ALPEN-ADRIA UNIVERSITÄT
KLAGENFURT | WIEN GRAZ

UNIVERSITÀ DEGLI STUDI DI UDINE
hic sunt futura

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/

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) [41]. 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 [60] or Groth16 [45], 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 [11].

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 [56] 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 [63] or SHA [29], 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, 43, 42] 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 `Arion` blockcipher and the `ArionHash` 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 this thesis

This work is organized as follows: Chapter 2 presents the mathematical background for both current ZK-SNARK systems and our hash function. Chapter 3 presents the computational background of ZK-SNARK systems, and their origin. Chapter 4 presents the cryptographical background of ZK-SNARK systems, their latest evolutions and their most important real-world applications. In Chapter 5, we review the history and the state of the art concerning ZK-friendly hash functions; we present the GTDS framework and its instantiation in the form of the `Arion` block cipher and the `ArionHash` hash function; and we study the implementation and performance of the latter by comparing it to the state of the art. Finally, in Chapter 6, we draw our conclusion and explore future directions of the work.

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

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 $\mathbb{A} = (A, \odot_1, \ldots, \odot_n)$, $\mathbb{A}' = (A', \odot'_1, \ldots, \odot'_n)$ 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:

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

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:

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

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 $\mathrm{e}_\oplus \neq \mathrm{e}_\otimes$ and $\mathbb{F}_\otimes \setminus \{\mathrm{e}_\oplus\}$ 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 $Z_q = \{0, \dots, q-1\}$, 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 $\mathbb{H}_\mathbb{G} = \mathrm{End}(\mathbb{G}) = \mathrm{End}(\mathbb{G})_+ \cup \mathrm{End}(\mathbb{G})_\circ$ is the *endomorphism ring* of $\mathbb{G}$: $\mathrm{End}(\mathbb{G})_+$ is an abelian group, $\mathrm{End}(\mathbb{G})_\circ$ 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 $\mathrm{End}(\mathbb{M}_+)$.

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 $\mathbb{V}^\mathsf{T}$ by superscripting a T symbol (e.g. $\boldsymbol{u}^\mathsf{T}, \boldsymbol{v}^\mathsf{T}, \boldsymbol{w}^\mathsf{T}, \ldots$). 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 $(\mathbb{F}^m)^\mathsf{T}$ 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 $(\mathbb{F}^m)^\mathsf{T} \cong \mathbb{F}^{1 \times m}$.

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

$$\boldsymbol{M}^\mathsf{T} \mid \forall i \leq n, \forall j \leq m \colon \boldsymbol{M}^\mathsf{T}_{i,j} = \boldsymbol{M}_{j,i}$$

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

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:

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

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

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:

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

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 [73]:

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

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:

$$\mathrm{lag}(\boldsymbol{X}, \boldsymbol{Y})\colon \mathbb{F}^{n \times m} \times \mathbb{F}^{n \times m} \to \mathbb{F}[x]^n = \Big(\mathrm{lag}(\boldsymbol{X}_1, \boldsymbol{Y}_1) \quad \cdots \quad \mathrm{lag}(\boldsymbol{X}_n, \boldsymbol{Y}_n)\Big)$$

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, $\forall v \in V$:

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

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*: $\mathcal{G}_{var} = \{v \mid L(v) \in X\}$.

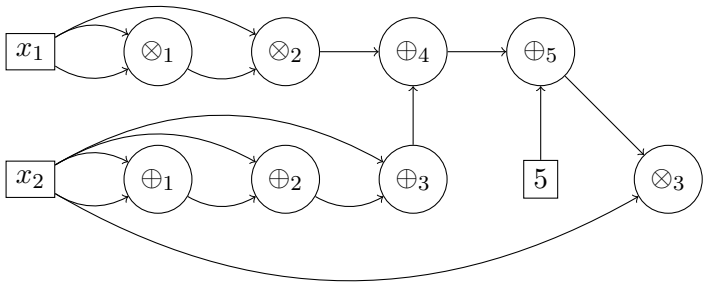

Figure 2.1: Circuit of the formula in Example 2.7. Rectangular nodes represent input vertices.

- *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*: $\mathcal{G}_{out} = \{v \mid \nexists w \in V : (v, w) \in E\}$.
- *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 $\oplus_1, \ldots, \oplus_5$ 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, $\forall v \in \mathcal{G}_{in} : L'(v) \in \mathbb{F}$.

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, $\forall v \in V : L'(v) \in \mathbb{F}$.

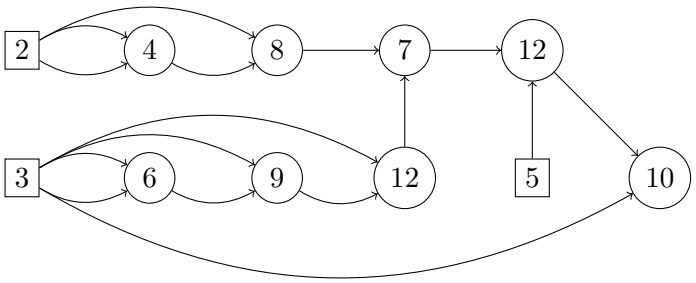

Figure 2.2: A valid assignment for the circuit in Figure 2.1. Remember that the underlying field is $\mathbb{Z}_{13}$.

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 [25])**.** 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:

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

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 $x_2 = 3$:

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

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

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 [37])**.** 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:

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

Then, given a solution $\boldsymbol{s}$ for $\mathcal{C}$, we can build a solution for $\mathcal{Q}$ by computing $p = (\boldsymbol{v}\boldsymbol{s})(\boldsymbol{w}\boldsymbol{s}) - \boldsymbol{y}\boldsymbol{s}$, 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:

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

Then, we compute the target polynomial t:

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

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 $\mathrm{lag}(\boldsymbol{z}, \boldsymbol{A}_2^\intercal)$, $\mathrm{lag}(\boldsymbol{z}, \boldsymbol{A}_4^\intercal)$ and $\mathrm{lag}(\boldsymbol{z}, \boldsymbol{A}_5^\intercal)$:

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

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

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

Recall that a possible solution to the R1CS was $\boldsymbol{s} = \begin{pmatrix} 1 & 2 & 3 & 4 & 12 & 10 \end{pmatrix}^\intercal$. Let's check if it is also a valid assignment for the QAP:

$$\begin{aligned} p &= (\boldsymbol{v}\boldsymbol{s})(\boldsymbol{w}\boldsymbol{s}) - \boldsymbol{y}\boldsymbol{s} = 8x^4 + 5x^3 + 4x^2 + 2x + 7 \\ h &= \frac{p}{t} = \frac{8x^4 + 5x^3 + 4x^2 + 2x + 7}{x^3 + 7x^2 + 11x + 7} = 8x - 51 + \frac{273x^2 + 507x + 364}{x^3 + 7x^2 + 11x + 7} = 8x + 1 \\ ht &= (8x + 1)\Big(x^3 + 7x^2 + 11x + 7\Big) = 8x^4 + 5x^3 + 4x^2 + 2x + 7 = p \end{aligned}$$

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* [64]. 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* [23] and Alan Turing's *Turing machine* (TM) [72]. Among several equivalent models [30], Turing machines became the standard model of computation.

Definition 3.1 (Turing machine [59])**.** 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 [39, 41], 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

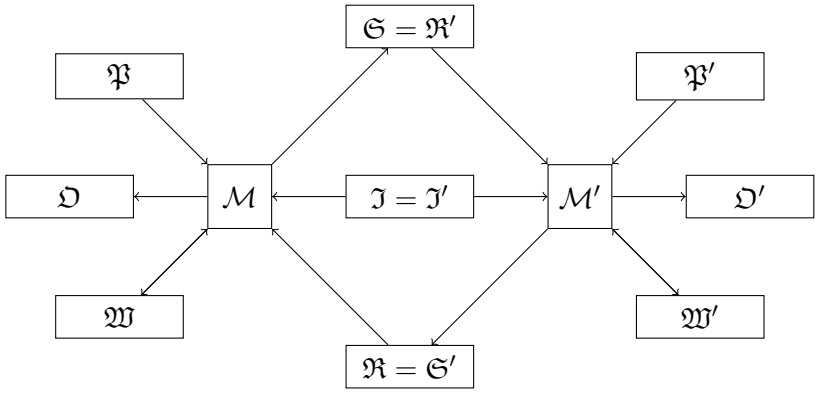

Figure 3.1: Visualization of an interactive protocol.

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 [67]. 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 [71]. 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* TIME, that is the amount of transition steps a TM performs before halting, and *space* SPACE, 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 $\mathrm{P} \stackrel{?}{=} \mathrm{NP}$, it is widely believed that $\mathrm{P} \subset \mathrm{NP}$: 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 [66], 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 [24]. 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 [36]). 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 $\mathsf{SPACE} \leq \mathsf{TIME}$.

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

Definition 3.10 (NP proof system [41])**.** An 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:

$$\forall w \in L\colon \mathcal{I}_{\mathcal{V}}(w) = \top \qquad (\textit{completeness})$$
$$\forall w \notin L\colon \mathcal{I}_{\mathcal{V}}(w) = \bot \qquad (\textit{soundness})$$
$$\exists k \in \mathbb{N}\colon \mathsf{TIME}(\mathcal{V}) = \mathcal{O}\Big(|x|^k\Big) \qquad (\textit{boundness})$$

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, 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 [41])**.** An *interactive proof system* (IPS) for a language L is an interactive protocol $\mathcal{I} = (\mathcal{P}, \mathcal{V})$ such that, for any arbitrarily big bound $c \in \mathbb{R}_+$:

$$\forall w \in L\colon \Pr(\mathcal{I}_{\mathcal{V}}(w) = \bot) < |w|^{-c} \qquad (\textit{statistical completeness})$$
$$\forall w \notin L\colon \Pr(\mathcal{I}_{\mathcal{V}}(w) = \top) < |w|^{-c} \qquad (\textit{statistical soundness})$$
$$\exists k \in \mathbb{N}\colon \mathsf{TIME}(\mathcal{V}) = \mathcal{O}\Big(|x|^k\Big) \qquad (\textit{verifier boundness})$$

The difference between a NP proof system and an IPS lies in the probabilistic nature 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.

Definition 3.12 (Interactive argument of knowledge system [22])**.** An *interactive arguments of knowledge system* (IARK) is an interactive proof system $\mathcal{I} = (\mathcal{P}, \mathcal{V})$ such that:

$$\exists k \in \mathbb{N}\colon \mathsf{TIME}(\mathcal{P}) = \mathcal{O}\Big(|x|^k\Big) \qquad (\textit{prover boundness})$$

In the IARK setup, the two parties are modelled in a more 'realistic' way: a prover is not intrinsically more efficient than a verifier, what really gives it power is the knowledge of some *secret*, initially stored on its working tape, which allows the prover to quickly find a proof/witness (in fact, the secret is often the witness itself).

Definition 3.13 (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 [5], and deciding whether two graphs are not isomorphic [39].

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 3.14 (Arthur-Merlin games [5])**.** 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 3.15 (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 [5]: in particular, for any *fixed* number k of communications between Arthur and Merlin, AM$[k]$ = AM[2], meaning that it is possible to build an equivalent Arthur-Merlin game using only 2 communications. In fact, it was also shown that IP (and hence AM) is equal to PSPACE, i.e. the class of problems solvable by a Turing machine in polynomial space [65].

3.4 Zero-Knowledge Protocols

Suppose that two parties are executing an IARK system for some hard problem: the instance is places on the shared input tape, and also suppose that the secret in possession of the prover is simply a witness for the instance. All the prover has to do is send the witness to the verifier, which will in turn check it and decide whether to accept or not. In this process, the verifier gained more knowledge than just the solvability of the problem: it also learned a solution, and not just any solution, but exactly the one available to the prover, which should have been a secret. To address this issue, researchers started exploring the field of so-called zero-knowledge proofs [41, 39].

Informally, two random variables U and V that map words of some language $L \subseteq \{0,1\}^*$ to words of $\{0,1\}^*$ are *perfectly indistinguishable* when no unbounded Turing machine is able to tell them apart, are *statistically indistinguishable* when no PSPACE Turing machine is able to tell them apart, and are *computationally indistinguishable* when no PTIME Turing machine $\mathcal{M}$ is able to tell them apart. By 'telling apart', we mean that the distribution of the words that are accepted/rejected by Turing machines respecting the imposed bounds is independent from U and V: intuitively, this means that U and V are interchangable with each other and using one over the other does not give an 'edge' to $\mathcal{M}$ [40, 41, 75].

Example 3.2. Consider the two random variables $U, V : L \to \{0,1\}^*$ for some $L \subseteq \{0,1\}^*$, such that,

for all words $x \in L$ and all words $w \in \{0,1\}^{|x|}$, it holds that:

$$\Pr(U(x) = w) = 2^{-|x|} \qquad \Pr(V(x) = w) = \begin{cases} 0 & x = 0 \dots 0 \\ 2^{-|x|+1} & x = 1 \dots 1 \\ 2^{-|x|} & \text{otherwise} \end{cases}$$

U and V have *almost* the same distribution, with the $1 \dots 1$ string happening twice as often in V. For increasingly longer strings, no Turing machine can tell the two distributions apart by collecting a polynomial amount of samples, since $\sum_w |\Pr(U(x) = w) - \Pr(V(x) = w)| = 2^{-|x|+1}$, hence U and V are statistically indistinguishable.

Definition 3.16 (Tape view)**.** A *tape view* is a random variable $\mathrm{V}_{\mathcal{M}}$ that models the concatenation of all the contents that are read/written by a halting Turing machine $\mathcal{M}$ over its tapes.

For a deterministic, non-probabilistic Turing machine, the tape view variable is quite pointless, but it is a useful tool to model the behaviour of machines that exploit randomness, and expecially for interactive protocols. For example, if we have a Turing machine with one tape $\mathfrak{T}$, then:

$$\Pr(\mathrm{V}_{\mathcal{M}}(x) = w) = \Pr(\mathrm{V}_{\mathcal{M},\mathfrak{T}}(x) = w) = \Pr(\mathcal{M}(x) = w)$$

Definition 3.17 (Approximability)**.** A random variable U is (perfectly, statistically, computationally) *approximable* by a probabilistic Turing machine $\mathcal{M}$ over some language L if U and $\mathrm{V}_{\mathcal{M}}$ are (perfectly, statistically, computationally) indistinguishable.

Note that for a random variable U and a halting PTM $\mathcal{M}$ to be perfectly indistinguishable over some language L, it must be the case that $\forall x \in L \colon \mathcal{M}(x) = \mathrm{V}_{\mathcal{M}}(x) = \mathcal{U}(x)$.

Definition 3.18 (Zero-knowledge interactive protocol)**.** A (perfectly, statistically, computationally) *Zero-knowledge interactive protocol* (ZKIP) over a language $L \subseteq \{0,1\}^*$ is an interactive protocol $\mathcal{I} = (\mathcal{M}, \mathcal{M}')$ such that, for every $\mathcal{M}'$, $\mathrm{V}_{\mathcal{M}'}$ is (perfectly, statistically, computationally) approximable by a Turing machine $\mathcal{M}''$ over the language $L' = \{(x,h) \mid x \in L \wedge w \in \{0,1\}^*\}$, where the string h represents the initial content of $\mathfrak{W}'$.

Naturally, a ZKIP which is also a proof system is a zero-knowledge proof system (ZKPS); similarly, if it is an interactive argument of knowledge system then it is a zero-knowledge interactive argument of knowledge system (ZK-IARK). From now on, by zero-knowledge we mean computational zero-knowledge, as assuming a polynomial-time bounded adversary is an acceptable restriction in the real world. The initial string h of a ZKIP can be interpreted as the *history* of previous interactions with the prover, or some eavesdropped information from the interactions that the prover had with other verifiers.

A proof system being zero-knowledge basically means that, even for curious or malicious verifiers, and even with additional knowledge on the behaviour of the prover, what can be computed is nothing more than what could have been computed in polynomial time, hence within the imposed computational power limits, without communicating with the prover. While it is obvious that every problem solvable

in probabilistic polynomial time (PP) has a zero-knowledge proof system (the prover does nothing and the verifier computes the solution by himself), it was proven that also all problems in NP have a ZKPS [39]. By assuming the existance of secure probabilistic encryption, it was finally shown that also all Arthur-Merlin games, and hence all problems in IP, have a ZKPS [10].

3.4.1 Non interactive Zero-Knowledge

In many scenarios, especially ones involving multiple parties, interaction can be a problem as the communication cost of bidirectional n-to-n grows quadratically. Such cases are in fact of great interest for zero-knowledge systems: multiple parties can be both provers and/or verifiers, and their number might be huge.

For this reason, researches explored the possibility of having zero knowledge *non-interactive* proof systems (ZK-NPS) or argument of knowledge systems (ZK-NARK). Unfortunately, only the languages in BPP, that is languages decidable in probabilistic polynomial time with a bounded error allow for zero-knowledge non-interactive proofs [58, 38]. Such languages are of course trivial, as the verifier has enough power to do all the computation by itself without the need of the prover.

However, by introducing an initial *preprocessing* phase, it is possible to regain the lost power [31], and the most prominent technique to achieve non-interaction is the *Common Reference String* (CRS) model [18] (sometimes also called *common random string* model). The main idea of the CRS model is that, before engaging in the protocol, the prover and the verifier have both obtained access to a shared string of random bits. In the simplest case, the string is generated by a *trusted third party*, although in practice this is oftentimes not a viable solution as the whole point of zero-knowledge is having to deal with untrusted parties. To circumnent this problem, it is possible to generate the CRS by a *majority vote* between n authorities, which can be untrusted if picked singularly, but are assumed to be honest in their majority [47]. In fact, it was shown that it is possible, without losing zero-knowledge, to re-use multiple times a single CRS both by a single [17] or multiple [34] provers, although only for arguments of knowledge and not for proofs.

The first zero-knowledge systems, both interactive and non-interactive, were tailor-made for specific problems, such as the quadratic residuosity problem QR [41], the hamiltonian path problem HAM-PATH [50], or the 3-SAT problem [17]. Although for any NP-COMPLETE problem PROB there is a polynomial-time algorithm [48] that converts every instance of PROB to an instance of, say, 3-SAT, such reductions are often not trivial to devise and very expensive to apply. For this reason, researchers started devising constructions to prove arbitrary NP statements embedded in the form of boolean circuits [27], which can neatly represent the computation of a Turing machine over any NP-COMPLETE problem [24], and therefore remove the need to go through polynomial-time reductions.

The first of such systems [27] required a CRS of size cubic in the length of the statement to be proven, although XOR and NOT gates didn't need to consume any bits from the CRS. In the following years, many improvements were proposed, reducing the complexity of the constructions from cubic to subquadratic [21] and eventually linear [26].

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) [74] and *Fully Homomorphic Encryption* (FHE) [4].

The main application of Zero knowledge protocols has been in *blockchain* infrastructures, with the cryptocurrency *ZCash* being the most prominent example [12]. 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, and a *trapdoor function* is a one-way function whose inverse becomes easy to compute given some additional knowledge, called the *key*.

Remark 4.2. The $\oplus$ symbol used in this chapter can have different meanings: in the literature, it is mostly used to denote bitwise XOR. Since a message/string over $\{0,1\}^*$ can be considered as a vector over $\mathbb{Z}_2^n$, bitwise XOR is the same as vector addition. However, there are instances, particularly in ZK systems, where messages are interpreted as elements of some prime field $\mathbb{Z}_p$: in such cases, $\oplus$ will denote addition modulo p, as defined in Chapter 2. However, such cases will be explicitly specified, so we decided to keep the ambiguity to be coherent with the literature.

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

Note that second preimage resistance implies first preimage resistance (if given any m_1 we can find a colliding m_2, we can do it also without being given m_1), and in turn preimage resistance implies second preimage resistance (if given any d we can find a colliding m, then given any m' we can compute d and find the collision).

A perfect n-bit CHF provides $n/2$ bits of security for collision resistance (birthday paradox), meaning that every possible adversary is expected to need at least $\mathcal{O}\left(2^{n/2}\right)$ time to find a collision, while for preimage resistance it provides n bits of security. A CHF can be built by applying some known secure construction to functions which are simpler to devise.

Definition 4.3 (Padding function)**.** An n-bit *padding function* is a function $\mathrm{Pad}\colon \{0,1\}^* \to (\{0,1\}^n)^*$.

Definition 4.4 (Pseudorandom permutation)**.** Given $l \in \mathbb{N}$, an l-bit *pseudorandom permutation* is a permutation $P\colon \{0,1\}^l \to \{0,1\}^l$ which is indistinguishable from an uniform random distribution.

Definition 4.5 (Keyed permutation)**.** Given $l, n \in \mathbb{N}$, an *l/n-bit keyed permutation* is a function $F\colon \{0,1\}^l \times \{0,1\}^n \to \{0,1\}^l$ which is a permutation on its first parameter.

Definition 4.6 (Block cipher)**.** A *block cipher* is a trapdoor pseudorandom keyed permutation.

Definition 4.7 (One-way compression function)**.** Given $l_1, n, l_2 \in \mathbb{N}$ such that $l_1 + n > l_2$, an *$l_1/n/l_2$-bit one-way compression function (OWCF)* is a one-way function $F\colon \{0,1\}^{l_1} \times \{0,1\}^n \to \{0,1\}^{l_2}$.

Since many compression functions exploit their second component (the key), rather than the first component to compress a message, we will call n-to-m(-bit) compression function any compression function which, in any way, reduces an input of length n to an output of length m.

A *pseudorandom keyed permutation* (PKP) is typically built by iterating a "somewhat pseudorandom" keyed permutation F for an adequate number of *rounds*, until inverting the function becomes hard. A block cipher can be built from a PKP by following some secure construction scheme, like the Feistel-Luby-Rackoff construction [54]. A OWCF can also be derived from a PKP or a block cipher, by

applying a secure construction scheme, like the Davies-Meyer construction [62]. Finally, CHF can again be derived either by a OWCF, for exaple through the Merkle-Damgård construction [56], or directly from a pseudorandom permutation, like in the sponge construction [15, 70].

Proposition 4.1 (Feistel-Luby-Rackoff construction [54]). *Given an l/n-bit pseudorandom keyed permutation P, some message $m = (m_1, m_2)$ such that $m_1, m_2 \in \{0,1\}^l$, a number of rounds $r > 3$, and a set of keys $k_1, \ldots, k_r \in \{0,1\}^n$, then the function E_r is a $2l/n$ block cipher, where:*

$$E_i(m, k_i) = (x_i, y_i) = \begin{cases} (m_1, m_2) & i = 0 \\ (y_{i-1}, x_{i-1} \oplus P(y_{i-1}, k_i)) & 1 \leq i \leq r \end{cases}$$

Proposition 4.2 (Davies-Meyer construction [62]). *Given an l/n-bit pseudorandom keyed permutation P, some number of blocks $b \in \mathbb{N}$, some* initial value *$v \in \{0,1\}^l$, and some message $m = (m_1, \ldots, m_b)$ such that each $m_i \in \{0,1\}^n$, then the function F_b is an $l/bn/l$ one-way compression function, where:*

$$F_i(v, m) = \begin{cases} v & i = 0 \\ P(F_{i-1}(v, m), m_i) \oplus F_{i-1}(v, m) & 1 \leq i \leq b \end{cases}$$

Proposition 4.3 (Merkle-Damgård construction [56]). *Given an $l/n/l$-bit one way compression function F, some initial value $v \in \{0,1\}^l$, some message $m \in \{0,1\}^*$, a padding extension length k and an n-bit padding function* Pad *such that, $\forall x, y \in \{0,1\}^*$:*

$$\begin{aligned} &|\mathrm{Pad}(x)| = |x| + (-|x| \bmod n) + kn \\ &|x| = |y| \implies |\mathrm{Pad}(x)| = |\mathrm{Pad}(y)| \\ &|x| \neq |y| \implies \mathrm{Pad}(x)_{|\mathrm{Pad}(x)|} \neq \mathrm{Pad}(y)_{|\mathrm{Pad}(y)|} \end{aligned}$$

then the function $H_{|\mathrm{Pad}(m)|}$ is an l-bit cryptographic hash function, where:

$$H_i(m) = \begin{cases} v & i = 0 \\ F(H_{i-1}(m), \mathrm{Pad}(m)_i) & 1 \leq i \leq |\mathrm{Pad}(m)| \end{cases}$$

Proposition 4.4 (Sponge construction [15]). *Given an l-bit pseudorandom permutation P, a message $m \in \{0,1\}^*$, a* rate *r and a* capacity *c such that $r + c = l$, an initial value $v \in \{0,1\}^b$ and an r-bit padding function* Pad*, the function $H_{|\mathrm{Pad}(m)|}$ is an $r-$bit cryptographic hash function, where:*

$$\begin{aligned} S_i(m) &= \begin{cases} v & i = 0 \\ P(S_{i-1}(m) \oplus \mathrm{Pad}(m)_i) & 1 \leq i \leq |\mathrm{Pad}(m)| \end{cases} \\ H_i(m) &= S_i(m)_{1,\ldots,r} \end{aligned}$$

The sponge construction is particularly interesting as it is very flexible: given a pseudorandom permutation, it can be used to build pseudorandom keyed permutations, cryptographic hash functions, random number generators and authenticated encryptions [16].

4.2 Tree-like modes of hashing

Consider an n-bit CHF H, and suppose that a prover claims to know some message m: the digest $d = H(m)$ can be considered as a *short binding commitment* for m: By asking the prover to share the digest, whose size $|d| = n$ is typically considered to be $\mathcal{O}(1)$ (or $\mathcal{O}(\log(|m|))$ in some cases), a verifier is convinced that the prover does know m with probability $\approx 1 - 1/2^n$. A modern standard CHF like SHA-256 [29] produces digests of length at least 256 bits, making the $1 - 1/2^n$ bound really hard to bruteforce through. Note that the verifier needs not to know m in advance: the commitment d is (temporarily) appended to a public *blockchain* and, at any point in the future, when the verifier becomes aware of some m' provided by the prover, if $H(m') = d$, the commitment can be approved or rejected.

Now, suppose that the prover wants to commit to a list of k messages: the simplest solution would be to publish the hash of every message, which would require to append $\mathcal{O}(k)$ elements on the blockchain. Another way would be for the prover to share $H((m_1, \ldots, m_k))$: the communication cost would only be $\mathcal{O}(1)$ but, in general, not all the messages belong to the same prover, so this method would not work, and we need a better solution.

4.2.1 Merkle tree

Definition 4.8 (Binary Merkle tree [55])**.** A *binary Merkle tree (MT)* of height $h \in \mathbb{N}$ over a $2n$-to-n compression function C, is the complete binary tree of height h such that, given a sequence of input messages $(m_1, \ldots, m_{2^{h-1}})$ over $\{0,1\}^{2n}$, produces an output digest $d \in \{0,1\}^n$ in the following way:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{h-1}}$ contain $C(m_1), \ldots, C(m_{2^{h-1}})$.
2. Every other node ν contains $C(\nu_l, \nu_r)$, where ν_l is the left child of ν and ν_r is the right child of ν.
3. The output digest d is the content of the root node.

The set of the sibling nodes visited in the path from a leaf of the tree to the root, including the leaf itself, is the *authentication path* of the leaf. By using Merkle trees, the prover only needs to send to the verifier, as a commitment for some message m_i among $n = 2^h$ messages, the contents of the co-path from the leaf containing m_i to the root, in addition plus the hash of m_i: this requires just $\mathcal{O}(\log(n))$ cost to validate the commitment. Merkle trees bottom-up construction is very easy to parallelize, and they can be used in the multiple-provers scenario: each prover only needs to commit to the path from its own leaf to the root of the tree. It is immediate to generalize the notion of binary Merkle tree to arbitrary arity.

Proposition 4.5 (Security of Merkle tree mode of hash [55])**.** *Given a one-way tn-to-n compression function C, the t-ary Merkle tree over C is a cryptographic hash function.*

Example 4.1. Consider the sequence of pre-hashed messages $S = (3, 4, 7, 7)$ and the compression function $C(x, y) : (x, y) \mapsto (xy \bmod 13) + 1$ (for ease of exposition, we work over integers instead of bit strings, but the two can be readily converted into one another). Figure 4.1 shows the contents of the associated Merkle Tree. Note that the real message is not stored in the Merkle Tree, but only the 'first level' of hashes. The authentication path of the leaf labelled with 3 consists of the tuple $(3, 4, 11)$: by computing $H(3, 4) = 13$ and then $H(13, 11) = 1$ we can verify that the commitment is respected.

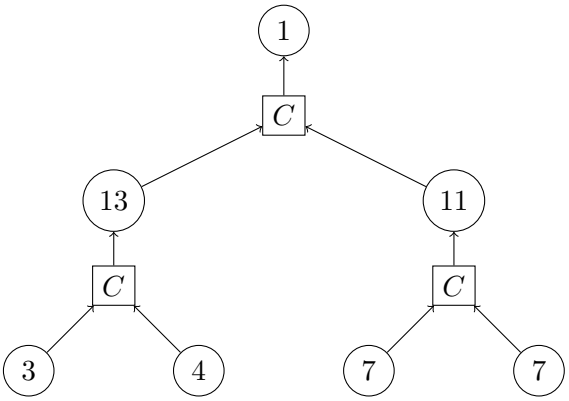

Figure 4.1: Merkle tree of Example 4.1.

4.2.2 Augmented Binary Tree

The Merkle tree is the de-facto standard for blockchain applications, and basically for any scenario for which a 'linear' hash function cannot be used. In [68], it was given a lower bound on the amount of queries necessary to obtain a collision for a $(m+s)$-to-s-bit CHF H (the m is variable) built from a $(n+c)$-to-n-bit OWCF C: if H makes r queries to C, it is possible to find a collision by making $2^{\frac{nr+cr-m}{r+1}}$ queries to H. By combining this result with the $2^{s/2}$ upper bound of the birthday paradox, one can immediately obtain a tight bound $m = \frac{2nr+2cr-sr-s}{2}$ for the variable length m of the message.

Definition 4.9 (Compactness [3])**.** The *compactness* of an $(m+s)$-to-s-bit hash function making r queries to an underlying $(n+c)$-to-n-bit one-way compression function is the value $\alpha = \frac{2m}{2nr+2cr-sr-s}$.

Example 4.2. Consider a $2n$-to-n bit OWCF and a Merkle Tree of height h: the computation of the tree is a $\left(2^{h-1}n\right)$-to-n-bit hash function, and makes exactly $r = 2^{h-1}-1$ queries to C. We have $s = c = n$ and $m = 2^{h-1}n - n = nr$, therefore the compactness of the Merkle Tree construction is:

$$\alpha = \frac{2m}{2nr+2cr-sr-s} = \frac{2nr}{2nr+2nr-nr-n} = \frac{2r}{3r-1}$$

Which tends to $2/3$ when r tends to infinity.

Definition 4.10 (Augmented Binary tRee [3])**.** An *Augmented Binary tRee (ABR)* of height $h \in \mathbb{N}$ over a $2n$-to$n$ compression function C is a complete binary tree of height h augmented with *middle* nodes such that, given a sequence of input messages $S = (m_1, \ldots, m_{2^{h-1}+2^{h-2}-1} \mid \forall i\colon m_i \in \{0,1\}^*)$, it produces an output digest $d \in \{0,1\}^n$ in the following way:

1. The leaf nodes $\nu_1, \ldots, \nu_{2^{h-1}}$ contain $C(m_1), \ldots, C(m_{2^{h-1}})$.
2. There are no middle nodes in the leaf layer.
3. The middle nodes $\nu_{2^{h-1}+1}, \ldots, \nu_{|S|}$ contain $C(m_{2^{h-1}+1}), \ldots, C\left(m_{|S|}\right)$.
4. Every other node ν contains $C(\nu_l \oplus \nu_m, \nu_r \oplus \nu_m) \oplus \nu_r$, 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.

The authentication path of the ABR is similar to the one of the Merkle tree, but also includes the middle nodes encountered during the traversal.

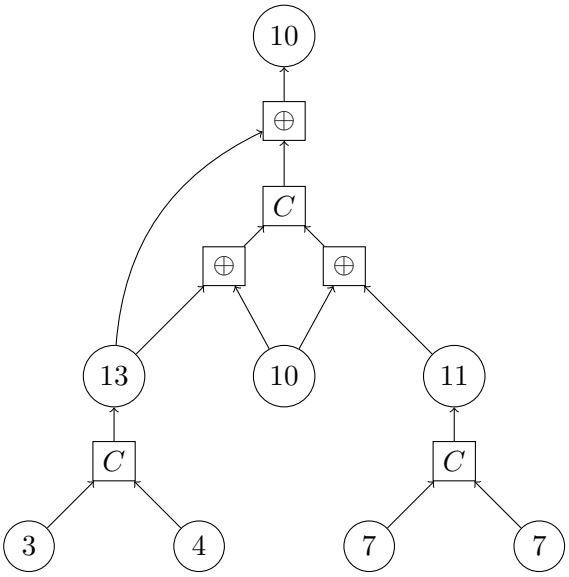

Figure 4.2: ABR of Example 4.4.

Proposition 4.6 (Security of ABR mode of hash [3])**.** *Given a one-way* $2n$*-to-*n *compression function* C*, the ABR over* C *is a cryptographic hash function.*

An ABR of height h can process 50% more messages than a Merkle Tree of the same height, while performing the same number of queries to the underlying compression function, with the additional cost introduced by the intermediate $\oplus$ operations being negligible in most scenarios.

Example 4.3. Consider a $2n$-to-n bit OWCF and an ABR of height h: the computation of the tree is a $\left(2^{h-1}+2^{h-2}-1\right)n$-to-$n$-bit hash function, and makes exactly $r = 2^{h-1}-1$ queries to C. Like in Example 4.2, we have $s = c = n$, but this time $m = \left(2^{h-1}+2^{h-2}-1\right)n - n = nr + nr/2 - n$, so the compactness of the ABR construction is:

$$\alpha = \frac{2m}{2nr + 2cr - sr - s} = \frac{2nr + nr - 2n}{2nr + 2nr - nr - n} = \frac{3r-2}{3r-1}$$

Which approaches 1 as r approaches infinity, meaning that the ABR construction achieves optimal compactness.

It is worth of notice that, while the ABR hash mode achieves collision resistance, it does not achieve *indifferentiability* (a weaker notion of indistinguishability between Turing machines [53]), hence a modified construction, called ABR+, was also proposed, although it does not achieve perfect compactness.

Example 4.4. Consider the same compression function C of Example 4.1, and consider the sequence of pre-hashed messages $S' = (3, 4, 7, 7, 10)$, in this case we interpret $x \oplus y \equiv (x + y \bmod 13) + 1$. Figure 4.2 shows the resulting ABR. The authentication path of the node labelled with 3 consists of the tuple $(3, 4, 10, 11)$: by computing $H(3, 4) = 13$ and then $H(13 \oplus 10, 11 \oplus 10) \oplus 13 = 10$ we can verify that the commitment is respected.

4.3 ZK-SNARK systems

As we saw in Section 3.4.1, researchers were able to construct ZK-NARK systems whose verification complexity was linear in the size of the problem instance, which is provided as a boolean circuit. Furthermore, in the CRS model, by using a block cipher, it is also possible to have *publicly verifiable* constructions [50], meaning that *any* verifier, not just the one that engages the protocol, is able to check the proof, which is encrypted with a *proving key*, by using a public *verification key*.

Proposition 4.7 (Fiat-Shamir heuristic [35])**.** *Suppose a probabilistic I/O TM $\mathcal{P}$ with access to a CHF H wants to prove its knowledge of the discrete logarithm $x = \log(y)$ for some value $y \in \mathbb{Z}_p$, where p is a large prime number. Then $\mathcal{P}$ can sample a random value v from $\mathfrak{P}$, compute the digest $d = H(p, y, p^v)$, the result $r = v - dx \bmod (p-1)$, and finally output the quadruple (p, y, p^v, r). Any* PTIME *TM $\mathcal{V}$ with access to (p, y, p^v, r) and H can recompute d and check whether $p^v = p^r y^d$ (If $\mathcal{P}$ is not cheating, then $p^r y^d = p^{v-dx}(p^x)^d = p^{v-dx}p^{dx} = p^{v-dx+dx} = p^v$). Assuming that the discrete logarithm is hard and that true CHF exist, if equality holds $\mathcal{V}$ is convinced that $\mathcal{P}$ knows x but is not able to retrieve it except with negligible probability.*

Definition 4.11 (Succint proof)**.** A *succint proof* for a statement σ over a language $L \subseteq \{0,1\}^*$ is a proof π such that $|\pi| = \mathcal{O}(\log(|\sigma|))$.

Similarly, one can define the notion of succint argument of knowledge, and in particular, a succint ZK-NARK system is called a ZK-SNARK system.

Definition 4.12 (Probabilistically checkable proof system [6, 33])**.** A *probabilistically checkable proof system* (PCP system) is an interactive proof system $(\mathcal{P}, \mathcal{V})$ such that for any proof π provided by $\mathcal{P}$: $\exists k\colon \mathsf{TIME}(\mathcal{V}) = \mathcal{O}\Big(\log^k(|\pi|)\Big)$.

In a PCP system, the prover $\mathcal{P}$ constructs a proof π of size polynomial in the length of the original statement σ; since the verifier $\mathcal{V}$ is polylogarithmically bound to the size of the proof, it can only query a small portion of it, however, it is enough to get statistical completeness and soundness.

In [49], the author uses Merkle trees to have the prover commit to a proof π, (the bits of π are the leaves and the root, whcih has constant size, is sent to the verifier). The verifier then queries a certain number of authentication paths, which have length $\mathcal{O}(\log(|\pi|))$, and decides whether to accept or reject. In this sense, the protocol is therefore succint. In [57], the construction was extended and, by applying the Fiat-Shamir heuristic, it is possible to make the protocol non-interactive.

One of the first *succint* ZK-NARK (ZK-SNARK) systems that didn't make explicit use of PCPs was devised in [44], but had one important drawback: while the size of the proof is constant, the size of the CRS, and the computation that the prover has to perform is *quadratic* in the size of the input circuit (this bound was slightly improved in [51]).

However, by first transforming the circuit into *quadratic span programs* (QSPs), the boolean equivalent of QAPs (Section 2.2.3), it was shown that it is possible to reduce both the size of the CRS and the prover's computational complexity to linear, while still having succint proofs [37]. Since all these constructions make use of encryption based on the hardness of finding the discrete logarithm of a number over a big finite field, dealing with boolean circuits and QSPs is not very efficient; although

both polynomially sized boolean and arithmetic circuits are equivalent to PTIME Turing machines [61], working over arithmetic programs, and hence using QAPs over QSPs, can greatly reduce the constant factors involved in such constructions, although this depends on the kind of input problem (numeric problems can exploit arithmetic circuits much better than, say, propositional problems).

4.3.1 Pinocchio

An important application of ZK-SNARK systems is in *verifiable computation*. Consider a client (say, a mobile phone) that wants to delegate to a server (say, a cloud provider) some computation, for which several inputs are required: some are provided by the client, and some by the server:

- The client does not trust the server, so we would need a proof system, but since the server is not computationally unbounded, an *argument of knowledge* system will suffice.
- The server might have to interact with many clients or, similarly, many different clients might require the same computation, the system must be *non-interactive*.
- Verifying the computation must be cheaper than performing it, otherwise the client wouldn't have to ask the server in the first place, the system must provide *succint* proofs.
- The server has too interests in to the client that the computation was correct, say to avoid legal liability, but it is not willing to share its own inputs, so our system must be *zero-knowledge*.

Clearly, among the various systems we saw up to now, ZK-SNARKs are the only one that can reasonably fulfill all of the requirements above. However, all the constructions we saw, due to the high overheads involved in generating the CRS, building the QSP/QAP, generating the proof, and even verifying it, were (much) slower than native execution by the client.

The first construction that was efficient enough to be usable in practice, and that in many cases broke the 'native execution' barrier for verification time, was *Pinocchio* [60].

Definition 4.13 (Bilinear map [19])**.** A *bilinear map* (w.r.t. exponentiation) over two groups $\mathbb{G}$ and $\mathbb{G}'$ is a map $B\colon \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ such that $\forall a, b \in \mathbb{Z}\colon B\Big(x^a, y^b\Big) = B(x, y)^{ab}$.

Example 4.5. Recall Example 2.6. Consider a group $\mathbb{G} = \langle g \rangle$, and define $B\colon \mathbb{G} \times \mathbb{G} \to \mathbb{Z}_{|\mathbb{G}|}$ such that $B(x, y) = g^{\log(x)\log(y)}$. The map B is bilinear w.r.t. exponentiation, since:

$$\forall a, b \in \mathbb{Z}\colon B\Big(x^a, y^b\Big) = g^{\log(x^a)\log\left(y^b\right)} = g^{a\log(x)b\log(y)} = \Big(g^{\log(x)\log(y)}\Big)^{ab} = B(x, y)^{ab}$$

The two main ingredients of Pinocchio are QAPs and bilinear maps. Given in input an arithmetic formula φ over some field $\mathbb{F} \cong \langle g \rangle$ equipped with a bilinear map $B\colon \langle g \rangle \times \langle g \rangle \to \mathbb{F}$ defined as in Example 4.5, the Pinocchio protocol is organized in three phases: the *setup phase*, the *prover phase*, and the *verifier phase*.

Setup phase

In the setup phase, any of the parties derives the explicit formula $\widehat{\varphi}$, the circuit $\mathcal{G}$ of $\widehat{\varphi}$, the R1CS $\mathcal{C}$ of $\mathcal{G}$ and finally the m/n QAP $\mathcal{Q} = (t, \boldsymbol{v}, \boldsymbol{w}, \boldsymbol{y})$ of $\mathcal{C}$. Recall that $m = |\boldsymbol{v}| = |\boldsymbol{w}| = |\boldsymbol{y}| = |\mathcal{G}_\otimes|$[1], that $n = \deg(t) = m+1+|\mathcal{G}_{in}|$, and that $\forall i\colon\ \deg(\boldsymbol{v}_i) = \deg(\boldsymbol{w}_i) = \deg(\boldsymbol{y}_i) = n-1$. The QAP (and eventually all the other componenets) are made public.

Either via a trusted third party or an ensamble of authorities [47], we generate a CRS $\bar{\sigma}$, which has length $|\bar{\sigma}| = 8\lceil\log(|\mathbb{F}|)\rceil$ and interpret every $\lceil\log(|\mathbb{F}|)\rceil$-bit subsequence as elements of $\mathbb{F}$:

$$\bar{\sigma} = (r_v, r_w, s, \alpha_v, \alpha_w, \alpha_y, \beta, \gamma)$$

Finally, after fixing $r_y = r_v r_w$, we build the *proving key* and the *verification key*:

$$\mathsf{K}_\mathcal{P} = \begin{pmatrix} \left\{g^{r_v \boldsymbol{v}_j(s)}\right\}, & \left\{g^{r_w \boldsymbol{w}_j(s)}\right\}, & \left\{g^{r_y \boldsymbol{y}_j(s)}\right\}, \\ \left\{g^{r_v \alpha_v \boldsymbol{v}_j(s)}\right\}, & \left\{g^{r_w \alpha_w \boldsymbol{w}_j(s)}\right\}, & \left\{g^{r_y \alpha_y \boldsymbol{y}_j(s)}\right\}, \\ \left\{g^{s^k}\right\}, & \left\{g^{r_v \beta \boldsymbol{v}_j(s)} g^{r_w \beta \boldsymbol{w}_j(s)} g^{r_y \beta \boldsymbol{y}_j(s)}\right\} & \end{pmatrix}$$

$$\mathsf{K}_\mathcal{V} = \left(g, g^{\alpha_v}, g^{\alpha_w}, g^{\alpha_y}, g^{\gamma}, g^{\beta\gamma}, g^{r_y t(s)}, \left\{g^{r_v \boldsymbol{v}_i(s)}\right\}, \left\{g^{r_w \boldsymbol{w}_i(s)}\right\}, \left\{g^{r_y \boldsymbol{y}_i(s)}\right\}\right)$$

where i ranges over $\{0, \ldots, |G_{in}|\}$, j ranges over $\{|\mathcal{G}_{in}|, \ldots, n-1\}$, and k ranges over $\{1, \ldots, n\}$.

It is important to remark that $\bar{\sigma}$ must be deleted immediately after generating the two keys, as it can be exploited to tamper the protocol (in jargon, we say that it is *toxic waste*).

Prover phase

After fixing the input vector $\boldsymbol{x}$, the prover creates a circuit input for $\mathcal{G}$, computes the induced evaluation and extracts the assignment $\mathcal{A}$. From $\mathcal{A}$, the prover derives the associated solution $\boldsymbol{c}$ of $\mathcal{C}$, and computes the polynomials $p = (\boldsymbol{vc})(\boldsymbol{wc}) - (\boldsymbol{yc})$ and $h = p/t$. Now, to build a short proof π, the prover should sum the contributions of all the left input, right input, and output polynomials:

$$v_\otimes = \sum_{i=|\mathcal{G}_{in}|}^{n-1} \boldsymbol{c}_i \boldsymbol{v}_i(s) \qquad w_\otimes = \sum_{i=|\mathcal{G}_{in}|}^{n-1} \boldsymbol{c}_i \boldsymbol{w}_i(s) \qquad y_\otimes = \sum_{i=|\mathcal{G}_{in}|}^{n-1} \boldsymbol{c}_i \boldsymbol{y}_i(s)$$

and obtain the proof:

$$\pi = \left(g^{r_v v_\otimes}, g^{r_w w_\otimes}, g^{r_y y_\otimes}, g^{h(s)}, g^{r_v \alpha_v v_\otimes}, g^{r_w \alpha_w w_\otimes}, g^{r_y \alpha_y y_\otimes}, g^{r_v \beta v_\otimes} g^{r_w \beta w_\otimes} g^{r_y \beta y_\otimes}\right)$$

Although r_v, r_w or r_y have been deleted, and finding the discrete logarithm is hard, by using $\mathsf{K}_\mathcal{P}$, it is possible to compute:

$$g^{r_v v_\otimes} = \prod_{i=|\mathcal{G}_{in}|}^{n-1} \left(g^{r_v v_i(s)}\right)^{\boldsymbol{c}_i} \qquad g^{r_w w_\otimes} = \prod_{i=|\mathcal{G}_{in}|}^{n-1} \left(g^{r_w w_i(s)}\right)^{\boldsymbol{c}_i} \qquad g^{r_y y_\otimes} = \prod_{i=|\mathcal{G}_{in}|}^{n-1} \left(g^{r_y y_i(s)}\right)^{\boldsymbol{c}_i} \tag{4.1}$$

[1] As we will see, it is possible to reduce this number in some special cases by exploiting the structure of R1CS constraints. Also note that the original protocol as in [60] does not make explicit use of R1CS, but we want to stress its importance as it is the standard way to represent arithmetic circuits in `libsnark` [14].

and similarly for the other values (except for $g^{h(s)}$ which can be computed directly). Finally, the prover publishes the pair $(\phi(\boldsymbol{x}), \pi)$.

Verifier phase

At this point, any potential verifier with access to $\mathsf{K}_\mathcal{V}$ and to the bilinear map B can check the alleged proof. First, the verifier should check whether $p = ht$, which, as we noted in Section 2.2.3, it is statistically equivalent to checking whether $p(s) = h(s)t(s)$. In the spirit of Equation (4.1), using $\mathsf{K}_\mathcal{V}$, the verifier can compute the input/output contributions $g^{r_v v_{I/O}}$, $g^{r_w w_{I/O}}$ and $g^{r_y y_{I/O}}$. Clearly, for any arbitrary $k \in \mathbb{F} \setminus \{0\}$, we have that $p(s) = h(s)t(s) \iff k^{p(s)} = k^{h(s)t(s)}$ (we "work in the exponent"). If we fix $k = B(g^{r_v}, g^{r_w})$, with some algebraic effort the previous equation can be transformed into the equivalent one:

$$B(g^{r_v v_{I/O}} g^{r_v v_\otimes}, g^{r_w w_{I/O}} g^{r_w w_\otimes}) = B\Big(g^{r_y t(s)}, g^{h(s)}\Big) B(g^{r_y y_{I/O}} g^{r_y y_\otimes}, g)$$

Even if the divisibility check passes, it might still be the case that the alleged proof was not built using the polynomials in $\mathcal{Q}$. To address this eventuality, the verifier checks whether:

$$B(g^{r_v \alpha_v v_\otimes}, g) = B(g^{r_v v_\otimes}, g^{\alpha_v}) \quad B(g^{r_w \alpha_w w_\otimes}, g) = B(g^{r_w w_\otimes}, g^{\alpha_w}) \quad B(g^{r_y \alpha_y y_\otimes}, g) = B(g^{r_y y_\otimes}, g^{\alpha_y})$$

Finally, the prover might have used the correct polynomials, but didn't use the same coefficients $\boldsymbol{c}_i$ when building $v_\otimes$, $w_\otimes$ and $y_\otimes$. This last concern is resolved by checking whether:

$$B\Big(g^{r_v \beta v_\otimes} g^{r_w \beta w_\otimes} g^{r_y \beta y_\otimes}, g^\gamma\Big) = B\Big(g^{r_v v_\otimes} g^{r_w w_\otimes} g^{r_y y_\otimes}, g^{\beta\gamma}\Big)$$

The protocol as described is not zero-knowledge, but it is quite simple to make it so: the verification key is extended to include $g^{r_v \alpha_v t(s)}$, $g^{r_v \beta v t(s)}$, and then the prover generates a random value δ_v and replaces each polynomial $\boldsymbol{v}_i$ with $\boldsymbol{v}_i + \delta_v t$. The same thing is done for $\boldsymbol{w}$ and $\boldsymbol{y}$: the validity of the checks performed by the verifier is not affected by this change, but it can be shown that the scheme is now statistically zero-knowledge.

It is immediate to see that the proving key $\mathsf{K}_\mathcal{P}$ contains $8|\mathcal{G}_\otimes| + |\mathcal{G}_{in}| + 1$ field elements, the verification key $\mathsf{K}_\mathcal{V}$ contains $3|\mathcal{G}_{in}| + 7$ field elements (6 more in the zero-knowledge setting), and the proof π contains 8 field elements, meaning that its size is independent from the input circuit.

4.4 `libsnark`

In the last ten years, new improvements were put forward to reduce the size of the messages and the complexity of the computation, especially on the prover's side [52] (for example, in [45] the size of the proof was reduced to just 3 field elements). Furthermore, much effort has been put into making working implementations of ZK-SNARK systems, such as `libsnark`[2], a C++ [69] library which implements and refines several ZK-SNARK protocols [28, 46, 7, 13], although the core component (pre-processing ZK-SNARK, or PPZK-SNARK) is based mostly on the Pinocchio protocol and on [14, 45], which are all

[2] `https://github.com/scipr-lab/libsnark`, you can also find a nice empirical comparison of the various protocols.

extensions and improvements of the QAP (and QSP) model of [37].

In order to implement some function f which can be expressed as an arithmetic expression φ (i.e. no variable-length loops or recursion) in `libsnark`, we must provide both the arithmetic circuit and the associated R1CS. In many cases, deriving the R1CS from the arithmetic circuit is quite trivial, but there are instances where having them separate allows for some quite nice optimizations. Although there has been work on *compilers* that translate high-level code to R1CS constraints [32, 9], as it is often the case, this comes at a cost of flexibility. Intuitively, we can divide the usage of `libsnark` in same three phases of the Pinocchio system (or really any other SNARK).

Preprocessing phase

The first thing to do in `libsnark` is choosing, at compile time (or *a priori*, in a theoretical interpretation), which bilinear group to use for the protocol. The standard choice is BN254 [8], but there are also other groups available, such as BLS12[3] [20]. It is important to note that all these groups are paired to a prime field. After choosing the group, we setup the *protoboard* which, as the name suggests, it is the object where one places the components of the circuit:

```
libsnark::protoboard<Field> board;
board.set_input_sizes(PUBLIC_N);
```

The template argument `Field` specifies the underlying field, and `PUBLIC_N` specifies the number of output (i.e. public) variables in the circuit. On the protoboard, we allocate *variables* that will carry the input/intermediate/output values of the circuit evaluation, together with an annotation (for debug). However, it is usually much more convenient to only declare the input and output variables, and delegate the wiring to *gadgets* which act as composable black-boxes:

```
libsnark::pb_variable<Field> output_var;
libsnark::pb_variable<Field> input_var;

output_var.allocate(board, "out");
input_var.allocate(board, "in");
FooGadget gadget{board, input_var, output_var};
```

Note that the first `PUBLIC_N` variables allocated will be considered public, while the remaining ones will be private. A typical gadget must provide two methods: one to generate the R1CS constraints, and one to compute the circuit evaluation given a circuit input, which will be used in the proving phase. First, we generate the constraints:

```
gadget.generate_r1cs_constraints();
```

Internally, `gadget` allocates the required intermediate variables and constrains their linear combinations. A linear combination can be expressed quite naturally:

```
libsnark::pb_linear_combination<Field> lc = a1*x1 + ... + an*xn;
```

An R1CS constraint of the type $\boldsymbol{ab} = \boldsymbol{c}$ is expressed as:

```
libsnark::r1cs_constraint<Field> constraint{lc_a, lc_b, lc_c};
board.add_r1cs_constraint(constraint);
```

[3] `https://electriccoin.co/blog/new-snark-curve/`

Once all the constraints have been specified, the last thing to do is to convert the R1CS to a QAP and get the keys $\mathsf{K}_{\mathcal{P}}$ and $\mathsf{K}_{\mathcal{V}}$. The whole process is done transparently by the library:

```
auto keypair = libsnark::r1cs_ppzksnark_generator(board.get_constraint_system());
```

Now the constraint system and $\mathsf{K}_{\mathcal{V}}$ can be made public, while $\mathsf{K}_{\mathcal{P}}$ is only known by the prover.

Proving phase

To generate a (valid) proof, the prover has to first provide a circuit input and compute the induced evaluation. At the highest level, this is done by setting the values of the input variables and letting the gadget generate the intermediate and the output values:

```
Field x; // some field element
board.val(input_var) = x;
gadget.generate_r1cs_witness(circuit_inputs);
```

Now, he can generate the proof, which is done transparently by the library by using the proving key and the primary (i.e. public) and auxiliary (i.e. private) inputs:

```
auto proof = libsnark::r1cs_ppzksnark_prover(keypair.pk, board.primary_input(),
    board.auxiliary_input());
```

Finally, the prover can make public the primary inputs of the protoboard together with the proof.

Verification phase

At this point, the verifier has to check the proof by using the public key, and there are two kinds of choices regarding how to perform such verification:

- *weak* vs. *strong* verification: in the former case, it is ok for the prover to only provide some of the primary inputs, which will be zero-padded. In the latter case, this is not considered acceptable.
- *offline* vs. *online* verification: in the former case, the verification key is used 'as is', in the latter, it is processed to get faster verification times when used multiple times.

In any case, the whole thing is managed transparently by the library. For example, if we choose strong, online verification:

```
auto proc_vk = libsnark::r1cs_ppzksnark_verifier_process_vk(keypair.vk);
bool halt = libsnark::r1cs_ppzksnark_online_verifier_strong_IC(proc_vk, board.
    primary_input(), proof);
```

The verifier will accept the proof if `halt` is `true`, while it will reject if it is `false`.

Example 4.6. Consider the R1CS in Example 2.10. Listing 4.1 shows the implementation of the corresponding gadget in `libsnark`.

Listing 4.1 The `libsnark` gadget corresponding to the arithmetic formula in Example 2.7.

```
#include <libsnark/gadgetlib1/gadget.hpp>

using namespace libsnark; // don't do this in real code please

template<typename FieldT>
class FooGadget : public gadget<FieldT>
{
    using Var = pb_variable<FieldT>;

    const Var x1, x2, y;
    Var t1, t2, t3;

    FooGadget(protoboard<FieldT> &board,
              const std::string &ann,
              const Var &x1,
              const Var &x2,
              const Var &y) :
        gadget<FieldT>{board, ann}, x1{x1}, x2{x2}, y{y}
    {
        t1.allocate(board, "t1");
        t2.allocate(board, "t2");
        t3.allocate(board, "t3");
    }

    void generate_r1cs_constraints()
    {
        board.add_r1cs_constraint(r1cs_constraint<FieldT>{x1, x1, t1});
        board.add_r1cs_constraint(r1cs_constraint<FieldT>{t1, x1, 8 + 9*x2 + t2});
        board.add_r1cs_constraint(r1cs_constraint<FieldT>{x2, t2, y});
    }

    void generate_r1cs_witness()
    {
        board.val(t1) = board.val(x1) * board.val(x1);
        board.val(t2) = board.val(t1) * board.val(x1) + 4*board.val(x2) + 5;
        board.val(y)  = board.val(x2) * board.val(t2);
    }
};
```

5

ArionHash: a ZK-friendly hash function

5.1 State of the art

5.1.1 MiMC

5.1.2 Poseidon

5.1.3 Griffin

5.1.4 Other designs

5.2 Arion and ArionHash

5.2.1 The Generalized Dynamic Triangular System

5.2.2 Security of Arion

5.3 Implementing ArionHash

5.3.1 Performance of ArionHash

6

Conclusions

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] Elena Andreeva, Rishiraj Bhattacharyya, and Arnab Roy. Compactness of hashing modes and efficiency beyond merkle tree. Cryptology ePrint Archive, Paper 2021/573, 2021. https://eprint.iacr.org/2021/573.

[4] 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.

[5] 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.

[6] László Babai, Lance Fortnow, Leonid A. Levin, and Mario Szegedy. Checking computations in polylogarithmic time. In *Proceedings of the Twenty-Third Annual ACM Symposium on Theory of Computing*, STOC '91, pages 21–32, New York, NY, USA, 1991. Association for Computing Machinery.

[7] Michael Backes, Manuel Barbosa, Dario Fiore, and Raphael M. Reischuk. Adsnark: Nearly practical and privacy-preserving proofs on authenticated data. Cryptology ePrint Archive, Paper 2014/617, 2014. https://eprint.iacr.org/2014/617.

[8] Paulo S. L. M. Barreto and Michael Naehrig. Pairing-friendly elliptic curves of prime order. Cryptology ePrint Archive, Paper 2005/133, 2005. https://eprint.iacr.org/2005/133.

[9] Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. New privacy practices for blockchain software. *IEEE Software*, 39(3):43–49, 2022.

[10] Michael Ben-Or, Oded Goldreich, Shafi Goldwasser, Johan Håstad, Joe Kilian, Silvio Micali, and Phillip Rogaway. Everything provable is provable in zero-knowledge. In Shafi Goldwasser, editor, *Advances in Cryptology — CRYPTO' 88*, pages 37–56, New York, NY, 1990. Springer New York.

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

[12] 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.

[13] Eli Ben-Sasson, Alessandro Chiesa, Daniel Genkin, Eran Tromer, and Madars Virza. Snarks for c: Verifying program executions succinctly and in zero knowledge. Cryptology ePrint Archive, Paper 2013/507, 2013. https://eprint.iacr.org/2013/507.

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

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

[16] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In Ali Miri and Serge Vaudenay, editors, *Selected Areas in Cryptography*, pages 320–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[17] Manuel Blum, Alfredo De Santis, Silvio Micali, and Giuseppe Persiano. Noninteractive zero-knowledge. *SIAM Journal on Computing*, 20(6):1084–1118, 1991.

[18] Manuel Blum, Paul Feldman, and Silvio Micali. Non-interactive zero-knowledge and its applications. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, STOC '88, pages 103–112, New York, NY, USA, 1988. Association for Computing Machinery.

[19] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the weil pairing. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 213–229. Springer, 2001.

[20] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the weil pairing. In Colin Boyd, editor, *Advances in Cryptology - ASIACRYPT 2001, 7th International Conference on the Theory and Application of Cryptology and Information Security, Gold Coast, Australia, December 9-13, 2001, Proceedings*, volume 2248 of *Lecture Notes in Computer Science*, pages 514–532. Springer, 2001.

[21] Joan Boyar, Gilles Brassard, and René Peralta. Subquadratic zero-knowledge. *J. ACM*, 42(6):1169–1193, nov 1995.

[22] Gilles Brassard, David Chaum, and Claude Crépeau. Minimum disclosure proofs of knowledge. *Journal of Computer and System Sciences*, 37(2):156–189, 1988.

[23] 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.

[24] 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.

[25] Ronald Cramer and Ivan Damgård. Zero-knowledge proofs for finite field arithmetic, or: Can zero-knowledge be for free? In Hugo Krawczyk, editor, *Advances in Cryptology — CRYPTO '98*, pages 424–441, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[26] Ronald Cramer and Ivan Damgård. Linear zero-knowledge—a note on efficient zero-knowledge proofs and arguments. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 436–445, New York, NY, USA, 1997. Association for Computing Machinery.

[27] Ivan Damgård. Non-interactive circuit based proofs and non-interactive perfect zero-knowledge with preprocessing. In Rainer A. Rueppel, editor, *Advances in Cryptology — EUROCRYPT' 92*, pages 341–355, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[28] George Danezis, Cedric Fournet, Jens Groth, and Markulf Kohlweiss. Square span programs with applications to succinct nizk arguments. Cryptology ePrint Archive, Paper 2014/718, 2014. `https://eprint.iacr.org/2014/718`.

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

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

[31] A. De Santis, S. Micali, and G. Persiano. Non-interactive zero-knowledge with preprocessing. In *Proceedings on Advances in Cryptology*, CRYPTO '88, pages 269–282, Berlin, Heidelberg, 1990. Springer-Verlag.

[32] Jacob Eberhardt and Stefan Tai. Zokrates - scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 1084–1091, 2018.

[33] U. Feige, S. Goldwasser, L. Lovasz, S. Safra, and M. Szegedy. Approximating clique is almost np-complete. In *[1991] Proceedings 32nd Annual Symposium of Foundations of Computer Science*, pages 2–12, 1991.

[34] U. Feige, D. Lapidot, and A. Shamir. Multiple non-interactive zero knowledge proofs based on a single random string. In *Proceedings [1990] 31st Annual Symposium on Foundations of Computer Science*, pages 308–317 vol.1, 1990.

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

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

[37] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. Cryptology ePrint Archive, Paper 2012/215, 2012. `https://eprint.iacr.org/2012/215`.

[38] Oded Goldreich and Hugo Krawczyk. On the composition of zero-knowledge proof systems. *SIAM Journal on Computing*, 25(1):169–192, 1996.

[39] 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.

[40] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270–299, 1984.

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

[42] 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.

[43] 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.

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

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

[46] Jens Groth and Mary Maller. Snarky signatures:
minimal signatures of knowledge from simulation-extractable snarks. Cryptology ePrint Archive, Paper 2017/540, 2017. `https://eprint.iacr.org/2017/540`.

[47] Jens Groth and Rafail Ostrovsky. Cryptography in the multi-string model. Cryptology ePrint Archive, Paper 2006/407, 2006. `https://eprint.iacr.org/2006/407`.

[48] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

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

[50] Dror Lapidot and Adi Shamir. Publicly verifiable non-interactive zero-knowledge proofs. In Alfred J. Menezes and Scott A. Vanstone, editors, *Advances in Cryptology-CRYPTO' 90*, pages 353–365, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[51] Helger Lipmaa. Progression-free sets and sublinear pairing-based non-interactive zero-knowledge arguments. Cryptology ePrint Archive, Paper 2011/009, 2011. `https://eprint.iacr.org/2011/009`.

[52] Helger Lipmaa. Succinct non-interactive zero knowledge arguments from span programs and linear error-correcting codes. Cryptology ePrint Archive, Paper 2013/121, 2013. `https://eprint.iacr.org/2013/121`.

[53] Ueli Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. Cryptology ePrint Archive, Paper 2003/161, 2003. `https://eprint.iacr.org/2003/161`.

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

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

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

[57] Silvio Micali. Computationally sound proofs. *SIAM J. Comput.*, 30(4):1253–1298, oct 2000.

[58] Yair Oren. On the cunning power of cheating verifiers: Some observations about zero knowledge proofs. In *28th Annual Symposium on Foundations of Computer Science (sfcs 1987)*, pages 462–471, 1987.

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

[60] 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`.

[61] Nicholas Pippenger and Michael J. Fischer. Relations among complexity measures. *J. ACM*, 26(2):361–381, apr 1979.

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

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

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

[65] Adi Shamir. Ip = pspace. *J. ACM*, 39(4):869–877, oct 1992.

[66] 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.

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

[68] Martijn Stam. Beyond uniformity: Better security/efficiency tradeoffs for compression functions. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 397–412, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[69] B. Stroustrup. *The C++ Programming Language: The C++ Programm Lang_p4.* Pearson Education, 2013.

[70] Harshvardhan Tiwari. Merkle-damgård construction method and alternatives: a review. *Journal of Information and Organizational Sciences*, 41(2):283–304, 2017.

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

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

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

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

[75] Andrew C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 80–91, 1982.