Efficient Compact-Tree Commitment Verification in ZK-SNARK Settings

Stefano Trevisani*

August 3, 2022

Abstract

Practical Zero Knowledge systems for verifiable computation have been an hot research topic of the last decade, particularly for their ties with blockchains and cryptocurrencies. In this report, we study the ZK-SNARK system Pinocchio, and we implement the MiMC cryptographic primitives, together with the ABR mode of hash, comparing them to the traditional SHA over Merkle Tree which is tipically used in multiple-message commitment protocols.

*I would like to thank Massimo Del Prato for collaborating with the project, Arnab Roy for supervisioning it, and Elisabeth Oswald for supporting the research.

Contents

1 Introduction

One of the biggest revolutions of the last decade has been the widespread adoption of blockchain-based technologies [11]. For example, cryptocurrencies like Bitcoin or Ethereum are widely known even among the non-crypto-enthusiasts and a huge market is growing around them. There are other highly interesting applications for the blockchain outside the cryptocurrency world: in fact, anything which requires some degree of 'verifiability' in a non-trusted environment can benefit from the usage of a blockchain technology. Typically, a block of the chain does not correspond to a single transaction, as the blockchain would grow too quickly for practical storage: many transactions are inserted into a tree data-structure, called Merkle Tree (MT) [9], which is computed bottom-up and whose root is then inserted into the blockchain. In a traditional setup (e.g. cryptocurrencies), all the data which is required to build a block of the blockchain is of public knowledge: when a transaction happens, the details of that transaction are shared with the network for it to be validated. There are however many scenarios where one would like for the data to remain secret, as there could be risks involving privacy violation (e.g. transactions) or intellectual property theft (e.g. shared computation).

Zero-Knowledge Succinct Non-interactive ARgument of Knowledge (ZK-SNARK) systems are cryptographic frameworks which allow for a party to convince other parties about the knowledge of some secret without revealing the secret. As a simple example, one would be able convince other parties that a transaction has been performed honestly, but without revealing the details of

the transaction. ZK-SNARK associate the claim made by the challenger with an instance of a problem which is almost impossible to solve if the secrets involved in the claim are not actually true. Since ZK-SNARK over prime fields and groups, and traditional hashing algorithms, like SHA-256 [5], are not efficient when adapted for these frameworks. For this reason, new hashing algorithms like MiMCHash [1] have been designed with the ZK scenario in mind, and the Augmented Binary tRee (ABR) [2] data structure tries to improve on the compactness of Merkle Trees by processing more transactions with the same amount of calls to the underlying hash function.

Organization of this report

In Section 2, we introduce some basic notions of computational and complexity theory, group and field theory, hash functions and tree hash modes; we also introduce arithmetic circuits, Rank-1 Constraint Systems and Quadratic Arithmetic Programs, which are an essential tool for ZK-SNARK systems. In Section 3, we introduce ZK-SNARK, and study the Pinocchio protocol, as well as the ZK-SNARK-friendly MiMC permutation. In Section 4, we compare our implementations of MiMC and ABRs with SHA and Merkle Trees using the `libsnark` library. Finally, in Section 5 we draw our conclusions and discuss possible future directions.

2 Preliminaries

In this section we are going to introduce some fundamental concepts; while some are relatively basic and wide-known, it can still be useful to skim over them to be sure of having a firm grasp on the main ideas behind ZK-SNARK systems.

2.1 Computational models and complexity

A *computational model* (or model of computation) is any kind of 'device', either physical or mathematical, which is able to compute algorithms to solve problems [15]. A particularly interesting class of problems are *decision problems*, the ones that can be answered with 'yes' (or 'accept', or $\top$) or 'no' (or 'reject', or $\bot$). Every computational model is able to *decide* only a subclass of all decision problems, and even then, not all can be solved *efficiently*, that is, by using an amount of resources (typically, time and space) which is upper-bounded by some polynomial function of the input length. Problems for which a polynomial bound doesen't exist or isn't known are said to be *hard* for the computational model. For example, finding solutions to boolean equations (the SAT problem) and, by extension, to arithmetic equations, is believed to be hard for deterministic Turing machines, but it is easy for non-deterministic ones [4]. With the advent of quantum-computing, problems which are believed to be hard for classical computers, like prime factorization, have been shown to be efficiently solved by quantum computers [16]. While still far from usable in practical cases,

this shows that one must be extremely careful when talking about the hardness of problems, especially when applied to cryptography, and must always make clear assumptions on the underlying model of computation.

2.2 Fields and groups

While Turing machines typically operate over binary strings, that is, elements of $\{0,1\}^*$, where $*$ indicates Kleene's closure [6], we often want to interpret such strings as elements of some algebraic structure.

Definition 1. A *field* is any triple $\mathbb{F} = (F, \oplus, \otimes)$, where F is called *underlying set*, $\oplus\colon F \times F \mapsto F$ is called *field addition* and $\otimes\colon F \times F \mapsto F$ is called *field multiplication*, such that both addition and multiplication are commutative and associative, multiplication distributes over addition, F contains an additive identity element $0_{\mathbb{F}}$ and a multiplicative identity element $1_{\mathbb{F}}$, $\forall x \in F$ there is an additive inverse element $-x$, and $\forall x \in F \setminus \{0_{\mathbb{F}}\}$ there is a multiplicative inverse element x^{-1}.

We denote elements of a field $\mathbb{F}$ (abusing notation, as they are actually elements of the underlying set F) with lowercase letters $a, b, c \ldots$ and variables over $\mathbb{F}$ with lowercase letters $x, y, z \ldots$. We are particularly interested in *finite fields* of the kind:

$$\mathbb{F}_{p^k} = \left(\{0, \ldots, p^k - 1\}, \oplus_p, \otimes_p\right)$$

where p is a prime number, $z \in \mathbb{N}$ and $\oplus_p, \otimes_p$ denote respectively addition and multiplication modulo p (for example, the Boolean field $\mathbb{B} = \mathbb{F}_2$). Typically, we consider n-bit strings either as elements of $\mathbb{F}_{2^n}$ or of $\mathbb{F}_{p^1}$, where $\log_2(p) \approx n$. We will often use $+$ in place of $\oplus_p$ when denoting addition and omit $\otimes_p$ when denoting multiplication, if $\mathbb{F}$ is clear from the context.

Any field $\mathbb{F}$ can be extended to an n-dimensional vector space $\mathbb{F}^n$, for some $n \in \mathbb{N}$. We denote vectors in $\mathbb{F}^n$ with lowercase bold letters ($\boldsymbol{v}, \boldsymbol{w}, \ldots$), and the ith element of a vector $\boldsymbol{v}$ with $\boldsymbol{v}_i$. Vector operations follow their natural definitions depending on the underlying field. We can also introduce matrices over $\mathbb{F}^{n \times m}$ for some $n, m \in \mathbb{N}$, which we denote with bold capital letters ($\boldsymbol{A}, \boldsymbol{B}, \ldots$). The ith row of a matrix $\boldsymbol{M}$ is denoted with $\boldsymbol{M}_i$, and the jth element of the ith row is denoted with $\boldsymbol{M}_{i,j}$, and $\boldsymbol{M}^\intercal \mid \forall i, j\colon \boldsymbol{M}^\intercal_{i,j} = \boldsymbol{M}_{j,i}$ is the transpose of $\boldsymbol{M}$. Matrix operations also follow their natural definitions over the underlying field. Given $\boldsymbol{A} \in \mathbb{F}^{n \times m}, \boldsymbol{B} \in \mathbb{F}^{n \times m'}$, we denote with $\begin{pmatrix}\boldsymbol{A} & \boldsymbol{B}\end{pmatrix}$ their concatenation along the rows, and with $\left(\boldsymbol{A}; \boldsymbol{B}\right) = \begin{pmatrix}\boldsymbol{A}^\intercal & \boldsymbol{B}^\intercal\end{pmatrix}^\intercal$ their concatenation along the columns.

A field $\mathbb{F}$ can also be extended to a monovariate polynomial ring $\mathbb{F}[x]$, we will denote polynomials with lowercase letters $(p, q, \ldots)$. Operations over polynomials are naturally derived from the underlying field. Vectors and matrices of polynomials are denoted with the usual notation ($\boldsymbol{p}, \boldsymbol{q}, \ldots$ and $\boldsymbol{P}, \boldsymbol{Q}, \ldots$).

Given some $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$, we can build the unique polynomial:

$$p \mid p \in \mathbb{F}[x] \wedge \deg(p) = n - 1 \wedge \forall i\colon p(\boldsymbol{x}_i) = \boldsymbol{y}_i$$

by using Lagrange interpolation:

$$p = L(\boldsymbol{x}, \boldsymbol{y}) = \sum_i \boldsymbol{y}_i \prod_{j \neq i} \frac{x - \boldsymbol{x}_j}{\boldsymbol{x}_i - \boldsymbol{x}_j}$$

We can extend Lagrange interpolation to any pair of matrices $\boldsymbol{X}, \boldsymbol{Y} \in \mathbb{F}^{n \times m}$ by applying L to every row:

$$L(\boldsymbol{X}, \boldsymbol{Y}) = (L(\boldsymbol{X}_1, \boldsymbol{Y}_1) \ldots, L(\boldsymbol{X}_n, \boldsymbol{Y}_n))$$

Definition 2. A *group* is a pair $\mathbb{G} = (G, \odot)$, where G is called *underlying set*, and $\odot \colon G \times G \mapsto G$ is called *group composition*, such that composition is associative, there is a compositive identity element $1_{\mathbb{G}}$, and $\forall x \in \mathbb{G}$ there is a compositive inverse x^{-1}.

We denote objects over groups in the same way we do for fields. We are particularly interested in *cyclic groups*, i.e. finite groups of the type $\mathbb{G}_q(g) = \langle g \rangle = \left(\left\{g^i \bmod q\right\}_{i \in \mathbb{N}}, \otimes_q\right)$, where $g \in \mathbb{F}_p$ is called *generator element* and $\mathbb{F}_p$ is called *underlying field*. Since every element of $\mathbb{G}_q(g)$ is obtained by exponentiating g, we can define a bijective *discrete logarithm* function:

$$\log_g \colon \mathbb{G}_q(g) \mapsto \mathbb{F}_p = \{(x, y) \mid x = g^y\}$$

Cyclic groups are very interesting because, while it's easy to compute exponentiation, no deterministic algorithm is known that can efficiently compute the discrete logarithm.

Definition 3. Given cyclic groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_{\mathsf{T}}$, a *bilinear map* is any function:

$$B \colon \mathbb{G}_1 \times \mathbb{G}_2 \mapsto \mathbb{G}_{\mathsf{T}} \mid \forall x \in \mathbb{G}_1, \forall y \in \mathbb{G}_2, \forall a, b \in \mathbb{Z} \colon B\left(x^a, y^b\right) = B(x, y)^{ab}$$

If $|\mathbb{G}| = |\mathbb{G}'|$, then B is *order-preserving*, and if $\mathbb{G}_1 = \langle g_1 \rangle \wedge \mathbb{G}_2 = \langle g_2 \rangle \wedge \mathbb{G}_{\mathsf{T}} = \langle B(g_1, g_2) \rangle$, then B is *non-trivial*.

We are interested in order-preserving, non-trivial bilinear maps where $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$. Bilinear maps have many application in cryptography, as they are used to exploit the hardness of the discrete logarithm problem.

2.3 Arithmetic Circuits

A sequence of operations over field elements and variables can be neatly represented by *arithmetic circuits*.

Definition 4 (Arithmetic circuit)**.** Given a field $\mathbb{F}$, some $n, m \in \mathbb{N}$, some constants $a_{1,1}, \ldots, a_{m,n} \in \mathbb{F}$, and some variables $x_1, \ldots, x_n$ over $\mathbb{F}$, an *implicit*

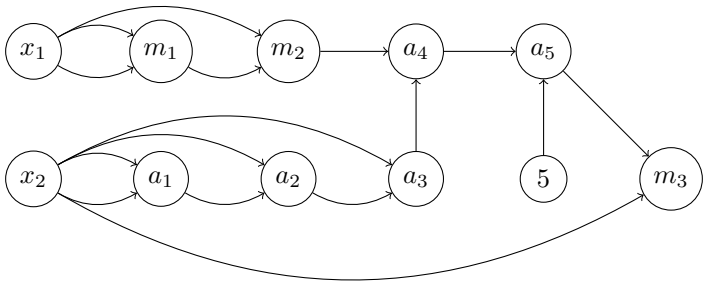

Figure 1: DAG of the circuit in Example 1. We have the two variable/input vertices x_1 and x_2, the constant vertex 5, the addition vertices $a_1, \ldots, a_5$ and the multiplication vertices m_1, m_2 and m_3, which is also an output vertex.

arithmetic circuit over $\mathbb{F}$ is any formula:

$\phi \equiv c$	with $c \in \mathbb{F}$
$\phi \equiv x$	with x variable over $\mathbb{F}$
$\phi \equiv \phi_1^c$	with $c \in \mathbb{F}$ and $\phi_1 \neq \phi$ arithmetic circuit
$\phi \equiv \phi_1 \oplus \phi_2$	with $\phi_1 \neq \phi, \phi_2 \neq \phi$ arithmetic circuits
$\phi \equiv \phi_1 \otimes \phi_2$	with $\phi_1 \neq \phi, \phi_2 \neq \phi$ arithmetic circuits
$\phi \equiv \phi_1, \phi_2$	with $\phi_1 \neq \phi, \phi_2 \neq \phi$ arithmetic circuits

An arithmetic circuit which does not contain multiplications and exponentiations by constants is called *explicit arithmetic circuit*.

Every arithmetic circuit can be represented by a Directed Acyclic Graph (DAG), where the vertices are labeled either with a variable name (*variable vertices*), a constant from the field (*constant vertices*) or one of the operations $\oplus$ (*addition vertices*, denoted $\phi_\oplus$) and $\otimes$ (*multiplication vertices*, denoted $\phi_\otimes$). With an analogy to digital circuits, vertices are also called *gates*. Only addition and multiplication vertices have incoming edges (exactly two), which represent the inputs of the operation, while the outgoing edge will represent the result. Vertices without incoming edges are called *input vertices*, while vertices without outgoing edges are called *output vertices*, together they are denoted ϕ_{IO}.

It is possible, without affecting the expressive power, to transform an implicit arithmetic circuit into an explicit one by replacing exponentiations (multiplications) by some constant c with a a sequence of c multiplications (additions)[1].

Example 1. *Let's consider the following implicit arithmetic circuit over* $\mathbb{F}_{13}$*:*

$$\phi = x_2\big(x_1^3 + 4x_2 + 5\big)$$

[1]However, this transformation can affect the succinctness of a circuit and its DAG (unrolling x^c or cx requires $\Theta(2^c)$ space), but this won't be a problem for us.

We can unroll it into an equivalent (explicit) arithmetic circuit:

$$\widehat{\phi} = x_2(x_1x_1x_1 + x_2 + x_2 + x_2 + x_2 + 5)$$

And draw the associated DAG, which is shown in Figure 1.

2.4 Rank-1 Constraint systems

Like it happens for boolean formulae and the famous SAT problem, arithmetic circuits can also be seen as a form of constraint whose solution is a set of valid assignments for all the intermediate values in the computation.

Definition 5 (Rank-1 Contraint System)**.** Given a field $\mathbb{F}$ and some $m, n \in \mathbb{N}$, a n/m *Rank-1 Constraint System (R1CS)* over $\mathbb{F}$ is any triple:

$$\mathcal{C} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}) \mid \boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{n \times m}$$

A *solution* to some R1CS $\mathcal{C}$ is any (column) vector:

$$\boldsymbol{s} \mid \boldsymbol{s} \in \mathbb{F}^m \wedge (\boldsymbol{A}\boldsymbol{s})(\boldsymbol{B}\boldsymbol{s}) = \boldsymbol{C}\boldsymbol{s}$$

Any explicit arithmetic circuit with n multiplicative gates and m variables $x_1, \ldots, x_n$ can be associated with an $n/(n+m+1)$ R1CS $\mathcal{C}$ which represents the constraints in the circuit, roughly in the following way:

1. Add a new 'constant variable' which always assumes value 1.
2. For every multiplicative gate $\otimes_i$ in the circuit, add a new *intermediate* variable t_i (t_n can be denoted y as it represents the circuit output).
3. Define the column vector $\boldsymbol{x} = \begin{pmatrix} 1 & x_1 & \cdots & x_m & t_1 & \cdots & t_n \end{pmatrix}^\intercal$.
4. Express every multiplication gate $\otimes_i$ as an equation in the canonical form:

$$(\boldsymbol{a_i}\boldsymbol{x})(\boldsymbol{b_i}\boldsymbol{x}) = \boldsymbol{c_i}\boldsymbol{x}$$

   where $\boldsymbol{a_i}$ $\boldsymbol{b_i}$ $\boldsymbol{c_i}$ will be the ith rows of $\mathcal{C}_{\boldsymbol{A}}$, $\mathcal{C}_{\boldsymbol{B}}$ and $\mathcal{C}_{\boldsymbol{C}}$ respectively.

Let's make an example to better understand the process.

Example 2. *Consider the explicit arithmetic circuit over $\mathbb{F}_{13}$ of Example 1:*

$$\widehat{\phi} = x_2(x_1x_1x_1 + x_2 + x_2 + x_2 + x_2 + 5)$$

We can see that there are a total of 3 multiplications in the circuit, and since we have two input variables, our associated R1CS will be a $3/6$ R1CS ($2+1+3=6$). Let's explicit all of the intermediate variables:

$$t_1 = x_1x_1 \qquad t_2 = t_1x_1 + 4x_2 + 5 \qquad y = t_2x_2$$

So, our variable vector will be:

$$\boldsymbol{x} = \begin{pmatrix} 1 & x_1 & x_2 & t_1 & t_2 & y \end{pmatrix}$$

Now, let's transform all the equations in canonical form:

$$\begin{aligned} &(x_1)(x_1) = t_1 \\ &(t_1)(x_1) + 4x_2 + 5 = t_2 \iff (t_1)(x_1) = 8 + 9x_2 + t_2 \\ &(x_2)(t_2) = y \end{aligned}$$

Remember that we are working over $\mathbb{F}_{13}$*, so in the second equation, when we bring* 4 *and* 8 *to the right side, we have* $-4 \equiv 9 \pmod{13}$ *and* $-5 \equiv 8 \pmod{13}$*. We can now extract our R1CS* $\mathcal{C} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$*:*

$$\mathcal{C} = \left(\begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 8 & 0 & 9 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \right)$$

By construction, a vector $\boldsymbol{s}$ *is a solution to* $\mathcal{C}$ iff *every element of* $\boldsymbol{s}$ *is assigned to the value derived by fixing* x_1, x_2 *and following the computation of the original arithmetic circuit. For example, if* $x_1 = 2, x_2 = 3$*, we have:*

$$\begin{aligned} t_1 &= x_1 x_1 = 2 \times 2 = 4 && \equiv 4 \pmod{13} \\ t_2 &= t_1 x_1 + 4x_2 + 5 = 4 \times 2 + 4 \times 3 + 5 = 25 && \equiv 12 \pmod{13} \\ y &= t_2 x_2 = 12 \times 3 = 36 && \equiv 10 \pmod{13} \end{aligned}$$

Therefore our solution vector will be:

$$\boldsymbol{s} = \begin{pmatrix} 1 & 2 & 3 & 4 & 12 & 10 \end{pmatrix}$$

It is a bit tedious, but easy, to verify that indeed $(\boldsymbol{As})(\boldsymbol{Bs}) = \boldsymbol{Cs}$*.*

2.5 Quadratic Arithmetic Programs

A problem with R1CS is that solutions have size linear in the number of multiplication gates of the corresponding arithmetic circuit. This can be solved by using Quadratic Arithmetic Programs.

Definition 6 (Quadratic Arithmetic Program)**.** Given a field $\mathbb{F}$ and some $n, m \in \mathbb{N}$, a n/m *Quadratic Arithmetic Program (QAP)* over $\mathbb{F}$ is any quadruple:

$$\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y}) \mid t \in \mathbb{F}[x] \wedge \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y} \in \mathbb{F}[x]^n$$

For which it holds that:

$$\forall i \colon \deg(\boldsymbol{v}_i) + 1 = \deg(\boldsymbol{w}_i) + 1 = \deg(\boldsymbol{y}_i) + 1 = \deg(t) = m$$

A *valid assignment* to a QAP $\mathcal{Q}$ is any vector:

$$\boldsymbol{s} \in \mathbb{F}^n \mid (\boldsymbol{vs})(\boldsymbol{ws}) - \boldsymbol{ys} \bmod t = 0$$

Then, the polynomials $p = (\boldsymbol{vs})(\boldsymbol{ws}) - \boldsymbol{ys}$ and $h = \frac{p}{t}$ are a *solution* to $\mathcal{Q}$.

Just like it was possible to represent any n/m arithmetic circuit ϕ with an $n/(n+m+1)$ R1CS $\mathcal{C}$, we can, in turn, represent any n/m R1CS $\mathcal{C} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ with a n/m QAP $\mathcal{Q}$. First, we choose some arbitrary $\boldsymbol{z} \in \mathbb{F}^n \mid \forall i, j \colon \boldsymbol{z}_i \neq \boldsymbol{z}_j$ (usually, $\boldsymbol{z} = \begin{pmatrix} 1 & \cdots & n \end{pmatrix}$). Let $\boldsymbol{Z} \in \mathbb{F}^{m \times n} \mid \forall i \colon \boldsymbol{Z}_i = \boldsymbol{z}$, then:

$$\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y}) = \left(\prod_i (x - \boldsymbol{z}_i), L\big(\boldsymbol{Z}, \boldsymbol{A}^\intercal\big), L\big(\boldsymbol{Z}, \boldsymbol{B}^\intercal\big), L\big(\boldsymbol{Z}, \boldsymbol{C}^\intercal\big) \right)$$

To make things more clear, let's make an example:

Example 3. *We want to compute the* $3/6$ *QAP* $\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ *associated with the* $3/6$ *R1CS* $\mathcal{C} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ *that we derived in Example 2. First, we set:*

$$\boldsymbol{z} = \begin{pmatrix} 1 & 2 & 3 \end{pmatrix} \qquad\qquad \boldsymbol{Z} = \begin{pmatrix} \boldsymbol{z}; \boldsymbol{z}; \boldsymbol{z}; \boldsymbol{z}; \boldsymbol{z}; \boldsymbol{z} \end{pmatrix}$$

Then, we compute the target polynomial t*, the left and right input constraint polynomial vectors* $\boldsymbol{v}$ *and* $\boldsymbol{w}$*, and the output constraint polynomial vector* $\boldsymbol{y}$ *(remember, we are working over* $\mathbb{F}_{13}$*). Notice how the 2nd, 4th and 5th columns of* $\boldsymbol{A}$ *form the canonical basis of* $\mathbb{F}_{13}^3$*, and since* L *is a linear operator, we can express all other polynomials as linear combinations of* $L(\boldsymbol{z}, \boldsymbol{A}_2^\intercal), L(\boldsymbol{z}, \boldsymbol{A}_4^\intercal)$ *and* $L(\boldsymbol{z}, \boldsymbol{A}_5^\intercal)$*:*

$$t = (x-1)(x-2)(x-3) = (x+12)(x+11)(x+10) = x^3 + 7x^2 + 11x + 7$$

$$\boldsymbol{v} = L\big(\boldsymbol{Z}, \boldsymbol{A}^\intercal\big) = \begin{pmatrix} L(\boldsymbol{z}, \boldsymbol{A}_1^\intercal) & \cdots & L(\boldsymbol{z}, \boldsymbol{A}_6^\intercal) \end{pmatrix} = \begin{pmatrix} 0 \\ 7x^2 + 4x + 3 \\ 0 \\ 12x^2 + 4x + 10 \\ 7x^2 + 5x + 1 \\ 0 \end{pmatrix}^\intercal$$

$$\boldsymbol{w} = \begin{pmatrix} 0 & \boldsymbol{v}_2 + \boldsymbol{v_4} & \boldsymbol{v}_5 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 6x^2 + 8x \\ 7x^2 + 5x + 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}^\intercal$$

$$\boldsymbol{y} = \begin{pmatrix} 8\boldsymbol{v}_4 & 0 & 9\boldsymbol{v}_4 & \boldsymbol{v}_2 & \boldsymbol{v}_4 & \boldsymbol{v}_5 \end{pmatrix} = \begin{pmatrix} 5x^2 + 6x + 2 \\ 0 \\ 4x^2 + 10x + 12 \\ 7x^2 + 4x + 3 \\ 12x^2 + 4x + 10 \\ 7x^2 + 5x + 1 \end{pmatrix}^\intercal$$

Recall that a possible solution to the R1CS was:

$$\boldsymbol{s} = \begin{pmatrix} 1 & 2 & 3 & 4 & 12 & 10 \end{pmatrix}$$

Let's check if it is also a valid assignment for the QAP:

$$p = (\boldsymbol{vs})(\boldsymbol{ws}) - \boldsymbol{ys} = \big(3x^2 + 6x + 6\big)\big(7x^2 + 5x + 3\big) - \big(12x^2 + 7x + 11\big)$$
$$= 8x^4 + 5x^3 + 4x^2 + 2x + 7$$

$$h = \frac{p}{t} = \frac{8x^4 + 5x^3 + 4x^2 + 2x + 7}{x^3 + 7x^2 + 11x + 7} = 8x - 51 + \frac{273x^2 + 507x + 364}{x^3 + 7x^2 + 11x + 7} = 8x + 1$$

Since:

$$ht = (8x + 1)\big(x^3 + 7x^2 + 11x + 7\big) = 8x^4 + 5x^3 + 4x^2 + 2x + 7 = p$$

this means that p and h are a solution to the QAP, and $\boldsymbol{s}$ is a valid assignment.

One might wonder how a solution (p, h) to a QAP is more succinct than the corresponding valid assignment $\boldsymbol{s}$ of the associated R1CS: as a matter of fact, given an n/m arithmetic circuit, $\boldsymbol{s}$ has size $n + m + 1$, while p can have degree (and therefore encoding size) $2(n - 1)$. Furthermore, in a typical circuit, $n \gg m$, so (p, h) would approximately be twice the size of $\boldsymbol{s}$ when encoded. Now, $p = ht \implies \forall x\colon p(x) = h(x)t(x)$; if we are working over a big field (say, $|\mathbb{F}| \approx 2^{256}$), it is hard to find even a single value of x for which the equation holds. This means that we can accept as a solution, with high confidence (although not certainty) any couple of values x, y such that $y \bmod t(x) = 0$.

Summing up: if $y \bmod t(x) = 0$, we are *almost sure* that y has been derived by computing $p(x)$, where p is a solution to our QAP. But if p is a solution to the QAP, then it derives from a valid assignment $\boldsymbol{s}$ to the associated R1CS, which in turn derives from a valid computation of the original arithmetic circuit.

2.6 Hash functions

Hash functions are a fundamental tool in many fields of computer science, and cryptography is arguably the most prominent. Formally, an hash function is any function $H\colon \{0,1\}^* \mapsto \{0,1\}^n$, that is, any function which maps arbitrarily long *messages* to fixed-size *digests*. From the definition, it is immediate to see that there are an infinite number of messages which map to the same digest. While an operation like truncation is a (very simple) hash function, in cryptography we are interested in functions that provide additional guarantees: the assumption is that a digest should represent a message in a one-way fashion: while there are infinite messages which map to the same digest, it must be hard to find them.

Definition 7 (Cryptographic hash function)**.** Given $n \in \mathbb{N}$, an *n(-bit) cryptographic hash function (CHF)* is any function $H\colon \{0,1\}^* \mapsto \{0,1\}^n$ which satisfies the following properties:

- **Collision resistance**: It is hard to find two messages m_1, m_2 such that $H(m_1) = H(m_2)$.
- **Preimage resistance**: Given some digest h, it is hard to find a message m such that $H(m) = h$ (H is a one-way function).

- **Second preimage resistance**: Given some message m_1, it is hard to find a message m_2 such that $H(m_1) = H(m_2)$.

With high probability, an ideal CHF requires about $2^{n/2}$ evaluations for collision resistance (birthday paradox), while for preimage resistance it would require about 2^n evaluations. A CHF can be built by applying some known secure construction to functions which are simpler to devise, specifically, we can derive a CHF from a one-way compression function (OWCF), which in turn can be derived from a pseudorandom keyed permutation (PKP) [8]. We will introduce the Davies-Meyer and the Merkle-Damgård constructions respectively, as those are the ones of interest to us.

Theorem 1 (Davies-Meyer construction [13])**.** *Given a l/n pseudo-random keyed permutation P, some $i, k \in \mathbb{N}$, some $v \in \{0,1\}^l$, and some $m \in \{0,1\}^{kn}$, then the function F_P such that:*

$$F_{P,i}(v,m) = \begin{cases} v & i = 0 \\ E\big(F_{P,i-1}(v,m), m_{(i-1)n,\dots,in}\big) & 1 \le i \le k \end{cases}$$
$$F_P = F_{P,k}$$

is a $l/kn/l$ OWCF.

Theorem 2 (Merkle-Damgård construction [10])**.** *Given a $l_1/n/l_2$ OWCF F, some $k \in \mathbb{N}$, some $v \in \{0,1\}^{l_1}$, some $m \in \{0,1\}^*$ and some padding function:*

$$P(m)\colon \{0,1\}^{|m|} \mapsto \{0,1\}^{|m|+(-|m| \bmod n)+kn}$$

such that, $\forall m, m' \in \{0,1\}^*$:

$$\Big(|m| = |m'| \Rightarrow |P(m)| = |P(m')|\Big) \wedge \Big(|m| \neq |m'| \Rightarrow m_{|P(m)|} \neq m'_{|P(m')|}\Big)$$

then the function H_F such that:

$$H_{F,i}(v,m) = \begin{cases} v & i = 0 \\ F\big(H_{F,i-1}(v,m), m_{(i-1)n,\dots,in}\big) & 1 \le i \le |P(m)| \end{cases}$$
$$H_F = H_{F,|P(m)|}$$

is a cryptographic hash function.

2.7 Tree hash modes

An important application of CHFs is in *prover-verifier games*: for any message m, the digest $h = H(m)$, where H is an n CHF, can be used as a *binding commitment* for m: a verifier is convinced that the prover knows m simply by asking him to share h, with overwhelming confidence ($\approx 1 - \frac{1}{2^n}$). While often referred to as if they were humans, provers and verifiers are formally described

by some model of computation, usually deterministic Turing machines, which often can only harness a limited amount of resources (time and space), tipically at most polynomial in the size of the game instance statement[2].

If the prover wants to commit to a list of k messages, a possibility would be to share with the verifier the hash of every message: this would require a $\mathcal{O}(k)$ communication cost and a $\mathcal{O}(k)$ verification cost. A slightly better alternative would be for the prover to share $H(\{m_1, \ldots, m_k\})$: the communication cost would only be $\mathcal{O}(1)$, but verification would still cost $\mathcal{O}(k)$.

Definition 8 (Merkle Tree [9])**.** Given some $k \in \mathbb{N}$, a CHF H and a set of messages $S = \left\{m_1, \ldots, m_{s^{k-1}} \mid \forall i \colon m_i \in \{0,1\}^*\right\}$, a *Merkle Tree (MT)* is a complete binary tree of height k such that:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{k-1}}$ contain $H(m_1), \ldots, H(m_{2^{k-1}})$.
2. Every other node ν contains $H(\nu_l, \nu_r)$, where ν_l is the left child of ν and ν_r is the right child of ν.

By using Merkle trees, the prover only needs to send to the verifier, as a commitment for some message m_i among $k = 2^{\lfloor \log_2(k) \rfloor}$ messages, the contents of the co-path from the leaf containing m_i to the root (plus the hash of m_i): this requires just $\mathcal{O}(\log_2(k))$ communication effort and $\mathcal{O}(\log_2(k))$ verification effort. Another advantage of Merkle trees is that bottom-up construction is very easy to parallelize, and its usefulness can be appreciated even more when considering a scenario where different messages actually belong to different provers.

Definition 9 (Augmented Binary tRee [2])**.** Given some $k \in \mathbb{N}$, a CHF H, and a set of messages $S = \left\{m_1, \ldots, m_{2^{k-1}+2^{k-2}-1} \mid \forall i \colon m_i \in \{0,1\}^*\right\}$, an *Augmented Binary tRee (ABR)* is a complete binary tree of height k augmented with *middle* nodes, such that:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{k-1}}$ contain $H(m_1), \ldots, H(m_{2^{k-1}})$.
2. There are no middle nodes in the leaf layer.
3. The middle nodes $\nu_{2^{k-1}+1}, \ldots, \nu_{|S|}$ contain $H(m_{2^{k-1}+1}), \ldots, H\big(m_{|S|}\big)$.
4. Every other node ν contains $H(\nu_l \oplus \nu_m, \nu_r \oplus \nu_m) \oplus \nu_r$, where ν_l is the left child of ν, ν_r is the right child of ν, and ν_m is the middle child of ν, or 0 if ν doesen't have a middle child.

Notice the use of the $\oplus$ operation inside the ABR: while messages of length n are usually treated as elements of $\{0,1\}^n$, they can also be treated as n-bit integers over some field $\mathbb{F}_q$, so $\oplus$ represents addition over the field of choice.

ABRs can store 50% more messages than Merkle Trees for the same height, resulting in the same number of calls to H, at the cost of performing 3 additional $\oplus$ operations for every call, whose cost is almost negligible, especially in ZK systems.

[2]Although humans can be assimilated to a computational model, it is not easy to formalize the eventuality of the prover threatening the verifier to make him accept his proof…

3 ZK-SNARK

We saw in Section 2 how a prover can convince a verifier about the knowledge of some message m, with a high confidence and a small communication effort, by using a CHF H. However, the underlying assumption was that m is known by the verifier: when the prover sends h, the verifier can check whether $H(m) = h$ and therefore accept or reject. In this Section, we will see how a prover can convince a verifier without the need to disclose possibly secret information. In particular, we will focus on provable computation, that is, when the prover wants to convince the verifier that he correctly computed some function.

3.1 Zero Knowledge Proofs

Before diving into provable computation, we must introduce the more general concept of Zero Knowledge Proof system.

Definition 10 (Zero-Knowledge Proof [7])**.** Given a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, a secret x, known only to $\mathcal{P}$, and some statement σ of whose truth $\mathcal{P}$ wants to convince $\mathcal{V}$ by means of some proof π, we call a Zero-Knowledge Proof (ZKP) system any protocol which satisfies the following properties:

- **Soundness**: $\neg\sigma \implies \mathcal{V}(\pi) = \bot$.
- **Completeness**: $\sigma \implies \mathcal{V}(\pi) = \top$.
- **Zero-Knowledge**: It is *hard* for $\mathcal{V}$ to derive x given σ and π.

While formal proofs have been known for millennia, only in the last century, with the advent of modern cryptography, researchers started considering the possibility of having proofs of statements which, while able to convince someone of their truth, didn't leak information about how they were obtained. Zero-Knowledge systems proves particularly useful in *ARgument of Knowledge* scenarios (ZK-ARK): the prover $*$ wants to convince the verifier $\mathcal{V}$ that he knows a solution to some problem, assuming there is one, without revealing it. It must be noted though that known ZK-ARK systems do not guarantee the formal soundness of the proof: there is a small probability that, given some false statement σ and an (invalid) proof π, then $\mathcal{V}(\pi) = \top$, so it is important to keep this probability small (say, 2^{-128}).

Definition 11 (ZK-SNARK [3])**.** Given a prover $\mathcal{P}$, a verifier $\mathcal{V}$, a statement σ, and a proof π, a Zero-Knowledge Non-interactive ARgument of Knowledge (ZK-SNARK) system is any ZK-ARK system which is:

- **Succinct**: $SPACE(\pi) = o(\log(\sigma))$.
- **Non-interactive**: The only communication required by the system is the exchange of σ and π.

Succinctness is an important property in many scenarios, like blockchains, since we cannot afford to use too much resources to transmit and store the proofs, and non-interactivity of the process allows for efficient verification when multiple parties are involved.

One of the most important applications of ZK-SNARK systems is in *provable computation*, where the prover wants to convince the verifier that he correctly performed some computation (e.g. a cryptocurrency transaction).

3.2 The Pinocchio Protocol

A very famous ZK-SNARK system for verifiable computation is the *Pinocchio* protocol, which was the first one efficient enough to be practical [12]. Pinocchio uses a lot of mathematical machinery, while we tried to introduce most of the required background in Section 2, it will still be not trivial to understand fully *how*, and, more importantly, *why*, it actually works. While it is out of the scope of this report to go into the deeper details of the protocol, we will provide an overview of the main parts which are going to affect the design of secure and efficient hash functions for ZK-SNARK.

First of all, Pinocchio does not allow the encoding of arbitrary languages, i.e. it is not Turing complete, but we are restricted to arithmetic circuits over some big prime field $\mathbb{F}_p$ (typically, $p \approx 2^{256}$). The main limitation arising from this restriction is that we can only express bounded computation (i.e. no loops whose length depends on some variable condition). This issue can be mitigated by writing a *circuit synthesizer* in a Turing complete language which is able to build parameterized arithmetic circuits 'on the fly'. Given an arithmetic circuit ϕ over a prime field $\mathbb{F}_p$, Pinocchio works as follows:

1. Fix some generator $g \in \mathbb{F}_p$ such that $\mathbb{G} = \langle g \rangle$ and an order-preserving non-trivial bilinear map $B \colon \mathbb{G} \times \mathbb{G} \mapsto \mathbb{G}_\mathsf{T}$.

2. Build the R1CS $\mathcal{C}$ associated with ϕ.

3. Build the QAP $\mathcal{Q}$ associated with $\mathcal{C}$.

4. A trusted third party $\mathcal{T}$ generates a set of random elements $R \in \mathbb{F}_p$ which, together with g, are used to build a *prover key* $K_\mathcal{P} \in \mathbb{G}$ of size $\Theta(|\phi|)$ and a *verifier key* $K_\mathcal{V} \in \mathbb{G}$ of size $\Theta(|\phi_{IO}|)$. The random data is deleted immediately after use (*toxic waste*).

5. The prover know executes ϕ, computes all the intermediate values, and uses them to solve $\mathcal{C}$ and $\mathcal{Q}$, finding a solution (p, h) for $\mathcal{Q}$.

6. The prover chooses some value $x \in \mathbb{F}_p$ with which he computes $p(x)$ and $h(x)$, and uses $K_\mathcal{P}$ to encrypt them, producing a proof of the kind $k^{p(x)} = k^{t(x)h(x)}$, which has size $\Theta(1)$, and is sent to the verifier.

7. The verifier uses the key $K_\mathcal{V}$, exploiting the bilinear map B, to verify that $k^{p(x)} = k^{t(x)h(x)}$, which implies that $p(x) = t(x)h(x)$, which implies

with high probability that $p = th$ which, as we know, implies that ϕ was correctly executed.

Note how all the work that has to be done from steps 1 to 4 depends only on the circuit, not on some particular execution, so everything that has been computed in those steps can be saved and reused later.

3.3 CHFs for ZK-SNARK systems

We mentioned that Pinocchio, like other ZK-SNARK systems, works on big prime fields. The most famous CHFs, like MD5 [14] or the SHA families, make extensive use of bit-wise operations, basically working on some field of the type $\mathbb{F}_{2^n}$. When translating these hash functions as arithmetic circuits over a prime field $\mathbb{F}_p$, we incur in a huge space and time overhead.

Example 4. *Consider some $S, T \in \{0,1\}^n$ for some $n \in \mathbb{N}$, and assume we want to compute S XOR T. If we work over $\mathbb{F}_{2^n}$ the resulting arithmetic circuit would simply be $x \oplus y$, with $x = \sum_i S_i 2^{i-1}$ and $y = \sum_i T_i 2^{i-1}$.*

On the other hand, to express the same operation over $\mathbb{F}_p$ we consider S and T as vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n \mid \forall i \colon \boldsymbol{x}_i = S_i \wedge \boldsymbol{y}_i = T_i$. For a single-bit XOR operation, we use the circuit $x \oplus y - 2(x \otimes y)$; to represent an n-bit XOR, we concatenate n such circuits. What required a single addition in $\mathbb{F}_{2^n}$ requires $3n$ additions and n multiplications in $\mathbb{F}_p$!

Furthermore, since $\boldsymbol{x}$ and $\boldsymbol{y}$ are defined over $\mathbb{F}_p^n$, we must also guarantee that their elements are either 0 or 1, by explicitly adding the constraint $(x) \otimes (x-1) = 0$ for every $x \in \boldsymbol{x}, \boldsymbol{y}$.

Having efficient CHFs for ZK systems is extremely important, as they are the fundamental building block of commitment protocols which are in turn the pivot of the secure transaction protocols used by cryptocurrencies like ZCash[3]. For this reason, we want CHFs which use native operations over $\mathbb{F}_p$: while they are typically much slower when implemented, say, in x86 assembly, they become immensely faster in ZK-SNARK frameworks due to their extremely lower multiplicative complexity.

Definition 12 (MiMC primitives [1])**.** Given a finite field $\mathbb{F}_p$, $r = \left\lceil \frac{\log_2(p)}{\log_2(3)} \right\rceil$, some $\boldsymbol{c} \in \mathbb{F}_p^r$ and the functions:

$$\forall i < r \colon f_i(x,k) \colon \mathbb{F}_p \times \mathbb{F}_p \mapsto \mathbb{F}_p = x^3 \oplus k \oplus \boldsymbol{c}_i$$

the *MiMC keyed permutation* is defined as:

$$E(x,k) \colon \mathbb{F}_p \times \mathbb{F}_p \mapsto \mathbb{F}_p = (f_r \circ \cdots \circ f_1)(x,k) \oplus k$$

By fixing $k = 0$, we obtain the *MiMC permutation* $P(x)$. By applying the Davies-Meyer and the Merkle-Damgård constructions to the MiMC permutation, we obtain the *MiMC hash function* $H(x)$.

[3] `https://z.cash/`

4 Experiments

We said that zero-knowledge provable computation is especially interesting in blockchain environments, like cryptocurrencies, where users commit to transactions by means of a tree-mode of hash. We implemented and tested several combinations of cryptographic primitives for tree-path verification, using the C++ language and the facilities based on the Pinocchio protocol provided by the `libsnark` library. In particular, we implemented:

- Hash-agnostic Merkle trees.
- Hash-agnostic ABRs.
- SHA-256 (adapted from the versionprovided by `libsnark`) and SHA-512.
- 1/2/1 MiMC-256, single and double key MiMC-512 Feistel (MiMC-512F) OWCFs over prime fields.

Since the computational complexity of Pinocchio is basically defined by the number of multiplications in the arithmetic circuit, we can predict the expected performance of the various hash modes. Given some n-bit hash function H and its arithmetic circuit $\phi(H)$, the Merkle tree $\mathcal{T}_H$ of height h and the path checking arithmetic circuit $\phi(\mathcal{T}_H)$, then:

$$\left|\phi(\mathcal{T}_H)_\otimes\right| = (h-1)\left|\phi(H)_\otimes\right|$$

If we consider an ABR $\mathcal{A}_H$ of the same height, the complexity of the $\otimes$ operations depends on whether H works natively over a prime field or a binary field. In the former case:

$$\left|\phi(\mathcal{A}_H)_\otimes\right| = \left|\phi(H)_\otimes\right| + (h-2)\left(3 + \left|\phi(H)_\otimes\right|\right)$$

In the latter case:

$$\left|\phi(\mathcal{A}'_H)_\otimes\right| = \left|\phi(H)_\otimes\right| + (h-2)\left(3n + \left|\phi(H)_\otimes\right|\right) + 2n(h-1)$$

If we compare the multiplication complexity of $\mathcal{T}_H$ and $\mathcal{A}_H$, we get:

$$\frac{\left|\phi(\mathcal{A}_H)_\otimes\right|}{\left|\phi(\mathcal{T}_H)_\otimes\right|} = 1 + 3\left(\frac{1}{\left|\phi(H)_\otimes\right|} - \frac{1}{(h-1)\left|\phi(H)_\otimes\right|}\right) \approx 1$$

Tipically, $\left|\phi(H)_\otimes\right| > 10^2$, so the 25% increment in density offered by ABRs over Merkle Trees comes at a negligible cost w.r.t. multiplicative complexity, as long as $\oplus$ is intended over the native ZK-SNARK field. Table 1 shows the multiplicative complexity of the hash functions under study. It is likely that the values for SHA-256 and SHA-512 can be significantly optimized, but are still in the range of the tens of thousands, with SHA-512 requiring $\approx 2.5\times$

Name	Rounds	muls/round	tot. muls
SHA-256	64	≈ 512	≈ 43776
SHA-512	80	≈ 1024	≈ 111104
MiMC-256	160	2	639
SK-MiMC-512F	320	2	2558
DK-MiMC-512F	400	2	1598

Table 1: Multiplicative complexity of the implemented hash functions. The 'tot. muls' column counts multiplications independent from the number of rounds, and ones dependent on the construction (e.g. Davies Meyer).

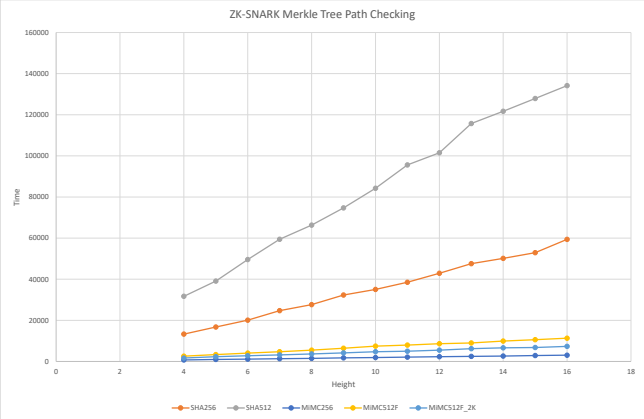

Figure 2: Total time (in ms) required for synthesizing and solving a ZK-SNARK Merkle Tree path veritication circuit at varying heights.

more multiplications than SHA-256. Tests of our implementations confirm the theoretical results[4].

Figure 2 shows the total time, in milliseconds, required to create an arithmetic circuit for Merkle tree path verification, convert it to R1CS, build the QAP and the proof, and verify the proof, using different underlying hash functions. As expected, MiMC-256 is the fastest one, followed by double-key MiMC-512F and single-key MiMC-512F in the respective order, with SHA-256 and SHA-512 being extremely slower.

Figure 3 shows the computational intensity of the different parts of the Pinocchio protocol: since key and proof generation times completely dominate the cost, and these are transparent to the library user, it is extremely important to optimize the arithmetic circuit layout, even if it makes the synthesizing code more complex.

Finally, in Figure 4 we compared the performance of path verification over

[4]For more details on how the various primitives have been implemented, refer to Massimo Del Prato's report, and the repository: `https://github.com/sca-research/merkle_tree`

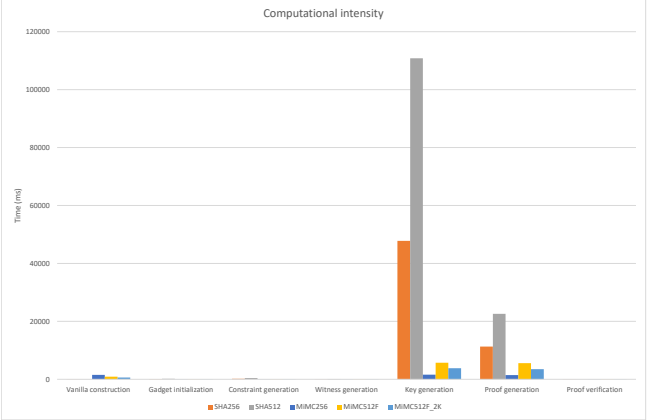

Figure 3: Computational intensity of the various phases of the ZK-SNARK protocol: key and proof generation dominate the cost.

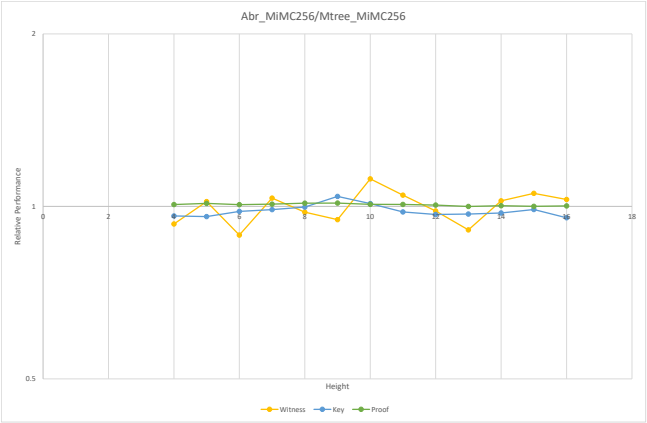

Figure 4: Relative performance of path verification over ABRs compared to Merkle Trees using MiMC-256; we can see that the difference is negligible.

ABRs compared to Merkle Trees when using a field-native CHF like MiMC-256. The difference, as expected, is negligible, and even for non-native functions like SHA-256, the Merkle Tree is about 10–20% faster than the ABR.

5 Conclusions and future directions

In this report we studied the ZK-SNARK Pinocchio protocol, implemented the 'Pinocchio-friendly' hash function MiMC, together with the compact ABR mode of hash, using the `libsnark` library, and we tested their performance in the path verification problem comparing them to the traditional SHA over Merkle Trees combination.

In the last years there has been a number of proposals for efficient primitives in Zero-Knowledge contexts, which are of particular interest for blockchain and cryptocurrency technologies. While MiMC is certainly extremely more efficient than SHA in a ZK setting, it seems like there is still room left for improvement, both in the design of ZK-SNARK systems and in cryptographic functions.

References

[1] Martin Albrecht, Lorenzo Grassi, Christian Rechberger, Arnab Roy, and Tyge Tiessen. Mimc: Efficient encryption and cryptographic hashing with minimal multiplicative complexity. Cryptology ePrint Archive, Paper 2016/492, 2016. `https://eprint.iacr.org/2016/492`.

[2] Elena Andreeva, Rishiraj Bhattacharyya, and Arnab Roy. Compactness of hashing modes and efficiency beyond merkle tree. *CoRR*, abs/2104.15055, 2021.

[3] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. Cryptology ePrint Archive, Paper 2013/879, 2013. `https://eprint.iacr.org/2013/879`.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[5] Quynh H. Dang. *Secure Hash Standard*. Jul 2015.

[6] Manfred Droste and Werner Kuich. *Semirings and Formal Power Series*, pages 3–28. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[7] Oded Goldreich. Zero-knowledge twenty years after its invention, 2002. oded@wisdom.weizmann.ac.il 12026 received 5 Dec 2002.

[8] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 2001.

[9] 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.

[10] Ralph Charles Merkle. *Secrecy, Authentication, and Public Key Systems*. PhD thesis, Stanford University, Stanford, CA, USA, 1979. AAI8001972.

[11] A. Narayanan, J. Bonneau, E. Felten, A. Miller, and S. Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction*. Princeton University Press, 2016.

[12] Bryan Parno, Craig Gentry, Jon Howell, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. Cryptology ePrint Archive, Paper 2013/279, 2013. `https://eprint.iacr.org/2013/279`.

[13] Bart Preneel. *Davies–Meyer Hash Function*, pages 136–136. Springer US, Boston, MA, 2005.

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

[15] John E. Savage. *Models of Computation: Exploring the Power of Computing*. Addison-Wesley Longman Publishing Co., Inc., USA, 1st edition, 1997.

[16] P.W. Shor. Algorithms for quantum computation: discrete logarithms and factoring. In *Proceedings 35th Annual Symposium on Foundations of Computer Science*, pages 124–134, 1994.