}_i$) have higher probability than others in producing a collision with previous queries, since a permutation by definition does not repeat the same output twice; still, we can use $p^{m-\ell}$ to upper bound the number of $\boldsymbol{v}$-$\boldsymbol{h}$-linking values after any number of queries. Consequently, the probability that $C_\pi(\boldsymbol{x}_i) = \boldsymbol{h}$ for any particular $\boldsymbol{h}$ can be upper bounded by $\frac{p^{m-\ell}}{p^m - q} \leq \frac{1}{p^\ell - q}$, and the probability of there being at least one combination $j < i \leq q$ such that $\boldsymbol{h}_i = \boldsymbol{h}_j$ can be bounded by $\sum_{i=0}^{q}\sum_{j=0}^{i} \frac{1}{p^\ell - q} \leq \frac{q^2 - q}{p^\ell - q}$, and the claim follows for adversary $\mathcal{A}$. Since all assumptions on $\mathcal{A}$ were without loss of generality, the claim immediately follows for all adversaries making at most q queries. □

Theorem 2 (Preimage resistance of PA)**.** *Given ℓ, m, $\mathcal{C}_{\boldsymbol{R}}$ and q as in Theorem 1, then it holds that:*

$$\mathbf{Adv}^{\mathrm{PRE}}_{\mathcal{C}_{\boldsymbol{R}}}(q) \leq \frac{q}{p^\ell - q}$$

Proof. The reasoning is similar as for the case of collision resistance. Consider an adversary $\mathcal{A}$ which only makes meaningful queries: at the cost of at most one additional query, we can assume that whenever $\mathcal{A}$ finds a valid preimage for the target digest $\boldsymbol{h}$, the query list Q contains a query Q_i such that $C_\pi(\boldsymbol{x}_i) = \boldsymbol{h}$. Equivalently, it must hold that $\boldsymbol{x}_i + \boldsymbol{y}_i \in [\boldsymbol{h}]_{\boldsymbol{R}}$. Independently of which among $\boldsymbol{x}_i$ and $\boldsymbol{y}_i$ is selected by the adversary, the other one is going to be randomly extracted from a set of size $p^m - i \leq p^m - q$, and the probability of their sum belonging to $[\boldsymbol{h}]$ is at most $\frac{p^{m-\ell}}{p^m - q}$. Summing over all queries, we obtain that the probability of finding a preimage for $\boldsymbol{h}$ is upper-bounded by $\frac{q p^{m-\ell}}{p^m - q} \leq \frac{q}{p^\ell - q}$, and the claim follows. □

Theorem 3 (Collision resistance of PAX)**.** *Given ℓ, m, n, $\boldsymbol{L}$, $\boldsymbol{R}$, and $\boldsymbol{F}$ as per Definition 5, let $\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}}$ be the m-to-ℓ-elements* PAX *compression mode. Given a number of queries $q < p^\ell$, then:*

$$\mathbf{Adv}^{\mathrm{COL}}_{\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}}}(q) \leq \frac{q^2 - q}{p^\ell - q}$$

Proof. We start from the setup laid out in the proof of Theorem 1. Notice that:

$$\boldsymbol{R}\pi(\boldsymbol{L}\boldsymbol{x}) + \boldsymbol{F}\boldsymbol{x} = \boldsymbol{R}\big(\pi(\boldsymbol{L}\boldsymbol{x}) + \boldsymbol{F}'\boldsymbol{x}\big)$$

where $\boldsymbol{F}' = \boldsymbol{R}^+\boldsymbol{F}$ is a left-invertible matrix over $\mathbb{F}^{n\times m}$. Since multiplication by an $n \times m$ left-invertible matrix induces an injective linear map from $\mathbb{F}_p^m$ to $\mathbb{F}_p^n$, there are exactly $p^n - p^m$ vectors which do not have a preimage (either with respect to $\boldsymbol{F}'$ or with respect to $\boldsymbol{L}$).

We say that a query $Q_i = (b_i, \boldsymbol{v}_i, \boldsymbol{w}_i)$, where $\boldsymbol{w}_i = \pi(\boldsymbol{v}_i)$, is *preimage-free* if $\boldsymbol{v}_i$ does not have a preimage with respect to $\boldsymbol{L}$, that is, if there is no $\boldsymbol{y} \in \mathbb{F}_p^m$ such that $\boldsymbol{L}\boldsymbol{y} = \boldsymbol{v}_i$ (equivalently, if $\boldsymbol{L}\big(\boldsymbol{L}^+\boldsymbol{v}_i\big) \neq \boldsymbol{v}_i$). As usual, we assume that if the adversary $\mathcal{A}$ finds a collision, there are two associated queries in the query list corresponding to the collision (this comes at a cost of at most two extra queries), and such queries are of course not preimage-free. Now, if Q_i is a forward query, it can be forced to not be preimage-free as $\mathcal{A}$ can set $\boldsymbol{v}_i = \boldsymbol{L}\boldsymbol{x}_i$. On the other hand, if Q_i is a backward query, $\boldsymbol{v}_i$ is going to be a random element among $p^n - i$ possible values, of which $p^n - p^m - \varepsilon_i$ will make Q_i preimage-free, where ε_i is the number of previously made preimage-free queries. In any case, this only reduces the probability of $\mathcal{A}$ being able to find a collision: even assuming that $\mathcal{A}$ is able to force all backwards query to not be preimage-free, the probability of finding a collision will still be upper bounded for the same reasons as in Theorem 1. □

Theorem 4 (Preimage resistance of PAX)**.** *Given ℓ, n, $\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}}$, and q as in Theorem 3, then:*

$$\mathbf{Adv}^{\mathrm{PRE}}_{\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}}}(q) \leq \frac{q}{p^\ell - q}$$

Proof (Sketch). The reasoning is similar to Theorem 2 with the additional considerations made in Theorem 3, so we omit the full proof. □

4.2 Indifferentiability

Let's now move on to consider indifferentiability from a fixed-input-length random oracle.

Theorem 5 (Differentiability of PA)**.** *Given ℓ, m, and $\mathcal{C}_{\boldsymbol{R}}$ as in Theorem 1, and any simulator algorithm $\mathcal{S}$ making at most* $\mathrm{poly}(\ell)$ *random oracle queries, then:*

$$\mathbf{Adv}^{\mathrm{INDIF}}_{\mathcal{C}_{\boldsymbol{R}},\mathcal{S}}(1,1) \approx 1$$

Proof. Consider the differentiator $\mathcal{D}$ described in Algorithm 1: by construction, when $\mathcal{H}$ is instantiated by C_π and $\mathcal{P}$ is instantiated by π, the $\mathcal{D}$ will accept. On the other hand, if $\mathcal{P}$ is instantiated by any simulator $\mathcal{S}$ making at most $\mathrm{poly}(\ell)$ queries to the random oracle R, whenever $\mathcal{D}$ asks for the inverse $\boldsymbol{x}$ of $\boldsymbol{y}$, the probability that $H(\boldsymbol{x}) = \boldsymbol{R}(\boldsymbol{x}+\boldsymbol{y})$ must be negligible. Since $\boldsymbol{y}$ is randomly sampled, so is $\boldsymbol{x}+\boldsymbol{y}$, independently of $\boldsymbol{x}$, and since $\boldsymbol{R}$ is right-invertible, $\boldsymbol{h}'$ is also going to be random. Hence, if $H(\boldsymbol{x}) = \boldsymbol{h}'$ with non-negligible probability, it would mean that $\mathcal{S}$ has a non-negligible advantage in finding preimages against the random oracle. As this is not possible for a polynomially-bounded $\mathcal{S}$, the claim follows. □

Algorithm 1 Differentiator for PA.

function $\mathcal{D}^{\mathcal{H},\mathcal{P}}() \to \{\bot,\top\}$
  $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_p^n$ ▷ *Pick a random value*
  $\boldsymbol{x} \leftarrow \mathcal{P}(-1,\boldsymbol{y})$ ▷ *Ask for its inverse*
  $\boldsymbol{h} \leftarrow \mathcal{H}(\boldsymbol{x})$ ▷ *Ask for the digest*
  $\boldsymbol{h}' \leftarrow \boldsymbol{R}(\boldsymbol{x}+\boldsymbol{y})$ ▷ *Compute expected value*
  if $\boldsymbol{h} = \boldsymbol{h}'$ **then**
    return $\top$ ▷ *Interacting with construction/ideal primitive*
  return $\bot$ ▷ *Interacting with RO/simulator*

Algorithm 2 The following straightforward adaptation of Algorithm 1 fails to break the indifferentiability of PAX.

function $\mathcal{D}^{\mathcal{H},\mathcal{P}}() \to \{\bot,\top\}$
  $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_p^n$ ▷ *Pick a random value*
  $\boldsymbol{x} \leftarrow \mathcal{P}(-1,\boldsymbol{y})$ ▷ *Ask for its inverse*
  $\boldsymbol{v} \leftarrow \boldsymbol{L}^+\boldsymbol{x}$ ▷ *Compute preimage*
  $\boldsymbol{h} \leftarrow \mathcal{H}(\boldsymbol{v})$ ▷ *Ask for the digest*
  $\boldsymbol{h}' \leftarrow \boldsymbol{R}\boldsymbol{y} + \boldsymbol{F}\boldsymbol{v}$ ▷ *Compute expected value*
  if $\boldsymbol{h} = \boldsymbol{h}'$ **then**
    return $\top$ ▷ *Interacting with construction/ideal primitive*
  return $\bot$ ▷ *Interacting with RO/simulator*

Theorem 6 (Indifferentiability of PAX)**.** *Given ℓ, m, n, and $\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}}$ as in Theorem 3, any $q < p^{n-m}$, any $q_\mathcal{P}, q_\mathcal{H}$ such that $q_\mathcal{P} + q_\mathcal{H} = q$, and the simulator $\mathcal{S}$ described in Algorithm 3, for which $q_\mathcal{S} = 1$, then:*

$$\mathbf{Adv}^{\mathrm{INDIF}}_{\mathcal{C}_{\boldsymbol{L},\boldsymbol{R},\boldsymbol{F}},\mathcal{S}}(q_\mathcal{P}, q_\mathcal{H}) \leq \frac{q}{p^{n-m} - q}$$

Proof. As a warm-up, consider the differentiator $\mathcal{D}$ shown in Algorithm 2, which is a straightforward adaptation of Algorithm 1 to attack PAX: if $m = n$, we are in the same situation as PA: $\boldsymbol{L}$ is fully invertible, and no query-efficient simulator can fool $\mathcal{D}$. However, when $n > m$, with probability $1 - p^{m-n}$, $\boldsymbol{v}$ is going to be preimage-free, that is

Algorithm 3 Simulator to achieve indifferentiability for PAX.

function $\mathcal{S}^{\mathcal{H}}(b \in \{-1,1\}, \boldsymbol{z} \in \mathbb{F}_p^n) \to \mathbb{F}_p^n$
  static $T \leftarrow \emptyset$
  if $b = 1$ **then** ▷ *Forward query*
    if $\boldsymbol{z} \in \mathsf{Dom}(T)$ **then**
      return $T(\boldsymbol{z})$
    $\boldsymbol{v} \leftarrow \boldsymbol{L}^+\boldsymbol{z}$
    if $\boldsymbol{z} = \boldsymbol{L}\boldsymbol{v}$ **then** ▷ *Queried point has a preimage*
      $\boldsymbol{h} \leftarrow \mathcal{H}(\boldsymbol{v})$
      $\boldsymbol{y} \xleftarrow{\$} [\boldsymbol{h} - \boldsymbol{F}\boldsymbol{v}]_{\boldsymbol{R}} \setminus \mathsf{Ran}(T)$ ▷ $[\boldsymbol{x}]_{\boldsymbol{R}}$ *is the equiv. class of* $\boldsymbol{x}$ *via* $\boldsymbol{R}$
    else ▷ *Queried point is preimage-free*
      $\boldsymbol{y} \xleftarrow{\$} \mathbb{F}_p^n \setminus \mathsf{Ran}(T)$
    $T \leftarrow T \cup \{(\boldsymbol{z},\boldsymbol{y})\}$
    return $\boldsymbol{y}$
  else ▷ *Backward query*
    if $\boldsymbol{z} \in \mathsf{Ran}(T)$ **then**
      return $T^{-1}(\boldsymbol{z})$
    $\boldsymbol{x} \xleftarrow{\$} \mathbb{F}_p^n \setminus \mathsf{Dom}(T)$
    $T \leftarrow T \cup \{(\boldsymbol{x},\boldsymbol{z})\}$
    return $\boldsymbol{x}$

$\boldsymbol{L}(\boldsymbol{L}^+\boldsymbol{v}) \neq \boldsymbol{v} \implies \pi(\boldsymbol{L}\boldsymbol{x}) \neq \boldsymbol{y}$; the equality test on Algorithm 2 is going to succeed with probability p^{m-n} when $\mathcal{P} = \pi$ and $\mathcal{H} = C_\pi$, while it is going to succeed with probability $O(p^{-\ell})$ when $\mathcal{P} = \mathcal{S}$ and $\mathcal{H} = H$. If $n-m$ is sufficiently large, say $n-m = \ell$, then the absolute difference of the two probabilities will be negligible, hence $\mathcal{D}$ will have negligible advantage. Exploiting this observation, we can construct the simulator $\mathcal{S}$ described in Algorithm 3, which we will show being able to fool any differentiator $\mathcal{D}$ according to our claimed bound.

First, we make some lossless assumptions on the behavior of the differentiator: we can assume that $\mathcal{D}$ stores two query lists, $\mathcal{Q}_\mathcal{H}$ and $\mathcal{Q}_\mathcal{P}$, that will store the ordered content of the queries (and their relative answers) obtained respectively by interacting with $\mathcal{H}$ and $\mathcal{P}$. A $\mathcal{H}$-query will be of the form $\mathcal{Q}_{\mathcal{H},i} = (\boldsymbol{v}_i, \boldsymbol{h}_i)$, where $\boldsymbol{h}_i = \mathcal{H}(\boldsymbol{v}_i)$, while a $\mathcal{P}$-query will be of the form $\mathcal{Q}_{\mathcal{P},i} = (b_i, \boldsymbol{x}_i, \boldsymbol{y}_i)$, where $b \in \{-1,1\}$ and either $\boldsymbol{x}_i = \mathcal{P}(b,\boldsymbol{y}_i)$ when $b = 1$ or $\boldsymbol{y}_i = \mathcal{P}(b,\boldsymbol{x}_i)$ when $b = -1$. We let $\mathcal{Q}$ be the union, sorted by time, of $\mathcal{Q}_\mathcal{H}$ and $\mathcal{Q}_\mathcal{P}$. We also assume that $\mathcal{D}$ does not repeat the same query twice: observe that for $\mathcal{P}$-queries, asking for both $\mathcal{P}(1,\boldsymbol{x})$ and $\mathcal{P}(-1,\boldsymbol{y})$ is meaningful, as it can reveal inconsistencies when $\mathcal{P} = \mathcal{S}$. Finally, we assume that $\mathcal{D}$ accepts whenever it thinks that it is interacting with π and C_π, and rejects whenever it thinks that it is interacting with $\mathcal{S}$ and H.

We now have to show that the behavior of the simulator $\mathcal{S}$ shown in Algorithm 3 guarantees consistency of the differentiator's view in the two cases. First, we can see that $\mathcal{S}$ behaves like a permutation, thanks to the table T that allows for random choices without replacement, and hence renders forward (resp. backward) queries on the output of previous backward

(resp. forward) queries pointless. Furthermore, $\mathcal{S}$ replies to backward queries just like a random permutation, betting on the fact that such a query Q_i is going to be preimage-free (with respect to $\boldsymbol{L}$, in the sense introduced in Theorem 3) which happens with probability $1 - p^m/(p^n - \varepsilon_i)$, where ε_i is the number of previously made preimage-free queries (obviously, $\varepsilon_i \leq \varepsilon_q \leq q$, and $q \leq p^{n-m}$ by the theorem's hypothesis). When receiving a forward query, $\mathcal{S}$ first checks whether the provided $\boldsymbol{x}$ has a preimage $\boldsymbol{v}$ with respect to $\boldsymbol{L}$: if this is not the case, then the query is going to be preimage-free, so $\mathcal{S}$ will just output a random (without replacement) element; otherwise, it queries the random oracle to find $\boldsymbol{h} = \mathcal{H}(\boldsymbol{v})$, and constructs the output $\boldsymbol{y}$ to guarantee a consistent result with what would be expected if $\boldsymbol{h} = C(\boldsymbol{v})$. Observe that $\boldsymbol{y}$ is randomly chosen (without replacement) from $[\boldsymbol{h} - \boldsymbol{F}\boldsymbol{v}]_{\boldsymbol{R}}$: as long as $q < p^{n-m}$, as per our hypothesis, there will be at least one such value to pick, so that consistency is guaranteed.

We can then conclude that the only way in which $\mathcal{D}$ can tell π and C_π apart from $\mathcal{S}$ and H is when a backward query is not preimage-free: however, the probability that any of the backward queries is not preimage free is upper bounded by the sum of the single probabilities, as per our claim. □

4.3 Application to Merkle Tree Commitments

The main target application of our compression modes is the instantiation of the standard Merkle Tree-based commitment scheme. Formally, a commitment scheme can be defined as a triple of algorithms $(\mathcal{G}, \mathcal{P}, \mathcal{V})$, where $\mathcal{G}$ is the commitment generator algorithm, $\mathcal{P}$ is the opening proof algorithm, and $\mathcal{V}$ is the opening verifier algorithm. In the case of Merkle tree commitments, let C be an m-ℓ-elements compression function. To commit to some message $\boldsymbol{m}$ of length $|\boldsymbol{m}| = (t\ell)^h$ for some h, $\mathcal{G}$ splits $\boldsymbol{m}$ in blocks $\boldsymbol{m}_i$ of length ℓ, which are interpreted as leaves of a full t-ary tree. The root of the tree, obtained by combining nodes by repeated application of C, is the output commitment $\boldsymbol{c}$. To generate an opening for the ith block of the committed message, $\mathcal{P}$ outputs the tuple $\boldsymbol{\omega}$ consisting of $\boldsymbol{m}_i$ and the co-nodes appearing along the path to the root, in some well-specified order. Finally, $\mathcal{V}$ computes $\boldsymbol{c}'$ from $\boldsymbol{\omega}$, and if $\boldsymbol{c}' = \boldsymbol{c}$, it outputs $\top$, otherwise outputs $\bot$.

The security of Merkle Tree commitments has been thoroughly studied in the literature [14, 44, 72], and several interesting security notions can be defined, requiring more or less strong assumptions on the underlying compression function. One of the most important properties is the inability of an adversary to forge openings.

Definition 6 (Merkle opening advantage)**.** Let C be an m-ℓ-elements compression function, and let $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ be the Merkle tree commitment scheme over C. Now let $\mathcal{A}$ be an adversary making at most q queries to C and outputting a triple $(\boldsymbol{m}, i, \tilde{\boldsymbol{\omega}})$. Then, the opening advantage of $\mathcal{A}$ is:

$$\mathbf{Adv}^{\text{OPEN}}_{C,(\mathcal{G},\mathcal{P},\mathcal{V})}(\mathcal{A}, q) = \Pr[\tilde{\boldsymbol{\omega}} \neq \mathcal{P}(\boldsymbol{m}, i) \wedge \mathcal{V}(C(\boldsymbol{m}), \tilde{\boldsymbol{\omega}}) = \top]$$

Opening forgery resistance for Merkle tree schemes can be reduced to the collision resistance of the underlying compression function, as shown for example in [5, Theorem 6] (albeit using slightly different notation). Combining this with either Theorems 1 and 3 gives us the following straightforward corollary:

Corollary 1. *Given a Merkle Tree commitment scheme $(\mathcal{G}, \mathcal{P}, \mathcal{V})$ over an m-ℓ-elements compression function C instantiated from one of the* PA *or* PAX *modes, it holds that:*

$$\mathbf{Adv}^{\text{OPEN}}_{C,(\mathcal{G},\mathcal{P},\mathcal{V})}(\mathcal{A}, q) \leq \frac{q^2 - q}{p^\ell - q}$$

4.4 Dealing with Variable-Length Inputs

While our main focus is to consider fixed-length compression, as the main application of interest for our modes lies in Merkle Tree hashing, for the sake of completeness, we will briefly address hashing variable-length inputs.

As means of example, we will consider the well-known Merkle-Damgård (MD) mode [26,52], when the underlying alphabet is $\mathbb{F}_p$ rather than $\{0,1\}$. Given a compression function $C\colon \mathbb{F}_p^{\ell+r} \to \mathbb{F}_p^\ell$, an initial value $\boldsymbol{h}_0 \in \mathbb{F}_p^\ell$, and a padding function $P\colon \mathbb{F}_p^* \to (\mathbb{F}^r)^*$, the MD hash function $H_{C,P,\boldsymbol{h}_0}\colon \mathbb{F}_p^* \to \mathbb{F}_p^\ell$ maps $\boldsymbol{x}$, to the last element of the sequence defined by the recurrence relation $\boldsymbol{h}_i = C(\boldsymbol{h}_{i-1} \,\|\, P(\boldsymbol{x})_i)$.

Corollary 2 (Collision resistance of MD-PA)**.** *Given $\ell < m$, and $\boldsymbol{R}$ as per Definition 4, let $\mathcal{C}_{\boldsymbol{R}}$ be the m-to-ℓ-elements* PA *compression mode. Additionally, let $P\colon \mathbb{F}_p^* \to \left(\mathbb{F}_p^{m-\ell}\right)^*$ be such that $\nexists \boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z} \in \mathbb{F}_p^*\colon \boldsymbol{x} \neq \boldsymbol{y} \wedge P(\boldsymbol{x}) = P(\boldsymbol{y}) \,\|\, \boldsymbol{z}$. Then, given $q < p^\ell$, it holds that:*

$$\mathbf{Adv}^{\text{COL}}_{\mathcal{H}_{\mathcal{C}_{\boldsymbol{R}},P,\boldsymbol{h}_0}}(q) \leq \mathbf{Adv}^{\text{COL}}_{\mathcal{C}_{\boldsymbol{R}},P}(q)$$

where $\mathcal{H}_{\mathcal{C}_{\boldsymbol{R}},P,\boldsymbol{h}_0}$ is the Merkle-Damgård compression mode over $\mathcal{C}_{\boldsymbol{R}}$ and P, for some $\boldsymbol{h}_0 \in \mathbb{F}_p^\ell$.

Sketch. The condition we imposed on P is commonly referred to as *suffix-freeness*. As shown by Nandi [56] for a binary alphabet, suffix-freeness is a necessary and sufficient condition to ensure that the MD mode $\mathcal{H}$ is *collision-preserving*, i.e. from any collision in $\mathcal{H}_{C,\boldsymbol{R}}$ an adversary can immediately derive a collision in C (regardless of C). The reasoning can be easily extended to a non-binary alphabet like $\mathbb{F}_p$. Together with Theorem 1, this implies collision-resistance of MD-PA. □

A similar result can be proven for PAX due to Theorem 3.

5 Implementation and Experiments

In order to assess the concrete efficiency improvements that can be obtained by employing the PA/PAX modes, we ran

Table 2: Size parameters of PAX and a Sponge required to achieve 128-bits of collision resistance, for different field orders. The input, output and permutation size are respectively of m, ℓ and n $\mathbb{F}_p$ elements.

128-bit COL		$\log_2(p) \approx 256$		$\log_2(p) \approx 64$	
Comp. ratio	Mode	(m,ℓ)	n	(m,ℓ)	n
2:1	PAX	$(2,1)$	2	$(8,4)$	8
	Sponge	$(2,1)$	3	$(8,4)$	12
4:1	PAX	$(4,1)$	4	$(16,4)$	16
	Sponge	$(4,1)$	5	$(16,4)$	20
8:1	PAX	$(8,1)$	8	$(32,4)$	32
	Sponge	$(8,1)$	9	$(32,4)$	36

a series of experiments to compare their performance with popular existing Sponge-based AO compression functions. Throughout our experiments, we will assume a target security level of 128-bits of collision resistance, which is a reasonable requirement in popular contexts such as Merkle-Tree hashing.

In Table 2, we report a few concrete parameterizations of the target modes that achieve this goal, assuming an ideal permutation, to get an idea of what we would expect the results to be when simply considering the size of the underlying permutation. Notice how, as per Theorem 1 and Table 1, unlike Sponge, our modes can achieve this goal without requiring input expansion, and we expect this fact to be the main efficiency enabler. However, we wish to assess the actual impact of this difference in practice.

Choosing the PAX transformation matrices Concerning modes in the PAX family, in our experiments we fix all involved expansion/compression matrices to be the (pseudo-)identity matrix of appropriate shape. However, we highlight an additional, more subtle, advantage that the PAX modes can offer in a practical scenario: it is often the case that a cryptographic permutation is obtained from a block cipher design, and fixing the key to some arbitrary constant value. Many designs follow a Substitution-Permutation Network (SPN) structure, where the permutation layer usually consists of some affine transformation (with the linear part usually being an MDS matrix). Again, many designs, including for example AES/Rijndael, 'optimize away' the last affine transformation as, from an encryption perspective, it does not contribute to the overall security (except possibly for related-key attacks [28]). On the other hand, this omission can have an effect when compressing: for example, if the output is simply truncated (equivalent to applying a pseudo-identity matrix), then we can skip also the last substitution layer. Of course, one can always amend the underlying's primitive implementation depending on the use-case, but PAX allows this additional flexibility directly at the mode level.

Target permutations In order to provide a fair and comprehensive comparison for the parameter choices that we presented in Table 2, we consider two of the most popular AO permutation designs, POSEIDON-π [36] and *Rescue* [3]. POSEIDON-π follows a so-called partial-SPN structure, where the initial and final rounds consist of a standard substitution-permutation network paradigm, while in the intermediate rounds the S-box is applied only to a single component of the state; the S-box is a low degree map, so that the number of rounds, especially intermediate ones, required to resist cryptanalysis is relatively high. *Rescue*, on the other hand, follows a 'double SPN' structure where, in each round, a low and a high degree S-box are interleaved by the mixing layer; since each round is 'high-degree', fewer (but 'heavier') rounds are needed to reach the same security level of POSEIDON-π (in particular, we chose the number of rounds as proposed in [69]).

We target compression rates of 2:1, 4:1 and 8:1, for the smallest output size that can satisfy the reported security target; 2:1 compression is particularly important as it is most commonly used in binary trees. So, when we refer to a certain compression ratio in our results, this is instantiated, depending on the field, according to the actual parameters given in Table 2.

Sponge parametrization We parametrize the Sponge-based compression functions over each of the target permutations (*Rescue*and POSEIDON-π) as suggested in the respective proposals. Namely, both cases propose using a single-iteration Sponge with the standard capacity bound, as shown in Table 2.

Target fields As for the target prime fields, we will consider the ≈ 254-bits scalar field of the BN254 elliptic curve [73], which is relevant for pairing-based ZK-SNARK systems [19, 33, 42, 57], and the 64-bit Goldilocks field, which is relevant for FRI-based ZK-STARK systems [10, 11]. In particular, BN254 is available in the `libsnark` [12], which implements the Groth16 [40] ZK-SNARK system, while Goldilocks is available in the `plonky2` library, which implements its namesake ZK-SNARK system [70].

Environment setup All benchmarks were executed in a native Clear Linux 43160 instance installed on a machine equipped with an Intel Core i9–13900KF CPU clocked at 3.0 GHz, and 64GB of DDR5–4800 RAM. Unless otherwise specified, all experiments were sequential/single-threaded, their execution affinity was set to P-core #0, hyper-threading and frequency boosting technologies were disabled, and the CPU governor was set to 'performance'.

C++ code was compiled with the Intel DPC++ 2025.1 compiler, and raw timing measurements were obtained via the high resolution clock provided by the standard library. Rust code was compiled with the `rustc` 1.83.0-nightly compiler,

Table 3: Plain performance of the target compression functions for a specialized implementation of the underlying field arithmetic. Each result is the average, normalized by the batch size, of 100 batches of 10 executions each, after 10 warm-up batches. The standard deviation is always below 1% of the average, and omitted for clarity.

Comp. ratio	BN254 PA	BN254 Sponge	Goldilocks PA	Goldilocks Sponge
POSEIDON-π				
2:1	**9.4 µs**	17.5 µs	**3.5 µs**	7.3 µs
4:1	**28.2 µs**	42.4 µs	**12.2 µs**	18.6 µs
8:1	**103.7 µs**	130.4 µs	**141.5 µs**	265.4 µs
Rescue				
2:1	**334.6 µs**	353.2 µs	**16.8 µs**	26.2 µs
4:1	**372.0 µs**	382.9 µs	**36.2 µs**	46.9 µs
8:1	**553.5 µs**	626.1 µs	**117.7 µs**	153.5 µs

and benchmark results were obtained via the `Criterion.rs` crate [41]. In both cases, compiler optimizations, native microarchitectural instruction dispatch, and link-time optimizations were enabled.

5.1 Plain Performance

We start by presenting the plain (a.k.a. native) performance of the target compression functions. Our results are reported in Table 3. Clearly, arithmetic over the smaller Goldilocks field is faster than over the larger BN254 field. This can be seen especially in *Rescue* instances, where evaluating the inverse map is much faster. However, this benefit is in part mitigated by the fact that the permutation has a larger internal state (in terms of field elements). As expected, PA is in all cases more efficient than Sponge, particularly over instances where the linear layer becomes a bottleneck. Indeed, it is anywhere between 1.2–2× faster when considering POSEIDON-π, while being 5–50% faster when considering *Rescue*. For more benchmarks over different arithmetic backends, we refer the reader to Section A.

5.1.1 Iterated Sponge compression.

Although the authors of both POSEIDON and *Rescue* propose to use wide, single-iteration Sponge instances to compress medium-sized inputs, one may also choose fix a small permutation (say, the one used to achieve 2:1 compression) and iterate it over again until the entire message is digested. In such a scenario, we would naturally expect the overall compression cost to scale exactly linearly. As expected, the experimental results shown in Table 4 confirm this hypothesis. For both POSEIDON-π and *Rescue* permutations, we can see that there is a sweet-spot after which the iterated Sponge becomes more efficient than widening the state. In general, we would expect such a sweet-spot to exist at some point, as the internal state of the permutation will keep growing, however, as we can see in the case of *Rescue*, the smallest permutation choice is not necessarily the best.

Table 4: Plain performance of the iterated Sponge to compress larger inputs when fixing a small permutation. The underlying permutation in all settings is the one used for the 2:1 setting of Table 3.

Compression ratio	BN254	Goldilocks
POSEIDON-π		
2:1	17.6 µs	7.2 µs
4:1	34.9 µs	14.3 µs
8:1	71.3 µs	28.5 µs
Rescue		
2:1	353.2 µs	26.1 µs
4:1	703.6 µs	52.2 µs
8:1	1.41 ms	104.4 µs

5.2 SNARK performance

As the primary application for Arithmetization-Oriented compression functions are modern Zero-Knowledge Proof systems, specifically ZK-SNARKs and ZK-STARKs, we considered for our comparisons two concrete frameworks, Groth16 [40] and *Plonky2* [70], which are based on very different principles. Without going into the details, the former is required to work over a field of order $\approx 2^{256}$ to guarantee ≈ 128 bits of security (as it fundamentally relies on the hardness of the discrete logarithm), while the latter can work even on smaller fields (of order $\approx 2^{64}$), as its security mainly relies on the collision resistance of an internal hash function.

The general structure required to generate and verify a proof of computation in all SNARK frameworks is similar: first, we must provide a representation of the computation (a.k.a. the 'circuit') following the framework's arithmetization technique; then, for each instance of the computation, the prover must supply a set of witnesses, from which an argument of knowledge is generated, that can then be checked by the verifier. In general, the most expensive task is proof generation, which is the main bottleneck to the scalability of SNARK systems [2], which depends on the arithmetization complexity of the target computation.

In our experiments, we will consider as our target a preimage verification circuit, that can be used to assess whether the prover knows a preimage, with respect to some chosen compression function, of some publicly known digest, without revealing the preimage. Such a circuit is the fundamental building block to privately open and verify Merkle Tree commitments, one of the main tasks required in many privacy-

Table 5: `libsnark` performance of a preimage verification circuit for the target compression functions (same as in Table 3), over the BN254 elliptic curve. Results obtained by averaging 10 batches with 5 executions per batch. Standard deviation is always within 2%.

Comp. ratio	# R1CS constraints PAX	Sponge	Proof Generation time PAX	Sponge
POSEIDON-π				
2:1	**221**	246	44.5 ms	47.3 ms
4:1	**268**	293	51.3 ms	54.9 ms
8:1	**368**	393	66.3 ms	71.5 ms
Rescue				
2:1	**240**	252	41.2 ms	42.3 ms
4:1	**264**	270	44.9 ms	45.1 ms
8:1	**384**	432	63.8 ms	67.7 ms

preserving applications [42, 59, 62].

Groth16 The Groth16 implementation provided by `libsnark` relies on the *Rank-1 Constraint Satisfaction* (R1CS) arithmetization, that is, a system of equations of the type $\boldsymbol{ax} \odot \boldsymbol{bx} = \boldsymbol{cx}$, where $\boldsymbol{x} \in \mathbb{F}_p[x_1, \ldots, x_n]$, $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{c} \in \mathbb{F}_p^n$, and $\odot$ denotes the Hadamard (element-wise) product of two vectors. The number of R1CS constraints is usually regarded as a good estimate for the prover time complexity, although the sparsity of the coefficient vectors can also have a considerable impact.

In Table 5, we report, alongside the number of R1CS constraints, the proof generation time in a `libsnark` implementation of a preimage verification circuit for the target compression functions. Considering the number of constraints, we can see that the PAX mode generally achieves a slight edge over Sponge across the board, usually being about 5–10% faster.

Plonky2 The *Plonky2* framework uses a variant of the $\mathcal{P}\mathrm{lon}\mathcal{K}$ arithmetization [33], and doing a pen-and-paper analysis of the cost is much harder than for R1CS arithmetization. A common approach in the literature is to count the number of 2-fan-in 1-fan-out gates (such as binary addition and multiplication), as well as lookup gates (when used), and even custom gates. However, different gates have different costs, and the library implementation seems to optimize the circuit descriptions, as we observed that the number of constraints specified when describing the circuit did not always match the number of constraints reported when querying the library's API, which are reported in Table 6. In fact, from the results it becomes quite evident that the relation between the number of gates and the actual efficiency is not as clear. For example, in the 8:1 compression setting, the Sponge-based compression function of POSEIDON-π requires about 40% more gates than its PAX-based counterpart, yet their proof generation times are almost the same. On the other hand, particularly at smaller state sizes, the speed-up becomes much more significant. For the 2:1 setting, in particular, proof generation in the PAX mode is about twice as fast.

Table 6: *Plonky2* performance of a preimage knowledge circuit for the target compression functions, over the Goldilocks prime field.

Comp. ratio	# gates PAX	Sponge	Proof Generation time PAX	Sponge
POSEIDON-π				
2:1	**122**	259	**11.3 ms**	16.5 ms
4:1	**439**	668	**26.0 ms**	27.1 ms
8:1	**2065**	2864	**90.8 ms**	92.9 ms
Rescue				
2:1	**91**	175	**10.9 ms**	17.2 ms
4:1	**284**	418	**16.8 ms**	27.1 ms
8:1	**976**	1213	**47.9 ms**	49.0 ms

Acknowledgments

We thank the anonymous reviewers for their comments, which have been helpful to improve the clarity of the paper. Rishiraj Bhattacharyya acknowledges the support of UKRI by EPSRC, grant number EP/Y001680/1. Stefano Trevisani and Elena Andreeva were funded in whole or in part by the Austrian Science Fund (FWF), grant number 10.55776/F85. Elena Andreeva and Arnab Roy were funded in part by the Vienna Science and Technology Fund (WWTF) CrossPings, grant number 10.47379/ICT25075.

Ethical Considerations

Our research did not require conducting studies or experiments involving sensitive or private information of individuals or organizations.

The development and deployment of cryptographic primitives, particularly those based on novel designs, come with inherent ethical responsibilities. Cryptographic systems play a critical role in ensuring privacy, security, and trust in digital systems, and any weaknesses or oversights in their design or implementation can have far-reaching consequences.

By enabling privacy-enhancing technologies such as Zero-Knowledge proof systems, arithmetization-oriented primitives contribute to empowering individuals and organizations to protect their data and maintain confidentiality. However, these technologies must be used responsibly to avoid misuse, such as enabling illicit activities or undermining accountability mechanisms.

Finally, it is essential to emphasize that while provable security techniques provide a strong theoretical foundation for constructing secure cryptographic systems, their real-world deployment must be approached with caution. This includes rigorous implementation practices, comprehensive testing, and ongoing scrutiny to identify and mitigate potential vulnerabilities. The reliance on relatively new designs, such as arithmetization-oriented constructions, necessitates an even higher level of diligence, as these designs may not yet have undergone extensive cryptanalysis. In this light, we stress that the reference implementation of the proposed modes, that is provided as part of the Open Science compliance effort, should only be employed for the purpose of experimental validation, and not for deployment in production-grade systems.

Open Science

To promote transparency and reproducibility, while also ensuring anonymity during the review process, we provided the reviewers with a reference implementation of the Rust plain implementation and Plonky2 circuits used in this work's experiments. We have then deanonymized and made permanently accessible the artifacts on Zenodo (see [71]).

References

[1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *Advances in Cryptology – ASIACRYPT 2016*, pages 191–219, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[2] Aligned. Aligned.co projects $10 billion market for web3 zero-knowledge proof generation by the year 2030. https://web.archive.org/web/20250807160524/https://www.aligned.co/post/10-billion-revenue-market-size-by-2030, 2023. Accessed: 2025-05-14.

[3] Abdelrahaman Aly, Tomer Ashur, Eli Ben-Sasson, Siemen Dhooghe, and Alan Szepieniec. Design of symmetric-key primitives for advanced cryptographic protocols. *IACR Transactions on Symmetric Cryptology*, 2020(3):1–45, Sep. 2020.

[4] Elena Andreeva, Rishiraj Bhattacharyya, and Arnab Roy. Compactness of hashing modes and efficiency beyond merkle tree. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology – EUROCRYPT 2021*, pages 92–123, Cham, 2021. Springer International Publishing.

[5] Elena Andreeva, Rishiraj Bhattacharyya, Arnab Roy, and Stefano Trevisani. On Efficient and Secure Compression Functions for Arithmetization-Oriented Hashing. In *2024 IEEE 37th Computer Security Foundations Symposium (CSF)*, pages 1–16, Los Alamitos, CA, USA, Jul 2024. IEEE Computer Society.

[6] Gal Arnon, Alessandro Chiesa, Giacomo Fenzi, and Eylon Yogev. Whir: Reed–solomon proximity testing with super-fast verification. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology – EUROCRYPT 2025*, pages 214–243, Cham, 2025. Springer Nature Switzerland.

[7] Tomer Ashur and Amit Singh Bhati. Generalized indifferentiable sponge and its application to polygon miden vm. *Designs, Codes and Cryptography*, 94(1):9, Dec 2025.

[8] Abdelhamid Bakhta, Eli Ben-Sasson, Avihu Levy, and David Levit Gurevich. Eip-5988: Add poseidon hash function precompile. Ethereum Improvement Proposal, November 2022. Draft, accessed 2025-05-15.

[9] Mihir Bellare and Phillip Rogaway. Random oracles are practical: a paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

[10] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast Reed-Solomon Interactive Oracle Proofs of Proximity. In Ioannis Chatzigiannakis, Christos Kaklamanis, Dániel Marx, and Donald Sannella, editors, *45th International Colloquium on Automata, Languages, and Programming (ICALP 2018)*, volume 107 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 14:1–14:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl – Leibniz-Zentrum für Informatik.

[11] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable, transparent, and post-quantum secure computational integrity. Cryptology ePrint Archive, Paper 2018/046, 2018.

[12] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Shaul Kfir, Eran Tromer, Madars Virza, Howard Wu, Michael Backes, Manuel Barbosa, Alexander Chernyakhovsky, Dario Fiore, Jens Groth, Joshua A. Kroll, Shigeo MITSUNARI, Aleksejs Popovs, Raphael Reischuk, and Tadanori TERUYA. libsnark: a c++ library for zksnark proofs, 2012. SCIPR Lab.

[13] Eli Ben-Sasson, Alessandro Chiesa, Michael Riabzev, Nicholas Spooner, Madars Virza, and Nicholas P. Ward. Aurora: Transparent succinct arguments for r1cs. Cryptology ePrint Archive, Paper 2018/828, 2018.

[14] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Proceedings, Part II, of the 14th International Conference on Theory of Cryptography - Volume 9986*, pages 31–60, Berlin, Heidelberg, 2016. Springer-Verlag.

[15] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007, 2007.

[16] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the indifferentiability of the sponge construction. In Nigel Smart, editor, *Advances in Cryptology – EUROCRYPT 2008*, pages 181–197, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[17] J. Black, P. Rogaway, T. Shrimpton, and M. Stam. An analysis of the blockcipher-based hash functions from pgv. *Journal of Cryptology*, 23(4):519–545, Oct 2010.

[18] John Black, Phillip Rogaway, and Thomas Shrimpton. Black-box analysis of the block-cipher-based hash-function constructions from pgv. In Moti Yung, editor, *Advances in Cryptology — CRYPTO 2002*, pages 320–335, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[19] Jonathan Bootle, Andrea Cerulli, Pyrros Chaidos, Jens Groth, and Christophe Petit. Efficient zero-knowledge arguments for arithmetic circuits in the discrete log setting. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 327–357, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[20] Niek J. Bouman. Multiprecision arithmetic for cryptology in c++ - compile-time computations and beating the performance of hand-optimized assembly at run-time. arXiv:1804.07236, 2018. https://arxiv.org/abs/1804.07236.

[21] Clémence Bouvier, Pierre Briaud, Pyrros Chaidos, Léo Perrin, Robin Salen, Vesselin Velichkov, and Danny Willems. New design techniques for efficient arithmetization-oriented hash functions: `Anemoi` permutations and `Jive` compression mode. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 507–539, Cham, 2023. Springer Nature Switzerland.

[22] Jeremy Bruestle, Paul Gafni, and The RISC Zero Team. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity, August 2023.

[23] Binyi Chen, Benedikt Bünz, Dan Boneh, and Zhenfei Zhang. Hyperplonk: Plonk with linear-time prover and high-degree custom gates. In *Advances in Cryptology - EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23–27, 2023, Proceedings, Part II*, pages 499–530, Berlin, Heidelberg, 2023. Springer-Verlag.

[24] Alessandro Chiesa, Dev Ojha, and Nicholas Spooner. Fractal: Post-quantum and transparent recursive proofs from holography. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020*, pages 769–793, Cham, 2020. Springer International Publishing.

[25] Jean-Sébastien Coron, Yevgeniy Dodis, Cécile Malinaud, and Prashant Puniya. Merkle-damgård revisited: How to construct a hash function. In Victor Shoup, editor, *Advances in Cryptology – CRYPTO 2005*, pages 430–448, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[26] Ivan Bjerre Damgård. A design principle for hash functions. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 416–427, New York, NY, 1990. Springer New York.

[27] Quynh H. Dang. *Secure Hash Standard*. National Institute of Standards and Technology, Jul 2015.

[28] Orr Dunkelman and Nathan Keller. The effects of the omission of last round's mixcolumns on aes. *Information Processing Letters*, 110(8):304–308, 2010.

[29] Morris Dworkin. Sha-3 standard: Permutation-based hash and extendable-output functions, 2015-08-04 2015.

[30] Ben Edgington. Bls12-381 for the rest of us. https://hackmd.io/@benjaminion/bls12-381#Resources-and-further-reading, Jun 2023.

[31] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks. Cryptology ePrint Archive, Paper 2023/1503, 2023.

[32] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Proceedings on Advances in Cryptology—CRYPTO '86*, pages 186–194, Berlin, Heidelberg, 1987. Springer-Verlag.

[33] Ariel Gabizon, Zachary J. Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. Cryptology ePrint Archive, Paper 2019/953, 2019.

[34] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo - a turing-complete STARK-friendly CPU architecture. Cryptology ePrint Archive, Paper 2021/1063, 2025.

[35] Lorenzo Grassi, Yonglin Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. Horst meets fluid-spn: Griffin for zero-knowledge applications. Cryptology ePrint Archive, Paper 2022/403, 2022.

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

[37] Lorenzo Grassi, Dmitry Khovratovich, and Markus Schofnegger. Poseidon2: A faster version of the poseidon hash function. In Nadia El Mrabet, Luca De Feo, and Sylvain Duquesne, editors, *Progress in Cryptology - AFRICACRYPT 2023*, pages 177–203, Cham, 2023. Springer Nature Switzerland.

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

[39] Jens Groth. Short non-interactive zero-knowledge proofs. In *Advances in Cryptology - ASIACRYPT 2010 - 16th International Conference on the Theory and Application of Cryptology and Information Security*, volume 6477 of *Lecture Notes in Computer Science*, pages 341–358. Springer, 2010.

[40] Jens Groth. On the size of pairing-based non-interactive arguments. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology – EUROCRYPT 2016*, pages 305–326, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[41] Brook Heisler and Jorge Aparicio. Criterion.rs: Statistics-driven benchmarking library for rust. GitHub repository, 2025. Accessed: 2025-05-15.

[42] Daira Hopwood, Sean Bowe, Taylor Hornby, and Nathan Wilcox. Zcash protocol specification. ZCash Improvement Proposals Website, Sep 2022.

[43] Dmitry Khovratovich, Mario Marhuenda Beltrán, and Bart Mennink. Generic security of the safe api and its applications. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023*, pages 301–327, Singapore, 2023. Springer Nature Singapore.

[44] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Theory of Computing*, STOC '92, pages 723–732, New York, NY, USA, 1992. Association for Computing Machinery.

[45] Stefan Kölbl, Martin M. Lauridsen, Florian Mendel, and Christian Rechberger. Haraka v2 — efficient short-input hashing for post-quantum applications. *IACR Transactions on Symmetric Cryptology*, 2016(2):1–29, Feb. 2017.

[46] Charlotte Lefevre. Indifferentiability of the sponge construction with a restricted number of message blocks. *IACR Transactions on Symmetric Cryptology*, 2023(1):224–243, Mar. 2023.

[47] Charlotte Lefevre, Mario Marhuenda Beltrán, and Bart Mennink. To pad or not to pad? padding-free arithmetization-oriented sponges. *IACR Transactions on Symmetric Cryptology*, 2025(1):97–137, Mar. 2025.

[48] Charlotte Lefevre and Bart Mennink. Tight preimage resistance of the sponge construction. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022*, pages 185–204, Cham, 2022. Springer Nature Switzerland.

[49] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *Theory of Cryptography*, pages 21–39, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[50] Bart Mennink and Bart Preneel. Efficient parallelizable hashing using small non-compressing primitives. *International Journal of Information Security*, 15(3):285–300, 06 2016. Copyright - Springer-Verlag Berlin Heidelberg 2016; Caratteristica del documento - Tables; ; Ultimo aggiornamento - 2023-07-17.

[51] Ralph C. Merkle. A digital signature based on a conventional encryption function. In Carl Pomerance, editor, *Advances in Cryptology — CRYPTO '87*, pages 369–378, Berlin, Heidelberg, 1988. Springer Berlin Heidelberg.

[52] Ralph C. Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3(1):43–58, Jan 1990.

[53] Ralph C. Merkle. One way hash functions and des. In Gilles Brassard, editor, *Advances in Cryptology — CRYPTO' 89 Proceedings*, pages 428–446, New York, NY, 1990. Springer New York.

[54] Meta Open Source. Winterfell stark prover: Benchmarks on chains of rescue hashes. GitHub repository README, 2025. Commit accessed 2025-05-15.

[55] Yusuke Naito and Kazuo Ohta. Improved indifferentiable security analysis of photon. In Michel Abdalla and Roberto De Prisco, editors, *Security and Cryptography for Networks*, pages 340–357, Cham, 2014. Springer International Publishing.

[56] Mridul Nandi. Characterizing padding rules of md hash functions preserving collision security. In Colin Boyd and Juan González Nieto, editors, *Information Security and Privacy*, pages 171–184, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[57] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Paper 2013/279, 2013.

[58] R. Penrose. A generalized inverse for matrices. *Mathematical Proceedings of the Cambridge Philosophical Society*, 51(3):406–413, 1955.

[59] Polygon Team. The polygon knowledge layer, 2025. Accessed 2025-05-15.

[60] Bart Preneel, René Govaerts, and Joos Vandewalle. Hash functions based on block ciphers: A synthetic approach. In *Advances in Cryptology - CRYPTO '93, 13th Annual International Cryptology Conference, Santa Barbara, California, USA, August 22-26, 1993, Proceedings*, volume 773 of *Lecture Notes in Computer Science*, pages 368–378. Springer, 1993.

[61] Protocol Labs. Poseidon hash function — filecoin specification, 2025. Accessed 2025-05-15.

[62] Yiannis Psaras and David Dias. The interplanetary file system and the filecoin network. In *2020 50th Annual IEEE-IFIP International Conference on Dependable Systems and Networks-Supplemental Volume (DSN-S)*, pages 80–80, 2020.

[63] Ronald L. Rivest. The md5 message-digest algorithm. In *RFC*, 1990.

[64] Phillip Rogaway and Thomas Shrimpton. Cryptographic hash-function basics: Definitions, implications, and separations for preimage resistance, second-preimage resistance, and collision resistance. In Bimal Roy and Willi Meier, editors, *Fast Software Encryption*, pages 371–388, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[65] Phillip Rogaway and John P. Steinberger. Constructing cryptographic hash functions from fixed-key blockciphers. In *Advances in Cryptology - CRYPTO 2008, 28th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2008*, volume 5157 of *Lecture Notes in Computer Science*, pages 433–450. Springer, 2008.

[66] Markku-Juhani O. Saarinen. Cryptanalysis of block ciphers based on sha-1 and md5. In Thomas Johansson, editor, *Fast Software Encryption*, pages 36–44, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[67] C. E. Shannon. Communication theory of secrecy systems. *The Bell System Technical Journal*, 28(4):656–715, 1949.

[68] Martijn Stam. Blockcipher-based hashing revisited. In Orr Dunkelman, editor, *Fast Software Encryption*, pages 67–83, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[69] Alan Szepieniec, Tomer Ashur, and Siemen Dhooghe. Rescue-prime: a standard specification (sok). Cryptology ePrint Archive, Paper 2020/1143, 2020.

[70] Polygon Zero Team. Plonky2: Fast recursive arguments with plonk and fri, September 2022.

[71] Stefano Trevisani. Boosting Efficiency and Security of AO Hashing for ZKP Systems — USENIX AE , May 2026. https://doi.org/10.5281/zenodo.20307357.

[72] Paul Valiant. Incrementally verifiable computation or proofs of knowledge imply time/space efficiency. In Ran Canetti, editor, *Theory of Cryptography*, pages 1–18, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[73] Jonathan Wang. Bn254 for the rest of us. https://hackmd.io/@jpw/bn254, Aug 2022.

[74] Robert S. Winternitz. Producing a one-way hash function from des. In David Chaum, editor, *Advances in Cryptology: Proceedings of Crypto 83*, pages 203–207, Boston, MA, 1984. Springer US.

[75] Robert S. Winternitz. A secure one-way hash function built from des. In *1984 IEEE Symposium on Security and Privacy*, pages 88–88, 1984.

A Additional Benchmarks

In an effort to validate the stability of our benchmarks, we implemented the target compression functions in several independent libraries providing finite-field arithmetic facilities, namely NTL[6], `libff`[7], and `ctbignum` [20]. Our results are reported in Table 7. While we can observe a variation in the efficiency of the single instances, the improvement due to the choice of mode is more or less the same.

[6] https://libntl.org
[7] https://github.com/scipr-lab/libff

Table 7: Plain performance of the target compression functions with different backends. The results were obtained as for Table 3. We were not able to get the results for the Goldilocks field in `libff`.

	`NTL`				`libff`		`ctbignum`			
	BN254		Goldilocks		BN254		BN254		Goldilocks	
Comp. ratio	PA	Sponge	PA	Sponge	PA	Sponge	PA	Sponge	PA	Sponge
POSEIDON-π										
2:1	**56.3 µs**	99.7 µs	**153.7 µs**	382.6 µs	**42.1 µs**	61.6 µs	**50.5 µs**	96.9 µs	**9.81 µs**	23.5 µs
4:1	**154.4 µs**	226.5 µs	**583.1 µs**	897.1 µs	**86.7 µs**	121.2 µs	**159.0 µs**	239.1 µs	**39.9 µs**	59.8 µs
8:1	**538.5 µs**	670.7 µs	**2.80 ms**	3.90 ms	**260.8 µs**	321.0 µs	**588.1 µs**	746.5 µs	**192.6 µs**	266.4 µs
Rescue										
2:1	**410.4 µs**	440.4 µs	**204.2 µs**	358.7 µs	**714.9 µs**	754.0 µs	**425.0 µs**	457.2 µs	**28.0 µs**	50.5 µs
4:1	**471.8 µs**	493.1 µs	**553.1 µs**	777.1 µs	**813.4 µs**	842.8 µs	**492.7 µs**	516.6 µs	**81.8 µs**	117.1 µs
8:1	**745.6 µs**	857.2 µs	**1.68 ms**	2.04 ms	**1.19 ms**	1.37 ms	**793.3 µs**	913.9 µs	**236.1 µs**	283.0 µs