Master Thesis in
Artificial Intelligence and Cybersecurity

Zero-Knowledge friendly cryptographic permutation: theory and implementation

Candidate
Stefano Trevisani

Supervisors
Dr. Arnab Roy
Prof. Alberto Policriti

Co-Supervisor
Prof. Elisabeth Oswald

Tutor
Msc. Matthias Steiner

Academic Year 2021-2022

Institute Contacts
Dipartimento di Scienze Matematiche, Informatiche e Fisiche
Università degli Studi di Udine
Via delle Scienze, 206
33100 Udine — Italia
+39 0432 558400
`https://www.dmif.uniud.it/`

Institute Contacts
Alpen-Adria-Universität Klagenfurt,
Universitätsstraße, 65–67
9020 Klagenfurt am Wörthersee — Austria
+43 463 2700
`https://www.aau.at/`

© 2022 Stefano Trevisani
This work is shared under the Creative Commons 4.0 License Attribution-NonCommercial-ShareAlike.

Abstract

Zero Knowledge (ZK) proof systems have been an increasingly studied subject in the last 40 years. In the last decade, the efficiency of the proposed frameworks, along with the processing power of computing devices, has improved to the point of making ZK computation feasible in real-world scenarios. One of the primary applications lies in hash-tree commitment verification, and in this past five years there has been intense research in proposing ZK-friendly cryptographic primitives. In this work, we begin by studying the history of ZK systems and reviewing the state of the art concerning ZK-friendly cryptographic permutations. We then present a novel, generic algebraic framework to design cryptographic permutations and we apply it to construct a new permutation. Finally, we implement our permutation together with the reviewed ones in the Groth16 ZK-SNARK framework and compare their efficiency for Merkle-Tree commitment verification.

Contents

1 Introduction

An important research branch of cryptography which emerged in the last fourty years is the study of *Zero Knowledge Interactive* (ZK-I) protocols, and more specifically zero knowledge proof systems (ZKP) [13]. The main idea behind ZKP systems is to have two (or, in some cases, more) parties, where one is the *prover* and the other is the *verifier*: in a classical proof system, the prover must be able to convince the verifier that a certain statement is true, when this is indeed the case, but the verifier cannot be fooled if the statement is actually false. In a ZKP system we also require that the verifier does not get any useful additional information (i.e. knowledge) other than the truth, or lack thereof, of the statement. This additional requirement is particularly interesting when dealing with statements that are notoriously (believed to be) hard to prove, so that the verifier would not be realistically able to prove them in a reasonable amount of time. As a simple example, a prover would like to show that a propositional logic formula is satisfiable (an instance of the famous SAT problem) without revealing the satisfying assignment to the verifier.

Along the years, additional interesting and useful properties have been added to extend and improve the capabilities of ZKP systems. For example, we would like to have a *Non-interactive* (ZK-NP) protocol, to minimize the amount of required communication and have it happen only at the beginning and at the end of the protocol. We could also want to relax the soundness requirement so that it is guaranteed only against computationally bounded provers: in this case, instead of 'proof' we use the term *ARgument of Knowledge*, and hence we can have ZK-IARK/ZK-NARK systems. More recently, there has been a research effort towards reducing the length of the ARK by ensuring that it is constant size or at most bounded by a logarithmic function in the length of the theorem statement: such systems are said to be *Succint*. Implementations of ZK-SNARK system, like Pinocchio [20] or Groth16 [16], represent the current state of the art (SoTA) of ZKP systems, and allow to generate proofs to verify any computation representable by means of *bounded arithmetic circuits*. A major downside of ZK-SNARK protocols is their need of a trusted third party (TTP) to setup the system, hence current research is studying *Transparent* systems (ZK-STARK) to address this issue [5].

An especially useful application of ZKP systems is proving knowledge of a preimage for a cryptographic hash function digest (a.k.a. commitment). Many data integrity systems, such as blockchains, rely on Merkle Trees [18] to ensure efficient commitment validation, especially in dynamic environments. In Merkle Trees, an hash function is applied in a bottom-up fashion: the leaves will contain the data owned by some parties, while the root will contain the tree commitment. In a non-ZK setting, a prover would send the verifier his leaf together with the co-path, the verifier would then recompute the tree commitment and compare it with the public one and be convinced whether or not the prover does actually own the leaf. On the other hand, in a ZK-SNARK setting, we first have to represent the computation through a bounded arithmetic circuit, i.e. we are allowed to use exclusively a constant number of additions and multiplications over some suitable finite field. The circuit, together with a *proving key* provided by a TTP, and some private and public data, is then used by the prover to generate a proof which is sent to the verifier, who in turn uses a *verification key*, again provided by the same TTP, to assert whether the circuit computation was performed correctly.

While the various ZK-SNARK (or ZK-STARK) frameworks differ in the details, it is intuitive to see that the complexity of generating the proof (which dominates the cost of the protocol) must depend on the size of the circuit, which in turn depends on the amount of multiplications and additions performed in the computation: in the case of Merkle Tree commitment verification, most of the computation consists in iterating the underlying hash function. Since the finite field over which ZK-SNARK frameworks works is typically a huge prime field ($\approx 2^{256}$ elements), traditional hash functions like MD5 [22] or SHA [9], which are designed to be extremely efficient on classical boolean circuits, become extremely inefficient in the ZK case.

It is no wonder then, that in the last years researchers began to study so-called ZK-friendly cryptographic permutation (ZKFCP) designs that exploit the features of large prime fields to be efficient when translated into airhtmetic circuit, fundamentally resulting in a one-to-one mapping. Being a new research topic, these designs have seen a rapid series of improvements [2, 15, 14] in the last three years: in a two-part series of papers undergoing publication, we presented an algebraic framework, called *Generalized Triangular Dynamical System* (GTDS), which allows to express many of the existing cryptographic permutation designs and eases the construction of new ones, while at the same time giving strong security guarantees, and we then applied it to devise the `Blocc` blockcipher and the `Stamp` hash function. Using the `libsnark`[1] library (an implementation of the Groth16 framework), we implemented our hash function, along with other competitor hash functions and a hash-agnostic variable-arity Merkle Tree circuit template, in a `C++` project which we then used to compare their real-world performance for same-level security gaurantees in various scenarios.

Structure of the thesis

This work is organized in two parts: Part I contains the background of the work, presenting all the mathematical, computational and cryprographical tools and concepts required to understand the theory, the history and the applications of ZKFCPs. In Part II, we begin with a review of state of the art ZKFCPs, we then present the GTDS algebraic framework, its instantiation in the form of the `Blocc` block cipher and the `Stamp` hash function and we conclude with an implementation analysis and experimental comparison between the current SoTA and the GTDS constructions.

[1]`https://github.com/scipr-lab/libsnark`

I

Foundations

Zero Knowledge Proof (ZKP) systems are a relatively recent research topic: while the idea in itself, like many other beautiful ideas, is simple and elegant, its formalization, and even more so its realization, is all but trivial. A first rigorous description of what it means for a proof system to be *Zero Knowledge* was given by S. Goldwasser, S. Micali and C. Rackoff in 1985 [13] (the work was later updated in 1989).

To fully understand the properties of ZKP system, one needs to have an understanding of both fundamental and more advanced notions from the fields of group theory, computational theory and cryptographical theory. This is even more necessary for ZK-SNARK systems and ZK-friendly hash functions. For this reason, in this first part of the work we will (hopefully) give an exhaustive description of the tools required to have a better grasp of the results that will be presented in the second part.

2 Mathematical Background

In this chapter we will introduce all the mathematical concepts behind ZKP and ZK-friendly functions. While we decided, for completeness, to include even some of the more fundamental notions, we still expect the reader to have at least a rough idea of these concepts. Section 2.1 will introduce prime fields, cyclic groups other related notions.

2.1 Finite algebra

In algebra, a *tuple* consisting of one or more *sets* together with one or more *operations* over the sets is called an *algebraic structure*. Such structures can be organized according to a quite wide taxonomy, depending on whether they satisfy certain properties or not. We will denote sets by capital letters (e.g. $S, T, U, \ldots$), a generic operation by a circled dot $\odot$ and algebraic structures by blackboard bold letters (e.g. $\mathbb{A}, \mathbb{B}, \mathbb{C}, \ldots$). We will also denote constants over a set by lowercase letters at the beginning of the alphabet (e.g. $a, b, c, \ldots$) and variables over a set by lowercase letters at the end of the alphabet (e.g. $x, y, z, \ldots$). Finally, we will often use the term algebra to mean algebraic structure, whenever we belive the meaning to be clear from the context.

Remark 2.1. Some symbols will be reserved to denote some common algebraic structures. In particular, $\mathbb{B}$ will denote the boolean algebra, while $\mathbb{N}, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$ and $\mathbb{C}$ will denote, respectively, the natural, integer, rational, real and complex numbers.

We will denote the *cardinality* of set S by $|S|$, and use the same notation for the *order* of an algebraic structure and for the *arity* of an operation: for example, if $\odot$ is a binary operation, like integer addition[1], then $|\odot| = 2$. When an algebraic structure $\mathbb{A}$ has exactly one *underlying set* A, we will identify the two, e.g. by writing $x \in \mathbb{A}$ to mean $x \in A$.

Definition 2.1 (Finite algebra)**.** A *finite algebra* is an algebraic structure $\mathbb{A}$ such that $|\mathbb{A}| \in \mathbb{N}$.

Definition 2.2 (Subalgebra)**.** An algebraic structure $\mathbb{A} = (A, \odot_1, \ldots, \odot_n)$ is a *subalgebra* of an algebraic structure $\mathbb{A}' = (A', \odot'_1, \ldots, \odot'_m)$, for some $n \leq m$, if $A \subseteq A'$ and $\forall i \leq n \colon \odot_i \subseteq \odot'_i$.

Elements of differents algebraic structures can be associated through *morphisms*.

[1]if considered as a relation, addition would be ternary.

Definition 2.3 (Homomorphism)**.** Given two algebras $\mathbb{A} = (A, \odot_1, \dots, \odot_n)$, $\mathbb{A}' = (A', \odot'_1, \dots, \odot'_n)$ such that $\forall i \leq n \colon |\odot_i| = |\odot'_i| = a_i$, an *homomorphism* is a map $h \colon A \to A'$ such that:

$$\forall i \leq n, \forall x_1, \dots, x_{a_i} \in A \colon h(\odot_i(x_1, \dots, x_{a_i})) = \odot'_i(h(x_1), \dots, h(x_{a_i})) \qquad \text{(linearity)}$$

We say that $\mathbb{A}$ is homomorphic to $\mathbb{A}'$ through h.

Definition 2.4 (Isomorphism)**.** An *isomorphism* is a bijective homomorphism.

Given two algebras $\mathbb{A}$ and $\mathbb{A}'$, if they are isomorphic through some map h, we write $\mathbb{A} \cong_h \mathbb{A}'$, or more succintly $\mathbb{A} \cong \mathbb{A}'$.

Definition 2.5 (Endomorphism, Automorphism)**.** An *endomorphism* is a homomorphism from an algebraic structure $\mathbb{A}$ to itself. An *automorphism* is an endomorphism which is also an isomorphism.

2.1.1 Groups

We will now introduce some important classes of algebraic structures equipped with one fundamental operation.

Definition 2.6 (Monoid)**.** A *monoid* is a pair $\mathbb{M} = (M, \odot)$, where M is the underlying set and $\odot \colon M \times M \to M$ is the *composition* operation, such that the following properties are satisfied:

$$\forall x, y \in M \colon x \odot (y \odot z) = (x \odot y) \odot z \qquad (\textit{associativity})$$
$$\exists \mathrm{e} \in M \colon \forall x \in M \colon x \odot \mathrm{e} = x \qquad (\textit{identity element})$$

$\mathbb{M}$ is a *commutative (or abelian) monoid*, if it also holds that:

$$\forall x, y \in M \colon x \odot y = y \odot x \qquad (\textit{commutativity})$$

Finally, we can define *exponentiation* as follows:

$$\forall x \in \mathbb{M}, \forall k \in \mathbb{N} \colon x^k = \begin{cases} \mathrm{e} & k = 0 \\ x^{k-1} \odot x & k > 0 \end{cases} \tag{2.1}$$

Definition 2.7 (Cyclic Monoid)**.** A *cyclic monoid* is a monoid $\mathbb{M} = (M, \odot)$ which has a *generator element* g such that:

$$\mathbb{M} = \langle g \rangle = \left(\left\{g^k \mid k \in \mathbb{N}\right\}, \odot\right)$$

Definition 2.8 (Group)**.** A *group* is a monoid $\mathbb{G} = (G, \odot)$, such that:

$$\forall x \in G \colon \exists \bar{x} \in G \colon x \odot \bar{x} = \mathrm{e} \qquad (\textit{inverse element})$$

With the notion of inverse element, we can extend exponentiation as follows:

$$\forall x \in \mathbb{G}, \forall k \in \mathbb{Z} \colon x^k = \begin{cases} x^{k-1} \odot x & k \geq 0 \\ x^{k+1} \odot x^{-1} & k < 0 \end{cases}$$

If $\mathbb{G}$ is also a commutative (resp. cyclic) monoid, then it is a commutative (resp. cyclic) group.

We will sometimes use the notation $\mathrm{e}_{\mathbb{A}}$ to specify the algebra over which we intend to pick the identity element, dropping the subscript when $\mathbb{A}$ is clear from the context.

Remark 2.2. Although we will strive, throughout this work, to be as unambiguous as possible, in some points we will likely be using a particualr symbol to denote different operations. For example, $+$ might denote addition between numbers, or polynomials, or vectors, and so on. This *overloading* will be done mostly in an effort to slim the notation and shift the attention from the operations themselves to the surrounding context. In any case, the semantics will always be clear from the operands and the context.

Example 2.1. The algebra $\mathbb{Z} \setminus \{\times\}$ (i.e. integer numbers without multiplication) is an abelian group: addition is associative and commutative, the identity element is $\mathrm{e} = 0$, and every number x has an inverse $\bar{x} = -x$ (e.g. $\overline{42} = -42$).

Example 2.2. Given a commutative group $\mathbb{G} = (G, \odot)$, consider the algebra $\mathrm{End}(\mathbb{G})_+ = (H, +)$, where H is the set of endomorphisms over $\mathbb{G}$ and $+\colon H \times H \to H$ is such that $\forall h_1, h_2 \in H, \forall x \in G\colon (h_1 + h_2)(x) = h_1(x) + h_2(x)$.

$\mathrm{End}(\mathbb{G})_+$ is a commutative group: $+$ is both associative and commutative, the identity element is $\mathrm{e}_{\mathrm{End}(\mathbb{G})_+} = z$, where z is the zero endomorphism (i.e. $\forall x \in G\colon z(x) = \mathrm{e}_{\mathbb{G}}$); finally, every homomorphism $h \in H$ has an inverse $\bar{h}$ such that $\forall x \in G\colon \bar{h}(x) = \overline{h(x)}$.

Example 2.3. Consider now the algebra $\mathrm{End}(\mathbb{G})_\circ = (H, \circ)$ where $\mathbb{G}$ and H are defined as in Example 2.2, and $\circ\colon H \times H \to H$ is defined as function composition: $(h_1 \circ h_2)(x) = h_1(h_2(x))$.

$\mathrm{End}(\mathbb{G})_\circ$ is a monoid: function composition is associative, and the identity element is $\mathrm{e}_{\mathrm{End}(\mathbb{G})_\circ} = \mathrm{id}$, where id is the identity endomorphism (i.e. $\forall x \in G\colon \mathrm{id}(x) = x$).

2.1.2 Fields

Many algebraic structures rely on two fundamental operations, called *addition* and *multiplication*: two important types of such structures are *rings* and *fields*.

Definition 2.9 (Ring)**.** A *ring* is a triple $\mathbb{O} = (O, \oplus, \otimes)$ where O is the underlying set, $\oplus\colon O \times O \to O$ is the *addition* operation and $\otimes\colon O \times O \to O$ is the *multiplication* operation, such that the following properties are satisfied:

$$\begin{aligned}
&\mathbb{O}_\oplus = \mathbb{O} \setminus \{\otimes\} \text{ is an abelian group} \\
&\mathbb{O}_\otimes = \mathbb{O} \setminus \{\oplus\} \text{ is a monoid} \\
&\forall x, y, z \in O\colon x \otimes (y \oplus z) = (x \otimes y) \oplus (x \otimes z) && (\textit{left distributivity}) \\
&\forall x, y, z \in O\colon (y \oplus z) \otimes x = (y \otimes x) \oplus (z \otimes x) && (\textit{right distributivity})
\end{aligned}$$

If $\mathbb{O}_\otimes$ is a commutative monoid, then $\mathbb{O}$ is a *commutative (abelian) ring.*

Given a ring $\mathbb{O}$ and an element $x \in \mathbb{O}$, its additive (resp. multiplicative) inverse is denoted by $\bar{x}_\oplus$ (resp. $\bar{x}_\otimes$). Similarly, the additive (resp. multiplicative) identity element is denoted by $\mathrm{e}_\oplus$ (resp. $\mathrm{e}_\otimes$). In numeric algebras, the additive and multiplicative inverses are typically equivalent to the opposite $-x$

and the reciprocal $\frac{1}{x}$ (or x^{-1}) respectively, while the additive and multiplicative inverses are typically equivalent to 0 and 1 respectively.

Definition 2.10 (Field)**.** A *field* is a ring $\mathbb{F} = (F, \oplus, \otimes)$ such that $e_\oplus \neq e_\otimes$ and $\mathbb{F}_\otimes \setminus \{e_\oplus\}$ is a commutative group.

Fields are one of the most important and studied algebraic structures: the algebra of real numbers $\mathbb{R}$ is a field, as is the algebra of complex numbers $\mathbb{C}$. Given the set of integers $Z_q = \{0, \ldots, q-1\}$, we denote by $\oplus_q$ integer sum modulo q, and by $\otimes_q$ integer multiplication modulo q. Furthermore, we will denote by $\langle g \rangle_q$ the cyclic group generated by g under the operation $\otimes_q$. The algebra $\mathbb{Z}_q = (Z_q, \oplus_q, \otimes_q)$ is a finite ring $\forall q \in \mathbb{N}$, and it is a finite field if and only if q is prime.

Definition 2.11 (Discrete logartithm)**.** The *discrete logartithm* over some cyclic group $\langle g \rangle$ of order q is the function:

$$\log_g(g^x) \colon \langle g \rangle \to \mathbb{Z}_q = x$$

When the group genreator is clear from the context, we simply write log instead of $\log_g$. Typically, cyclic groups are obtained as the subset of a larger finite field (see Example 2.6).

Example 2.4. Boolean circuits with XOR and AND gates behave like elements of the boolean field $\mathbb{B} = (\{\bot, \top\}, \text{XOR}, \text{AND})$. It is easy to show that $\mathbb{B} \cong \mathbb{Z}_2$. Similarly, k-bit unsigned integers sum and multiplication work as in $\mathbb{Z}_{2^k}$.

Example 2.5. Given an abelian group $\mathbb{G}$, the algebra $\mathbb{H}_\mathbb{G} = \text{End}(\mathbb{G}) = \text{End}(\mathbb{G})_+ \cup \text{End}(\mathbb{G})_\circ$ is the *endomorphism ring* of $\mathbb{G}$: $\text{End}(\mathbb{G})_+$ is an abelian group, $\text{End}(\mathbb{G})_\circ$ is a monoid (cfr. Examples 2.2 and 2.3), and it is easy to show that $\circ$ distributes over $+$ both on the left and the right.

Example 2.6. Consider the cyclic group $\mathbb{G} = \langle 2 \rangle_{23} = (\{1, 2, 3, 4, 6, 8, 9, 12, 13, 16, 18\}, \otimes_{23})$ which has order $|G| = 11$. Let's show that $\log = \log_2$ is an homomorphism between $\mathbb{G}$ and $\mathbb{Z}_{11,\oplus}$: for any two elements $x, y \in \mathbb{G}$, we have:

$$x \otimes_{23} y = 2^{\log(x)} \otimes_{23} 2^{\log(y)} = 2^{\log(x) \oplus_{11} \log(y)}$$

Since $\log_2$ is a bijection, it is also an isomorphism. In fact, one can show that $\forall q \in \mathbb{N}$ and $\forall g < q$ such that $\gcd(g, q) = 1$ (otherwise $\langle g \rangle_q$ would not be a group), then $\mathbb{G} = \langle g \rangle_q \cong_{\log_g} \mathbb{Z}_{|\mathbb{G}|,\oplus}$.

2.1.3 Vector spaces

All the algebraic structures we have seen in the previous section operate on an underlying set whose elements we consider to be, in some sense, atomic. On the other hand, many objects interact with each other exhibiting a multi-dimensional behaviour (e.g. physical forces). The standard structure to deal with such objects are *vector spaces.*

Definition 2.12 (Module)**.** A *module* is a quadruple $\mathbb{M} = (M, \mathbb{O}, +, \odot)$ where M is the underlying vector set, $\mathbb{O} = (O, \oplus, \otimes)$ is the underlying scalar ring, $+ \colon M \times M \to M$ is the *module addition* operation and $\odot \colon O \times M \to M$ is the *scalar multiplication* operation, such that $\mathbb{M}_+ = (M, +)$ is a commutative group and $\odot$ is an homomorphism between $\mathbb{O}$ and $\text{End}(\mathbb{M}_+)$.

Definition 2.13 (Vector space)**.** A *vector space* is a module $\mathbb{V} = (V, \mathbb{F}, +, \odot)$ such that the underlying scalar ring $\mathbb{F}$ is a field.

The most common vector space is the one of n-dimensional *column vectors* over a field $\mathbb{F} = (F, \oplus, \otimes)$ such that $\mathbb{F}^n = (F^n, \mathbb{F}, +, \odot)$, where $+$ is entry-wise field addition between column vectors and $\odot$ is element-wise field multiplication of scalars with column vectors. We will denote elements of a column vector space $\mathbb{V}$ by bold letters (e.g. $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}, \ldots$), and elements of the dual *row vector* space $\mathbb{V}^\intercal$ by superscripting a $\intercal$ symbol (e.g. $\boldsymbol{u}^\intercal, \boldsymbol{v}^\intercal, \boldsymbol{w}^\intercal, \ldots$). Finally, we denote the ith element of a column vector $\boldsymbol{v}$ by $\boldsymbol{v}_i$.

Definition 2.14 (Dot product)**.** Given a field $\mathbb{F} = (F, \oplus, \otimes)$ and an n-dimensional vector space $\mathbb{V} = (V, \mathbb{F}, +, \odot)$, the *dot product* operation is the map:

$$\boldsymbol{v} \cdot \boldsymbol{w} \colon \mathbb{V} \times \mathbb{V} \to \mathbb{F} = \bigoplus_{i=1}^{n} \boldsymbol{v}_i \otimes \boldsymbol{w}_i$$

Another important vector space is the one of $(n \times m)$-dimensional *matrices* over some base field $\mathbb{F}$: $\mathbb{F}^{n \times m} = ((F^n)^m, \mathbb{F}, +, \odot)$, where $+$ is element-wise field addition between matrices, and $\odot$ is element-wise field multiplication of scalars with matrices. We will denote elements of a matrix space $\mathbb{M}$ by bold capital letters (e.g. $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \ldots$), we denote the ith row of a matrix $\boldsymbol{M}$ by $\boldsymbol{M}_i$, and the jth element of the ith row with $\boldsymbol{M}_{i,j}$.

From now on, for vectors we will only deal with column vector space extensions of the kind $\mathbb{F}^n$ and row vector space extensions of the kind $(\mathbb{F}^m)^\intercal$ for some base field $\mathbb{F}$ and some $n, m \in \mathbb{N}$. Similarly, we will only deal with matrix space extensions of the kind $\mathbb{F}^{n \times m}$. Therefore, the ith column of a matrix will always be an element of $\mathbb{F}^n \cong \mathbb{F}^{n \times 1}$, and the jth row of a matrix will always be an element of $(\mathbb{F}^m)^\intercal \cong \mathbb{F}^{1 \times m}$.

Definition 2.15 (Transpose matrix)**.** The *transpose* of a matrix $\boldsymbol{M} \in \mathbb{F}^{n \times m}$ is the matrix:

$$\boldsymbol{M}^\intercal \mid \forall i \le n, \forall j \le m \colon \boldsymbol{M}^\intercal_{i,j} = \boldsymbol{M}_{j,i}$$

Therefore, given a matrix $\boldsymbol{M}$, we can denote the ith column by $\boldsymbol{M}^\intercal_i$.

Definition 2.16 (Matrix concatenation)**.** Given two matrices $\boldsymbol{A} \in \mathbb{F}^{n \times m_1}$ and $\boldsymbol{B} \in \mathbb{F}^{n \times m_2}$, their row-wise *concatenation* is the matrix $\boldsymbol{C} = \begin{pmatrix} \boldsymbol{A} & \boldsymbol{B} \end{pmatrix} \in \mathbb{F}^{n \times (m_1 + m_2)}$, such that:

$$\forall i \le n \colon (\forall j \le m_1 \colon \boldsymbol{C}_{i,j} = \boldsymbol{A}_{i,j}) \land (\forall j \le m_2 \colon \boldsymbol{C}_{i,j} = \boldsymbol{B}_{i,j})$$

And their column-wise concatenation is the matrix $\begin{pmatrix} \boldsymbol{A}; \boldsymbol{B} \end{pmatrix} = \begin{pmatrix} \boldsymbol{A}^\intercal & \boldsymbol{B}^\intercal \end{pmatrix}^\intercal$.

Definition 2.17 (Matrix multiplication)**.** *Matrix multiplication* over a base field $\mathbb{F}$ and some $m, n_1, n_2 \in \mathbb{N}$, is the map:

$$\boldsymbol{AB} \colon \mathbb{F}^{n_1 \times m} \times \mathbb{F}^{m \times n_2} \to \mathbb{F}^{n_1 \times n_2} \mid \forall i \le n_1, \forall j \le n_2 \colon (\boldsymbol{AB})_{i,j} = \boldsymbol{A}_i \cdot \boldsymbol{B}^\intercal_j$$

Definition 2.18 (Linear map)**.** A *linear map* is a homomorphism between two modules.

Definition 2.19 (k-linear map)**.** Given k vector spaces $\mathbb{V}_1, \ldots, \mathbb{V}_k, \mathbb{W}$ over the same scalar field $\mathbb{F}$, a *k-linear map* is a map $f \colon \mathbb{V}_1 \times \cdots \times \mathbb{V}_k \to \mathbb{W}$ such that, $\forall i \in \mathbb{N}$, every map obtained by fixing all but the ith argument is a linear map.

As we will see, bilinear (2-linear) maps are a fundamental component of modern ZK-SNARK systems.

2.1.4 Polynomials

The last fundamental object that we will need are polynomials and their relative algebras.

Definition 2.20 (Monovariate polynomial ring)**.** A *monovariate polynomial ring* over a field $\mathbb{F}$ is the triple $\mathbb{F}[x] = (F[x], +, \cdot)$ where $F[x]$ is the set of monovariate polynomials with coefficients over F in the indeterminate x, $+\colon F[x] \times F[x] \to F[x]$ is the *polynomial addition* operation and $\cdot\colon F[x] \times F[x] \to F[x]$ is the *polynomial multiplication* operation, such that all the properties of a ring are satisfied.

We will denote polynomials by lowercase letters (e.g. $p, q, r, \ldots$), and the degree of some polynomial p by $\deg(p)$. Given a field $\mathbb{F}$ and its corresponding polynomial ring $\mathbb{F}[x]$, we will denote by $\mathbb{F}[x]^n$ the n-dimensional module[2] of column vectors of polynomials over $\mathbb{F}$, with addition and scalar product defined in the standard way.

Given two vectors $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{F}^n$, we can define the *Lagrange interpolation* function [28]:

$$\operatorname{lag}(\boldsymbol{x}, \boldsymbol{y}) \colon \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}[x] = \sum_i \boldsymbol{y}_i \prod_{j \neq i} \frac{x - \boldsymbol{x}_i}{\boldsymbol{x}_i - \boldsymbol{x}_j}$$

we can build the unique polynomial of degree $n-1$ which, $\forall i \leq n$ assumes value $\boldsymbol{y}_i$ at point $\boldsymbol{x}_i$. We can extend the Lagrange interpolation function to a matrix space $\mathbb{F}^{n \times m}$ by applying lag separately to each row, as follows:

$$\operatorname{lag}(\boldsymbol{X}, \boldsymbol{Y}) \colon \mathbb{F}^{n \times m} \times \mathbb{F}^{n \times m} \to \mathbb{F}[x]^n = \begin{pmatrix} \operatorname{lag}(\boldsymbol{X}_1, \boldsymbol{Y}_1) & \cdots & \operatorname{lag}(\boldsymbol{X}_n, \boldsymbol{Y}_n) \end{pmatrix}$$

2.2 Arithmetic Programs

Suppose we have some algebra $\mathbb{A}$: we can represent and deal with finite sequences of operations, called *expressions*, between elements of $\mathbb{A}$ and/or variables over $\mathbb{A}$.

For example, given the expression $x^2 + x + 1$ over $\mathbb{R}[x]$, we might be interested to know what is the *evaluation* of the expression given some value for x. We will limit our analysis to fields (and rings).

Definition 2.21 (Arithmetic formula over a field)**.** Given a field $\mathbb{F}$, an *explicit arithmetic formula* over $\mathbb{F}$ is any expression φ of the kind:

$\varphi \equiv a$	with a constant over $\mathbb{F}$
$\varphi \equiv x$	with x variable over $\mathbb{F}$
$\varphi \equiv \varphi_1 \oplus \varphi_2$	with φ_1, φ_2 formulae over $\mathbb{F}$
$\varphi \equiv \varphi_1 \otimes \varphi_2$	with φ_1, φ_2 formulae over $\mathbb{F}$

[2]Since $\mathbb{F}[x]$ is a commutative ring, $\mathbb{F}[x]^n$ is not a vector space, but we will nevertheless call its elements 'vectors' for the sake of simplicity.

Additionally, an *implicit arithmetic formula* also allows expressions involving exponentiations:

$$\varphi \equiv \varphi_1^k \qquad\qquad \forall k \in \mathbb{N}, \text{ with } \varphi_1 \text{ formula over } A$$

It is always possible to translate an implicit formula into an equivalent explicit one by unrolling exponentiations. We denote the explicit version of an implcit formula φ with $\widehat{\varphi}$. Note also that multiplication by constants can also be unrolled to a sequence of additions (in fact, they can be viewed as exponentiations w.r.t. addition). From now on, we will only deal with arithmetic formulae over some field (or ring) $\mathbb{F}$, in which case implicit arithmetic expressions are equivalent to multi-variate polynomials.

Example 2.7. Consider the finite field $\mathbb{Z}_{13}$. For ease of notation, we will use $+$, juxtaposition and superscripting to denote, respectively, field addition, field multiplication, and exponentiation w.r.t. field multiplication. A possible implicit arithemtic formula over $\mathbb{Z}_{13}$ is the following expression:

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

Since in a finite field multiplication by a constant is simply repeated addition, i.e. $cx = +^c(x)$, the explicit version of φ then is:

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

2.2.1 Arithmetic circuits

It is possible to visually represent an arithmetic formula using a particualr kind of labeled *directed acyclic graph* (DAG), called the *arithmetic circuit.*

Definition 2.22 (Arithmetic circuit). An *arithmetic circuit* over a field $\mathbb{F}$ and a set of variables X over $\mathbb{F}$ is a triple $\mathcal{G} = (V, E, L)$ where V is the set of *vertices*, $E \subseteq V \times V$ is the set of *edges*, and $L\colon V \to A \cup X \cup (\{\oplus, \otimes\} \times \mathbb{N})$ is the vertex *labeling map*, such that, $\forall v \in V$:

$$\begin{aligned}
&L(v) \in A && \implies \nexists w \in V\colon (w, v) \in E && \text{(no in-edges for constant nodes)}\\
&L(v) \in X && \implies \nexists w \in V\colon (w, v) \in E && \text{(no in-edges for variable nodes)}\\
&\forall \odot_i \in L(v) && \implies |\{(w, v)\}_{w \in V} \cap E| = |\odot| && \text{(exactly } |\odot| \text{ in-edges for } \odot_i \text{ nodes)}
\end{aligned}$$

As an abuse of notation, we will sometimes identify a node v with its label $L(v)$. Given any explicit arithmetic formula φ over an algebra $\mathbb{A}$ and a set of variables X, we can build the corresponding arithmetic circuit $\mathcal{G} = (V, E, L)$ in the following way: for every distinct (i.e. ignoring repetitions) variable x appearing in φ, we add a vertex v with label $L(v) = x$; for every distinct constant c appearing in φ we add a vertex v with label $L(v) = c$; finally, for every occurence i of some operation $\odot$ in φ, we add a vertex v with label $L(v) = \odot_i$. We can partition V as follows:

- *Constant vertices*: $\mathcal{G}_{const} = \{v \mid L(v) \in \mathbb{F}\}$.
- *Variable vertices*: $\mathcal{G}_{var} = \{v \mid L(v) \in X\}$.

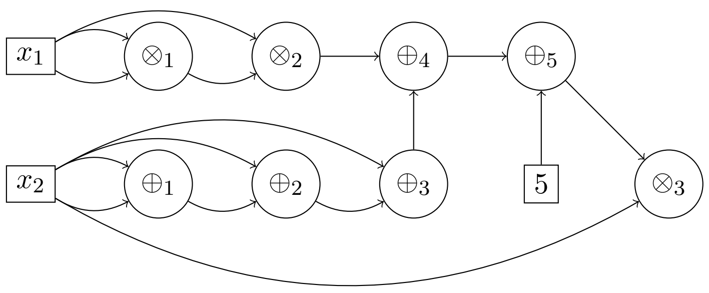

Figure 2.1: Circuit of the formula in Example 2.7. Rectangular nodes represent input vertices.

- *Addition vertices*: $\mathcal{G}_\oplus = \{v \mid \oplus \in L(v)\}$.
- *Multiplication vertices*: $\mathcal{G}_\otimes = \{v \mid \otimes \in L(v)\}$.
- *Operation vertices*: $\mathcal{G}_\odot = \mathcal{G}_\oplus \cup \mathcal{G}_\otimes$.
- *Input vertices*: $\mathcal{G}_{in} = \mathcal{G}_{const} \cup \mathcal{G}_{var}$.
- *Output vertices*: $\mathcal{G}_{out} = \{v \mid \nexists w \in V \colon (v, w) \in E\}$.
- *I/O vertices*: $\mathcal{G}_{IO} = \mathcal{G}_{in} \cup \mathcal{G}_{out}$.

To build the set of edges E, for every operation occuring in φ, we connect the vertices representing the operands to the vertex representing said operation, e.g. if we have the formula $(x \odot y) \odot z$ we add the edges $(x, \odot_1)$, $(y, \odot_1)$ and $(z, \odot_2)$. We also consider operation nodes as holding the intermediate values of the computation: in the previous example, we will also have the edge $(\odot_1, \odot_2)$, where $\odot_1$ represents the intermediate value $x \odot y$. The fact that we store all the intermediate values of a computation can be greatly exploited when optimizing the design of a circuit for some formula.

Since arithmetic circuits contain no cycles, they can only be used to represent a fixed number of operations (aka *bounded computations*). In general though, this is not really a big issue, as oftentimes we can easily synthesize circuits *on-the-fly*.

Example 2.8. Figure 2.1 shows the arithmetic circuit derived from the formula shown in Example 2.7. We can see the two variable vertices x_1 and x_2 which are also input vertices, the constant vertex 5, which is an input vertex too, the addition vertices $\oplus_1, \dots, \oplus_5$ and the multiplication vertices $\otimes_1, \otimes_2, \otimes_3$, of which the latter is also an output vertex.

Definition 2.23 (Circuit input)**.** A *circuit input* for an arithmetic circuit $\mathcal{G} = (V, E, L)$ over a field $\mathbb{F}$ is a triple $\mathcal{I}_\mathcal{G} = (V, E, L')$ such that, $\forall v \in \mathcal{G}_{in} \colon L'(v) \in \mathbb{F}$.

In particular, since every vertex in G_{in} has now a constant value, $\mathcal{I}$ induces an *evaluation* $\mathcal{E}$ over $\mathcal{G}$ which associates to every operation vertex $v \in \mathcal{G}_\odot$ the value $\text{eval}(v)$.

Definition 2.24 (Circuit assignment)**.** A *circuit assignment* for an arithmetic circuit $\mathcal{G} = (V, E, L)$ over a field $\mathbb{F}$ is a triple $\mathcal{A}_\mathcal{G} = (V, E, L')$ such that, $\forall v \in V \colon L'(v) \in \mathbb{F}$.

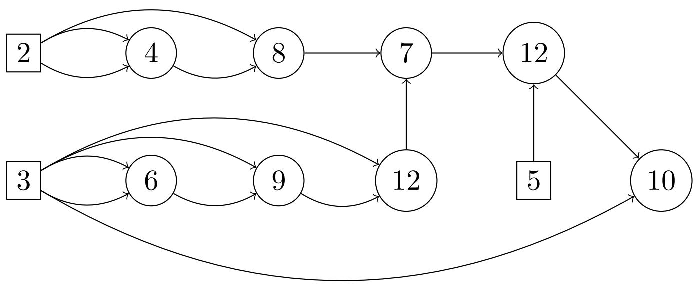

Figure 2.2: A valid assignment for the circuit in Figure 2.1. Remember that the underlying field is $\mathbb{Z}_{13}$.

An assignment $\mathcal{A}$ for a circuit $\mathcal{G}$ associates to every node of the circuit a value over the underlying field, but not necessarily a sensible one. However, as an assignment is also a circuit input, it induces an evaluation. Therefore, we say that $\mathcal{A}$ is *valid* if and only if $L'(v) = \text{eval}(v)$.

Example 2.9. Figure 2.2 shows a valid assignement for the circuit $\mathcal{G}$ shown in Example 2.8: we fix the inputs $x_1 = 2$ and $x_2 = 3$, and label every operation vertex v with the correct value $\text{eval}(v)$.

2.2.2 Rank-1 Constraint Systems

An arithmetic circuit tells us two things: (i) how to compute the intermediate values, after fixing the inputs, and (ii) how the intermediate values are constrained, depending on the inputs.

Definition 2.25 (Rank-1 Contraint System)**.** Given $m, n \in \mathbb{N}$, a m/n *Rank-1 Constraint System (R1CS)* over a field $\mathbb{F}$ is a triple $\mathcal{C} = (\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C})$ such that $\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C} \in \mathbb{F}^{m \times n}$.

Definition 2.26 (R1CS solution)**.** A *solution* to an R1CS $\mathcal{C}$ over a field $\mathbb{F}$ is a vector $\boldsymbol{s} \in \mathbb{F}^n$ such that $(\boldsymbol{A}\boldsymbol{s})(\boldsymbol{B}\boldsymbol{s}) = \boldsymbol{C}\boldsymbol{s}$.

Any explicit arithmetic circuit $\mathcal{G}$ with m multiplicative gates (i.e. $m = |\mathcal{G}_\otimes|$) and n input variables (i.e. $n = |\mathcal{G}_{var}|$) can be associated with a $m/(n + m + 1)$ R1CS $\mathcal{C}$ representing the constraints in the circuit, as follows:

1. Add a new 'constant variable' x_0 which always assumes value 1.
2. For every multiplicative gate $\otimes_i$ in the circuit, add a new *intermediate* variable t_i.
3. Define the column vector $\boldsymbol{x} = \begin{pmatrix} 1 & x_1 & \cdots & x_n & t_1 & \cdots & t_m \end{pmatrix}^\intercal$.
4. Express every multiplication gate $\otimes_i$ as an equation in the canonical form $(\boldsymbol{A}_i\boldsymbol{x})(\boldsymbol{B}_i\boldsymbol{x}) = \boldsymbol{C}_i\boldsymbol{x}$.

The intermediate variable t_m, which corresponds to the 'last' multiplication gate, is often denoted y as it represents the output of the circuit.

Example 2.10. Consider the circuit of Example 2.8: there are three multiplications in total, and since we have two input variables, the associated R1CS $\mathcal{C}$ will be a 3/6 R1CS. So, our variable vector will be:

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

Let's explicit all of the intermediate variables, and transform the constraint equations in canonical form:

$$\begin{aligned}
t_1 &= x_1x_1 &&\iff && (x_1)(x_1) = t_1\\
t_2 &= t_1x_1 + 4x_2 + 5 &&\iff && (t_1)(x_1) = 8 + 9x_2 + t_2\\
y &= t_2x_2 &&\iff && (x_2)(t_2) = y
\end{aligned}$$

We can now extract the $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ matrices:

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \qquad \boldsymbol{B} = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \qquad \boldsymbol{C} = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0\\ 8 & 0 & 9 & 0 & 1 & 0\\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Let's check the assignment that was given in Example 2.9, where $x_1 = 2$ and $x_2 = 3$:

$$\begin{aligned}
t_1 &= x_1x_1 = 2 \times 2 = 4 &&\equiv 4 \pmod{13}\\
t_2 &= t_1x_1 + 4x_2 + 5 = 4 \times 2 + 4 \times 3 + 5 = 25 &&\equiv 12 \pmod{13}\\
y &= t_2x_2 = 12 \times 3 = 36 &&\equiv 10 \pmod{13}
\end{aligned}$$

The solution vector $\boldsymbol{s}$ will be:

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

It is not hard to verify that indeed $(\boldsymbol{As})(\boldsymbol{Bs}) = \boldsymbol{Cs}$.

2.2.3 Quadratic Arithmetic Programs

The size of a solution for a R1CS grows linearly in the number of multiplication gates of the corresponding arithmetic circuit. This is an issue, as such solution is not *compact*: by compact, we mean that its size should be constant (i.e. independent of the circuit's size) or at most logarithmic (i.e. upper-bounded by a logarithmic function in the circuit's size).

Definition 2.27 (Quadratic Arithmetic Program)**.** Given some $m, n \in \mathbb{N}$, a m/n *Quadratic Arithmetic Program (QAP)* over $\mathbb{F}$ is a quadruple $\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ where $t \in \mathbb{F}[x]$ is the *target* polynomial, and $\boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y} \in \mathbb{F}[x]^m$ are, respectively, the *left input polynomial*, the *right input polynomial* and the *output polynomial*, such that:

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

Definition 2.28 (QAP solution)**.** A *solution* to a QAP $\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ over a field $\mathbb{F}$ is a pair of polynomials $h, p \in \mathbb{F}[x]$ such that $p = ht$.

It is possible to represent any m/n R1CS $\mathcal{C}$ with a m/n QAP $\mathcal{Q}$. First, we choose any arbitrary row vector $\boldsymbol{z} \in (\mathbb{F}^m)^{\intercal}$ such that $\forall i, j\colon \boldsymbol{z}_i \neq \boldsymbol{z}_j$; typically $\boldsymbol{z} = \begin{pmatrix} 1 & \cdots & m \end{pmatrix}$. Now, let $\boldsymbol{Z} \in \mathbb{F}^{n \times m}$ be a matrix such that $\forall i\colon \boldsymbol{Z}_i = \boldsymbol{z}$, then the resulting QAP $Q = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ will be:

$$t = \prod_i (x - \boldsymbol{z}_i) \qquad \boldsymbol{v} = \operatorname{lag}\Big(\boldsymbol{Z}, \boldsymbol{A}^\intercal\Big) \qquad \boldsymbol{w} = \operatorname{lag}\Big(\boldsymbol{Z}, \boldsymbol{B}^\intercal\Big) \qquad \boldsymbol{y} = \operatorname{lag}\Big(\boldsymbol{Z}, \boldsymbol{C}^\intercal\Big)$$

Then, given a solution $\boldsymbol{s}$ for $\mathcal{C}$, we can build a solution for $\mathcal{Q}$ by computing $p = (\boldsymbol{vs})(\boldsymbol{ws}) - \boldsymbol{ys}$, which, by construction, is divisible by t, and h will be their common factor. One might see that p is not really more compact than $\boldsymbol{s}$: as a matter of fact, given a m/n arithmetic circuit, $\boldsymbol{s}$ has size n, while p has degree (and therefore size) up to $2n$.

However, the fact that $p = ht$ implies that, for every $x \in \mathbb{F}$, $p(x) = h(x)t(x)$. If we are working over a big finite field $\mathbb{F}$ (say, $|\mathbb{F}| \approx 2^{256}$), it is very unlikely that, without knowing p, one is able to guess some y such that $y = p(x)$ (the probability is $1/|\mathbb{F}|$). This means that, if we wish to check that someone knows p with high confidence, we can just ask for a couple of values x, y and check whether $y = h(x)t(x)$. Note that we can exponentially increase our confidence by asking for more pairs.

Example 2.11. Consider the 3/6 R1CS $\mathcal{C}$ of Example 2.10: we want to compute the corresponding 3/6 QAP $\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$. First, we set:

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

Then, we compute the target polynomial t:

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

We can now compute the left and right input constraint polynomial vectors $\boldsymbol{v}$ and $\boldsymbol{w}$, and the output constraint polynomial vector $\boldsymbol{y}$. Notice how the 2nd, 4th and 5th columns of $\boldsymbol{A}$ form the canonical basis of $\mathbb{Z}_{13}^3$, and since lag is a linear function, we can express all other polynomials as linear combinations of $\operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_2^\intercal)$, $\operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_4^\intercal)$ and $\operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_5^\intercal)$:

$$\boldsymbol{v} = \operatorname{lag}\Big(\boldsymbol{Z}, \boldsymbol{A}^\intercal\Big) = \begin{pmatrix} \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_1^\intercal) \\ \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_2^\intercal) \\ \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_3^\intercal) \\ \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_4^\intercal) \\ \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_5^\intercal) \\ \operatorname{lag}(\boldsymbol{z}, \boldsymbol{A}_6^\intercal) \end{pmatrix} = \begin{pmatrix} 0 \\ 7x^2 + 4x + 3 \\ 0 \\ 12x^2 + 4x + 10 \\ 7x^2 + 5x + 1 \\ 0 \end{pmatrix}$$

$$\boldsymbol{w} = \begin{pmatrix} 0 \\ \boldsymbol{v}_2 + \boldsymbol{v}_4 \\ \boldsymbol{v}_5 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 6x^2 + 8x \\ 7x^2 + 5x + 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

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

Recall that a possible solution to the R1CS was $\boldsymbol{s} = \begin{pmatrix} 1 & 2 & 3 & 4 & 12 & 10 \end{pmatrix}^\mathsf{T}$. Let's check if it is also a valid assignment for the QAP:

$$\begin{aligned}
p &= (\boldsymbol{vs})(\boldsymbol{ws}) - \boldsymbol{ys} = 8x^4 + 5x^3 + 4x^2 + 2x + 7 \\
h &= \frac{p}{t} = \frac{8x^4 + 5x^3 + 4x^2 + 2x + 7}{x^3 + 7x^2 + 11x + 7} = 8x - 51 + \frac{273x^2 + 507x + 364}{x^3 + 7x^2 + 11x + 7} = 8x + 1 \\
ht &= (8x + 1)\left(x^3 + 7x^2 + 11x + 7\right) = 8x^4 + 5x^3 + 4x^2 + 2x + 7 = p
\end{aligned}$$

Since $ht = p$, clearly p is divisible by t, meaning that h and p are in fact a solution to the $\mathcal{Q}$.

3

Computational Background

A *computational model* (or model of computation) is any kind of system able to describe how to produce some *output* given some *input* [23]. Different models do this in different ways, each one with its own strength and weaknesses in terms of *expressivness*, *complexity* and *succintness*. Two historically important models of computations are Alonzo Church's *λ-calculus* [7] and Alan Turing's *Turing machine* (TM) [27]. Among several equivalent models [10], Turing machines became the standard model of computation.

Definition 3.1 (Turing machine [19])**.** A Turing machine is a quadruple $\mathcal{M} = (\Sigma, Q, q_0, \delta)$, where the *alphabet* Σ is a set of symbols such that $\sqcup \in \Sigma$, the *state set* Q is a set of symbols such that $\{\bot, \top\} \subseteq Q$, $q_0 \in Q$ is the *initial state*, and $\delta\colon (Q \setminus \{\bot, \top\}) \times \Sigma \to Q \times \Sigma \times \{\leftarrow, \rightarrow\}$ is the *transition function.*

By only requiring δ to be a relation instead of a function, we obtain the so-called *non-deterministic* Turing machine (NTM): given a state and an alphabet symbol, the machine can take different choiches at every step. A TM $\mathcal{M}$ manipulates a string $\bar{\sigma}$ over the alphabet $\Sigma \setminus \{\sqcup\}$ by placing it over an *infinite, discrete working tape* $\mathfrak{W}$, a total order isomorphic to $\mathbb{Z}$. The input string is positioned such that its first symbol is matched with the position 0 of the tape; all the positions before the first symbol and after the last symbol are filled with the *blank* symbol $\sqcup$. The computation $\mathcal{M}(\bar{\sigma})$ starts in the initial state q_0 with the *head* of the TM positioned over the position 0 of the tape, and proceeds according to the transition function: depending on the current state q and the symbol σ written in the current location of the head, it replaces σ with a new symbol σ', it moves the head to the left ($\leftarrow$) or to the right ($\rightarrow$) and it transitions into a new state q'. The computation *terminates* whenever one of the two *halting* states is reached: if $\mathcal{M}(\bar{\sigma}) = \bot$, then the input string $\bar{\sigma}$ is *rejected*, else if $\mathcal{M}(\bar{\sigma}) = \top$, then $\bar{\sigma}$ is *accepted.* It can also happen that the computation does not terminate: in such cases, we write $\mathcal{M}(\bar{\sigma}) = \uparrow$ and we say that the computation *hangs.*

3.1 Interactive Turing machines

In many scenarios, it is useful to extend Turing machines to include additional features, for example to represent the ability to access some source of randomness, or to communicate with an external environment to read inputs and produce outputs in an interactive manner.

Definition 3.2 (Input/Output Turing machine)**.** An input/output Turing machine is a quadruple $\mathcal{M} = (\Sigma, Q, q_0, \delta)$ where Σ, Q and q_0 are defined as in Definition 3.1, and:

$$\delta\colon (Q \setminus \{\bot, \top\}) \times \Sigma^2 \to Q \times \Sigma^2 \times \{\leftarrow, \rightarrow\}^2$$

The additional parameters in the transition function of an input/output Turing machine (I/O TM) account for two new tapes, namely the *read-only input tape* $\mathfrak{I}$ and the *write-only output tape* $\mathfrak{O}$: now, depending on the state q, the input symbol σ_i and the working symbol σ_w, the machine overwrites σ_w with a new symbol σ'_w and moves left/right on $\mathfrak{W}$, it writes a new symbol σ_o on $\mathfrak{O}$, where it can move only to the right, and it moves to the left/right on $\mathfrak{I}$. Additionally, in an I/O TM, the input string is placed on $\mathfrak{I}$ instead of $\mathfrak{W}$, which is instead blank at the beginning of the computation.

Definition 3.3 (Probabilistic Turing machine)**.** A probabilistic Turing machine is a quadruple $\mathcal{M} = (\Sigma, Q, q_0, \delta)$ where Σ, Q and q_0 are defined as in Definition 3.1, and:

$$\delta\colon (Q \setminus \{\bot, \top\}) \times \Sigma \times \{0, 1\} \to Q \times \Sigma \times \{\leftarrow, \rightarrow\}$$

In a probabilistic Turing machine (PTM), we have an additional *read-only random tape* $\mathfrak{P}$ which is populated with an infinite, uniformly random sequence of *coin tosses* (zeros and ones), that are used by the transition function to decide what to do. As for the writing tape of an I/O TM, the head on $\mathfrak{P}$ can only move to the right.

Definition 3.4 (Interactive Turing machine)**.** An interactive Turing machine is a quadruple $\mathcal{M} = (\Sigma, Q, q_0, \delta)$ where Σ, Q and q_0 are defined as in Definition 3.1, and:

$$\delta\colon (Q \setminus \{\bot, \top\}) \times \Sigma^2 \to Q \times \Sigma^2 \times \{\leftarrow, \rightarrow\}$$

An interactive Turing machine (ITM) is quite similar to an I/O TM, as it also has two additional tapes, called the *send tape* $\mathfrak{S}$ and the *receive tape* $\mathfrak{R}$. However, unlike for the input tape $\mathfrak{I}$ of an I/O TM, the head on $\mathfrak{R}$ cannot move backwards.

Remark 3.1. Our definition of ITM differs slightly from the standard one in the literature [12, 13], but we find it to be more modular. In any case, from now on, we will say *interactive Turing machine* to actually mean an *interactive, probabilistic, input/output Turing machine.*

Definition 3.5 (Interactive protocol)**.** An interactive protocol is a pair $\mathcal{I} = (\mathcal{M}, \mathcal{M}')$ where $\mathcal{M}$ and $\mathcal{M}'$ are interactive Turing machines such that $\mathfrak{I} = \mathfrak{I}'$, $\mathfrak{S} = \mathfrak{R}'$, $\mathfrak{R} = \mathfrak{S}'$, and their state sets Q, Q' contain the special *idle state* $q_{idle} \in Q, Q'$.

The computation of an interactive protocol (IP) over some string $\bar{\sigma}$, $\mathcal{I}(\bar{\sigma})$, proceeds in the following manner: initially, the tapes $\mathfrak{S}$, $\mathfrak{R}$, $\mathfrak{W}$, $\mathfrak{W}'$, $\mathfrak{O}$ and $\mathfrak{O}'$ are all empty (i.e. filled with blank symbols), the tapes $\mathfrak{P}$ and $\mathfrak{P}'$ are filled with random bits, and the tape $\mathfrak{I}$ contains $\bar{\sigma}$. The ITM $\mathcal{M}$ is said to be *active* and works normally until it transitions in the special state q_{idle}, becoming *inactive*. When this happens, control passes to $\mathcal{M}'$, which becomes active and works normally until it reaches its own idle state; control goes back to $\mathcal{M}$, and the process repeats. When one of the two machines halts, control

$$\begin{tikzcd}
 & & & & \mathfrak{S}=\mathfrak{R}' \arrow[ddrr] & & & & \\
 & \mathfrak{P} \arrow[dr] & & & & & & \mathfrak{P}' \arrow[dl] & \\
\mathfrak{O} & & \mathcal{M} \arrow[ll] \arrow[uurr] \arrow[dl, leftrightarrow] & & \mathfrak{I}=\mathfrak{I}' \arrow[ll] \arrow[rr] & & \mathcal{M}' \arrow[rr] \arrow[ddll] \arrow[dr, leftrightarrow] & & \mathfrak{O}' \\
 & \mathfrak{W} & & & & & & \mathfrak{W}' & \\
 & & & & \mathfrak{R}=\mathfrak{S}' \arrow[uull] & & & &
\end{tikzcd}$$

Figure 3.1: Visualization of an interactive protocol.

passes over the other one until it also halts. The protocol *succeeds* if both machines halt in the accepting state $\top$, and it *fails* if at least one of them halts in the rejecting state $\bot$. To denote the final states reached by one of the machines at the end of the computation, we write $\mathcal{I}_{\mathcal{M}}(\bar{\sigma})$ and $\mathcal{I}_{\mathcal{M}'}(\bar{\sigma})$ respectively. Figure 3.1 depicts the fundamental structure of an interactive protocol.

3.2 Problems and complexity

Historically, the most important class of problems that have been analyzed are so-called *decision problems*, i.e. probles whose solution is a binary *yes-or-no* answer [25]. This perfectly suits Turing machiens as we can interpret their acceptance or rejection of the input string respectively as a yes and a no answer.

Definition 3.6 (Kleene's closure)**.** The Kleene's closure of a set S is the set $S^* = \bigcup_{n\in\mathbb{N}} S^n$.

As Turing machine operate over strings in Σ^*, also called *words*, they partition Σ^* into three *languages* (a language is any set of strings): the language of accepted words, the languages of rejected words and the language of hanging words.

Definition 3.7 (Turing-recognizable language)**.** A language $L \subseteq \Sigma^*$ is recognized by some Turing machine $\mathcal{M}$ if $\forall w \in L\colon \mathcal{M}(w) = \top$.

Definition 3.8 (Turing-decidable language)**.** A language $L \subseteq \Sigma^*$ is decided by some Turing machine $\mathcal{M}$ if it is recognized by $\mathcal{M}$ and $\forall w \notin L\colon \mathcal{M}(w) = \bot$.

We denote the language recognized by a Turing machine $\mathcal{M}$ with $L(\mathcal{M})$. To solve an *instance* Π of some decision problem PROB, we first encode the instance into a string $\langle\Pi\rangle \in \Sigma^*$ such that $\langle\Pi\rangle \in L(\mathcal{M})$ if and only if the answer to Π is 'yes'.

The class of recognizable languages, called RE, strictly includes the class of decidable languages, called DEC [26]. But even decidable languages are not all equal: their *computational complexity*, that is, the amount of some resource which is required by a Turing machine to decide membership words in function of their length, can vary wildly. In general, we are only interested in the *asymptotic behaviour* of the machine.

Definition 3.9 (Big-O notation)**.** Given two functions $f, g\colon \mathbb{N} \to \mathbb{N}$, then $f = \mathcal{O}(g)$ if and only if $\exists c, n$ such that $\forall x \geq n\ f(x) \leq c \cdot g(x)$.

We write $f = \Omega(g)$ when $g = \mathcal{O}(f)$, and we write $f = \Theta(g)$ when $f = \mathcal{O}(g)$ and $g = \mathcal{O}(f)$. When there exists a TM $\mathcal{M}$ for which some complexity metric C is upper-bounded at most by a polynomial function in the length n of the input word (i.e. $\mathsf{C}(\mathcal{M}) = \mathcal{O}(n^c)$ for some constant $c \in \mathbb{N}$), we say that deciding the language is *feasible* w.r.t. C. On the other hand, if C is upper-bounded at least by an exponential function (i.e. $\mathsf{C}(\mathcal{M}) = \mathcal{O}(c^n)$ for some constant $c > 1$), we say that the problem is *infeasible* w.r.t. C. The standard complexity metrics are *time* T, that is the amount of transition steps a TM performs before halting, and *space* S, that is the amount of tape locations visited by a TM before halting[1].

Two of the most important *complexity classes*[2] are PTIME (P for short) and NPTIME (NP for short), which are the classes of languages decidable respectively by a deterministic TM and a nondeterministic TM using at most polynomial time. While we do not know if P $\overset{?}{=}$ NP, it is widely believed that P $\subset$ NP: for a deterministic Turing machine, deciding NP-COMPLETE problems (i.e. the hardest problems in NP) will generally take an exponential amount of time, and there is no known way in the physical world to build non-deterministic Turing machines. Although quantum computers have been shown to be able to crack problems which are believed to be infeasible for standard computers, like integer factorization [24], NP-COMPLETE problems seem to be out of reach also for such powerful machines.

3.3 Interactive proof systems

Even though NP-COMPLETE problems are infeasible, they are *efficient* to *verify*: given an *instance* Π of some NP-COMPLETE problem, and an additional *witness* string, we can build a deterministic TM that checks whether the witness *proves* or not that the problem admits a positive answer.

Example 3.1. Consider the problem SAT of deciding whether a propositional logic formula ϕ is satisfiable, which is the most famous NP-COMPLETE problem [8]. If we had a TM $\mathcal{M}$ with access to an *oracle* that, in $\mathcal{O}(1)$ time, provides a valid assignment for the variables in ϕ, it would be easy to verify that ϕ is indeed satisfiable. However, if the provided assignment was not valid, while $\mathcal{M}$ would reject it, there would be still no easy way to know whether ϕ is actually satisfiable or not!

Now, let's say we want to prove some theorem Π: computationally, this is equivalent to deciding whether Π is word which belongs to the language of the valid propositions over some formal system (say, the ZFC set theory [11]). A *proof* of the theorem plays the same role of the *witness* we discussed before: in general, verifying a proof for a theorem is (believed to be) much easier than finding the proof in the first place. Hence, we can extend the logical/mathematical concept of theorem to the more computational concept of language: for example, by NP theorem, we mean any language in NP.

[1]It is always the case that $\mathsf{S} \leq \mathsf{T}$.

[2]`https://complexityzoo.net/Complexity_Zoo`

Definition 3.10 (NP proof system [13]). A NP proof system for a language L is an interactive protocol $\mathcal{I} = (\mathcal{P}, \mathcal{V})$, where $\mathcal{P}$ is the *prover* and $\mathcal{V}$ is the *verifier*, such that:

$$\begin{aligned} &\forall w \in L \colon \mathcal{I}_{\mathcal{V}}(w) = \top && (\textit{correctness}) \\ &\forall w \notin L \colon \mathcal{I}_{\mathcal{V}}(w) = \bot && (\textit{completeness}) \\ &\exists k \in \mathbb{N} \colon \mathsf{T}(\mathcal{V}) = \mathcal{O}\Big(|x|^k\Big) && (\textit{boundness}) \end{aligned}$$

In an NP proof system (NPPS), the common input tape of $\mathcal{P}$ and $\mathcal{V}$ contains some word w representing some statement: in typical scenario, the statement is provided by the prover himself, who wants to convince the verifier of the truthness of such statement. During the protocol, $\mathcal{P}$ and $\mathcal{V}$ exchange messages through their communication tapes; at some point, $\mathcal{P}$ sends to $\mathcal{V}$ a candidate proof π: the verifier checks the proof and, if the proof is valid, it is always convinced of its validity (correctness), hence it will accept. On the other hand, if the proof happens to be wrong (e.g. if $\mathcal{P}$ is trying to deceive $\mathcal{V}$), then the verifier will never be convinced by such a proof, and it will reject. The polynomial bound on the execution time of $\mathcal{V}$ is necessary to force cooperation, and avoid the case where $\mathcal{V}$ simply ignores $\mathcal{P}$ and computes the proof by itself.

Definition 3.11 (Interactive proof system [13]). An interactive proof system for a language L is an interactive protocol $\mathcal{I} = (\mathcal{P}, \mathcal{V})$ such that, for any arbitrarily small $\epsilon \in \mathbb{R}_+$:

$$\begin{aligned} &\forall w \in L \colon \Pr(\mathcal{I}_{\mathcal{V}}(w) = \bot) < |w|^{\epsilon} && (\textit{probabilistic correctness}) \\ &\forall w \notin L \colon \Pr(\mathcal{I}_{\mathcal{V}}(w) = \top) < |w|^{\epsilon} && (\textit{probabilistic completeness}) \\ &\exists k \in \mathbb{N} \colon \mathsf{T}(\mathcal{V}) = \mathcal{O}\Big(|x|^k\Big) && (\textit{boundness}) \end{aligned}$$

Clearly, the main difference between a NP proof system and an interactive proof system (IPS) lays in the probabilistic behaviour of the latter: there is a (negligible) probability that a verifier will be convinced of the truthness of a false statement, or of the falseness of a true statement, although the latter is not relevant in most instances, as the prover *wants* the verifier to accept. In fact, the proofs generated by the prover in an IPS are often called *arguments of knowledge*, to denote their probabilistic nature.

Definition 3.12 (IP complexity class). The Interactive Polynomial class IP is the class of languages which are decided by some interactive proof system.

Of course NP $\subseteq$ IP, as the verifier can check, with no error, a proof for any NP-COMPLETE problem statement in polynomial time. Howvever, there are problems in IP which are not known to be in NP, such as the problems of deciding the order and the membership of a matrix group [4], and deciding whether two graphs are not isomorphic [12].

3.3.1 Arthur-Merlin games

In an interactive proof system, the coin tosses of the two parties are *private*, and they can exchange any kind of message.

Definition 3.13 (Arthur-Merlin games [4])**.** An Arthur-Merlin game is an interactive proof system $\mathcal{I} = (M, A)$ such that Arthur can only send coin tosses over $\mathfrak{S}$.

In Arthur-Merlin games, the messages that the verifier (Arthur) can send to the prover (Merlin) are limited to the coin tosses from its random tape (in the sense that, whatever Arthur sends Merlin, he interprets it as a coin toss).

Definition 3.14 (AM complexity class)**.** The Arthur-Merlin complexity class AM is the class of languages which are decided by an Arthur-Merlin game.

While it is obvious that AM $\subseteq$ IP, the converse was also shown to be true [4]: in particular, for any fixed number k of communications between Arthur and Merlin, it is possible to build an equivalent game involving only 2 communications.

3.3.2 Zero-Knowledge Interactive Protocols

Suppose that two parties are executing an IPS for some NP-COMPLETE problem: the instance will be avilable on the input tape to both parties, then the prover will solve the problem and it will send the witness/proof to the verifier, which will in turn check the proof and decide whether to accept or not. In this process, the verifier gained more knowledge than just the solvability of the problem: it will also know a solution!

In fact, in real systems, the leaked information will be of much greater relevance. The prover is not really more powerful nor efficient than the verifier, as the computational bound on the latter is purely a formal need. What really gives power to the prover is the knowledge of some secret, stored on its working tape, which is often the witness itself, from which the input instance was built 'in reverse'. In revealing the witness to the verifier, the prover would lose the advantage that it had over the verifier.

Definition 3.15 (Computational indistinguishability)**.**

4

Cryptographical Background

The main application of Zero Knowledge proof systems has been, arguably unsurprisingly, in the cryptography field. The possibility of two or more parties to cooperate and exchange information one with another in a zero-knowledge manner is the fundamental idea behind many branches of cryptography such as *Multi Party Computation* (MPC) [29] and *Fully Homomorphic Encryption* (FHE) [3].

The main application of Zero knowledge protocols has been in *blockchain* infrastructures, with the cryptocurrency *ZCash* being the most prominent example [6]. In a public blockchain, a user (the prover) wants to convince the other users (the verifiers) that he posseses some data: to this end, he exhibits a *commitment*, typically a short message which can be easily computed when knowing the original data, but for which it is hard to find a *collision*. In this scenario, the prover would like to be able to convince the verifiers of the validity of its commitment, without having to hand them out the original data.

4.1 Secure Hash functions

Secure Hash functions are a fundamental tool of cryptography, as they can be used to produce $\mathcal{O}(1)$ or $\mathcal{O}(\log(n))$ commitments for any message of length $|n|$.

Definition 4.1 (Hash function)**.** Given some $n \in \mathbb{N}$, an *n-bit hash function* is a function $H \colon \{0,1\}^* \to \{0,1\}^n$.

The input of an hash function is called the *message*, while its output is called the *digest.* From the definition, it is immediate to see that there are an infinite number of messages which map to the same digest. Now, a very simple hash function might be truncation (i.e. take the first n bits and discard anything coming afterwards), but it is not of much interest for cryptography, as we require additional *security guarantees.*

Remark 4.1. When we say that it is *easy* (resp. *hard*) to compute a function, we mean that there is (resp. there is not) a probabilistic Turing machine which can compute such function in at most polynomial time. In particular, a *one-way function* is an easily computable function f whose inverse is hard to compute.

Definition 4.2 (Cryptographic hash function [1])**.** Given $n \in \mathbb{N}$, an *n-bit cryptographic hash function (CHF)* is a n-bit hash function 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 d, it is hard to find a message m such that $H(m) = d$ (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, a perfect n-bit CHF provides $n/2$ bits of security for collision resistance, meaning that even an optimal adversary needs to guess $2^{n/2}$ messages before finidng a collision, while for preimage resistance it would probie n bits of security. 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) [17]. We will introduce the Davies-Meyer and the Merkle-Damgård constructions respectively, as those are the ones of interest to us.

Theorem 4.1 (Davies-Meyer construction [21]). *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 4.2 (Merkle-Damgård construction [18]). *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.

4.2 Verification Trees

4.2.1 Merkle Trees

4.2.2 Augmented Binary Trees

4.3 ZK-SNARK systems

4.3.1 Pinocchio

4.3.2 Groth16

4.3.3 ZK-STARK

4.3.4 PLONK

ZK - efficiency

II
Zero Knowledge friendly permutations

5

~~State of the art~~

6

The Generalized Dynamic Triangular System

7
Arion and ArionHash

III

Appendix

A
Titolo della prima appendice

Sed purus libero, vestibulum ut nibh vitae, mollis ultricies augue. Pellentesque velit libero, tempor sed pulvinar non, fermentum eu leo. Duis posuere eleifend nulla eget sagittis. Nam laoreet accumsan rutrum. Interdum et malesuada fames ac ante ipsum primis in faucibus. Curabitur eget libero quis leo porttitor vehicula eget nec odio. Proin euismod interdum ligula non ultricies. Maecenas sit amet accumsan sapien.

Bibliography

[1] Saif Al-Kuwari, James H. Davenport, and Russell J. Bradford. Cryptographic hash functions: Recent design trends and security notions. Cryptology ePrint Archive, Paper 2011/565, 2011. `https://eprint.iacr.org/2011/565`.

[2] 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.

[3] Frederik Armknecht, Colin Boyd, Christopher Carr, Kristian Gjøsteen, Angela Jäschke, Christian A. Reuter, and Martin Strand. A guide to fully homomorphic encryption. Cryptology ePrint Archive, Paper 2015/1192, 2015. `https://eprint.iacr.org/2015/1192`.

[4] L. Babai and E. Szemeredi. On the complexity of matrix group problems i. In *25th Annual Symposium onFoundations of Computer Science, 1984.*, pages 229–240, 1984.

[5] 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. `https://eprint.iacr.org/2018/046`.

[6] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[7] Alonzo Church. The calculi of lambda conversion.(am-6), volume 6. In *The Calculi of Lambda Conversion.(AM-6), Volume 6*. Princeton University Press, 1941.

[8] 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.

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

[10] Martin Davis. *The undecidable: Basic papers on undecidable propositions, unsolvable problems and computable functions*. Courier Corporation, 2004.

[11] Abraham Adolf Fraenkel, Yehoshua Bar-Hillel, and Azriel Levy. *Foundations of set theory*. Elsevier, 1973.

[12] Oded Goldreich, Silvio Micali, and Avi Wigderson. Proofs that yield nothing but their validity or all languages in np have zero-knowledge proof systems. *J. ACM*, 38(3):690–728, jul 1991.

[13] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *SIAM Journal on Computing*, 18(1):186–208, 1989.

[14] Lorenzo Grassi, Yongling Hao, Christian Rechberger, Markus Schofnegger, Roman Walch, and Qingju Wang. A new feistel approach meets fluid-spn: Griffin for zero-knowledge applications. *IACR Cryptol. ePrint Arch.*, 2022:403, 2022.

[15] Lorenzo Grassi, Dmitry Khovratovich, Christian Rechberger, Arnab Roy, and Markus Schofnegger. Poseidon: A new hash function for zero-knowledge proof systems. In *USENIX Security Symposium*, 2021.

[16] Jens Groth. On the size of pairing-based non-interactive arguments. Cryptology ePrint Archive, Paper 2016/260, 2016. `https://eprint.iacr.org/2016/260`.

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

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

[19] C.H. Papadimitriou. *Computational Complexity.* Theoretical computer science. Addison-Wesley, 1994.

[20] 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`.

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

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

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

[24] 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.

[25] Michael Sipser. *Introduction to the Theory of Computation.* Course Technology, Boston, MA, third edition, 2013.

[26] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[27] A. M. Turing. Computing machinery and intelligence. *Mind*, 59(236):433–460, 1950.

[28] Edward Waring. Vii. problems concerning interpolations. *Philosophical Transactions of the Royal Society of London*, 69:59–67, 1779.

[29] Andrew C. Yao. Protocols for secure computations. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 160–164, 1982.