Implementing compact tree-hashing commitment verification in ZK-SNARK

Stefano Trevisani

June 1, 2022

Contents

1 Introduction

One of the biggest revolutions of the last decade has been the widespread adoption of blockchain-based technologies. In particular, cryptocurrencies like Bitcoin or Ethereum are widely known even among the non-crypto-enthusiasts and a huge market is growing around them. There are highly interesting applications of the blockchain even outside the financial world: in fact, anything which requires some degree of 'verifiability' in a non-trusted environment can benefit from using a blockchain. Tipically, a block of the chain does not correspond to a single transaction, as the blockchain would become too big to be stored: many transactions are inserted into a tree data-structure, called Merkle Tree (MT), which is computed bottom-up and whose root is then inserted into the blockchain. In a traditional setup (e.g. the aformentioned cryptocurrencies), all the data which is required to build a block of the blockchain is of public knowledge: when an Ethereum transaction happens, the details of that transaction are shared with the network for it to be validated. Of course, there are many scenarios were one would like the data to remain secret, as there could be risks for privacy (e.g. transactions) or for intellectual property (e.g. algorithms).

Zero-Knowledge Succint Non-interactive ARgument of Knowledge (ZK-SNARK) systems are cryptographic frameworks which allow for a party to convince other parties that he 'knows something' without revealing anything else. For example, one could convice other users that the hash of a transaction is valid without revealing the details of that transaction. ZK systems work over prime fields, and hashing algorithms like SHA-256, which are quite efficient in a 'vanilla' scenario, can become extremely slow when translated. For this reason, new hashing algorithms like MiMCHash have been designed with the ZK scenario in mind, and Augmented Binary tRee (ABR) tries to improve Merkle Trees by processing more transactions without requiring more hash function calls.

In the remainder of the report, for clarity we will always have in mind the cryptocurrency scenario, but we want to make clear that all the other use-cases are the same[1]. In Section 2, we introduce hash functions and tree-based modes of hashing. In Section 3, we introduce ZK-SNARK systems and brifely describe how they can be implemented. In Section 4, we introduce `libsnark`, a C++ library that provides many facilities to implement ZKSNARK algorithms with relative ease, and we will discuss the implementation of MiMC, MTs and ABRs using this library. In Section 5, we perform some experiments to compare MiMC with SHA, and MTs with ABRs in a ZK setting. Finally, in Section 6 we draw our conclusions and discuss possible future work directions.

[1] Up to isomorphism, of course.

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. 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 (tipically, 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) is believed to be hard for deterministic Turing machines, but it is easy for non-deterministic ones. Unfortunately, non-deterministic Turing machines (along with any other non-deterministic model of computation) are more of a mathematical tool than anything, and there seems to be no practical way to efficiently solve the problems which would take Non-deterministic Polynomial time (NP-complete problems) as stated by the strong Church-Turing thesis. While it is widely belived that efficiently solving NP-complete problems is impossible, there are some problems which lie in a 'gray zone' between NP-COMPLETE and P (i.e. problems which can be solved in deterministic polynomial time): they are believed to be hard for deterministic models (hence, they are not in P), but there is no proof that they are NP-COMPLETE. The most famous of such problems is factorization: with the advent of quantum-computing, which challenges the strong Church-Turing thesis, Shor devised an efficient quantum algorithm for factorizing numbers. While still far from usable in practical cases, its existence proves 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 computational models tipically operate over binary strings, that is, elements of $\{0,1\}^*$, where $*$ indicates Kleene's closure, 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}.

For example, $\mathbb{R}$ and $\mathbb{C}$ are fields, as is Boole's algebra $\mathbb{B} = (\{0,1\}, \text{XOR}, \text{AND})$. 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 $x, y, z \ldots$.

If $|\mathbb{F}| \in \mathbb{N}$, then $\mathbb{F}$ is a *finite field*. We are particularly interested in finite fields of the kind:

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

where p is a prime and $\oplus_p, \otimes_p$ denote addition and multiplication modulo p (for example, $\mathbb{B} = \mathbb{F}_2$). Tipically, 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 $\boldsymbol{M}_{i,j}$. 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 $\begin{pmatrix} \boldsymbol{A}; \boldsymbol{B} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}^\intercal & \boldsymbol{B}^\intercal \end{pmatrix}^\intercal$ 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}.

For example, $\mathbb{Z}$ with only addition is a group (where $1_{\mathbb{Z}} = 0$), as is $\mathbb{Q}$ with only multiplication and without 0 (as it's not invertible). We denote elements and variables over groups in the same way we do for fields. We can define *group exponentiation* following the natural definition:

$$\forall x \in \mathbb{G}, \forall n \in \mathbb{N} \colon x^n = \bigodot^{n} x$$

Any group $\mathbb{G}$ for which $|\mathbb{G}| \in \mathbb{N}$ is a *finite group*. We can build a finite group from a *generator* set S and a composition operation $\odot$ by closing S under $\odot$:

$$\langle S \rangle^i = \begin{cases} S & i = 0 \\ \langle S \rangle^{i-1} \cup \{x \odot y \mid x, y \in \langle S \rangle^{i-1}\} & i > 0 \end{cases}$$
$$\mathbb{G} = \langle S \rangle = (\min_i \langle S \rangle^i \mid \langle S \rangle^i = \langle S \rangle^{i-1}, \odot)$$

We are particularly interested in *cyclic groups*, i.e. finite groups of the type $\mathbb{G}_q(g) = \langle g \rangle = (\{g^i \bmod q\}_{i \in \mathbb{N}}, \otimes_q)$, 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 $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_\intercal$, a *bilinear map* is any function:

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

If $|\mathbb{G}| = |\mathbb{G}'|$, then B is *order-preserving*, and if $\mathbb{G}_1 = \langle g_1 \rangle \wedge \mathbb{G}_2 = \langle g_2 \rangle \wedge \mathbb{G}_\intercal = \langle B(g_1, g_2) \rangle$, then B is *admissible*.

We are interested in (order-preserving, admissible) bilinear maps where $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}$. Bilinar maps have many application in cryptography, as they are an essential tool to exploit the hardness of the discrete logarithm problem.

2.3 Arithmetic Circuits

Sequences 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*

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*) and $\otimes$ (*multiplication vertices*). 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 ingoing edges are called *input vertices*, while vertices without outgoing edges are called *output vertices*.

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* $\mathbb{F}_{13}$*:*

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

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:

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

[1]However, this transformation can affect the succintness 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.

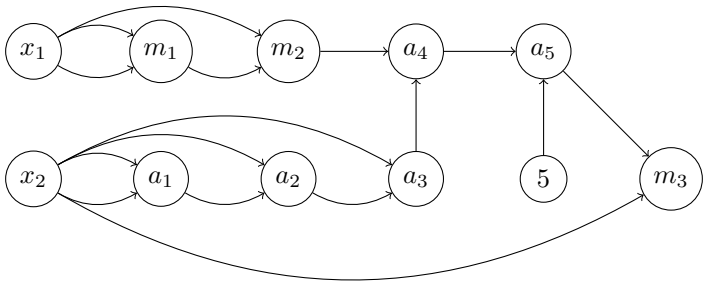

Figure 1: DAG of the circuit in Example 1. We have the two variable/input vertices x_1 and x_2, the constant vertex 5, the addition vertices $a_1, \ldots, a_5$ and the multiplication vertices m_1, m_2 and m_3, which is also an output vertex.

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 $(\boldsymbol{A}\boldsymbol{s})(\boldsymbol{B}\boldsymbol{s}) = \boldsymbol{C}\boldsymbol{s}$*.*

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:

$$\boldsymbol{s} \in \mathbb{F}^n \mid (\boldsymbol{v}\boldsymbol{s})(\boldsymbol{w}\boldsymbol{s}) - \boldsymbol{y}\boldsymbol{s} \bmod t = 0$$

Then, the polynomials $p = (\boldsymbol{v}\boldsymbol{s})(\boldsymbol{w}\boldsymbol{s}) - \boldsymbol{y}\boldsymbol{s}$ 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:

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

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* $L(\boldsymbol{z}, \boldsymbol{A}_5^\intercal)$*:*

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

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

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

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

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:

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

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

Since:

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

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 succint than the corresponding valid assigment $\boldsymbol{s}$ 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 confindence (although not certainity) 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 arbitrarly 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 sohuld 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. Ideally, a cryptographic hash function should behave like a perfect random function. This is of course impossible, as the output of an hash function must only depend deterministically on its input; the aim then is to build functions which are hard to distinguish from a random function.

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 $H(m_1) = H(m_2)$.

While some of the requirements might seem redundant (for example, if it is hard for an attacker to find a collision for chosen messages, it must be hard when one is fixed), the difference usually lies in how exactly we define hardness for each property. For collision resistance, an ideal CHF requires about $2^{n/2}$ evaluations to find a collision (birthday paradox), while for preimage resistance it would require about 2^n evaluations. Tipically, a CHF is built by applying some known secure constructions to functions which are simpler to devise.

Definition 8 (Pseudorandom keyed permutation)**.** Given $l, n \in \mathbb{N}$, an *l/n(-bit) pseudorandom keyed permutation (PKP)* is any bijective function:

$$F\colon \{0,1\}^l \times \{0,1\}^n \mapsto \{0,1\}^l$$

which is hard to distinguish from an uniform random distribution.

PKPs are often built by iterating a keyed permutation F for some number r of rounds, since F by itself might be relatively easy to invert. A block cipher is a pseudorandom keyed permutation which changes the key being used in each round through a key-scheduling function. Unkeyed permutations can be derived from keyed ones simply by fixing the key to some arbitrary value.

Definition 9 (One-way compression function)**.** Given $l_1, l_2, n \in \mathbb{N}$, an *$l_1/n/l_2$(-bit) one-way compression function (OWCF)* is any function:

$$F\colon \{0,1\}^{l_1} \times \{0,1\}^n \mapsto \{0,1\}^{l_2}$$

There are many known ways to build OWCFs from pseudorandom keyed permutations, and, in turn, CHFs from OWCFs. 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)**.** *Given a l/n pseudorandom keyed permutation E, some $i, k \in \mathbb{N}$, some $v \in \{0,1\}^l$, and some $m \in \{0,1\}^{kn}$, then any function F_E such that:*

$$F_{E,i}(v, m) = \begin{cases} E(v, m_{1\ldots n}) & i = 1 \\ E(F_{E,i-1}(v, m), m_{i(n-1)\ldots in}) & 2 \leq i \leq k \end{cases}$$

$$F_E = F_{E,k}$$

is a $l/kn/l$ OWCF.

Theorem 2 (Merkle-Damgård construction)**.** *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 funcion:*

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

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

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

then any function H_F such that:

$$H_{F,i}(v,m) = \begin{cases} F(v, m_1) & i = 1 \\ F(H_{F,i-1}(v,m), m_i) & 1 < i \leq |P(m)| \end{cases}$$
$$H_F = H_{F,|P(m)|}$$

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 10 (Merkle Tree)**.** Given some $k \in \mathbb{N}$, a CHF H and a set of messages $S = \{m_1, \ldots, m_{s^{k-1}} \mid \forall i\colon m_i \in \{0,1\}^*\}$, 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

[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…

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 11 (Augmented Binary tRee)**.** Given some $k \in \mathbb{N}$, a CHF H, and a set of messages $S = \{m_1, \ldots, m_{2^{k-1}+2^{k-2}-1} \mid \forall i\colon m_i \in \{0,1\}^*\}$, 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 $H(m_{2^{k-1}+1}), \ldots, H(m_{|S|})$.
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 $\mathbb{F}_q$: if $q = 2^n$, then $\oplus$ means bitwise XOR (i.e. addition in $\mathbb{F}_2$), and if $q = p$ for some prime p, then $\oplus$ means addition in the field $\mathbb{F}_p$.

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 (usually, $TIME(\oplus) \ll TIME(H)$).

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 convice the verifier that he correctly computed some function.

3.1 Zero Knowledge Proofs

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

Definition 12 (Zero-Knowledge Proof)**.** Given a prover P and a verifier V, a secret x, known only to P, and some statement σ of whose truth P wants to convince V by means of some proof π, we call a Zero-Knowledge Proof (ZKP) system any protocol which satisfies the following properties:

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

While formal proofs have been known for millenia, 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 convice 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 P wants to convince the verifier 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 $V(\pi) = \top$, so it is important to keep this probability small (say, 2^{-128}).

Definition 13 (ZK-SNARK)**.** Given a prover P, a verifier 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:

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

Succintness 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. Pinocchio uses a lot of mathematical machinery, and it's not trivial to fully understand *how*, and even more importantly, *why*, it actually works. We will not go into all of the details of the protocol, especially in the last stages which involve *elliptic curve* mathematics, but we will still try to give a good idea of the first stages and an intuition of the last ones, focusing on what determines the computational complexity of this protocol.

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 prime field $\mathbb{F}_p$. The main limitation arising from this restriction is that we cannot express unbounded computation (like infinite loops) or even variably-bounded computation (like loops whose exit condition depends on some non-constant value). This issue can be mitigated by writing a *circuit synthesizer* in a Turing complete language which is able to build parametrized arithmetic circuits 'on the fly'. After we have our arithmetic circuit ϕ over a prime field $\mathbb{F}_p$, the Pinocchio works as follows:

1. The circuit ϕ is made public.
2. Build the R1CS $\mathcal{C}$ associated with ϕ.
3. Build the QAP $\mathcal{Q}$ associated with $\mathcal{C}$.
4. A trusted third party generates some random data, which is used to create a prover key (k_P) and a verifier key k_V. The random data must be kept secret (or even better, deleted after use).
5. The prover executes ϕ, computes all the intermediate values, and uses them to solve $\mathcal{C}$ and $\mathcal{Q}$; in the end, he finds a solution (p, h) for $\mathcal{Q}$.
6. The prover chooses some value x to compute $p(x)$ and $h(x)$, and uses k_P to generate an encrypted proof, of size $\mathcal{O}(1)$, which is sent to the verifier. The encryption scheme exploits group theory so that $p(x) = h(x)t(x)$ if and only if a different (but still easy) operation involving the encrypted values holds in the encrypted space. This involves using a public cyclic group $\mathbb{G}_q = (\{g^i \bmod q\}_{i \in \mathbb{F}_p}, \otimes)$ which is generated by some public element $g \in \mathbb{F}_p$ and a prime number $q \in \mathbb{N}$, together with a bilinear

mapping $B\colon \mathbb{G}_q \times \mathbb{G}_q \mapsto \mathbb{F}_q$ (bilinear means that $B(x^a, y^b) = B(x, y)^{ab}$). The group $\mathbb{G}_q$ was also used to generate k_P and k_V.

7.

Due to the homomorphism of the mappings and the properties of QAPs and R1CSs, if the verification is successful, it means that the original algorithm was in fact correctly executed, with high probability. If the verification fails, then certainly the original algorithm was not executed correctly. The verifier learns cannot learn any additional information from this process without performing an infeasible amount of work, therefore this is indeed a ZK-SNARK protocol.