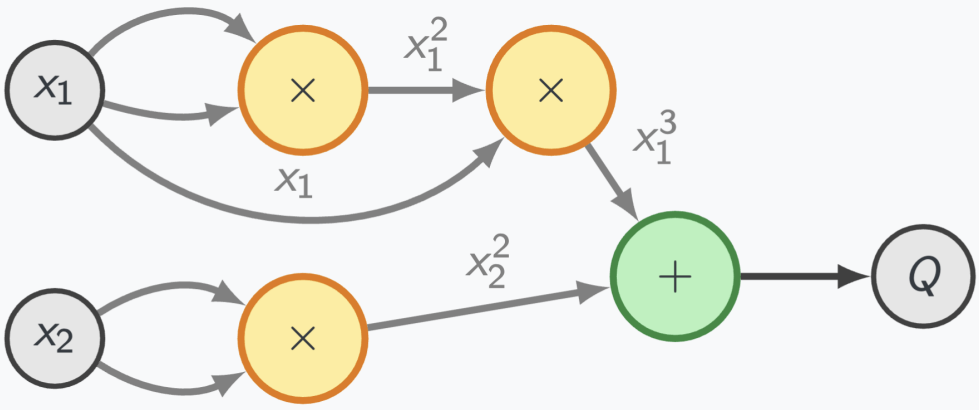
ZKDL Camp Lecture Notes

Distributed Lab

January 16, 2025



Contents

1 Group Theory and Polynomials

1.1 Notation

Before going into the details, let us introduce some notation.

1.1.1 Set Theory

First, let us enumerate some fundamental sets:

- $\mathbb{N}$ – a set of natural numbers. Examples: $10, 13, 193, \ldots$.
- $\mathbb{Z}$ – a set of integers. Examples: $-2, -6, 0, 62, 103, \ldots$.
- $\mathbb{Q}$ – a set of rational numbers. Examples: $\{\frac{n}{m} : n \in \mathbb{Z}, m \in \mathbb{N}\}$.
- $\mathbb{R}$ – a set of real numbers. Examples: $2.2, 1.4, -6.7, \ldots$.
- $\mathbb{R}_{>0}$ – a set of positive real numbers. Examples: $2.6, 10.4, 100.2$.
- $\mathbb{C}$ – a set of complex numbers[1]. Examples: $1 + 2i, 5i, -7 - 5.7i, \ldots$.

Typically we write $a \in A$ to say "element a is in set A". To represent the number of elements in a set A, we write $|A|$. If the set is finite, $|A| \in \mathbb{N}$, otherwise $|A| = \infty$. $A \subset B$ denotes "A is a subset of B" (meaning that all elements of A are also in B, e.g., $\mathbb{Q} \subset \mathbb{R}$).

$A \cap B$ means the intersection of A and B (a set of elements belonging to both A and B), while $A \cup B$ – the union of A and B (the set of elements belonging to either A or B). $A \setminus B$ denotes the set difference (the set of elements belonging to A, but not B). $\overline{A}$ denotes the complement of A (the set of elements not belonging to A). All operations are illustrated in Figure 1.1 (this picture is typically called the *Venn Diagram*).

To define the set, we typically write $\{f(a) : \phi(a)\}$, where $f(a)$ is some function and $\phi(a)$ is a predicate (function, inputting a and returning true/false if a certain condition on a is met). For example, $\{x^3 : x \in \mathbb{R}, x^2 = 4\}$ is "a set of values x^3 which are the real solutions to equation $x^2 = 4$". It is quite easy to see that this set is simply $\{2^3, (-2)^3\} = \{8, -8\}$.

The notation $A \times B$ means a set of pairs (a, b) where $a \in A$ and $b \in B$ (or, written shortly, $A \times B = \{(a, b) : a \in A, b \in B\}$), called a Cartesian product. We additionally introduce notation $A^n := \underbrace{A \times A \times \cdots \times A}_{n \text{ times}}$ – Cartesian product n times. For example, $\mathbb{Q}^3$ is a set of triplets (a, b, c) where $a, b, c \in \mathbb{Q}$, while $\mathbb{Q}^2 \times \mathbb{R}$ is a set of triplets (a, b, c) where $a, b \in \mathbb{Q}$ and $c \in \mathbb{R}$.

1.1.2 Logic

Statement beginning with $\forall$ means "for all...". For instance, $(\forall a \in A \subset \mathbb{R}) : \{a < 1\}$ is read as: "For any a in set A (which is a subset of real numbers), it is true that $a < 1$". Or, more shortly, "Any (real) a from A is less than 1".

Statement beginning from $\exists$ means "there exists such...". Let us consider the following example: $(\exists \varepsilon > 0)(\forall a \in A) : \{a > \varepsilon\}$ is read as "there exists such a positive ε such that for any element a from A, a is greater than ε", or, more concisely, "there exists a positive constant ε such that any element from A is greater than ε".

Statement beginning from $\exists!$ means "there exists a unique...". For example, $(\exists! x \in \mathbb{R}_{>0}) : \{x^2 = 4\}$ is read as "there exists a unique positive real x such that $x^2 = 4$".

Symbol $\wedge$ means "and". For example, $\{x \in \mathbb{R} : x^2 = 4 \wedge x > 0\}$ is read as "a set of real x

[1] Complex number is an expression in a form $x + iy$ for $i^2 = -1$

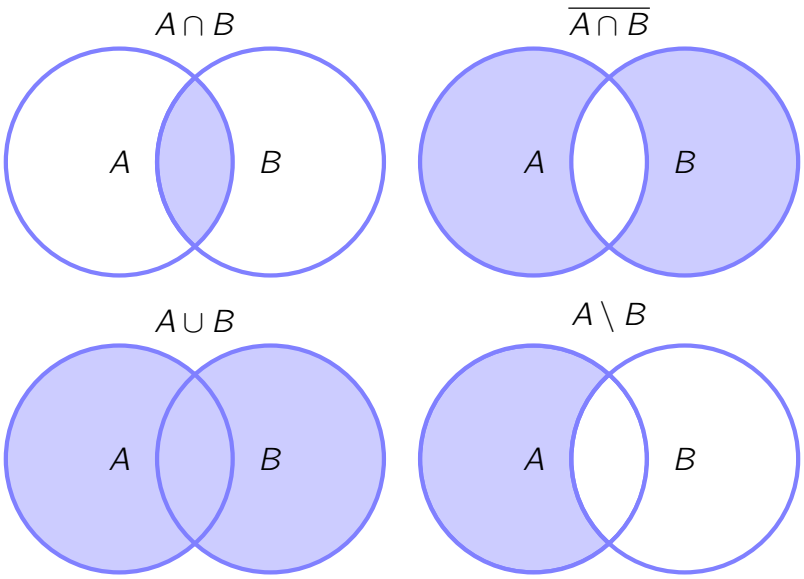


Figure 1.1: Set operations illustrated with Venn diagrams.

such that $x^2 = 4$ and x is positive". Of course, $\{x \in \mathbb{R} : x^2 = 4 \wedge x > 0\} = \{2\}$.

Symbol $\vee$ means "or". For example, $\{x \in \mathbb{R} : x^2 = 4 \vee x^2 = 9\}$ is read as "a set of real x such that either $x^2 = 4$ or $x^2 = 9$". Here, this set is equal to $\{-2, 2, -3, 3\}$.

1.1.3 Randomness and Probability

To denote the probability of an event A happening, we write $\Pr[A]$. For example, if event A represents that a coin lands heads, then $\Pr[A] = 0.5$.

Fix some set A. To denote that we are uniformly randomly picking some element from A, we write $a \xleftarrow{R} A$. For example, $a \xleftarrow{R} \{1, 2, 3, 4, 5, 6\}$ means that we are picking a number from 1 to 6 uniformly at random.

1.1.4 Sequences and Vectors

To denote the infinite sequence $\{x_1, x_2, x_3, \dots\}$ we write $\{x_n\}_{n \in \mathbb{N}}$. To denote the finite sequence $\{x_1, x_2, \dots, x_n\}$ we write $\{x_k\}_{k=1}^{n}$.

Vector is a collection of elements $\mathbf{x} = (x_1, \dots, x_n) \in A^n$. Finally, the scalar product[2] is denoted as $\langle \mathbf{x}, \mathbf{y} \rangle := \sum_{k=1}^{n} x_k y_k$.

1.2 Introduction to Abstract Algebra

1.2.1 Groups

Throughout the lectures, probably the most important topic is the *group theory*.

As you can recall from the high school math, typically real-world processes are described using real numbers, denoted by $\mathbb{R}$. For example, to describe the position or the velocity of an object, you would rather use real numbers.

When it comes to working with computers though, real numbers become very inconvenient

[2] It is totally normal if you do not know what that is, we will explain more in the Bulletproof lecture

to work with. For instance, different programming languages might output different values for quite a straightforward operation 2.01 + 2.00. This becomes a huge problem when dealing with cryptography, which must check *precisely* whether two quantities are equal. For example, if the person's card number is N and the developed system operates with a different, but very similar card with number $N + k$ for $k \ll N$, then this system can be safely thrown out of the window. See Figure 1.2.

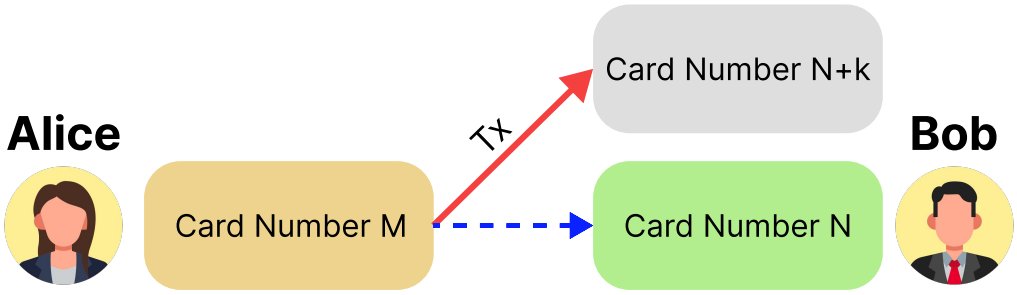


Figure 1.2: Alice pays to Bob to a card number N, but our awesome system pays to $N + k$ instead. Bob would not be happy...

This motivates us to work with integers (denoted by $\mathbb{Z}$), instead. This solves the problem with card numbers, but for cryptography this object is still not really suitable since it is hard to build a secure and reliable protocol exploiting pure integers (without using a more complex structure).

This motivates us to use a different primitive for dealing with cryptographic systems. Similarly to programmers working with interfaces (or traits, if you are the *Rust* developer), mathematicians also use the so-called *groups* to represent objects obeying a certain set of rules. The beauty is that we do not concretize *how* operations in this set are performed, but rather state the fact that we can somehow combine elements with the pre-defined properties. We can then discover properties of such objects and whenever we apply the concrete "implementation" (spoiler, group of points on elliptic curve), these properties would still hold.

Remark. Further discussion with abstract objects should be regarded as "interfaces" which do not concretize the "implementation" of an object. It merely shows the nature of an object without going into the details.

Now, let us get dirty and define what the **group** is.

Definition 1.1. Group, denoted by $(\mathbb{G}, \oplus)$, is a set with a binary operation $\oplus$, obeying the following rules:

1. **Closure:** Binary operations always outputs an element from $\mathbb{G}$, that is $\forall a, b \in \mathbb{G} : a \oplus b \in \mathbb{G}$.
2. **Associativity:** $\forall a, b, c \in \mathbb{G} : (a \oplus b) \oplus c = a \oplus (b \oplus c)$.
3. **Identity element:** There exists a so-called identity element $e \in \mathbb{G}$ such that $\forall a \in \mathbb{G} : e \oplus a = a \oplus e = a$.
4. **Inverse element:** $\forall a \in \mathbb{G}\ \exists b \in \mathbb{G} : a \oplus b = b \oplus a = e$. We commonly denote the inverse element as $(\ominus a)$.

Quite confusing at first glance, right? The best way to grasp this concept is to consider a couple of examples.

Example. A group of integers with the regular addition $(\mathbb{Z}, +)$ (also called the *additive* group of integers) is a group. Indeed, an identity element is $e_{\mathbb{Z}} = 0$, associativity obviously holds, and an inverse for each element $a \in \mathbb{Z}$ is $(\ominus a) := -a \in \mathbb{Z}$.

Remark. We use the term **additive group** when we mean that the binary operation is addition $+$, while **multiplicative group** means that we are multiplying two numbers via $\times$[a].

[a] In this section, regard $\cdot$ and $\times$ as the same operation of multiplication.

Example. The multiplicative group of positive real numbers $(\mathbb{R}_{>0}, \times)$ is a group for similar reasons. An identity element is $e_{\mathbb{R}_{>0}} = 1$, while the inverse for $a \in \mathbb{R}_{>0}$ is defined as $\frac{1}{a}$.

Example. The additive set of natural numbers $(\mathbb{N}, +)$ is not a group. Although operation of addition is closed, there is no identity element nor inverse element for, say, 2 or 10.

Example. That is possible to have the situation when the element $a \in \mathbb{G}$ can be its own inverse, meaning $a = a^{-1}$. This happens when $a^2 = e$. Additionally, we can mention that for any group $\mathbb{G} = \{g, e\}$ with the order $|\mathbb{G}| = 2$ we have $g^2 = e$.

One might ask a reasonable question: suppose you pick $a, b \in \mathbb{G}$. Is $a \oplus b$ the same as $b \oplus a$? Unfortunately, for some groups, this is not true.

For this reason, it makes sense to give a special name to a group in which the operation is commutative (meaning, we can swap the elements in the operation).

Definition 1.2. A group $(\mathbb{G}, \oplus)$ is called **abelian** if $\forall a, b \in \mathbb{G} : a \oplus b = b \oplus a$.

Example. The additive group of integers $(\mathbb{Z}, +)$ is an abelian group. Indeed, $a + b = b + a$ for any $a, b \in \mathbb{Z}$.

Example. The set of 2×2 matrices with real entries and determinant 1 (denoted by $\mathrm{SL}(2, \mathbb{R})$) is a group with respect to matrix multiplication. However, this group is not abelian! Take

$$A = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix}, \quad B = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix}. \tag{1}$$

Then, it is easy to verify that

$$AB = \begin{pmatrix} 2 & 1 \\ 1 & 1 \end{pmatrix}, \quad BA = \begin{pmatrix} 1 & 1 \\ 1 & 2 \end{pmatrix}, \tag{2}$$

so clearly $AB \neq BA$ – the elements of $\mathrm{SL}(2, \mathbb{R})$ do not commute.

Remark. Further, we will write ab instead of $a \times b$ and a^{-1} instead of $\ominus a$ for the sake of simplicity (and because it is more common in the literature). As mentioned before, it is usually called the *multiplicative notation*.

Finally, for cryptography it is important to know the number of elements in a group. This number is called the *order* of the group.

Definition 1.3. The **order** of a finite group $\mathbb{G}$ is the number of elements in the group. We denote the order of a group as $|\mathbb{G}|$.

Example. Integers modulo 13, denoted by $\mathbb{Z}_{13}$, is a group with respect to addition modulo 13 (e.g., $5 + 12 = 4$ in $\mathbb{Z}_{13}$). The order of this group is 13.

Despite the aforementioned definitions, many things are not generally obvious. For example, one might ask whether the identity element is unique. Or, whether the inverse element is unique for each group element. For that reason, we formulate the following lemma.

Lemma 1.4. Suppose $\mathbb{G}$ is a group. Then, the following statements hold:

1. The identity element is unique.
2. The inverse element is unique for each element: $\forall a \in \mathbb{G}\, \exists! a^{-1} \in \mathbb{G} : aa^{-1} = a^{-1}a = e$.
3. For all $a, b \in \mathbb{G}$ there is a unique $x \in \mathbb{G}$ such that $ax = b$.
4. If $ab = ac$ then $b = c$. Similarly, if $xy = zy$ then $x = z$.

Since this guide is not a textbook on abstract algebra, we will not prove all the statements. However, we will prove the first and second one to show the nature of the proofs in abstract algebra.

First Statement Proof. Suppose $e_1, e_2 \in \mathbb{G}$ are both identity elements. Consider $e_1 e_2$. From the definition of the identity element, we know that $e_1 e_2 = e_1$ and $e_1 e_2 = e_2$. Therefore, $e_1 = e_2$.

Second Statement Proof. Take $g \in \mathbb{G}$ and suppose $a, b \in \mathbb{G}$ are both inverses of g. By defininition,

$$ag = ga = e, \quad bg = gb = e. \tag{3}$$

Now, notice that

$$a = ae = a(gb) = (ag)b = eb = b \tag{4}$$

Thus, $a = b$.

Exercise. Prove the third and fourth statements.

1.2.2 Subgroups

When we are finally comfortable with the concept of a group, we can move on to the concept of a *subgroup*.

Suppose we have a group $(\mathbb{G}, \oplus)$. Suppose one takes the subset $\mathbb{H} \subset \mathbb{G}$. Of course, since all elements in $\mathbb{H}$ are still elements in $\mathbb{G}$, we can conduct operations between them via $\oplus$. The natural question to ask is whether $\mathbb{H}$ is a group itself. Yes, but at the same time $\mathbb{H}$ is called a **subgroup** of $\mathbb{G}$.

Definition 1.5. A subset $\mathbb{H} \subset \mathbb{G}$ is called a **subgroup** of $\mathbb{G}$ if $\mathbb{H}$ is a group with respect to the same operation $\oplus$. We denote this as $\mathbb{H} \leq \mathbb{G}$.

Example. Of course, not every subset of $\mathbb{G}$ is a subgroup. Take $(\mathbb{Z}, +)$. If we cut, say, 3 out of $\mathbb{Z}$ (so we get $\mathbb{H} = \mathbb{Z} \setminus \{3\}$), then $\mathbb{H}$ is not a subgroup of $\mathbb{Z}$ since an element -3 does not have an inverse in $\mathbb{H}$. Moreover, it is not closed: take $1, 2 \in \mathbb{H}$. In this case, $1 + 2 = 3 \notin \mathbb{H}$.

Example. Now, let us define some valid subgroup of $\mathbb{Z}$. Take $\mathbb{H} = \{3k : k \in \mathbb{Z}\}$ – a set of integers divisible by 3 (commonly denoted as $3\mathbb{Z}$). This is a subgroup of $\mathbb{Z}$, since it is closed under addition, has an identity element 0, and has an inverse for each element $3k$ (namely, $-3k$). That being said, $3\mathbb{Z} \leq \mathbb{Z}$.

These are good examples, but let us consider a more interesting one, which we call a lemma. It is frequently used further when dealing with cosets and normal subgroups, but currently regard this just as an exercise.

Lemma 1.6. Let $\mathbb{G}$ be a group and $g \in \mathbb{G}$. The centralizer of g is defined to be

$$C_g = \{h \in \mathbb{G} : hg = gh\} \tag{5}$$

Then, C_g is a subgroup of $\mathbb{G}$.

Exercise. Prove the lemma.

1.2.3 Cyclic Groups

Probably, cyclic groups are the most interesting groups in the world of cryptography. But before defining them, we need to know how to add/subtract elements multiple times (that is, multiplying by an integer). Suppose we have a group $\mathbb{G}$ and $g \in \mathbb{G}$. Then, g^n means multiplying (adding) g to itself n times. If n is negative, then we add g^{-1} to itself $|n|$ times. For $n = 0$ we define $g^0 = e$. Now, let us define what the cyclic group is.

Definition 1.7. Given a group $\mathbb{G}$ and $g \in \mathbb{G}$ the cyclic subgroup generated by g is

$$\langle g \rangle = \{g^n : n \in \mathbb{Z}\} = \{\dots, g^{-3}, g^{-2}, g^{-1}, e, g, g^2, g^3, \dots\}. \tag{6}$$

Example. Consider the group of integers modulo 12, denoted by $\mathbb{Z}_{12}$. Consider $2 \in \mathbb{Z}_{12}$, the group generated by 2 is then

$$\langle 2 \rangle = \{2, 4, 6, 8, 10, 0\} \tag{7}$$

Definition 1.8. We say that a group $\mathbb{G}$ is **cyclic** if there exists an element $g \in \mathbb{G}$ such that $\mathbb{G}$ is generated by g, that is, $\mathbb{G} = \langle g \rangle$.

Example. The group of integers $(\mathbb{Z}, +)$ is an infinite cyclic group. Indeed, it is generated by 1.

1.2.4 Isomorphisms and Endomorphisms

Finally, we will define the concept of isomorphisms and endomorphisms. These are important concepts in the world of cryptography, since they allow us to compare different groups. Namely, suppose we have two groups $(\mathbb{G}, \oplus)$ and $(\mathbb{H}, \odot)$. Is there any way to state that these two groups are the same? The answer is yes, and this is done via isomorphisms.

Definition 1.9. A function $\varphi : \mathbb{G} \to \mathbb{H}$ is called an **homomorphism** if it is a function that preserves the group operation, that is,

$$\forall a, b \in \mathbb{G} : \varphi(a \oplus b) = \varphi(a) \odot \varphi(b). \tag{8}$$

Definition 1.10. An **isomorphism** is a bijective homomorphism.

Definition 1.11. If there exists an isomorphism between two groups $\mathbb{G}$ and $\mathbb{H}$, we say that these groups are isomorphic and write $\mathbb{G} \cong \mathbb{H}$.

Example. Consider the group of integers $(\mathbb{Z}, +)$ and the group of integers modulo 12 $(\mathbb{Z}_{12}, +)$. The function $\varphi : \mathbb{Z} \to \mathbb{Z}_{12}$ defined as $\varphi(x) = x \bmod 12$ is a homomorphism. Indeed:

$$\varphi(a + b) = (a + b) \bmod 12 = (a \bmod 12) + (b \bmod 12) = \varphi(a) + \varphi(b). \tag{9}$$

However, this function is not an isomorphism, since it is not bijective. For example, $\varphi(0) = \varphi(12) = 0$.

Example. Additive group of reals $(\mathbb{R}, +)$ and the multiplicative group of positive reals $(\mathbb{R}_{>0}, \times)$ are isomorphic. The function $\varphi : \mathbb{R} \to \mathbb{R}_{>0}$ defined as $\varphi(x) = e^x$ is an isomorphism. Indeed:

$$\varphi(a + b) = e^{a+b} = e^a \cdot e^b = \varphi(a) \cdot \varphi(b). \tag{10}$$

Thus, φ is a homomorphism. It is also injective since $e^x = e^y \implies x = y$. Finally, it is obviously onto. This means $(\mathbb{R}, +) \cong (\mathbb{R}_{>0}, \times)$.

Example. All groups of order 2 are isomorphic to $\mathbb{Z}_2$. Indeed, let $\mathbb{G} = \{g, e\}$ – any group of order 2, and define $\varphi : \mathbb{Z}_2 \to \mathbb{G}$ as $\varphi(0) = e$ and $\varphi(1) = g$. This is an isomorphism.

A generalization of the above example is the following quite interesting theorem:

Theorem 1.12. Suppose $\mathbb{G} = \langle g \rangle$ is a finite cyclic group, meaning $|G| = n \in \mathbb{N}$. Then, $\mathbb{G} \cong \mathbb{Z}_n$.

Idea of the proof. Define a function $\varphi : \mathbb{Z}_n \to \mathbb{G}$ as $m \mapsto g^m$. One can prove that this is an isomorphism.

Here, it is quite evident that isomorphism tells us that the groups have the same structure. Moreover, it is correct to say that if $\mathbb{G} \equiv \mathbb{H}$, then $\mathbb{G}$ and $\mathbb{H}$ are *equivalent* since $\cong$ is an

equivalence relation.

Exercise (*). Prove that $\cong$ is an equivalence relation.

Finally, we will define the concept of an endomorphism and automorphism to finish the section.

Definition 1.13. An **endomorphism** is a function φ which maps set X to itself ($\varphi : X \to X$).

Definition 1.14. An **automorphism** is an isomorphic endomorphism.

Example. Given a group $\mathbb{G}$, fixate $a \in \mathbb{G}$. The map $\varphi : x \mapsto axa^{-1}$ is an automorphism.

Last two definitions are especially frequently used in Elliptic Curves theory.

1.3 Fields

1.3.1 Formal Definition

Although typically one introduces rings before fields, we believe that for the basic understanding, it is better to start with fields.

Notice that when dealing with groups, we had a single operation $\oplus$, which, depending on the context, is either interpreted as addition or multiplication. However, fields allow to extend this concept a little bit further by introducing a new operation, say, $\odot$, which, combined with $\oplus$, allows us to perform the basic arithmetic.

This is very similar to the real or rational numbers, for example. We can add, subtract, multiply, and divide them. This is exactly what fields are about, but in a more abstract way. That being said, let us see the definition.

Definition 1.15. A **field** is a set $\mathbb{F}$ with two operations $\oplus$ and $\odot$ such that:

1. $(\mathbb{F}, \oplus)$ is an abelian group with identity $e_\oplus$.
2. $(\mathbb{F} \setminus \{e_\oplus\}, \odot)$ is an abelian group.
3. The **distributive law** holds: $\forall a, b, c \in \mathbb{F} : a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$.

What this definition basically states is that we can perform the following operations:

1. Addition: $a \oplus b$, inherited from group structure $(\mathbb{F}, \oplus)$.
2. Subtraction: $a \oplus (\ominus b)$, inherited from group structure $(\mathbb{F}, \oplus)$.
3. Multiplication: $a \odot b$, inherited from group structure $(\mathbb{F} \setminus \{e_\oplus\}, \odot)$.
4. Division: $a \odot b^{-1}$, except for $b = 0$, inherited from group structure $(\mathbb{F} \setminus \{e_\oplus\}, \odot)$.

Example. The set of real numbers $(\mathbb{R}, +, \times)$ is obviously a field.

Example. The set of complex numbers $(\mathbb{C}, +, \times)$ is also a field. Indeed, let us see how we can perform operations. Suppose we are given $z = a_0 + a_1 i$ and $w = b_0 + b_1 i$ with $i^2 + 1 = 0$. In this case:

1. Addition: $z + w = (a_0 + b_0) + (a_1 + b_1)i$.
2. Subtraction: $z - w = (a_0 - b_0) + (a_1 - b_1)i$.
3. Multiplication: $z \cdot w = (a_0 b_0 - a_1 b_1) + (a_0 b_1 + a_1 b_0)i$.
4. Division: $z/w = \frac{a_0 b_0 + a_1 b_1}{b_0^2 + b_1^2} + \frac{a_1 b_0 - a_0 b_1}{b_0^2 + b_1^2} i$.

Interestingly though, it is very difficult to come up with some more complicated, non-trivial examples. For that reason, we will simply move to the most central field used in cryptography – finite fields.

1.3.2 Finite Fields

Recall: we do not like reals, we want to operate with integers! But notice that $(\mathbb{Z}, +, \times)$ does not form a field since division is not closed. For that reason, fixate some integer p and consider the set $\mathbb{Z}_p := \{0, 1, 2, \ldots, p-2, p-1\}$. Now, we will define operations as follows:

Addition. To add $a, b \in \mathbb{Z}_p$, add them as usual to get $c \leftarrow a + b$. However, this way, operation is not closed, since c might be easily greater than $p-1$ (e.g., for $a = b = p-2$). To fix this, take $c' \in \mathbb{Z}_p$ such that $c \equiv c' \pmod{p}$ (or, written more concisely, $c' = (a+b) \bmod p$).

Example. Take $p = 5$. Then, $3 + 4 = 2$ in $\mathbb{Z}_5$ since $c = 3 + 4 = 7$ and $7 \equiv 2 = c' \pmod 5$.

Multiplication and subtraction. The algorithm is the same. Find $c \leftarrow ab$ or $c \leftarrow a - b$, respectively, and find $c' \in \mathbb{Z}_p$ such that $c' \equiv c \pmod{p}$.

Example. Again, suppose $p = 5$. Then, $3 \cdot 4 = 2$ in $\mathbb{F}_5$ since $c = 3 \cdot 4 = 12$ and $12 \equiv 2 = c' \pmod 5$. Similarly, $3 - 4 = 4$ in $\mathbb{F}_5$ since $c = 3 - 4 = -1$ and $-1 \equiv 4 = c' \pmod 5$.

Inversion. Inversion is a bit more tricky. Recall that $(\mathbb{Z}_p \setminus \{0\}, \times)$ must be an abelian group, meaning that for each $a \in \mathbb{Z}_p$ there should be some $x \in \mathbb{Z}_p$ such that $ax = 1$ (multiplication in a sense of definition above). In other words, we need to solve the modular equation:

$$ax \equiv 1 \pmod{p}. \tag{11}$$

Note that there is no guarantee that for any $a \in \mathbb{Z}_p \setminus \{0\}$ we might find such x. For example, take $p = 10$ and $a = 2$. Then, $2x \equiv 1 \pmod{10}$ has no solution.

The only way to guarantee that for any $a \in \mathbb{Z}_p \setminus \{0\}$ we might find such x is to take p to be a prime number. This is the reason why we call such fields **prime fields** (or, in many cases, one calls them **finite fields**).

So finally, with all the definitions, we can define the finite field.

Definition 1.16. A **finite field** (or *prime field*) is a set with prime number p of elements $\{0, 1, \ldots, p-2, p-1\}$, in which operations are defined "modulo p" (see details above). Typically, finite fields are denoted as $\mathbb{F}_p$ or $\mathrm{GF}(p)$.

Finite fields is the core object in cryptography. Instead of real numbers or pure integers, we

will almost always use finite fields.

Remark. In many cases, one might encounter both $\mathbb{F}_p$ and $\mathbb{Z}_p$ notations. The difference is the following: when one refers to $\mathbb{Z}_p$, it is typically assumed that the operations are performed in the ring[a] of integers modulo p (meaning, we need only addition, subtraction, and multiplication in the protocol), while division is of little interest. When one refers to $\mathbb{F}_p$, it is typically assumed that we need full arithmetic (including division) for the procool.

[a] We have not defined as of now what ring is, but, roughly speaking, this is a field without multiplicative inverses

Example. Consider $9, 14 \in \mathbb{F}_{17}$. Some examples of calculations:

1. $9 + 14 = 6$.
2. $9 - 14 = 12$.
3. $9 \times 14 = 7$.
4. $14^{-1} = 11$ since $14 \cdot 11 = 154 \equiv 1 \pmod{17}$.

1.4 Polynomials

1.4.1 Basic Definition

Polynomials are intensively used in almost all areas of cryptography. In our particular case, polynomials will encode the information about statements we will need to prove. That being said, let us define what polynomial is.

Definition 1.17. A **polynomial** $f(x)$ is a function of the form

$$p(x) = c_0 + c_1 x + c_2 x^2 + \cdots + c_n x^n = \sum_{k=0}^{n} c_k x^k, \tag{12}$$

where $c_0, c_1, \ldots, c_n$ are coefficients of the polynomial.

Notice that for now we did not specify what are c_i's. We are interested in the case where $c_i \in \mathbb{F}$, where $\mathbb{F}$ is a field.

Definition 1.18. A set of polynomials depending on x with coefficients in a field $\mathbb{F}$ is denoted as $\mathbb{F}[x]$, that is

$$\mathbb{F}[x] = \left\{ p(x) = \sum_{k=0}^{n} c_k x^k : c_k \in \mathbb{F},\ k = 0, \ldots, n \right\}. \tag{13}$$

Definition 1.19. Evaluation of a polynomial $p(x) \in \mathbb{F}[x]$ at point $x_0 \in \mathbb{F}$ is simply finding the value of $p(x_0) \in \mathbb{F}$.

Example. Consider the finite field $\mathbb{F}_3$. Then, some examples of polynomials from $\mathbb{F}_3[x]$ are listed below:

1. $p(x) = 1 + x + 2x^2$.
2. $q(x) = 1 + x^2 + x^3$.
3. $r(x) = 2x^3$.

If we were to evaluate these polynomials at $1 \in \mathbb{F}_3$, we would get:

1. $p(1) = 1 + 1 + 2 \cdot 1 \bmod 3 = 1$.
2. $q(1) = 1 + 1 + 1 \bmod 3 = 0$.
3. $r(1) = 2 \cdot 1 = 2$.

Definition 1.20. The **degree** of a polynomial $p(x) = c_0 + c_1 x + c_2 x^2 + \dots$ is the largest $k \in \mathbb{Z}_{\geq 0}$ such that $c_k \neq 0$. We denote the degree of a polynomial as $\deg p$. We also denote by $\mathbb{F}^{(\leq m)}[x]$ a set of polynomials of degree at most m.

Example. The degree of the polynomial $p(x) = 1 + 2x + 3x^2$ is 2, so $p(x) \in \mathbb{F}_3^{(\leq 2)}[x]$.

Theorem 1.21. For any two polynomials $p, q \in \mathbb{F}[x]$ and $n = \deg p$, $m = \deg q$, the following two statements are true:

1. $\deg(pq) = n + m$.
2. $\deg(p + q) = \max\{n, m\}$ if $n \neq m$ and $\deg(p + q) \leq m$ for $m = n$.

1.4.2 Roots and divisibility

Definition 1.22. Let $p(x) \in \mathbb{F}[x]$ be a polynomial of degree $\deg p \geq 1$. A field element $x_0 \in \mathbb{F}$ is called a root of $p(x)$ if $p(x_0) = 0$.

Example. Consider the polynomial $p(x) = 1 + x + x^2 \in \mathbb{F}_3[x]$. Then, $x_0 = 1$ is a root of $p(x)$ since $p(x_0) = 1 + 1 + 1 \bmod 3 = 0$.

One of the fundamental theorems of polynomials is following.

Theorem 1.23. Let $p(x) \in \mathbb{F}[x], \deg p \geq 1$. Then, $x_0 \in \mathbb{F}$ is a root of $p(x)$ if and only if there exists a polynomial $q(x)$ (with $\deg q = n - 1$) such that

$$p(x) = (x - x_0)q(x) \tag{14}$$

Example. Note that $x_0 = 1$ is a root of $p(x) = x^2 + 2$. Indeed, we can write $p(x) = (x - 1)(x - 2)$, so here $q(x) = x - 2$.

Also, this might not be obvious, but we can also divide polynomials in the same way as we divide integers. The result of division is not always a polynomial, so we also get a remainder.

Theorem 1.24. Given $f, g \in \mathbb{F}[x]$ with $g \neq 0$, there are unique polynomials $p, q \in \mathbb{F}[x]$ such that

$$f = q \cdot g + r, \ 0 \leq \deg r < \deg g \tag{15}$$

Example. Consider $f(x) = x^3 + 2$ and $g(x) = x + 1$ over $\mathbb{R}$. Then, we can write $f(x) = (x^2 - x + 1)g(x) + 1$, so the remainder of the division is 1. Typically, we denote this as:

$$f \operatorname{div} g = x^2 - x + 1, \quad f \bmod g = 1. \tag{16}$$

The notation is pretty similar to one used in integer division.

Similarly, one can define gcd, lcm, and other number field theory operations for polynomials. However, we will not go into details here, besides mentioning the divisibility.

Definition 1.25. A polynomial $f(x) \in \mathbb{F}[x]$ is called **divisible** by $g(x) \in \mathbb{F}[x]$ (or, g **divides** f, written as $g \mid f$) if there exists a polynomial $h(x) \in \mathbb{F}[x]$ such that $f = gh$.

Theorem 1.26. If $x_0 \in \mathbb{F}$ is a root of $p(x) \in \mathbb{F}[x]$, then $(x - x_0) \mid p(x)$.

Definition 1.27. A polynomial $f(x) \in \mathbb{F}[x]$ is said to be **irreducible** in $\mathbb{F}$ if there are no polynomials $g, h \in \mathbb{F}[x]$ both of degree more than 1 such that $f = gh$.

Example. A polynomial $f(x) = x^2 + 16$ is irreducible in $\mathbb{R}$. In turn, $f(x) = x^2 - 2$ is not irreducible since $f(x) = (x - \sqrt{2})(x + \sqrt{2})$.

Example. There are no polynomials over complex numbers $\mathbb{C}$ with degree more than 2 that are irreducible. This follows from the *fundamental theorem of algebra*.

1.4.3 Interpolation

Now, let us ask the question: what defines the polynomial? Well, given expression $p(x) = \sum_{k=0}^{n} c_k x^k$ one can easily say: "hey, I need to know the coefficients $\{c_k\}_{k=0}^{n}$".

Indeed, each polynomial of degree n is uniquely determined by the vector of its coefficients $(c_0, c_1, \ldots, c_n) \in \mathbb{F}^n$. However, that is not the only way to define a polynomial.

Suppose I tell you that $p(x) = ax + b$ – just a simple linear function over $\mathbb{R}$. Suppose I tell you that $p(x)$ intercepts $(0, 0)$ and $(1, 2)$. Then, you can easily say that $p(x) = 2x$.

The more general question is: suppose $\deg p = n$, how many points do I need to define the polynomial $p(x)$ uniquely? The answer is $n + 1$ distinct points. This is the idea behind the interpolation: the polynomial is uniquely defined by $n + 1$ distinct points on the plane. An example is depicted in Figure 1.3. Now, let us see how we can interpolate the polynomial practically.

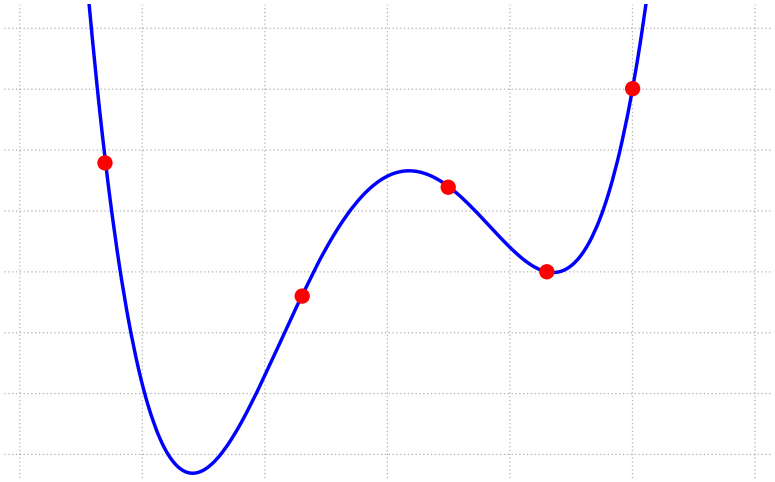


Figure 1.3: 5 points on the plane uniquely define the polynomial of degree 4.

Theorem 1.28. Given a set of points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \subset \mathbb{F} \times \mathbb{F}$, there is a unique polynomial $L(x)$ of degree n such that $L(x_i) = y_i$ for all $i = 0, \ldots, n$. This polynomial is called the **Lagrange interpolation polynomial** and can be found through the following formula:

$$L(x) = \sum_{i=0}^{n} y_i \ell_i(x), \quad \ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}. \tag{17}$$

Lemma 1.29. The polynomials $\{\ell_i\}_{i=1}^n$, in fact, have quite an interesting property:

$$\ell_i(x_j) = \delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases}, \tag{18}$$

where δ_{ij} is the Kronecker delta. Moreover, $\{\ell_i\}_{i=1}^n$ form a basis of $\mathbb{F}^{(\leq n)}[x]$: for any polynomial $p(x) \in \mathbb{F}^{(\leq n)}[x]$ there exist unique coefficients $\alpha_0, \ldots, \alpha_n \in \mathbb{F}$ such that

$$p(x) = \sum_{i=0}^{n} \alpha_i \ell_i(x). \tag{19}$$

Example. Suppose we have points $(0, 1)$ and $(1, 2)$. Then, the Lagrange interpolation polynomial is

$$L(x) = 1 \cdot \frac{x-1}{0-1} + 2 \cdot \frac{x-0}{1-0} = (-1) \cdot (x-1) + 2 \cdot x = x + 1 \tag{20}$$

1.4.4 Some Fun: Shamir's Secret Sharing

Shamir's Secret Sharing, also known as (t, n)-threshold scheme, is one of the protocols exploiting Lagrange Interpolation.

But first, let us define what secret sharing is. Suppose we have a secret data α, which is represented as an element from some finite set F. We divide this secret into n pieces $\alpha_1, \ldots, \alpha_n \in F$ in such a way:

1. Knowledge of any t shares can reconstruct the secret α.
2. Knowledge of any number of shares below t cannot be used to reconstruct the secret α.

Now, let us define the sharing scheme.

Definition 1.30. Secret Sharing scheme is a pair of efficient algorithms $(\mathsf{Gen}, \mathsf{Comb})$ which work as follows:

- $\mathsf{Gen}(\alpha, t, n)$: probabilistic sharing algorithm that yields n shards $(\alpha_1, \ldots, \alpha_t)$ for which t shards are needed to reconstruct the secret α.
- $\mathsf{Comb}(\mathcal{I}, \{\alpha_i\}_{i\in\mathcal{I}})$: deterministic reconstruction algorithm that reconstructs the secret α from the shards $\mathcal{I} \subset \{1, \ldots, n\}$ of size t.

Here, we require the **correctness**: for every $\alpha \in F$, for every possible output $(\alpha_1, \ldots, \alpha_n) \leftarrow \mathsf{Gen}(\alpha, t, n)$, and any t-size subset $\mathcal{I}$ of $\{1, \ldots, n\}$ we have

$$\mathsf{Comb}(\mathcal{I}, \{\alpha_i\}_{i\in\mathcal{I}}) = \alpha. \tag{21}$$

Now, Shamir's protocol is one of the most famous secret sharing schemes. It works as follows: our finite set is $\mathbb{F}_q$ for some large prime q. Then, algorithms in the protocol are defined as follows:

- $\mathsf{Gen}(\alpha, k, n)$: choose random $k_1, \ldots, k_{t-1} \xleftarrow{R} \mathbb{F}_q$ and define the polynomial

$$\omega(x) := \alpha + k_1 x + k_2 x^2 + \cdots + k_{t-1} x^{t-1} \in \mathbb{F}_q^{\leq (t-1)}[x], \tag{22}$$

  and then compute $\alpha_i \leftarrow \omega(i) \in \mathbb{F}_q,\ i = 1, \ldots, n$. Return $(\alpha_1, \ldots, \alpha_n)$.
- $\mathsf{Comb}(\mathcal{I}, \{\alpha_i\}_{i\in\mathcal{I}})$: reconstruct the polynomial $\omega(x)$ using Lagrange interpolation and return $\omega(0) = \alpha$.

The combination function is possible since, having t points $\{i, \alpha_i\}_{i\in\mathcal{I}}$ with $\omega(i) = \alpha_i$, we can fully reconstruct the polynomial $\omega(x)$ and then evaluate it at 0 to get α.

Instead, suppose we have only $t-1$ (or less) pairs $\{i, \alpha_i\}_{i\in\mathcal{I}}$. Then, there are many polynomials $\omega(x)$ that pass through these points (in fact, if we were in the field of real numbers, this number would be infinite), and thus the secret α is not uniquely determined.

The intuition behind the Shamir's protocol is illustrated in Figure 1.4.

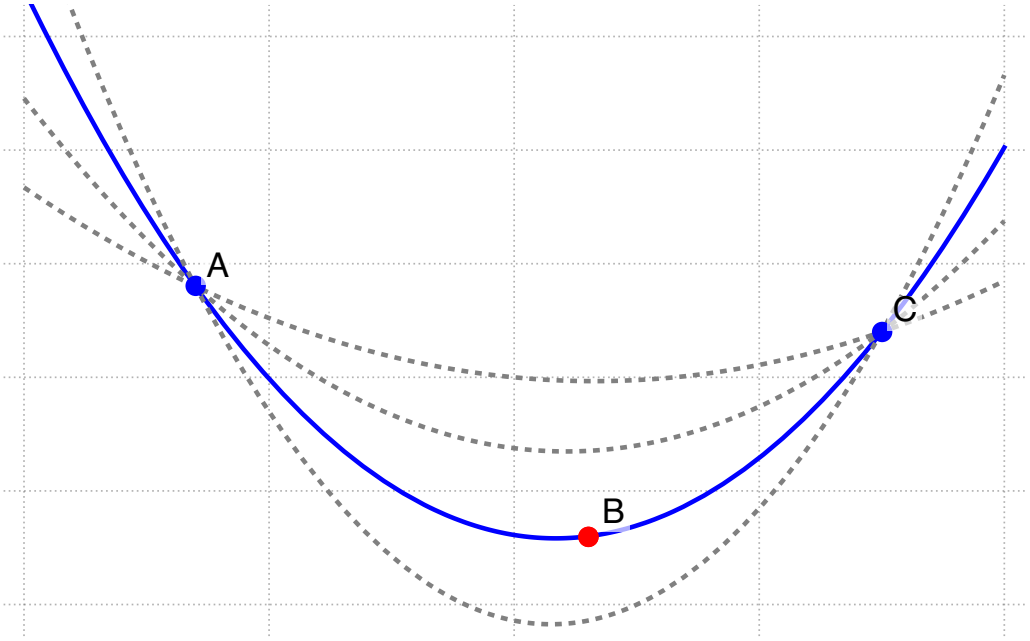


Figure 1.4: Suppose we have $t = 3$. Having only 2 points means knowing two blue points without knowing the red one. There are infinitely many quadratic polynomials passing through these two points (gray dashed lines). However, knowing the third red point allows us to uniquely determine the polynomial and thus get its value at 0. Note that this is illustrated over $\mathbb{R}$, but for $\mathbb{F}_q$ the logic is similar.

1.4.5 Some Fun: Group Implementation in Rust

In programming, we can think of a group as an interface, having a single binary operation defined, that obeys the rules of closure, associativity, identity element, and inverse element.

For that reason, we might even code a group in Rust! We will also write a simple test to check whether the group is valid and whether the group is abelian.

Trait for Group. First, we define a trait for a group. We will define a group as a trait with the following methods:

```rust
/// Trait that represents a group.
pub trait Group: Sized {
    /// Checks whether the two elements are equal.
    fn eq(&self, other: &Self) -> bool;
    /// Returns the identity element of the group.
    fn identity() -> Self;
    /// Adds two elements of the group.
    fn add(&self, a: &Self) -> Self;
    /// Returns the negative of the element.
    fn negate(&self) -> Self;
    /// Subtracts two elements of the group.
    fn sub(&self, a: &Self) -> Self {
        self.add(&a.negate())
    }
}
```

Checking group validity. Now observer the following: we get closure for free, since the compiler will check whether the return type of the operation is the same as the type of the group. However, there is no guarantee that associativity holds, and our identity element is at all valid. For that reason, we need to somehow additionally check the validity of implementation.

We propose to do the following: we will randomly sample three elements from the group $a, b, c \xleftarrow{R} \mathbb{G}$ and check our three properties:

1. $a \oplus (b \oplus c) \stackrel{?}{=} (a \oplus b) \oplus c$.
2. $a \oplus e \stackrel{?}{=} e \oplus a \stackrel{?}{=} a$.
3. $a \oplus (\ominus a) \stackrel{?}{=} (\ominus a) \oplus a \stackrel{?}{=} e$.

Additionally, if we want to verify whether the group is abelian, we can check whether $a \oplus b \stackrel{?}{=} b \oplus a$.

For that reason, for the check, we require the group to be samplable (i.e. we can randomly sample elements from the group):

```rust
/// Trait for sampling a random element from a group.
pub trait Samplable {
    /// Returns a random element from the group.
    fn sample() -> Self;
}
```

And now, our test looks as follows:

```rust
/// Number of tests to check the group properties.
const TESTS_NUMBER: usize = 100;

/// Asserts that the given group G is valid.
/// A group is valid if the following properties hold:
/// 1. Associativity: (a + b) + c = a + (b + c)
/// 2. Identity: a + e = a = e + a
/// 3. Inverse: a + (-a) = e = (-a) + a
pub fn assert_group_valid<G>()
where
    G: Group + Samplable,
{
    for _ in 0..TESTS_NUMBER {
        // Take random three elements
        let a = G::sample();
        let b = G::sample();
        let c = G::sample();

        // Check whether associativity holds
        let ab_c = a.add(&b).add(&c);
        let a_bc = a.add(&b.add(&c));
        let associativity_holds = ab_c.eq(&a_bc);
        assert!(associativity_holds, "Associativity does not hold
        ↪ for the given group");

```

```
        // Check whether identity element is valid
        let e = G::identity();
        let ae = a.add(&e);
        let ea = e.add(&a);
        let identity_holds = ae.eq(&a) && ea.eq(&a);
        assert!(identity_holds, "Identity element does not hold for
        ↪ the given group");

        // Check whether inverse element is valid
        let a_neg = a.negate();
        let a_neg_add_a = a_neg.add(&a);
        let a_add_a_neg = a.add(&a_neg);
        let inverse_holds = a_neg_add_a.eq(&e) && a_add_a_neg.eq(&e);
        assert!(inverse_holds, "Inverse element does not hold for
        ↪ the given group");
    }
}

/// Asserts that the given group G is abelian.
/// A group is an abelian group if the following property holds:
/// a + b = b + a for all a, b in G (commutativity)
pub fn assert_group_abelian<G>()
where
    G: Group + Samplable,
{
    for _ in 0..TESTS_NUMBER {
        assert_group_valid::<G>();

        // Take two random elements
        let a = G::sample();
        let b = G::sample();

        // Check whether commutativity holds
        let ab = a.add(&b);
        let ba = b.add(&a);
        assert!(ab.eq(&ba), "Commutativity does not hold for the
        ↪ given group");
    }
}
```

Testing the group $(\mathbb{Z}, +)$**.** And now, we can define a group for integers and check whether it is valid and abelian:

```
use crate::group::{Group, Samplable};
use rand::Rng;

/// Implementing group for Rotation3<f32>
impl Group for i64 {
    fn eq(&self, other: &Self) -> bool {
```

```
        self == other
    }

    fn identity() -> Self {
        0i64
    }

    fn add(&self, a: &Self) -> Self {
        self + a
    }

    fn negate(&self) -> Self {
        -self
    }
}

impl Samplable for i64 {
    fn sample() -> Self {
        let mut gen = rand::thread_rng();

        // To prevent overflow, we choose a smaller range for i64
        let min = i64::MIN / 3;
        let max = i64::MAX / 3;
        gen.gen_range(min..max)
    }
}
```

Just a small note: since we cannot generate infinite integers, we restrict the range of integers to prevent overflow. So, for the sake of simplicity, we divide the range of integers by 3, in which overflow never occurs.

And now, the moment of truth! Let us define some tests and run them:

```
#[cfg(test)]
mod tests {
    use super::*;
    use group::*;

    #[test]
    fn test_integers_are_group() {
        assert_group_valid::<i64>()
    }

    #[test]
    fn test_integers_are_abelian() {
        assert_group_abelian::<i64>();
    }
}
```

Both tests pass! Now let us consider something a bit trickier.

Testing the group SO(3). We can define a group for 3×3 rotation matrices. Of course, composition of two rotation is not commutative, so we expect the abelian test to fail. However, the group is still valid! For example, there is an identity rotation matrix E, and for each rotation matrix $A \in \mathrm{SO}(3)$, there exists a rotation matrix $A^{-1} \in \mathrm{SO}(3)$ such that $AA^{-1} = A^{-1}A = E$. Finally, the associativity holds as well.

We will use the `nalgebra` library for this purpose, which contains the implementation of rotation matrices. So our implementation can look as follows:

```rust
/// A threshold below which two floating point numbers are
↪ considered equal.
const EPSILON: f32 = 1e-6;

/// Implementing group for Rotation3<f32>
impl Group for Rotation3<f32> {
    fn eq(&self, other: &Self) -> bool {
        // Checking whether the norm of a difference is small
        let difference = self.matrix() - other.matrix();
        difference.norm_squared() < EPSILON
    }

    fn identity() -> Self {
        Rotation3::identity()
    }

    fn add(&self, a: &Self) -> Self {
        self * a
    }

    fn negate(&self) -> Self {
        self.inverse()
    }
}

impl Samplable for Rotation3<f32> {
    fn sample() -> Self {
        let mut gen = rand::thread_rng();

        // Pick three random angles
        let roll = gen.gen_range(0.0..1.0);
        let pitch = gen.gen_range(0.0..1.0);
        let yaw = gen.gen_range(0.0..1.0);

        Rotation3::from_euler_angles(roll, pitch, yaw)
    }
}
```

Here, there are two tricky moments:

1. We cannot compare floating point numbers directly, since they might differ by a small

amount. For that reason, we define a small threshold ε. We say that two matrices are equal iff the norm[3] of their difference is less than ε.

2. To generate a random rotation matrix, we generate three random angles and create a rotation matrix from these angles.

1.5 Exercises

Exercise 1. Which of the following statements is **false**?

1. $(\forall a, b \in \mathbb{Q}, a \neq b)\ (\exists q \in \mathbb{R}) : \{a < q < b\}$.
2. $(\forall \varepsilon > 0)\ (\exists n_\varepsilon \in \mathbb{N})\ (\forall n \geq n_\varepsilon) : \{1/n < \varepsilon\}$.
3. $(\forall k \in \mathbb{Z})\ (\exists n \in \mathbb{N}) : \{n < k\}$.
4. $(\forall x \in \mathbb{Z} \setminus \{-1\})\ (\exists! y \in \mathbb{Q}) : \{(x+1)y = 2\}$.

Exercise 2. Denote $X := \{(x, y) \in \mathbb{Q}^2 : xy = 1\}$. Oleksandr claims the following:

1. $X \cap \mathbb{N}^2 = \{(1, 1)\}$.
2. $|X \cap \mathbb{Z}^2| = 2|X \cap \mathbb{N}^2|$.
3. X is a group under the operation $(x_1, y_1) \oplus (x_2, y_2) = (x_1 x_2, y_1 y_2)$.

Which statements are **true**?

a) Only 1.
b) Only 1 and 2.
c) Only 1 and 3.
d) Only 2 and 3.
e) All statements are correct.

Exercise 3. Does a tuple $(\mathbb{Z}, \oplus)$ with operation $a \oplus b = a + b - 1$ define a group?

a) Yes, and this group is abelian.
b) Yes, but this group is not abelian.
c) No, since the associativity property does not hold.
d) No, since there is no identity element in this group.
e) No, since there is no inverse element in this group.

Exercise 4. Consider the Cartesian plane $\mathbb{R}^2$, where two coordinates are real numbers. For two points A, B define the operation $\oplus$ as follows: $A \oplus B$ is the midpoint on segment AB. Does $(\mathbb{R}^2, \oplus)$ define a group?

a) Yes, and this group is abelian.
b) Yes, but this group is not abelian.
c) No, since the associativity property does not hold and there is no identity element in this group.
d) No, since the associativity property does not hold, but we might define an identity element nonetheless.

Exercise 5. Find the inverse of 4 in $\mathbb{F}_{11}$.

a) 8
b) 5

[3] one can think of norm as being the measure of "distance" between two objects. Similarly, we can define norm not only on matrices, but on vectors as well.

c) 3

d) 7

Exercise 6. Suppose for three polynomials $p, q, r \in \mathbb{F}[x]$ we have $\deg p = 3, \deg q = 4, \deg r = 5$. Which of the following is true for $n := \deg\{(p-q)r\}$?

a) $n = 9$.

b) n might be less than 9.

c) $n = 20$.

d) n is less than $\deg\{qr\}$.

Exercise 7. Define the polynomial over $\mathbb{F}_5$: $f(x) := 4x^2 + 7$. Which of the following is the root of $f(x)$?

a) 2

b) 3

c) 4

d) This polynomial has no roots over $\mathbb{F}_5$.

Exercise 8. Quadratic polynomial $p(x) = ax^2 + bx + c \in \mathbb{R}[x]$ has zeros at 1 and 2 and $p(0) = 2$. Find the value of $a + b + c$.

a) 0

b) -1

c) 1

d) Not enough information to determine.

Exercise 9. Which of the following is a **valid** endomorphism $f : X \to X$?

a) $X = [0, 1]$, $f : x \mapsto x^2$.

b) $X = [0, 1]$, $f : x \mapsto x + 1$.

c) $X = \mathbb{R}_{>0}$, $f : x \mapsto (x-1)^3$.

d) $X = \mathbb{Q}_{>0}$, $f : x \mapsto \sqrt{x}$.

Exercise 10*. Denote by $\mathrm{GL}(2, \mathbb{R})$ a set of 2×2 invertable matrices with real entries. Define two functions $\varphi : \mathrm{GL}(2, \mathbb{R}) \to \mathbb{R}$:

$$\varphi_1\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = ad - bc, \ \varphi_2\left(\begin{bmatrix} a & b \\ c & d \end{bmatrix}\right) = a + d \tag{23}$$

Den claims the following:

1. φ_1 is a group homomorphism between multiplicative groups $(\mathrm{GL}(2, \mathbb{R}), \times)$ and $(\mathbb{R}, \times)$.
2. φ_2 is a group homomorphism between additive groups $(\mathrm{GL}(2, \mathbb{R}), +)$ and $(\mathbb{R}, +)$.

Which of the following is **true**?

a) Only statement 1 is correct.

b) Only statement 2 is correct.

c) Both statements 1 and 2 are correct.

d) None of the statements is correct.

2 Basics of Security Analysis

2.1 Basics of Security Analysis

In many cases, technical papers include the analysis on the key question: "How secure is this cryptographic algorithm?" or rather "Why this cryptographic algorithm is secure?". In this section, we will shortly describe the notation and typical construction for justifying the security of cryptographic algorithms.

Typically, the cryptographic security is defined in a form of a game between the adversary (who we call $\mathcal{A}$) and the challenger (who we call $\mathcal{C}h$). The adversary is trying to break the security of the cryptographic algorithm using arbitrary (but still efficient) protocol, while the challenger is following a simple, fixed protocol. The game is played in a form of a challenge, where the adversary is given some information and is asked to perform some task. The security of the cryptographic algorithm is defined based on the probability of the adversary to win the game.

2.1.1 Cipher Semantic Security

Let us get into specifics. Suppose that we want to specify that the encryption scheme is secure. Recall that cipher $\mathcal{E} = (E, D)$ over the space $(\mathcal{K}, \mathcal{M}, \mathcal{C})$ (here, $\mathcal{K}$ is the space containing all possible keys, $\mathcal{M}$ – all possible messages and $\mathcal{C}$ – all possible ciphers) consists of two efficiently computable methods:

- $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ – encryption method, that based on the provided message $m \in \mathcal{M}$ and key $k \in \mathcal{K}$ outputs the cipher $c = E(k, m) \in \mathcal{C}$.
- $D : \mathcal{K} \times \mathcal{C} \to \mathcal{M}$ – decryption method, that based on the provided cipher $c \in \mathcal{C}$ and key $k \in \mathcal{K}$ outputs the message $m = D(k, c) \in \mathcal{M}$.

Of course, we require the **correctness**:

$$(\forall k \in \mathcal{K})\,(\forall m \in \mathcal{M}) : \{D(k, E(k, m)) = m\} \tag{24}$$

Now let us play the following game between adversary $\mathcal{A}$ and challenger $\mathcal{C}h$:

1. $\mathcal{A}$ picks any two messages $m_0, m_1 \in \mathcal{M}$ on his choice.
2. $\mathcal{C}h$ picks a random key $k \xleftarrow{R} \mathcal{K}$ and random bit $b \xleftarrow{R} \{0, 1\}$ and sends the cipher $c = E(k, m_b)$ to $\mathcal{A}$.
3. $\mathcal{A}$ is trying to guess the bit b by using the cipher c.
4. $\mathcal{A}$ outputs the guess $\hat{b}$.

Now, what should happen if our encryption scheme is secure? The adversary should not be able to guess the bit b with a probability significantly higher than $1/2$ (a random guess). Formally, define the **advantage** of the adversary $\mathcal{A}$ as:

$$\mathrm{SSAdv}[\mathcal{E}, \mathcal{A}] := \left| \Pr\left[\hat{b} = b\right] - \frac{1}{2} \right| \tag{25}$$

We say that the encryption scheme is **semantically secure**[4] if for any efficient adversary $\mathcal{A}$ the advantage $\mathrm{SSAdv}[\mathcal{A}]$ is negligible. In other words, the adversary cannot guess the bit b with a probability significantly higher than $1/2$.

[4] This version of definition is called a **bit-guessing** version.

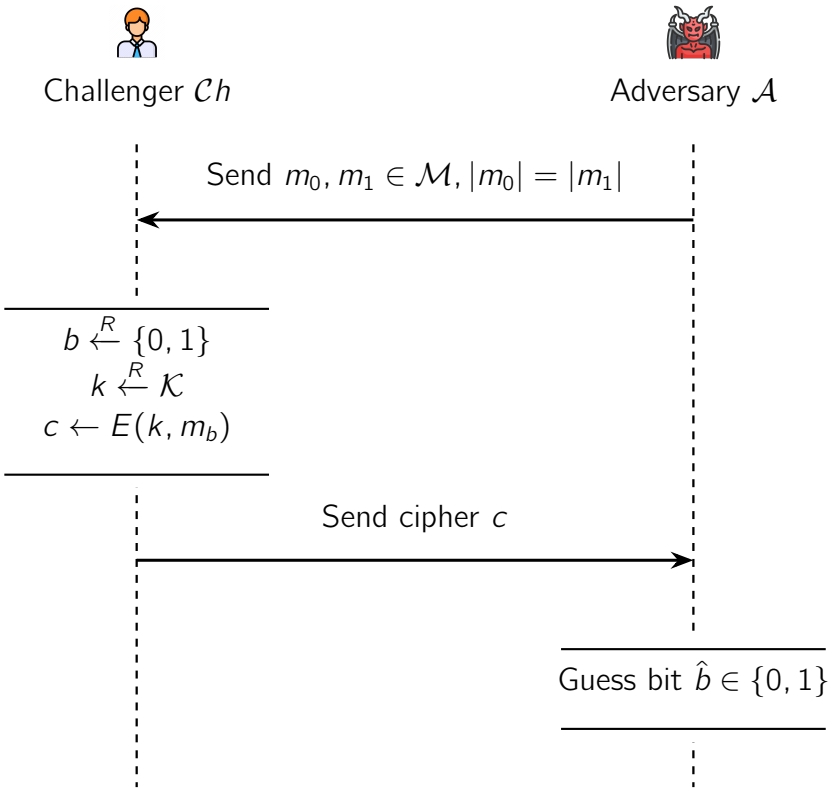


Figure 2.1: The game between the adversary $\mathcal{A}$ and the challenger $\mathcal{C}h$ for defining the semantic security.

Now, what negligible means? Let us give the formal definition!

Definition 2.1. A function $f : \mathbb{N} \to \mathbb{R}$ is called **negligible** if for all $c \in \mathbb{R}_{>0}$ there exists $n_c \in \mathbb{N}$ such that for any $n \geq n_c$ we have $|f(n)| < 1/n^c$.

The alternative definition, which is problably easier to interpret, is the following.

Theorem 2.2. A function $f : \mathbb{N} \to \mathbb{R}$ is **negligible** if and only if for any $c \in \mathbb{R}_{>0}$, we have

$$\lim_{n\to\infty} f(n)n^c = 0 \tag{26}$$

Example. The function $f(n) = 2^{-n}$ is negligible since for any $c \in \mathbb{R}_{>0}$ we have

$$\lim_{n\to\infty} 2^{-n}n^c = 0 \tag{27}$$

The function $g(n) = \frac{1}{n!}$ is also negligible for similar reasons.

Example. The function $h(n) = \frac{1}{n}$ is not negligible since for $c = 1$ we have

$$\lim_{n\to\infty} \frac{1}{n} \times n = 1 \neq 0 \tag{28}$$

Well, that is weird. For some reason we are considering a function the depends on some natural number n, but what is this number?

Typically, when defining the security of the cryptographic algorithm, we are considering the security parameter λ (e.g., the length of the key). The function is negligible if the probability of the adversary to break the security of the cryptographic algorithm is decreasing with the increasing of the security parameter λ. Moreover, we require that the probability of the adversary to break the security of the cryptographic algorithm is decreasing faster than any polynomial function of the security parameter λ.

So all in all, we can define the semantic security as follows.

Definition 2.3. The encryption scheme $\mathcal{E}$ with a security paramter $\lambda \in \mathbb{N}$ is **semantically secure** if for any efficient adversary $\mathcal{A}$ we have:

$$\left| \Pr\left[b = \hat{b} \;\middle|\; \begin{array}{c} m_0, m_1 \leftarrow \mathcal{A},\ k \xleftarrow{R} \mathcal{K},\ b \xleftarrow{R} \{0,1\} \\ c \leftarrow E(k, m_b) \\ \hat{b} \leftarrow \mathcal{A}(c) \end{array} \right] - \frac{1}{2} \right| < \mathrm{negl}(\lambda) \tag{29}$$

Do not be afraid of such complex notation, it is quite simple. Notation $\Pr[A \mid B]$ means "the probability of A, given that B occurred". So our inner probability is read as "the probability that the guessed bit $\hat{b}$ equals b given the setup on the right". Then, on the right we define the setup: first we generate two messages $m_0, m_1 \in \mathcal{M}$, then we choose a random bit b and a key k, cipher the message m_b, send it to the adversary and the adversary, based on provided cipher, gives $\hat{b}$ as an output. We then claim that the probability of the adversary to guess the bit b is close to $1/2$.

Let us see some more examples of how to define the security of certain crypographic objects.

2.1.2 Discrete Logarithm Assumption (DL)

Now, let us define the fundamental assumption used in cryptography formally: the **Discrete Logarithm Assumption** (DL).

Definition 2.4. Assume that $\mathbb{G}$ is a cyclic group of prime order r generated by $g \in \mathbb{G}$. Define the following game:

1. Both challenger $\mathcal{C}h$ and adversary $\mathcal{A}$ take a description $\mathbb{G}$ as an input: order r and generator $g \in \mathbb{G}$.
2. $\mathcal{C}h$ computes $\alpha \xleftarrow{R} \mathbb{Z}_r$, $u \leftarrow g^\alpha$ and sends $u \in \mathbb{G}$ to $\mathcal{A}$.
3. The adversary $\mathcal{A}$ outputs $\hat{\alpha} \in \mathbb{Z}_r$.

We define $\mathcal{A}$'s **advantage in solving the discrete logarithm problem in** $\mathbb{G}$, denoted as $\mathsf{DLadv}[\mathcal{A}, \mathbb{G}]$, as the probability that $\hat{\alpha} = \alpha$.

Definition 2.5. The **Discrete Logarithm Assumption** holds in the group $\mathbb{G}$ if for any efficient adversary $\mathcal{A}$ the advantage $\mathsf{DLadv}[\mathcal{A}, \mathbb{G}]$ is negligible.

Informally, this assumption means that given u, it is very hard to find α such that $u = g^\alpha$. But now we can write down this formally!

2.1.3 Computational Diffie-Hellman (CDH)

Another fundamental problem in cryptography is the **Computational Diffie-Hellman** (CDH) problem. It states that given g^α, g^β it is hard to find $g^{\alpha\beta}$. This property is frequently used in the construction of cryptographic protocols such as the Diffie-Hellman key exchange.

Let us define this problem formally.

Definition 2.6. Let $\mathbb{G}$ be a cyclic group of prime order r generated by $g \in \mathbb{G}$. Define the following game:

1. Both challenger $\mathcal{C}h$ and adversary $\mathcal{A}$ take a description $\mathbb{G}$ as an input: order r and generator $g \in \mathbb{G}$.
2. $\mathcal{C}h$ computes $\alpha, \beta \xleftarrow{R} \mathbb{Z}_r$, $u \leftarrow g^\alpha$, $v \leftarrow g^\beta$, $w \leftarrow g^{\alpha\beta}$ and sends $u, v \in \mathbb{G}$ to $\mathcal{A}$.
3. The adversary $\mathcal{A}$ outputs $\hat{w} \in \mathbb{G}$.

We define $\mathcal{A}$'s **advantage in solving the computational Diffie-Hellman problem in** $\mathbb{G}$, denoted as $\mathsf{CDHadv}[\mathcal{A}, \mathbb{G}]$, as the probability that $\hat{w} = w$.

Definition 2.7. The **Computational Diffie-Hellman Assumption** holds in the group $\mathbb{G}$ if for any efficient adversary $\mathcal{A}$ the advantage $\mathsf{CDHadv}[\mathcal{A}, \mathbb{G}]$ is negligible.

2.2 Decisional Diffie-Hellman (DDH)

Now, we loosen the requirements a bit. The **Decisional Diffie-Hellman** (DDH) problem states that given $g^\alpha, g^\beta, g^{\alpha\beta}$ it is "hard" to distinguish $g^{\alpha\beta}$ from a random element in $\mathbb{G}$. Formally, we define this problem as follows.

Definition 2.8. Let $\mathbb{G}$ be a cyclic group of prime order r generated by $g \in \mathbb{G}$. Define the following game:

1. Both challenger $\mathcal{C}h$ and adversary $\mathcal{A}$ take a description $\mathbb{G}$ as an input: order r and generator $g \in \mathbb{G}$.
2. $\mathcal{C}h$ computes $\alpha, \beta, \gamma \xleftarrow{R} \mathbb{Z}_r$, $u \leftarrow g^\alpha$, $v \leftarrow g^\beta$, $w_0 \leftarrow g^{\alpha\beta}$, $w_1 \leftarrow g^\gamma$. Then, $\mathcal{C}h$ flips a coin $b \xleftarrow{R} \{0, 1\}$ and sends u, v, w_b to $\mathcal{A}$.
3. The adversary $\mathcal{A}$ outputs the predicted bit $\hat{b} \in \{0, 1\}$.

We define $\mathcal{A}$'s **advantage in solving the Decisional Diffie-Hellman problem in** $\mathbb{G}$, denoted as $\mathsf{DDHadv}[\mathcal{A}, \mathbb{G}]$, as

$$\mathsf{DDHadv}[\mathcal{A}, \mathbb{G}] := \left| \Pr\left[b = \hat{b}\right] - \frac{1}{2} \right| \tag{30}$$

Now, let us break this assumption for some quite generic group! Consider the following example.

Theorem 2.9. Suppose that $\mathbb{G}$ is a cyclic group of an even order. Then, the Decision Diffie-Hellman Assumption does not hold in $\mathbb{G}$. In fact, there is an efficient adversary $\mathcal{A}$ that can distinguish $g^{\alpha\beta}$ from a random element in $\mathbb{G}$ with an advantage $1/4$.

Proof. If $|\mathbb{G}| = 2n$ for $n \in \mathbb{N}$, it means that we can split the group into two subgroups

of order n, say, $\mathbb{G}_1$ and $\mathbb{G}_2$. The first subgroup consists of elements in a form g^{2k}, while the second subgroup consists of elements in a form g^{2k+1}.

Now, if we could efficiently determine, based on group element $g \in \mathbb{G}$, whether $g \in \mathbb{G}_1$ or $g \in \mathbb{G}_2$, we essentially could solve the problem. Fortunately, there is such a method! Consider the following lemma.

Lemma 2.10. Suppose $u = g^{\alpha}$. Then, α is even if and only if $u^n = 1$.

Proof. If α is even, then $\alpha = 2\alpha'$ and thus

$$u^n = (g^{2\alpha'})^n = g^{2n\alpha'} = (g^{2n})^{\alpha'} = 1^{\alpha'} = 1 \tag{31}$$

Conversely, if $u^n = 1$ then $u^{\alpha n} = 1$, meaning that $2n \mid \alpha n$, implying that α is even. Lemma is proven.

Now, we can construct our adversary $\mathcal{A}$ as follows. Suppose $\mathcal{A}$ is given (u, v, w). Then,

1. Based on u, get the parity of α, say $p_\alpha \in \{\text{even}, \text{odd}\}$.
2. Based on v, get the parity of β, say $p_\beta \in \{\text{even}, \text{odd}\}$.
3. Based on w, get the parity of γ, say $p_\gamma \in \{\text{even}, \text{odd}\}$.
4. Calculate $p'_\gamma \in \{\text{even}, \text{odd}\}$ — parity of $\alpha\beta$.
5. Return $\hat{b} = 0$ if $p'_\gamma = p_\gamma$, and $\hat{b} = 1$, otherwise.

Suppose γ is indeed $\alpha \times \beta$. Then, condition $p'_\gamma = p_\gamma$ will always hold. If γ is a random element, then the probability that $p'_\gamma = p_\gamma$ is $1/2$. Therefore, the probability that $\mathcal{A}$ will guess the bit b correctly is $3/4$, and the advantage is $1/4$ therefore. ■

2.2.1 Why this is needed?

Typically, it is impossible to prove the predicate "for every efficient adversary $\mathcal{A}$ this probability is negligible" and therefore we need to make assumptions, such as the Discrete Logarithm Assumption or the Computational Diffie-Hellman Assumption. In turn, proving the statement "if X is secure then Y is also secure" is manageable and does not require solving any fundamental problems. So, for example, knowing that the probability of the adversary to break the Diffie-Hellman assumption is negligible, we can prove that the Diffie-Hellman key exchange is secure.

2.3 Basic Number Theory

2.3.1 Primes

Primes are often used when doing almost any cryptographic computation. A prime number is a natural number ($\mathbb{N}$) that is not a product of two smaller natural number. In other words, the prime number is divisible only by itself and 1. The first primes are: 2, 3, 5, 7, 11...

2.3.2 Deterministic prime tests

A primality test is deterministic if it outputs `True` when the number is a prime and `False` when the input is composite with probability 1. An example of a deterministic prime test is `Trial_Division_Test`. Here is an example implementation in Rust:

```
1     fn is_prime(n: u32) -> bool {
2         let square_root = (n as f64).sqrt() as u32;
3
```

```
        for i in 2.. = square_root {
            if n % i == 0 {
                return false;
            }
        }

        true
    }
```

Deterministic tests often lack efficiency. For instance, even with square root optimization, the asymptotic complexity is $O(\sqrt{N})$. While further optimizations are possible, they do not change the overall asymptotic complexity.

In cryptography, N can be extremely large — 256 bits, 512 bits, or even 6144 bits. An algorithm is impractical when dealing with such large numbers.

2.3.3 Probabilistic prime tests

A primality test is probabilistic if it outputs `True` when the number is a prime and `False` when the input is composite with probability less than 1. Such test is often called a pseudoprimality test. Fermat Primality and Miller-Rabin Primality Tests are examples of probabilistic primality test. Both of them use the idea of **Fermat's Little Theorem**:

> **Theorem 2.11.** Let p be a prime number and a be an integer not divisible by p. Then $a^{p-1} - 1$ is always divisible by p: $a^{p-1} \equiv 1 \pmod{p}$

The key idea behind the Fermat Primality Test is that if for some a not divisible by n we have $a^{n-1} \not\equiv 1 \pmod{n}$ then n is definitely NOT prime. Athough, with such an approach, we might get a false positive, as you cannot state for sure that n is prime. For example, consider $n = 15$ and $a = 4$. $4^{15-1} \equiv 1 \pmod{15}$, but $n = 15 = 3 \cdot 5$ is composite. To solve this issue, a is picked many times, decreasing the chances of a false positive. The probability that a composite number is mistakenly called prime for k iterations is $2^{-k} = \frac{1}{2^k}$.

There exists a problem with such an algorithm in the form of **Carmichael numbers**, which are numbers that are Fermat pseudoprime to all bases. To put it simply, no matter how many times you check whether the number is prime using this type of primality test, it will always stay positive, even though the number is composite. The good thing is that Carmichael numbers are pretty rare. The bad thing is that there are infinitely many of them.

Even though this algorithm is probabilistic (which does not guarantee the correctness of the output) and has a vulnerability in the form of *Carmichael numbers*, it runs with an asymptotic complexity $O(\log^3 n)$. This is much better for large numbers and is often used in cryptography. Here is a pseudocode implementation of this algorithm:

```
# n = number to be tested for primality
# k = number of times the test will be repeated
def is_prime(n, k):
    i = 1
    while i <= k:
        a = rand(2, n - 1)

```

```
        if a^(n - 1) != 1 (mod n):
            return False

        i++

    return True
```

Miller-Rabin primality test, is a more advanced form of Fermat primality test. The main difference is it is not vulnerable to *Carmichael numbers*, which makes it much better to use in practice.

2.3.4 Greatest Common Divisor

Greatest common divisor (GCD) of two or more integers, which are not all zero, is the largest positive integer that divides each of the integers.

> **Example.** $gcd(8, 12) = 4$, $gcd(3, 15) = 3$, $gcd(15, 10) = 5$.

Computing GCD using Euclid's algorithm. The is based on the fact that, given two positive integers a and b such that $a > b$, the common divisors of a and b are the same as the common divisors of $a - b$ and b. It can be observed, that it can be further optimized, by using $amodb$, instead of $a - b$. For example, $gcd(26, 8) = gcd(18, 8) = gcd(10, 8) = gcd(2, 8)$ can be optimized to $gcd(26, 8) = gcd(26 \pmod 8, 8) => gcd(2, 8)$ Algorithm can be implemented using recursion. Base of the recursion is $gcd(a, 0) = a$.

```
int gcd(a, b):
 if (b == 0):
     return a
 return gcd(b, a % b)
```

Provided algorithm work with $O(log(N))$ asymptotic complexity.

2.3.5 Least common multiple

Least common multiple (LCM) of two integers a and b, is the smallest positive integer that is divisible by both a and b.

The least common multiple can be computed from the greatest common divisor with the formula: $lcm(a, b) = \frac{|ab|}{gcd(a,b)}$

```
int lcm(a, b):
 return a * (b / gcd(a, b))
```

2.3.6 Modular inverse

Modular multiplicative inverse of an integer a is an integer b such that $a \cdot b \equiv 1 \pmod m$. In prime fields it is commonly used as a division operation.

One of the ways to compute the modular inverse is by using Euler's theorem:

$a^{\phi(m)} \equiv 1 \pmod m$, where ϕ is Euler's totient function.

For prime numbers, where $\phi(m) = m - 1$:

$a^{m-2} \equiv a^{-1} \pmod m$.

```
a_inverse = powmod(a, m-2, m) # where powmod(base, power,
↪ modulus)
```

2.3.7 Reed-Solomon codes

Reed-Solomon codes allows to restore lost or corrupted data, implement threshold secret sharing and is used in some ZK protocols. Given a vector of data V a polynomial P is constructed using Lagrange interpolation. Polynomial with degree n can be uniquely defined using $(n+1)$ unique points. Defining more points on the same polynomial add a redundancy, which can be used to restore the polynomial even if some points are missing. Common choices for a set of evaluation points include $0, 1, 2, ..., n-1$.

The error-correcting ability of a Reed-Solomon code is $n-k$, the measure of redundancy in the block. If the locations of the error symbols are not known in advance, then a Reed-Solomon code can correct up to $n-k/2$ erroneous symbols, i.e., it can correct half as many errors as there are redundant symbols added to the block.

2.3.8 Schwartz-Zippel Lemma

Lemma 2.12. Let $\mathbb{F}$ be a field. Let $f(x_1, x_2, ..., x_n)$ be a polynomial of total degree d. Suppose that f is not the zero polynomial. Let S be a finite subset of $\mathbb{F}$. Let $r_1, r_2, ...r_n$ be chosen at random uniformly and independently from S. Then the probability that $f(r_1, r_2, ..., r_n) = 0$ is $\leq \frac{d}{|S|}$.

Example. Let $F = \mathbb{F}_3$, $f(x) = x^2 - 5x + 6$, $S = F$, $r \xleftarrow{R} \mathbb{F}_3$.
Schwartz-Zippel lemma says that the probability that $f(r) = 0$ is $\leq \frac{2}{3}$.

Given two polynomials P, Q with degree d in a field $\mathbb{F}_p$, for $r \xleftarrow{R} \mathbb{F}_3$: $\Pr[P(r) == Q(r)] \leq \frac{d}{p}$. For large fields, where $\frac{d}{p}$ is negligible, this property allows to succinctly check the equality of polynomials. Let $H(x) := P(x) - Q(x)$. Than for each $P(x) = Q(x) \rightarrow H(x) = 0$. Applying Schwartz-Zippel lemma, the probability of $H(x) = 0$ for $x \xleftarrow{R} \mathbb{F}$ is $\leq \frac{d}{|S|}$.

2.4 Exercises

Exercise 1. Suppose that for the given cipher with a security parameter λ, the adversary $\mathcal{A}$ can deduce the least significant bit of the plaintext from the ciphertext. Recall that the advantage of a bit-guessing game is defined as $\mathsf{SSAdv}[\mathcal{A}] = \left|\Pr\left[b = \hat{b}\right] - \frac{1}{2}\right|$, where b is the randomly chosen bit of a challenger, while $\hat{b}$ is the adversary's guess. What is the maximal advantage of $\mathcal{A}$ in this case?

Hint: The adversary can choose which messages to send to challenger to further distinguish the plaintexts.

a) 1
b) $\frac{1}{2}$
c) $\frac{1}{4}$
d) 0
e) Negligible value (negl(λ)).

Exercise 2. Consider the cipher $\mathcal{E} = (E, D)$ with encryption function $E : \mathcal{K} \times \mathcal{M} \to \mathcal{C}$ over the message space $\mathcal{M}$, ciphertext space $\mathcal{C}$, and key space $\mathcal{K}$. We want to define the security that, based on the cipher, the adversary $\mathcal{A}$ cannot restore the message (*security against message recovery*). For that reason, we define the following game:

1. Challenger chooses random $m \xleftarrow{R} \mathcal{M}, k \xleftarrow{R} \mathcal{K}$.
2. Challenger computes the ciphertext $c \leftarrow E(k, m)$ and sends to $\mathcal{A}$.
3. Adversary outputs $\hat{m}$, and wins if $\hat{m} = m$.

We say that the cipher $\mathcal{E}$ is secure against message recovery if the **message recovery advantage**, denoted as $\mathrm{MRadv}[\mathcal{A}, \mathcal{E}]$ is negligible. Which of the following statements is a valid interpretation of the message recovery advantage?

a) $\mathrm{MRadv}[\mathcal{A}, \mathcal{E}] := \left|\Pr[m = \hat{m}] - \frac{1}{2}\right|$
b) $\mathrm{MRadv}[\mathcal{A}, \mathcal{E}] := |\Pr[m = \hat{m}] - 1|$.
c) $\mathrm{MRadv}[\mathcal{A}, \mathcal{E}] := \Pr[m = \hat{m}]$
d) $\mathrm{MRadv}[\mathcal{A}, \mathcal{E}] := \left|\Pr[m = \hat{m}] - \frac{1}{|\mathcal{M}|}\right|$

Exercise 3. Suppose that f and g are negligible functions. Which of the following functions is not neccessarily negligible?

a) $f + g$
b) $f \times g$
c) $f - g$
d) f/g
e) $h(\lambda) := \begin{cases} 1/f(\lambda) & \text{if } 0 < \lambda < 100000 \\ g(\lambda) & \text{if } \lambda \geq 100000 \end{cases}$

Exercise 4. Suppose that $f \in \mathbb{F}_p[x]$ is a d-degree polynomial with d **distinct** roots in $\mathbb{F}_p$. What is the probability that, when evaluating f at n random points, the polynomial will be zero at all of them?

a) Exactly $(d/p)^n$.
b) Strictly less that $(d/p)^n$.
c) Exactly nd/p.
d) Exactly d/np.

Exercise 5-6. To demonstrate the idea of Reed-Solomon codes, consider the toy construction. Suppose that our message is a tuple of two elements $a, b \in \mathbb{F}_{13}$. Consider function $f : \mathbb{F}_{13} \to \mathbb{F}_{13}$, defined as $f(x) = ax + b$, and define the encoding of the message (a, b) as $(a, b) \mapsto (f(0), f(1), f(2), f(3))$.

Question 5. Suppose that you received the encoded message $(3, 5, 6, 9)$. Which number from the encoded message is corrupted?

a) First element (3).
b) Second element (5).
c) Third element (6).
d) Fourth element (9).
e) The message is not corrupted.

Question 6. Consider the previous question. Suppose that the original message was (a, b). Find the value of $a \times b$ (in $\mathbb{F}_{13}$).

a) 4
b) 6
c) 12
d) 2
e) 1

3 Field Extensions and Elliptic Curves

3.1 Finite Field Extensions

3.1.1 General Definition

Previously, our discussion resolved around the finite field $\mathbb{F}_p$ for a prime p. However, many protocols need more than just a prime field. For example, elliptic curve pairings and certain STARK constructions require extending $\mathbb{F}_p$ to, in a sense, the analogous of complex numbers.

From school and, possibly, university, you might remember how complex numbers $\mathbb{C}$ are constructed. You take two real numbers, say, $x, y \in \mathbb{R}$, introduce a new symbol i satisfying $i^2 = -1$, and define the complex number as $z = x + iy$. In certain cases, one might encounter a bit more rigorous and abstract definition of complex numbers as the set of pairs $(x, y) \in \mathbb{R}^2$ where addition in naturally defined as $(x_1, y_1)+(x_2, y_2) = (x_1+x_2, y_1+y_2)$, and the multiplication is:

$$(x_1, y_1) \cdot (x_2, y_2) = (x_1x_2 - y_1y_2, x_1y_2 + x_2y_1)^5. \tag{32}$$

In spite of what interpretation you have seen, the complex number is just a tuple of two real numbers that satisfy a bit different rules of multiplication (since addition is typically defined in the same way). What is even more important to us, is that $\mathbb{C}$ is our first example of the so-called **field extension** of $\mathbb{R}$.

Formally, definition of the field extension is very straightforward:

Definition 3.1. Let $\mathbb{F}$ be a field and $\mathbb{K}$ be another field. We say that $\mathbb{K}$ is an **extension** of $\mathbb{F}$ if $\mathbb{F} \subset \mathbb{K}$ and we denote it as $\mathbb{K}/\mathbb{F}$.

Despite just a simplicity of the definition, the field extensions are a very powerful tool in mathematics. But first, let us consider a few non-trivial examples of field extensions.

Example. Denote by $\mathbb{Q}(\sqrt{2}) = \{x + y\sqrt{2} : x, y \in \mathbb{Q}\}$. This is a field extension of $\mathbb{Q}$. It is obvious that $\mathbb{Q} \subset \mathbb{Q}(\sqrt{2})$, but why is $\mathbb{Q}(\sqrt{2})$ a field? Addition and multiplication operations are obviously closed:

$$\begin{aligned}(x_1 + y_1\sqrt{2}) + (x_2 + y_2\sqrt{2}) &= (x_1 + x_2) + (y_1 + y_2)\sqrt{2},\\ (x_1 + y_1\sqrt{2}) \cdot (x_2 + y_2\sqrt{2}) &= (x_1x_2 + 2y_1y_2) + (x_1y_2 + x_2y_1)\sqrt{2}.\end{aligned} \tag{33}$$

But what about the inverse element? Well, here is the trick:

$$\frac{1}{x + y\sqrt{2}} = \frac{x - y\sqrt{2}}{(x + y\sqrt{2})(x - y\sqrt{2})} = \frac{x - y\sqrt{2}}{x^2 - 2y^2} = \frac{x}{x^2 - 2y^2} - \frac{y}{x^2 - 2y^2}\sqrt{2} \in \mathbb{Q}(\sqrt{2}). \tag{34}$$

[5] Notice that $(x_1 + iy_1)(x_2 + iy_2) = x_1x_2 + iy_2x_1 + iy_1x_2 + i^2y_1y_2 = (x_1x_2 - y_1y_2) + (x_1y_2 + x_2y_1)i$.

Example. Consider $\mathbb{Q}(\sqrt{2}, i) = \{a + bi : a, b \in \mathbb{Q}(\sqrt{2})\}$ where $i^2 = -1$. This is a field extension of $\mathbb{Q}(\sqrt{2})$ and, consequently, of $\mathbb{Q}$. The representation of the element is:

$$(a + b\sqrt{2}) + (c + d\sqrt{2})i = a + b\sqrt{2} + ci + d\sqrt{2}i \tag{35}$$

Showing that this is a field is a bit more tedious, but still straightforward. Suppose we take $\alpha + \beta i \in \mathbb{Q}(\sqrt{2}, i)$ with $\alpha, \beta \in \mathbb{Q}(\sqrt{2})$. Then:

$$\frac{1}{\alpha + \beta i} = \frac{\alpha - \beta i}{\alpha^2 + \beta^2} = \frac{\alpha}{\alpha^2 + \beta^2} - \frac{\beta}{\alpha^2 + \beta^2} i \tag{36}$$

Since $\mathbb{Q}(\sqrt{2})$ is a field, both $\frac{\alpha}{\alpha^2+\beta^2}$ and $\frac{\beta}{\alpha^2+\beta^2}$ are in $\mathbb{Q}(\sqrt{2})$, and, consequently, $\mathbb{Q}(\sqrt{2}, i)$ is a field as well.

Remark. Notice that basically, $\mathbb{Q}(\sqrt{2}, i)$ is just a linear combination of $\{1, \sqrt{2}, i, \sqrt{2}i\}$. This has a very important implication: $\mathbb{Q}(\sqrt{2}, i)$ is a four-dimensional vector space over $\mathbb{Q}$, where elements $\{1, \sqrt{2}, i, \sqrt{2}i\}$ naturally form **basis**. We are not going to use it implicitly, but this observation might make further discussion a bit more intuitive.

Remark. One might have defined $\mathbb{Q}(\sqrt{2}, i) = \{x + \sqrt{2}y : x, y \in \mathbb{Q}(i)\}$ instead. Indeed, $\mathbb{Q}(\sqrt{2})(i) = \mathbb{Q}(i)(\sqrt{2}) = \mathbb{Q}(\sqrt{2}, i)$.

3.1.2 Polynomial Quotient Ring

Now, we present a more general way to construct field extensions. Notice that when constructing $\mathbb{C}$, we used the magical element i that satisfies $i^2 = -1$. But here is another way how to think of it.

Consider the set of polynomials $\mathbb{R}[x]$, then I pick $p(x) := x^2 + 1 \in \mathbb{R}[x]$ and ask you to find roots of $p(x)$. Of course, you would claim "hey, this equation has no solutions over $\mathbb{R}$" and that is totally true. That is why mathematicians introduced a new element i that we formally called the root of $x^2 + 1$. Note however, that i is not a number in the traditional sense, but rather a fictional symbol that we artifically introduced to satisfy the equation.

Now, could we have picked another polynomial, say, $q(x) = x^2 + 4$? Sure! As long as its roots cannot be found in $\mathbb{R}$, we are good to go.

Example. Suppose β is the root of $q(x) := x^2 + 4$. Then we could have defined complex numbers as a set of $x + y\beta$ for $x, y \in \mathbb{R}$. In this case, multiplication, for example, would be defined a bit differently than in the case of $\mathbb{C}$:

$$(x_1 + y_1\beta) \cdot (x_2 + y_2\beta) = (x_1x_2 - 4y_1y_2) + (x_1y_2 + x_2y_1)\beta. \tag{37}$$

We shifted to the polynomial consideration for a reason: now, instead of considering the complex number $\mathbb{C}$ as "some" tuple of real numbers (c_0, c_1), now let us view it as a polynomial[6] $c_0 + c_1X$ modulo polynomial $X^2 + 1$.

[6] Here, we use X to represent the polynomial variable to avoid confusion with the notation $x + yi$.

Example. Indeed, take, for example, $p_1(X) := 1 + 2X$ and $p_2(X) := 2 + 3X$. Addition is performed as we are used to:

$$p_1 + p_2 = (1 + 2X) + (2 + 3X) = 3 + 5X, \tag{38}$$

but multiplication is a bit different:

$$p_1 p_2 = (1 + 2X) \cdot (2 + 3X) = 2 + 3X + 4X + 6X^2 = 6X^2 + 7X + 2. \tag{39}$$

Well, and what next? Recall that we are doing arithmetic modulo X^2+1 and for that reason, we divide the polynomial by $X^2 + 1$:

$$6X^2 + 7X + 2 = 6(X^2 + 1) + 7X - 4 \implies (6X^2 + 7X + 2) \bmod (X^2 + 1) = 7X - 4, \tag{40}$$

meaning that $p_1 p_2 = 7X - 4$. Oh wow, hold on! Let us come back to our regular complex number representation and multiply $(1 + 2i)(2 + 3i)$. We get $2 + 3i + 4i + 6i^2 = -4 + 7i$. That is exactly the same result if we change X to i above! In fact, what we have observed is the fact that our polynomial quotient ring $\mathbb{R}[X]/(X^2 + 1)$ is isomorphic to $\mathbb{C}$.

So, let us generalize this observation to any field $\mathbb{F}$ and any irreducible polynomial $\mu(x) \in \mathbb{F}[x]$.

Theorem 3.2. Let $\mathbb{F}$ be a field and $\mu(x)$ — irreducible polynomial over $\mathbb{F}$ (sometimes called a **reduction polynomial**). Consider a set of polynomials over $\mathbb{F}[x]$ modulo $\mu(x)$, formally denoted as $\mathbb{F}[x]/(\mu(x))$. Then, $\mathbb{F}[x]/(\mu(x))$ is a field.

Example. As we considered above, let $\mathbb{F} = \mathbb{R}$, $\mu(x) = x^2 + 1$, then $\mathbb{R}[X]/(X^2 + 1)$ (a set of polynomials modulo $X^2 + 1$) is a field.

Example. Suppose $\mathbb{F} = \mathbb{Q}$ and $\mu(x) := x^2 - 2$. Then, $\mathbb{Q}[X]/(X^2 - 2)$ is a field isomorphic to $\mathbb{Q}(\sqrt{2})$, considered above.

Example. Suppose $\mathbb{F} = \mathbb{Q}$ and $\mu(x) := (x^2 + 1)(x^2 - 2) = x^4 - x^2 - 2$. Then, $\mathbb{Q}[X]/(x^4 - x^2 - 2)$ is a field isomorphic to $\mathbb{Q}(\sqrt{2}, i)$.

Remark. Although we have not defined the isomorphism between two rings/fields, it is defined similarly to group isomorphism. Suppose we have fields $(\mathbb{F}, +, \times)$ and $(\mathbb{K}, \oplus, \otimes)$. Bijective function $\phi : \mathbb{F} \to \mathbb{K}$ is called an isomorphism if it preserves additive and multiplicative structures, that is for all $a, b \in \mathbb{F}$:

$$\begin{aligned} \phi(a + b) &= \phi(a) \oplus \phi(b), \\ \phi(a \times b) &= \phi(a) \otimes \phi(b). \end{aligned} \tag{41}$$

This theorem (aka definition) corresponds to viewing complex numbers as a polynomial quotient ring $\mathbb{R}[X]/(X^2 + 1)$. But, we can give a theorem (aka definition) for our classical representation via magical root i of $x^2 + 1$.

Theorem 3.3. Let $\mathbb{F}$ be a field and $\mu \in \mathbb{F}[X]$ is an irreducible polynomial of degree n and let $\mathbb{K} := \mathbb{F}[X]/(\mu(X))$. Let $\theta \in \mathbb{K}$ be the root of μ over $\mathbb{K}$. Then,

$$\mathbb{K} = \{c_0 + c_1\theta + \cdots + c_{n-1}\theta^{n-1} : c_0, \ldots, c_{n-1} \in \mathbb{F}\} \tag{42}$$

Although this definition is quite useful, we will mostly rely on the polynomial quotient ring definition. Let us define the **prime field extension**.

Definition 3.4. Suppose p is prime and $m \geq 2$. Let $\mu \in \mathbb{F}_p[X]$ be an irreducible polynomial of degree m. Then, elements of $\mathbb{F}_{p^m}$ are polynomials in $\mathbb{F}_p^{(\leq m)}[X]$. In other words,

$$\mathbb{F}_{p^m} = \{c_0 + c_1X + \cdots + c_{m-1}X^{m-1} : c_0, \ldots, c_{m-1} \in \mathbb{F}_p\}, \tag{43}$$

where all operations are performed modulo $\mu(X)$.

Again, let us consider a few examples.

Example. Consider the $\mathbb{F}_{2^4}$. Then, there are 16 elements in this set:

$$\begin{aligned} &0, 1, X, X+1, \\ &X^2, X^2+1, X^2+X, X^2+X+1, \\ &X^3, X^3+1, X^3+X, X^3+X+1, \\ &X^3+X^2, X^3+X^2+1, X^3+X^2+X, X^3+X^2+X+1. \end{aligned} \tag{44}$$

One might choose the following reduction polynomial: $\mu(X) = X^4 + X + 1$ (of degree 4). Then, operations are performed in the following manner:

- Addition: $(X^3 + X^2 + 1) + (X^2 + X + 1) = X^3 + X$.
- Subtraction: $(X^3 + X^2 + 1) - (X^2 + X + 1) = X^3 + X$.
- Multiplication: $(X^3 + X^2 + 1) \cdot (X^2 + X + 1) = X^2 + 1$ since:

$$(X^3 + X^2 + 1) \cdot (X^2 + X + 1) = X^5 + X + 1 \bmod (X^4 + X + 1) = X^2 + 1 \tag{45}$$

- Inversion: $(X^3 + X^2 + 1)^{-1} = X^2$ since $(X^3 + X^2 + 1) \cdot X^2 \bmod (X^4 + X + 1) = 1$.

Now, in the subsequent sections, we would need to extend $\mathbb{F}_p$ at least to $\mathbb{F}_{p^2}$. A convenient choice, similarly to the complex numbers, is to take $\mu(X) = X^2 + 1$. However, in contrast to $\mathbb{R}$, equation $X^2 = -1 \pmod p$ might have solutions over certain prime numbers p. Thus, we consider proposition below.

Proposition 3.5. Let p be an odd prime. Then $X^2 + 1$ is irreducible in $\mathbb{F}_p[X]$ if and only if $p \equiv 3 \pmod 4$.

Corollary 3.6. $\mathbb{F}_{p^2} = \mathbb{F}_p[u]/(u^2+1)$ is a valid prime field extension for odd primes p satisfying $p \equiv 3 \pmod 4$. In this case, extended elements are of the form $c_0 + c_1u$ where $c_0, c_1 \in \mathbb{F}_p$ and $u^2 = -1$.

3.1.3 Multiplicative Group of a Finite Field

The non-zero elements of $\mathbb{F}_p$, denoted as $\mathbb{F}_p^\times$, form a multiplicative cyclic group. In other words, there exist elements $g \in \mathbb{F}_p^\times$, called *generators*, such that

$$\mathbb{F}_p^\times = \{g^k : 0 \leq k \leq p-2\} \tag{46}$$

The order of $x \in \mathbb{F}_p^\times$ is the smallest positive integer r such that $x^r = 1$. It is also not difficult to show that $r \mid (p-1)$.

Definition 3.7. $\omega \in \mathbb{F}$ is the *primitive root* in the finite field $\mathbb{F}$ if $\langle \omega \rangle = \mathbb{F}^\times$.

Example. $\omega = 3$ is the primitive root of $\mathbb{F}_7$. Indeed,

$$3^1 = 3, 3^2 = 2, 3^3 = 6, 3^4 = 4, 3^5 = 5, 3^6 = 1. \tag{47}$$

So clearly $\langle \omega \rangle = 7$.

In STARKs (and in optimizing operations) for DFT (Discrete Fourier Transform) we would need the so-called nth primitive roots of unity.

Example. For those who studied complex numbers a bit (it is totally OK if you did not, so you might skip this example), recall an equation $\zeta^n = 1$ over $\mathbb{C}$. The solutions are $\zeta_k = \cos\left(\frac{2\pi k}{n}\right) + i \sin\left(\frac{2\pi k}{n}\right)$ for $k \in \{0, 1, \ldots, n-1\}$, so one has exactly n solutions (in contrast to $x^n = 1$ over $\mathbb{R}$ where there are at most 2 solutions[a]). For any solution ζ_k, it is true that $\zeta_k^n = 1$, but if one were to consider the subgroup generated by ζ_k (that is, $\{1, \zeta_k, \zeta_k^2, \ldots\}$), then not neccecerily $\langle \zeta_k \rangle$ would enumerate all the roots of unity $\{\zeta_j\}_{j=0}^{n-1}$. For that reason, we call ζ_k the nth primitive root of unity if $\langle \zeta_k \rangle$ enumerates all roots of unity. One can show that this is the case if and only if $\gcd(k, n) = 1$. This is always the case for $k = 1$, so commonly mathematicians use ζ_n to denote an expression $\cos\frac{2\pi}{n} + i \sin\frac{2\pi}{n} = e^{2\pi i/n}$.

[a] Think why.

Yet, let us give the broader definition, including the finite fields case.

Definition 3.8. ω is the nth primitive root of unity if $\omega^n = 1$ and $\omega^k \neq 1$ for all $1 \leq k < n$.

Note that such ω exists if and only if $n \mid (p-1)$.

3.1.4 Algebraic Closure

Consider the following interesting question: suppose we have a field $\mathbb{F}$. Is there an extension $\mathbb{K}/\mathbb{F}$ such that $\mathbb{K}$ contains all roots of any polynomial in $\mathbb{F}[X]$? The answer is yes, and such a field is called the **algebraic closure** of $\mathbb{F}$, although not always this algebraic closure has a nice form. But first, let us define what it means for field $\mathbb{F}$ to be algebraically closed.

Definition 3.9. A field $\mathbb{F}$ is called **algebraically closed** if every non-constant polynomial $p(x) \in \mathbb{F}[X]$ has a root in $\mathbb{F}$.

Example. $\mathbb{R}$ is not algebraically closed since $X^2 + 1$ has no roots in $\mathbb{R}$. However, $\mathbb{C}$ is algebraically closed, which follows from the fundamental theorem of algebra. Since $\mathbb{C}$ is a field extension of $\mathbb{R}$, it is also an algebraic closure of $\mathbb{R}$. This is commonly denoted as $\overline{\mathbb{R}} = \mathbb{C}$.

Definition 3.10. A field $\mathbb{K}$ is called an **algebraic closure** of $\mathbb{F}$ if $\mathbb{K}/\mathbb{F}$ is algebraically closed. This is denoted as $\overline{\mathbb{F}} = \mathbb{K}$.

Since we are doing cryptography and not mathematics, we are interested in the algebraic closure of $\mathbb{F}_p$. Well, I have two news for you (as always, one is good and one is bad). The good news is that any finite field $\mathbb{F}_{p^m}$ has an algebraic closure. The bad news is that it does not have a form $\mathbb{F}_{p^k}$ for $k > m$ and there are infinitely many elements in it (so in other words, the algebraic closure of a finite field is not finite). This is due to the following theorem.

Theorem 3.11. No finite field $\mathbb{F}$ is algebraically closed.

Proof. Suppose $f_1, f_2, \ldots, f_n \in \mathbb{F}$ are all elements of $\mathbb{F}$. Consider the following polynomial:

$$p(x) = \prod_{i=1}^{n}(x - f_i) + 1 = (x - f_1)(x - f_2)\cdots(x - f_n) + 1. \tag{48}$$

Clearly, $p(x)$ is a non-constant polynomial and has no roots in $\mathbb{F}$, since for any $f \in \mathbb{F}$, one has $p(f) = 1$. ■

But what form does the $\overline{\mathbb{F}}_p$ have? Well, it is a union of all $\mathbb{F}_{p^k}$ for $k \geq 1$. This is formally written as:

$$\overline{\mathbb{F}}_p = \bigcup_{k \in \mathbb{N}} \mathbb{F}_{p^k}. \tag{49}$$

Remark. But this definition is super counter-intuitive! So here how we usually interpret it. Suppose I tell you that polynomial $q(x)$ has a root in $\overline{\mathbb{F}}_p$. What that means is that there exists some extension $\mathbb{F}_{p^m}$ such that for some $\alpha \in \mathbb{F}_{p^m}$, $q(\alpha) = 0$. We do not know how large this m is, but we know that it exists. For that reason, $\overline{\mathbb{F}}_p$ is defined as an infinite union of all possible field extensions.

3.2 Elliptic Curves

3.2.1 Classical Definition

Probably, there is no need to explain the importance of elliptic curves. Essentially, the main group being used for cryptographic protocols is the group of points on an elliptic curve. If elliptic curve is "good enough", then the discrete logarithm problem assumption, Diffie-Hellman assumption and other core cryptographic assumptions hold. Moreover, this group does not require a large field size, which is a huge advantage for many cryptographic protocols.

So, let us formally define what an elliptic curve is. Further assume that, when speaking of the finite field $\mathbb{F}_p$, the underlying prime number is greater than 3.[7]. The definition is the following.

[7] Note that, for example, for $\mathbb{F}_{2^n}$ equation of elliptic curve is very different, but usually we do not deal with binary field elements.

Definition 3.12. Suppose that $\mathbb{K}$ is a field. An **elliptic curve** E over $\mathbb{K}$ is defined as a set of points $(x, y) \in \mathbb{K}^2$:

$$y^2 = x^3 + ax + b, \tag{50}$$

called a **Short Weierstrass equation**, where $a, b \in \mathbb{K}$ and $4a^3 + 27b^2 \neq 0$. We denote $E/\mathbb{K}$ to denote the elliptic curve over field $\mathbb{K}$.

Remark. One might wonder why $4a^3 + 27b^2 \neq 0$. This is due to the fact that the curve $y^2 = x^3 + ax + b$ might have certain degeneracies and special points, which are not desirable for us. So we require this condition to make $E/\mathbb{K}$ "good".

Definition 3.13. We say that $P = (x_P, y_P) \in \mathbb{A}^2(\mathbb{K})$ is the **affine representation** of the point on the elliptic curve $E/\mathbb{K}$ if it satisfies the equation $y_P^2 = x_P^3 + ax_P + b$.

Example. Consider the curve $E/\mathbb{Q} : y^2 = x^3 - x + 9$. This is an elliptic curve. Consider $P = (0, 3), Q = (-1, -3) \in \mathbb{A}^2(\mathbb{Q})$: both are valid affine points on the curve. See Figure 3.1.

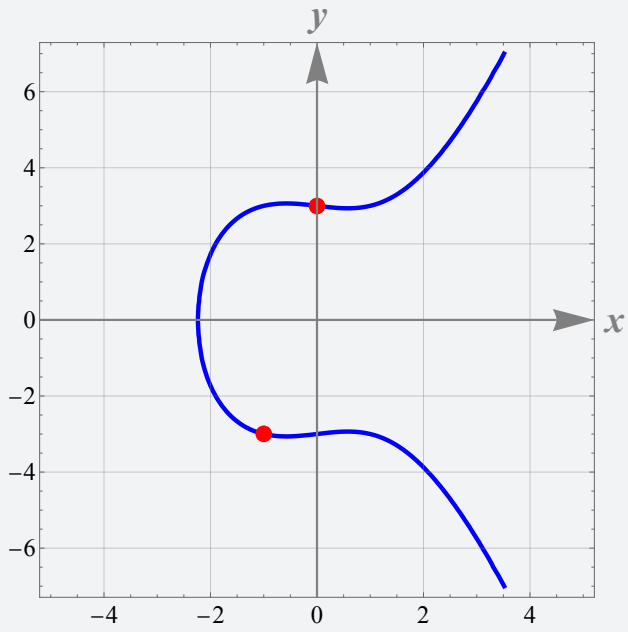


Figure 3.1: Elliptic curve $E/\mathbb{Q} : y^2 = x^3 - x + 9$ with points $P = (0, 3), Q = (-1, -3)$ depicted on it.

Typically, our elliptic curve is defined over a finite field $\mathbb{F}_p$, so we are interested in this paricular case.

Remark. Although, in many cases one might encounter the definition where an elliptic curve E is defined over the algebraic closure of $\mathbb{F}_p$, that is $E/\overline{\mathbb{F}}_p$. This is typically important when considering elliptic curve pairings. However, for the sake of simplicity, we will consider elliptic curves over $\mathbb{F}_p$ and corresponding finite extensions $\mathbb{F}_{p^m}$ as of now.

Remark. It is easy to see that if $(x, y) \in E/\mathbb{K}$, then $(x, -y) \in E/\mathbb{K}$. We will use this fact intensively further.

Now, elliptic curves are useless without any operation defined on top of them. But as will be seen later, it is quite unclear how to define the identity element. For that reason, we introduce a bit different definition of a set of points on the curve.

Definition 3.14. The set of points on the curve, denoted as $E_{a,b}(\mathbb{K})$, is defined as:

$$E_{a,b}(\mathbb{K}) = \{(x, y) \in \mathbb{A}^2(\mathbb{K}) : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\}, \tag{51}$$

where $\mathcal{O}$ is the so-called **point at infinity**.

Remark. The difference between $E(\mathbb{K})$ and $E/\mathbb{K}$ is that the former includes the point at infinity, while the latter does not. We also omit the index a, b, so instead of $E_{a,b}(\mathbb{K})$ we write simply $E(\mathbb{K})$.

Now, the reason we introduced the point at infinity $\mathcal{O}$ is because it allows us to define the group binary operation $\oplus$ on the elliptic curve. The operation is sometimes called the **chord-tangent law**. Let us define it.

Definition 3.15. Consider the curve $E(\mathbb{F}_{p^m})$. We define $\mathcal{O}$ as the identity element of the group. That is, for all points P, we set $P \oplus \mathcal{O} = \mathcal{O} \oplus P = P$. For any other non-identity elements $P = (x_P, y_P), Q = (x_Q, y_Q) \in E(\mathbb{F}_{p^m})$, define the $P \oplus Q = (x_R, y_R)$ as follows:

1. If $x_P \neq x_Q$, use the **chord method**. Define $\lambda := \frac{y_P - y_Q}{x_P - x_Q}$ — the slope between P and Q. Set the resultant coordinates as:

$$x_R := \lambda^2 - x_P - x_Q, \quad y_R := \lambda(x_P - x_R) - y_P. \tag{52}$$

2. If $x_P = x_Q \wedge y_P = y_Q$ (that is, $P = Q$), use the **tangent method**. Define the slope of the tangent at P as $\lambda := \frac{3x_P^2 + a}{2y_P}$ and set

$$x_R := \lambda^2 - 2x_P, \quad y_R := \lambda(x_P - x_R) - y_P. \tag{53}$$

3. Otherwise, define $P \oplus Q := \mathcal{O}$.

The aforementioned definition is illustrated in the Figure below[8].

[8] Illustration taken from *"Pairing for Beginners"*

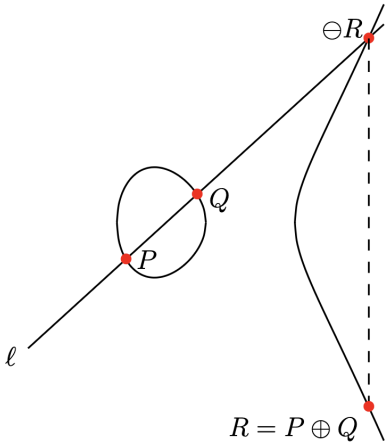


Figure 2.5: Elliptic curve addition.

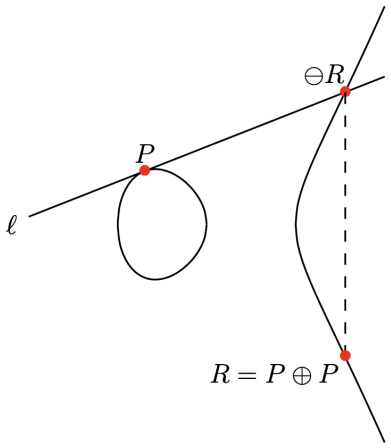


Figure 2.6: Elliptic curve doubling.

Example. Consider $E/\mathbb{R} : y^2 = x^3 - 2x$. The points $(-1,-1), (0,0), (2,2)$ are all on E and also on the line $\ell : y = x$. Therefore, $(-1,1) \oplus (0,0) = (2,-2)$ or, similarly, $(2,2) \oplus (-1,-1) = (0,0)$.
Now, let us compute $[2](-1,-1)$. Calculate the tangent slope as $\lambda := \frac{3\cdot(-1)^2-2}{2\cdot(-1)} = -\frac{1}{2}$. Thus, the tangent line has an equation $\ell' : y = -\frac{1}{2}x + c$. Substituting $(-1,-1)$ into the equation, we get $c = -\frac{3}{2}$. Therefore, the equation of the tangent line is $y = -\frac{1}{2}x - \frac{3}{2}$. The intersection of the curve and the line is $\left(\frac{9}{4}, -\frac{21}{8}\right)$, yielding $[2](-1,-1) = \left(\frac{9}{4}, -\frac{21}{8}\right)$.
The whole illustration is depicted in Figure 3.2.

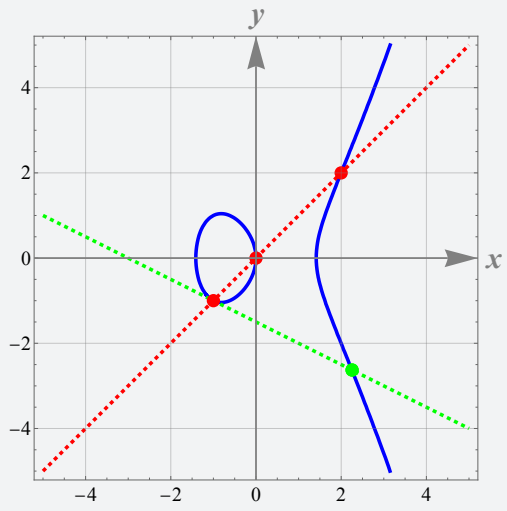


Figure 3.2: Illustration of the group law on the elliptic curve $E/\mathbb{R} : y^2 = x^3 - 2x$. In red we marked points lying on the line $\ell : y = x$. In dashed red, we marked the line ℓ, while in dashed green — the tangent line ℓ' at $(-1,-1)$, which is used to calculate $[2](-1,1)$.

Theorem 3.16. $(E(\mathbb{F}_{p^m}), \oplus)$ forms an abelian group.

Proof Sketch. The identity element is $\mathcal{O}$. Every point $\mathcal{O} \neq P = (x_P, y_P) \in E(\mathbb{F}_{p^m})$ has

an additive inverse: indeed, $\ominus P := (x_P, -y_P)$. Finally, a bit of algebra might show that the operation is associative. It is also clearly commutative: even geometrically it is evident, that the result of $P \oplus Q$ does not depend on the order of P and Q ("drawing a line between P and Q" and "drawing a line between Q and P" are equivalent statements). ■

Now, let us talk a bit about the group order. The group order is the number of elements in the group. For elliptic curves, the group order is typically denoted as r or n, but we are going to use r. Also, the following theorem is quite important.

Theorem 3.17. Define $r := |E(\mathbb{F}_{p^m})|$. Then, $r = p^m + 1 - t$ for some integer $|t| \leq 2\sqrt{p^m}$. A bit more intuitive explanation: the number of points on the curve is close to $p^m + 1$. This theorem is commonly called the **Hasse's theorem on elliptic curves**, and the value t is called the **trace of Frobenius**.

Remark. In fact, $r = |E(\mathbb{F}_{p^m})|$ can be computed in $O(\log(p^m))$, so the number of points can be computed efficiently even for fairly large primes p.

Finally, let us define the scalar multiplication operation.

Definition 3.18. Let $P \in E(\mathbb{F}_{p^m})$ and $\alpha \in \mathbb{Z}_r$. Define the scalar multiplication $[\alpha]P$ as:

$$[\alpha]P = \underbrace{P \oplus P \oplus \cdots \oplus P}_{\alpha \text{ times}}. \tag{54}$$

Question. Why do we restrict α to $\mathbb{Z}_r$ and not to $\mathbb{Z}$?

3.2.2 Discrete Logarithm Problem on Elliptic Curves

Finally, as defined in the previous section, the **discrete logarithm** problem on the elliptic curve is the following: typically, $E(\mathbb{F}_p)$ is cyclic, meaning there exist some point $G \in E(\mathbb{F}_p)$, called the **generator**, such that $\langle G \rangle = E(\mathbb{F}_p)$. Given $P \in E(\mathbb{F}_p)$, the problem consists in finding such a scalar $\alpha \in \mathbb{Z}_r$ such that $[\alpha]G = P$.

Now, if the curve is "good", then the discrete logarithm problem is hard. In fact, the best-known algorithms have a complexity $O(\sqrt{r})$. However, there are certain cases when the discrete log problem is much easier.

1. If r is composite, and all its prime factors are less than some bound $r_{\max}$, then the discrete log problem can be solved in $O(\sqrt{r_{\max}})$. For this very reason, typically r is prime.
2. If $|E(\mathbb{F}_p)| = p$, then the discrete logarithm can be solved in polynomial time. These curves are called **anomalous curves**.
3. Suppose that there is some small integer $\tau > 0$ such that $r \mid (p^\tau - 1)$. The discrete log in that case reduces to the discrete log in the finite field $\mathbb{F}_{p^\tau}$, which is typically not hard for small enough τ.

3.3 Exercises

Warmup (Oleksandr in search of perfect field extension)

Exercise 1. Oleksandr decided to build $\mathbb{F}_{49}$ as $\mathbb{F}_7[i]/(i^2 + 1)$. Compute $(3 + i)(4 + i)$.

a) $6 + i$.

b) 6.

c) $4 + i$.

d) 4.

e) $2 + 4i$.

Exercise 2. Oleksandr came up with yet another extension $\mathbb{F}_{p^2} = \mathbb{F}_p[i]/(i^2 + 2)$. He asked interns to calculate $2/i$. Based on five answers given below, help Oleksandr to find the correct one.

a) 1.

b) $p - 2$.

c) $(p - 3)i$.

d) $(p - 1)i$.

e) $p - 1$.

Exercise 3*. After endless tries, Oleksandr has finally found the perfect field extension: $\mathbb{F}_{p^2} := \mathbb{F}_p[v]/(v^2 + v + 1)$. However, Oleksandr became very frustrated since not for any p this would be a valid field extension. For which of the following values p such construction would **not** be a valid field extension? Use the fact that equation $\omega^3 = 1$ over $\mathbb{F}_p$ has non-trivial solutions (meaning, two others except for $\omega = 1$) if $p \equiv 1 \pmod 3$. You can assume that listed numbers are primes.

a) 8431.

b) 9173.

c) 9419.

d) 6947.

Exercises 4-9. Tower of Extensions

You are given the passage explaining the topic of tower of extensions. The text has gaps that you need to fill in with the correct statement among the provided choices.

This question demonstrates the concept of the so-called **tower of extensions**. Suppose we want to build an extension field $\mathbb{F}_{p^4}$. Of course, we can find some irreducible polynomial $p(X)$ of degree 4 over $\mathbb{F}_p$ and build $\mathbb{F}_{p^4}$ as $\mathbb{F}_p[X]/(p(X))$. However, this method is very inconvenient since implementing the full 4-degree polynomial arithmetic is inconvenient. Moreover, if we were to implement arithmetic over, say, $\mathbb{F}_{p^{24}}$, that would make the matters worse. For this reason, we will build $\mathbb{F}_{p^4}$ as $\mathbb{F}_{p^2}[j]/(q(j))$ where $q(j)$ is an irreducible polynomial of degree 2 over $\mathbb{F}_{p^2}$, which itself is represented as $\mathbb{F}_p[i]/(r(i))$ for some suitable irreducible quadratic polynomial $r(i)$. This way, we can first implement $\mathbb{F}_{p^2}$, then $\mathbb{F}_{p^4}$, relying on the implementation of $\mathbb{F}_{p^2}$ and so on.

For illustration purposes, let us pick $p := 5$. As noted above, we want to build $\mathbb{F}_{5^2}$ first. A valid way to represent $\mathbb{F}_{5^2}$ would be to set $\mathbb{F}_{5^2} :=$ **4**. Given this representation, the zero of a linear polynomial $f(x) = ix - (i+3)$, defined over $\mathbb{F}_{5^2}$, is **5**.

Now, assume that we represent $\mathbb{F}_{5^4}$ as $\mathbb{F}_{5^2}[j]/(j^2-\xi)$ for $\xi = i+1$. Given such representation, the value of j^4 is **6**. Finally, given $c_0 + c_1 j \in \mathbb{F}_{5^4}$ we call $c_0 \in \mathbb{F}_{5^2}$ a **real part**, while $c_1 \in \mathbb{F}_{5^2}$ an **imaginary part**. For example, the imaginary part of number $j^3 + 2i^2\xi$ is **7**, while the real part of $(a_0 + a_1 j)b_1 j$ is **8**. Similarly to complex numbers, it motivates us to define the number's **conjugate**: for $z = c_0 + c_1 j$, define the conjugate as $\overline{z} := c_0 - c_1 j$. The expression $z\overline{z}$ is then **9**.

Exercise 4.

a) $\mathbb{F}_5[i]/(i^2+1)$
b) $\mathbb{F}_5[i]/(i^2+2)$
c) $\mathbb{F}_5[i]/(i^2+4)$
d) $\mathbb{F}_5[i]/(i^2+2i+1)$
e) $\mathbb{F}_5[i]/(i^2+4i+4)$

Exercise 5.

a) $1+i$
b) $1+2i$
c) $1+4i$
d) $2+3i$
e) $3+i$

Exercise 6.

a) $4+2i$
b) $4i$
c) 1
d) $1+2i$
e) $2+4i$

Exercise 7.

a) equal to zero.
b) equal to one.
c) equal to the real part.
d) $2(1+i)$
e) -4

Exercise 8.

a) $a_1 b_1$
b) $a_1 b_1 \xi$
c) $a_0 b_1$
d) $a_0 b_1 \xi$
e) $a_0 a_1$

Exercise 9.

a) $c_0^2 + c_1^2$
b) $c_0^2 - c_1^2 \xi$
c) $c_0^2 + c_1^2 \xi^2$
d) $(c_0^2 + c_1^2 \xi)j$
e) $(c_0^2 - c_1^2)j$

Elliptic Curves

Exercise 10. Suppose that elliptic curve is defined as $E/\mathbb{F}_7 : y^2 = x^3 + b$. Suppose $(2, 3)$ lies on the curve. What is the value of b?

Exercise 11. Sum of which of the following pairs of points on the elliptic curve $E/\mathbb{F}_{11}$ is equal to the point at infinity $\mathcal{O}$ for any valid curve equation?

a) $P = (2, 3), Q = (2, 8)$.
b) $P = (9, 2), Q = (2, 8)$.
c) $P = (9, 9), Q = (5, 7)$.
d) $P = \mathcal{O}, Q = (2, 3)$.
e) $P = [10]G, Q = G$ where G is a generator.

Exercise 12. Consider an elliptic curve E over $\mathbb{F}_{167^2}$. Denote by r the order of the group of points on E (that is, $r = |E|$). Which of the following **can** be the value of r?

a) $167^2 - 5$
b) $167^2 - 1000$
c) $167^2 + 5 \cdot 167$
d) 170^2
e) 160^2

Exercise 13. Suppose that for some elliptic curve E the order is $|E| = qr$ where both q and r are prime numbers. Among listed, what is the most optimal complexity of algorithm to solve the discrete logarithm problem on E?

a) $O(qr)$
b) $O(\sqrt{qr})$
c) $O(\sqrt{\max\{q, r\}})$
d) $O(\sqrt{\min\{q, r\}})$
e) $O(\max\{q, r\})$

4 Projective Coordinates and Pairing

4.1 Relations

Before delving into the projective coordinates and further zero-knowledge topics, let us first discuss the concept of relations, which will be intensively used from now on. Now, what is a relation? The definition is incredibly concise.

Definition 4.1. Let $\mathcal{X}, \mathcal{Y}$ be some sets. Then, $\mathcal{R}$ is a **relation** if

$$\mathcal{R} \subset \mathcal{X} \times \mathcal{Y} = \{(x, y) : x \in \mathcal{X}, y \in \mathcal{Y}\} \tag{55}$$

Interpretation is approximately the following: suppose we have sets $\mathcal{X}$ and $\mathcal{Y}$. Then, relation $\mathcal{R}$ gives a set of pairs (x, y), telling that $x \in \mathcal{X}$ and $y \in \mathcal{Y}$ are *related*.

Example. Let $\mathcal{X} = \{\text{Oleksandr}, \text{Phat}, \text{Anton}\}$ and $\mathcal{Y} = \{\text{Backend}, \text{Frontend}, \text{Research}\}$. Define the following relation of "person x works in field y":

$$\mathcal{R} = \{(\text{Oleksandr}, \text{Research}), (\text{Phat}, \text{Frontend}), (\text{Anton}, \text{Backend})\} \tag{56}$$

Obviously, $\mathcal{R} \subset \mathcal{X} \times \mathcal{Y}$, so $\mathcal{R}$ is a relation.

Remark. There are many ways to express that $(x, y) \in \mathcal{R}$. Most common are $x\mathcal{R}y$ and $x \sim y$. Also, sometimes, one might encounter relation definition as a boolean function $\mathcal{R} : \mathcal{X} \times \mathcal{Y} \to \{0, 1\}$, where $\mathcal{R}(x, y)$ is 1 if (x, y) is in the relation, and 0 otherwise.
Further, we will use notation $x \sim y$ to denote that $(x, y) \in \mathcal{R}$.

Example. Let E be a cyclic group of points on the Elliptic Curve of order $r \geq 2$ with a generator $\langle G \rangle = E$. Let $\mathcal{X} = \mathbb{Z}_r$ and $\mathcal{Y} = E$. Define a relation $\mathcal{R} \subset \mathcal{X} \times \mathcal{Y}$ by:

$$\mathcal{R} = \{(\alpha, P) \in \mathbb{Z}_r \times E : [\alpha]G = P\} \tag{57}$$

Essentially, such a relation is a set of secret keys α and corresponding public keys P. In this case, for example, $0\mathcal{R}\mathcal{O}$ and $1\mathcal{R}G$ or $0 \sim \mathcal{O}$ and $1 \sim G$.

Remark. When we say that $\sim$ is a relation on a set $\mathcal{X}$, we mean that $\sim$ is a relation $\mathcal{R}$ on the following Cartesian product: $\mathcal{R} \subset \mathcal{X} \times \mathcal{X}$.

Remark. The provided example is relevant in most cases (ecdsa, eddsa, schnorr signatures etc.). But for some algorithms, the relation between secret key α and public key P can be defined as:

$$\mathcal{R} = \{(\alpha, P) \in \mathbb{Z}_r \times E : \ominus[\alpha]G = P\} \tag{58}$$

for DSTU 4145 standard or even:

$$\mathcal{R} = \{(\alpha, P) \in \mathbb{Z}_r \times E : [\alpha^{-1}]G = P\} \tag{59}$$

for twisted ElGamal algorithm.

Now, let us formally define the term **equivalence relation**.

Definition 4.2. Let $\mathcal{X}$ be a set. A relation $\sim$ on $\mathcal{X}$ is called an **equivalence relation** if it satisfies the following properties:

1. **Reflexivity:** $x \sim x$ for all $x \in \mathcal{X}$.
2. **Symmetry:** If $x \sim y$, then $y \sim x$ for all $x, y \in \mathcal{X}$.
3. **Transitivity:** If $x \sim y$ and $y \sim z$, then $x \sim z$ for all $x, y, z \in \mathcal{X}$.

Example. Let $\mathcal{X}$ be the set of all people. Define a relation $\sim$ on $\mathcal{X}$ by $x \sim y$ if $x, y \in \mathcal{X}$ have the same birthday. Then $\sim$ is an equivalence relation on $\mathcal{X}$. Let us demonstrate that:

1. **Reflexivity:** $x \sim x$ since x has the same birthday as x.
2. **Symmetry:** If $x \sim y$, then $y \sim x$ since x has the same birthday as y.
3. **Transitivity:** If $x \sim y$ and $y \sim z$, then $x \sim z$ since x has the same birthday as y and y has the same birthday as z.

Example. Suppose $\mathcal{X} = \mathbb{Z}$ and n is some fixed integer. Let $a \sim b$ mean that $a \equiv b \pmod{n}$. It is easy to verify that $\sim$ is an equivalence relation:

1. **Reflexivity:** $a \equiv a \pmod{n}$, so $a \sim a$.
2. **Symmetry:** If $a \equiv b \pmod{n}$, then $b \equiv a \pmod{n}$, so $b \sim a$.
3. **Transitivity:** If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$. It is not that obvious, so we can prove it: from the first equality we have $\exists q \in \mathbb{Z} : a - b = nq$. From the second, $\exists r \in \mathbb{Z} : b - c = nr$. Adding both we get $(a-b)+(b-c) = n(r+q)$ or, equivalently, $a - c = n(r+q)$, meaning $a \equiv c \pmod{n}$.

The example below is less obvious with a bit more difficult proof, which we will skip. Yet, it is quite curious, so here it is.

Example. Let $\mathcal{G}$ be the set of all possible groups. Define a relation $\sim$ on $\mathcal{G}$ by $\mathbb{G} \sim \mathbb{H}$ if $\mathbb{G} \cong \mathbb{H}$ (in other words, $\mathbb{G}$ and $\mathbb{H}$ are isomorphic). Then $\sim$ is an equivalence relation.

Now, suppose I give you a set $\mathcal{X}$ with some equivalence relation $\sim$ (say, $\mathcal{X} = \mathbb{Z}$ and $a \equiv b \pmod{n}$). Notice that you can find some subset $\mathcal{X}' \subset \mathcal{X}$ in which all elements are equivalent (and any other element from $\mathcal{X} \backslash \mathcal{X}'$ is not). In the case of modulo relation above, $\mathcal{X}'$ could be the set of all integers that are congruent to 1 modulo n, so $\mathcal{X}' = \{\dots, -n+1, 1, n+1, 2n+1, \dots\}$. This way, we can partition the set $\mathcal{X}$ into disjoint subsets, where all elements in each subset are equivalent. Such subsets are called **equivalence classes**. Now, let us give a formal definition.

Definition 4.3. Let $\mathcal{X}$ be a set and $\sim$ be an equivalence relation on $\mathcal{X}$. For any $x \in \mathcal{X}$, the **equivalence class** of x is the set

$$[x] = \{y \in \mathcal{X} : x \sim y\} \tag{60}$$

The **set of all equivalence classes** is denoted by $\mathcal{X}/\sim$ (or, if the relation $\mathcal{R}$ is given explicitly, then $\mathcal{X}/\mathcal{R}$), which is read as "$\mathcal{X}$ modulo relation $\sim$".

Example. Let $\mathcal{X} = \mathbb{Z}$ and n be some fixed integer. Define $\sim$ on $\mathcal{X}$ by $x \sim y$ if $x \equiv y \pmod{n}$. Then the equivalence class of x is the set

$$[x] = \{y \in \mathbb{Z} : x \equiv y \pmod{n}\} \tag{61}$$

For example, $[0] = \{\ldots, -2n, -n, 0, n, 2n, \ldots\}$ while $[1] = \{\ldots, -2n+1, -n+1, 1, n+1, 2n+1, \ldots\}$.

Now, as we have said before, a set of all equivalence classes form a partition of the set $\mathcal{X}$. This means that any element $x \in \mathcal{X}$ belongs to exactly one equivalence class. This is a very important property, which we will use in the next section. Formally, we have the following lemma.

Lemma 4.4. Let $\mathcal{X}$ be a set and $\sim$ be an equivalence relation on $\mathcal{X}$. Then,

1. For each $x \in \mathcal{X}, x \in [x]$ (quite obvious, follows from reflexivity).
2. For each $x, y \in \mathcal{X}$, $x \sim y$ if and only if $[x] = [y]$.
3. For each $x, y \in \mathcal{X}$, either $[x] = [y]$ or $[x] \cap [y] = \emptyset$.

Example. Let $n \in \mathbb{N}$ and, again, $\mathcal{X} = \mathbb{Z}$ with a "modulo n" equivalence relation $\mathcal{R}_n$. Define the equivalence class of x by $[x]_n = \{y \in \mathbb{Z} : x \equiv y \pmod{n}\}$. Then,

$$\mathbb{Z}/\mathcal{R}_n = \{[0]_n, [1]_n, [2]_n, \ldots, [n-2]_n, [n-1]_n\} \tag{62}$$

forms a partition of $\mathbb{Z}$, that is

$$\bigcup_{i=0}^{n-1} [i]_n = \mathbb{Z}, \tag{63}$$

and for all $i, j \in \{0, 1, \ldots, n-1\}$, if $i \neq j$, then $[i]_n \cap [j]_n = \emptyset$. Commonly, we denote the set of all equivalence classes as $\mathbb{Z}/n\mathbb{Z}$ or, as we got used to, $\mathbb{Z}_n$. Moreover, we can naturally define the addition as:

$$[x]_n + [y]_n = [x+y]_n \tag{64}$$

Then, the set $(\mathbb{Z}/n\mathbb{Z}, +)$ with the defined addition is a group.

The primary reason we considered equivalence relations is that we will define the projective space as a set of equivalence classes. Besides this, when defining proofs of knowledge, argument of knowledge and zero-knowledge protocols, we will use the concept of relations and equivalence relations intensively.

4.2 Elliptic Curve in Projective Coordinates

4.2.1 Projective Space

Recall that we defined the elliptic curve as

$$E(\overline{\mathbb{F}}_p) := \{(x, y) \in \mathbb{A}^2(\overline{\mathbb{F}}_p) : y^2 = x^3 + ax + b\} \cup \{\mathcal{O}\} \tag{65}$$

The above definition is the definition of the elliptic curve in *the affine space*. However, notice that in this case we need to append a somewhat artificial point $\mathcal{O}$ to the curve. This is done

to make the curve a group since without this point it is unclear how to define addition of two, say, negative points on the curve (since the resultant vertical line does not intersect the curve at any other point). The way to unify all the points $E/\overline{\mathbb{F}}_p$ with this magical point at infinity $\mathcal{O}$ is to use the **projective space**.

Essentially, instead of working with points in affine n-space (in our case, with two-dimensional points $\mathbb{A}^2(\mathbb{K})$), we work with lines that pass through the origin in $(n+1)$-dimensional space (in our case, 3-dimensional space $\mathbb{A}^3(\mathbb{K})$). We say that two points from this $(n+1)$-dimensional space are **equivalent** if they lie on the same line that passes through the origin (we will show the illustration a bit later).

It seems strange that we need to work with 3-dimensional space to describe 2-dimensional points, but this is the way to unify all the points on the curve. Because, in this case, the point at infinity is represented by a set of points on the line that passes through the origin and is parallel to the y-axis. We will get to understanding how to interpret that. Moreover, by defining operations on the projective space, we can make the operations on the curve more efficient.

Now, to the formal definition.

Definition 4.5. Projective coordinate, denoted as $\mathbb{P}^2(\mathbb{K})$ (or sometimes simply $\mathbb{KP}^2$) is a triple of elements $(X : Y : Z)$ from $\mathbb{A}^3(\overline{\mathbb{K}}) \setminus \{0\}$ modulo the equivalence relation[a]:

$$(X_1 : Y_1 : Z_1) \sim (X_2 : Y_2 : Z_2) \text{ iff } \exists \lambda \in \overline{\mathbb{K}} : (X_1 : Y_1 : Z_1) = (\lambda X_2 : \lambda Y_2 : \lambda Z_2) \tag{66}$$

[a] Although we specify the definition for $n = 2$, the definition can be generalized to any $\mathbb{P}^n(\overline{\mathbb{K}})$.

This definition on itself might be a bit too abstract, so let us consider the concrete example for projective space $\mathbb{P}^2(\mathbb{R})$.

Example. Consider the projective space $\mathbb{P}^2(\mathbb{R})$. Then, two points $(x_1, y_1, z_1), (x_2, y_2, z_2) \in \mathbb{R}^3$ are equivalent if there exists $\lambda \in \mathbb{R}$ such that $(x_1, y_1, z_1) = (\lambda x_2, \lambda y_2, \lambda z_2)$. For example, $(1, 2, 3) \sim (2, 4, 6)$ since $(1, 2, 3) = 0.5(2, 4, 6)$.

Example. Now, how to geometrically interpret $\mathbb{P}^2(\mathbb{R})$? Consider the Figure below.

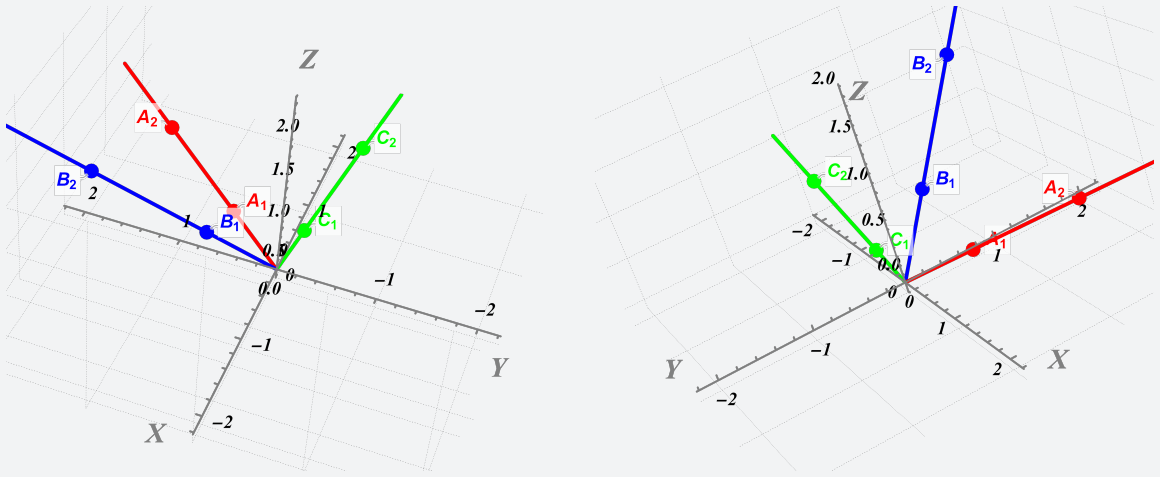


Illustration: Geometric interpretation of $\mathbb{P}^2(\mathbb{R})$, the same scene from different perspectives. The red line is represented by equation $(2t, 3t, t)$, blue line by $(-2t, 3t, 3t)$, and green line is represented by $(t, -2t, 5t)$ for parameter $t \in \mathbb{R}$.

Here, the figure demonstrates three equivalence classes, being a set of points on the **red**, **blue**, and **green** lines (except for the origin).
The reason why geometrically the set of equivalence classes lie on the same line that passes through the origin is following: suppose we have a point $\vec{v}_0 = (x_0, y_0, z_0) \in \mathbb{R}^3$, represented as a vector. Then, the set of all points that are equivalent to (x_0, y_0, z_0) is the set of all points $(\lambda x_0, \lambda y_0, \lambda z_0) = \lambda \vec{v}_0$ for $\lambda \in \mathbb{R} \setminus \{0\}$. So $\vec{v}_0$ is the representative of equivalence class $[\vec{v}_0] = \{\lambda \vec{v}_0 : \lambda \in \mathbb{R}, \lambda \neq 0\}$. Now notice, that this is a parametric equation of a line that passes through the origin and the point $\vec{v}_0$: notice that for $\lambda = 0$ (if we assume that expression is also defined for zero λ) we have the origin $\vec{0}$, while for $\lambda = 1$ we have the point $\vec{v}_0$. Then, any other values of λ in-between $[0, 1]$ or outside define the set of points lying on the same line.

Now, projective coordinates are not that useful unless we can come back to the affine space. This is done by defining the map $\phi : \mathbb{P}^2(\overline{\mathbb{K}}) \to \mathbb{A}^2(\overline{\mathbb{K}})$ as follows: $\phi : (X : Y : Z) \mapsto (X/Z, Y/Z)$. If, in turn, we want to go from the affine space to the projective space, we can define the map $\psi : \mathbb{A}^2(\overline{\mathbb{K}}) \to \mathbb{P}^2(\overline{\mathbb{K}})$ as follows: $\psi : (x, y) \mapsto (x : y : 1)$. Geometrically, map ϕ means that we take a point $(X : Y : Z)$ and project it onto the plane $Z = 1$.

Example. Again, consider three lines from the previous example. Now, we additionally draw a plane $\pi : z = 1$ in our 3-dimensional space (see Illustration below).

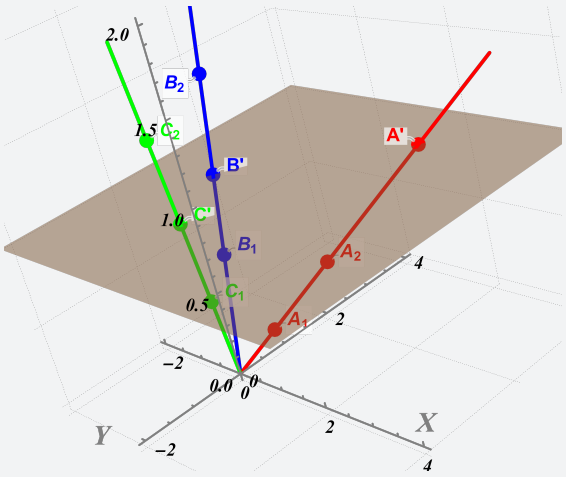


Illustration: Geometric interpretation of converting projective form to the affine form.

By using the map $(X : Y : Z) \mapsto (X/Z, Y/Z)$, all points on the line get mapped to the itersection of the line with the plane $\pi : z = 1$. This way, for example, points on the **red line** ℓ_{red} get mapped to the point $A' = (2, 3, 1)$, corresponding to $(2, 3)$ in affine coordinates. So, for example, point $(6, 9, 3) \in \ell_{\text{red}}$, lying on the same line, gets mapped to $(6/3, 9/3) = (2, 3)$. Similarly, all **blue line** points get mapped to the point $B' = (-2/3, 1, 1)$, while all **green line** points get mapped to the point $C' = (0.2, -0.4, 1)$[a].

[a] One can verify that based on the equations provided from the previous example

4.2.2 Elliptic Curve Equation in Projective Form

Now, quite an interesting question is following: how to represent (basically, rewrite) the "affine" elliptic curve equation[9]

$$E_{\mathbb{A}}(\overline{\mathbb{F}}_p) : y^2 = x^3 + ax + b, \;\; a, b \in \overline{\mathbb{F}}_p \tag{67}$$

in the projective form? Since currently, we defined the curve as the 2D curve, but now we are working in 3D space! The answer is following: recall that if $(X : Y : Z) \in \mathbb{P}^2(\overline{\mathbb{F}}_p)$ lies on the curve, so does the point $(X/Z, Y/Z)$. The condition on the latter point to lie on $E_{\mathbb{A}}(\overline{\mathbb{F}}_p)$ is following:

$$\left(\frac{Y}{Z}\right)^2 = \left(\frac{X}{Z}\right)^3 + a \cdot \frac{X}{Z} + b \tag{68}$$

But now multiply both sides by Z^3 to get rid of the fractions:

$$E_{\mathbb{P}}(\overline{\mathbb{F}}_p) : Y^2 Z = X^3 + aXZ^2 + bZ^3 \tag{69}$$

This is an equation of the elliptic curve in **projective form**.

[9] Further, we will use notation $E_{\mathbb{A}}$ to represent the elliptic curve equation in the affine form, and $E_{\mathbb{P}}$ to represent the elliptic curve in the projective form.

Now, one of the motivations to work with the projective form was to unify affine points $E_{\mathbb{A}}/\overline{\mathbb{F}}_p$ and the point at infinity $\mathcal{O}$, which acted as an identity element in the group $E_{\mathbb{A}}(\overline{\mathbb{F}}_p)$. So how do we encode the point at infinity in the projective form?

Well, notice the following observation: all points $(0 : \lambda : 0)$ always lie on the curve $E_{\mathbb{P}}(\overline{\mathbb{F}}_p)$. Moreover, the map from the projective form to the affine form is ill-defined for such points, since we would need to divide by zero. So, we can naturally make the points $(0 : \lambda : 0)$ to be the set of points at infinity. This way, we can define the point at infinity as $\mathcal{O} = (0 : 1 : 0)$.

Finally, let us summarize what we have observed so far.

Definition 4.6. The **homogenous projective form of the elliptic curve** $E_{\mathbb{P}}(\overline{\mathbb{F}}_p)$ is defined as the set of all points $(X : Y : Z) \in \mathbb{P}^2(\overline{\mathbb{F}}_p)$ in the projective space that satisfy the equation

$$E_{\mathbb{P}}(\overline{\mathbb{F}}_p) : Y^2Z = X^3 + aXZ^2 + bZ^3, \quad a, b \in \overline{\mathbb{F}}_p, \tag{70}$$

where the point at infinity is encoded as $\mathcal{O} = (0 : 1 : 0)$.

Example. Consider the BN254 curve $y^2 = x^3 + 3$ over reals $\mathbb{R}$. Its projective form is given by the equation $Y^2Z = X^3 + 3Z^3$, which gives a surface, depicted below.

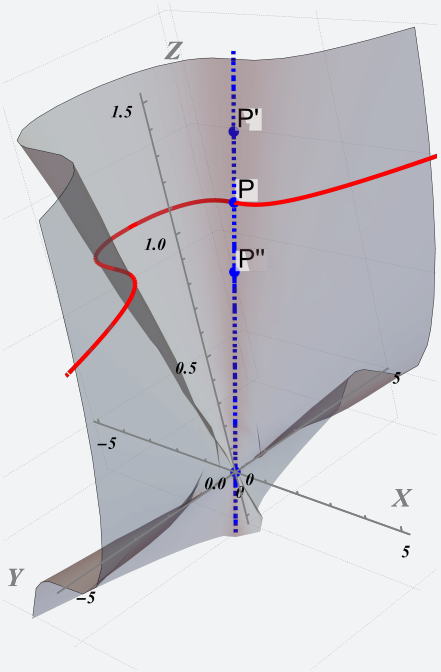


Illustration: BN254 Curve Elliptic Curve in Projective Form over $\mathbb{R}$. In **gray** is the surface, while **red** points are the points on the affine curve (lying on the plane $\pi : z = 1$).

Points $P' \approx (0 : 2.165 : 1.25)$ and $P'' \approx (0 : 1.3 : 0.75)$ in projective form both lie on the curve and get mapped to the same point $P \approx (0, 1.732)$ in affine coordinates.

4.2.3 General Projective Coordinates

Hold on, but why did we use the term *homogenous*? The reason why is because we defined equivalence as follows: $(X : Y : Z) \sim (\lambda X : \lambda Y : \lambda Z)$ for some $\lambda \in \overline{\mathbb{K}}$, called **homogenous**

coordinates. However, this is not the only way to define equivalence. Consider a more general form of equivalence relation:

$$(X : Y : Z) \sim (X' : Y' : Z') \text{ iff } \exists \lambda \in \overline{\mathbb{K}} : (X, Y, Z) = (\lambda^n X', \lambda^m Y', \lambda Z') \tag{71}$$

In this case, to come back to the affine form, we need to use the map $\phi : (X : Y : Z) \mapsto (X/Z^n, Y/Z^m)$.

Example. The case $n = 2, m = 3$ is called the **Jacobian Projective Coordinates**. An Elliptic Curve equation might be then rewritten as:

$$Y^2 = X^3 + aXZ^4 + bZ^6 \tag{72}$$

The reason why we might want to use such coordinates is that they can be more efficient in some operations, such as point addition. However, we will not delve into this topic much further.

Example. Consider the BN254 curve $y^2 = x^3 + 3$ over reals $\mathbb{R}$, again. Its *Jacobian projective form* is given by the equation $Y^2 = X^3 + 3Z^6$, which gives a surface, depicted below.

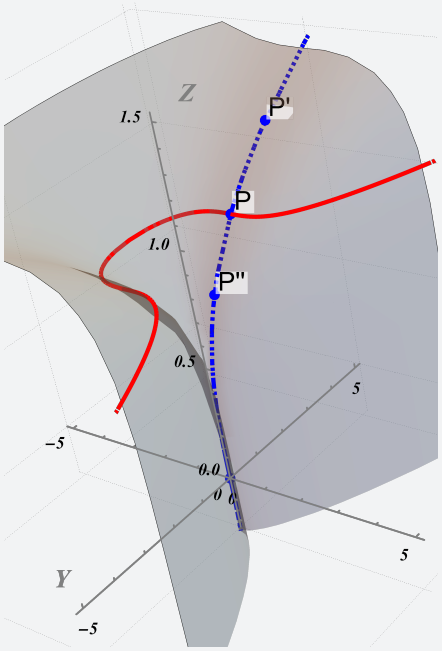


Illustration: BN254 Curve Elliptic Curve in Jacobian Projective Form over $\mathbb{R}$. In **gray** is the surface, while **red** points are the points on the affine curve (lying on the plane $\pi : z = 1$).

Notice that now, under the map $(X : Y : Z) \mapsto (X/Z^2, Y/Z^3)$, points in the same equivalence class (in $\mathbb{R}^3$) do not lie on the same line, but rather on the same *curve*. Namely, equivalence class has a form $[(x_0, y_0, z_0)] = \{t^2 x_0, t^3 y_0, t z_0 : t \in \mathbb{R} \setminus \{0\}\}$.

4.2.4 Fast Addition

Let us come back to the affine case and assume that the underlying field is the prime field $\mathbb{F}_p$. Recall that for adding two points $P = (x_P, y_P)$ and $Q = (x_Q, y_Q)$ to get $R = (x_R, y_R) \leftarrow P \oplus Q$

one used the following formulas (there is no need to understand the derivation fully, just take it as a fact):

$$x_R \leftarrow \left(\frac{y_Q - y_P}{x_Q - x_P}\right)^2 - x_P - x_Q, \quad y_R \leftarrow \left(\frac{y_Q - y_P}{x_Q - x_P}\right)(x_P - x_R) - y_P \tag{73}$$

Denote by M the cost of multiplication, by S the cost of squaring, and by I the cost of inverse operation in $\mathbb{F}_p$. Note that we do not count addition/inverse costs as they are significantly lower than operations listed. Then, the cost of additing two points using above formula is 2M+1S+1I. Indeed, our computation can proceed as follows:

1. Calculate $t_1 \leftarrow (x_Q - x_P)^{-1}$, costing 1I.
2. Calculate $\lambda \leftarrow (y_Q - y_P)t_1$, costing 1M.
3. Calculate $t_2 \leftarrow \lambda^2$, costing 1S.
4. Calculate $x_R \leftarrow t_2 - x_P - x_Q$, costing almost nothing.
5. Calculate $y_R \leftarrow \lambda(x_P - x_R) - y_P$, costing 1M.

Well, there are just 4 operations in total, so what can go wrong? The problem is that we need to calculate the inverse of $(x_Q - x_P)$, which is a very, very costly operation. In fact, typically $1\mathsf{I} \gg 20\mathsf{M}$ or even worse, the ratio might reach 80 in certain cases.

Now imagine we want to add 4 points, say $P_1 \oplus P_2 \oplus P_3 \oplus P_4$: this costs $6\mathsf{M} + 3\mathsf{S} + 3\mathsf{I}$. Now we have 3 inverses, which is a lot. Finally, if we are to add much larger number of points (for example, when finding the scalar product), this gets even worse.

Projective coordinates is a way to solve this problem. The idea is to represent points in the projective form $(X : Y : Z)$, so when adding two numbers in projective form, you still get a point in a form $(X : Y : Z)$. Then, after conducting a series of additions, you can convert the point back to the affine form.

But why adding two points, say, $(X_P : Y_P : Z_P)$ and $(X_Q : Y_Q : Z_Q)$, in the projective form is more efficient? We will not derive the formulas, but trust us that they have the following form:

$$\begin{aligned} X_R &= (X_P Z_Q - X_Q Z_P)(Z_P Z_Q (Y_P Z_Q - Y_Q Z_P)^2 - (X_P Z_Q - X_Q Z_P)^2 (X_P Z_Q + X_Q Z_P)); \\ Y_R &= Z_P Z_Q (X_Q Y_P - X_P Y_Q)(X_P Z_Q - X_Q Z_P)^2 - (Y_P Z_Q - Y_Q Z_P)((Y_P Z_Q - Y_Q Z_P)^2 Z_P Z_Q \\ &\quad -(X_P Z_Q + X_Q Z_P)(X_P Z_Q - X_Q Z_P)^2); \\ Z_R &= Z_P Z_Q (X_P Z_Q - X_Q Z_P)^3. \end{aligned} \tag{74}$$

Do not be afraid, you do not need to understand how this formula is derived. But notice that despite the very scary look, there is no inversions involved! Moreover, this formula can be calculated in only $12\mathsf{M} + 2\mathsf{S}$! So all in all, this is much more effective than $2\mathsf{M} + 1\mathsf{S} + 1\mathsf{I}$.

The only inversion which is unavoidable in the projective form is the inversion of Z since after all additions (and doublings) have been made, we need to use map $(X : Y : Z) \mapsto (X/Z, Y/Z)$ to return back to the affine form. However, this inversion is done only once at the end of the computation, so it is not that costly.

Proposition 4.7. To conclude, typically, when working with elliptic curves, one uses the following strategy:

1. Convert affine points to projective form using the map $(x, y) \mapsto (x, y, 1)$.
2. Perform all operations in the projective form, which do not involve inversions.
3. Convert the result back to the affine form using the map $(X : Y : Z) \mapsto (X/Z, Y/Z)$.

This is illustrated in the Figure below.

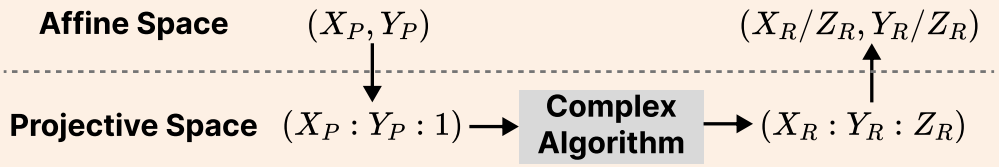


Illustration: General strategy when performing operations over Elliptic Curves.

4.2.5 Scalar Multiplication Basic Implementation

Now, the question is: how do we implement the scalar multiplication $[k]P$ for the given scalar $k \in \mathbb{Z}_r$ and point $P \in E(\overline{\mathbb{F}}_q)$?

First idea: let us simply add P to itself k times. Well, the complexity would be $O(k)$ in this case, which is even harder than solving the discrete logarithm problem (recall that the discrete logarithm problem has a complexity of $O(\sqrt{k})$). Yikes.

So there should be a better way. In this section we will limit ourselves to the double-and-add method, but the curious reader can look up the NAF (Non-Adjacent Form) method, windowed methods, GLV scalar decomposition and many other methods, which we are not going to cover in this course.

The idea of the double-and-add method is following: we represent the N-bit scalar k in binary form, say $k = (k_{N-1}, k_{N-2}, \ldots, k_0)_2$, then we calculate $P, [2]P, [4]P, [8]P, \ldots, [2^{N-1}]P$ (which is simply applying the doubling multple times) and then add the corresponding points (corresponding to positions where $k_i = 1$) to get the result. Formally, we specify the Algorithm 1.

Algorithm 1: Double-and-add method for scalar multiplication

Input : $P \in E(\mathbb{F}_q)$ and $k \in \mathbb{Z}_r$
Output: Result of scalar multiplication $[k]P \in E(\mathbb{F}_q)$

Decompose k to the binary form: $(k_0, k_1, \ldots, k_{N-1})$
$R \leftarrow \mathcal{O}$
$T \leftarrow P$
for $i \in \{0, \ldots, N-1\}$ **do**
  if $k_i = 1$ **then**
    $R \leftarrow R \oplus T$
  end
  $T \leftarrow [2]T$
end
Return : Point R

Good news: now we have a complexity of $O(N) = O(\log k)$, which is way much better that a linear one. In fact, many more optimized methods have the same assymptotic complexity

(meaning, a logarithmic one), so it turns out that we cannot do much better than that. However, the main advantages of other, more optimized methods is that we can avoid making too many additions (here, in the worst case, we have to make N additions), which is a costly operation (and more expensive than doubling).

Moreover, here we can use projective coordinates to make the addition and doubling operation more efficient! After all, typically the number of operations is even more than 300, so making 300 inversions in affine form is not an option.

4.3 Elliptic Curve Pairing

Pairing is the core object used in threshold signatures, zk-SNARKs constructions, and other cryptographic applications.

Consider the *Decisional Diffie-Hellman problem* which we described in Section 2 (based on g^α, g^β and g^γ, decide whether $\gamma = \alpha\beta$). Turns out that for curves where the so-called *embedding degree*[10] is small enough, this problem is easy to solve. This might sound like a quite bad thing, but it turns out that although some information about the discrete logarithm is leaked, it is not enough to break the security of the system (basically, solve the *Computational Diffie-Hellman problem*). Pairings is the exact object that allows us to solve the Decisional Diffie-Hellman problem.

However, a more interesting use-case which we are going to use in SNARKs is that pairings allows us to check **quadratic conditions** on scalars using their corresponding elliptic curve representation. For example, just given $u = g^\alpha, v = g^\beta$ we can check whether $\alpha\beta + 5 = 0$ (which is impossible to check without having a pairing).

So what is pairing?

4.3.1 Definition

Definition 4.8. Pairing is a bilinear, non-degenerate, efficiently computable map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1, \mathbb{G}_2$ are two groups (typically, elliptic curve groups) and $\mathbb{G}_T$ is a target group (typically, a set of scalars). Let us decipher the definition:

- **Bilinearity** means essentially the following:

$$e([a]P, [b]Q) = e([ab]P, Q) = e(P, [ab]Q) = e(P, Q)^{ab}.$$

- **Non-degeneracy** means that $e(G_1, G_2) \neq 1$ (where G_1, G_2 are generators of $\mathbb{G}_1, \mathbb{G}_2$, respectively). This property basically says that the pairing is not trivial.
- **Efficient computability** means that the pairing can be computed in a reasonable time.

The definition is illustrated in Figure 4.1.

[10] We will mention what that is is later, but still this term is quite hard to define.

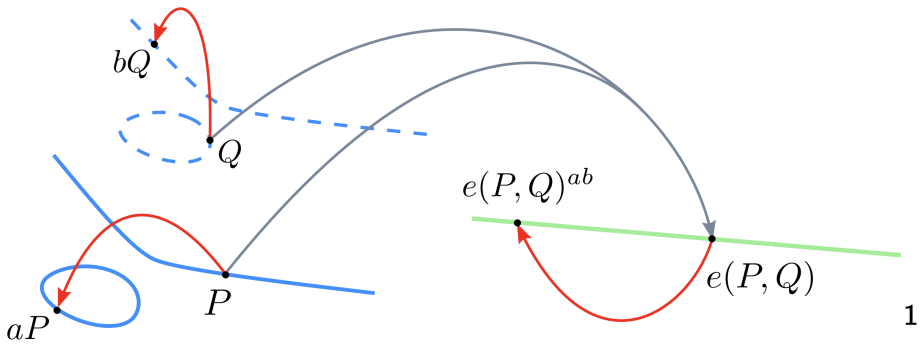


Figure 4.1: Pairing illustration. It does not matter what we do first: (a) compute $[a]P$ and $[b]Q$ and then compute $e([a]P, [b]Q)$ or (b) first calculate $e(P, Q)$ and then transform it to $e(P, Q)^{ab}$. Figure taken from "Pairings in R1CS" talk by Youssef El Housni

Example. Suppose $\mathbb{G}_1 = \mathbb{G}_2 = \mathbb{G}_T = \mathbb{Z}_r$ are scalars. Then, the map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, defined as:

$$e(x, y) = 2^{xy} \tag{75}$$

is pairing. Indeed, it is bilinear. For example, $e(ax, by) = 2^{abxy} = (2^{xy})^{ab} = e(x, y)^{ab}$ or $e(ax, by) = 2^{abxy} = 2^{(x)(aby)} = e(x, aby)$. Moreover, it is non-degenerate, since $e(1, 1) = 2 \neq 1$. And finally, it is obviously efficiently computable.
However, this is a quite trivial example since working over integers is typically not secure. For example, the discrete logarithm over $\mathbb{Z}_r$ can be solved in subexponential time. For that reason, we want to build pairings over elliptic curves.

Example. Pairing for BN254. For BN254 (with equation $y^2 = x^3 + 3$), the pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is defined over the following groups:

- $\mathbb{G}_1$ — points on the regular curve $E(\mathbb{F}_p)$.
- $\mathbb{G}_2$ — r-torsion points on the twisted curve $E'(\mathbb{F}_{p^2})$ over the field extension $\mathbb{F}_{p^2}$ (with equation $y^2 = x^3 + \frac{3}{\xi}$ for $\xi = 9 + u \in \mathbb{F}_{p^2}$).
- $\mathbb{G}_T$ — rth roots of unity $\Omega_r \subset \mathbb{F}_{p^{12}}^{\times}$.

Well, this one is quite intense and even understanding the input and output parameters is quite hard. So let us decipher some components:

- **r-torsion subgroup on the curve** $E(\mathbb{F}_{p^m})$ is simply a set of points, which multiplied by r give the point at infinity (that is, $[r]P = \mathcal{O}$). Formally, $E(\mathbb{F}_{p^m})[r] = \{P \in E(\mathbb{F}_{p^m}) : [r]P = \mathcal{O}\}$. Of course, for the curve $E(\mathbb{F}_p)$, the r-torsion subgroup is simply the whole curve, but that is generally not the case for the twisted curve over the extension field.
- **rth roots of unity** is a set of elements $\Omega_r = \{z \in \mathbb{F}_{p^{12}}^{\times} : z^r = 1\}$. This is a group under multiplication, and it has exactly r elements.

Remark. One might a reasonable question: where does this 12 come from? The answer is following: the so-called **embedding degree** of BN254 curve is $k = 12$. This number is the key to understanding why we are working over such large extensions when calculating the pairing. The formal description is quite hard, but the intuition is following: the embedding degree is the smallest number k such that all the rth roots of unity lie inside the extended field $\mathbb{F}_{p^k}$. If k was smaller, the output of pairing would contain less that r points and some points would be missing, which would make the pairing more trivial. For that reason, we need to have $\Omega_r \subset \mathbb{F}_{p^k}$.

Definition 4.9. The following conditions are equivalent **definitions** of an embedding degree k of an elliptic curve $E(\overline{\mathbb{F}}_p)$:

- k is the smallest positive integer such that $r \mid (p^k - 1)$.
- k is the smallest positive integer such that $\mathbb{F}_{p^k}$ contains all of the r-th roots of unity in $\overline{\mathbb{F}}_p$, that is $\Omega_r \subset \mathbb{F}_{p^k}$.
- k is the smallest positive integer such that $E(\overline{\mathbb{F}}_p)[r] \subset E(\mathbb{F}_{p^k})$

Pretty obvious observation: lower embedding degree is faster to work with, since it allows us to work over smaller fields. But usually, this embedding degree is quite large and we need to craft elliptic curves specifically to have a small embedding degree. For example, a pretty famous curve `secp256k1` has an embedding degree

$$k = \texttt{0x2aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa74727a26728c1ab49ff8651778090ae0},$$

which is 254-bit long. For that reason, it is natural to define the term *pairing-friendly elliptic curve*.

Definition 4.10. An elliptic curve is called **pairing-friendly** if it has a relatively small embedding degree k (typically, $k \leq 16$).

Remark. One might ask why usually, when dealing with pairings, we do not get to work with field extensions that much (most likely, if you were to write `groth16` from scratch using some mathematical libraries, you will not need to work with $\mathbb{F}_{p^{12}}$ arithmetic specifically). The reason is that typically, libraries implement the following abstraction: given a set of points $\{(P_i, Q_i)\}_{i=1}^n \subset \mathbb{G}_1^n \times \mathbb{G}_2^n$, the function checks whether

$$\prod_{i=1}^{n} e(P_i, Q_i) = 1 \tag{76}$$

Note that in this case, we do not need to work with $\mathbb{F}_{p^{12}}$ arithmetic, but rather checking the equality in the target group $\mathbb{G}_T$.
Interesting fact: this condition is specified in the `ecpairing` precompile standard used in Ethereum.

4.3.2 Case Study: BLS Signature

One of the most elegant applications of pairings is the BLS Signature scheme. Compared to ECDSA or other signature schemes, BLS can be formulated in three lines.

Suppose we have pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ (with generators G_1, G_2, respectively), and a hash function H, mapping message space $\mathcal{M}$ to $\mathbb{G}_1$.

Definition 4.11. BLS Signature Scheme consists of the following algorithms:

- $\mathsf{Gen}(1^\lambda)$: Key generation. $\mathsf{sk} \xleftarrow{R} \mathbb{Z}_q, \mathsf{pk} \leftarrow [\mathsf{sk}]G_2 \in \mathbb{G}_2$.
- $\mathsf{Sign}(\mathsf{sk}, m)$. Signature is $\sigma \leftarrow [\mathsf{sk}]\mathsf{H}(m) \in \mathbb{G}_1$.
- $\mathsf{Verify}(\mathsf{pk}, m, \sigma)$. Check whether $e(\mathsf{H}(m), \mathsf{pk}) = e(\sigma, G_2)$.

Let us check the correctness:

$$e(\sigma, G_2) = e([\mathsf{sk}]\mathsf{H}(m), G_2) = e(\mathsf{H}(m), [\mathsf{sk}]G_2) = e(\mathsf{H}(m), \mathsf{pk}) \tag{77}$$

As we see, the verification equation holds.

Remark. $\mathbb{G}_1$ and $\mathbb{G}_2$ might be switched: public keys might live instead in $\mathbb{G}_1$ while signatures in $\mathbb{G}_2$.

This scheme is also quite famous for its aggregation properties, which we are not going to consider today.

4.3.3 Case Study: Verifying Quadratic Equations

Example. Suppose Alice wants to convince Bob that he knows such α, β such that $\alpha+\beta = 2$, but she does not want to reveal α, β. She can do the following trick:

1. Alice computes $P \leftarrow [\alpha]G, Q \leftarrow [\beta]G$ — points on the curve.
2. Alice sends (P, Q) to Bob.
3. Bob verifies whether $P \oplus Q = [2]G$.

It is easy to verify the correctness of the scheme: suppose Alice is honest and she sends the correct values of α, β, satisfying $\alpha+\beta = 2$. Then, $P \oplus Q = [\alpha]G \oplus [\beta]G = [\alpha+\beta]G = [2]G$. Moreover, Bob cannot learn α, β since the computational discrete logarithm problem is hard.

Example. Well, now suppose I make the problem just a bit more complicated: Alice wants to convince that she knows α, β such that $\alpha\beta = 2$. And it turns out that elliptic curve points on their own are not enough to verify this. However, using pairings, we can do the following trick: assume we have a pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$, where $\mathbb{G}_1$ is generated by G_1 and $\mathbb{G}_2$ is generated by G_2. Then, Alice can do the following:

1. Alice computes $P \leftarrow [\alpha]G_1 \in \mathbb{G}_1, Q \leftarrow [\beta]G_2 \in \mathbb{G}_2$ — points on two curves.
2. Alice sends $(P, Q) \in \mathbb{G}_1 \times \mathbb{G}_2$ to Bob.
3. Bob checks whether: $e(P, Q) = e(G_1, G_2)^2$.

Remark. The last verification can be also rewritten as $e(P, Q)e(G_1, G_2)^{-2} = 1$, which is more frequently used in practice.

Example. Finally, let us prove something more interesting. Like, based on (x_1, x_2), whether

$$x_1^2 + x_1 x_2 = x_2 \tag{78}$$

Alice can calculate $P_1 \leftarrow [x_1]G_1 \in \mathbb{G}_1, P_2 \leftarrow [x_1]G_2 \in \mathbb{G}_2, Q \leftarrow [x_2]G_2 \in \mathbb{G}_2$. Then, the condition can be verified by checking whether

$$e(P_1, P_2 \oplus Q)e(G_1, \ominus Q) = 1 \tag{79}$$

Let us see the correctness of this equation:

$$\begin{aligned} e(P_1, P_2 \oplus Q)e(G_1, \ominus Q) &= e([x_1]G_1, [x_1 + x_2]G_2)e(G_1, [x_2]G_2)^{-1} \\ &= e(G_1, G_2)^{x_1(x_1+x_2)}e(G_1, G_2)^{-x_2} = e(G_1, G_2)^{x_1^2+x_1x_2-x_2} \end{aligned} \tag{80}$$

Now, if this is 1, then $x_1^2 + x_1 x_2 = x_2$, which was exactly what we wanted to prove.

4.4 Exercises

Exercise 1. What is **not** a valid equivalence relation $\sim$ over a set $\mathcal{X}$?

(A) $a \sim b$ iff $a + b < 0$, $\mathcal{X} = \mathbb{Q}$.
(B) $a \sim b$ iff $a = b$, $\mathcal{X} = \mathbb{R}$.
(C) $a \sim b$ iff $a \equiv b \pmod 5$, $\mathcal{X} = \mathbb{Z}$.
(D) $a \sim b$ iff the length of $a =$ the length of b, $\mathcal{X} = \mathbb{R}^2$.
(E) $(a_1, a_2, a_3) \sim (b_1, b_2, b_3)$ iff $a_3 = b_3$, $\mathcal{X} = \mathbb{R}^3$.

Exercise 2. Suppose that over $\mathbb{R}$ we define the following equivalence relation: $a \sim b$ iff $a - b \in \mathbb{Z}$ $(a, b \in \mathbb{R})$. What is the equivalence class of 1.4 (that is, $[1.4]_\sim$)?

(A) A set of all real numbers.
(B) A set of all integers.
(C) A set of reals $x \in \mathbb{R}$ with the fractional part of x equal to 0.4.
(D) A set of reals $x \in \mathbb{R}$ with the integer part of x equal to 1.
(E) A set of reals $x \in \mathbb{R}$ with the fractional part of x equal to 0.6.

Exercise 3. Which of the following pairs of points in homogeneous projective space $\mathbb{P}^2(\mathbb{R})$ are **not** equivalent?

(A) $(1 : 2 : 3)$ and $(2 : 4 : 6)$.
(B) $(2 : 3 : 1)$ and $(6 : 9 : 3)$.
(C) $(5 : 5 : 5)$ and $(2 : 2 : 2)$.
(D) $(4 : 3 : 2)$ and $(16 : 8 : 4)$.

Exercise 4. The main reason for using projective coordinates in elliptic curve cryptography is:

(A) To reduce the number of point additions in algorithms involving elliptic curves.
(B) To make the curve more secure against attacks.
(C) To make the curve more efficient in terms of memory usage.
(D) To reduce the number of field multiplications when performing scalar multiplication.

(E) To avoid making too many field inversions in complicated algorithms involving elliptic curves.

Exercise 5. Suppose $k = 19$ is a scalar and we are calculating $[k]P$ using the double-and-add algorithm. How many elliptic curve point addition operations will be performed?

(A) 0.
(B) 1.
(C) 2.
(D) 3.
(E) 4.

Exercise 6. What is the minimal number of inversions needed to calculate the value of expression (over $\mathbb{F}_p$)

$$\frac{a-b}{(a+b)^4} + \frac{c}{a+b} + \frac{d}{a^2+c^2},$$

for the given scalars $a, b, c, d \in \mathbb{F}_p$?

(A) 1.
(B) 2.
(C) 3.
(D) 4.
(E) 5.

Exercise 7. Given pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ with G_1 — generator of $\mathbb{G}_1$ and $G_2 \in \mathbb{G}_2$ — generator of $\mathbb{G}_2$, which of the following is **not** equal to $e([3]G_1, [5]G_2)$?

(A) $e([5]G_1, [3]G_2)$.
(B) $e([4]G_1, [4]G_2)$.
(C) $e([15]G_1, G_2)$.
(D) $e([3]G_1, G_2)e(G_1, [12]G_2)$.
(E) $e(G_1, G_2)^{15}$.

Exercise 8*. *Unit Circle Proof.* Suppose Alice wants to convince Bob that she knows a point on the unit circle $x^2 + y^2 = 1$. Suppose we are given a symmetric pairing $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ for $\mathbb{G}_1 = \mathbb{G}_2 = \langle G \rangle$ and Alice computes $P \leftarrow [x]G, Q \leftarrow [y]G$. She then proceeds to sending (P, Q) to Bob. Which of the following checks should Bob perform to verify that Alice indeed knows a point on the unit circle?

(A) Check if $e(P, Q)e(Q, P) = 1$.
(B) Check if $e([2]P, [2]Q) = e(G, G)$.
(C) Check if $e([2]P, Q)e(Q, [2]P) = 1$.
(D) Check if $e(P, P) + e(Q, Q) = 1$.
(E) Check if $e(P, P)e(Q, Q) = e(G, G)$.

5 Commitment Schemes

5.1 Commitments

Definition 5.1. A cryptographic commitment scheme allows one party to commit to a chosen statement (such as a value, vector, or polynomial) without revealing the statement itself. The commitment can be revealed in full or in part at a later time, ensuring the integrity and secrecy of the original statement until the moment of disclosure.

Before delving into the details, here is the intuition of cryptographic commitments.

Imagine putting a letter with some message into a box and locking it with your key. You then give that box to your friend, who cannot open it without the key. In this scenario, you have made a commitment to the message inside the box. You cannot change the content of the letter, as it is in your friend's possession. At the same time, your friend cannot access the letter since they do not have the key to unlock the box.

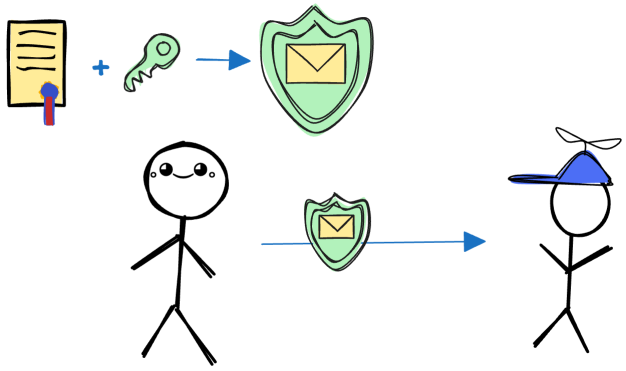


Figure 5.1: Commitment scheme

Definition 5.2 (Commitment Scheme)**.** Commitment Scheme $\Pi_{\text{commitment}}$ is a tuple of three algorithms: $\Pi_{\text{commitment}} = (\mathsf{Setup}, \mathsf{Commit}, \mathsf{Verify})$.

1. $\mathsf{Setup}(1^\lambda)$: returns public parameter pp for both comitter and verifier;
2. $\mathsf{Commit}(\mathsf{pp}, m)$: returns a commitment c to the message m using public parameters pp and, optionally, a secret opening hint r;
3. $\mathsf{Open}(\mathsf{pp}, c, m, r)$: verifies the opening of the commitment c to the message m with an opening hint r.

Definition 5.3 (Commitment Scheme)**.** Properties of commitment schemes:

1. *Hiding:* verifier should not learn any information about the message given only the commitment c. To put it formally, we define a game:
   (a) Adversary chooses two messages m_1, m_2 and sends to the challenger.
   (b) Challenger chooses a random bit b, commits to both messages: $c_1 \leftarrow \mathsf{Commit}(pp, m_1)$, $c_2 \leftarrow \mathsf{Commit}(pp, m_2)$, and sends c_b to the adversary.
   (c) Adversary guesses a bit $\hat{b}$.

   We define the hiding advantage of a PPT adversary $\mathcal{A}$ as

$$\mathsf{HideAdv}[\mathcal{A}, \Pi_{\mathsf{commitment}}] := \left| \Pr[b = \hat{b}] - \frac{1}{2} \right| \tag{81}$$

   We say that the commitment scheme $\Pi_{\mathsf{commitment}}$ is *hiding* if for any adversary, the aforementioned advantage is negligible.
2. *Binding:* prover could not find another message m_1 and open the commitment c without revealing the commited message m. To put it formally, we define a game:
   (a) Adversary chooses five values: commitment c and two distinct pairs (m_0, r_0) and (m_1, r_1).
   (b) Adversary computes $b_j \leftarrow \mathsf{Open}(\mathsf{pp}, c, m_j, r_j)$.

   Define the advantage in the binding game as:

$$\mathsf{BindAdv}[\mathcal{A}, \Pi_{\mathsf{commitment}}] = \Pr[b_0 = b_1 \neq 0 \wedge m_0 \neq m_1] \tag{82}$$

   We say that the commitment scheme is binding if for any adversary, such advantage is negligible.

5.1.1 Hash-based commitments

As the name implies, we are using a cryptographic hash function H in such scheme.

1. Prover selects a message m from a message space M which he wants to commit to: $m \leftarrow \mathbb{M}$
2. Prover samples random value r (usually called blinding factor) from a challange space $\mathcal{C} \subset \mathbb{Z}$: $r \xleftarrow{R} \mathcal{C}$
3. Both values will be concatenated and hashed with the hash function H to produce the commitment: $c = H(m \parallel r)$

Commitment should be shared with a verifier. During the opening stage, prover reveals (m, r) to the *Verifier*. To check the commitment, verifier computes: $c_1 = H(m \parallel r)$.

If $c_1 = c$, prover has revealed the correct pair (m, r).

It should be noted that a cryptographic hash function aims to provide collision resistance, meaning that the probability two different messages will result in one output is negligible. Because the *Verifier* knows the hash function digest c before the *Prover* reveals m and r, the *Prover* would need to find a collision $H(m' \parallel r') = H(m \parallel r)$ to be able to convince the *Verifier* that m' value was committed.

However, due to the collision resistance, finding such m' and r' is computationally infeasible.

Which means the *Prover* won't be able to convince the *Verifier* that the commitment was done to another value providing a *binding* property.

A cryptographically secure hash function is a one-way function, which means that finding the hash preimage is almost as hard as bruteforcing all possibile input values. Given large challenge space, the probability of the *Verifier* of finding (m, r) such that $H(m, r) = c$ is negligible, which ensures *hiding* property of the commitment scheme.

5.1.2 Pedersen commitments

Pedersen commitments allow us to represent arbitrarily large vectors with a single elliptic curve point, while optionally hiding any information about the vector. Pedersen commitment uses a public group $\mathbb{G}$ of order q and two random public generators G and U: $U = [u]G$. Secret parameter u should be unknown to anyone, otherwise the *Binding* property of the commitment scheme will be violated. EC point U is chosen randomly using "Nothing-up-my-sleeve" to assure no one knows the discrete logarithm of a selected point.

Remark. Transparent random points generation
User can pick the publicly chosen random number (like a hash of project name, first numbers of π, etc), and hash that result to obtain another value. If that results in an x value that lies on the elliptic curve, use that as the random point and hash the (x, y) pair again (to obtain the next one, it needed). Otherwise, if the x-value does not land on the curve, increment x until it does. Because the committer is not generating the points, they don't know their discrete log.

Pedersen commitment scheme algorithm:

1. Prover and Verifier agrees on G and U points in a elliptic curve point group $\mathbb{G}$, q is the order of the group.
2. Prover selects a value m to commit and a blinder factor r: $m \leftarrow \mathbb{Z}_q$, $r \xleftarrow{R} \mathbb{Z}_q$
3. Prover generates a commitment and sends it to the Verifier: $c \leftarrow [m]G + [r]U$

During the opening stage, prover reveals (m, r) to the verifier. To check the commitment, verifier computes: $c_1 = [m]G + [r]U$.

If $c_1 = c$, prover has revealed the correct pair (m, r).

Remark. In case the discrete logarithm of U is leaked, the *binding* property can be violated by the *Prover*:

$$c = [m]G + [r]U = [m]G + [r \cdot u]G = [m + r \cdot u]G$$

For example, $(m + u, r - 1)$ will have the same commitment value:

$$[m + u + (r - 1) \cdot u]G = [m + u - u + r \cdot u]G = [m + r \cdot u]G$$

Commitment aggregation

Pedersen commitment have some advantages compared to hash-based commitments. Additively homomorphic property allows to accumulate multiple commitments into one. Consider two pairs: $(m_1, r_1), (m_2, r_2)$.

$$c_2 = [m_1]G + [r_1]U,$$

$$c_2 = [m_2]G + [r_2]U,$$
$$c_a = c_1 + c_2 = [m_1 + m_2]G + [r_1 + r_2]U$$

This works for any number of commitments, so we can encode as many points as we like in a single one. For example, if a set of balances is committed, the sum of any subset can be proven without revealing the exact value of each balance. This is achieved by disclosing the sum of the balances and the corresponding sum of the blinding factors.

5.1.3 Vector commitments

Vector commitment schemes allows to commit to a vector of values rather than a value and a blinding term.

Pedersen Vector Commitments

Suppose we have a set of random elliptic curve points $(G_1, \ldots, G_n)$ of cyclic group $\mathbb{G}$ (that we do not know the discrete logarithm of), a vector $(m_1, m_2 \ldots m_n)$ and a random value r. We can do the following:

$$c = [m_1]G_1 + [m_2]G_2 \ldots + [m_n]G_n + [r]Q$$

Since the *Prover* does not know the discrete logarithm of the generators, they don't know the discrete logarithm of [C]. Hence, this scheme is binding: they can only reveal $(v_1, \ldots, v_n)$ to produce $[C]$ later, they cannot produce another vector.

Prover can later open the commitment by revealing the vector $(m_1, m_2 \ldots m_n)$ and a blinding term r.

Merkle Tree based Vector Commitments

A naive approach for a vector commitment would be hash the whole vector. More sophisticated scheme uses divide-and-conquer approach by building a binary tree out of vector elements.

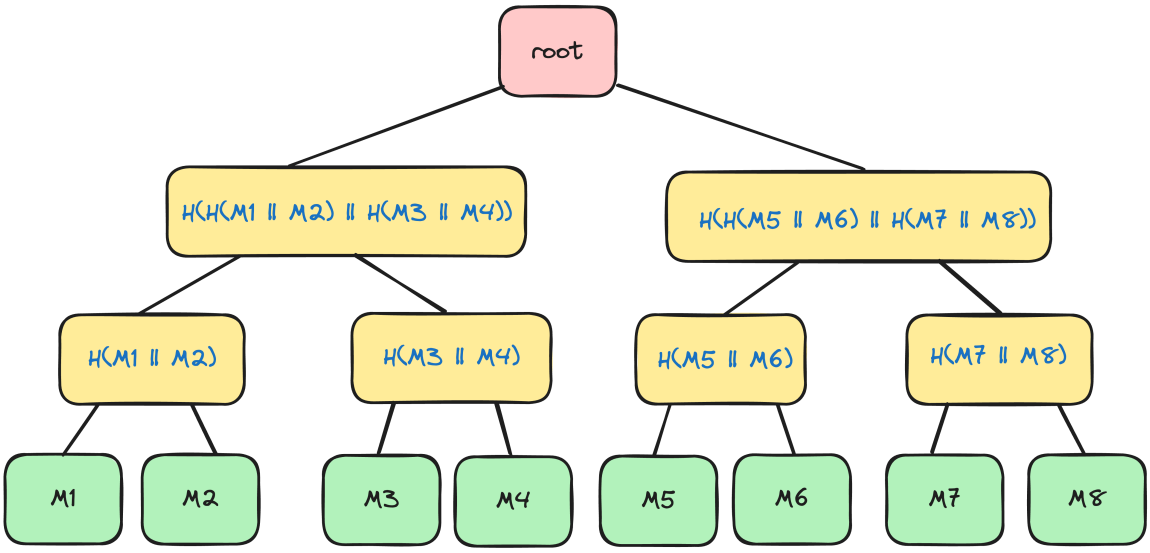


Figure 5.2: Merkle Tree structure

A Merkle Tree is a data structure to efficiently and securely verify the commiments to a vector of data. It is a binary tree where each leaf node represents a hash of a data block, and each non-leaf node is a hash of its child nodes' concatenated hashes. The top node, called the root hash or Merkle root, uniquely represents the entire data set. By comparing this root with a known valid root, one can quickly verify the authenticity and integrity of the data without needing to examine the entire dataset.

To prove the inclusion of element into the tree, a corresponding Merkle Branch is used. On the example below, M_1 inclusion is proved, and $(M_2, H(M_3 \parallel M_4), H(H(M_5 \parallel M_6) \parallel H(M_7 \parallel M_8)))$ is an inclusion branch vector.

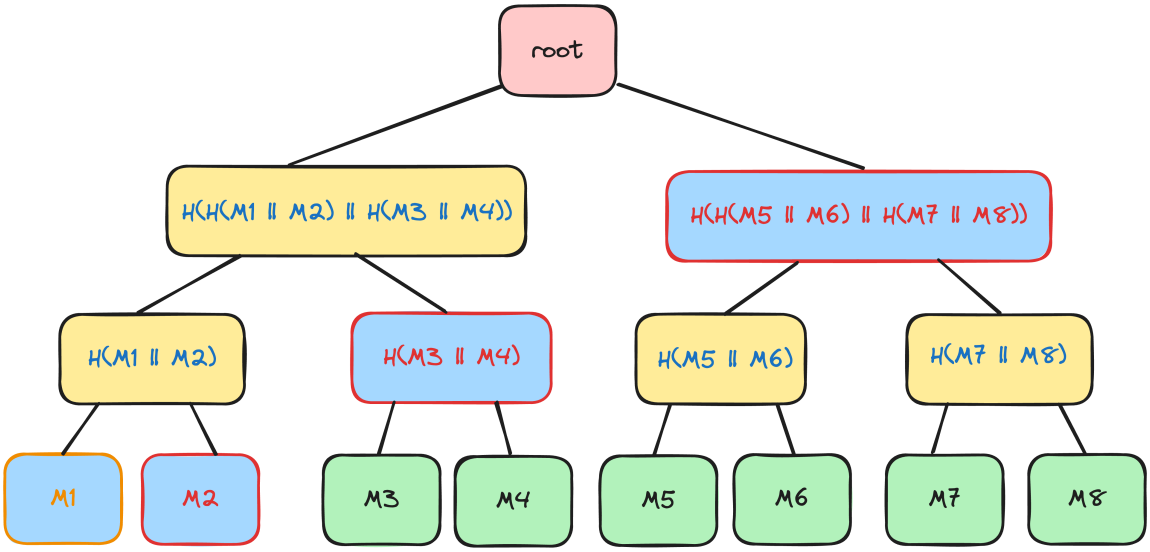


Figure 5.3: Merkle Tree inclusion proof branch

One of Merkle tree key advantages is that it allows for the selective disclosure of specific elements within the data set without revealing the rest.

5.1.4 Polynomial commitment

Polynomial commitment can be used to prove that the commited polynomial satisfies certain properties $P(x_1, x_2, \ldots, x_n) = y$, without revealing what the polynomial is. The commitment is generally succint, which means that it is much smaller than the polynomial it represents.

The KZG polynomial commitment scheme

The KZG (Kate-Zaverucha-Goldberg) is a polynomial commitment scheme:

1. *One-time "Powers-of-tau" trusted setup stage*. During trusted setup a set of elliptic curve points is generated. Let G be a generator point of some pairing-friendly elliptic curve group $\mathbb{G}$, s some random value in the order of the G point and d be the maximum degree of the polynomials we want to commit to. Public parameters of a trusted setup are calculated as:

$$[\tau^0]G, [\tau^1]G, \ldots, [\tau^d]G$$

Parameter τ should be deleted after the ceremony. If it is revealed, the *binding* property of the commitment scheme can be broken. This parameter is usually called the *toxic waste*.

2. *Commit to polynomial*. Given the polynomial $p(x) = \sum_{i=0}^{d} p_i x^i$, compute the commitment $c = [p(\tau)]G$ using the trusted setup. Although the committer cannot compute $p(\tau)$ directly since the value of τ is unknown, he can compute it using values $([\tau^0]G, [\tau^1]G, \ldots, [\tau^d]G)$:

$$[p(\tau)]G = [\sum_{i=0}^{d} p_i \tau^i]G = \sum_{i=0}^{d} p_i [\tau^i]G$$

3. *Prove an evaluation*. To prove that at some point x_0 polynomial equals y_0 ($p(x_0) = y_0$), compute polynomial

$$q(x) = \frac{p(x)-y_0}{x-x_0}.$$

Polynomial $q(x)$ is called "quotient polynomial" and only exists if and only if $p(x_0) = y_0$:
   (a) If $p(x_0) = y_0$, we define $r(x) := p(x) - y_0$;
   (b) $r(x)$ has x_0 as a root, as $r(x_0) = 0$ by the definition. That is why there exists $q(x)$, such that $r(x) = q(x) \cdot (x - x_0)$;
   (c) Hence, the expression $q(x) = \frac{p(x)-y_0}{x-x_0}$ is a polynomial.

   The existance of this quotient polynomial serves as a proof of the evaluation. *Prover* calculates proof $\pi = [q(\tau)]G$ and sends it to the *Verifier*.

4. *Verify the proof*. Given a commitment $c = [p(\tau)]G$, an evaluation $p(x_0) = y_0$ and a proof $[q(\tau)]G$, we need to ensure that $q(\tau) \cdot (\tau - x_0) = p(\tau) - y_0$. This can be done using trusted setup without knowledge of τ using bilinear mapping:

$$e([q(\tau)]G_1, [\tau]G_2 - [x']G_2) = e([p(\tau)]G_1 - [y]G_1, G_2)$$

   Polynomial commiment schemes such as KZG are used in zero knowledge proof system to encode circuit constraints as a polynomial, so that verifier could check random points to ensure that the constraints are met.

5.2 Exercises

Exercise 1. Dmytro and Denis were watching a horse race. Confident in his ability to predict the outcome, Dmytro decided to commit to his prediction. However, in his haste, he forgot to use a blinding factor. Now, Dmytro is concerned that Denis might discover his prediction before the race ends, which would defeat the purpose of his commitment.

We define a dummy hash function $H(a) = (a \cdot 13 + 17) \pmod{41}$. Dmytro used a *hash-based commitment* and H as a hash function. Set of race horse numbers is $(3, 5, 8, 15)$. Help Denis to find out the horse number Dmytro have made a commitment to, if commitment equals $C = 39$.

(A) 3.
(B) 5.
(C) 8.
(D) 15.

Exercise 2. Denis made a setup (points G and U) for a Pedersen commitment scheme and commited values $(m, r) = (3, 7)$ to Dmytro by sending him $\mathcal{C} = [3]G + [7]U$. Dmytro did not

verify the setup. Turns out that Denis knows that $U = [6]G$. Denis is planning to send a different message from the one he originally committed to to $m_2 = 15$. Which values (m_2, r_2) should he send to Dmytro at the opening stage?

(A) $(15, 5)$
(B) $(15, 7)$
(C) $(15, 4)$
(D) $(3, 5)$

Exercise 3. We define a dummy hash function $H(a, b) = (a \cdot 3 + b \cdot 7) \pmod{41}$. You have a Merkle tree built with depth 4 using hash function H with root equal 37. Which inclusion proof is valid for element 3? Position defines how leaves should be hashed:

- if $left \rightarrow h_i = Hash(h_{i-1}, branch[i])$
- if $right \rightarrow h_i = Hash(branch[i], h_{i-1})$

(A) branch: $[4, 16, 13]$, position: $[left, right, left]$
(B) branch: $[1, 40, 3]$, position: $[left, left, left]$
(C) branch: $[5, 12, 13]$, position: $[right, right, left]$
(D) branch: $[4, 17, 13]$, position: $[left, right, left]$

Exercise 4. Given a polynomial $p(x) = x^3 - 10x^2 + 31x - 30$, Oleksandr wants to prove that $p(2) = 0$. To do that, according to the KZG commitment scheme, he constructs the quotient polynomial $q(x)$ and wants to show that $q(\tau) \cdot (\tau - 2) = p(\tau)$. Assuming Oleksandr has conducted these steps correctly, what value of $q(x)$ has Oleksandr calculated?

(A) $q(x) = 2x^2 + 4x - 6$
(B) $q(x) = x^3 - 10x^2 + 30x - 28$
(C) $q(x) = x^2 - 8x + 15$
(D) $q(x) = x^2 + 5x + 18$

6 Introduction to Zero-Knowledge Proofs

6.1 Motivation

Finally, we came to the most interesting part of the course: zero-knowledge proofs. Before we start with SNARKs, STARKs, Bulletproofs, and other zero-knowledge proof systems, let us first define what the zero-knowledge is. But even before that, we need to introduce some formalities. For example, what are "proof", "witness", and "statement" — terms that are so widely used in zero-knowledge proofs.

Let us describe the typical setup. We have two parties: **prover** $\mathcal{P}$ and a **verifier** $\mathcal{V}$. The prover wants to convince the verifier that some statement is true. Typically, the statement is not obvious (well, that is the reason for building proofs after all!) and therefore there might be some "helper" data, called **witness**, that helps the prover to prove the statement. The reasonable question is whether the prover can simply send witness to verifier and call it a day. Of course since you are here, reading this lecture, it is obvious that the answer is no. More specifically, by introducing zk-SNARKs, STARKs, and other proving systems, we will try to mitigate the following issues:

- **Zero-knowledge:** The prover wants to convince the verifier that the statement is true without revealing the witness.
- **Argument of knowledge:** Moreover, typically we want to make sure that the verifier, besides the statement correctness, ensures that the prover **knows** such a witness related to the statement.
- **Succinctness:** The proof should be short, ideally logarithmic in the size of the statement. This is crucial for practical applications, especially in the blockchain space where we cannot allow to publish long proofs on-chain. Moreover, verification should be efficient as well.

Example. Suppose, given a hash function[a] $H : \{0,1\}^* \to \{0,1\}^\ell$, the prover $\mathcal{P}$ wants to convince the verifier $\mathcal{V}$ that he knows the preimage $x \in \{0,1\}^*$ such that $H(x) = y$ for some given public value $y \in \{0,1\}^\ell$. The properties listed above are interpreted as follows:

- **Zero-knowledge:** The prover $\mathcal{P}$ does not want to reveal *anything* about the pre-image x to the verifier $\mathcal{V}$.
- **Argument of knowledge:** Given a string $y \in \{0,1\}^\ell$ it is not sufficient for a prover to merely state that y has a pre-image. The prover $\mathcal{P}$ must demonstrate to a verifier $\mathcal{V}$ that he **knows** such a pre-image $x \in \{0,1\}^*$.
- **Succinctness:** If the hash function takes n operations to compute[b], the proof should be **much** shorter than n operations. State-of-art solutions can provide proofs that are $O((\log n)^c)$ (polylogarithmic) is size! Moreover, verification time of such proof is also typically polylogarithmic (or even $O(1)$ in some cases).

[a] The notation $\{0,1\}^*$ means binary strings of arbitrary length

[b] Note that "number of operations" is very vague term. One way to measure the "size" of some computational problem is specifying the number of gates in the arithmetical circuit $\mathtt{C}(x, w)$, representing the computation of this problem (denoted as $|\mathtt{C}|$, respectively).

6.2 Relations and Languages

Recall that **relation** $\mathcal{R}$ is just a subset of $\mathcal{X} \times \mathcal{Y}$ for two arbitrary sets $\mathcal{X}$ and $\mathcal{Y}$. Now, we are going to introduce the notion of a *language of true statements* based on $\mathcal{R}$.

Definition 6.1 (Language of true statements)**.** Let $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{Y}$ be a relation. We say that a statement $x \in \mathcal{X}$ is a **true** statement if $(x, y) \in \mathcal{R}$ for some $y \in \mathcal{Y}$, otherwise the statement is called **false**. We define by $\mathcal{L}_{\mathcal{R}}$ (the language over relation $\mathcal{R}$) the set of all true statements, that is:

$$\mathcal{L}_{\mathcal{R}} = \{x \in \mathcal{X} : \exists y \in \mathcal{Y} \text{ such that } (x, y) \in \mathcal{R}\}.$$

Now, what is the purpose of introducing relations and languages? The idea is that relation is a natural way to formalize the notion of a statement and witness. Namely, we denote the elements of $\mathcal{X}$ as statements and the elements of $\mathcal{Y}$ as witnesses.

Further, we denote by w the witness for the statement $x \in \mathcal{L}_{\mathcal{R}}$. Oftentimes, one might also encounter notation ϕ to denote the statement, but we will stick to x for simplicity.

Example. Suppose we want to prove the following claim: number $n \in \mathbb{N}$ is the product of two large prime numbers $(p, q) \in \mathbb{N} \times \mathbb{N}$. Here, the relation is the following:

$$\mathcal{R} = \{(n, p, q) \in \mathbb{N}^3 : n = p \cdot q \text{ where } p, q \text{ are primes}\}$$

In this particular case, the language of true statements is defined as

$$\mathcal{L}_{\mathcal{R}} = \{n \in \mathbb{N} : \exists p, q \text{ are primes such that } n = pq\}$$

Therefore, our initial claim we want to prove is $n \in \mathcal{L}_{\mathcal{R}}$. The witness for this statement is the pair (p, q), where p and q are prime numbers such that $n = p \cdot q$ and typically (but not always) we want to prove this without revealing our witness: p and q. For example, one valid witness for $n = 15$ is $(3, 5)$, while $n = 16$ does not have any valid witness, so $16 \notin \mathcal{L}_{\mathcal{R}}$.

Example. Another example of claim we want to prove is the following: number $x \in \mathbb{Z}_N^{\times}$[a] is a **quadratic residue** modulo N, meaning there exists some $w \in \mathbb{Z}_N^{\times}$ such that $x \equiv w^2 \pmod N$ (naturally, w is called a *square root* of x). The relation in this case is:

$$\mathcal{R} = \{(x, w) \in (\mathbb{Z}_N^{\times})^2 : x \equiv w^2 \pmod N\}$$

In this case, $\mathcal{L}_{\mathcal{R}} = \{x \in \mathbb{Z}_N^{\times} : \exists w \in \mathbb{Z}_N^{\times} \text{ such that } x \equiv w^2 \pmod N\}$. Here, our square root of x modulo N, that is w, is the witness for the statement $x \in \mathcal{L}_{\mathcal{R}}$. For example, for $N = 7$ we have $4 \in \mathcal{L}_{\mathcal{R}}$ since 5 is a valid witness: $5^2 \equiv 4 \pmod 7$, while $3 \notin \mathcal{L}_{\mathcal{R}}$ since there is no valid witness for 3.

[a] By $\mathbb{Z}_N^{\times}$ we denote the multiplicative group of integers modulo N. In other words, this is a set of integers $\{x \in \mathbb{Z}_N : \gcd\{x, N\} = 1\}$.

However, we want to limit ourselves to languages that has reasonably efficient verifier (since otherwise the problem is not really practical and therefore of little interest to us). For that reason, we define the notion of a *NP Language* and from now on, we will be working with such

languages.

Definition 6.2 (NP Language)**.** A language $\mathcal{L}_{\mathcal{R}}$ belongs to the NP class if there exists a polynomial-time verifier $\mathcal{V}$ such that the following two properties hold:

- **Completeness:** If $x \in \mathcal{L}_{\mathcal{R}}$, then there is a witness w such that $\mathcal{V}(x, w) = 1$ with $|w| = \mathsf{poly}(|x|)$. Essentially, it states that true claims have *short*[a] proofs.
- **Soundness:** If $x \notin \mathcal{L}_{\mathcal{R}}$, then for any w it holds that $\mathcal{V}(x, w) = 0$. Essentially, it states that false claims have no proofs.

[a] "Short" is a pretty relative term which would further differ based on the context. Here, we assume that the proof is "short" if it can be computed in polynomial time. However, in practice, we will want to make the proofs even shorter: see SNARKs and STARKs.

However, this construction on its own is not helpful to us. In particular, without having any randomness and no interaction, building practical proving systems is very hard. Therefore, we need some more ingredients to make proving NP statements easier.

6.3 Interactive Probabilistic Proofs

As mentioned above, we will bring two more ingredients to the table: **randomness** and **interaction**:

- **Interaction:** rather than simply passively receiving the proof, the verifier $\mathcal{V}$ can interact with the prover $\mathcal{P}$ by sending challenges and receiving responses.
- **Randomness:** verifier can send random coins (challenges) to the prover, which the prover can use to generate responses.

Remark. For those who have already worked with SNARKs, the above introduction might seem very confusing. After all, what we are aiming for is to build **non-interactive** zero-knowledge proofs. However, as it turns out, there are a plenty of ways to make *some* interactive proofs non-interactive. We will discuss this in more detail later.

Now, one of the drastic changes is the following: if $x \notin \mathcal{L}_{\mathcal{R}}$, then the verifier $\mathcal{V}$ should reject the claim with overwhelming[11] probability (in contrast to strict probability of 1). Let us consider the concrete example.

6.3.1 Example: Quadratic Residue Test

Again, suppose for relation $\mathcal{R} = \{(x, w) \in (\mathbb{Z}_N^\times)^2 : x \equiv w^2 \pmod N\}$ and corresponding language $\mathcal{L}_{\mathcal{R}} = \{x \in \mathbb{Z}_N^\times : \exists w \in \mathbb{Z}_N^\times$ such that $x \equiv w^2 \pmod N\}$ the prover $\mathcal{P}$ wants to convince the verifier that the given x is in language $\mathcal{L}_{\mathcal{R}}$. Again, sending w is not an option, as we want to avoid revealing the witness. So how can we proceed? The idea is that the prover $\mathcal{P}$ should prove that he *could* prove it if he felt like it.

So here how it goes. The prover $\mathcal{P}$ can first sample a random $r \xleftarrow{R} \mathbb{Z}_N^\times$, calculate $a \leftarrow r^2 \pmod N$ and say to the verifier $\mathcal{V}$:

- Hey, I could give you the square roots of both a and $ax \pmod N$ and that would convince

[11] Some technicality: as you know from the Lecture 2, the value $\varepsilon = \mathsf{negl}(\lambda)$ is called negligible since it is very close to 0. In turn, the value $1 - \varepsilon$ is called *overwhelming* since it is close to 1.

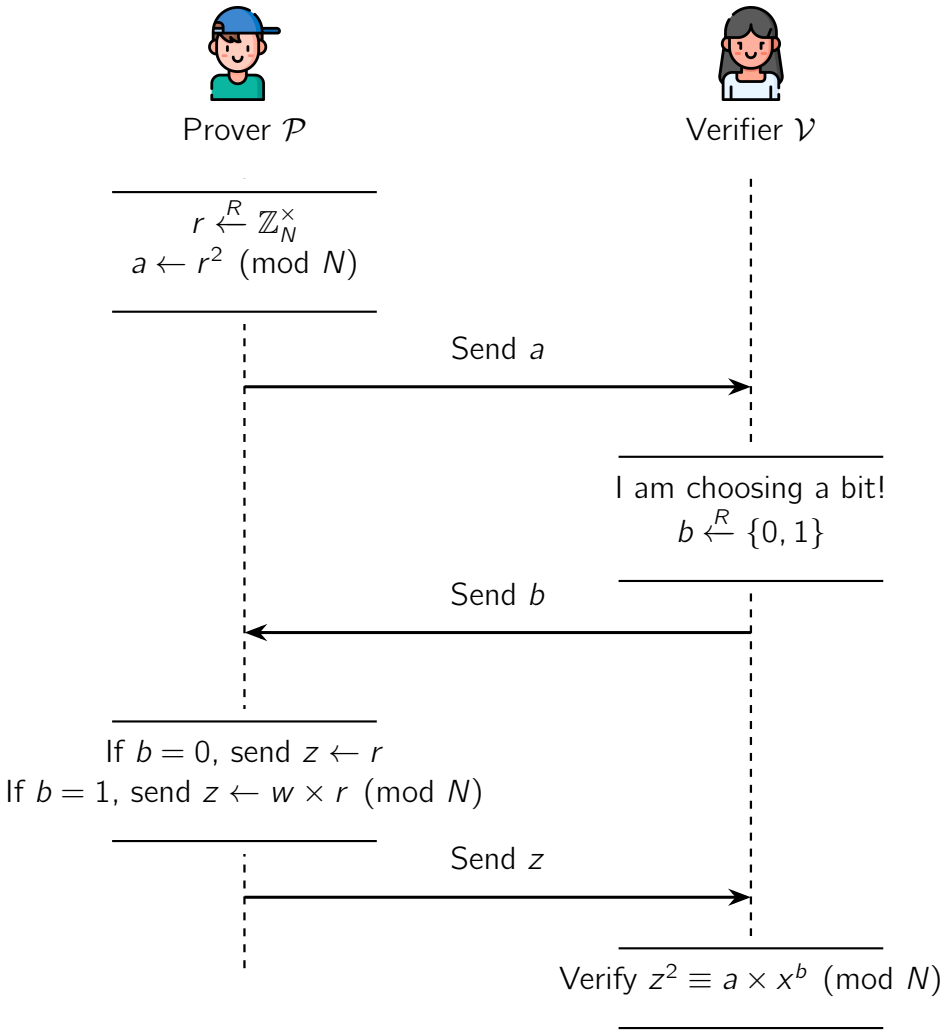


Figure 6.1: The interactive protocol between prover $\mathcal{P}$ and verifier $\mathcal{V}$ for the quadratic residue test.

you that the statement is true! But in this case, you would know w[12].

- So instead of providing both values simultaneously, you will choose which one you want to see: either r or $r \times w \pmod N$. This way, after a couple of such rounds, you will not learn w but you will be convinced that I know it.

That being said, formally the interaction between prover $\mathcal{P}$ and verifier $\mathcal{V}$ can be described as follows:

1. $\mathcal{P}$ samples $r \xleftarrow{R} \mathbb{Z}_N^\times$ and sends $a \leftarrow r^2 \pmod N$ to $\mathcal{V}$.
2. $\mathcal{V}$ sends a random bit $b \xleftarrow{R} \{0,1\}$ to $\mathcal{P}$.
3. If $b = 0$, the prover sends $z \leftarrow r$, otherwise, if $b = 1$, he sends $z \leftarrow rw \pmod N$.
4. The verifier checks whether $z^2 \equiv a \times x^b \pmod N$.
5. Repeat the process for $\lambda \in \mathbb{N}$ rounds.

Now, let us show that the provided protocol is indeed **complete** and **sound**.

[12] If verifier gets both r and $rw \pmod N$, he can divide the latter by former and get w

Completeness. Suppose the verifier chose $b = 0$ and thus the prover has sent $z = r$. The check would be $r^2 \equiv a \times x^0 \pmod N$ which is equivalent to $r^2 \equiv a \pmod N$. This obviously holds.

If, in turn, the verifier chose $b = 1$ and the prover sent rw, the check would be $(rw)^2 \equiv ax^{1-0} \pmod N$ which is equivalent to $r^2w^2 \equiv ax \pmod N$. Since $a = r^2 \pmod N$ and $x = w^2 \pmod N$, this check also obviously holds.

Soundness. Here, we need to prove that for any dishonest prover who does not know w, the verifier will reject the claim with overwhelming probability. One can show the following, which we are not going to prove (yet, this is quite easy to show):

Proposition 6.3. If $x \notin \mathcal{L}_\mathcal{R}$, then for any prover $\mathcal{P}$, the verifier $\mathcal{V}$ will reject the claim with probability at least $1/2$.

By making λ rounds, the probability of rejection is $\left(\frac{1}{2}\right)^\lambda = \mathsf{negl}(\lambda)$ and therefore the verifier can be convinced that $x \in \mathcal{L}_\mathcal{R}$ with overwhelming probability of $1 - 2^{-\lambda}$.

To denote the interaction between algorithms $\mathcal{P}$ and $\mathcal{V}$ on the statement x, we use notation $\langle \mathcal{P}, \mathcal{V} \rangle(x)$. Finally, now we are ready to define the notion of an **interactive proof system**.

Definition 6.4. A pair of algorithms $(\mathcal{P}, \mathcal{V})$ is called an **interactive proof** for a language $\mathcal{L}_\mathcal{R}$ if $\mathcal{V}$ is a polynomial-time verifier and the following two properties hold:

- **Completeness:** For any $x \in \mathcal{L}_\mathcal{R}$, $\Pr[\langle \mathcal{P}, \mathcal{V} \rangle(x) = \mathsf{accept}] = 1$.
- **Soundness:** For any $x \notin \mathcal{L}_\mathcal{R}$ and for any prover $\mathcal{P}^*$, we have

$$\Pr[\langle \mathcal{P}^*, \mathcal{V} \rangle(x) = \mathsf{accept}] \leq \mathsf{negl}(\lambda)$$

Definition 6.5. Besides classes **P** and **NP**, we now have one more class: **the class of interactive proofs** (**IP**):

$$\mathbf{IP} = \{\mathcal{L} : \text{there is an interactive proof } (\mathcal{P}, \mathcal{V}) \text{ for } \mathcal{L}\}.$$

6.4 Zero-Knowledge

Turns out that defining the zero-knowledge to even such a simplistic interactive proof system is not that easy. Informally, we give the following definition.

Definition 6.6. An interactive proof system $(\mathcal{P}, \mathcal{V})$ is called **zero-knowledge** if for any polynomial-time verifier $\mathcal{V}^*$ and any $x \in \mathcal{L}_\mathcal{R}$, the interaction $\langle \mathcal{P}, \mathcal{V}^* \rangle(x)$ gives nothing new about the witness w.

Definition 6.7. The pair of algorithms $(\mathcal{P}, \mathcal{V})$ is called a **zero-knowledge interactive protocol** if it is *complete*, *sound*, and *zero-knowledge*.

Basically, the specified interaction is a **proof**! The prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ that the statement is true without revealing the witness – that is what we need (quite of).

Remark. The above definition is very informal and, for the most part, complete for the purposes of this course. If you do not want to dive into the formalities, you can skip the next part of this section. However, if you are curious about some technicalities, feel free to continue reading.

6.4.1 The Verifier's View

Suppose that the interaction between $\mathcal{V}$ and $\mathcal{P}$ has ended with the successful verification. What has $\mathcal{V}$ learned? Well, first things first, he has learned that the statement is true, that is $x \in \mathcal{L}_{\mathcal{R}}$. However, he has also learned something more: he has learned the transcript of the interaction, that is the sequence of messages between $\mathcal{P}$ and $\mathcal{V}$.

Definition 6.8. Interaction between $\mathcal{P}$ and $\mathcal{V}$ consists of prover's messages (e.g., commitments and responses) $(m_1, m_2, \ldots, m_\ell)$, verifier's queries $(q_1, q_2, \ldots, q_\ell)$, and $\mathcal{V}$'s random coins $(r_1, r_2, \ldots, r_\ell)$. The view of the verifier $\mathcal{V}$, denoted as $\mathsf{view}_{\mathcal{V}}(\mathcal{P}, \mathcal{V})[x]$, is a random variable $(m_1, r_1, q_1, m_2, r_2, q_2, \ldots, m_\ell, r_\ell, q_\ell)$ that is determined by the random coins of $\mathcal{V}$ and the messages of $\mathcal{P}$ after the interaction with the statement x. See Figure below.

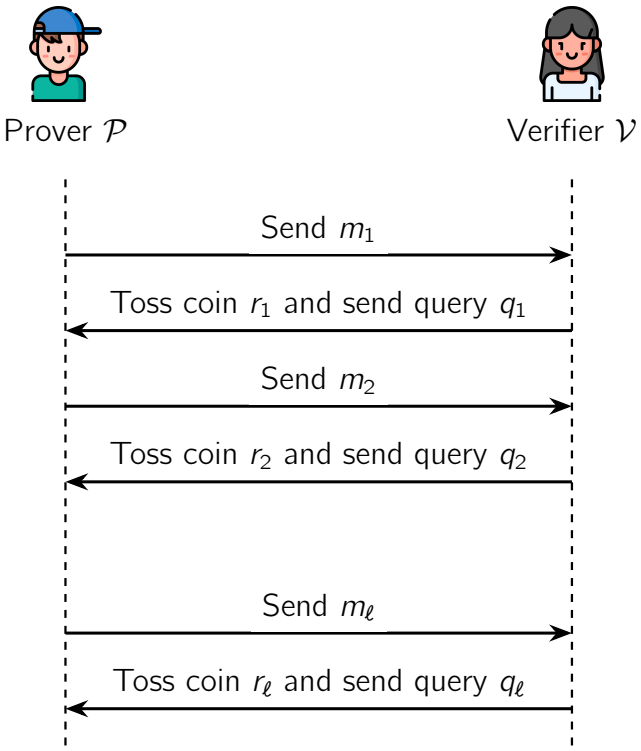


Figure 6.2: The interactive protocol between prover $\mathcal{P}$ and verifier $\mathcal{V}$. Prover's messages consist of messages $\{m_k\}_{k=1}^{\ell}$, verifier's messages consist of queries $\{q_k\}_{k=1}^{\ell}$, and additionally verifier samples random coins $\{r_k\}_{k=1}^{\ell}$.

Example. Suppose that for the aforementioned protocol with $N = 3 \times 2^{30} + 1$, the conversation between the prover $\mathcal{P}$, who wants to convince that $1286091780 \in \mathcal{L}_R$, and $\mathcal{V}$ is the following:

1. During the first round, $\mathcal{P}$ sends 672192003 to $\mathcal{V}$.
2. $\mathcal{V}$ sends $b = 0$ to $\mathcal{P}$.
3. $\mathcal{P}$ sends 2606437826 to $\mathcal{V}$.
4. $\mathcal{V}$ verifies that indeed $2606437826^2 \equiv 672192003 \pmod{N}$.
5. During the second round, $\mathcal{P}$ sends 2619047580 to $\mathcal{V}$.
6. $\mathcal{V}$ chooses $b = 1$ and sends to $\mathcal{P}$.
7. $\mathcal{P}$ sends 1768388249 to $\mathcal{V}$.
8. $\mathcal{V}$ verifies that indeed $1768388249^2 \equiv 2619047580 \times 1286091780 \pmod{N}$.
9. Conversation ends.

The view of the verifier $\mathcal{V}$ is the following:

$$\mathsf{view}_{\mathcal{V}}(\mathcal{V}, \mathcal{P})[1286091780] = (672192003, 0, 2606437826, 2619047580, 1, 1768388249)$$

Essentially, the view that you currently has witnessed is the same as one that $\mathcal{V}$ has seen. After this interaction, you have not learned anything about the witness w that prover $\mathcal{P}$ knows and which we, as of now, has not revealed to you.
In fact, you can verify by yourself, that the witness was $w = 3042517305$ and two randomnesses were $r_1 = 2606437826$ and $r_2 = 3023142760$.
One final note that is essential for the further discussion: variable $\mathsf{view}_{\mathcal{V}}(\mathcal{P}, \mathcal{V})[x]$ is a random variable. For example, for our particular case, both bits could be 0 or both bits could be 1.

6.4.2 The Simulation Paradigm

The key idea is the following: $\mathsf{view}_{\mathcal{V}}(\mathcal{V}, \mathcal{P})[x]$ gives nothing new to the verifier $\mathcal{V}$ about the witness w. But it gives nothing new, if he could have *simulated* this view on his own, without even running the interaction. In other words, the "simulated" and "real" views should be *computationally-indistinguishable*. But let us define the computational indistinguishability first.

Definition 6.9 (Computational Indistinguishability)**.** Given two random distributions D_0 and D_1, define the following challenger-adversary game:

1. The challenger randomly samples $x_0 \xleftarrow{R} D_0, x_1 \xleftarrow{R} D_1$ and a bit $b \xleftarrow{R} \{0,1\}$.
2. The challenger sends (x_0, x_1, b) to the adversary.
3. The adversary $\mathcal{A}$ outputs a bit $\hat{b}$.

We define the advantage of the adversary $\mathcal{A}$ in distinguishing D_0 and D_1 as

$$\text{Indadv}[\mathcal{A}, D_0, D_1] = \left|\Pr[b = \hat{b}] - \frac{1}{2}\right|$$

Distributions D_0 and D_1 are called **computationally indistinguishable**, denoted as $D_0 \approx D_1$, if for any polynomial-time adversary $\mathcal{A}$ and polynomial number of rounds in the game, the advantage $\text{Indadv}[\mathcal{A}, D_0, D_1]$ is negligible.

Finally, we are ready to define the **zero-knowledge**.

Definition 6.10 (Zero-Knowledge)**.** An interactive protocol $(\mathcal{P}, \mathcal{V})$ is **zero-knowledge** for a language $\mathcal{L}_\mathcal{R}$ if for every poly-time verifier $\mathcal{V}^*$ there exists a poly-time simulator Sim such that for any valid statement $x \in \mathcal{L}_\mathcal{R}$:

$$\text{view}_{\mathcal{V}^*}(\mathcal{P}, \mathcal{V}^*)[x] \approx \mathsf{Sim}(x)$$

However, the condition that verifier might be arbitrary is rather strong. Therefore, we introduce the notion of **honest-verifier zero-knowledge**.

Definition 6.11. Honest-Verifier Zero-Knowledge (HVZK) An interactive protocol $(\mathcal{P}, \mathcal{V})$ is **honest-verifier zero-knowledge** for a language $\mathcal{L}_\mathcal{R}$ if there exists a probabilistic poly-time simulator Sim such that for any valid statement $x \in \mathcal{L}_\mathcal{R}$[a]:

$$\text{view}_{\mathcal{V}}(\mathcal{P}, \mathcal{V})[x] \approx \mathsf{Sim}(x)$$

[a] Below, we assume that the verifier $\mathcal{V}$ is honest: he is following the protocol.

6.5 Proof of Knowledge

Now, the main issue with the above definition is that *we have proven the statement correctness, but we have not proven that the prover* **knows** *the witness*. These are completely two different things. Let us demonstrate why.

Example. Suppose that the prover $\mathcal{P}$ wants to convince the verifier that he knows the discrete logarithm of a given point $P \in E(\mathbb{F}_p)$ on a cyclic elliptic curve $E(\mathbb{F}_p)$ of order r. This corresponds to the relation and the corresponding language:

$$\mathcal{R} = \{(P, \alpha) \in E(\mathbb{F}_p) \times \mathbb{Z}_r : P = [\alpha]G\},$$
$$\mathcal{L}_{\mathcal{R}} = \{P \in E(\mathbb{F}_p) : \exists \alpha \in \mathbb{Z}_r \text{ such that } P = [\alpha]G\}$$

But here is the catch: actually, $\mathcal{L}_{\mathcal{R}} = E(\mathbb{F}_p)$ since any point P has a witness α such that $P = [\alpha]G$ (recall that the curve is cyclic)! So proving that $P \in \mathcal{L}_{\mathcal{R}}$ is completely useless! Rather, we want to prove that the prover knows α, not the fact that the point has a discrete logarithm.

That is why instead of **proof**, we need a **proof of knowledge**. This leads to even another weird paradigm used for the rigorous definition: the **extractor**. Basically, the knowledge of witness means that we can *extract* the witness while interacting with the prover. Yet, the *extractor* can do more than the verifier: he can call the prover however he wants and he can also rewind the prover (for example, run some pieces multiple times). This sometimes is referred to as "extractor $\mathcal{E}$ uses $\mathcal{P}$ as an oracle".

Now, let us move to the formal definition.

Definition 6.12 (Proof of Knowledge)**.** The interactive protocol $(\mathcal{P}, \mathcal{V})$ is a **proof of knowledge** for $\mathcal{L}_{\mathcal{R}}$ if exists a poly-time extractor algorithm $\mathcal{E}$ such that for any valid statement $x \in \mathcal{L}_{\mathcal{R}}$, in expected poly-time $\mathcal{E}^{\mathcal{P}}(x)$ outputs w such that $(x, w) \in \mathcal{R}$.

Lemma 6.13. The protocol from Section 6.3.1 is a proof of knowledge for the language $\mathcal{L}_{\mathcal{R}}$.

Proof. Let us define the extractor $\mathcal{E}$ for the statement x as follows:

1. Run the prover to receive $a \equiv r^2 \pmod N$ (r is chosen randomly from $\mathbb{Z}_N^*$).
2. Set verifier's message to $b = 0$ to get $z_1 \leftarrow r$.
3. **Rewind** and set verifier's message to $b = 1$ to get $z_2 \leftarrow rw \pmod N$.
4. Output $z_2/z_1 \pmod N$

The extractor $\mathcal{E}$ will always output w if $x \in \mathcal{L}_{\mathcal{R}}$. □

Remark. Note that extractor is given much more than the verifier: he can call the prover multiple times and he can also rewind the prover. This is the main difference between the verifier and the extractor.

6.6 Fiat-Shamir Heuristic

6.6.1 Random Oracle

In cryptography, one frequently encounters the term *cryptographic oracle*. In this section, we are not going to dive into the technical details of what that is, yet it is useful to have a general understanding of what it is.

Definition 6.14 (Cryptographic Oracle)**.** Informally, *cryptographic oracle* is simply a function $\mathcal{O}$ that gives in $O(1)$ an answer to some typically very hard problem.

Example. Consider the Computational Diffie-Hellman (CDH) problem on the cyclic elliptic curve $E(\mathbb{F}_p)$ of prime order r with a generator G. Recall that such problem consists in computing $[\alpha\beta]G$ given $[\alpha]G$ and $[\beta]G$ where $\alpha, \beta \in \mathbb{Z}_r$.
Typically, it is believed that the Diffie-Hellman problem is hard (meaning, for any adversary strategy the probability of solving the problem is negligible). However, we *could* assume that such problem can be solved in $O(1)$ by a cryptographic oracle $\mathcal{O}_{\text{CDH}} : ([\alpha]G, [\beta]G) \mapsto [\alpha\beta]G$. This way, we can rigorously prove the security of some cryptographic protocols *even* if the Diffie-Hellman problem is suddenly solved.

One of the most popular cryptographic oracles is the **random oracle** $\mathcal{O}_{\text{R}}$. Let us define how the random oracle works.

Suppose someone is inputting x to the random oracle $\mathcal{O}_{\text{R}}$. The oracle $\mathcal{O}_{\text{R}}$ does the following:

1. If x has been queried before, the oracle returns the same value as it returned before.
2. If x has not been queried before, the oracle returns a randomly uniformly sampled value from the output space.

Remark. Of course, the sudden appearance of the random oracle is not a magic trick. In practice, the random oracle is typically implemented as a hash function. Of course, formally, the hash function is not a random oracle, yet it is a very good approximation and it is reasonable to assume that the hash function behaves like a random oracle.

6.6.2 Fiat-Shamir Transformation

Now, the main issue with the interactive proofs is that they are... Well, *interactive*. Ideally, we simply want to accumulate a proof π, publish it (say, in blockchain) so that anyone (essentially, being the verifier) could check its validity. So we need some tools to make *some* interactive protocols non-interactive. This is, of course, not always possible, but there are some ways to achieve this.

While different protocols use different ways to achieve this, one of the most popular methods (which, in particular, is used in STARKs) is the **Fiat-Shamir heuristic**. The idea is the following: instead of verifier sending the challenges, we can replace them with the random oracle applied to all the previous messages.

Here how it goes. Suppose we have an interactive protocol $(\mathcal{P}, \mathcal{V})$ for the statement x. As previously defined, the interaction between $\mathcal{P}$ and $\mathcal{V}$ consists of prover's messages $(m_1, m_2, \ldots, m_\ell)$, verifier's queries $(q_1, q_2, \ldots, q_\ell)$, and verifier's random coins $(r_1, r_2, \ldots, r_\ell)$. In case all the queries are public random coins, such interactive protocol is called **public-coin protocol** (or, more formally, **Arthur-Merlin protocol**). However, as it turns out, when all the verifier's queries are simply uniformly sampled random values, it is an overkill to use the interactive protocol. Instead, suppose at some point the verifier got messages $m_1, m_2, \ldots, m_{\ell'}$ $(\ell' \leq \ell)$ from the prover. Then, instead of verifier sampling some random value $r_{\ell'}$, we can simply use the random oracle $\mathcal{O}_{\text{R}}$ as follows: $r_{\ell'} \leftarrow \mathcal{O}_{\text{R}}(x, m_1, m_2, \ldots, m_{\ell'})$. Practically, instead of random oracle $\mathcal{O}_{\text{R}}$ we use the hash function H, and use: $r_{\ell'} \leftarrow H(x \parallel m_1 \parallel m_2 \parallel \cdots \parallel m_{\ell'})$.

Remark. Sometimes, to simulate the "interaction" with the verifier, one uses the "Fiat-Shamir Channel". Its main purpose is to simulate the verifier's queries and random coins. For example, one might implement it as a class/struct with the following methods:

1. `send_message(`m`)`: "sends" the message m to the verifier. Under the hood, the proof stream accumulates the current state s and appends m to it.
2. `sample()`: returns the challenge r from the random oracle $\mathcal{O}_{\mathrm{R}}$, applied to the current state s.
3. `get_proof()`: returns the proof π, being the history of interaction, that the prover can publish.

One can check the `winterfell` Rust library or a simpler non-production implementation of the Fiat-Shamir Channel in Golang for more details.

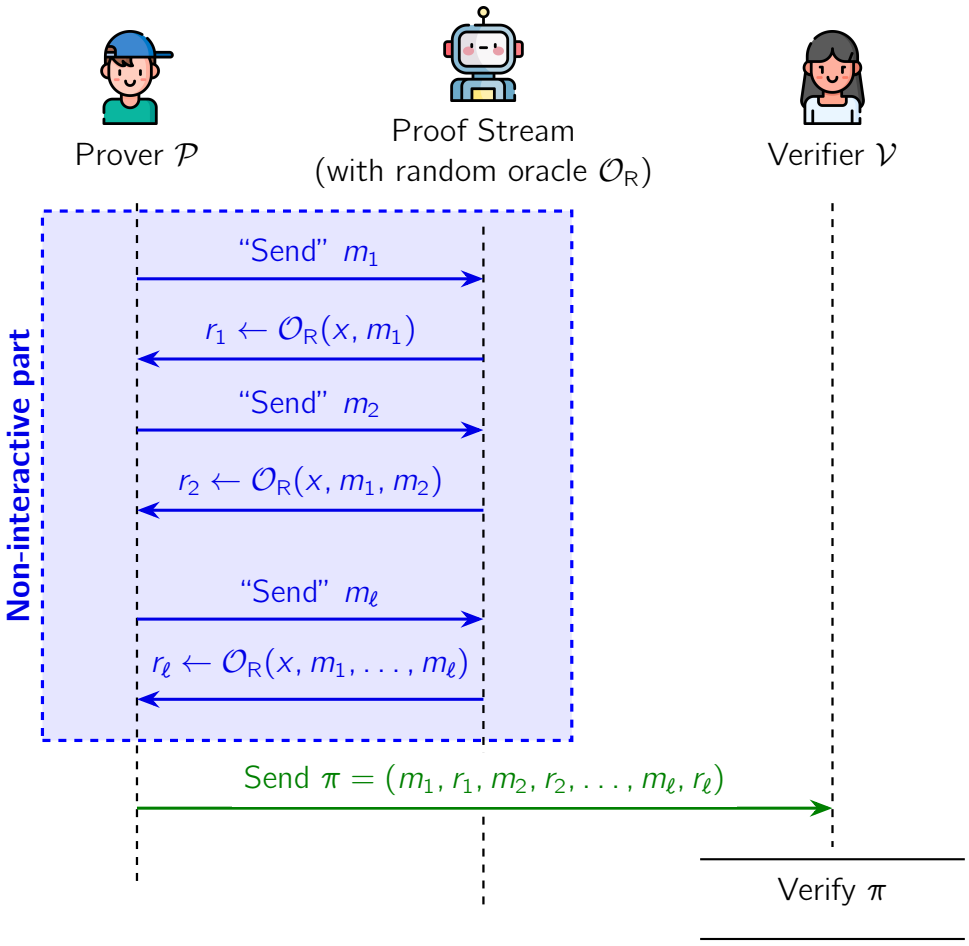


Figure 6.3: The non-interactive protocol between prover $\mathcal{P}$ and verifier $\mathcal{V}$ using Fiat-Shamir Transformation. In **blue** we marked a non-interactive part of the protocol, being the "communication" between a prover and a proof stream. In **green** we marked the final proof π that is sent to the verifier.

The process is illustrated in Figure 6.3. The Fiat-Shamir looks as follows:

1. First, the prover $\mathcal{P}$ "sends" the first message m_1 to the verifier $\mathcal{V}$. Here, "sending" is not an actual sending, but rather its simulation.

2. If we had an interactive protocol, the verifier $\mathcal{V}$ would send the random challenge r_1 to the prover $\mathcal{P}$. Instead, we use the random oracle $\mathcal{O}_{\mathsf{R}}$ to get $r_1 \leftarrow \mathcal{O}_{\mathsf{R}}(x, m_1)$.
3. Then, using this challenge, prover does his part in the protocol, and sends the next message m_2.
4. Again, if we had an interactive protocol, the verifier would send the next challenge r_2 to the prover. Instead, we use the random oracle $\mathcal{O}_{\mathsf{R}}$ to get $r_2 \leftarrow \mathcal{O}_{\mathsf{R}}(x, m_1, m_2)$, which gets "sent" to the prover.
5. The process continues until the protocol is finished.

Note that the whole process can be done by a prover with no interaction with the "verifier". In this case, one of the ways to represent the proof π is to publish the transcript of the interaction (that is, all the messages sent by the prover and challenges computed using the random oracle). This is exactly what is done in STARKs.

The reason why such transformation works is that the random oracle $\mathcal{O}_{\mathsf{R}}$ is a *random* function. Therefore, the challenges $r_1, r_2, \ldots$ are *random* values, and the prover cannot predict them (for example, by fabricating messages to have some specific output). That being said, the following theorem holds (which, of course, we are not going to prove since the proof is complicated).

Theorem 6.15. Suppose that $(\mathcal{P}, \mathcal{V})$ is a public-coin interactive argument of knowledge for some language $\mathcal{L}_{\mathcal{R}}$ with a negligible soundness error. Then, the Fiat-Shamir transformation of $(\mathcal{P}^{\mathcal{O}_{\mathsf{R}}}, \mathcal{V}^{\mathcal{O}_{\mathsf{R}}})$ is a non-interactive argument for $\mathcal{L}_{\mathcal{R}}$ with negligible soundness error in the random oracle model $\mathcal{O}_{\mathsf{R}}$.

6.7 Exercises

Exercise 1. When dealing with RSA protocol, one frequently encounters the following relation where e is a prime number and $n \in \mathbb{N}$:

$$\mathcal{R} = \left\{(w, x) \in \mathbb{Z}_n^\times \times \mathbb{Z}_n^\times : w^e = x\right\}$$

Which of the following is the language $\mathcal{L}_{\mathcal{R}}$ that corresponds to the relation $\mathcal{R}$?

(A) Integers from $\mathbb{Z}_n^\times$ which have a modular root of e-th degree.
(B) Integers from $\mathbb{Z}_n^\times$ which are divisible by e.
(C) Integers x from $\mathbb{Z}_n^\times$ with properly defined expression x^e.
(D) Integers from $\mathbb{Z}_n^\times$ which are prime.
(E) Integers from $\mathbb{Z}_n^\times$ for which e is a primitive root.

Exercise 2. Suppose that for some interactive protocol $(\mathcal{P}, \mathcal{V})$ during one round, the probability that the verifier $\mathcal{V}$ accepts a false statement is $1/8$. How many rounds of interaction are needed to guarantee 120 bits of security? Assume here that n bits of security means that the probability of accepting a false statement is at most 2^{-n}.

(A) 30.
(B) 40.
(C) 60.
(D) 90.
(E) 120.

Exercise 3. Recall that for relation $\mathcal{R} = \{(w, x) \in \mathbb{Z}_N^\times \times \mathbb{Z}_N^\times : x = w^2\}$ we defined the following interactive protocol $(\mathcal{P}, \mathcal{V})$ to prove that $x \in \mathcal{L}_\mathcal{R}$:

- $\mathcal{P}$ samples $r \xleftarrow{R} \mathbb{Z}_N^\times$ and sends $a = r^2$ to $\mathcal{V}$.
- $\mathcal{V}$ sends a random bit $b \in \{0, 1\}$ to $\mathcal{P}$.
- $\mathcal{P}$ sends $z = r \cdot w^b$ to $\mathcal{V}$.
- $\mathcal{V}$ accepts if $z^2 = a \cdot x^b$, otherwise it rejects.

Suppose we use the protocol $(\mathcal{P}, \mathcal{V}^*)$ where the "broken" verifier $\mathcal{V}^*$ always outputs $b = 1$. Which of the following statements is true?

(A) Both the soundness and completeness of the protocol are preserved.
(B) The soundness of the protocol is preserved, but the completeness is broken.
(C) The completeness of the protocol is preserved, but the soundness is broken.
(D) Both the soundness and completeness of the protocol are broken.

Exercise 4. What is the difference between the cryptographic proof and the proof of knowledge?

(A) Cryptographic proof is a proof of knowledge that is secure against malicious verifiers.
(B) Cryptographic proof is a proof of knowledge that is secure against malicious provers.
(C) Cryptographic proof merely states the correctness of a statement, while the proof of knowledge also guarantees that the prover knows the witness.
(D) While cryptographic proof states that witness exists for the given statement, the proof of knowledge makes sure to make this witness unknown to the verifier.
(E) Proof of knowledge does not require verifier to know the statement, while cryptographic proof does.

Exercise 5. What is the purpose of introducing the extractor?

(A) To introduce the algorithm that simulates the malicious verifier trying to extract the witness from the prover.
(B) To define what it means that the prover knows the witness.
(C) To give the verifier the ability to extract the witness from the prover during the interactive protocol.
(D) To define the security of the interactive protocol that uses a more powerful verifier that can extract additional information from the prover.
(E) To give prover more power to extract randomness generated by the verifier.

Exercise 6. What it means that the interactive protocol $(\mathcal{P}, \mathcal{V})$ is a zero-knowledge?

(A) The verifier $\mathcal{V}$ cannot know whether the given statement is true or false.
(B) The verifier $\mathcal{V}$ cannot know whether the prover $\mathcal{P}$ knows the witness.
(C) View of the prover $\mathcal{P}$ in the protocol is indistinguishable from the view of the verifier $\mathcal{V}$.
(D) Any view of any verifier $\mathcal{V}$ can be simulated using some polynomial-time algorithm, outputting computationally indistinguishable distribution from the given view.
(E) The prover $\mathcal{P}$ can convince the verifier $\mathcal{V}$ that the statement is true without knowing the witness.

Hint: View of the participant in the protocol consists of all data he has access to during the protocol execution. For example, verifier $\mathcal{V}$'s view consists of the messages he sends and

receives, as well as the random coins he generates.

Exercise 7. Which of the following is **not** true about the Fiat-Shamir heuristic?

(A) If the public-coin protocol is sound, the Fiat-Shamir transformation preserves the soundness.

(B) The Fiat-Shamir heuristic does not break the completeness of the public-coin protocol it is applied to.

(C) Practically, it allows to convert any interactive protocol into a non-interactive one.

(D) To make Fiat-Shamir transformation pratical, the function modelling the random oracle should be hard to invert.

(E) It is reasonable to use SHA256 to model the random oracle in the Fiat-Shamir transformation.

7 Sigma Protocols

7.1 Schnorr's Identification Protocol

To better illustrate the Fiat-Shamir Transformation in practice, let us consider one of the most basic Sigma protocols: non-interactive Schnorr Identification Protocol. It is a simple and elegant protocol that allows one party to prove to another party that it knows a discrete logarithm of a given element. It is also quite straightforward to generalize it to a signature scheme.

Let us formalize it using theory from Section 6.2. Suppose $\mathbb{G}$ is a cyclic group of order q with a generator g. Then, the relation and language being considered are:

$$\mathcal{R} = \{(u, \alpha) \in \mathbb{G} \times \mathbb{Z}_q : u = g^\alpha\},\ \mathcal{L}_\mathcal{R} = \{u \in \mathbb{G} : \exists \alpha \in \mathbb{Z}_q : u = g^\alpha\}$$

Now, suppose prover $\mathcal{P}$ has a valid statement and a witness $(u, \alpha) \in \mathcal{R}$ and he wants to convince the verifier $\mathcal{V}$ that he knows the witness α to the public statement u (that is, we are building the proof of knowledge). Well, the easiest way how to proceed is simply giving α to $\mathcal{V}$, but this is obviously not what we want. Instead, the Schnorr protocol allows $\mathcal{P}$ to prove the knowledge of α without revealing it.

First, let us start with the interactive version of the protocol.

Definition 7.1. The Schnorr interactive identification protocol $\Pi_{\mathsf{Sch}} = (\mathsf{Gen}, \mathcal{P}, \mathcal{V})$ with a generation function Gen and prover $\mathcal{P}$ and verifier $\mathcal{V}$ is defined as follows:

- $\mathsf{Gen}(1^\lambda)$: As with most public-key cryptosystems, we take $\alpha \xleftarrow{R} \mathbb{Z}_q$ and $u \leftarrow g^\alpha$. We output the *verification key* as $\mathsf{vk} := u$, and the *secret key* as $\mathsf{sk} := \alpha$.
- The protocol between $(\mathcal{P}, \mathcal{V})$ is run as follows:
  - $\mathcal{P}$ computes $r \leftarrow \mathbb{Z}_q^\times, a \leftarrow g^r$ and sends a to $\mathcal{V}$.
  - $\mathcal{V}$ sends a random challenge $e \xleftarrow{R} \mathbb{Z}_q$ to $\mathcal{P}$.
  - $\mathcal{P}$ computes $\sigma \leftarrow r + \alpha e \in \mathbb{Z}_q$ and sends σ to $\mathcal{V}$.
  - $\mathcal{V}$ accepts if $g^\sigma = a \cdot u^e$, otherwise it rejects.

This protocol is illustrated in Figure 7.1.

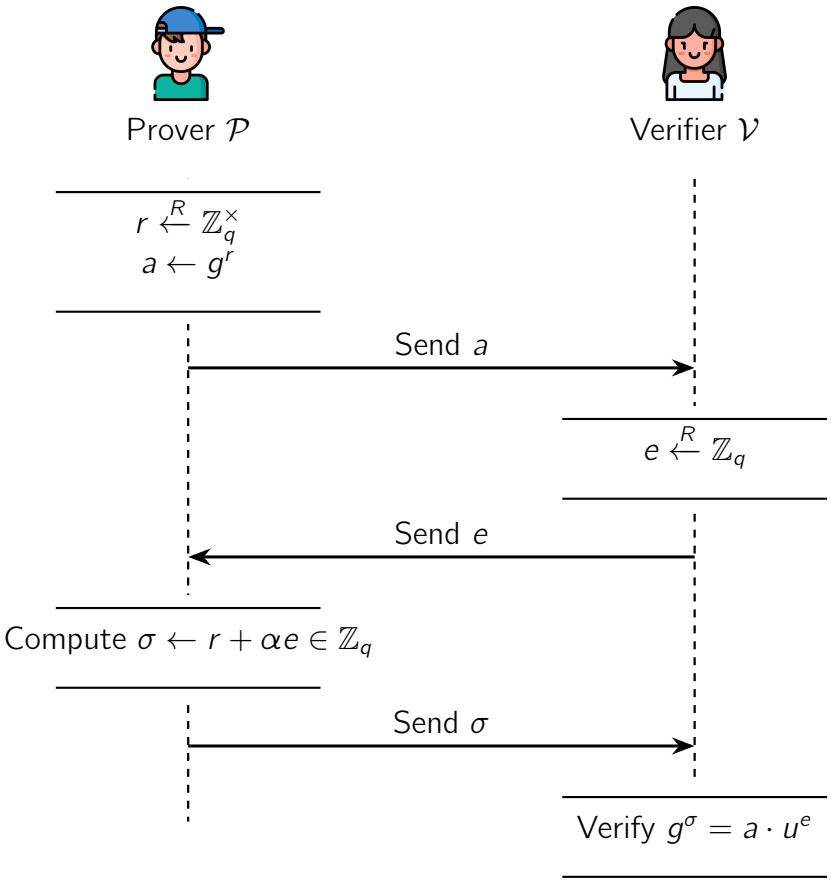


Figure 7.1: The interactive Schnorr protocol between prover $\mathcal{P}$ and verifier $\mathcal{V}$ for proof of knowledge of discrete logarithm relation.

Definition 7.2. An interaction between $\mathcal{P}$ and $\mathcal{V}$ produces the so-called **conversation** $(a, e, \sigma) \in \mathbb{G} \times \mathbb{Z}_q \times \mathbb{Z}_q$. We call such a conversation an **accepting conversation** if $\mathcal{V}$ accepts the proof.

Example. In case of a Schnorr protocol, the accepting conversation is such that $g^\sigma = a \cdot u^e$.

Now, one can prove the following theorem.

Theorem 7.3. The Schnorr protocol Π_{Sch} is complete, sound, and (honest verifier) zero-knowledge proof of knowledge.

Proof. We are not going to prove the zero-knowledge and soundness properly, but completeness and proof of knowledge are quite straightforward to show.

- **Completeness.** Just observe that $g^\sigma = g^{r+\alpha e} = g^r (g^\alpha)^e = a \cdot u^e$.
- **Proof of Knowledge.** To prove that the protocol is a proof of knowledge, we need to construct an extractor $\mathcal{E}^{\mathcal{P}}$. We construct it as follows:
  1. Extractor runs the prover and gets a, e, and σ as a response.
  2. Extractor rewinds back to the verifier's challenge step, generates a new challenge $e' \xleftarrow{R} \mathbb{Z}_q$ and gets new prover's response σ' (for the same prover's randomness r).

3. Extractor outputs the witness $\alpha \leftarrow (\sigma - \sigma')(e - e')^{-1}$.

The reason why this works is following: notice that $g^{\sigma} = a \cdot u^{e}, g^{\sigma'} = a \cdot u^{e'}$. Therefore, by dividing former by latter, we obtain $g^{\sigma-\sigma'} = u^{e-e'} = g^{\alpha(e-e')}$. It immediately follows that $\alpha = (\sigma - \sigma')(e - e')^{-1}$.

Remark. Before considering how to make such protocol non-interactive correctly, suppose that we instead do the following: after interaction with the verifier, the prover publishes the conversation as a proof of knowledge. Would that be a valid non-interactive proof? In other words, can we convince the independent observer of the interaction that the prover knows the witness? The answer is no (and it is generally so for any interactive protocol). The reason why is that the prover can first sample randomly $e, \sigma \xleftarrow{R} \mathbb{Z}_q$, compute $a \leftarrow g^{\sigma}/u^{e}$ and simply publish (a, e, σ) as a proof. This is a valid conversation since $g^{\sigma} = a \cdot u^{e} = (g^{\sigma}/u^{e}) \cdot u^{e}$ and thus the observer would be convinced that the prover knows the witness. However, the prover might not know the witness at all!
Therefore, either (1) the prover needs to get a challenge e **before** he commits to the value σ, or (2) challenge must be randomized. Otherwise, he can precompute σ and publish it as a proof (or simply make a deal with the verifier to fool the observer).

Now, notice that the provided protocol is a public-coin protocol. Therefore, we can apply the Fiat-Shamir transformation to make it non-interactive. Suppose we have a random oracle $\mathcal{O}_{\mathrm{R}} : \mathbb{G} \times \mathbb{G} \to \mathbb{Z}_q$:

1. The prover $\mathcal{P}$ computes $r \leftarrow \mathbb{Z}_q^{\times}$, $a \leftarrow g^{r}$ and sends a to the Fiat-Shamir Channel.
2. The Fiat-Shamir channel responds with the challenge $e \leftarrow \mathcal{O}_{\mathrm{R}}(u, a)$.
3. The prover $\mathcal{P}$ computes $\sigma \leftarrow r + \alpha e$ and sends σ to the Fiat-Shamir Channel.
4. The Fiat-Shamir channel outputs the proof $\pi = (a, e, \sigma)$, which the verifier can check via previously mentioned equation $g^{\sigma} = a \cdot u^{e}$.

Now, notice that e might not be included in the proof since the verifier can compute it by himself. Therefore, the final proof π can be reduced to $(a, \sigma) \in \mathbb{G} \times \mathbb{Z}_q$ and its computation does not need any interaction with the verifier. Moreover, it is still complete, sound, and proof of knowledge due to the Fiat-Shamir transformation. It is also (not easy to prove) zero-knowledge.

7.2 Schnorr's Signature Scheme

Now, turning the Schnorr's Identification Protocol into a signature scheme is quite straightforward. The only modification to the non-interactive proof described in the previous section is that we include the message $m \in \mathcal{M}$ instead of our statement $u \in \mathbb{G}$ in the computation of the challenge e. Additionally, suppose we use the hash function H as a random oracle from the previous section. Now, let us give a formal definition.

Definition 7.4. The Schnorr Signature Scheme is $\Sigma_{\text{Sch}} = (\text{Gen}, \text{Sign}, \text{Verify})$, where:

- $\text{Gen}(1^\lambda)$: We take $\alpha \xleftarrow{R} \mathbb{Z}_q$ and $u \leftarrow g^\alpha$. The *public key* is $\text{pk} := u$, while the *secret key* as $\text{sk} := \alpha$.
- $\text{Sign}(m, \text{sk})$: The signer computes $r \leftarrow \mathbb{Z}_q^\times, a \leftarrow g^r, e \leftarrow H(m, a), \sigma \leftarrow r + \alpha e$ and outputs the signature (a, σ).
- $\text{Verify}((a, \sigma), m, \text{pk})$: The verifier checks if $g^\sigma = a \cdot u^e$ for $e \leftarrow H(m, a)$.

Remark. Typically, one also uses a so-called "*key-prefixed*" variant of the scheme, where the challenge e is computed as $e \leftarrow H(\text{pk}, m, a)$ for a random oracle $H : \mathbb{G} \times \mathcal{M} \times \mathbb{G} \to \mathbb{Z}_q$. It was argued that such variant has a better multi-user security bound than the classical one.

7.3 Sigma Protocols

Now, the Schnorr Protocol is just one of the many examples of a so-called **Sigma Protocol**. Sigma protocols are a class of interactive proof systems that are used to prove the knowledge of a witness to a statement. They are quite general and can be used to prove the knowledge of a witness to any effective relation $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{W}$, where $\mathcal{X}$ is the set of public statements and $\mathcal{W}$ is the set of witnesses. Let us define them formally.

Definition 7.5. Let $\mathcal{R} \subset \mathcal{X} \times \mathcal{W}$ be an effective relation. A **Sigma protocol** for $\mathcal{R}$ is an interactive protocol $(\mathcal{P}, \mathcal{V})$ that satisfies the following properties:

- In the beginning, $\mathcal{P}$ computes a **commitment** a and sends it to $\mathcal{V}$.
- $\mathcal{V}$ chooses a random **challenge** $c \in \mathcal{C}$ from the challenge space $\mathcal{C}$ and sends it to $\mathcal{P}$.
- Upon receiving c, $\mathcal{P}$ computes the response z and sends it to $\mathcal{V}$.
- $\mathcal{V}$ outputs either accept or reject based on the conversation transcript (a, c, z).

Remark. The name "Sigma" protocol comes from the fact that the "shape" of the message flow vaguely resembles the Greek letter Σ: see Figure 7.2.

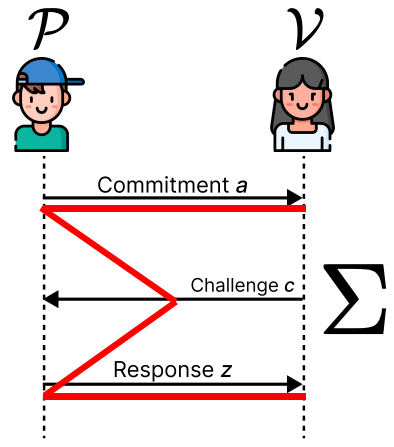


Figure 7.2: Sigma Protocol Illustration: the flow of messages between prover $\mathcal{P}$ and verifier $\mathcal{V}$ closely resembles the Greek letter Σ, which is marked in **red** in the Figure.

Example. In particular, for the Schnorr Protocol, the Sigma protocol is defined over the relation $\mathcal{R} \subset \mathcal{X} \times \mathcal{W}$ where:

$$\mathcal{X} = \mathbb{G},\ \mathcal{W} = \mathbb{Z}_q,\ \mathcal{R} = \{(u, \alpha) \in \mathbb{G} \times \mathbb{Z}_q : u = g^\alpha\}$$

Here, the challenge space $\mathcal{C}$ is a subset of $\mathbb{Z}_q$ (or, typically, the whole set).

Similarly to interactive protocols, Sigma protocols also have a property called *soundness*. However, there is an additional property called *special soundness* that simplifies the general notion of soundness.

Definition 7.6 (Special Soundness)**.** Let $(\mathcal{P}, \mathcal{V})$ be a Σ-protocol for $\mathcal{R} \subseteq \mathcal{X} \times \mathcal{Y}$. We that that $(\mathcal{P}, \mathcal{V})$ is **special sound** if there exists a witness extractor $\mathcal{E}$ such that, given statement $x \in \mathcal{X}$ and two accepting conversations (a, c, z) and (a, c', z') (where $c \neq c'$)[a], the extractor can always efficiently compute the witness w such that $(x, w) \in \mathcal{R}$.

[a] Notice that initial commitments in both conversations are the same!

Example. In case of the Schnorr Protocol, the special soundness property is satisfied by the extractor $\mathcal{E}$ that we have constructed in the proof of knowledge. In other words, we can extract the discrete logarithm $\alpha = \mathsf{DLog}_{\mathbb{G}}(u)$ given two accepting conversations (a, e, σ) and (a', e', σ').

Now, let us consider some more examples of Sigma protocols.

7.4 More Sigma Protocol Examples

7.4.1 Okamoto's Protocol for Representations

Again, let $\mathbb{G}$ be a cyclic group of prime order q with a generator $g \in \mathbb{G}$ and let $h \in \mathbb{G}$ an arbitrary group element (for example, it might be yet another group generator). While considering Pedersen Commitments, you already encountered form $g^\alpha h^\beta$. Now, let us generalize this concept a bit.

Definition 7.7. For $u \in \mathbb{G}$, a **representation** relative to g and h is a pair $(\alpha, \beta) \in \mathbb{Z}_q \times \mathbb{Z}_q$ such that $u = g^\alpha h^\beta$.

Remark. Notice that for the given u there are exactly q representations relative to g and h. Indeed, $\forall \beta \in \mathbb{Z}_q\, \exists! \alpha \in \mathbb{Z}_q : g^\alpha = u h^{-\beta}$.

Now, the *Okamoto's Protocol* is a Sigma protocol that allows one party to prove the knowledge of a representation of a given $u \in \mathbb{G}$ relative to g and h. In other words, we are working with the relation

$$\mathcal{R} = \left\{(u, (\alpha, \beta)) \in \mathbb{G} \times \mathbb{Z}_q^2 : u = g^\alpha h^\beta\right\}$$

Now, let us describe the protocol.

Definition 7.8 (Okamoto's Identification Protocol)**. Okamoto's Protocol** consists of two algorithms: $(\mathcal{P}, \mathcal{V})$, where the prover is assumed to know $(u, (\alpha, \beta)) \in \mathcal{R}$ defined above. The protocol is defined as follows:

1. $\mathcal{P}$ computes $\alpha_r \xleftarrow{R} \mathbb{Z}_q$, $\beta_r \xleftarrow{R} \mathbb{Z}_q$, $u_r \leftarrow g^{\alpha_r} h^{\beta_r}$ and sends commitment u_r to $\mathcal{V}$.
2. $\mathcal{V}$ samples the challenge $c \xleftarrow{R} \mathbb{Z}_q$ and sends c to $\mathcal{P}$.
3. $\mathcal{P}$ computes $\alpha_z \leftarrow \alpha_r + \alpha c, \beta_z \leftarrow \beta_r + \beta c$ and sends $\mathbf{z} = (\alpha_z, \beta_z)$ to $\mathcal{V}$.
4. $\mathcal{V}$ checks whether $g^{\alpha_z} h^{\beta_z} = u_r u^c$ and accepts or rejects the proof accordingly.

Theorem 7.9. Okamoto's Protocol is a Σ-protocol for the relation $\mathcal{R}$ which is Honest-Verifier Zero-Knowledge.

Part of the proof. Again, let us show *correctness* and *special soundness* without honest-verifier zero-knowledge properties.

Completeness. Suppose indeed that $(u, (\alpha, \beta)) \in \mathcal{R}$. Then, the verification condition can be written as follows:

$$g^{\alpha_z} h^{\beta_z} = g^{\alpha_r + \alpha c} h^{\beta_r + \beta c} = g^{\alpha_r} g^{\alpha c} h^{\beta_r} h^{\beta c} = \underbrace{(g^{\alpha_r} h^{\beta_r})}_{=u_r} \cdot \underbrace{(g^{\alpha} h^{\beta})}_{=u}{}^c = u_r u^c$$

Special Soundness. Suppose we are given two accepting conversations: $(u_r, c, (\alpha_z, \beta_z))$ and $(u_r, c', (\alpha'_z, \beta'_z))$ and we want to construct an extractor $\mathcal{E}$ which would give us a witness (α, β). In this case, we have the following holding:

$$g^{\alpha_z} h^{\beta_z} = u_r u^c, \; g^{\alpha'_z} h^{\beta'_z} = u_r u^{c'}$$

We can divide the former by the latter to obtain:

$$g^{\alpha_z - \alpha'_z} h^{\beta_z - \beta'_z} = u^{c - c'} = g^{\alpha(c - c')} h^{\beta(c - c')},$$

from which the extractor $\mathcal{E}$ can efficiently compute witness as follows: $\alpha \leftarrow (\alpha_z - \alpha'_z)/(c - c')$ and $\beta \leftarrow (\beta_z - \beta'_z)/(c - c')$.

7.4.2 Chaum-Pedersen protocol for DH-triplets

As with previous examples, suppose we are given the cyclic group $\mathbb{G}$ or prime order q and generator $g \in \mathbb{G}$. Recall that *the Diffie-Hellman Triple* (or, *DH-triple*) is a triple $(g^\alpha, g^\beta, g^\gamma)$ with $\gamma = \alpha\beta$. Now, this definition is not really convenient for us, so we will reformulate the DH-triple using the proposition below.

Proposition 7.10 (Alternative DH-triple Definition)**.** (u, v, w) is a DH-triplet iff $\exists \beta \in \mathbb{Z}_q$: $v = g^\beta, w = u^\beta$.

Now, this makes it easier to define the relation $\mathcal{R}$ for the Chaum-Pedersen protocol:

$$\mathcal{R} = \left\{((u, v, w), \beta) \in \mathbb{G}^3 \times \mathbb{Z}_q : v = g^\beta \wedge w = u^\beta\right\}$$

In other words, here we have a witness $\beta \in \mathbb{Z}_q$, while the statement is a triplet $(u, v, w) \in \mathbb{G}^3$. Again, we want to convert this into a Sigma protocol. We do it as follows.

Definition 7.11 (Chaum-Pedersen Protocol)**. Chaum-Pedersen Protocol** consists of two algorithms: $(\mathcal{P}, \mathcal{V})$, where the prover is assumed to know $(\beta, (u, v, w)) \in \mathcal{R}$ defined above. The protocol is defined as follows:

1. $\mathcal{P}$ computes $\beta_r \xleftarrow{R} \mathbb{Z}_q$, $v_r \xleftarrow{R} g^{\beta_r}$, $w_r \leftarrow u^{\beta_r}$ and sends commitment (u_r, w_r) to $\mathcal{V}$.
2. $\mathcal{V}$ samples the challenge $c \xleftarrow{R} \mathbb{Z}_q$ and sends c to $\mathcal{P}$.
3. $\mathcal{P}$ computes $\beta_z \leftarrow \beta_r + \beta c$ and sends β_z to $\mathcal{V}$.
4. $\mathcal{V}$ checks whether two conditions hold: $g^{\beta_z} = v_r v^c$ and $u^{\beta_z} = w_r w^c$, and accepts or rejects the proof accordingly.

Theorem 7.12. Chaum-Pedersen Protocol is a Σ-protocol for the relation $\mathcal{R}$ which is Honest-Verifier Zero-Knowledge.

Part of the proof. As always, let us show *correctness* and *special soundness* without honest-verifier zero-knowledge properties.

Correctness. Again, consider the expression g^{β_z} more closely:

$$g^{\beta_z} = g^{\beta_r + \beta c} = g^{\beta_r} g^{\beta c} = \underbrace{g^{\beta_r}}_{=v_r} (\underbrace{g^{\beta}}_{=v})^c = v_r v^c$$

The similar reasoning can be applied to the second verification condition: indeed, here we have $u^{\beta_z} = u^{\beta_r}(u^{\beta})^c = w_r w^c$

Special Soundness. Suppose we are given two accepting conversations: $((u_r, w_r), c, \beta_z)$ and $((u_r, w_r), c', \beta'_z)$ and we want to construct an extractor $\mathcal{E}$ which would give us a witness β. Notice that the following equations hold:

$$g^{\beta_z} = v_r v^c,\ g^{\beta'_z} = v_r v^{c'},$$
$$u^{\beta_z} = w_r w^c,\ u^{\beta'_z} = w_r w^{c'}.$$

Divide left equations by the right ones to obtain:

$$g^{\beta_z - \beta'_z} = v^{c-c'},\ u^{\beta_z - \beta'_z} = w^{c-c'}.$$

Consider the first equation. Since $v = g^{\beta}$ we derive $(\beta_z - \beta'_z) = \beta(c - c')$, from which $\mathcal{E}$ outputs $\beta = \frac{\beta_z - \beta'_z}{c - c'}$. The same value can be extracted from the second equation.

7.5 Generalizing Sigma Protocols

Now, the most interesting part! Probably, you have noticed, that all protocols above (Schnorr, Okamoto, Chaum-Pedersen) have a similar structure. So is there any way to generalize them? The answer is yes and moreover, this done in a very elegant way.

Let $(\mathbb{H}, \oplus)$ and $(\mathbb{T}, \otimes)$ be two finite abelian groups and suppose we have some concrete homomorphism $\psi : \mathbb{H} \to \mathbb{T}$. Moreover, we require that given $t \in \mathbb{T}$, finding the pre-image of t (meaning, finding some $h \in \mathbb{H}$ such that $\psi(h) = t$) is computationally hard. Suppose $\mathcal{F}$ is a set of all homomorphisms from $\mathbb{H}$ to $\mathbb{T}$ (sometimes denoted as $\mathsf{Hom}(\mathbb{H}, \mathbb{T})$). Now, define the following relation:

$$\mathcal{R} = \{((t, \psi), h) \in (\mathbb{T} \times \mathcal{F}) \times \mathbb{H} : \psi(h) = t\}$$

And now the prover $\mathcal{P}$ wants to convince the verifier $\mathcal{V}$ that he knows the witness h to the statement (t, ψ).

Proposition 7.13. Now, why does this generalize the previous protocols? Well, let us consider all previous examples:

- **Schnorr Protocol:** Here we have $\mathbb{H} = \mathbb{Z}_q$, $\mathbb{T} = \mathbb{G}$, and $\psi : \mathbb{Z}_q \to \mathbb{G}$ is defined as $\psi(\alpha) = g^\alpha$. Moreover, here ψ is an isomorphism!
- **Okamoto Protocol:** Here we have $\mathbb{H} = \mathbb{Z}_q^2$, $\mathbb{T} = \mathbb{G}$, and $\psi : \mathbb{Z}_q^2 \to \mathbb{G}$ is defined as $\psi(\alpha, \beta) = g^\alpha h^\beta$. It is also quite easy to see that ψ is a homomorphism:

  $$\psi((\alpha,\beta)+(\alpha',\beta')) = \psi(\alpha+\alpha', \beta+\beta') = g^{\alpha+\alpha'} h^{\beta+\beta'} = g^\alpha h^\beta g^{\alpha'} h^{\beta'} = \psi(\alpha,\beta)\psi(\alpha',\beta')$$

- **Chaum-Pedersen Protocol:** Here we have $\mathbb{H} = \mathbb{Z}_q$, $\mathbb{T} = \mathbb{G}^2$, and $\psi : \mathbb{Z}_q \to \mathbb{G}^2$ is defined as $\psi(\beta) = (g^\beta, u^\beta)$. Again, it is easy to see that ψ is a homomorphism.

Now, we formulate the general Sigma protocol for the relation $\mathcal{R}$ over homomorphism.

Definition 7.14 (Sigma Protocol for the pre-image of a homomorphism)**.** The protocol consists of two algorithms: $(\mathcal{P}, \mathcal{V})$, where the prover is assumed to know the witness $h \in \mathbb{H}$ defined above. The protocol is defined as follows:

1. $\mathcal{P}$ computes $h_r \xleftarrow{R} \mathbb{H}$, $t_r \leftarrow \psi(h_r) \in \mathbb{T}$ and sends t_r to the verifier $\mathcal{V}$.
2. $\mathcal{V}$ samples the challenge $c \xleftarrow{R} \mathcal{C} \subset \mathbb{Z}$ from the challenge space and sends c to $\mathcal{P}$.
3. $\mathcal{P}$ computes $h_z \leftarrow h_r \oplus h \cdot c$ and sends h_z to $\mathcal{V}$.
4. $\mathcal{V}$ checks whether $\psi(h_z) = t_r \otimes t^c$, and accepts or rejects the proof accordingly.

7.6 Combining Sigma Protocols

Now, suppose we have the Sigma interactive protocol $(\mathcal{P}_0, \mathcal{V}_0)$ for one relation $\mathcal{R}_0 \subseteq \mathcal{W}_0 \times \mathcal{X}_0$ and another Sigma interactive protocol $(\mathcal{P}_1, \mathcal{V}_1)$ for another relation $\mathcal{R}_1 \subseteq \mathcal{W}_1 \times \mathcal{X}_1$. Now, we want to combine these two protocols into a single one. Namely, we want our prover to be able to convince the verifier that:

1. He knows the witnesses w_0, w_1 to both statements x_0, x_1.
2. He knows the witness $w \in \mathcal{W}_0 \cup \mathcal{W}_1$ to either statement x_0 or x_1.

Among two, the second one is a bit more interesting since it allows us to prove the knowledge of a witness to either of the statements. This is called the *OR-composition* of Sigma protocols.

7.6.1 The AND Sigma Protocol

Now, let the prover $\mathcal{P}$ prove the witness knowledge of the following relation:

$$\mathcal{R}_{\text{AND}} = \{((x_0, x_1), (w_0, w_1)) \in (\mathcal{X}_0 \times \mathcal{X}_1) \times (\mathcal{W}_0 \times \mathcal{W}_1) : (w_0, x_0) \in \mathcal{R}_0 \wedge (w_1, x_1) \in \mathcal{R}_1\}$$

We define the following protocol.

Definition 7.15 (The AND Sigma Protocol)**.** Define a pair of algorithms $(\mathcal{P}, \mathcal{V})$ which are run as follows:

1. The prover $\mathcal{P}$ runs $\mathcal{P}_0(w_0, x_0)$ to get commitment a_0 and runs $\mathcal{P}_1(w_1, x_1)$ to get a_1 and sends the pair $\mathbf{a} = (a_0, a_1)$ to $\mathcal{V}$.
2. The verifier computes the challenge $c \xleftarrow{R} \mathcal{C}$ and sends it to $\mathcal{P}$.
3. The prover feeds provers $\mathcal{P}_0(w_0, x_0)$ and $\mathcal{P}_1(w_1, x_1)$ with the challenge to get responses z_0 and z_1, respectively. He then sends $\mathbf{z} = (z_0, z_1)$ to $\mathcal{V}$.
4. The verifier checks whether both $\mathcal{V}_0(a_0, c, z_0)$ and $\mathcal{V}_1(a_1, c, z_1)$ pass.

However, such protocol is not very interesting since what we did essentially is just running two protocols separately: one for $(\mathcal{P}_0, \mathcal{V}_0)$, and the other for $(\mathcal{P}_1, \mathcal{V}_1)$. The only difference is that we use the single challenge for both protocols.

7.6.2 The OR Sigma Protocol

The less trivial example is the following: define the relation

$$\mathcal{R}_{\text{OR}} = \{((x_0, x_1), (w, b)) \in (\mathcal{X}_0 \times \mathcal{X}_1) \times ((\mathcal{W}_0 \cup \mathcal{W}_1) \times \{0, 1\}) : (x, w_b) \in \mathcal{R}_b\}$$

Here, the statement is x_0 and x_1, but the witness is the witness w to either x_0 or x_1, and the bit $b \in \{0, 1\}$, marking to which of the statement w belongs to. That being said, w might be from either set $\mathcal{W}_0$ or $\mathcal{W}_1$: that is why we say that $w \in \mathcal{W}_0 \cup \mathcal{W}_1$.

To make the interactive protocol work, we add one more assumption about both relations $\mathcal{R}_0$ and $\mathcal{R}_1$. Suppose that the challenge space $\mathcal{C} \subseteq \{0, 1\}^\ell$. This assumption is not very strong as typically $\mathcal{C}$ is some subspace of integers and thus decomposing some $c \in \mathcal{C}$ into the fixed-length bit representation is a trivial task.

Now, we describe the algorithm.

Definition 7.16 (The OR Sigma Protocol)**.** Define a pair of algorithms $(\mathcal{P}, \mathcal{V})$ for relation $\mathcal{R}_{\text{OR}}$ with $b^* := 1 - b$ as follows:

1. The prover chooses a random challenge $c_{b^*} \xleftarrow{R} \mathcal{C}$ and generates random commitment and response (a_{b^*}, z_{b^*}) that form a valid accepting conversation $(a_{b^*}, c_{b^*}, z_{b^*})$ (essentially, the prover runs the simulator $(a_{b^*}, z_{b^*}) \xleftarrow{R} \text{Sim}_{b^*}(x_{b^*}, c_{b^*})$). Then, $\mathcal{P}$ also runs $\mathcal{P}_b(x_b, w)$ to get a valid commitment a_b and sends (a_0, a_1) to $\mathcal{V}$.
2. The verifier sends a random challenge $c \xleftarrow{R} \mathcal{C} \subseteq \{0, 1\}^\ell$.
3. The prover XORs both challenges: $c_b \leftarrow c \oplus c_{b^*}$. Then it feeds the challenge c_b to the prover $\mathcal{P}_b(x_b, w)$ to get the responses z_b ($b \in \{0, 1\}$) and sends (c_0, z_0, z_1) to $\mathcal{V}$.
4. Verifier computes $c_1 \leftarrow c \oplus c_0$ and checks that both verifications $\mathcal{V}_0(a_0, c_0, z_0)$ and $\mathcal{V}_1(a_1, c_1, z_1)$ pass.

7.7 Exercises

Exercises 1-5. In search of correct Schnorr's Identification Protocol...

You are given the protocol and five ways to implement it. Most of them lack the crucial properties. For each attempt, you need to determine whether the protocol is correct and, if not, specify which of the properties are violated.

Recall, that given the cyclic group $\mathbb{G}$ of order q, the prover wants to convince the verifier that he knows the discrete logarithm α of $h \in \mathbb{G}$ with respect to the generator $g \in \mathbb{G}$ (so that $g^\alpha = h$).
Here are five attempts to construct the protocol:

Attempt 1. Prover sends witness α to the verifier. Verifier checks whether $h = g^\alpha$.

Attempt 2. Prover chooses random $r \xleftarrow{R} \mathbb{Z}_q$ and sends $a \leftarrow \alpha + r$ to the verifier. Verifier checks whether $h = g^a$.

Attempt 3. Prover chooses random $r \xleftarrow{R} \mathbb{Z}_q$, calculates $a \leftarrow \alpha + r$ and sends both (a, r) to the verifier. Verifier checks whether $g^r h = g^a$.

Attempt 4. Prover chooses random $r \xleftarrow{R} \mathbb{Z}_q$, calculates $a \leftarrow g^r, z \leftarrow \alpha + r$ and sends (a, z) to the verifier. Verifier checks whether $a \cdot h = g^z$.

Attempt 5. Prover chooses random $r \xleftarrow{R} \mathbb{Z}_q$, calculates $a \leftarrow g^r$, and sends a to the verifier. Verifier chooses $e \xleftarrow{R} \mathbb{Z}_q$ and sends to the prover. Prover calculates $z \leftarrow \alpha e + r$ and sends to the prover. Verifier checks whether $a \cdot h^e = g^z$.

Below, mark whether the properties of *completeness*, *soundness*, and *zero-knowledge* hold for each attempt.

Attempt #	**1**	**2**	**3**	**4**	**5**
Completeness holds?	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗
Soundness holds?	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗
Zero-Knowledge holds?	✓/✗	✓/✗	✓/✗	✓/✗	✓/✗

Exercises 6-10. Non-Interactive Chaum-Pedersen Protocol.

This section explores how to make the previously considered Chaum-Pedersen protocol non-interactive. Fill in the gaps in the following text with the correct statements.

Recall that the Chaum-Pedersen protocol allows the prover $\mathcal{P}$ to convince the skeptical verifier $\mathcal{V}$ that the given triplet $(u, v, w) \in \mathbb{G}^3$ is a Diffie-Hellman (DH) triplet in the cyclic group $\mathbb{G}$ of prime order q with a generator $g \in \mathbb{G}$, meaning that $u = g^\alpha, v = g^\beta, w = g^{\alpha\beta}$ for some $\alpha, \beta \in \mathbb{Z}_q$. However, instead of making (α, β) as a witness, observe that β is sufficient. Indeed, if $u = g^\alpha, v = g^\beta$, then $w =$ **6**. Thus, the relation is:

$$\mathcal{R} = \left\{((u, v, w), \beta) \in \mathbb{G}^3 \times \mathbb{Z}_q : \boxed{7}\right\}$$

Now, we apply the *Fiat-Shamir Transformation*. Recall that prover, instead of getting the random challenge $c \xleftarrow{R} \mathcal{C} \subset \mathbb{Z}_q$ from the verifier interactively, calculates it as the hash function from the public statement (u, v, w) and the prover's commitment. For that reason, define the non-interactive proof system $\Phi = (\mathsf{Gen}, \mathsf{Verify})$ as follows:

- Gen: On input $(u, v, w) \in \mathbb{G}^3$,
  1. Sample $\beta_r \xleftarrow{R} \mathbb{Z}_q$ and compute the commitment **8**.
  2. Use the hash function **9** to get the challenge $c \leftarrow$ **10**.
  3. Compute response $\beta_z \leftarrow \beta_r + \beta c$ and output commitment (v_r, w_r) and β_z as a proof π.
- Verify: Upon receiving statement (u, v, w) and a proof $\pi = (v_r, w_r, \beta_z)$, the verifier:
  1. Recomputes the challenge c using the hash function.
  2. Accepts if and only if $g^{\beta_z} = v_r v^c$ and $u^{\beta_z} = w_r w^c$.

Exercise 6.

A) v^β
B) u^β
C) vu
D) v^u
E) $v^\beta u$

Exercise 7.

A) $v = g^\beta$ and $w = vu$
B) $v = g^\beta$ and $w = v^\beta$
C) $v = g^\beta$ and $w = u^\beta$
D) $u = g^\beta$ and $w = u^\beta$
E) $u/w = g^\beta$

Exercise 8.

A) $(v_r, w_r) = (g^{\beta_r}, g^{\beta_r\beta})$
B) $(v_r, w_r) = (g^{\beta_r}, w^{\beta_r})$
C) $(v_r, w_r) = (g^{\beta_r}, u^{\beta_r})$
D) $(v_r, w_r) = (g^\beta, g^{\beta_r})$
E) $(v_r, w_r) = (g^\beta, g^{\beta_r} g^\beta)$

Exercise 9.

A) $H : \mathbb{G}^3 \times \mathbb{G}^2 \to \mathcal{C}$
B) $H : \mathbb{G}^3 \times (\mathbb{G} \times \mathbb{Z}_q) \to \mathcal{C}$
C) $H : \mathbb{G}^3 \to \mathcal{C}$
D) $H : \mathbb{G}^3 \times \mathbb{Z}_q \to \mathcal{C}$
E) $H : \mathbb{G}^2 \times \mathbb{Z}_q \to \mathcal{C}$

Exercise 10.

A) $H((u, v, w), (v_r, w_r))$
B) $H((u, v, w), (v_r, \beta_r))$
C) $H(u, v, w)$
D) $H((u, v, w), \beta_r)$
E) $H((v_r, w_r), \beta_r)$

8 Introduction to SNARKs. Arithmetic Circuits. R1CS

8.1 What is zk-SNARK?

8.1.1 Informal Overview

Finally, we've reached the most interesting part of the course, where we will consider various zk-SNARK constructions we are using on the daily basis. Again, recall that we have the presence of two parties:

- **Prover** $\mathcal{P}$ — the party who knows the data that can resolve the given problem.
- **Verifier** $\mathcal{V}$ — the party that wants to verify the given proof.

Here, the prover wants to convince the verifier that they know the data that resolves the problem (typically, some complex computation) without revealing the data (witness) itself. In the previous lecture, we defined the first practical primitive: zk-NARK — a *zero-knowledge non-interactive argument of knowledge*, and gave the first widely used example: non-interactive Schnorr protocol (which is a special case of a Σ-protocol with the Fiat-Shamir transformation applied). Now, we add one more component which completely changes the game and significantly extends the number of applications: **succinctness**.

Definition 8.1. zk-SNARK — Zero-Knowledge **Succinct** Non-interactive ARgument of Knowledge.

Again, since this is a central question considered, we need to recall what do terms like "argument of knowledge", "succinct", "non-interactive", and "zero-knowledge" mean in this context:

- **Argument of Knowledge** — a proof that the prover knows the data (witness) that resolves a certain problem, and this knowledge can be "extracted".
- **Succinctness** — the proof size and verification time is relatively small relative to the computation size and sometimes even does not depend on the size of the data or statement. This will be explained with examples later.
- **Non-interactiveness** — to produce the proof, the prover does not need any interaction with the verifier.
- **Zero-Knowledge** — the verifier learns nothing about the data used to produce the proof, despite knowing that this data resolves the given problem and that the prover possesses it.

In essence, zk-SNARKs allow one party to prove to another that they know a value without revealing any information about the value itself, and do so with a proof that is both very small and quick to verify. This makes zk-SNARKs a powerful tool for maintaining privacy and efficiency in various cryptographic applications.

This is pretty wide defined and maybe not so obvious if you do not have any background. Let us take a look at the example.

Example. Imagine you are the part of a treasure hunt, and you've found a hidden treasure chest. You want to prove to the treasure hunt organizer that you know where the chest is hidden without revealing its location. Here's how zk-SNARKs can be used in this context:

The problem: you have found a hidden treasure chest (the secret data), and you want to prove to the organizer (the verifier) that you know its location without actually revealing where it is.

How zk-SNARKs Help:

- **Argument of Knowledge**: You create a proof that demonstrates you know the exact location of the treasure chest. This proof convinces the organizer that you have this knowledge.
- **Succinctness**: The proof you provide is very small and concise. It doesn't matter how large the treasure map is or how many steps it took you to find the chest, the proof remains compact and easy to check.
- **Non-interactiveness**: You don't need to have a back-and-forth conversation with the organizer to create this proof. You prepare it once. The organizer can verify it without needing to ask you any questions.
- **Zero-Knowledge**: The proof doesn't reveal any information about the actual location of the treasure chest. The organizer knows you found it, but they don't learn anything about where it is hidden.

Here you can think of zk-SNARK as a golden coin from the chest where the pirates' sign is engraved, so the organizer can be sure you've found the treasure.

But the problems that we want to solve are in a slightly different format. We can't bring a coin to the verifier. Our goal is to prove that we've executed a specific program on a set of data that resolves a specific challenge or gives us a particular result.

8.1.2 Formal Definition

In this section, we will provide a more formal definition of zk-SNARKs. In case you do not want to dive into the technical details, you can skip this part and move to the next sections where we will consider the arithmetic circuits and the Quadratic Arithmetic Programs.

Previously, we considered NARKs that did not require any setup procedure. However, zk-SNARKs are more complex and require a setup phase. This setup phase is used to generate the proving and verification keys (which we call prover parameters pp and verifier parameters vp, respectively), which are then used to create and verify proofs. That being said, let us introduce the **preprocessing NARK**.

Definition 8.2. A **preprocessing non-interactive argument of knowledge** (**preprocessing NARK**) $\Pi_{\mathsf{preNARK}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ consists of three algorithms:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pp}, \mathsf{vp})$ — the setup algorithm that takes the security parameter λ and outputs the public parameters: proving and verification keys.
- $\mathsf{Prove}(\mathsf{pp}, x, w) \to \pi$ — the proving algorithm that takes the prover parameters pp, statement x, and witness w, and outputs a proof π.
- $\mathsf{Verify}(\mathsf{vp}, x, \pi) \to \{\mathsf{accept}, \mathsf{reject}\}$ — the verification algorithm that takes the verification key, statement x, and proof π, and outputs a bit indicating whether the proof is valid.

Recall, that from NARK (and now preprocessing NARK, respectively) over relation $\mathcal{R}$ we require the following properties:

- **Completeness** — if the prover is honest and the statement is true, the verifier will always accept the proof:

$$\forall (x, w) \in \mathcal{R} : \Pr[\mathsf{Verify}(\mathsf{vp}, x, \mathsf{Prove}(\mathsf{pp}, x, w)) = \mathsf{accept}] = 1$$

- **Knowledge Soundness** — the prover cannot (statistically) generate a false proof π that convinces the verifier.
- **Zero-knowledge** — the verifier "learns nothing" about the witness w from $(\mathcal{R}, \mathsf{pp}, \mathsf{vp}, x, \pi)$.

While we have formally defined all the terms here, including statistical soundness, we have not defined what **knowledge soundness** is. We give a brief informal definition below.

Definition 8.3 (Knowledge Soundness)**.** Π_{preNARK} is (adaptively) **knowledge sound** for a relation $\mathcal{R}$ if for every PPT adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$, split into two algorithms, such that:

$$\Pr\left[\mathsf{Verify}(\mathsf{vp}, x, \pi) = \mathsf{accept} \;\middle|\; \begin{array}{c} (\mathsf{pp}, \mathsf{vp}) \leftarrow \mathsf{Setup}(\cdot) \\ x \leftarrow \mathcal{A}_0(\cdot) \\ \pi \leftarrow \mathcal{A}_1(\mathsf{pp}, x) \end{array}\right] > \alpha,$$

where $\alpha = \alpha(\lambda) \neq \mathsf{negl}(\lambda)$ is a non-negligible probability, there exists a PPT extractor $\mathcal{E}^{\mathcal{A}}$ such that

$$\Pr\left[(x, w) \in \mathcal{R} \;\middle|\; x \leftarrow \mathcal{A}_0(\cdot),\ w \leftarrow \mathcal{E}^{\mathcal{A}}(x)\right] > \alpha - \epsilon,$$

where $\epsilon = \epsilon(\lambda)$ is a negligible function.

Remark. Informally, the aforementioned definition means that if the prover can generate a false proof with a non-negligible probability, then there exists an extractor that can extract the witness with a probability that is almost as high (and thus is also non-negligible).

Finally, to make zk-NARKs more universal and applicable to a wider range of problems, we introduce the **zk-SNARK** by adding the **succinctness** property.

Definition 8.4. A **zk-SNARK** (Succint NARK) is a preprocessing NARK, where the proof's length $|\pi|$ and verification time $T_{\mathcal{V}}$ are short: the verification time is sublinear in the size of the computation C (denoted by $|C|$), while the proof size is sublinear in the witness size $|w|$:

$$|\pi| = \mathsf{sublinear}(|w|),\ T_{\mathcal{V}} = O_\lambda(|x|, \mathsf{sublinear}(|C|)).$$

Remark. Sublinearity means that the function $f : \mathbb{N} \to \mathbb{R}$ grows slower than linearly. For example, functions $f(n) = \log n$ or $f(n) = \sqrt{n}$ are sublinear, while $f(n) = 3n + 2$ is linear. Generally, if $f(n)/(c \cdot n) \xrightarrow[n\to\infty]{} 0$ for any $c \in \mathbb{R} \setminus \{0\}$, then $f(n)$ is sublinear.

Example. Consider the protocol where the proof size is $|\pi| = O(\sqrt{|w|})$ and $T_{\mathcal{V}} = O(\sqrt[3]{|C|})$. Such protocol is a zk-SNARK, as the proof size is sublinear in the witness size and the verification time is sublinear in the size of the computation.

Although having a proof size and verification time lower than linear is nice, that is still not sufficient to make zk-SNARKs practical in the wild. For that reason, typically, in practice, we require a stricter definition of the succinctness property, where the proof size and verification time are constant or logarithmic in the size of the computation. This is the case for most zk-SNARKs used in practice.

Definition 8.5. A **zk-SNARK** is **strongly succinct** if the proof size and verification time are constant or logarithmic in the size of the computation:

$$|\pi| = O_\lambda(\log |C|),\ T_{\mathcal{V}} = O_\lambda(|x|, \log |C|).$$

Example. Consider three major proving systems used in practice with $N = |C|$ being the complexity of a computation:

- **Groth16** with $|\pi| = O_\lambda(1)$, $T_{\mathcal{V}} = O_\lambda(1)$ is definitely a strongly succinct zk-SNARK since both the proof size and verification time are constant.
- **STARK**s with $|\pi| = O_\lambda(\mathsf{polylog}(N))$ and $T_{\mathcal{V}} = O_\lambda(\mathsf{polylog}(N))$ are also strongly succinct zk-SNARKs since both the proof size and verification time are logarithmic in the size of the computation.
- **Bulletproofs** with $|\pi| = O_\lambda(\log N)$ and $T_{\mathcal{V}} = O_\lambda(N)$ is not a strongly succinct zk-SNARK since the verification time is linear in the size of the computation.

8.2 Arithmetic Circuits

8.2.1 What is Arithmetic Circuit?

The cryptographic tools we have learned in the previous lectures operate with numbers or certain primitives above them (like finite field extensions or elliptic curves), so the first question is: how do we convert a program into a mathematical language? Additionally, we need to do this in a way that can be further (a) made succinct, (b) allows us to prove something about it, and (c) be as universal as possible (to be able to prove quite general statements unlike Σ-protocols considered in the previous lecture).

The **Arithmetic Circuits** can help us with these problems. Similar to **Boolean Circuits**, they

consist of **gates** and **wires**: gates represent operations acting all elements, connected by wires (see figure below for details). Yet, instead of operations `AND`, `OR`, `NOT` and such, in arithmetic circuits only multiplication/addition/subtraction operations are allowed. Additionally, arithmetic circuits manipulate over elements from some finite field $\mathbb{F}$ (see right figure below).

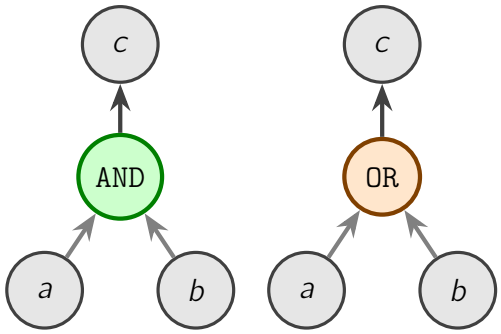


Figure 8.1: Boolean `AND` and `OR` Gates

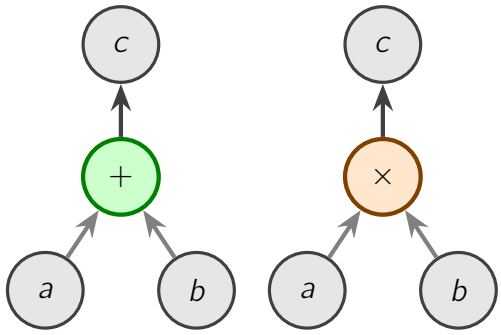


Figure 8.2: Addition and Multiplication Gates

Let us come back to boolean circuits for a moment and consider the `AND` gate. The *`AND` Gate Truth Table 1* shows us the results we receive if particular values are supplied to the gate. The main point here is that with this table, we can verify the validity of logical statements. Boolean circuits receive an input vector of $\{0, 1\}$ and resolve to `true` (1) or `false` (0); basically, they determine if the input values satisfy the statement.

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: `AND` Gate Truth Table

However, more notably, we can combine these gates to create more complex circuits that can resolve more complex problems. For example, we might construct a circuit depicted in Figure 8.3, calculating $(a \texttt{ AND } b) \texttt{ OR } c$.

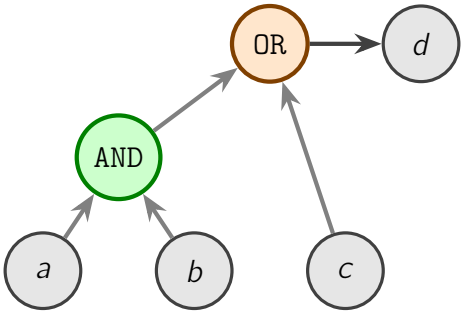


Figure 8.3: Example of a circuit evaluating $d = (a \texttt{ AND } b) \texttt{ OR } c$.

Although we can already represent very complex computations using boolean circuits[13], they are not the most convenient way to represent arithmetic operations.

[13] . . . such as `SHA-256` hash function computation, one might take a look here: http://stevengoldfeder.com/projects/circuits/sha2circuit.html

That being said, we can do the same with **arithmetic circuits** to verify computations over some finite field $\mathbb{F}$ without excessive verbosity due to a binary arithmetic, where we had to perceive all intermediate values as binary $\{0, 1\}$.

8.2.2 More advanced examples

Let us take a look at some examples of programs and how can we translate them to the arithmetic circuits.

Example 1: Multiplication. Consider a very simple program, where we are to simply multiply two field elements $a, b \in \mathbb{F}$:

```
def multiply(a: F, b: F) -> F:
    return a * b
```

Since we are doing all the arithmetic in a finite field $\mathbb{F}$, we denote it by `F` in the code. This can be represented as a circuit with only one (multiplication) gate:

$$r = a \times b$$

The witness vector (essentially, our solution vector) is $\mathbf{w} = (r, a, b)$, for example: $(6, 2, 3)$. We assume that the a and b are input values.

We can think of the "=" in the gate as an assertion, meaning that if $a \times b$ does not equal r, the assertion fails, and the input values do not resolve the circuit.

Good, but this one is quite trivial. Let's consider a more complex example.

Example 2: Multivariate Polynomial. Now, suppose we want to implement the evaluation of the polynomial $Q(x_1, x_2) = x_1^3 + x_2^2 \in \mathbb{F}[X_1, X_2]$ using arithmetic circuits. The corresponding program is as follows:

```
def evaluate(x1: F, x2: F) -> F:
    return x1**3 + x2**2
```

Looks easy, right? But the circuit is now much less trivial. Consider Figure 8.5. Notice that to calculate x_1^3 we cannot use the single gate: we need to multiply x_1 by itself two times. For that reason, we need three multiplication and one addition gate to represent $Q(x_1, x_2)$ calculation.

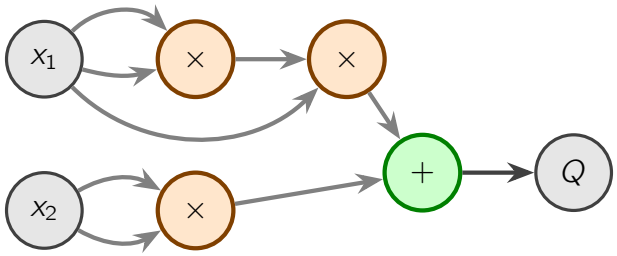


Figure 8.4: Example of a circuit evaluating $x_1^3 + x_2^2$.

Example 3. `if` **statements.** Well, it is quite clear how to represent any polynomial-like expressions. But how can we translate `if` statements? Consider the program below:

```
def if_statement_example(a: bool, b: F, c: F) -> F:
    return b * c if a else b + c
```

We can express this logic in mathematical terms as follows: "If a is true, compute $b \times c$; otherwise, compute $b + c$." However, only numerical expressions are allowed, so how can we proceed? Assuming that `true` is represented by 1 and `false` by 0, we can transform this logic as follows:

$$r = a \times (b \times c) + (1 - a) \times (b + c)$$

Now, what is the witness vector in this case? One might assume that $\mathbf{w} = (r, a, b, c)$ would suffice. Then, examples of valid witnesses include $(6, 1, 2, 3)$, $(5, 0, 2, 3)$.

But, we need to verify all the intermediate steps! This can be achieved by transforming the above equation using the simplest terms (the gates), ensuring the correctness of each step in the program.

Below, we show to visualize the arithmetic circuit for the `if` statement example.

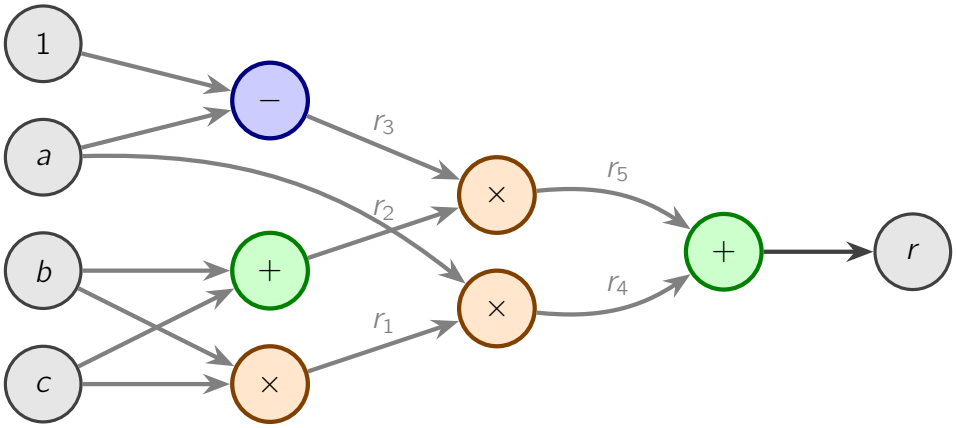


Figure 8.5: Example of a circuit evaluating the `if` statement logic.

Corresponding equations for the circuit are:

$$\begin{aligned} r_1 &= b \times c & r_2 &= b + c \\ r_3 &= 1 - a & r_4 &= a \times r_1 \\ r_5 &= r_3 \times r_2 & r &= r_4 + r_5 \end{aligned}$$

With the witness vector: $\mathbf{w} = (r, r_1, r_2, r_3, r_4, r_5, a, b, c)$. One example of a valid witness is $(6, 6, 5, 0, 6, 0, 1, 2, 3)$.

8.2.3 Circuit Satisfability Problem

Now, let us generalize what we have constructed so far. First, we begin with the arithmetic circuit.

Definition 8.6. Arithmetic circuit $C : \mathbb{F}^n \to \mathbb{F}$ with n inputs over a finite field $\mathbb{F}$ is a directed acyclic graph where internal nodes are labeled via $+$, $-$, and $\times$, and inputs are labeled $1, x_1, x_2, \ldots, x_n$. By $|C|$ we denote the number of gates in the circuit.

Example. For example, previously considered multivariate polynomial $C(x_1, x_2) = x_1^3 + x_2^2$ can be represented as an arithmetic circuit with three multiplication and one addition gates, as shown in Figure 8.5. It is defined over inputs $\mathbf{x} = (x_1, x_2)$ with $n = 2$ and $|C| = 4$.

Now, suppose that the circuit is defined over n inputs. We can always split this input into two parts: the first ℓ inputs are the *public inputs*, being our statement $\mathbf{x} \in \mathbb{F}^\ell$, and the remaining $n-\ell$ inputs are the *private inputs*, being our secret witness $\mathbf{w} \in \mathbb{F}^{n-\ell}$. The public inputs are known to everyone, while the private inputs are known only to the prover. The goal of the prover is to show that the circuit is satisfiable, i.e., that for the given $\mathbf{x}$, he *knows* a witness $\mathbf{w}$ that resolves the circuit. Resolving in this context means that the output of the circuit is zero.

Definition 8.7. The **Circuit Satisfiability Problem** is defined as follows: given an arithmetic circuit C and a public input $\mathbf{x} \in \mathbb{F}^\ell$, determine if there exists a private input $\mathbf{w} \in \mathbb{F}^{n-\ell}$ such that $C(\mathbf{x}, \mathbf{w}) = 0$. More formally, the problem is determined by relation $\mathcal{R}_C$ and corresponding language $\mathcal{L}_C$ as follows:

$$\mathcal{R}_C = \{(\mathbf{x}, \mathbf{w}) \in \mathbb{F}^\ell \times \mathbb{F}^{n-\ell} : C(\mathbf{x}, \mathbf{w}) = 0\},\ \mathcal{L}_C = \{\mathbf{x} \in \mathbb{F}^\ell : \exists \mathbf{w} \in \mathbb{F}^{n-\ell},\ C(\mathbf{x}, \mathbf{w}) = 0\}$$

Let us consider some concrete example of the Circuit Satisfiability Problem.

Example. Suppose our problem (as a prover) is to prove the verifier that we know the point on the circle of "radius $\sqrt{\rho}$"[a], but over the finite field $\mathbb{F}$. More formally, suppose we want to claim that for the given ρ, we have $x_1, x_2 \in \mathbb{F}$ such that:

$$x_1^2 + x_2^2 = \rho$$

For that reason, define the circuit $C(\rho, x_1, x_2) := x_1^2 + x_2^2 - \rho$. It is constructed as shown in the Figure below.

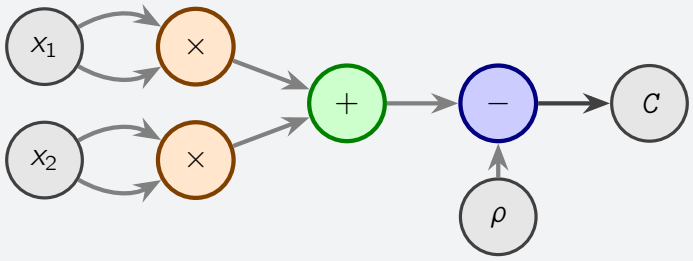


Illustration: *Arithmetic circuit for the equation* $x_1^2 + x_2^2 - \rho$.

Now, our statement vector is $\mathbf{x} = \rho \in \mathbb{F}$ (so $\ell = 1$) and the witness vector is $\mathbf{w} = (x_1, x_2) \in \mathbb{F}^2$ (so $n-\ell = 2$). The prover wants to prove that he knows the witness $\mathbf{w}$ such that $C(\mathbf{x}, \mathbf{w}) = 0$. For example, for $\rho = 5$, the prover might have the witness $\mathbf{w} = (2, 1)$ that he wants to show to the verifier[b].

[a] Note that in the finite field the circle equation does not have the geometrical form we are used to (similarly to Elliptic Curve equation, for instance)

[b] Here, $\mathbb{F} = \mathbb{F}_p$ for some prime $p > 5$

Now, as with any other previously considered proving systems, suppose we are not concerned about the zero-knowledge property and simply want to prove the evaluation integrity of the circuit. Can the prover simply send the witness $\mathbf{w}$ to the verifier? Prover can send the witness, but this will not be a SNARK (and, surely, not a zk-SNARK either).

Proposition 8.8 (Trivial SNARK is not a SNARK)**.** The protocol in which $\mathcal{P}$ sends the witness $\mathbf{w}$ to $\mathcal{V}$ is not a SNARK for the Circuit Satisfiability Problem. Indeed, in this case, the proof size is $|\pi| = |w|$ (since $\pi = w$) and the verification time is $T_{\mathcal{V}} = O(|\mathcal{C}|)$ (since $\mathcal{C}$ must be evaluated fully). We do not have succinctness (not even mentioning the strong succinctness) in this case.

Proposition above motivates us to look for more advanced techniques to prove the satisfiability of the arithmetic circuits. In the next section, we introduce the Rank-1 Constraint System, which is a more flexible and general way to describe the arithmetic circuits, allowing to further encode the constraints in a more succinct way.

8.3 Rank-1 Constraint System

Almost any program written in high-level programming language can be translated (compiled) into arithmetic circuits, that are really powerfull tool. But for the ZK proof we need slightly different format of it — **Rank-1 Constraint System**, where the simpliest term is **constraint**. This offers a more flexible and general way to describe these parts. However, we need a bit of Linear Algebra to be comfortable with this concept.

8.3.1 Linear Algebra Basics

Although we will not dive deep into the Linear Algebra, we need to understand some basic concepts to be able to work with the Rank-1 Constraint System.

Similarly to group theory working with *groups*, the linear algebra also has a special designated primitive — **vector space**. If previously we were working with the (finite) field $\mathbb{F}$, now we will work with the vector space V over this field. In many practical applications, vector space is formed by **vectors** consisting of a finite fixed collection of elements from the field $\mathbb{F}$. For example, the vector space might be simply $\mathbb{F}^n$: the set of all n-tuples $(x_1, x_2, \ldots, x_n)$ of elements from $\mathbb{F}$. Yet, let us give a bit more general definition.

Definition 8.9. A **vector space** V over the field $\mathbb{F}$ is an abelian group for addition $+$ together with a scalar multiplication operation $\cdot$ from $\mathbb{F} \times V$ to V, sending $(\lambda, x) \mapsto \lambda x$ and such that for any $\mathbf{v}, \mathbf{u} \in V$ and $\lambda, \mu \in \mathbb{F}$ we have:

- $\lambda(\mathbf{u} + \mathbf{v}) = \lambda\mathbf{u} + \lambda\mathbf{v}$
- $(\lambda + \mu)\mathbf{v} = \lambda\mathbf{v} + \mu\mathbf{v}$
- $(\lambda\mu)\mathbf{v} = \lambda(\mu\mathbf{v})$
- $1\mathbf{v} = \mathbf{v}$

Any element $\mathbf{v} \in V$ is called a **vector**, and any element $\lambda \in \mathbb{F}$ is called a **scalar**. We also mark vector elements in boldface.

Example. For example, $V = \mathbb{F}^n$ with operations defined as:

$$\lambda \cdot (x_1, x_2, \ldots, x_n) = (\lambda x_1, \lambda x_2, \ldots, \lambda x_n)$$
$$(x_1, x_2, \ldots, x_n) + (y_1, y_2, \ldots, y_n) = (x_1 + y_1, x_2 + y_2, \ldots, x_n + y_n)$$

is a vector space. Similarly, the following three sets V_1, V_2, V_3 with operations defined above are also valid vector spaces:

$$V_1 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_1 = 0\}$$
$$V_2 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_3 = 2\}$$
$$V_3 = \{(x_1, x_2, \ldots, x_n) \in \mathbb{F}^n : x_1 + x_2 + \cdots + x_n = 1\}$$

Now, besides vectors, frequently we are working with the **matrices**. The matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. For example, the matrix A with m rows and n columns, consisting of elements from the finite field $\mathbb{F}$ is denoted as $A \in \mathbb{F}^{m \times n}$. Additionally, we use notation $A = \{a_{i,j}\}_{i,j=1}^{m \times n}$ to denote the square matrix A of size $m \times n$ with elements $a_{i,j}$. Now, let us define operations on matrices.

Definition 8.10. Let A, B be two matrices over the field $\mathbb{F}$. The following operations are defined:

- **Matrix addition/subtraction**: $A \pm B = \{a_{i,j} \pm b_{i,j}\}_{i,j=1}^{m \times n}$. The matrices A and B must have the same size $m \times n$.
- **Scalar multiplication**: $\lambda A = \{\lambda a_{i,j}\}_{1 \le i,j \le n}$ for any $\lambda \in \mathbb{F}$.
- **Matrix multiplication**: $C = AB$ is a matrix $C \in \mathbb{F}^{m \times p}$ with elements $c_{i,j} = \sum_{\ell=1}^{n} a_{i,\ell} b_{\ell,j}$. The number of columns in A must be equal to the number of rows in B, that is $A \in \mathbb{F}^{m \times n}$ and $B \in \mathbb{F}^{n \times p}$.

Example. Suppose $\mathbb{F} = \mathbb{R}$. Then, consider

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 3}, \quad B = \begin{bmatrix} 2 & 1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$

We cannot add A and B since they have different sizes. However, we can multiply them:

$$AB = \begin{bmatrix} 5 & 6 \\ 7 & 9 \end{bmatrix}, \quad BA = \begin{bmatrix} 4 & 4 & 5 \\ 7 & 7 & 5 \\ 3 & 3 & 3 \end{bmatrix}$$

To see why, for example, the upper left element of AB is 5, we can calculate it as $\sum_{\ell=1}^{3} a_{1,\ell} b_{\ell,1} = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$.

Remark. Now, we add a very important remark. It just so happens that when working with vectors, we usually assume that they are **column vectors**. This means that the vector $v = (v_1, v_2, \ldots, v_n)$ is represented as a matrix:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

This is a common convention in linear algebra, and we will use it in the following sections.

One important operation we will be frequently working with is the **transpose** of the matrix. The transpose of a matrix is an operator that flips a matrix over its diagonal, that is, it switches the row and column indices of the matrix by producing another matrix denoted as $A^\top$.

Definition 8.11 (Transposition)**.** Given a matrix $A \in \mathbb{F}^{m \times n}$, the **transpose** of A is a matrix $A^\top \in \mathbb{F}^{n \times m}$ with elements $A^\top_{ij} = A_{ji}$.

Example. For example, consider the square matrix $A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$. Then, the transpose of A is $A^\top = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$. However, we can transpose any matrix, for example, the matrix $B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$ has the transpose $B^\top = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$. Finally, what is probably very important to us, the column vector $\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$ has the transpose $\mathbf{v}^\top = [1, 2, 3]$.

Finally, is just happens that we can construct matrix from the vectors. Therefore, let us introduce the corresponding notation.

Definition 8.12 (Composing Matrix from vectors)**.** Suppose we are given n vectors $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n \in \mathbb{F}^m$. Then, we might define matrix A as a matrix with columns $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$ as follows:

$$A = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 & \ldots & \mathbf{v}_n \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{2,1} & \ldots & v_{n,1} \\ v_{1,2} & v_{2,2} & \ldots & v_{n,2} \\ \vdots & \vdots & \ddots & \vdots \\ v_{1,m} & v_{2,m} & \ldots & v_{n,m} \end{bmatrix}$$

Alternatively, vectors might be represented as rows, and the matrix A might be defined as a matrix with rows $\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_n$:

$$A = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \\ \vdots \\ \mathbf{v}_n^\top \end{bmatrix} = \begin{bmatrix} v_{1,1} & v_{1,2} & \ldots & v_{1,m} \\ v_{2,1} & v_{2,2} & \ldots & v_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n,1} & v_{n,2} & \ldots & v_{n,m} \end{bmatrix}$$

Example. For example, consider the vectors $\mathbf{v}_1 = (1, 2, 3)$ and $\mathbf{v}_2 = (4, 5, 6)$. Then, the matrix A with columns $\mathbf{v}_1$ and $\mathbf{v}_2$ is:

$$A = \begin{bmatrix} \mathbf{v}_1 & \mathbf{v}_2 \end{bmatrix} = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

Similarly, the matrix B with rows $\mathbf{v}_1$ and $\mathbf{v}_2$ is:

$$B = \begin{bmatrix} \mathbf{v}_1^\top \\ \mathbf{v}_2^\top \end{bmatrix} = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}$$

We might go on with the vector spaces and define the **linear independence** and **basis** concepts, but for now we will skip them and move to the more important concept for us — the **inner** and **dot products**. Although **inner product** is typically introduced for ordered fields, we give a definition for our finite field $\mathbb{F}$.

Definition 8.13. Consider the vector space $\mathbb{F}^n$. The **inner product** is a function $\langle \cdot, \cdot \rangle : \mathbb{F}^n \times \mathbb{F}^n \to \mathbb{F}$ satisfying the following conditions for all $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{F}^n$:

- $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$.
- $\langle \mathbf{u}, \mathbf{v} + \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle + \langle \mathbf{u}, \mathbf{w} \rangle$.
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$ for all $\mathbf{u} \in \mathbb{F}^n$ iff $\mathbf{v} = \mathbf{0}$.
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$ for all $\mathbf{v} \in \mathbb{F}^n$ iff $\mathbf{u} = \mathbf{0}$.

Plenty of functions can be built that satisfy the inner product definition, we will use the one that is usually called **dot product**.

Definition 8.14. Consider the vector space $\mathbb{F}^n$. The **dot product** on $\mathbb{F}^n$ is a function $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \to \mathbb{F}$, defined for every $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ as follows:

$$\langle \mathbf{u}, \mathbf{v} \rangle := \mathbf{u}^\top \mathbf{v} = \sum_{i=1}^{n} u_i v_i$$

Alternatively, the dot product can also be denoted using the dot notation as $\mathbf{u} \cdot \mathbf{v}$. That is why it is called the "dot" product.

Example. Let $\mathbf{u}, \mathbf{v}$ are vectors over the real number $\mathbb{R}$, where

$$\mathbf{u} = (1, 2, 3), \quad \mathbf{v} = (2, 4, 3)$$

Then:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{3} u_i v_i = 2 \cdot 1 + 2 \cdot 4 + 3 \cdot 3 = 2 + 8 + 9 = 19$$

Yet another product we are going to use is the **Hadamard product**. Let us see how it works.

Definition 8.15. Suppose $A, B \in \mathbb{F}^{m \times n}$. The **Hadamard product** $A \odot B$ gives a matrix C such that $C_{i,j} = A_{i,j} B_{i,j}$. Essentially, we multiply elements elementwise.

Example. Consider $A = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 0 & 3 \end{bmatrix}, B = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}$. Then, the Hadamard product is:

$$A \odot B = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 2 & 2 \cdot 1 \\ 3 \cdot 0 & 0 \cdot 2 & 3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

Finally, the final ingredient is the **outer product** and some of its properties. So here it goes!

Definition 8.16. Given two vectors $\mathbf{u} \in \mathbb{F}^n$, $\mathbf{v} \in \mathbb{F}^m$ the **outer product** is a the matrix whose entries are all products of an element in the first vector with an element in the second vector:

$$\mathbf{u} \otimes \mathbf{v} := \mathbf{u}\mathbf{v}^\top = \begin{bmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{bmatrix}$$

Lemma 8.17 (Properties of outer product)**.** For any scalar $c \in \mathbb{F}$ and $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^p$:

- Transpose: $(\mathbf{u} \otimes \mathbf{v}) = (\mathbf{v} \otimes \mathbf{u})^\top$
- Distributivity: $\mathbf{u} \otimes (\mathbf{v} + \mathbf{w}) = \mathbf{u} \otimes \mathbf{v} + \mathbf{u} \otimes \mathbf{w}$
- Scalar Multiplication: $c(\mathbf{v} \otimes \mathbf{u}) = (c\mathbf{v}) \otimes \mathbf{u} = \mathbf{v} \otimes (c\mathbf{u})$
- Rank: the outer product $\mathbf{u} \otimes \mathbf{v}$ is a rank-1 matrix if $\mathbf{u}$ and $\mathbf{v}$ are non-zero vectors

Example. Let $\mathbf{u}, \mathbf{v}$ are vectors over the real number $\mathbb{R}$, where

$$\mathbf{u} = (1, 2, 3), \quad \mathbf{v} = (2, 4, 3)$$

Then:

$$\mathbf{u} \otimes \mathbf{v} = \mathbf{u}\mathbf{v}^\top = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \begin{bmatrix} 2 & 4 & 3 \end{bmatrix} = \begin{bmatrix} 1 \cdot 2 & 1 \cdot 4 & 1 \cdot 3 \\ 2 \cdot 2 & 2 \cdot 4 & 2 \cdot 3 \\ 3 \cdot 2 & 3 \cdot 4 & 3 \cdot 3 \end{bmatrix} = \begin{bmatrix} 2 & 4 & 3 \\ 4 & 8 & 6 \\ 6 & 12 & 9 \end{bmatrix}$$

Additionally, as we can see the rows number 2 and 3 in the result matrix can be represented as a linear combination of the first row, specifically by multiplying it by 2 and 3, respectively. The same property applies to the columns. This demonstrates the property of the outer product, that the resulting matrix has a rank of 1.

8.3.2 Constraint Definition

With knowledge of the inner product of two vectors, we can now formulate a definition of the constraint in the context of an R1CS.

Definition 8.18. Each **constraint** in the Rank-1 Constraint System must be in the form:

$$\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$$

Where $\mathbf{w}$ is a vector containing all the *input*, *output*, and *intermediate* variables involved in the computation. The vectors $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ are vectors of coefficients corresponding to these variables, and they define the relationship between the linear combinations of $\mathbf{w}$ on the left-hand side and the right-hand side of the equation.

Example. Consider the most basic circuit with one multiplication gate:

$$r = x_1 \times x_2$$

Since we have 3 variables, the constraint is written as:

$$(a_1 w_1 + a_2 w_2 + a_3 w_3)(b_1 w_1 + b_2 w_2 + b_3 w_3) = c_1 w_1 + c_2 w_2 + c_3 w_3$$

Coefficients and witness vectors are: $\mathbf{w} = (r, x_1, x_2), \mathbf{a} = (0, 1, 0), \mathbf{b} = (0, 0, 1), \mathbf{c} = (1, 0, 0)$. Therefore, our expression above reduces to:

$$(0w_1 + 1w_2 + 0w_3)(0w_1 + 0w_2 + 1w_3) = (1w_1 + 0w_2 + 0w_3)$$

$$w_2 \times w_3 = w_1$$

$$x_1 \times x_2 = r$$

The interesting thing is where to take a constants from. The solution is straightforward: by placing 1 in the witness vector, so we can obtain any desired value by multiplying it by an appropriate coefficient.

Example. Now, let us consider a more complex example. Remember that we want to verify each computational step.

```
def r(x1: bool, x2: F, x3: F) -> F:
    return x2 * x3 if x1 else x2 + x3
```

We know that it can be expressed as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3)$$

However, one important consideration was overlooked. If x_1 is neither 0 nor 1, it implies that something else is being computed instead of the desired program. Since we need to add a restriction for x_1: $x_1 \times (1 - x_1) = 0$, this effectively checks that x_1 is binary.
The next constraints can be build:

$$x_1 \times x_1 = x_1 \quad \text{(binary check)} \tag{1}$$
$$x_2 \times x_3 = \text{mult} \tag{2}$$
$$x_1 \times \text{mult} = \text{selectMult} \tag{3}$$
$$(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \tag{4}$$

For every constraint we need the coefficients vectors a_i, b_i, c_i, but all of them have the same witness vector $\mathbf{w}$.

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \text{mult}, \text{selectMult})$$

The coefficients vectors:

$$\mathbf{a}_1 = (0,0,1,0,0,0,0) \quad \mathbf{b}_1 = (0,0,1,0,0,0,0) \quad \mathbf{c}_1 = (0,0,1,0,0,0,0)$$
$$\mathbf{a}_2 = (0,0,0,1,0,0,0) \quad \mathbf{b}_2 = (0,0,0,0,1,0,0) \quad \mathbf{c}_2 = (0,0,0,0,0,1,0)$$
$$\mathbf{a}_3 = (0,0,1,0,0,0,0) \quad \mathbf{b}_3 = (0,0,0,0,0,1,0) \quad \mathbf{c}_3 = (0,0,0,0,0,0,1)$$
$$\mathbf{a}_4 = (1,0,-1,0,0,0,0) \quad \mathbf{b}_4 = (0,0,0,1,1,0,0) \quad \mathbf{c}_4 = (0,1,0,0,0,0,-1)$$

Now, let us use some specific values to compute an example. Using the arithmetic in a large finite field $\mathbb{F}_p$, consider the following values:

$$x_1 = 1, \quad x_2 = 3, \quad x_3 = 4$$

Verifying the constraints:

1. $x_1 \times x_1 = x_1 \quad (1 \times 1 = 1)$
2. $x_2 \times x_3 = \text{mult} \quad (3 \times 4 = 12)$
3. $x_1 \times \text{mult} = \text{selectMult} \quad (1 \times 12 = 12)$
4. $(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \quad (0 \times 7 = 12 - 12)$

Each constraint enforces that the product of the linear combinations defined by $\mathbf{a}$ and $\mathbf{b}$ must equal the linear combination defined by $\mathbf{c}$. Collectively, these constraints describe the computation by ensuring that every step, from inputs through intermediates to outputs, satisfies the defined relationships, thus encoding the entire computational process in the form of a system

of rank-1 quadratic equations.

8.3.3 Why Rank-1?

The last unresolved question is where the "rank-1" comes from. Using the outer product we can express the constraint in another form.

Lemma 8.19. Suppose we have a constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$ with coefficient vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and witness vector $\mathbf{w}$ (all from $\mathbb{F}^n$). Then it can be expressed in the form:

$$\mathbf{w}^\top A \mathbf{w} + \mathbf{c}^\top \mathbf{w} = 0$$

Where A is the outer product of vectors $\mathbf{a}$, $\mathbf{b}$ (denoted as $\mathbf{a} \otimes \mathbf{b}$), consequently a **rank-1** matrix.

Lemma proof. Consider the constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$, where $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{w} \in \mathbb{F}^n$. Let us expand the inner products:

$$\left(\sum_{i=1}^{n} a_i w_i \right) \times \left(\sum_{j=1}^{n} b_j w_j \right) = \sum_{k=1}^{n} c_k w_k$$

Combine the products into a double sum on the left side:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j w_i w_j = \mathbf{w}^\top (\mathbf{a} \otimes \mathbf{b}) \mathbf{w} = \mathbf{w}^\top A \mathbf{w}$$

Thus, the constraint can be written as:

$$\mathbf{w}^\top A \mathbf{w} + \mathbf{c}^\top \mathbf{w} = 0$$

So, the rank-1 means the rank of the coefficients matrix A in one of the constraint formats.

9 Quadratic Arithmetic Program. Probabilistically Checkable Proofs

9.1 Quadratic Arithmetic Program

9.1.1 R1CS in Matrix Form

While the Rank-1 Constraint System provides a powerful way to represent computations, it is not succinct at all, since the number of constraints depends linearly on the complexity of the problem being solved. In practical scenarios, this can require tens or even hundreds of thousands of constraints, sometimes even millions. The Quadratic Arithmetic Program (QAP) can address this issue.

Remark. Understanding polynomials and their properties is crucial for this section. If you are not confident in this area, it is better to revisit the corresponding chapter and refresh your knowledge. See Section 1.4.

To define a constraint in the R1CS we need four vectors: three coefficient vectors ($\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$) and the witness one ($\mathbf{w}$). And that's just for one constraint. As you can imagine, many of the values in these vectors are zeros. In circuits with thousands of inputs, outputs, and auxiliary variables, where there are also thousands of constraints, you could end up with a millions of zeroes.

Remark. A matrix in which most of the elements are zero in numerical analysis is usually called **sparse matrix**.

So, we need to change the way how we manage coefficients and make the representation of such matrices and vectors succint (as required by the definition of SNARK).

Theorem 9.1. Consider a Rank-1 Constraint System (R1CS) defined by m constraints. Each constraint is associated with coefficient vectors $\mathbf{a}_i$, $\mathbf{b}_i$, and $\mathbf{c}_i$, where $i \in \{1, 2, \ldots, m\}$ and also a witness vector $\mathbf{w}$ consisting of n elements.
Then this system can also be represented using the corresponding matrices A, B, and C.

$$A = \begin{bmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{bmatrix} \quad B = \begin{bmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{bmatrix} \quad C = \begin{bmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \ldots & c_{mn} \end{bmatrix}$$

such that all constraints can be reduced to the single equation:

$$A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w}$$

In this representation:

- Each i-th row of the matrices corresponds to the coefficients of a specific constraint.
- Each column of these matrices corresponds to the coefficients associated with a particular element of the witness vector $\mathbf{w}$.

Proof. Matrices defined this way can be expressed as

$$A = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}, \quad B = \begin{bmatrix} \mathbf{b}_1^\top \\ \mathbf{b}_2^\top \\ \vdots \\ \mathbf{b}_m^\top \end{bmatrix}, \quad C = \begin{bmatrix} \mathbf{c}_1^\top \\ \mathbf{c}_2^\top \\ \vdots \\ \mathbf{c}_m^\top \end{bmatrix}$$

Consider an expression $A\mathbf{w}$:

$$A\mathbf{w} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{w} \\ \mathbf{a}_2^\top \mathbf{w} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{w} \end{bmatrix}$$

The last equality is a bit tricky to observe, so let us explain how we ended up with such expression. Notice that since $A \in \mathbb{F}^{m \times n}$ and $\mathbf{w} \in \mathbb{F}^n$, the product $A\mathbf{w}$ is a vector from $\mathbb{F}^m$. Now, for j^{th} element of such vector, based on the matrix product definition, we have $(A\mathbf{w})_j = \sum_{\ell=1}^{n} a_{j,\ell} w_\ell$ which is exactly an inner product between $\mathbf{a}_j$ and $\mathbf{w}$! Therefore, we have:

$$A\mathbf{w} = \begin{bmatrix} \langle \mathbf{a}_1, \mathbf{w} \rangle \\ \langle \mathbf{a}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{a}_m, \mathbf{w} \rangle \end{bmatrix}, \quad B\mathbf{w} = \begin{bmatrix} \langle \mathbf{b}_1, \mathbf{w} \rangle \\ \langle \mathbf{b}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{b}_m, \mathbf{w} \rangle \end{bmatrix}, \quad C\mathbf{w} = \begin{bmatrix} \langle \mathbf{c}_1, \mathbf{w} \rangle \\ \langle \mathbf{c}_2, \mathbf{w} \rangle \\ \vdots \\ \langle \mathbf{c}_m, \mathbf{w} \rangle \end{bmatrix}$$

Therefore, $A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w}$ is equivalent to the system of m constraints:

$$\langle \mathbf{a}_j, \mathbf{w} \rangle \times \langle \mathbf{b}_j, \mathbf{w} \rangle = \langle \mathbf{c}_j, \mathbf{w} \rangle, \; j \in \{1, \ldots, m\}.$$

Example. The vectors $\mathbf{a}_i$ from the previous examples have the form:

$$\begin{aligned} \mathbf{a}_1 &= (0, 0, 1, 0, 0, 0, 0) \\ \mathbf{a}_2 &= (0, 0, 0, 1, 0, 0, 0) \\ \mathbf{a}_3 &= (0, 0, 1, 0, 0, 0, 0) \\ \mathbf{a}_4 &= (1, 0, -1, 0, 0, 0, 0) \end{aligned}$$

This corresponds to $n = 7, m = 4$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,4} & a_{1,5} & a_{1,6} & a_{1,7} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,4} & a_{2,5} & a_{2,6} & a_{2,7} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,4} & a_{3,5} & a_{3,6} & a_{3,7} \\ a_{4,1} & a_{4,2} & a_{4,3} & a_{4,4} & a_{4,5} & a_{4,6} & a_{4,7} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

9.1.2 Polynomial Interpolation

OK, now is the time to define how we are going to build polynomials! Notice that the columns of these matrices (say, column $(a_{1,i}, a_{2,i}, a_{3,i}, a_{4,i})$ in matrix A from example above) represent

the mappings from constraint number i to the corresponding coefficient of the j element in the witness vector!

Example. Consider the witness from the previous examples:

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \mathsf{mult}, \mathsf{selectMult})$$

For element x_1 we are interested in the third columns of the A, B and C matrices, as it's placed on the third position in the witness vector, so $j = 3$.
For matrix A:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Thus, for constraint number 4 ($i = 4$) the coefficient of x_1 is -1:

$$\begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Good, so now we know that for the jth element in the witness vector there are m (the number of constraints) corresponding values in matrices A, B, and C. Now, we want to encode this statement in a form of a polynomial. As we know from the previous chapters, such a mapping in math can be built using the Lagrange polynomial interpolation.

Remark. As a remainder, the Lagrange interpolation polynomial for a given set of points $\{(x_0, y_0), (x_1, y_1), \ldots, (x_n, y_n)\} \subset \mathbb{F} \times \mathbb{F}$ can be built with the following formula:

$$L(x) = \sum_{i=0}^{n} y_i \ell_i(x), \quad \ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j}.$$

For a given column $j \in \{1, 2, \ldots, n\}$ in a matrix A the set of points that define the variable polynomial $A_j(x)$ can be defined as $\{(i, a_{ij}) : i \in \{1, 2, \ldots, m\}\}$. In other words, we want to interpolate n polynomials $A_j \in \mathbb{F}[X]$ such that:

$$A_j(i) = a_{i,j}, \ i \in \{1, 2, \ldots, m\}, \ j \in \{1, 2, \ldots, n\}$$

The same is true for matrices B and C, resulting in $3n$ polynomials, n for each of the coefficients matrices:

$$A_1(x), A_2(x), \ldots, A_n(x), B_1(x), B_2(x), \ldots, B_n(x), C_1(x), C_2(x), \ldots, C_n(x)$$

Example. Considering the witness vector **w** and matrix A from the previous example, for the variable x_1, the next set of points can be derived:

$$\{(1,1),(2,0),(3,1),(4,-1)\}$$

We can see that it is used in the 1st, 3rd, and 4th constraints as the values of the coefficients are not zero.
The Lagrange interpolation polynomial for this set of points can be built as follows (for the demonstration purposes, assume we are working in the field $\mathbb{R}$):

$$\begin{aligned}
\ell_1(x) &= \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} = -\frac{(x-2)(x-3)(x-4)}{6}, \\
\ell_2(x) &= \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} = \frac{(x-1)(x-3)(x-4)}{2}, \\
\ell_3(x) &= \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} = -\frac{(x-1)(x-2)(x-4)}{2}, \\
\ell_4(x) &= \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} = \frac{(x-1)(x-2)(x-3)}{6}.
\end{aligned}$$

Thus, the polynomial is given by:

$$\begin{aligned}
A_1(x) &= 1 \cdot \ell_1(x) + 0 \cdot \ell_2(x) + 1 \cdot \ell_3(x) + (-1) \cdot \ell_4(x) \\
&= -\frac{(x-2)(x-3)(x-4)}{6} - \frac{(x-1)(x-2)(x-4)}{2} - \frac{(x-1)(x-2)(x-3)}{6} \\
&= -\frac{5}{6}x^3 + 6x^2 - \frac{79}{6}x + 9
\end{aligned}$$

Therefore, the final Lagrange interpolation polynomial is:

$$A_1(x) = -\frac{5}{6}x^3 + 6x^2 - \frac{79}{6}x + 9$$

As shown in Illustration below, the curve intersects all the given points. In this figure, the x-axis represents the constraint number, and the y-axis represents the coefficients of the x_1 witness element.

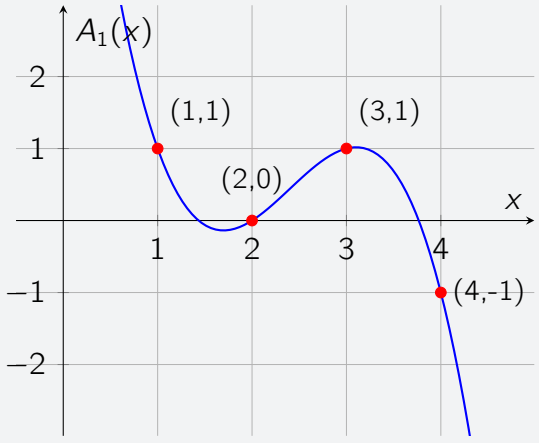


Illustration: The Lagrange inteprolation polynomial for points $\{(1,1),(2,0),(3,1),(4,-1)\}$ visualized over $\mathbb{R}$.

Remark. One might a reasonable question: why do we choose x-coordinates to be the indeces of the corresponding constraints? Actually, just for convenience purposes. We could have assigned any *unique* index from $\mathbb{F}$ to each constraint (say, t_i for each $i \in \{1, \dots, m\}$) and interpolate through these points:

$$A_j(t_i) = a_{i,j},\ i \in \{1, 2, \dots, m\},\ j \in \{1, 2, \dots, n\}$$

As we will see in the subsequent lectures, we can define the x-coordinates in much more clever way to reduce the workload needed for interpolation. But for now, we will stick to this simple version.

Remark. The degree of the coefficient polynomials does not exceed $m - 1$, which follows from the Lagrange interpolation properties.

9.1.3 Putting All Together!

Now, using coefficients encoded with polynomials, a constraint number $X \in \{1, \dots\ m\}$, from a constraint system with a witness vector $\mathbf{w}$ can be built in the next way:

$$(w_1A_1(X) + w_2A_2(X) + \dots + w_nA_n(X)) \times (w_1B_1(X) + w_2B_2(X) + \dots + w_nB_n(X)) = \\ = (w_1C_1(X) + w_2C_2(X) + \dots + w_nC_n(X))$$

Or, written more concisely:

$$\left(\sum_{i=1}^{n} w_iA_i(X)\right) \times \left(\sum_{i=1}^{n} w_iB_i(X)\right) = \left(\sum_{i=1}^{n} w_iC_i(X)\right)$$

Remark. Hold on, but why does it hold? Let us substitute any $X = j$ into this equation:

$$\left(\sum_{i=1}^{n} w_iA_i(j)\right) \times \left(\sum_{i=1}^{n} w_iB_i(j)\right) = \left(\sum_{i=1}^{n} w_iC_i(j)\right)\ \forall j \in \{1, \dots, m\}$$

Recall that we interpolated polynomials to have $A_i(j) = a_{j,i}$. Therefore, the equation above can be reduced to:

$$\left(\sum_{i=1}^{n} w_ia_{j,i}\right) \times \left(\sum_{i=1}^{n} w_ib_{j,i}\right) = \left(\sum_{i=1}^{n} w_ic_{j,i}\right)\ \forall j \in \{1, \dots, m\}$$

But hold on again! Notice that $\sum_{i=1}^{n} w_ia_{j,i} = \langle \mathbf{w}, \mathbf{a}_j \rangle$ and therefore we have:

$$\langle \mathbf{w}, \mathbf{a}_j \rangle \times \langle \mathbf{w}, \mathbf{b}_j \rangle = \langle \mathbf{w}, \mathbf{c}_j \rangle\ \forall j \in \{1, \dots, m\},$$

so we ended up with the initial m constraint equations!

Now let us define polynomials $A(X)$, $B(X)$, $C(X)$ for easier notation:

$$A(X) = \sum_{i=1}^{n} w_iA_i(X), \quad B(X) = \sum_{i=1}^{n} w_iB_i(X), \quad C(X) = \sum_{i=1}^{n} w_iC_i(X)$$

Therefore, our constraint can be rewritten as $A + B = C$ — much less scary-looking than what we have written before. OK, but what does it give us?

Notice that if $A(X) \times B(X) = C(X)$ for all $j \in \{1, \dots, m\}$ then polynomial, defined as $M(X) := A(X) \times B(X) - C(X)$, has zeros at all elements from the set $\Omega = \{1, \dots, m\}$. Define the so-called **vanishing polynomial** on Ω as:

$$Z_\Omega(X) := \prod_{\omega \in \Omega} (X - \omega) = \prod_{i=1}^{m} (X - i)$$

Now, if $M(X)$ vanishes on all points from Ω, it means that Z_Ω must divide M, so $Z_\Omega \mid M$. But that means that M can be divided by Z_Ω without remainder! In other words, there exists some polynomial H such that $M = Z_\Omega H$. We further drop index Ω for simplicity.

All in all, let us give the definition of a **Quadratic Arithmetic Program**.

Definition 9.2 (Quadratic Arithmetic Program)**.** Suppose that m R1CS constraints with a witness of size n are written in a form

$$A\mathbf{w} \odot B\mathbf{w} = C\mathbf{w},\ A, B, C \in \mathbb{F}^{m \times n}$$

Then, the **Quadratic Arithmetic Program** consists of $3n$ polynomials $A_1, \dots, A_n$, $B_1, \dots, B_n$, $C_1, \dots, C_n$ such that:

$$A_j(i) = a_{i,j},\ B_j(i) = b_{i,j},\ C_j(i) = c_{i,j},\ \forall i \in \{1, \dots, m\}\ \forall j \in \{1, \dots, n\}$$

Then, $\mathbf{w} \in \mathbb{F}^n$ is a valid assignment for the given QAP and **target polynomial** $Z(X) = \prod_{i=1}^{m} (X - i)$ if and only if there exists such a polynomial $H(X)$ such that

$$\left(\sum_{i=1}^{n} w_i A_i(X)\right)\left(\sum_{i=1}^{n} w_i B_i(X)\right) - \left(\sum_{i=1}^{n} w_i C_i(X)\right) = Z(X)H(X)$$

This was our final step in representing a high-level programming language to some math primitive. We have managed to encode our computation to a single polynomial!

Remark on operations between polynomials

Remark. Some pretty obvious property should be noted. In the theorem **??** it was said about the degree of polynomials after their multiplication or addition, but what about their values?

Let $p(x), q(x) \in \mathbb{F}[x]$ be two polynomials over a field $\mathbb{F}$. Define the polynomial $r(x)$ as the sum of $p(x)$ and $q(x)$:

$$r(x) = p(x) + q(x)$$

Then, for any point $x \in \mathbb{F}$, the value of $r(x)$ is equal to the sum of the values of $p(x)$ and $q(x)$ at that point. Therefore, the set of points corresponding to the polynomial $r(x)$ is given by:

$$\{(x, y) \in \mathbb{F} \times \mathbb{F} \mid x \in \mathbb{F},\ y = p(x) + q(x)\}$$

The same is true for product.

Example. Consider two polynomials $p(x)$ and $q(x)$ defined over the real numbers $\mathbb{R}$:

$$p(x) = -\frac{1}{2}x^2 + \frac{3}{2}x, \quad q(x) = \frac{1}{3}x^3 - 2x^2 + \frac{8}{3}x + 1.$$

The sets of points $\{(0,0),(1,1),(2,1),(3,0)\}$ and $\{(0,1),(1,2),(2,1),(3,0)\}$ lie on the graphs of $p(x)$ and $q(x)$, respectively.

The sum of these polynomials can be calculated as:

$$\begin{aligned} r(x) &= (-\frac{1}{2}x^2 + \frac{3}{2}x) + (\frac{1}{3}x^3 - 2x^2 + \frac{8}{3}x + 1) \\ &= \frac{1}{3}x^3 - 2\frac{1}{2}x^2 + 4\frac{1}{6}x + 1 \end{aligned}$$

The resulting polynomial $r(x)$ corresponds to the set of points $\{(0,1),(1,3),(2,2),(3,0)\}$. As you can see (Figure 9.1), the values at each point for the corresponding x are the sum of the initial polynomials' points.

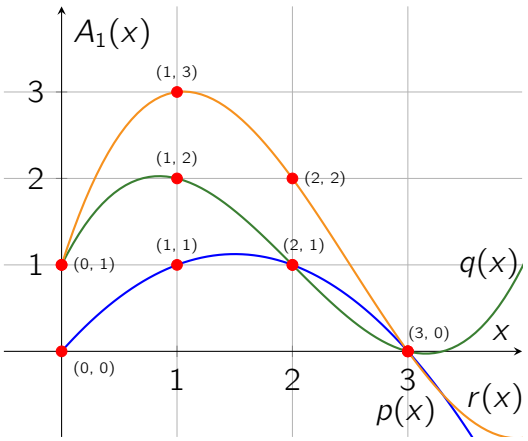


Figure 9.1: Addition of two polynomials

9.2 Probabilistically Checkable Proofs

Before going further we should get acquainted with one more concept from the computational complexity theory, that have an important application in zk-SNARK and provides the theoretical backbone.

A Probabilistically Checkable Proof (PCP) is a type of proof system where the verifier can efficiently check the correctness of a proof by examining only a small, random portion of it, rather than verifying it entirely.

Definition 9.3. A language $\mathcal{L} \subseteq \Sigma^*$ (for some given alphabet Σ) is in the class $\mathsf{PCP}(r, q)$ (**probabilistically checkable proofs**), where r is the *randomness complexity* and q is the *query complexity*, if for a given pair of algorithms $(\mathcal{P}, \mathcal{V})$:

- *Syntax:* $\mathcal{P}$ calculates a proof (bit string) $\pi \in \Sigma^*$ in polynomial time $\mathsf{poly}(|x|)$ of the common input x. The prover $\mathcal{P}$ and verifier $\mathcal{V}$ interact, where the verifier has an oracle access to π (meaning, he queries it at any position).
- *Complexity:* $\mathcal{V}$ uses at most r random bits to decide which part of the proof to query and the verifier queries at most q bits of the proof.

Such pair of algorithms $(\mathcal{P}, \mathcal{V})$ should satisfy the following properties (with a security parameter $\lambda \in \mathbb{N}$):

- **Completeness**: If $x \in \mathcal{L}$, then $\Pr[\mathcal{V}^{\pi}(x) = 1] = 1$.
- **Soundness**: If $x \notin \mathcal{L}$, then for any possible (malicious) proof π^*,

$$\Pr\left[\mathcal{V}^{\pi^*}(x) = 1\right] = \mathsf{negl}(\lambda).$$

This allows a verification of huge statements with high confidence while using limited computational resources. See Figure 9.3.

Theorem 9.4. PCP theorem (PCP characterization theorem)
Any decision problem in NP has a PCP verifier that uses logarithmic randomness $O(\log n)$ and a constant number of queries $O(1)$, independent of n.

$$\mathsf{NP} = \mathsf{PCP}(O(\log n), O(1))$$

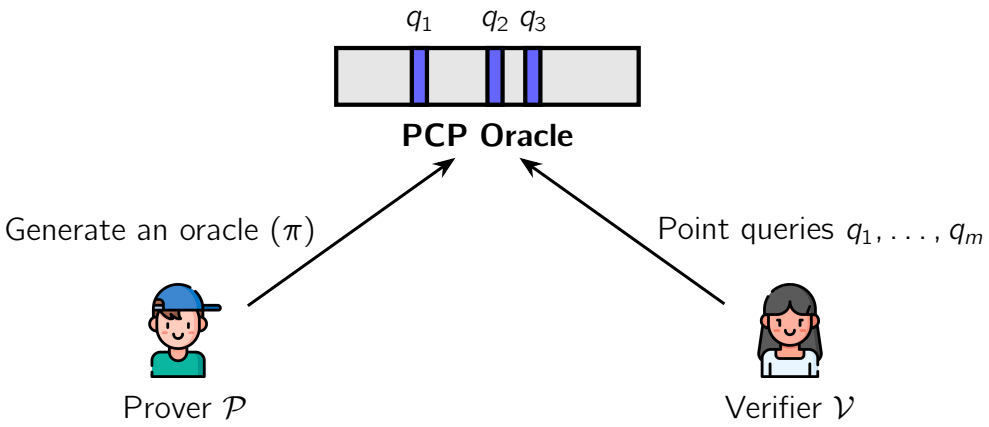


Figure 9.2: Illustration of a Probabilistically Checkable Proof (PCP) system. The prover $\mathcal{P}$ generates a PCP oracle π that is queried by the verifier $\mathcal{V}$ at specific points $q_1, \ldots, q_m$.

However, despite the fact that PCP is a very powerful tool, it is not used directly in zk-SNARKs. We need to extend it a bit to make it more suitable for our purposes.

Three main extensions of PCPs that are frequently used in SNARKs are:

- **IPCP** (**Interactive PCP**): The prover commits to the PCP oracle and then, based on the interaction between the prover and verifier, the verifier queries the oracle and decides whether to accept the proof.

- **IOP** (**Interactive Oracle Proof**): The prover and verifier interact and on each round, the prover commits to a new oracle. The verifier queries the oracle and decides whether to accept the proof.
- **LPCP** (**Linear PCP**): The prover commits to a linear function and the verifier queries the function at specific points.

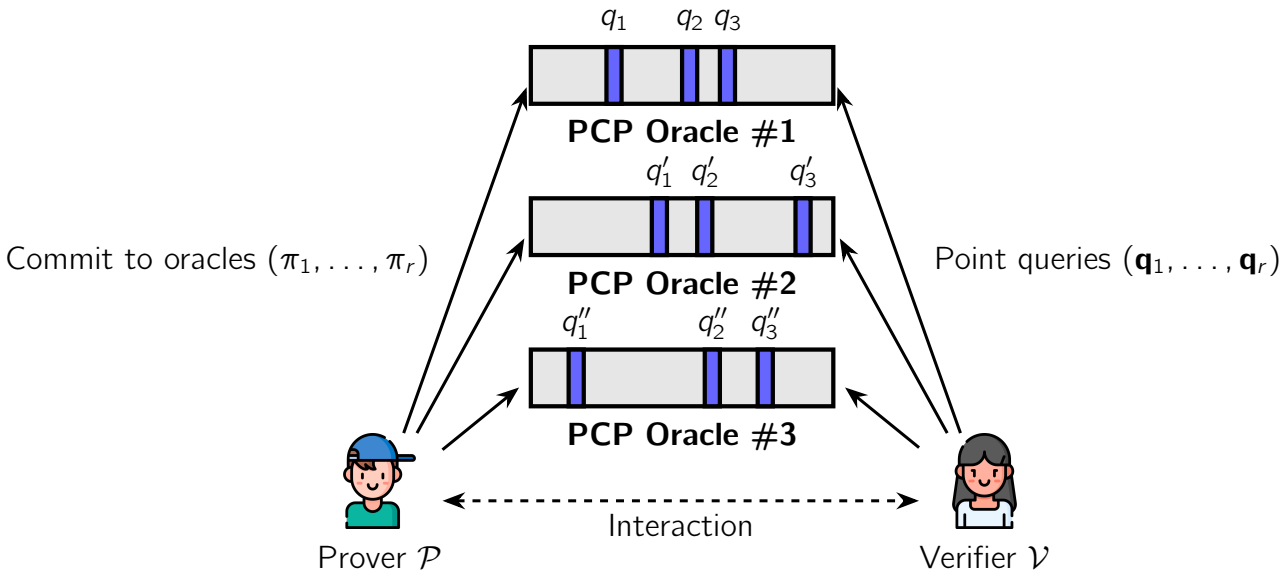


Figure 9.3: Illustration of an Interactive Oracle Proof (IOP). On each round i ($1 \leq i \leq r$), $\mathcal{V}$ sends a message m_i, and $\mathcal{P}$ commits to a new oracle π_i, which $\mathcal{V}$ can query at $\mathbf{q}_i = (q_{i,1}, \ldots, q_{i,m})$.

While IOPs will be later used for PLONK and zk-STARKs, we will focus on Linear PCPs in the context of Groth16 zk-SNARK. Let us define it below.

Definition 9.5 (Linear PCP)**.** A **Linear PCP** is a PCP where the prover commits to a linear function $\boldsymbol{\pi} = (\pi_1, \ldots, \pi_k)$ and the verifier queries the function at specific points $\boldsymbol{q}_1, \ldots, \boldsymbol{q}_r$. Then, the prover responds with the values of the function at these points:

$$\langle \boldsymbol{\pi}_1, \mathbf{q}_1 \rangle, \langle \boldsymbol{\pi}_2, \mathbf{q}_2 \rangle, \ldots, \langle \boldsymbol{\pi}_r, \mathbf{q}_r \rangle.$$

Example (QAP as a Linear PCP)**.** Recall that key QAP equation is:

$$\left(\sum_{i=1}^{n} w_i L_i(x)\right)\left(\sum_{i=1}^{n} w_i R_i(x)\right) - \left(\sum_{i=1}^{n} w_i O_i(x)\right) = Z(x)H(x).$$

Now, the notation might be confusing, but firstly, we denote vectors of polynomials:

$$\begin{aligned}\boldsymbol{L}(x) &= (L_1(x), \ldots, L_n(x)),\\ \boldsymbol{R}(x) &= (R_1(x), \ldots, R_n(x)),\\ \boldsymbol{O}(x) &= (O_1(x), \ldots, O_n(x)).\end{aligned}$$

Now, consider the following **linear PCP for QAP**:

1. $\mathcal{P}$ commits to an extended witness $\mathbf{w}$ and coefficients $\mathbf{h} = (h_1, \ldots, h_n)$ of $H(x)$.
2. $\mathcal{V}$ samples $\gamma \xleftarrow{R} \mathbb{F}$ and sends query $\boldsymbol{\gamma} = (\gamma, \gamma^2, \ldots, \gamma^n)$ to $\mathcal{P}$.
3. $\mathcal{P}$ reveals the following values:

$$\pi_1 \leftarrow \langle \mathbf{w}, \boldsymbol{L}(\gamma)\rangle, \quad \pi_2 \leftarrow \langle \mathbf{w}, \boldsymbol{R}(\gamma)\rangle, \quad \pi_3 \leftarrow \langle \mathbf{w}, \boldsymbol{O}(\gamma)\rangle, \quad \pi_4 \leftarrow Z(\gamma) \cdot \langle \mathbf{h}, \boldsymbol{\gamma}\rangle.$$

4. $\mathcal{V}$ checks whether $\pi_1\pi_2 - \pi_3 = \pi_4$.

Of course, the above example cannot be used as it is: at the very least, we have not specified how the prover commits to the extended witness $\mathbf{w}$ and coefficients $\mathbf{h}$ and then ensures consistency of $\pi_1, \ldots, \pi_4$ with these commitments. For that reason, we need some more tools to make it work which we learned in the previous lectures.

9.3 QAP as a Linear PCP

When constructing a Quadratic Arithmetic Program (QAP) for a circuit C, we represented the whole circuit's computation using the following relation:

$$L(x)R(x) - O(x) = Z(x)H(x),$$

where by $L(x)$, $R(x)$, $O(x)$ we denote the polynomials that represent the left, right and output wires of the circuit, respectively. $Z(x)$ is the target polynomial, while $H(x) := M(x)/Z(x)$ for master polynomial $M(x) = L(x)R(x) - O(x)$ is the quotient polynomial.

We effectively managed to transform all the circuit's constraints, and computations in the short form. It still allows one to verify that each computational step is preserved by verifying the polynomial evaluation in specific (random) points, instead of recomputing everything. However, it is not quite clear why such a check is safe and how it can be used in a PCP. In other words, why checking that $L(s)R(s) - O(s) = Z(s)H(s)$ for randomly selected s is enough to verify the circuit C?

Soundness justification. Why is it safe to use such a check? As it was said early, we perform all the computations in some finite field $\mathbb{F}$. The polynomials $L(x)$, $R(x)$ and $O(x)$ are interpolated polynomials using $|C|$ (number of gates) points, so

$$\deg(L) \leq |C|, \quad \deg(R) \leq |C|, \quad \deg(O) \leq |C|$$

Thus, using properties of polynomials' degrees, we can estimate the degree of polynomial $M(x) = L(x)R(x) - O(x)$.

$$\deg(M) \leq \max\{\deg(L) + \deg(R), \deg(O)\} \leq 2\,|\mathcal{C}|$$

Now, using the Schwartz-Zippel Lemma (see Lemma 2.12), we can deduce that if an adversary $\mathcal{A}$ does not know a valid witness $\mathbf{w}$, resolving the circuit $\mathcal{C}$, he can compute a polynomial $\widetilde{M}(x) \leftarrow \mathcal{A}(\cdot)$ that satisfies a verifier $\mathcal{V}$ with probability less than $2\,|\mathcal{C}|\,/|\mathbb{F}|$. To put it formally, we can write:

$$\Pr_{s \xleftarrow{R} \mathbb{F}}\left[\widetilde{M}(s) = M(s)\right] \leq \frac{2\,|\mathcal{C}|}{|\mathbb{F}|}$$

This probability becomes negligible as $|\mathbb{F}|$ grows large (which is typically the case), giving us soundness. In the same time, the verifier accepts the $M(x)$ generated using a valid witness with probability 1 giving us the completeness, so, we can categorize QAP as PCP.

We will modify the form of our proof with the next modifications, but still preserve the PCP properties.

In the following sections, we will introduce tools needed to succintly verify the equality above using the PCP properties. Since the overall proof is very complex from the very first glance, we will break it down into smaller parts and explain each of them in detail.

10 Pairing-based SNARKs. Pinocchio and Groth16

10.1 Building Pairing-based SNARK

10.1.1 Attempt #1: Encrypted Verification

Now, assume we have the cyclic group $\mathbb{G}$ of prime order q with a generator g. Typically, this is the group of points on an elliptic curve. Assume for simplicity that $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$ is a symmetric pairing function, where $(\mathbb{G}_T, \times)$ is a target group.

Now, suppose during the setup phase, we have a trusted party that generated a random value $\tau \xleftarrow{R} \mathbb{F}$ and public parameters $g^\tau, g^{\tau^2}, \ldots, g^{\tau^d}$ for $d = 2|\mathcal{C}|$ — maximum degree of used polynomials (later we will use notation $\{g^{\tau^i}\}_{i\in[d]}$ for brevity). Then, party **deleted** τ. This way, we can now find the KZG commitment for each polynomial. Indeed, for example,

$$\mathsf{com}(L) \triangleq g^{L(\tau)} = g^{\sum_{i=0}^{d} L_i \tau^i} = \prod_{i=0}^{d} (g^{\tau^i})^{L_i},$$

and the same goes for $g^{R(\tau)}, g^{O(\tau)}, g^{H(\tau)}, g^{Z(\tau)}$. Now, with these points, how can we verify that the polynomial $M(x) = L(x)R(x) - O(x)$ is correct? Well, first notice that the check is equivalent to

$$L(\tau)R(\tau) = Z(\tau)H(\tau) + O(\tau).$$

Notice that we transferred $O(\tau)$ to the right side of the equation to further avoid finding the inverse. Now, we can check this equality using encrypted values as follows:

$$e(\mathsf{com}(L), \mathsf{com}(R)) = e(\mathsf{com}(Z), \mathsf{com}(H)) \cdot e(\mathsf{com}(O), g),$$

Remark. One might ask: why is the above equation correct? Well, let us see:

$$\begin{aligned}
& e(\mathsf{com}(L), \mathsf{com}(R)) = e(\mathsf{com}(Z), \mathsf{com}(H)) \cdot e(\mathsf{com}(O), g) && \text{// Initial statement} \\
\Leftrightarrow\ & e(g^{L(\tau)}, g^{R(\tau)}) = e(g^{Z(\tau)}, g^{H(\tau)}) \cdot e(g^{O(\tau)}, g) && \text{// KZG commitment def.} \\
\Leftrightarrow\ & e(g, g)^{L(\tau)R(\tau)} = e(g, g)^{Z(\tau)H(\tau)} e(g, g)^{O(\tau)} && \text{// Pairing bilinearity} \\
\Leftrightarrow\ & e(g, g)^{L(\tau)R(\tau)} = e(g, g)^{Z(\tau)H(\tau)+O(\tau)} && \text{// Exponent product rule} \\
\Leftrightarrow\ & L(\tau)R(\tau) = Z(\tau)H(\tau) + O(\tau) && \text{// QAP Check} \\
\Leftrightarrow\ & L(x)R(x) \equiv Z(x)H(x) + O(x) && \text{// Schwarz-Zippel Lemma}
\end{aligned}$$

So, sounds like we are done. Let us summarize what we have done so far:

Attempt #1: Non-sound SNARK Protocol

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks a random value $\tau \xleftarrow{R} \mathbb{F}$.
- ✓ Calculates the public parameters $\{g^{\tau^i}\}_{i\in[d]}$.
- ✓ **Deletes** τ (toxic waste).
- ✓ **Outputs** prover parameters $\mathsf{pp} \leftarrow \{g^{\tau^i}\}_{i\in[d]}$ and verifier parameters $\mathsf{vp} \leftarrow \mathsf{com}(Z)$.

Prove($\mathsf{pp}, \mathbf{x}, \mathbf{w}$)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit with the statement $\mathbf{x}$ and witness $\mathbf{w}$, obtains the intermediate constraint values, and calculates the polynomials $L(x)$, $R(x)$, $O(x)$ through Lagrange Interpolation.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Calculates the KZG commitments as follows:

$$\pi_L \leftarrow \mathsf{com}(L), \pi_R \leftarrow \mathsf{com}(R), \pi_O \leftarrow \mathsf{com}(O), \pi_H \leftarrow \mathsf{com}(H),$$

  using powers of τ from the prover parameters pp.
- ✓ Publishes $\boldsymbol{\pi} = (\pi_L, \pi_R, \pi_O, \pi_H)$ as a proof.

Verify($\mathsf{vp}, \mathbf{x}, \boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi_R, \pi_O, \pi_H)$, using $\mathsf{com}(Z)$ from the verifier parameters vp, the verifier $\mathcal{V}$ checks:

$$e(\pi_L, \pi_R) = e(\mathsf{com}(Z), \pi_H) \cdot e(\pi_O, g).$$

This sounds like an end to the story. However, there is a problem with this approach: there is no guarantee that commitments π_L, π_R, π_O, π_H were indeed obtained through exponentiating the base g by the corresponding values $L(\tau)$, $R(\tau)$, $O(\tau)$, and $H(\tau)$. In other words, how can the verifier know that prover indeed knows, say, polynomial $L(x)$ such that $\pi_L = g^{L(\tau)}$?

10.1.2 Attempt #2: Including Proof of Exponent

In this section, we introduce the **Proof of Exponent assumption** (PoE) which makes KZG knowledge sound. Let us define it below.

Definition 10.1 (Proof of Exponent for KZG Commitment)**.** A **Proof of Exponent** (PoE) is a protocol that allows the prover $\mathcal{P}$ to convince the verifier $\mathcal{V}$ that he obtained a value $\mathsf{com}(f)$ through exponentiating a base g by $f(\tau)$. The protocol works as follows:

1. **Setup:** Proper parameters pp now contain not only $\{g^{\tau^i}\}_{i\in[d]}$, but also $\{g^{\alpha\tau^i}\}_{i\in[d]}$ for randomly selected $\tau, \alpha \xleftarrow{R} \mathbb{F}$ and further **deleted** values.
2. **Commit:** $\mathcal{P}$ commits to two values: $\mathsf{com}(f) = g^{f(\tau)}$ and $\mathsf{com}'(f) = g^{\alpha f(\tau)}$. The latter can be computed using pp as follows:
$$\mathsf{com}'(f) = \prod_{i=0}^{d} (g^{\alpha\tau^i})^{f_i}$$
3. **Verify:** $\mathcal{V}$ additionally checks $e(\mathsf{com}(f), g^{\alpha}) = e(\mathsf{com}'(f), g)$.

The informal reason why it makes KZG commitment sound is following: suppose we have an adversary prover $\mathcal{P}^*$ that published commitment c without knowing underlying polynomial $f(x)$. Now, there is no way for him to cheat the verifier. Indeed, what $\mathcal{P}^*$ needs to do is calculating c^{α}, but he does now know α since, similarly to τ, it was deleted as a part of the toxic waste. Besides, he cannot obtain α for the same reason he cannot obtain τ.

For that reason, we modify the SNARK protocol to include not only commitments to polynomials but also PoE for each of them. Let us see how it looks like.

Attempt #2: SNARK with PoE included

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks a random value $\tau, \alpha \xleftarrow{R} \mathbb{F}$.
- ✓ Calculates the public parameters $\{g^{\tau^i}\}_{i\in[d]}$, $\{g^{\alpha\tau^i}\}_{i\in[d]}$.
- ✓ **Deletes** τ, α (toxic waste).
- ✓ **Outputs** proper parameters $\mathsf{pp} \leftarrow \{\{g^{\tau^i}\}_{i\in[d]}, \{g^{\alpha\tau^i}\}_{i\in[d]}\}$, and verification parameters $\mathsf{vp} \leftarrow \{g^{Z(\tau)}, g^\alpha\}$.

Prove($\mathsf{pp}, \mathbf{x}, \mathbf{w}$)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit with the statement $\mathbf{x}$ and witness $\mathbf{w}$, obtains the intermediate constraint values, and calculates the polynomials $L(x), R(x), O(x)$ through Lagrange Interpolation.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Calculates the sound KZG commitments as follows:

$$\begin{aligned} \pi_L &\leftarrow g^{L(\tau)}, \quad & \pi'_L &\leftarrow g^{\alpha L(\tau)} \\ \pi_R &\leftarrow g^{R(\tau)}, \quad & \pi'_R &\leftarrow g^{\alpha R(\tau)} \\ \pi_O &\leftarrow g^{O(\tau)}, \quad & \pi'_O &\leftarrow g^{\alpha O(\tau)} \\ \pi_H &\leftarrow g^{H(\tau)}, \quad & \pi'_H &\leftarrow g^{\alpha H(\tau)}. \end{aligned}$$

  using powers $\{g^{\tau^i}\}_{i\in[d]}$ and $\{g^{\alpha\tau^i}\}_{i\in[d]}$ from the proper parameters pp.
- ✓ Publishes $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H)$ as a proof.

Verify($\mathsf{vp}, \mathbf{x}, \boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H)$, the verifier $\mathcal{V}$ checks:

$$\begin{aligned} e(\pi_L, \pi_R) &= e(g^{Z(\tau)}, \pi_H) \cdot e(\pi_O, g), && \text{// Polynomial Equality Test} \\ e(\pi_L, g^\alpha) &= e(\pi'_L, g), \quad e(\pi_R, g^\alpha) = e(\pi'_R, g), && \text{// Proof of Exponent} \\ e(\pi_O, g^\alpha) &= e(\pi'_O, g), \quad e(\pi_H, g^\alpha) = e(\pi'_H, g). && \text{// Proof of Exponent} \end{aligned}$$

The provided protocol is secure under the PoE assumption. However, it is still not fully sound. Currently, there is no guarantee that when evaluating π_L, π_R, π_O we used the same extended witness $\mathbf{w}$. In other words, the prover can still cheat by using different extended witnesses for each polynomial (faking the proof system is still hard in this situation, but we want to make sure to eliminate all possible weaknesses). Let us see how to fix this!

10.1.3 Attempt #3: Making SNARK Sound

Besides fixing the issue with consistent use of witness $\mathbf{w}$, we additionally include one more optimization we have not included so far.

Optimization. Left/Right/Output Polynomial Preprocessing. Recall that:

$$L(x) = \sum_{i=0}^{n} w_i L_i(x), \quad R(x) = \sum_{i=0}^{n} w_i R_i(x), \quad O(x) = \sum_{i=0}^{n} w_i O_i(x).$$

while $M(x) = L(x)R(x) - O(x)$ is only known to the prover $\mathcal{P}$ since it contains the extended witness ($\mathbf{w}$) coefficients. However, the set of polynomials

$$\{L_i(x)\}_{i\in[n]}, \{R_i(x)\}_{i\in[n]}, \{O_i(x)\}_{i\in[n]}$$

are known in advance. Meaning, we can precompute the values of $\{g^{L_i(\tau)}\}_{i\in[n]}$, $\{g^{\alpha L_i(\tau)}\}_{i\in[n]}$, $\{g^{R_i(\tau)}\}_{i\in[n]}$, $\{g^{\alpha R_i(\tau)}\}_{i\in[n]}$, $\{g^{O_i(\tau)}\}_{i\in[n]}$, $\{g^{\alpha O_i(\tau)}\}_{i\in[n]}$ and use them in the prover parameters pp.

How? Suppose the prover $\mathcal{P}$ knows the extended witness $\mathbf{w}$. Consider the polynomial $L(x) = \sum_{i=0}^{n} w_i L_i(x)$. $\mathcal{P}$ can compute the KZG commitment π_L and its PoE π'_L as follows:

$$\pi_L \triangleq g^{L(\tau)} = g^{\sum_{i=0}^{n} w_i L_i(\tau)} = \prod_{i=0}^{n} (g^{L_i(\tau)})^{w_i}, \quad \pi'_L \triangleq g^{\alpha L(\tau)} = g^{\alpha \sum_{i=0}^{n} w_i L_i(\tau)} = \prod_{i=0}^{n} (g^{\alpha L_i(\tau)})^{w_i}.$$

Fix. Witness consistency check.

Introducing new term with β. To prove that the same $\mathbf{w}$ is used in all commitments, we need some "checksum" term that will somehow combine all polynomials $L(x)$, $R(x)$, and $O(x)$ with the witness $\mathbf{w}$. Moreover, we will need to compare this term with proofs π_L, π_R, and π_O. The best candidate for this is the following group element for some other random $\beta \xleftarrow{R} \mathbb{F}$:

$$\pi_\beta = g^{L(\tau)+R(\tau)+O(\tau)} = \prod_{i=1}^{n} (g^{L_i(\tau)+R_i(\tau)+O_i(\tau)})^{w_i}, \ \pi'_\beta = \prod_{i=1}^{n} (g^{\beta(L_i(\tau)+R_i(\tau)+O_i(\tau))})^{w_i}$$

This way, we get a term that includes all three polynomials $L(x)$, $R(x)$, and $O(x)$, and all coefficients of the extended witness $\mathbf{w}$. Moreover, verifier $\mathcal{V}$ can compare π_β with three other commitments π_L, π_R, π_O to ensure that all of them are consistent. This is done through the following check:

$$e(\pi_L \pi_R \pi_O, g^\beta) = e(\pi'_\beta, g).$$

Again, this approach still has a weakness (yeah-yeah, I am also tired of this). Indeed, this check is complete (meaning, if $\mathbf{w}$ is used consistently across π_L, π_R, π_O, then the check will pass), but it is still not sound with an overwhelming probability. Indeed, suppose $\mathbf{w}$ is used inconsistently, meaning, we have extended witnesses $\mathbf{w}_L$, $\mathbf{w}_R$, $\mathbf{w}_O$, and $\mathbf{w}_\beta$, each for corresponding polynomials. If the witness is consistent, the following condition must hold:

$$(w_{L,i} L_i(\tau) + w_{R,i} R_i(\tau) + w_{O,i} O_i(\tau))\beta = w_{\beta,i}\beta(L_i(\tau) + R_i(\tau) + O_i(\tau)) \quad \forall i \in [n]$$

Assume otherwise. Consider a simple situation where it happens that $L_i \equiv R_i$, meaning $L_i(\tau)$ and $R_i(\tau)$ are the same (call them q). Then,

$$(w_{L,i} + w_{R,i})q + w_{O,i} O_i(\tau) = w_{\beta,i}(2q + O_i(\tau)) \quad \forall i \in [n]$$

For arbitrarily chosen $w_{R,i}$ and $w_{O,i}$, the adversary prover $\mathcal{P}^*$ can set $w_{\beta,i} := w_{O,i}$ and $w_{L,i} = 2w_{O,i} - w_{R,i}$. It is easy to verify that the above equation would hold, meaning that $\mathbf{w}$ is not the same across all polynomials.

One might ask: well, situation when $L_i \equiv R_i$ is very rare! Indeed, but there also might be situations where $L_i \equiv 5R_i$, $R_i \equiv 100O_i$, or $L_i \equiv 235O_i$ — all of them would lead to the same issue. So what is the solution?

Introducing separate β_L, β_R, β_O. Our proposal is to make β different for each polynomial $L(x)$, $R(x)$, $O(x)$, making it much harder for the adversary to find inconsistent witnesses. That being said, during the setup phase, we choose arbitrary $\beta_L, \beta_R, \beta_O \xleftarrow{R} \mathbb{F}$ and publish $\{g^{\beta_L L_i(\tau)+\beta_R R_i(\tau)+\gamma_O O_i(\tau)}\}_{i\in[n]}$ as a part of the prover parameters pp. Then, the prover $\mathcal{P}$ calculates the following additional commitment:

$$\pi_\beta \leftarrow \prod_{i=1}^{n} (g^{\beta_L L_i(\tau)+\beta_R R_i(\tau)+\beta_O O_i(\tau)})^{w_i}$$

and then publishes π_β as a part of the proof. The verifier $\mathcal{V}$ checks the following condition:

$$e(\pi_L, g^{\beta_L}) \cdot e(\pi_R, g^{\beta_R}) \cdot e(\pi_O, g^{\beta_O}) = e(\pi_\beta, g).$$

Even that is not the end of the story! We also need to make sure that g^{β_L}, g^{β_R}, and g^{β_O} are incompatible with $g^{\sum_{i=0}^{n}(\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau))w_i}$ (for reasons we drop here due to already long explanation). For that reason, we multiply β_L, β_R, β_O by some random factor $\gamma \xleftarrow{R} \mathbb{F}$. This way, the proving part remains the same, but the check becomes:

$$e(\pi_L, g^{\gamma\beta_L}) \cdot e(\pi_R, g^{\gamma\beta_R}) \cdot e(\pi_O, g^{\gamma\beta_O}) = e(\pi_\beta, g^{\gamma}).$$

Oof, that was a long fix! Let us come back to the new version of the SNARK protocol (not yet zero-knowledge)!

Attempt #3: Sound SNARK Protocol

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks random values $\tau, \alpha, \beta_L, \beta_R, \beta_O, \gamma \xleftarrow{R} \mathbb{F}$.
- ✓ **Outputs** prover parameters pp and verification parameters vp:

$$\begin{aligned}\mathsf{pp} \leftarrow \Big\{ &\{g^{\tau^i}\}_{i\in[d]}, \{g^{L_i(\tau)}, g^{\alpha L_i(\tau)}, g^{R_i(\tau)}, g^{\alpha R_i(\tau)}, \\ &g^{O_i(\tau)}, g^{\alpha O_i(\tau)}, g^{\beta_L L_i(\tau)+\beta_R R_i(\tau)+\beta_O O_i(\tau)}\}_{i\in[n]}\Big\} \\ \mathsf{vp} \leftarrow \Big\{ &g^{Z(\tau)}, g^{\alpha}, g^{\beta_L}, g^{\beta_R}, g^{\beta_O}, g^{\beta_L\gamma}, g^{\beta_R\gamma}, g^{\beta_O\gamma}, g^{\gamma}\Big\}\end{aligned}$$

- ✓ **Deletes** aforementioned random scalars (toxic waste).

Prove(pp, **x**, **w**)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit to get **w** and $L(x), R(x), O(x)$.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Calculates the sound KZG commitments as follows:

$$\begin{aligned}\pi_L &\leftarrow g^{L(\tau)}, \quad & \pi'_L &\leftarrow g^{\alpha L(\tau)}, \\ \pi_R &\leftarrow g^{R(\tau)}, \quad & \pi'_R &\leftarrow g^{\alpha R(\tau)}, \\ \pi_O &\leftarrow g^{O(\tau)}, \quad & \pi'_O &\leftarrow g^{\alpha O(\tau)}, \\ \pi_H &\leftarrow g^{H(\tau)}, \quad & \pi'_H &\leftarrow g^{\alpha H(\tau)}.\end{aligned}$$

- ✓ Calculates the additional commitment π_β as follows:

$$\pi_\beta \leftarrow g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}$$

- ✓ Publishes $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$ as a proof.

Verify(vp, **x**, $\boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$, the verifier $\mathcal{V}$ checks:

$$\begin{aligned} e(\pi_L, \pi_R) &= e(g^{Z(\tau)}, \pi_H) \cdot e(\pi_O, g), && \textit{// Polynomial Equality Test} \\ e(\pi_L, g^\alpha) &= e(\pi'_L, g), \quad e(\pi_R, g^\alpha) = e(\pi'_R, g), && \textit{// Proof of Exponent} \\ e(\pi_O, g^\alpha) &= e(\pi'_O, g), \quad e(\pi_H, g^\alpha) = e(\pi'_H, g), && \textit{// Proof of Exponent} \\ e(\pi_L, g^{\gamma\beta_L}) \cdot e(\pi_R, g^{\gamma\beta_R}) &\cdot e(\pi_O, g^{\gamma\beta_O}) = e(\pi_\beta, g^\gamma). && \textit{// Extended Witness Consistency} \end{aligned}$$

10.1.4 Attempt #4: Splitting the Extended Witness

Now, recall that the actual circuit $\mathcal{C}$ is defined for some statement $\mathbf{x}$ and witness $\mathbf{w}$. According to the Circuit Satisfability Problem, the prover $\mathcal{P}$ wants to convince the verifier $\mathcal{V}$ that he knows the witness $\mathbf{w}$ such that the circuit $\mathcal{C}(\mathbf{x}, \mathbf{w}) = 0$. Up until now, we have been using the extended witness $\widetilde{\mathbf{w}}$ to represent the trace of computation. However, we can split the witness into two parts: the first part $\mathbf{w}_{\text{mid}}$ — intermediate witness that contains the private witness $R\mathbf{w}$ and intermediate variables values, and the second part $\mathbf{w}_{\text{io}}$ — input/output witness that contains public statement information (e.g., $\mathbf{x}$). Suppose for vector $\widetilde{\mathbf{w}} = (\widetilde{w}_1, \ldots, \widetilde{w}_n)$ we pick out the set of indeces $\mathcal{I}_{\text{mid}} \subset [n]$ to represent the intermediate witness and $\mathcal{I}_{\text{io}} \subset [n]$ to represent the input/output witness (of course, $\mathcal{I}_{\text{mid}} \cap \mathcal{I}_{\text{io}} = \emptyset$ and $\mathcal{I}_{\text{mid}} \cup \mathcal{I}_{\text{io}} = [n]$).

Now, how do we split the proofs $\pi_L, \pi_R, \ldots$ into two parts? Well, consider for instance π_L:

$$\pi_L = g^{\sum_{i=0}^{n} w_i L_i(\tau)}.$$

We split this expression as follows:

$$\pi_L = \underbrace{g^{\sum_{i \in \mathcal{I}_{\text{mid}}} w_i L_i(\tau)}}_{\text{new } \pi_L} \times \underbrace{g^{\sum_{i \in \mathcal{I}_{\text{io}}} w_i L_i(\tau)}}_{\pi_{L,\text{io}}}.$$

This way, the prover $\mathcal{P}$ first calculates the new commitment π_L using only the intermediate witness $\mathbf{w}_{\text{mid}}$ and then the verifier can compute the "public" portion of the proof $\pi_{L,\text{io}}$ using the input/output witness $\mathbf{w}_{\text{io}}$ (which is typically quite easy to do since $|\mathcal{I}_{\text{io}}|$ is typically much smaller than $|\mathcal{I}_{\text{mid}}|$). The same goes for other parts of the proof $\boldsymbol{\pi}$.

Now, let us formulate the updated SNARK protocol with the extended witness split.

Attempt #4: Sound SNARK for Public/Private Inputs

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$. Additionally, we know that $\mathcal{I}_{\text{io}}$ corresponds to public signals, while $\mathcal{I}_{\text{mid}}$ corresponds to private signals.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks random values $\tau, \alpha, \beta_L, \beta_R, \beta_O, \gamma \xleftarrow{R} \mathbb{F}$.
- ✓ **Outputs** prover parameters pp and verification parameters vp:

$$\begin{aligned}\mathsf{pp} \leftarrow \Big\{ &\{g^{\tau^i}\}_{i\in[d]}, \{g^{L_i(\tau)}, g^{\alpha L_i(\tau)}, g^{R_i(\tau)}, g^{\alpha R_i(\tau)}, \\ &g^{O_i(\tau)}, g^{\alpha O_i(\tau)}, g^{\beta_L L_i(\tau)+\beta_R R_i(\tau)+\beta_O O_i(\tau)}\}_{i\in\mathcal{I}_{\text{mid}}}\Big\} \\ \mathsf{vp} \leftarrow \Big\{ &g^{Z(\tau)}, g^{\alpha}, g^{\beta_L}, g^{\beta_R}, g^{\beta_O}, g^{\beta_L\gamma}, g^{\beta_R\gamma}, g^{\beta_O\gamma}, g^{\gamma}, \{g^{L_i(\tau)}, g^{R_i(\tau)}, g^{O_i(\tau)}\}_{i\in\mathcal{I}_{\text{io}}}\Big\}\end{aligned}$$

- ✓ **Deletes** aforementioned random scalars (toxic waste).

Prove(pp, **x**, **w**)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit to get **w** and $L(x), R(x), O(x)$.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Splits $L(x) = L_{\text{mid}}(x) + L_{\text{io}}(x)$ — intermediate and input/output parts. That being said, $L_{\text{mid}}(x) = \sum_{i\in\mathcal{I}_{\text{mid}}} L_i(x)$. Repeat for $R(x)$ and $O(x)$.
- ✓ Calculates the following values:

$$\begin{aligned}\pi_L &\leftarrow g^{L_{\text{mid}}(\tau)}, \quad \pi'_L \leftarrow g^{\alpha L_{\text{mid}}(\tau)}, \\ \pi_R &\leftarrow g^{R_{\text{mid}}(\tau)}, \quad \pi'_R \leftarrow g^{\alpha R_{\text{mid}}(\tau)}, \\ \pi_O &\leftarrow g^{O_{\text{mid}}(\tau)}, \quad \pi'_O \leftarrow g^{\alpha O_{\text{mid}}(\tau)}, \\ \pi_H &\leftarrow g^{H(\tau)}, \quad \pi'_H \leftarrow g^{\alpha H(\tau)}, \\ \pi_\beta &\leftarrow g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}.\end{aligned}$$

- ✓ Publishes $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$ as a proof.

Verify(vp, **x**, $\boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$, the verifier $\mathcal{V}$:

- ✓ Finds $\pi^*_L \leftarrow \pi_L \prod_{i\in\mathcal{I}_{\text{io}}} (g^{L_i(\tau)})^{w_i}$, $\pi^*_R \leftarrow \pi_R \prod_{i\in\mathcal{I}_{\text{io}}} (g^{R_i(\tau)})^{w_i}$, $\pi^*_O \leftarrow \pi_O \prod_{i\in\mathcal{I}_{\text{io}}} (g^{O_i(\tau)})^{w_i}$
- ✓ Checks whether all of the following conditions hold:

$$\begin{aligned}e(\pi^*_L, \pi^*_R) &= e(g^{Z(\tau)}, \pi_H) \cdot e(\pi^*_O, g), && \text{// Polynomial Equality Test} \\ e(\pi_L, g^\alpha) &= e(\pi'_L, g), \quad e(\pi_R, g^\alpha) = e(\pi'_R, g), && \text{// Proof of Exponent} \\ e(\pi_O, g^\alpha) &= e(\pi'_O, g), \quad e(\pi_H, g^\alpha) = e(\pi'_H, g), && \text{// Proof of Exponent} \\ e(\pi_L, g^{\gamma\beta_L}) \cdot e(\pi_R, g^{\gamma\beta_R}) &\cdot e(\pi_O, g^{\gamma\beta_O}) = e(\pi_\beta, g^\gamma). && \text{// Extended Witness Consistency}\end{aligned}$$

10.1.5 Attempt #5: Making SNARK Zero-Knowledge

Finally, we came to the point where we need to make the protocol zero-knowledge. As it turns out, it is not that hard to do (compared to what we have done so far)!

Remark. Currently, you might have a reasonable question: the proof contains quantities such as $g^{L(\tau)}$, $g^{R(\tau)}$, $g^{O(\tau)}$ and variations of them. As long as discrete logarithm holds, there is no PPT adversary that would be able to extract coefficients of $L(x)$, $R(x)$, $O(x)$ from published KZG commitments (and respective PoE shifts and witness consistency proof). So what could be the issue?
This reasoning is correct. In other words, the adversary will not learn coefficients of polynomials and therefore will not be able to get the witness fully. However, the zero-knowledge property ensures that the adversary cannot draw conclusions based on proof $\boldsymbol{\pi}$. However, in our current version, we do have SNARK, but not zk-SNARK. For instance, currently anyone can check whether $L \equiv R$ (by checking $\pi_L = \pi_R$) or that $L \equiv 11R$ (by checking $\pi_L = \pi_R^{11}$).

The main idea to make our protocol zero-knowledge is to "shift" our commitments by some random factor δ. Of course, this "shift" δ must be different for each commitment and must be chosen by the prover $\mathcal{P}$. This way, we propose to modify the commitments as follows:

$$\pi_L = g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i L_i(\tau)+\delta_L Z(\tau)} = (g^{Z(\tau)})^{\delta_L} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{L_i(\tau)})^{w_i}, \qquad \pi'_L = (g^{\alpha Z(\tau)})^{\delta_L} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{\alpha L_i(\tau)})^{w_i}$$

$$\pi_R = g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i R_i(\tau)+\delta_R Z(\tau)} = (g^{Z(\tau)})^{\delta_R} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{R_i(\tau)})^{w_i}, \qquad \pi'_R = (g^{\alpha Z(\tau)})^{\delta_R} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{\alpha R_i(\tau)})^{w_i}$$

$$\pi_O = g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i O_i(\tau)+\delta_O Z(\tau)} = (g^{Z(\tau)})^{\delta_O} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{O_i(\tau)})^{w_i}, \qquad \pi'_O = (g^{\alpha Z(\tau)})^{\delta_O} \prod_{i\in\mathcal{I}_{\text{mid}}} (g^{\alpha O_i(\tau)})^{w_i}$$

for randomly selected $\delta_L, \delta_R, \delta_O \xleftarrow{R} \mathbb{Z}_q$. Good, we concealed all the information about the witness $\mathbf{w}$ in the commitment. However, we need to make sure that the verifier $\mathcal{V}$ can still verify the proof, but without modifying the verification mechanism itself. Notice that PoE verifications are still valid, but we need to modify something to make polynomial equality test hold. This can be done by perturbating the polynomial $H(x)$ by some (currently) unknown value Δ_H. Let us derive it from the polynomial equality test:

$$(L(x)+\delta_L Z(x))(R(x)+\delta_R Z(x)) = (H(x)+\Delta_H)Z(x) + (O(x)+\delta_O Z(x)),$$

which, by expanding, gives us the following equation:

$$\cancel{L(x)R(x)} + \delta_R L(x)Z(x) + \delta_L Z(x)R(x) + \delta_L\delta_R Z(x)^2 = \cancel{H(x)Z(x)+O(x)} + \Delta_H Z(x) + \delta_O Z(x),$$

where we can cancel out $L(x)R(x)$ and $H(x)Z(x)+O(x)$ terms since they are equal based on initial construction. This way, we get the following expression for Δ_H:

$$\Delta_H = \delta_O + \delta_R L(x) + \delta_L R(x) + \delta_L\delta_R Z(x)$$

Therefore, our fourth proof system becomes:

$$\pi_H = g^{H(\tau)+\delta_O+\delta_R L(\tau)+\delta_L R(\tau)+\delta_L\delta_R Z(\tau)}, \qquad \pi'_H = g^{\alpha(H(\tau)+\delta_O+\delta_R L(\tau)+\delta_L R(\tau)+\delta_L\delta_R Z(\tau))}.$$

Finally, we need to make sure that our witness consistency proof π_β is still valid. Since previously we had $\pi_\beta = g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}$, we need to modify it to include the new δ values. Namely, we change $L(\tau)$ to $L(\tau)+\delta_L Z(\tau)$, $R(\tau)$ to $R(\tau)+\delta_R Z(\tau)$, and $O(\tau)$ to $O(\tau)+\delta_O Z(\tau)$. This way, our new π_β becomes:

This can be done by changing it into:

$$\begin{aligned}\pi_\beta &= g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)+(\delta_L\beta_L+\delta_R\beta_R+\delta_O\beta_O)Z(\tau)} \\ &= \left(g^{\beta_L Z(\tau)}\right)^{\delta_L}\left(g^{\beta_R Z(\tau)}\right)^{\delta_R}\left(g^{\beta_O Z(\tau)}\right)^{\delta_O} g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}.\end{aligned}$$

Oof. Finally, we also have a zero-knowledge property for our SNARK. Let us summarize what has changed for the prover $\mathcal{P}$.

Proposition 10.2 (Including ZK in general-purpose SNARK)**.** Now, the prover $\mathcal{P}$ samples random scalars $\delta_L, \delta_R, \delta_O \xleftarrow{R} \mathbb{Z}_q$ and calculates the following commitments:

- **Updated Commitments for L, R, O:** Now, the prover $\mathcal{P}$ needs to calculate the commitments π_L, π_R, π_O with the additional δ's values to ensure zero-knowledge property:

$$\begin{aligned}\pi_L &\leftarrow g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i L_i(\tau)+\delta_L Z(\tau)}, \quad & \pi'_L &\leftarrow g^{\alpha\sum_{i\in\mathcal{I}_{\text{mid}}} w_i L_i(\tau)+\delta_L Z(\tau)}, \\ \pi_R &\leftarrow g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i R_i(\tau)+\delta_R Z(\tau)}, \quad & \pi'_R &\leftarrow g^{\alpha\sum_{i\in\mathcal{I}_{\text{mid}}} w_i R_i(\tau)+\delta_R Z(\tau)}, \\ \pi_O &\leftarrow g^{\sum_{i\in\mathcal{I}_{\text{mid}}} w_i O_i(\tau)+\delta_O Z(\tau)}, \quad & \pi'_O &\leftarrow g^{\alpha\sum_{i\in\mathcal{I}_{\text{mid}}} w_i O_i(\tau)+\delta_O Z(\tau)}.\end{aligned}$$

- **Updated Commitment for H:** The prover $\mathcal{P}$ needs to calculate the commitment π_H with the additional δ's values to ensure polynomial equality test is satisfied:

$$\pi_H \leftarrow g^{H(\tau)+\delta_O+\delta_R L(\tau)+\delta_L R(\tau)+\delta_L\delta_R Z(\tau)}, \quad \pi'_H \leftarrow g^{\alpha(H(\tau)+\delta_O+\delta_R L(\tau)+\delta_L R(\tau)+\delta_L\delta_R Z(\tau))}.$$

- **Updated Witness Consistency Proof:** The prover $\mathcal{P}$ needs to calculate the commitment π_β with the additional δ's values to ensure witness consistency:

$$\pi_\beta \leftarrow \left(g^{\beta_L Z(\tau)}\right)^{\delta_L}\left(g^{\beta_R Z(\tau)}\right)^{\delta_R}\left(g^{\beta_O Z(\tau)}\right)^{\delta_O} g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}.$$

Let us now look at the final version of our basic SNARK protocol.

Attempt #5: Turning SNARK into zk-SNARK

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$. Additionally, we know that $\mathcal{I}_{\text{io}}$ corresponds to public signals, while $\mathcal{I}_{\text{mid}}$ corresponds to private signals.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks random values $\tau, \alpha, \beta_L, \beta_R, \beta_O, \gamma \xleftarrow{R} \mathbb{F}$.
- ✓ **Outputs** prover parameters pp and verification parameters vp:

$$\begin{aligned}\mathsf{pp} \leftarrow \Big\{ &\{g^{\tau^i}\}_{i\in[d]}, \{g^{Z(\tau)}, g^{L_i(\tau)}, g^{\alpha L_i(\tau)}, g^{R_i(\tau)}, g^{\alpha R_i(\tau)}, \\ &g^{O_i(\tau)}, g^{\alpha O_i(\tau)}, g^{\beta_L L_i(\tau)}, g^{\beta_R R_i(\tau)}, g^{\beta_O O_i(\tau)}\}_{i\in\mathcal{I}_{\text{mid}}}\Big\} \\ \mathsf{vp} \leftarrow \Big\{ &g^{Z(\tau)}, g^{\alpha}, g^{\beta_L}, g^{\beta_R}, g^{\beta_O}, g^{\beta_L\gamma}, g^{\beta_R\gamma}, g^{\beta_O\gamma}, g^{\gamma}, \{g^{L_i(\tau)}, g^{R_i(\tau)}, g^{O_i(\tau)}\}_{i\in\mathcal{I}_{\text{io}}}\Big\}\end{aligned}$$

- ✓ **Deletes** aforementioned random scalars (toxic waste).

Prove(pp, $\mathbf{x}$, $\mathbf{w}$)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit to get $\mathbf{w}$ and $L(x), R(x), O(x)$.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Splits $L(x) = L_{\text{mid}}(x) + L_{\text{io}}(x)$ — intermediate and input/output parts. That being said, $L_{\text{mid}}(x) = \sum_{i\in\mathcal{I}_{\text{mid}}} L_i(x)$. Repeat for $R(x)$ and $O(x)$.
- ✓ Samples $\delta_L, \delta_R, \delta_O \xleftarrow{R} \mathbb{F}$ and calculates the following values:

$$\begin{aligned}\pi_L &\leftarrow g^{L_{\text{mid}}(\tau)}\left(g^{Z(\tau)}\right)^{\delta_L}, \quad \pi'_L \leftarrow g^{\alpha L_{\text{mid}}(\tau)}\left(g^{\alpha Z(\tau)}\right)^{\delta_L}, \\ \pi_R &\leftarrow g^{R_{\text{mid}}(\tau)}\left(g^{Z(\tau)}\right)^{\delta_R}, \quad \pi'_R \leftarrow g^{\alpha R_{\text{mid}}(\tau)}\left(g^{Z(\tau)}\right)^{\delta_R}, \\ \pi_O &\leftarrow g^{O_{\text{mid}}(\tau)}\left(g^{Z(\tau)}\right)^{\delta_O}, \quad \pi'_O \leftarrow g^{\alpha O_{\text{mid}}(\tau)}\left(g^{Z(\tau)}\right)^{\delta_O}, \\ \pi_H &\leftarrow g^{H(\tau)}(g^{\delta_O})(g^{R(\tau)})^{\delta_L}(g^{L(\tau)})^{\delta_R}(g^{Z(\tau)})^{\delta_L\delta_R}, \\ \pi'_H &\leftarrow g^{\alpha H(\tau)}(g^{\delta_O})(g^{R(\tau)})^{\delta_L}(g^{L(\tau)})^{\delta_R}(g^{Z(\tau)})^{\delta_L\delta_R}, \\ \pi_\beta &\leftarrow \left(g^{\beta_L Z(\tau)}\right)^{\delta_L}\left(g^{\beta_R Z(\tau)}\right)^{\delta_R}\left(g^{\beta_O Z(\tau)}\right)^{\delta_O} g^{\beta_L L(\tau)+\beta_R R(\tau)+\beta_O O(\tau)}.\end{aligned}$$

- ✓ Publishes $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$ as a proof.

Verify(vp, $\mathbf{x}$, $\boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi'_H, \pi_\beta)$, the verifier $\mathcal{V}$:

- ✓ Finds $\pi^*_L \leftarrow \pi_L \prod_{i\in\mathcal{I}_{\text{io}}}(g^{L_i(\tau)})^{w_i}$, $\pi^*_R \leftarrow \pi_R \prod_{i\in\mathcal{I}_{\text{io}}}(g^{R_i(\tau)})^{w_i}$, $\pi^*_O \leftarrow \pi_O \prod_{i\in\mathcal{I}_{\text{io}}}(g^{O_i(\tau)})^{w_i}$
- ✓ Checks whether all of the following conditions hold:

$$\begin{aligned} e(\pi^*_L, \pi^*_R) &= e(g^{Z(\tau)}, \pi_H)\cdot e(\pi^*_O, g), && \text{// Polynomial Equality Test} \\ e(\pi_L, g^\alpha) &= e(\pi'_L, g), \quad e(\pi_R, g^\alpha) = e(\pi'_R, g), && \text{// Proof of Exponent} \\ e(\pi_O, g^\alpha) &= e(\pi'_O, g), \quad e(\pi_H, g^\alpha) = e(\pi'_H, g), && \text{// Proof of Exponent} \\ e(\pi_L, g^{\gamma\beta_L})\cdot e(\pi_R, g^{\gamma\beta_R})\cdot e(\pi_O, g^{\gamma\beta_O}) &= e(\pi_\beta, g^\gamma). && \text{// Extended Witness Consistency} \end{aligned}$$

10.2 Real Protocols

While our built protocol is zk-SNARK, it is still not optimized for practical use. In fact, let us recap what is complexity of our protocol.

Proposition 10.3 (Complexity of the Basic Protocol)**.** Suppose circuit consists of n gates. Then, the complexity of the basic protocol is as follows:

- **Proof Size:** $O(1)$ — we have a constant number of group elements.
- **Setup Time:** $O(n)$ — we need to calculate powers of τ and evaluations at τ.
- **Prover Time:** $O(n \log n)$ — using FFT and wise choice of Ω.
- **Verifier Time:** $O(1)$ — we have a constant number of pairings to evaluate.

However, $O(1)$ is not very descriptive for proof and verifier complexities, so let us provide a more detailed analysis.

- **Proof Size:** 9 $\mathbb{G}$ group elements.
- **Verifier Time:** 15 pairings and $O(|\mathcal{I}_{\text{io}}|)$ group multiplications.

Now, this is not bad at all! In fact, this is already practical for many applications. However, we can do better by a more clever choice of constants and terms. This is exactly what is done by Bryan Parno and Craig Gentry in their research "Pinocchio: Nearly Practical Verifiable Computation".

10.2.1 Pinocchio Protocol

We first begin from the non-zero-knowledge version of the Pinocchio Protocol and then extend it to the zero-knowledge version.

Setup Procedure. The Pinocchio Protocol exploits the idea that we might choose different generators for $L(x)$, $R(x)$, and $O(x)$. Namely, the protocol begins from choosing random $\rho_L, \rho_R \xleftarrow{R} \mathbb{F}$. Then, we define $\rho_O \triangleq \rho_L \rho_R$, and finally introduce the following generators:

$$g_L \triangleq g^{\rho_L}, \quad g_R \triangleq g^{\rho_R}, \quad g_O \triangleq g^{\rho_O} = g^{\rho_L \rho_R}$$

The next change is that we use different α values for $L(x)$, $R(x)$, and $O(x)$. Namely, we define $\alpha_L, \alpha_R, \alpha_O \xleftarrow{R} \mathbb{F}$ and use them in the commitments. This way, in the setup phase, we additionally prepare the following values:

$$\{g_L^{L_i(\tau)}, g_L^{\alpha_L L_i(\tau)}, g_R^{R_i(\tau)}, g_R^{\alpha_R R_i(\tau)}, g_O^{O_i(\tau)}, g_O^{\alpha_O O_i(\tau)}\}_{i \in \mathcal{I}_{\text{mid}}}$$

Instead of using three different β's, the Pinocchio Protocol uses a single $\beta \xleftarrow{R} \mathbb{F}$:

$$\{g_L^{\beta L_i(\tau)}, g_R^{\beta R_i(\tau)}, g_O^{\beta O_i(\tau)}\}_{i \in \mathcal{I}_{\text{mid}}}$$

And finally, since we also have a $\gamma \xleftarrow{R} \mathbb{F}$ for the extended witness consistency and some other missing quantities, we also prepare the following values:

$$g_O^{Z(\tau)}, g^{\alpha_L}, g^{\alpha_R}, g^{\alpha_O}, g^{\beta\gamma}, g^{\gamma}$$

Proving. Again, suppose intermediate polynomials for indices $\mathcal{I}_{\text{mid}}$ are $L_{\text{mid}}(x) = \sum_{i \in \mathcal{I}_{\text{mid}}} w_i L_i(x)$ and similarly for $R_{\text{mid}}(x)$ and $O_{\text{mid}}(x)$. The prover $\mathcal{P}$ calculates the following commitments:

$$\pi_L \leftarrow g_L^{L_{\text{mid}}(\tau)}, \quad \pi'_L \leftarrow g_L^{\alpha_L L_{\text{mid}}(\tau)},$$
$$\pi_R \leftarrow g_R^{R_{\text{mid}}(\tau)}, \quad \pi'_R \leftarrow g_R^{\alpha_R R_{\text{mid}}(\tau)},$$
$$\pi_O \leftarrow g_O^{O_{\text{mid}}(\tau)}, \quad \pi'_O \leftarrow g_O^{\alpha_O O_{\text{mid}}(\tau)},$$
$$\pi_H \leftarrow g^{H(\tau)}, \quad \pi_\beta \leftarrow g_L^{\beta L_{\text{mid}}(\tau)} g_R^{\beta R_{\text{mid}}(\tau)} g_O^{\beta O_{\text{mid}}(\tau)},$$

and then uses these commitments in the proof $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi_\beta)$.

Verification. First, the verifier checks the PoE properties:

$$e(\pi_L, g_L^{\alpha_L}) = e(\pi'_L, g_L), \quad e(\pi_R, g_R^{\alpha_R}) = e(\pi'_R, g_R), \quad e(\pi_O, g_O^{\alpha_O}) = e(\pi'_O, g_O).$$

Next, we check the extended witness consistency:

$$e(\pi_L \pi_R \pi_O, g^{\beta\gamma}) = e(\pi_\beta, g^\gamma)$$

The verifier restores the full proof $\pi_L^*, \pi_R^*, \pi_O^*$ by multiplying the public part of the proof with the corresponding private part. This is done as follows:

$$\pi_L^* \leftarrow \pi_L \prod_{i \in \mathcal{I}_{\text{io}}} (g_L^{L_i(\tau)})^{w_i}, \quad \pi_R^* \leftarrow \pi_R \prod_{i \in \mathcal{I}_{\text{io}}} (g_R^{R_i(\tau)})^{w_i}, \quad \pi_O^* \leftarrow \pi_O \prod_{i \in \mathcal{I}_{\text{io}}} (g_O^{O_i(\tau)})^{w_i}.$$

And finally, we check the polynomial equality test:

$$e(\pi_L^*, \pi_R^*) = e(g_O^{Z(\tau)}, \pi_H) \cdot e(\pi_O^*, g)$$

Zero-Knowledge Extension. The zero-knowledge extension of the Pinocchio Protocol is done exactly in the same way as we did for the basic SNARK protocol. Namely, we introduce random scalars $\delta_L, \delta_R, \delta_O \xleftarrow{R} \mathbb{F}$ and calculate the following commitments:

$$\pi_L \leftarrow g_L^{L_{\text{mid}}(\tau)} \left(g_L^{Z(\tau)}\right)^{\delta_L}, \quad \pi'_L \leftarrow g_L^{\alpha_L L_{\text{mid}}(\tau)} \left(g_L^{\alpha_L Z(\tau)}\right)^{\delta_L},$$
$$\pi_R \leftarrow g_R^{R_{\text{mid}}(\tau)} \left(g_R^{Z(\tau)}\right)^{\delta_R}, \quad \pi'_R \leftarrow g_R^{\alpha_R R_{\text{mid}}(\tau)} \left(g_R^{\alpha_R Z(\tau)}\right)^{\delta_R},$$
$$\pi_O \leftarrow g_O^{O_{\text{mid}}(\tau)} \left(g_O^{Z(\tau)}\right)^{\delta_O}, \quad \pi'_O \leftarrow g_O^{\alpha_O O_{\text{mid}}(\tau)} \left(g_O^{\alpha_O Z(\tau)}\right)^{\delta_O},$$
$$\pi_H \leftarrow g^{H(\tau)} (g^{\delta_O})(g^{\delta_L})(g^{\delta_R})(g^{\delta_L \delta_R}),$$
$$\pi_\beta \leftarrow \left(g_L^{\beta Z(\tau)}\right)^{\delta_L} \left(g_R^{\beta Z(\tau)}\right)^{\delta_R} \left(g_O^{\beta Z(\tau)}\right)^{\delta_O} g_L^{\beta L_{\text{mid}}(\tau)} g_R^{\beta R_{\text{mid}}(\tau)} g_O^{\beta O_{\text{mid}}(\tau)}$$

Let us see what has changed in the Pinocchio Protocol.

Proposition 10.4. Pinocchio Protocol Proof size and verifier time complexity, compared to the basic SNARK protocol, are as follows:

- **Proof Size: 8** $\mathbb{G}$ group elements.
- **Verifier Time: 11** pairings and $O(|\mathcal{I}_{\text{io}}|)$ group multiplications (**4 less!**)

Let us see the complete specification of the Pinocchio Protocol.

Pinocchio Protocol

Suppose we are given a circuit $\mathcal{C}$ with a maximum degree d of polynomials used underneath. Thus, all parties additionally know the target polynomial $Z(x)$. Additionally, we know that $\mathcal{I}_{\text{io}}$ corresponds to public signals, while $\mathcal{I}_{\text{mid}}$ corresponds to private signals.

Setup(1^λ)

The *trusted party* conducts the following steps:

- ✓ Picks random values $\tau, \alpha_L, \alpha_R, \alpha_O, \beta, \gamma, \rho_L, \rho_R \xleftarrow{R} \mathbb{F}$.
- ✓ Calculates $\rho_O \leftarrow \rho_L \rho_R$ and defines generators $g_L \leftarrow g^{\rho_L}, g_R \leftarrow g^{\rho_R}, g_O \leftarrow g^{\rho_O}$.
- ✓ **Outputs** prover parameters pp and verification parameters vp:

$$\begin{aligned}
\mathsf{pp} \leftarrow \Big\{ &\{g^{\tau^i}\}_{i\in[d]}, \{g_L^{L_i(\tau)}, g_L^{\alpha_L L_i(\tau)}, g_R^{R_i(\tau)}, g_R^{\alpha_R R_i(\tau)}, g_O^{O_i(\tau)}, g_O^{\alpha_O O_i(\tau)}, \\
&g_L^{\beta L_i(\tau)}, g_R^{\beta R_i(\tau)}, g_O^{\beta O_i(\tau)}\}_{i\in\mathcal{I}_{\text{mid}}}, \\
&g_L^{\alpha_L Z(\tau)}, g_L^{Z(\tau)}, g_R^{Z(\tau)}, g_O^{Z(\tau)}, g_R^{\alpha_R Z(\tau)}, g_O^{\alpha_O Z(\tau)}, g_L^{\beta Z(\tau)}, g_R^{\beta Z(\tau)}, g_O^{\beta Z(\tau)}\Big\} \\
\mathsf{vp} \leftarrow \Big\{ &g_O^{Z(\tau)}, g^{\alpha_L}, g^{\alpha_R}, g^{\alpha_O}, g^{\beta\gamma}, g^{\gamma}, \{g_L^{L_i(\tau)}, g_R^{R_i(\tau)}, g_O^{O_i(\tau)}\}_{i\in\mathcal{I}_{\text{io}}}\Big\}
\end{aligned}$$

- ✓ **Deletes** aforementioned random scalars (toxic waste).

Prove(pp, **x**, **w**)

The prover $\mathcal{P}$ conducts the following steps:

- ✓ Runs the circuit to get **w** and $L(x), R(x), O(x)$.
- ✓ Calculates $H(x) \leftarrow (L(x)R(x) - O(x))/Z(x)$.
- ✓ Splits $L(x) = L_{\text{mid}}(x) + L_{\text{io}}(x)$ — intermediate and input/output parts. Repeat for $R(x)$ and $O(x)$.
- ✓ Samples $\delta_L, \delta_R, \delta_O \xleftarrow{R} \mathbb{F}$ and publishes the following values as a proof $\boldsymbol{\pi}$:

$$\begin{aligned}
\pi_L &\leftarrow g_L^{L_{\text{mid}}(\tau)} \left(g_L^{Z(\tau)}\right)^{\delta_L}, \quad \pi'_L \leftarrow g_L^{\alpha_L L_{\text{mid}}(\tau)} \left(g_L^{\alpha_L Z(\tau)}\right)^{\delta_L}, \\
\pi_R &\leftarrow g_R^{R_{\text{mid}}(\tau)} \left(g_R^{Z(\tau)}\right)^{\delta_R}, \quad \pi'_R \leftarrow g_R^{\alpha_R R_{\text{mid}}(\tau)} \left(g_R^{\alpha_R Z(\tau)}\right)^{\delta_R}, \\
\pi_O &\leftarrow g_O^{O_{\text{mid}}(\tau)} \left(g_O^{Z(\tau)}\right)^{\delta_O}, \quad \pi'_O \leftarrow g_O^{\alpha_O O_{\text{mid}}(\tau)} \left(g_O^{\alpha_O Z(\tau)}\right)^{\delta_O}, \\
\pi_H &\leftarrow g^{H(\tau)} (g^{\delta_O})(g^{\delta_L})(g^{\delta_R})(g^{\delta_L\delta_R}), \\
\pi_\beta &\leftarrow \left(g_L^{\beta Z(\tau)}\right)^{\delta_L} \left(g_R^{\beta Z(\tau)}\right)^{\delta_R} \left(g_O^{\beta Z(\tau)}\right)^{\delta_O} g_L^{\beta L_{\text{mid}}(\tau)} g_R^{\beta R_{\text{mid}}(\tau)} g_O^{\beta O_{\text{mid}}(\tau)}
\end{aligned}$$

Verify(vp, **x**, $\boldsymbol{\pi}$)

Upon receiving $\boldsymbol{\pi} = (\pi_L, \pi'_L, \pi_R, \pi'_R, \pi_O, \pi'_O, \pi_H, \pi_\beta)$, the verifier $\mathcal{V}$:

- ✓ Finds $\pi_L^* \leftarrow \pi_L \prod_{i\in\mathcal{I}_{\text{io}}} (g_L^{L_i(\tau)})^{w_i}, \pi_R^* \leftarrow \pi_R \prod_{i\in\mathcal{I}_{\text{io}}} (g_R^{R_i(\tau)})^{w_i}, \pi_O^* \leftarrow \pi_O \prod_{i\in\mathcal{I}_{\text{io}}} (g_O^{O_i(\tau)})^{w_i}$
- ✓ Checks whether all of the following conditions hold:

$$\begin{aligned}
e(\pi_L^*, \pi_R^*) &= e(g_O^{Z(\tau)}, \pi_H) \cdot e(\pi_O^*, g), && \text{// Polynomial Equality Test} \\
e(\pi_L, g_L^{\alpha_L}) = e(\pi'_L, g_L), \quad e(\pi_R, g_R^{\alpha_R}) &= e(\pi'_R, g_R), && \text{// Proof of Exponent} \\
e(\pi_O, g_O^{\alpha_O}) &= e(\pi'_O, g_O), && \text{// Proof of Exponent} \\
e(\pi_L \pi_R \pi_O, g^{\gamma\beta}) &= e(\pi_\beta, g^{\gamma}). && \text{// Extended Witness Consistency}
\end{aligned}$$

10.2.2 Groth16 Protocol

Finally, Groth16 allows to reduce the number of pairings **down to 3**! This is done through a technique called **Generic Group Model** (GGM for short). Simply put, GGM allows the adversary to only make oracle requests to compute the group operations. For example, having a set $\{g^{\alpha R_i(\tau)}\}_{i\in[d]}$, adversary can compute only linear combinations of these values. In the particular case of Groth16, instead of considering $L_i(x)$, $R_i(x)$, and $O_i(x)$ separately, we construct their linear combinations as $Q_i(x) := \beta L_i(x) + \alpha R_i(x) + O_i(x)$, where α and β are toxic parameters.

Let us now concretely describe the Groth16 construction.

Asymmetric Pairing. One change which was made in Groth16 is using the asymmetric pairing function $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ over groups $\mathbb{G}_1$ and $\mathbb{G}_2$ (for more details, see Section 4.3). Suppose respective group generators are $g_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$. The primary reason for this is that calculating the pairing in asymmetric setting is more efficient than in symmetric setting. Other than that, Groth16 can be easily formulated using symmetric pairings.

Setup Procedure. In Groth16, we need only five random scalars: $\alpha, \beta, \gamma, \delta, \tau \xleftarrow{R} \mathbb{F}$. Now, the prover key pk looks as follows:

$$\mathsf{pp} \leftarrow \left(g_1^{\alpha}, g_1^{\beta}, g_1^{\delta}, \left\{ g_1^{\tau^i}, \frac{\beta L_i(\tau) + \alpha R_i(\tau) + O_i(\tau)}{\gamma}, \frac{\tau^i Z(\tau)}{\delta} \right\}_{i\in[n]}, g_2^{\beta}, g_2^{\delta}, g_2^{\gamma}, \{g_2^{\tau^i}\}_{i\in[d]} \right)$$

Proving. Sample random $\delta_L, \delta_R \xleftarrow{R} \mathbb{F}$ and compute the following values:

$$\pi_L \leftarrow g_1^{\alpha+\sum_{i=1}^{n} w_i L_i(\tau)+\delta_L\delta}, \quad \pi_R \leftarrow g_2^{\beta+\sum_{i=0}^{n} w_i R_i(\tau)+\delta_R\delta}, \quad \pi_O \leftarrow g_1^{\frac{Q_{\mathsf{mid}}(\tau)+H(\tau)Z(\tau)}{\delta}+L\delta_R+R\delta_L-\delta_L\delta_R\delta},$$

where we denoted $Q_{\mathsf{mid}} := \sum_{i\in\mathcal{I}_{\mathsf{mid}}} w_i(\beta L_i(\tau) + \alpha R_i(\tau) + O_i(\tau))$.

Verification. The verifier first calculates the following value:

$$\pi_{\mathsf{io}} \leftarrow g_1^{\sum_{i\in\mathcal{I}_{\mathsf{io}}} w_i(\beta L_i(\tau)+\alpha R_i(\tau)+O_i(\tau))/\gamma},$$

and then checks the following single condition:

$$e(\pi_L, \pi_R) = e(g_1^{\alpha}, g_2^{\beta}) e(\pi_{\mathsf{io}}, g_2^{\gamma}) e(\pi_O, g_2^{\delta})$$

Note that $e(g_1^{\alpha}, g_2^{\beta})$ can be additionally hard-coded in the verifier, thus reducing the number of pairings to 3. Finally, the proof's size is now reduced to 3 group elements: two from $\mathbb{G}_1$, and one from $\mathbb{G}_2$.

11 Circom

11.1 Circom Walkthrough

In this final lecture, we bridge the gap between the theoretical concepts presented in previous lectures and their practical realization using the **Circom** DSL.

Definition 11.1. Circom is a domain-specific language for building arithmetic circuits that can be used to produce zk-SNARK proofs.

Throughout this lecture, we will walk through how concepts like R1CS, witness, trusted setup, and verification keys appear in actual code and practice.

11.1.1 Journey Begins

In the previous lectures, we covered a variety of theoretical concepts: zk-SNARKs, trusted setup, arithmetic circuits, constraints, witnesses, and the Rank-1 Constraint System (R1CS) representation. Now, let's see how these appear in practice.

11.1.2 From Theory to Practice: Circom Basics

We learned that a circuit can represent a complex arithmetic computation over a finite field. Circom allows us to write these circuits in a high-level syntax. To begin, consider the arithmetic circuit $r = x \times y$.

It can be represented in Circom syntax as follows:

```
pragma circom 2.1.6;

template Math() {
    signal input x;
    signal input y;

    signal output r <== x * y;
}

component main = Math();
```

Here, we see how easy it is to define a circuit that takes two inputs x, y and outputs their product r. The `template` defines a reusable circuit component, while `signal input` and `signal output` represent inputs and outputs, respectively. Intermediate signals (without `input` or `output`) are internal primitives within the circuit.

Public vs Private Signals:

Output signals are always public. You may also define public inputs by specifying them in the main component, for example:

```
component main {public [x]} = Math();
```

This means x is a public input and will appear in the verification context. The order of public signals in the final proof verification step follows the order of their definition inside the template, starting with outputs.

For example, if your circuit looks like this:

```
template Circuit() {
    signal x;

    signal output o2;

    signal input c;
    signal input a;

    signal k1;

    signal input b;

    signal output o1;
}

component main {public [a, b, c]} = Circuit();
```

The order of the public signals that should be passed to the verifier is as follows:

$$(o2, o1, c, a, b)$$

11.1.3 Arguments, Functions, and Vars

Sometimes we need to calculate some values as constants for our circuit. For example, if you want your circuit to be a multi-tool that, based on provided arguments, can work with different cases. For this purpose, we can declare *functions* and *vars* inside the circuit, as shown below:

```
function transformNumber(value) {
    return value ** 2;
}

template Math(padding) {
    signal input x;
    signal input y;

    var elementsNumber = transformNumber(padding);
    signal b <== x * elementsNumber;

    signal output r <== b * y;
}

component main {public [x]} = Math(12);
```

Here, `var elementsNumber` and the function `transformNumber` are evaluated at compile time. Remember that assignments using `var` and functions do not produce constraints by themselves. Only <==, ==>, or === and actual arithmetic on signals produce constraints reflected in R1CS.

Remark. Sometimes, one needs to perform operations like division or non-quadratic multiplication on the signal. For this purpose, you can use the --> and <-- notations to compute the values "out-of-circuit". For example:

```
template Math() {
    signal input x;
    signal input y;

    signal b <-- x / y;

    signal output r <== b * y;
}

component main = Math();
```

In this case, no constraints are generated with the x input, and it does not even participate in the witness directly.
Notice! This is the main difference between Circom and other languages. When you write a function evaluation $y = f(x)$ in any other language (say, Python or Rust), you are specifying the set of instructions to compute y from x (commonly sequentially). In Circom (or in any other R1CS language) you are merely asserting the **correctness** of the computation and therefore of all intermediate computations. This way, if your task would have been to compute $y = \frac{1}{x}$, you could simply ask y to be the result of the division (that can be computed out of circuit) and then asserting that $x \times y = 1$ with $x \neq 0$. This way, you are not writing the division itself, but the constraints that the division should satisfy.

11.1.4 Theoretical Recap: Using the Learned Concepts

Next, let us apply the learned concepts to a more complex examples. We start with the `if` statement logic.

Example. Recall the complex example we analyzed in earlier lectures:

```
def r(x1: bool, x2: F, x3: F) -> F:
    return x2 * x3 if x1 else x2 + x3
```

This can be represented as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3).$$

We also had the additional constraint $x_1 \times (1 - x_1) = 0$ to ensure x_1 is binary.
The resulting system of constraints was:

$$x_1 \times x_1 = x_1 \tag{1}$$
$$x_2 \times x_3 = \mathsf{mult} \tag{2}$$
$$x_1 \times \mathsf{mult} = \mathsf{selectMult} \tag{3}$$
$$(1 - x_1) \times (x_2 + x_3) = r - \mathsf{selectMult} \tag{4}$$

It took us quite some time to understand and come up with the constraint system, which can be visualized as follows:

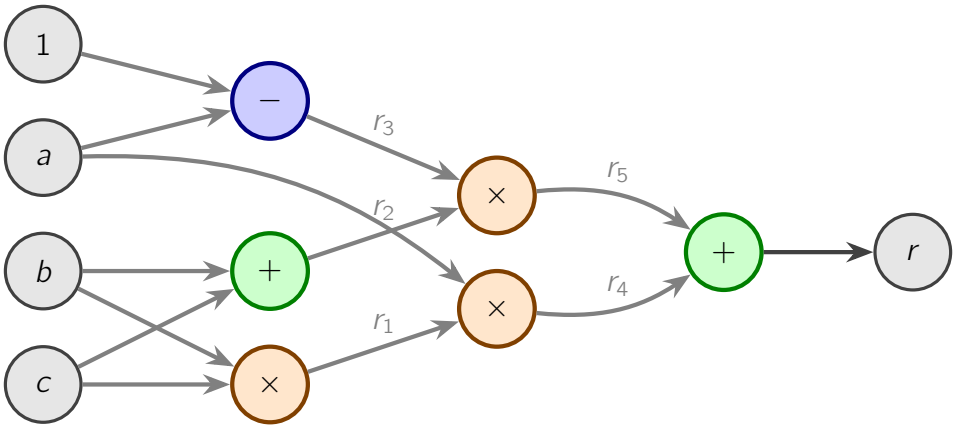


Figure 11.1: Example of a circuit evaluating the `if` statement logic.

The inputs can be directly transformed into signals like below:

```
template Math() {
    signal output r;

    signal input x1;

    signal input x2;
    signal input x3;
}
```

In our case, we have an additional output signal, so we can "return" it from the circuit. Now, let's compare the mathematical and Circom representations.

Mathematical Constraints:

$x_1 \times x_1 = x_1$

$x_2 \times x_3 = \text{mult}$

$x_1 \times \text{mult} = \text{selectMult}$

$(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult}$

Circom Representation:

```
x1 * x1 === x1;
signal mult <== x2 * x3;
signal selectMult <== x1 * mult;
(1 - x1) * (x2 + x3) + selectMult ==> r;
```

As we can see, the translation from math to Circom is straightforward. We have used *signals* for constraint definitions.

Remark. If you wish to follow along with the explanations in the following chapters:

1. Clone the repository https://github.com/ZKDL-Camp/hardhat-zkit-template.
2. Run `npm install` to install dependencies and `npx hardhat zkit make` to compile Circom circuits and generate the necessary artifacts.

11.1.5 From R1CS to Proof Generation

Now, let us break down everything that is happening during the proof generation process. After compilation, you will find the following files in the `zkit/artifacts/circuits` folder

(starting from the project root):

- `.r1cs` file: The Rank-1 Constraint System representation of the circuit.
- `.wasm and *.js` files: The code to compute the witness from the given inputs.
- `.zkey` file: Proving keys after the trusted setup.
- `.sym` file: Symbolic reference for signals.

R1CS File. Let us start with the `.r1cs` file. In Section 8, we defined the following coefficient vectors (in simple constraints) for our task:

$$\mathbf{a}_1 = (0,0,1,0,0,0,0) \quad \mathbf{b}_1 = (0,0,1,0,0,0,0) \quad \mathbf{c}_1 = (0,0,1,0,0,0,0)$$
$$\mathbf{a}_2 = (0,0,0,1,0,0,0) \quad \mathbf{b}_2 = (0,0,0,0,1,0,0) \quad \mathbf{c}_2 = (0,0,0,0,0,1,0)$$
$$\mathbf{a}_3 = (0,0,1,0,0,0,0) \quad \mathbf{b}_3 = (0,0,0,0,0,1,0) \quad \mathbf{c}_3 = (0,0,0,0,0,0,1)$$
$$\mathbf{a}_4 = (1,0,-1,0,0,0,0) \quad \mathbf{b}_4 = (0,0,0,1,1,0,0) \quad \mathbf{c}_4 = (0,1,0,0,0,0,-1)$$

On the other hand, using the test from the `test/Math.witness.test.ts` file and reading the R1CS file, we can see:

test/Math.witness.test.ts

```
expect(constraint1[0]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n, 0n ]);
expect(constraint1[1]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n, 0n ]);
expect(constraint1[2]).to.deep.equal([ 0n, 0n, 1n, 0n, 0n, 0n, 0n ]);

expect(constraint2[0]).to.deep.equal([ 0n, 0n, 0n, babyJub.F.negone, 0n, 0n, 0n ]);
expect(constraint2[1]).to.deep.equal([ 0n, 0n, 0n, 0n, 1n, 0n, 0n ]);
expect(constraint2[2]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n, babyJub.F.negone, 0n ]);

expect(constraint3[0]).to.deep.equal([ 0n, 0n, babyJub.F.negone, 0n, 0n, 0n, 0n ]);
expect(constraint3[1]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n, 1n, 0n ]);
expect(constraint3[2]).to.deep.equal([ 0n, 0n, 0n, 0n, 0n, 0n, babyJub.F.negone ]);

expect(constraint4[0]).to.deep.equal([ babyJub.F.negone, 0n, 0n, 0n, 0n, 0n, 0n ]);
expect(constraint4[1]).to.deep.equal([ 0n, 0n, 0n, 1n, 0n, 0n, 0n ]);
expect(constraint4[2]).to.deep.equal([ 0n, babyJub.F.negone, 0n, 0n, 0n, 0n, 0n ]);
```

Mostly, the structure generated by Circom aligns with what we had devised, except for the last constraint. The difference occurs because of Circom's optimization to make proof generation and verification more efficient.

Now, let's take a closer look at how the witness is computed. In Section 8, we had:

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \mathsf{mult}, \mathsf{selectMult})$$

Given the inputs: $x_1 = 1, x_2 = 3, x_3 = 4$, we can quickly do the math and find out that the actual witness should look like this: $\mathbf{w} = (1, 12, 1, 3, 4, 12, 12)$ based on:

$$\mathsf{mult} = 3 \times 4 = 12$$
$$\mathsf{selectMult} = 1 \times 12 = 12$$
$$r = 1 \times (3 \times 4) + (1 - 1) \times (3 + 4) = 12 + 0 = 12$$

Indeed, it aligns with the test from `test/Math.witness.test.ts`:

test/Math.witness.test.ts

```
expect(witness[0]).to.equal(1n);
expect(witness[1]).to.equal(12n); // r
expect(witness[2]).to.equal(1n);  // x1
expect(witness[3]).to.equal(3n);  // x2
expect(witness[4]).to.equal(4n);  // x3
expect(witness[5]).to.equal(12n); // mult
expect(witness[6]).to.equal(12n); // selectMult
```

The initial 1 in the witness is a constant to facilitate the usage of constants inside the circuit. This corresponds to the fact that $w_0 = 1$ is often used to handle constant terms in R1CS.

The Circom also provides a named representation of all witness elements, which is stored in the `.sym` file and looks as follows:

.sym file for $x_1 ? x_2 \times x_3 : x_2 + x_3$

```
1,1,0,main.r
2,2,0,main.x1
3,3,0,main.x2
4,4,0,main.x3
5,5,0,main.mult
6,6,0,main.selectMult
```

It not only tells us the names of all signals, but also includes information about the optimized signals.

Recall the previous example where we used division and `<--` to store it in the intermediate signal. The generated `.sym` file would look like this:

.sym file for b <-- $x/y, r$ <== by

```
1,1,0,main.r
2,-1,0,main.x
3,2,0,main.y
4,3,0,main.b
```

As we can see, the -1 was added to the signal x to indicate that it is not used in the witness.

Also, according to the documentation of Circom, there are three levels of optimization (ref: https://docs.circom.io/getting-started/compilation-options).

In the `2.1.9` version of Circom, the default optimization was `O2`, but it was lowered to `O1` in following versions because `O2` was too aggressive leading to vulnerable circuits.

Remark. In addition, all linear constraints are optimized on the `O2` optimization.

Now, let's examine what the third column in the `.sym` file means.

Consider the following circuit:

```
1 template BinaryCheck() {
2     signal input x1;
3
4     x1 * x1 === x1;
5 }
6
```

```
7  template SelectMult() {
8      signal input x1;
9
10     signal input x2;
11     signal input x3;
12
13     signal mult <== x2 * x3;
14
15     signal output out <== x1 * mult;
16 }
17
18 template Math() {
19     signal output r;
20
21     signal input x1;
22
23     signal input x2;
24     signal input x3;
25
26     component binCheck = BinaryCheck();
27     binCheck.x1 <== x1;
28
29     component selectMult = SelectMult();
30     selectMult.x1 <== x1;
31     selectMult.x2 <== x2;
32     selectMult.x3 <== x3;
33
34     (1 - x1) * (x2 + x3) + selectMult.out ==> r;
35 }
```

We split the circuit into three parts, and the `.sym` file will look like this:

.sym file for $x_1 ? x_2 \times x_3 : x_2 + x_3$ with templates

```
1,1,2,main.r
2,2,2,main.x1
3,3,2,main.x2
4,4,2,main.x3
5,-1,0,main.binCheck.x1
6,5,1,main.selectMult.out
7,-1,1,main.selectMult.x1
8,-1,1,main.selectMult.x2
9,-1,1,main.selectMult.x3
10,6,1,main.selectMult.mult
```

As we can see, the third column indicates the locality of the signal. Essentially, it tells us which signals are grouped together under the same template.

Another interesting aspect is the order: 0 represents the first component, 1 represents the second component used during computation, and the last component used is the actual 'main'.

Remark. Pay attention to the difference between the modified `Math` circuit's `.sym` file and the original one. Even though we added more signals (i.e., constraints), they were actually optimized by Circom back to the original state.

And the last column is the full name of the constraint, including the path to where it is defined in the code.

11.1.6 Parallel and Custom Keywords

In Circom, there are two special keywords designed to address specific use cases: `custom` and `parallel`.

The `custom` keyword introduces *custom templates* that do not emit R1CS constraints directly. Instead, they delegate logic to `snarkjs` or other libraries at a later stage. Consequently, `custom` templates **cannot** declare subcomponents or add R1CS constraints within their bodies.

The `custom` keyword is used as follows:

```
pragma circom 2.0.6;
pragma custom_templates;

template custom MyCustomGate() {
    // Custom template's code
    // No R1CS constraints or subcomponents can be desclared here
    // Logic will be handled by snarkjs as a PLONK custom gates
}
```

> **Remark.** At the moment of writing the document the `snarkjs` does not support any custom gates (as stated in their documentation). Also, they can be used only in turbo-PLONK or UltraPlonk schemes. Nevertheless, you can find an example of how they have been used here: https://github.com/zkFHE/circomlib-fhe/tree/main.

Meanwhile, the `parallel` keyword (available from Circom 2.0.8 onward) can be applied at either the template or the component instantiation level to parallelize witness generation for independent computations, thereby accelerating large circuits. Parallelism is *only* applied to the C++ witness generator; it does not affect the constraints themselves.

This keyword will be useful in the structures as below:

```
template parallel ParallelExample(n) {
    signal input in[n];
    signal output out[n];

    // Each iteration is independent, so we can parallelize
    for (var i = 0; i < n; i++) {
        out[i] <== in[i] * 2;
    }
}
```

In summary, you should use the `custom` keyword whenever you want to define a template handled **only** by turbo-PLONK or UltraPlonk schemes. You can also find the exact section in the R1CS binary format where custom gates are stored for later processing by the library here: https://github.com/iden3/r1csfile/blob/master/doc/r1cs_bin_format.md#custom-gates-list-section-plonk

The `parallel` keyword is helpful when dealing with large or repetitive computations, as it can speed up witness generation. However, in small circuits or wherever computation is inherently sequential (i.e., where the output of one part is the input to another), `parallel` has no effect.

You can find additional examples of the `parallel` keyword usage here: https://github.com/zkFHE/circomlib-fhe/tree/main.

With this, we have covered all the important files generated by Circom.

11.1.7 Generating and Verifying Proofs

Now, it is time to look at proof generation and verification. In this chapter, our main focus will be on the code from `test/Math.circuit.ts`.

To generate a proof, we need to call the `generateProof` method on the circuit object:

```
const proof = await circuit.generateProof(inputs);
```

The actual proof looks like this:

proof.json


```
{
 "proof": {
  "pi_a": [
   "4705801711565477046837119510773988173091957417270766918367441244292047980064",
   "1400811599548904237959319989696481634963162026439383059052135976273120564167",
   "1"
  ],
  "pi_b": [
   [
    "12538508168416900299033726521685163817792614632620657244409429354131980454661",
    "10914283679966848917795247355212516197618338956682374874239005506750384424444"
   ],
   [
    "11504632457518572930719312464170675169899321263873993433191427524966381618623",
    "15524163713890313070296837080299781036987071183397727452907670321368057103914"
   ],
   [
    "1",
    "0"
   ]
  ],
  "pi_c": [
   "26099670053328208608403811624767970928571072694687572526354364758598879899 8",
   "14278428069254250939292704696175748719031859166075451182707331713513969403299",
   "1"
  ],
  "protocol": "groth16",
  "curve": "bn128"
 },
 "publicSignals": {
  "r": "18"
 }
}
```


Also, at the end of the proof, we have the public signals.

Remark. Usually, public signals are represented by an array of elements, but when using the `hardhat-zkit` plugin, they are typed, and we have actual names for them.

The third element of each program does not participate in any computations; it is needed as additional metadata for the library that implements Groth16 verification.

Remark. When submitting the proof, we have to swap elements inside the arrays of the b point, so that the proof can be verified correctly.

These three points π_L, π_R, π_O are used by the verifier to check the equality:

$$e(\pi_L, \pi_R) = e(g_1^\alpha, g_2^\beta) e(\pi_{\text{io}}, g_2^\gamma) e(\pi_O, g_2^\delta).$$

Other constants needed for the verifier (for example, points g_1^α or g_2^β) are defined in the following file: `zkit/artifacts/circuits/Math.circom/Math.vkey.json` — see Figure 11.2.

Quick reminder about the structure of points in the proof for BN254 (BN128) curve:

- Each point is either from the regular curve $\mathbb{G}_1 : y^2 = x^3 + b$ over $\mathbb{F}_p$ or from the quadratic extension curve $\mathbb{G}_2 : y'^2 = x'^3 + b'$ over $\mathbb{F}_{p^2}$. For BN254, the quadratic extension is defined as $\mathbb{F}_{p^2} = \mathbb{F}_p(i)$ with $i^2 + 1$. Curve coefficients are $b = 3 \in \mathbb{F}_p$ and $b' = \frac{3}{9+i} \in \mathbb{F}_{p^2}$.
- Left inputs to the pairing function e are the points on the regular curve $\mathbb{G}_1$. They are specified in the form of two field elements $(x, y) \in \mathbb{G}_1$, where $x, y \in \mathbb{F}_p$ are the coordinates.
- Right inputs to the pairing function e are the points over the quadratic extension curve $\mathbb{G}_2$. They are specified of the form of four prime field elements $(x_1, y_1, x_2, y_2) \in \mathbb{G}_2$, where the coordinates are $x_1 + iy_1$, $x_2 + iy_2 \in \mathbb{F}_{p^2}$.
- $e(g_1^\alpha, g_2^\beta)$ is the element from the multiplicative group $\mathbb{F}_{p^{12}}^\times$. Therefore, we need 12 prime field elements to represent it.

Remark (On representing $\mathbb{F}_{p^{12}}$ element)**.** One might wonder: why is the element from $\mathbb{F}_{p^{12}}$ is represented as a pair of two arrays, each consisting of three pairs of prime field elements? The primary reason is that the most convenient way to construct $\mathbb{F}_{p^{12}}$ element is to use the so-called **tower of extensions**: we represent an element from $\mathbb{F}_{p^{12}}$ as a pair of two $\mathbb{F}_{p^6}$ elements, while each $\mathbb{F}_{p^6}$ consists of a triplet of $\mathbb{F}_{p^2}$ elements. For more details, see Section 3

Thus, we have covered all the information about the internal structure of the Circom files needed for proof generation and verification.

Finally, we verify the proof in the code:

```
expect(await math.verifyProof(proof)).to.be.true
```

This concludes our first journey into learning Circom.

Remark. Here is a set of links that can be used for a deeper dive into the Circom ecosystem:

1. Circom Documentation: https://docs.circom.io/
2. Circom Libraries (like circomlib): https://github.com/iden3/circomlib

vkey.json


```
{
  "protocol": "groth16",
  "curve": "bn128",
  "nPublic": 1,
  "vk_alpha_1": [
    "20491192805390485299153009773594534940189261866228447918068658471970481763042",
    "9383485363053290200918347156157836566562967994039712273449902621266178545958",
    "1"
  ],
  "vk_beta_2": [
    [
      "6375614351688725206403948262868962793625744043794305715222011528459656738731",
      "4252822878758300859123897981450591353533073413197771768651442665752259397132"
    ],
    [
      "1050524262637026227755290108209435669740983568022059097187317114037133120 6856",
      "2184703510552874540328823269114758472819116273229986533837715969235005913 6679"
    ],
    [ "1", "0" ]
  ],
  "vk_gamma_2": [
    [
      "1085704699902305713594457076223282948137075635957851808699051999328565585 2781",
      "1155973203298638710799100402139228578392581286182119253091740315145239180 5634"
    ],
    [
      "8495653923123431417604973247489272438418190587263600148770280649306958101930",
      "4082367875863433681332203403145435568316851327593401208105741076214120093531"
    ],
    [ "1", "0" ]
  ],
  "vk_delta_2": [
    [
      "1085704699902305713594457076223282948137075635957851808699051999328565585 2781",
      "1155973203298638710799100402139228578392581286182119253091740315145239180 5634"
    ],
    [
      "8495653923123431417604973247489272438418190587263600148770280649306958101930",
      "4082367875863433681332203403145435568316851327593401208105741076214120093531"
    ],
    [ "1", "0" ]
  ],
  "vk_alphabeta_12": [
    [
      [
        "2029413683389138792403550203267699914886160938906632433982220835551125967885",
        "2107270004756275781716103122299751798154334762837936063592554900844203025 2106"
      ],
      [
        "5940354580057074848093997050200682056184807770593307860589430076672439820312",
        "1215663887393161855417182912679219304542105265227936302138216989732475242827 6"
      ],
      [
        "7898200236362823042373859371574133993780991612861777490112507062703164551277",
        "7074218545237549455313236346927434013100842096812539264420499035217050630853"
      ]
    ],
    [
      [
        "7077479683546002997211712695946002074877511277312570035766170199895071832130",
        "1009348341986592038991324502103818229123345154902302522911214827410956543546 5"
      ],
      [
        "4595479056700221319381530156280926371456704509942304414423590385166031118820",
        "1983132848448933378447543278042164129392972613924067517967285627438826939368"
      ],
      [
        "1193412959645552104062078694482782620571362163370628593405704536919395824450 0",
        "8037395052364110730298837004334506829870972346962140206007064471173334027475"
      ]
    ]
  ],
  "IC": [
    [
      "4162541565828872643496914921393902054824387648641933177665940781539334781623",
      "4678293780284819015763290392952715769540194300841323348855962545628746384938",
      "1"
    ],
    [
      "7846408072049176620553358542204120795817938985459251067840222635524693287955",
      "1749475457270506481968184305680826943475404713453868190756625624090780797585 0",
      "1"
    ]
  ]
}
```


Figure 11.2: Verification key for the proof stored in the generated `vkey.json` file.

12 PlonK

12.1 Plonk Arithmetization

Consider we have a certain relation $\mathcal{R}$, which we would like to write down into a processing-prone format over the field $\mathbb{F}$. Plonk arithmetizes this relation into a set of *8 polynomials*, which are then used to verify the witness knowledge. Let us start with the concrete example.

Example. To begin with, observe this fairly simple relation $\mathcal{R}_{\text{example}}$: suppose we have a public input $x \in \mathbb{F}$ and public output $y \in \mathbb{F}$, and we want to prove the knowledge of $e \in \mathbb{F}$ such that $e \times x + x - 1 = y$. Formally, we have the following relation:

$$\mathcal{R}_{\text{example}} = \left\{ \begin{matrix} \textbf{Public Statement:} & x, y \in \mathbb{F} \\ \textbf{Witness}: & e \in \mathbb{F} \end{matrix} \;\middle|\; e \times x + x - 1 = y \right\}$$

Remark. Note that of course, from x and y, it is fairly simple to find e: simply take $\frac{1-x+y}{x}$. However, the Plonk arithmetization is not limited to this simple example, and can be applied to more complex relations, such as hash function pre-image knowledge or any NP statement.

12.1.1 Execution Trace

Standard Plonk is defined as a system with two types of gates: **addition** and **multiplication**. We would explain how to build custom gates later. So, let us consider our program in terms of gates with left, right operands and output.

Example. We need **three gates** to encode our program:

1. **Gate #1**: left e, right x, output $u = e \times x$
2. **Gate #2**: left u, right x, output $v = u + x$
3. **Gate #3**: left v, right x, output $w = v + (-1)$

You might have glanced the intuitive formation of what is called *execution trace table* — a matrix T with columns L, R and O (it is common to denote those as A, B, C to distinguish from another matrix we will discuss later). Moreover, we will mark columns **A**, **B** and **C** in bold to indicate that they are vectors from $\mathbb{F}^N$, where here and hereafter, unless stated otherwise, N is the number of gates in the program.

Example. We might visualize the execution trace table T for the example program as follows:

A	**B**	**C**
2	3	6
6	3	9
9	✗	8

Notice how the last row has no value in **B** column (marked by ✗) — this is reasoned by the fact it is not a variable, but rather a constant, meaning it doesn't depend on execution. Also note that the number of gates in this particular circuit is $N = 3$.

Remark. As you might notice, in contrast to classic R1CS (which we used for Groth16), the standard Plonk arithmetization as is only allows two input values to be processed at a time. This way, if Groth16 requires only one constraint for verifying $x_1(x_2 + x_3 + x_4) = x_5$, Plonk would need three constraints to verify the same statement. Custom gates partially solve this problem as we will see later, but it is important to keep in mind.

12.1.2 Encode the program

It is essential to distinguish the definition of the program and its specific evaluation for the sake of simplicity and efficiency — once having established encoding for the program, you might apply it for any reasonable inputs. Therefore, let us at first focus on what defines whether execution trace table will be considered valid for our circuit, because having a table by itself does not tell much, since it can be populated with any values.

For that reason, we would define two matrices — $Q \in \mathbb{F}^{N\times 5}$ and $V \in \mathbb{Z}_{\geq 0}^{N\times 3}$ where $N \in \mathbb{N}$, again, is the number of gates in the program:

- Q is the **Gate Matrix**, which encodes the values of the gates and stores all the intermediate values computed.
- V is the **Wiring Matrix**, which encodes the wiring of the gates, i.e., how the output of one gate is carried as input to another.

Definition 12.1. The **gate matrix** $Q \in \mathbb{F}^{N\times 5}$ has one row per each gate with columns $\mathbf{Q}_L$, $\mathbf{Q}_R$, $\mathbf{Q}_O$, $\mathbf{Q}_M$, $\mathbf{Q}_C$ from $\mathbb{F}^N$. If columns $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C} \in \mathbb{F}^N$ of the execution trace table form valid evaluation of the circuit, then the following holds:

$$A_i(Q_L)_i + B_i(Q_R)_i + A_iB_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i = 0,\ \forall i \in [N]$$

Using Hadamard product notation, this can be concisely rewritten as:

$$\mathbf{A} \odot \mathbf{Q}_L + \mathbf{B} \odot \mathbf{Q}_R + \mathbf{A} \odot \mathbf{B} \odot \mathbf{Q}_M + \mathbf{C} \odot \mathbf{Q}_O + \mathbf{Q}_C = \mathbf{0}$$

Example. For our program, we would have a following Q table:

$\mathbf{Q}_L$	$\mathbf{Q}_R$	$\mathbf{Q}_M$	$\mathbf{Q}_O$	$\mathbf{Q}_C$
0	0	1	−1	0
1	1	0	−1	0
1	0	0	−1	−1

You can verify that our claim holds for aforementioned trace matrix:

$$2 \times 0 + 3 \times 0 + 2 \times 3 \times 1 + 6 \times (-1) + 0 = 0$$
$$6 \times 1 + 3 \times 1 + 6 \times 3 \times 0 + 9 \times (-1) + 0 = 0$$
$$9 \times 1 + 0 \times 0 + 9 \times 0 \times 0 + 8 \times (-1) + (-1) = 0$$

Recall that columns of trace matrix T are $\mathbf{A} = \begin{bmatrix}2\\6\\9\end{bmatrix}$, $\mathbf{B} = \begin{bmatrix}3\\3\\\times\end{bmatrix}$, $\mathbf{C} = \begin{bmatrix}6\\9\\8\end{bmatrix}$.

Now, we do have a way of encoding gates separately, yet in order to guarantee how result of one gate is carried in as input of the other (*wirings*), we need another matrix — V.

Definition 12.2. The **wiring matrix** $V \in \mathbb{Z}_{\geq 0}^{N \times 3}$ consists of indices of all inputs and intermediate values, so that if T is a valid trace,

$$\forall (i,j)\ \forall (k,\ell):\ V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$$

Put more simply, if two values are equal in V, then the corresponding values (corresponding to these indices) in T must be equal as well.

Example. For our program, V can be defined as follows:

L	R	O
0	1	2
2	1	3
3	✗	4

Here 0 is an index of e, 1 is an index of x, 2 — of intermediate value u, 3 — of v and finally 4 — of output w.

12.1.3 Custom Gates

In order to reach beyond classical operations such as addition and multiplication, one may consider composing a custom gate. The main streamliner of this functionality is a matrix Q, using 5 basic columns of which, you already may build custom logic.

Example. Our entire program may be encoded as one custom gate.

$$Q = \begin{array}{|c|c|c|c|c|}\hline Q_L & Q_R & Q_M & Q_O & Q_C \\ \hline 0 & 1 & 1 & -1 & -1 \\ \hline \end{array} \quad V = \begin{array}{|c|c|c|}\hline \text{L} & \text{R} & \text{O} \\ \hline 0 & 1 & 2 \\ \hline \end{array} \quad T = \begin{array}{|c|c|c|}\hline \text{A} & \text{B} & \text{C} \\ \hline 2 & 3 & 8 \\ \hline \end{array}$$

$$2 \times 0 + 3 \times 1 + 2 \times 3 \times 1 + 8 \times (-1) + (-1) = 0$$

As you can see, custom gates is a good way to reduce the number of constraints needed for the same program.

Remark. Real-world PlonK applications commonly have additional columns in the Q matrix, enabling an even broader set of custom functionality.

12.1.4 Public Inputs

With the current design, we can prove that the computations were done correctly, but we have no restrictions on the values of inputs. For example, when the prover wants to convince the verifier that he knows e for $x = 3$ and $y = 7$, the verifier does not even check whether x is 3 (not to mention whether the result of execution $y = 7$ is correct) in the trace table T. One way of doing this is by incorporating them in three previously defined matrices Q, V, T.

Proposition 12.3. One way to solve this is to use the **equality gates**. Introduce two gadgets:

- **Constant Equality Gate:** Suppose we want to check whether the certain variable equals to the constant value $\alpha \in \mathbb{F}$ at gate with index i. For ith gate, set $(Q_L)_i = -1$, $(Q_C)_i = \alpha$ and other columns to 0. Then, add a row to V with $L = i$, $R = \text{✗}$ and $O = \text{✗}$. Then, to satisfy the condition, the ith left input **must** be equal to α.
- **Nodes Equality Gate:** Suppose we want to check whether the ith and jth gates have equal outputs in the kth gate. Set $(Q_L)_k = 1$, $(Q_R)_k = -1$ with other columns to 0. Add a row to V with $L = i$, $R = j$ and $O = \text{✗}$. Then, to satisfy the condition, the ith and jth outputs **must** be equal.

Example. Suppose the prover wants to prove that he knows e for the public statement $(x, y) = (3, 8)$. We can encode this as follows:

$Q =$

$\mathbf{Q}_L$	$\mathbf{Q}_R$	$\mathbf{Q}_M$	$\mathbf{Q}_O$	$\mathbf{Q}_C$
-1	0	0	0	3
-1	0	0	0	8
1	1	1	-1	1
1	-1	0	0	0

$V =$

L	**R**	**O**
0	✗	✗
1	✗	✗
2	0	3
1	3	✗

$T =$

A	**B**	**C**
3	✗	✗
8	✗	✗
2	3	8
8	8	✗

As can be seen, besides the original program gate, inscribed in the third row, we have three additional gates:

- The first two gates "allocate" two nodes with indices 0 and 1 to the values 3 and 8 respectively. This is done through the *constant equality gates*.
- The last gate checks whether the result of the third gate is equal to the index 1, corresponding to the allocated value 8. This is done through the *nodes equality gate*.

The primary problem with this approach, is that now we have lost agnosticism in Q and V of concrete evaluations. In other words, our circuit is now "hardcoded" to the specific values of public inputs. In order to resolve this, we would define a separate one-column matrix named $\mathbf{\Pi} \in \mathbb{F}^N$, in which we would encode the public inputs.

Example. With only Q modified, we now have:

$\mathbf{\Pi}$
3
8
0
0

$Q =$

$\mathbf{Q}_L$	$\mathbf{Q}_R$	$\mathbf{Q}_M$	$\mathbf{Q}_O$	$\mathbf{Q}_C$
-1	0	0	0	0
-1	0	0	0	0
1	1	1	-1	1
1	-1	0	0	0

Proposition 12.4 (Wrap-up)**.** The matrix T with columns **A**, **B** and **C** $\in \mathbb{F}^N$ encodes correct execution of the program, if the following two conditions hold:

1. $\forall i \in [N] : A_i(Q_L)_i + B_i(Q_R)_i + A_iB_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i + \Pi_i = 0$
2. $\forall (i, j)\, \forall (k, \ell) : V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$

12.1.5 Matrices to Polynomials

Gates Satisfability. Now we can traduce the sets of constraints on matrices to just a few equations on polynomials, as we have already done for Groth16. Let ω be a primitive N-th root of unity[14] and let $\Omega = \{\omega^j\}_{0 \le j < N}$. Although currently the choice of set Ω might seem totally random, in the next sections we will see how the usage of Fast-Fourier Transform (FFT) will make this choice convenient.

Let $a, b, c, q_L, q_R, q_M, q_O, q_C, \pi \in \mathbb{F}^{(\le N)}[X]$ be polynomials of degree at most N that interpolate corresponding columns from matrices at the domain Ω. In other words, we have $\forall j \in [N] : a(\omega^j) = A_j$ and the same holds for other polynomials.

Notice that if our trace matrix is correct, then the first condition of Proposition 12.4 can be reduced to the following polynomial equation:

$$a(\omega^j)q_L(\omega^j) + b(\omega^j)q_R(\omega^j) + a(\omega^j)b(\omega^j)q_M(\omega^j) + c(\omega^j)q_O(\omega^j) + q_C(\omega^j) + \pi(\omega^j) = 0, \ \forall j \in [N]$$

Notice that this essentially means that the left polynomial $aq_L + bq_R + abq_M + cq_O + q_C + \pi$ has roots at ω^j for all $j \in [N]$. This is equivalent to stating that the polynomial $z_\Omega(X) = \prod_{j=0}^{N-1}(X - \omega^j)$ divides the left hand side. Now, the interesting fact...

Lemma 12.5. It so happens that if Ω is a set of N-th roots of unity, then the polynomial $z_\Omega(X) = X^N - 1$ is the vanishing polynomial of Ω.

Proof Idea. If ω is the Nth primitive root, then for any $h \in \Omega$ we have $h^N = 1$ and therefore all elements of Ω are the roots of $X^N - 1$. There are precisely N such roots, so $X^N - 1$ can be decomposed as a product of linear factors $c \cdot \prod_{j=0}^{N-1}(X - \omega^j)$. It is easy to see that $c = 1$ by comparing the leading coefficient.

Aha! So we have that $X^N - 1$ must divide the left polynomial. Let us wrap this up in the following proposition.

Proposition 12.6. Now we can reduce down our first condition of Proposition 12.4 to checking valid execution trace into the following claim over polynomials:

$$\exists t \in \mathbb{F}^{(\le 3N)}[X] : \ aq_L + bq_R + abq_M + cq_O + q_C + \pi = z_\Omega t,$$

where $z_\Omega(X)$ is the vanishing polynomial $X^N - 1$.

Wiring Satisfability. The next step is to shrink the second condition imposed by the V matrix. This may be achieved by introducing the concept of permutation.

Remark. Permutation of the set S is commonly denoted as $\sigma : S \to S$. This function is bijective, meaning that for every $s \in S$ there exists a unique $s' \in S$ such that $\sigma(s) = s'$.

[14] Suppose such ω exists, then $\omega^N = 1$ and $\omega^j \ne 1$ for $0 \le j < N$.

Example. A permutation is a rearrangement of the set, which is in our case:

$$\mathcal{I} = \{(i,j) : \text{such that } 0 \le i < N, \text{ and } 0 \le j < 3\}$$

Naturally, the matrix V induces a permutation σ of this set where $\sigma((i,j))$ equals to the pair of indices of the next occurrence of the value at position (i,j). So, for our example:

$V =$

L	R	O
0	✗	✗
1	✗	✗
2	**0**	3
1	3	✗

We have the following permutation:

$$\sigma(\textbf{(0,0)}) = \textbf{(2,1)},\ \sigma((0,1)) = (0,3),\ \sigma((0,2)) = (0,2)$$

$$\sigma((0,3)) = (0,1),\ \sigma((2,1)) = (0,0),\ \sigma((3,1)) = (2,2)$$

For demonstration purposes, we marked in **green** the index of the first and second occurance of the value 0. For proper σ definition (as it has to be bijective), the application of σ to the last occurance outputs the first one.

Permutation Check. This is probably the most tedious part of PlonK. We split the following derivation into two parts:

- **Set Equality using Polynomials.** We will show how to check whether two sets of field elements are equal using polynomials.
- **Permutation Check using Polynomials.** We will show how to check whether a given function is a permutation using polynomials in several forms.

Set equality. Having defined permutation, we can now reduce the second condition of Proposition 12.4 of valid execution trace matrix into the following check:

$$\forall (i,j) \in \mathcal{I} : T_{i,j} = T_{\sigma(i,j)}$$

You may have noticed how this can be reformulated as equality of two sets A and B:

$$A := \{((i,j), T_{i,j}) : (i,j) \in \mathcal{I}\}$$
$$B := \{(\sigma((i,j)), T_{i,j}) : (i,j) \in \mathcal{I}\}$$

We can reduce this check down to polynomial equations! Here is how: suppose for simplicity we have two sets with two elements $A = \{a_0, a_1\}$ and $B = \{b_0, b_1\}$. Introduce two sets of polynomials $A' = \{a_0 + X, a_1 + X\}$ and $B' = \{b_0 + X, b_1 + X\}$.

When do we have the set equality $A' = B'$? Well, $(a_0 + X)(a_1 + X) = (b_0 + X)(b_1 + X)$ works fine. This is true because of linear polynomial unique factorization property, working as prime factors. Now, we can utilize Schwartz-Zippel lemma to replace the latter formula with $(a_0 + \gamma)(a_1 + \gamma) = (b_0 + \gamma)(b_1 + \gamma)$ for some random $\gamma \xleftarrow{R} \mathbb{F}$ with overwhelming probability, being at least $1 - 2/|\mathbb{F}|$. If we wish to generalize this for arbitrary sets $A = \{a_0, \ldots, a_{k-1}\}$ and $B = \{b_0, \ldots, b_{k-1}\}$, apply the following equivalent check:

$$\prod_{i=0}^{k-1}(a_i+\gamma)=\prod_{i=0}^{k-1}(b_i+\gamma)$$

Let Ω be a domain of the form $\{1,\omega,\ldots,\omega^{k-1}\}$ for some k-th root of unity ω. Let f and g be polynomials that we interpolate at Ω as follows:

$$f(\omega^j)=a_j+\gamma,\quad g(\omega^j)=b_j+\gamma,\quad j\in[k]$$

Then, $\prod_{i=0}^{k-1}(a_i+\gamma)=\prod_{i=0}^{k-1}(b_i+\gamma)$ holds if and only if there is a polynomial $Z\in\mathbb{F}[X]$ such that for all $h\in\Omega$ we have $Z(\omega^0)=1$ and $Z(h)f(h)=g(h)Z(\omega h)$.

Now that we can encode equality of sets of field elements, let's expand this to sets of tuples of field elements. Let $A=\{(a_0,a_1),(a_2,a_3)\}$ and $B=\{(b_0,b_1),(b_2,b_3)\}$. Then, similarly, if

$$A'=\{a_0+a_1Y+X,a_2+a_3Y+X\},\quad B'=\{b_0+b_1Y+X,b_2+b_3Y+X\},$$

then $A=B$ if and only if $A'=B'$. As before, we can leverage Schwartz-Zippel lemma to reduce this down into sampling two random β and $\gamma\xleftarrow{R}\mathbb{F}$ and checking equality of:

$$(a_0+\beta a_1+\gamma)(a_2+\beta a_3+\gamma)=(b_0+\beta b_1+\gamma)(b_2+\beta b_3+\gamma)$$

Permutation Check. Now, to go back to the second condition of Proposition 12.4 which we are trying to formulate in the polynomial domain, it becomes clear that if we somehow encoded inner indices tuple (i,j) into a one field element, we could use the above fact. Recall that $i\in[N]$ and $j\in\{0,1,2\}$. Thus, take the $3N$-th primitive root of unity η and define the bijective map $((i,j),v)\mapsto(\eta^{3i+j},T_{i,j})$. Thus, consider the modified sets:

$$\begin{aligned}A&=\{(\eta^{3i+j},T_{i,j}):(i,j)\in\mathcal{I}\}\\B&=\{(\eta^{3k+\ell},T_{i,j}):(i,j)\in\mathcal{I},\sigma((i,j))=(k,\ell)\}\end{aligned}$$

Sample two random field elements β and $\gamma\xleftarrow{R}\mathbb{R}$. Let $\mathcal{D}=\{1,\eta,\eta^2,\ldots,\eta^{3N-1}\}$. Then, interpolate two polynomials f and g over the defined set $\mathcal{D}$ as follows:

$$\begin{aligned}f(\eta^{3i+j})&=T_{i,j}+\eta^{3i+j}\beta+\gamma,\quad(i,j)\in\mathcal{I}\\g(\eta^{3k+\ell})&=T_{i,j}+\eta^{3k+\ell}\beta+\gamma,\quad(i,j)\in\mathcal{I},\quad\sigma((i,j))=(k,\ell)\end{aligned}$$

Similarly to our previous discussion, there should be a polynomial $Z\in\mathbb{F}[X]$ such that $\forall d\in\mathcal{D}$, we have $Z(\eta^0)=1$ and $Z(d)f(d)=g(d)Z(\eta d)$. This would imply the set equality $A=B$ with overwhelming probability according to Schwartz-Zippel lemma.

Shorter Form. Now, using the $3N$-th root of unity is a bit of overkill, so let us try compressing it down to $\Omega=\{\omega^j\}_{0\le j<N}$ where ω is the Nth root of unity. We will define three polynomials $S_{\sigma,1}$, $S_{\sigma,2}$, $S_{\sigma,3}\in\mathbb{F}[X]$, which are interpolated as follows:

$$\begin{aligned}S_{\sigma,1}(\omega^i)&=\eta^{3k+\ell},\quad(i,0)\in\mathcal{I},\quad\sigma((i,0))=(k,\ell)\\S_{\sigma,2}(\omega^i)&=\eta^{3k+\ell},\quad(i,1)\in\mathcal{I},\quad\sigma((i,1))=(k,\ell)\\S_{\sigma,3}(\omega^i)&=\eta^{3k+\ell},\quad(i,2)\in\mathcal{I},\quad\sigma((i,2))=(k,\ell)\end{aligned}$$

Let k_1 and k_2 be two field elements such that $\omega^i \neq \omega^j k_1 \neq \omega^\ell k_2$ for all possible triplets i, j, ℓ. Recall that β and γ are random field elements. Let f and g be the polynomials that interpolate, respectively, the following values at Ω:

$$f(\omega^i) = \left(T_{i,0} + \omega^i\beta + \gamma\right)\left(T_{i,1} + \omega^i k_1\beta + \gamma\right)\left(T_{i,2} + \omega^i k_2\beta + \gamma\right), \quad i \in [N]$$
$$g(\omega^i) = \left(T_{i,0} + S_{\sigma,1}(\omega^i)\beta + \gamma\right)\left(T_{i,0} + S_{\sigma,2}(\omega^i)\beta + \gamma\right)\left(T_{i,0} + S_{\sigma,3}(\omega^i)\beta + \gamma\right), \quad i \in [N]$$

That being said, there is a polynomial $Z \in \mathbb{F}[X]$ such that $\forall d \in \mathcal{D}$ we have $Z(\omega^0) = 1$ and $Z(d)f(d) = g(d)Z(\omega d)$, implying $A = B$ with overwhelming probability. That being said, we now can encode our program using 8 polynomials mentioned at the very beginning:

$$q_L, q_R, q_M, q_O, q_C, S_{\sigma,1}, S_{\sigma,2}, S_{\sigma,3}$$

These are typically called **common preprocessed input**.

12.1.6 Summary

Having a program for relation $\mathcal{R}$, we saw how it can be represented as a sequence of gates with left, right operands and output. The circuit may be encoded using two matrices Q — for capturing gates, and V — for encoding value carries (*wirings*). Upon execution, we get trace execution matrix T and Π for public inputs.

Definition 12.7. Let $T \in \mathbb{F}^{N\times 3}$ be a trace matrix with columns $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C} \in \mathbb{F}^N$ and let $\mathbf{\Pi} \in \mathbb{F}^N$ be a public input vector. They correspond to a valid execution instance with public input given by $\mathbf{\Pi}$ if and only if:

1. $\forall i \in [N] : A_i(Q_L)_i + B_i(Q_R)_i + A_iB_i(Q_M)_i + C_i(Q_O)_i + (Q_C)_i + \Pi_i = 0$
2. $\forall (i,j), \forall (k,\ell) : V_{i,j} = V_{k,\ell} \implies T_{i,j} = T_{k,\ell}$
3. $\forall i > n : \Pi_i = 0$, where n is the number of public inputs.

Then, we encode these conditions in terms of polynomials.

Definition 12.8. Let $z_\Omega = X^N - 1$ be a vanishing polynomial. Let $T \in \mathbb{F}^{N\times 3}$ be a trace matrix with columns $\mathbf{A}, \mathbf{B}, \mathbf{C} \in \mathbb{F}^N$ and $\mathbf{\Pi} \in \mathbb{F}^N$ be a vector of public signals. They correspond to a valid execution instance with public input given by $\mathbf{\Pi}$ if and only if:

1. $\exists t_1 \in \mathbb{F}[X] : aq_L + bq_R + abq_M + cq_O + q_C + \pi = z_\Omega t_1$
2. $\exists t_2, t_3, z \in \mathbb{F}[X] : zf - gz' = z_\Omega t_2$ and $(z-1)L_1 = z_\Omega t_3$, where $z'(X) = z(X\omega)$.

Remark. We can reduce every needed check down to one equation, if we introduce randomness. Let α be a random field element, then:

$$\begin{aligned} z_\Omega t &= aq_L + bq_R + abq_M + cq_O + q_C + \pi \\ &= \alpha(gz' - fz) \\ &= \alpha^2(z-1)L_1 \end{aligned}$$

The transition between second and third line is very unobvious and requires a bit of algebraic manipulation. Don't worry if you don't see it immediately.

13 ZK-STARK protocol

13.1 Introduction to ZK-STARKs

ZK-STARK (Zero-Knowledge Scalable Transparent Argument of Knowledge) is a cryptographic proof system that allows one party to prove to another the knowledge of a piece of information without revealing the information itself, while ensuring scalability and transparency.

In this context, "scalable" implies that the time required by the prover grows at most quasilinearly (linear up to the logarithmic factor) relative to the runtime of the witness-checking process. Additionally, the verification (including both time and proof size) is limited to a polylogarithmic growth concerning this runtime.

In turn, "transparent" means there is no requirement for a trusted setup, unlike SNARKs. STARK protocol utilizes advanced mathematical techniques like Fast Reed-Solomon IOP of Proximity and Merkle trees to achieve this. The security of STARK lies on the difficulty of computing the inverse function to the hash function, so we can consider STARK as a quantum-safe protocol if the used hash function also owns this property.

Note, that the term "STARK" does not specify the protocol interactivity. But today, most of the STARK protocols (or probably all of the existing protocols) are deployed in the non-interactive environment (which makes all of them SNARKs). This means that we really do not need the additional abbreviation for the existing STARK protocols – all of them can be considered as a "transparent SNARKs".

13.2 STARK-frieldy fields

In general, STARK protocol can work over any field $\mathbb{F}$ with high two-adicity.

Definition 13.1. We call two-adicity fields, the fields where we can select the multiplicative subgroup of order 2^k.

To be honest, all protocol steps are followed with powers of two. It will be shown, why the groups we are working over must be of size 2^k and why the input data also follows this rule. As the result, the maximum size of the statement that we can prove using the STARK protocol is strictly depends on the size of two-adicity subgroup (that is why we label some fields to have *high two-adicity* or *low two-adicity*).

Remark. In our example we consider using field over prime modulus $N = 3 \cdot 2^{30} + 1$ and subgroups of size 2^{13} and 2^{10}.

As we will work in the new subgroup we may want to specify the subgroup generator to be used in future equations. So, for the multiplicative group generator $w \in \mathbb{F}_N^\times$, the generator of the subgroup will be $w^{\frac{N-1}{2^k}}$.

Example. For the prime field $\mathbb{F}_N$ where $N = 3 \cdot 2^{30} + 1$, the order of $\mathbb{F}^\times$ is $N - 1 = 3 \cdot 2^{30}$. If we take $w = 5$ as the primitive element, the multiplicative subgroup of 2^{13} elements generator will be $5^{3 \cdot 2^{17}}$

This kind of subgroups comes with very useful property: for each element in two-adicity subgroup H, the additive inverse element can be calculated by a simple equation over the

element power.

Let's denote the generator of this subgroup h and, as we already examined, $h = w^{\frac{N-1}{|H|}}$, for simplicity. Then, each element can be represented as $x = h^i = w^{\frac{N-1}{|H|} \cdot i}$ where its additive inverse is $-x = h^j = w^{\frac{N-1}{|H|} \cdot j}$. Then, the i and j values obtain the following property: $j = i + \frac{|H|}{2} \mod |H|$.

Evaluation of the additive inverse element. The sum of x and $-x$ must equal to zero by modulus N, so:

$$x + (-x) \equiv w^{\frac{N-1}{|H|} \cdot i}(1 + w^{\frac{N-1}{|H|} \cdot \frac{|H|}{2}}) \equiv 0 \mod N$$

If the multiplication of two equations equals to zero by modulus N, then we can try to check whether one of these equations is zero modulo N:

$$\begin{aligned} 1 + w^{\frac{N-1}{|H|} \cdot \frac{|H|}{2}} &\equiv 0 \mod N \\ 1 + w^{\frac{N-1}{2}} &\equiv 0 \mod N \\ w^{\frac{N-1}{2}} &\equiv N - 1 \mod N \\ w^{N-1} &\equiv (N-1)^2 \mod N \\ w^{N-1} &\equiv N^2 - 2N + 1 \mod N \\ 1 &\equiv 1 \mod N \end{aligned}$$

□

Remark. The equation $w^{N-1} \equiv 1 \mod N$ is obtained from the order property of the primitive element w in the multiplicative group $\mathbb{F}^\times$.

13.3 Protocol definition

13.3.1 Trace, evaluation domain and commitment

Now, we are going to prove that some statement holds on the given sequence of elements.

Definition 13.2. We call **trace** a sequence of elements from $\mathbb{F}$ that represents our witness. This sequence contains private and public values together and follows certain constraints.

Example. The Fibonacci square sequence is a sequence of elements defined as follows:

$$a_i = a_{i-1}^2 + a_{i-2}^2$$

Then, we can for example prove the following statement: *I know a field element x such that the* 1023*rd element of the Fibonacci square sequence starting with* 1 *and x is* 2338775057. (The private x in this case equals to 3141592).

Following the Unisolvence Theorem, the trace a is implied to be an evaluation of some unknown **trace polynomial** of degree $|a| - 1$. Also, to be evaluable on the two-adicity subgroup, the size of the trace has to be a power of two.

Definition 13.3. We call **domain** a two-adicity subgroup $G \in \mathbb{F}$ where we evaluate our polynomials.

Example. In our example, we put trace a sequence a of first 1023 elements of the Fibonacci square sequence over $\mathbb{F}_N$, where $N = 3 \cdot 2^{30} + 1$.

$$1, 1, 2, 5, 29, ...$$

To interpolate our trace polynomial we select as a domain a two-adicity subgroup of 2^{10} elements from $\mathbb{F}^\times$ with generator $g = 5^{\frac{3 \cdot 2^{30}}{2^{10}}}$ (here 5 stands for the primitive element in $\mathbb{F}_N^\times$):

$$G = \{g^i \mid g = 5^{3 \cdot 2^{20}} \wedge i \in [0; 1024)\}$$

Next, using the Lagrange interpolation over $(g^i, a_i)_0^{|a|-1}$ points we compute a trace polynomial $f \in \mathbb{F}[x]$.

Remark. Note, that in practice, you may want to use more efficient algorithm for interpolation, for example – *Fast Fourier Transform (FFT)*.

Definition 13.4. We call **evaluation domain** a two-adicity coset $E = wH \in \mathbb{F}$, where $H \in \mathbb{F}$ is a two-adicity subgroup, that is larger ρ times (some small constant) then the domain.

Example. In our case we select a two-adicity subgroup of 2^{13} elements from $\mathbb{F}^\times$ ($\rho = 8$):

$$H = \{h^i \mid h = 5^{3 \cdot 2^{17}} \wedge i \in [0; 8192)\}$$

Then, we define the evaluation domain as:

$$E = \{5 \cdot h_i \mid \forall h_i \in H\}$$

We build a Merkle tree over the values $f(e_i),\ \forall e_i \in E$ and label it's root as a **trace polynomial commitment**. This approach will also be used to commit other polynomials during the protocol walkthrough.

The **constraints** in STARK protocol are expressed as polynomials evaluated over the trace cells, which are satisfied if and only if the computations are correct.

Example. Obviously, our initial statement consists of the following three requirements:

1. The element a_0 is equal to 1;
2. The element a_{1022} is equal to 2338775057;
3. Each element a_{i+2} is equal to $a_{i+1}^2 + a_i^2 \mod N$.

To verify that our committed trace polynomial satisfies all constraints, we can check that it has corresponding roots. In particular, according to the selected interpolation points (g^i, a_i), the relation $r(a_i, a_j) = 0$ can be rewritten as $r(f(g^i), f(g^j)) = 0$.

Example. For our Fibonacci trace we have the following constraints to be checked over the interpolated polynomial:

1. *The element* a_0 *is equal to* 1 translated to: $f(x) - 1$ has root at $x = g^0 = 1$;
2. *The element* a_{1022} *is equal to* 2338775057 translated to: $f(x) - 2338775057$ has root at $x = g^{1022}$;
3. *Each element* a_{i+2} *is equal to* $a_{i+1}^2 + a_i^2$ translated to: $f(g^2x) - f(gx)^2 - f(x)^2$ has roots in $G \setminus \{g^{1021}, g^{1022}, g^{1023}\}$

To ensure that the specified polynomials have roots in given values, we can use the following property: if polynomial $f(x) \in \mathbb{F}[x]$ has root in x_0 then the $\frac{f(x)}{x - x_0}$ is also a polynomial in $\mathbb{F}[x]$.

Example. Finally, we define the following STARK constraints:

$$p_0(x) = \frac{f(x) - 1}{x - 1}$$
$$p_1(x) = \frac{f(x) - 2338775057}{x - g^{1022}}$$
$$p_2(x) = \frac{f(g^2x) - f(gx)^2 - f(x)^2}{\prod_{i=0}^{1020} x - g^i}$$

Unfortunately, the p_2 polynomial still looks inconvenient to work with, so we may want to simplify it (this is not a part of the protocol in general, but you always may want to simplify your equations to achieve better proving time). Using the following property we can reduce the denominator of p_2:

$$x^{|G|} - 1 = \prod_{g \in G} (x - g), \forall x \in G$$

This equation works because both sides are polynomials whose roots are exactly the elements of G. Note, that while evaluating our polynomial on lager domain then G we should only ensure that the resulting polynomial still holds the relation $f(g^i) = a_i$, so it is acceptable to use properties that only work over G. So, finally we have:

$$p_2(x) = \frac{(f(g^2x) - f(gx)^2 - f(x)^2)(x - g^{2021})(x - g^{2022})(x - g^{2024})}{x^{1024} - 1}$$

In addition, there is one obvious requirement for the STARK constraints: the verifier should be able to compute the constraints polynomials $p_i(x)$ using only the given trace polynomial evaluations for the certain x.

Example. In our Fibonacci example, verifier can check the constraint polynomials evaluation by requesting only $f(x)$, $f(gx)$ and $f(g^2x)$ – the values committed in the trace polynomial commitment.

To combine all our constraints into a single polynomial, we can follow a commonly used principle by taking a linear combination with the challenges from the verifier. In particular, after receiving trace polynomial commitment from the prover, the verifier selects $\{\alpha_i \in \mathbb{F}\}$ and sends

it to the prover. Then, the prover puts the **composition polynomial** as:

$$CP(x) = \sum \alpha_i \cdot p_i(x)$$

Additionally, prover also commits this polynomial by evaluating on the evaluation domain and building a Merkle tree.

Example. The Fibonacci composition polynomial looks like as follows:

$$CP(x) = \alpha_0 p_0(x) + \alpha_1 p_1(x) + \alpha_2 p_2(x) =$$
$$\alpha_0 \frac{f(x) - 1}{x - 1} + \alpha_1 \frac{f(x) - 2338775057}{x - g^{1022}} +$$
$$\alpha_2 \frac{(f(g^2x) - f(gx)^2 - f(x)^2)(x - g^{2021})(x - g^{2022})(x - g^{2024})}{x^{1024} - 1}$$

13.3.2 FRI protocol

In general, our goal is to verify that the committed polynomial $CP(x)$ satisfies all our constraints, by checking it's evaluation at a random point from the evaluation domain that the verifier selects. Anyway, we can face the problem when the malicious prover constructs a lager polynomial that accepts lots of possible roots from our field (even 2^{64} field is still insecure for just checking the evaluation at one point). That is why we have to make sure that the committed polynomial degree lies in the acceptable range (the upper bound depends on the trace size).

The final stage of the STARK protocol is a **Fast Reed-Solomon IOP of Proximity (FRI)**. FRI is a protocol between a prover and a verifier, which establishes that a given evaluation belongs to a polynomial of low-degree. In this context *low* means no more than ρ times bigger than the trace.

The key idea of FRI protocol is to move from a polynomial of degree n to a polynomial of degree $n/2$ until we get a constant value. Let's consider the polynomial $z_0(x) = \sum a_i \cdot x^i$ of degree $n = 2^t$ and the evaluation domain $E_0 = E$. We suppose to group the *odd* and the *even* coefficients of the z_0 together into the two separate polynomials(z_0^o and z_0^e respectively):

$$z_0^o(x^2) = \sum_{i=0}^{n/2} (a_{2i+1} \cdot x^{2i})$$
$$z_0^e(x^2) = \sum_{i=0}^{n/2} (a_{2i} \cdot x^{2i})$$

Or, in more comfortable form (we have already examined why searching of $-x$ can be done easily in our two-adicity subgroup):

$$z_0^e(x^2) = \frac{z_0(x) + z_0(-x)}{2}$$
$$z_0^o(x^2) = \frac{z_0(x) - z_0(-x)}{2x}$$

Then, we define a next-layer of the FRI polynomial as $z_1(x^2) = z_0^e(x^2) + \beta z_0^o(x^2)$, where β is a challenge received from verifier. The next-layer evaluation domain is also simple to compute: $E_1 = \{(w \cdot h_i)^2 \mid i \in [0; \frac{|E_0|}{2})\}$

Next, we commit to the $z_1(x^2)$ using a next-layer evaluation domain E_1 (is also reduced by a factor two) and continue to repeat the described operations until $z_i(x^{2^i})$ becomes constant.

Interactive ZK-STARK protocol

The prover and the verifier run the interactive version of the ZK-STARK protocol. Both know the statement to be proved, that is defined by the constraint polynomials and the field $\mathbb{F}$ to work over. Prover also knows the witness to be able to generate the trace.

Preparation

- ✓ The prover interpolates trace polynomial $f(x)$ and submits it's commitment to the verifier.
- ✓ The verifier selects challenges random $\alpha_0, \alpha_1, \alpha_2 \in \mathbb{F}$ and sends to the prover.
- ✓ The prover builds the composition polynomial $CP(x)$ and submits it's commitment to the verifier.

FRI

- ✓ The verifier selects random $i \in [0; |E|)$, puts $c = w \cdot h^i$ and sends it to the prover.
- ✓ The prover responds with the $CP(c), CP(-c)$ and all $f(x)$ required to check CP evaluation with corresponding Merkle proofs to them.
- ✓ The verifier checks Merkle proofs and the evaluation of $CP(c)$ by evaluating the constraints polynomials $p_i(c)$.
- ✓ The prover and the verifier go through the FRI protocol for $z_0(x) = CP(x)$ where the prover commits to the layer-i polynomial $z_i(x)$, the verifier selects a challenge β and queries from the prover $z_i(c), z_i(-c)$ to compute $z_{i+1}(c)$ until $z_i(x), i \leq log_2(\deg CP(x))$ becomes constant.

The non-interactive version of the presented protocol can be easily built obtaining the Fiat-Shamir heuristics.

The soundness of the presented STARK protocol follows from the impossibility to commit any possible evaluation of the forgery $CP(x)$ over evaluation domain E and simultaneously prove that $CP(x)$ is a low-degree polynomial by the FRI protocol. Since the size of E is ρ times bigger then the maximum allowed polynomial degree (that directly depends on the size of the trace), the attacker either can't construct such a polynomial or can't construct a low-degree polynomial, so a valid low-degree composite polynomial can only be obtained using a valid trace.

Example. Finally, let's overview the first steps of the ZK-STARK protocol applied to our Fibonacci example:

1. The protocol defines the public constaints such as 2023-th element of sequence, field $\mathbb{F}$, etc.
2. The prover generates the trace a where $a_0 = 1, a_1 = 3141592, a_i = a_{i-1}^2 + a_{i-2}^2$, evaluates the trace polynomial $f(x)$ over the evaluation domain and sends it's commitments to the verifier
3. The verifier selects challenges $\alpha_0, \alpha_1, \alpha_2 \in \mathbb{F}$ and shares them with the prover
4. The prover evaluates the composition polynomial $CP(x)$ over evaluation domain and sends it's commitments to the verifier
5. The verifier selects random $i \in [0; 8192 - 16)$, puts $c = 5 \cdot h^i$ and sends it to the prover
6. The prover responds with the $f(c), f(gc), f(g^2c), CP(c), CP(-c)$ and corresponding Merkle proofs to them
7. The verifier checks Merkle proofs and the evaluation of $CP(c)$ by evaluating the constraint polynomials $p_0(c), p_1(c), p_2(c)$
8. The prover and the verifier go through the FRI protocol for $z_0(x) = CP(x)$ until $z_i(x), i \in [1; 12)$ becomes constant

13.4 Protocol security

Most of the existing versions of the STARK protocol leverage on several optimizations to achieve better proving and verification time. The key point here is that each FRI query check adds $log_2(\rho)$ bits of security, so we can skip some of these checks if the security level is already satisfied. One more optimization is to include a proof-of-work computation into the protocol that should be done before FRI with dependency on the committed values. It can be useful because the verification of the proof-of-work is less expensive then the verification of the FRI step while still increases the computation cost for the malicious prover.

More precisely, let's assume that the desired security level of the protocol is λ. First of all, we obviously have to use a proper collision-resistant hash function with 2λ bits output. Then, according to the StarkWare's definition of the STARK protocol, the resulting security is defined as follows:

$$\lambda \geq min\{\delta + log_2(\rho) \cdot s, log_2(|F|)\} - 1$$

where δ – number of the proof-of-work bits, s – number of the FRI queries.

Example. If the protocol is deployed over 256-bit field and the domain ratio is $\rho = 3$, to achieve the 128 bit security we can for example execute 33 FRI query and evaluate 29 proof-of-work bits: $min\{29 + 3 \cdot 33, 256\} = 128$.