0.1 What is zk-SNARK?

0.1.1 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: Schnorr protocol (which is a special case of a Σ-protocol). Now, we add one more component which completely changes the game and significantly extends the number of applications: **succinctness**.

Definition 0.1. 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 typically 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.

0.1.2 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 0.2. A **preprocessing non-interactive argument of knowledge** (**preprocessing NARK**) $\Pi_{\mathsf{preNARK}} = (\mathsf{Setup}, \mathsf{Prove}, \mathsf{Verify})$ consists of three algorithms:

- $\mathsf{Setup}(1^\lambda) \to (\mathsf{pk}, \mathsf{vp})$ — 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 0.3 (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.

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 0.4. 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 $|w|$:

$$|\pi| = \mathsf{sublinear}(|w|),\ T_{\mathcal{V}} = O_\lambda(|x|, \mathsf{sublinear}(|C|)).$$

However, 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 compu-

tation. This is the case for most zk-SNARKs used in practice.

Definition 0.5. A **zk-SNARK** is **strongly succinct** if the proof size and verification time are constant or logarithmic in the size of the computation:

$$|\pi| = O_\lambda(\log |C|),\ T_\mathcal{V} = O_\lambda(|x|, \log |C|).$$

0.2 Arithmetic Circuits

0.2.1 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 and addition operations are allowed. Additionally, arithmetic circuits manipulate over elements from some finite field $\mathbb{F}$ (see right figure below).

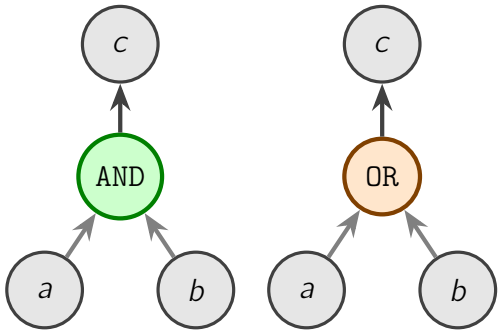

Figure 1: Boolean `AND` and `OR` Gates

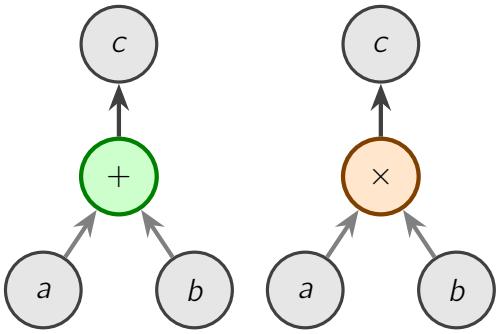

Figure 2: Addition and Multiplication Gates

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 3, calculating $(a$ `AND` $b)$ `OR` c.

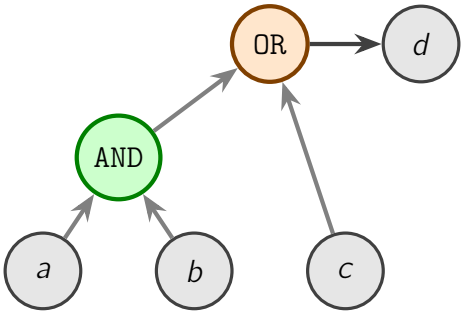

Figure 3: Example of a circuit evaluating $d = (a \texttt{ AND } b) \texttt{ OR } c$.

Although we can already represent very complex computations using boolean circuits[1], they are not the most convenient way to represent arithmetic operations.

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 $\{0, 1\}$.

0.2.2 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 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

[1] ...such as SHA-256 hash function computation, one might take a look here: http://stevengoldfeder.com/projects/circuits/sha2circuit.html

reason, we need three multiplication and one addition gate to represent $Q(x_1, x_2)$ calculation.

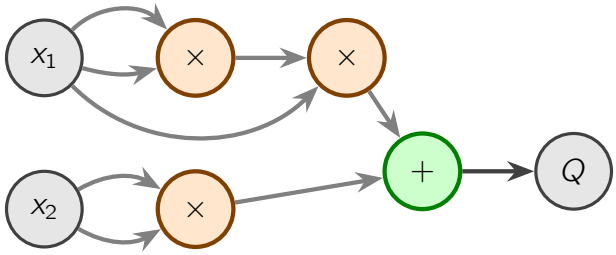

Figure 4: Example of a circuit evaluating $x_1^3 + x_2^2$.

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.

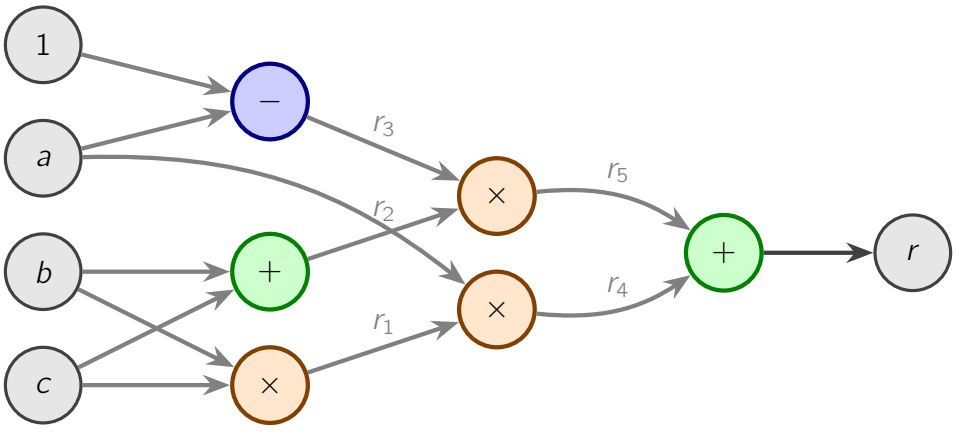

Figure 5: Example of a circuit evaluating the `if` statement logic.

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 $(6, 6, 5, 0, 6, 0, 1, 2, 3)$.

0.2.3 Circuit Satisfability Problem

Now, let us generalize what we have constructed so far. First, we begin with the arithmetic circuit.

Definition 0.6. Arithmetic circuit $\mathtt{C} : \mathbb{F}^N \to \mathbb{F}$ 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 $|\mathtt{C}|$ we denote the number of gates in the circuit.

Now, suppose that the circuit is defined over N inputs. We can always split this input into two parts: the first n inputs are the *public* inputs, being our statement, and the remaining $m = N - n$ inputs are the *private* inputs. 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 there exists a witness that resolves the circuit.

Definition 0.7. The **Circuit Satisfiability Problem** is defined as follows: given an arithmetic circuit $\mathtt{C}$ and a public input $x \in \mathbb{F}^n$, determine if there exists a private input $w \in \mathbb{F}^m$ such that $\mathtt{C}(x, w) = 0$. More formally, the problem is determined by relation $\mathcal{R}_\mathtt{C}$ and corresponding language $\mathcal{L}_\mathtt{C}$ as follows:

$$
\mathcal{R}_\mathtt{C} = \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^m \mid \mathtt{C}(x, w) = 0\}, \ \mathcal{L}_\mathtt{C} = \{x \in \mathbb{F}^n \mid \exists w \in \mathbb{F}^m : \mathtt{C}(x, w) = 0\}
$$

0.3 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.

Definition 0.8. The **inner product** of a linear space $\mathbb{V}$ is any symmetric, linear in the first argument, and positive binary function from vector space to a set of scalars.

$$\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \to \mathbb{F}$$

$\forall \mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{V}, \forall a \in \mathbb{F}$ the following properties are satisfied:

- Symmetry: $\langle \mathbf{u}, \mathbf{v} \rangle = \langle \mathbf{v}, \mathbf{u} \rangle$
- Linearity in the first argument: $\langle c\mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = c\langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$
- Positivity: $\langle \mathbf{u}, \mathbf{u} \rangle \geq 0$ and $\langle \mathbf{u}, \mathbf{u} \rangle = 0 \Leftrightarrow \mathbf{u} = 0$

Plenty of functions can be built that satisfy the inner product definition, we'll use the one that is usually called **dot product**.

Definition 0.9. Let $\mathbb{V}$ be a vector space over the field $\mathbb{F}$. The **dot product** on $\mathbb{V}$ is a function:

$$\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \to \mathbb{F}$$

defined for $\mathbf{u}, \mathbf{v} \in \mathbb{V}$ as follows:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{n} u_i v_i$$

Alternatively, the dot product can also be denoted using the dot notation as:

$$\mathbf{u} \cdot \mathbf{v}$$

That is why it's 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:

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=1}^{3} u_i v_i = 2 \cdot 1 + 2 \cdot 4 + 3 \cdot 3 = 2 + 8 + 9 = 19$$

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 0.10. 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. Let's get back to our example from the previous chapter about the arithmetic circuits.

$$r = x_1 \times x_2$$

The constraint is:

$$(a_1 w_1 + a_2 w_2 + a_3 w_3)(b_1 w_1 + b_2 w_2 + b_3 w_3) = c_1 w_1 + c_2 w_2 + c_3 w_3$$

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:

$$(0w_1 + 1w_2 + 0w_3)(0w_1 + 0w_2 + 1w_3) = (1w_1 + 0w_2 + 0w_3)$$

$$w_2 \times w_3 = w_1$$

$$x_1 \times x_2 = r$$

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. And a more complex example. Remember that we want to verify each computational step.

```
if (x1) {
    return x2 * x3
} else {
    return 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:

$$x_1 \times x_1 = x_1 \quad \text{(binary check)} \tag{1}$$
$$x_2 \times x_3 = \mathsf{mult} \tag{2}$$
$$x_1 \times \mathsf{mult} = \mathsf{selectMult} \tag{3}$$
$$(1 - x_1) \times (x_2 + x_3) = r - \mathsf{selectMult} \tag{4}$$

For every constraint we need the coefficients vectors a_i, b_i, c_i, but all of them have the same witness vector $\mathbf{w}$.

$$\mathbf{w} = (1, r, x_1, x_2, x_3, \mathsf{mult}, \mathsf{selectMult})$$

The coefficients vectors:

$$a_1 = (0,0,1,0,0,0,0) \qquad b_1 = (0,0,1,0,0,0,0) \qquad c_1 = (0,0,1,0,0,0,0)$$
$$a_2 = (0,0,0,1,0,0,0) \qquad b_2 = (0,0,0,0,1,0,0) \qquad c_2 = (0,0,0,0,0,1,0)$$
$$a_3 = (0,0,1,0,0,0,0) \qquad b_3 = (0,0,0,0,0,1,0) \qquad c_3 = (0,0,0,0,0,0,1)$$
$$a_4 = (1,0,-1,0,0,0,0) \qquad b_4 = (0,0,0,1,1,0,0) \qquad c_4 = (0,1,0,0,0,0,-1)$$

Now, let's use specific values to compute an example. Using the arithmetic in a large finite field $\mathbb{F}_p$.

$$x_1 = 1, \quad x_2 = 3, \quad x_3 = 4$$

Verifying the constraints:

1. $x_1 \times x_1 = x_1 \quad (1 \times 1 = 1)$
2. $x_2 \times x_3 = \mathsf{mult} \quad (3 \times 4 = 12)$
3. $x_1 \times \mathsf{mult} = \mathsf{selectMult} \quad (1 \times 12 = 12)$
4. $(1 - x_1) \times (x_2 + x_3) = r - \mathsf{selectMult} \quad (0 \times 7 = 12 - 12)$

Each constraint enforces that the product of the linear combinations defined by **a** and **b** must equal the linear combination defined by **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.

The last unresolved question is where the "rank-1" comes from.

To understand this, we need the knowledge of the outer product and its properties.

Definition 0.11. 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:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} u_1 v_1 & u_1 v_2 & \cdots & u_1 v_n \\ u_2 v_1 & u_2 v_2 & \cdots & u_2 v_n \\ \vdots & \vdots & \ddots & \vdots \\ u_m v_1 & u_m v_2 & \cdots & u_m v_n \end{pmatrix}$$

With the following properties $\forall (c, \mathbf{u}, \mathbf{v}, \mathbf{w}) \in \mathbb{F} \times \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^p$:

- Transpose: $(\mathbf{u} \otimes \mathbf{v}) = (\mathbf{v} \otimes \mathbf{u})^\mathsf{T}$
- 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:

$$\mathbf{u} \otimes \mathbf{v} = \begin{pmatrix} 1 \cdot 2 & 1 \cdot 4 & 1 \cdot 3 \\ 2 \cdot 2 & 2 \cdot 4 & 2 \cdot 3 \\ 3 \cdot 2 & 3 \cdot 4 & 3 \cdot 3 \end{pmatrix} = \begin{pmatrix} 2 & 4 & 3 \\ 4 & 8 & 6 \\ 6 & 12 & 9 \end{pmatrix}$$

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.

Using the outer product we can express the constraint in another form.

Lemma 0.12. Suppose there is a constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$ with coefficient vectors $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ and witness vector $\mathbf{w}$. Then it can be expressed in the form:

$$\mathbf{w}^T A \mathbf{w} + \mathbf{c}^T \mathbf{w} = 0$$

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. Suppose there is a constraint $\langle \mathbf{a}, \mathbf{w} \rangle \times \langle \mathbf{b}, \mathbf{w} \rangle = \langle \mathbf{c}, \mathbf{w} \rangle$. Where vectors

$\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{w} \in \mathbb{F}^n$. Let's expand the inner products:

$$\sum_{i=1}^{n} a_i w_i \times \sum_{j=1}^{n} b_j w_j = \sum_{k=1}^{n} c_k w_k$$

Combine the products into a double sum on the left side:

$$\sum_{i=1}^{n} \sum_{j=1}^{n} a_i b_j w_i w_j = w^T (a \otimes b) w = w^T A w$$

Thus, the constraint can be written as:

$$\mathbf{w}^T A \mathbf{w} + \mathbf{c}^T \mathbf{w} = 0$$

So, the rank-1 means the rank of the coefficients matrix A in one of the constraint formats.

0.4 Quadratic Arithmetic Program

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.

To define a constraint in the R1CS we need four vectors: three coefficient vectors (**a**, **b**, and **c**) and the witness one (**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.

Definition 0.13. Consider a Rank-1 Constraint System (R1CS) defined by m constraints. Each constraint is associated with coefficient vectors $a_i b_i$ and 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.

$$A = \begin{pmatrix} a_{11} & a_{12} & \ldots & a_{1n} \\ a_{21} & a_{22} & \ldots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \ldots & a_{mn} \end{pmatrix} \quad B = \begin{pmatrix} b_{11} & b_{12} & \ldots & b_{1n} \\ b_{21} & b_{22} & \ldots & b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ b_{m1} & b_{m2} & \ldots & b_{mn} \end{pmatrix} \quad C = \begin{pmatrix} c_{11} & c_{12} & \ldots & c_{1n} \\ c_{21} & c_{22} & \ldots & c_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ c_{m1} & c_{m2} & \ldots & c_{mn} \end{pmatrix}$$

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 $\mathbf{w}$.

Example. The vectors a_i from the previous examples:

$$\begin{aligned} a_1 &= (0, 0, 1, 0, 0, 0, 0) \\ a_2 &= (0, 0, 0, 1, 0, 0, 0) \\ a_3 &= (0, 0, 1, 0, 0, 0, 0) \\ a_4 &= (1, 0, -1, 0, 0, 0, 0) \end{aligned}$$

This corresponds to $n = 7, m = 4$

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} & a_{16} & a_{17} \\ a_{21} & a_{22} & a_{23} & a_{24} & a_{25} & a_{26} & a_{27} \\ a_{31} & a_{32} & a_{33} & a_{34} & a_{35} & a_{36} & a_{37} \\ a_{41} & a_{42} & a_{43} & a_{44} & a_{45} & a_{46} & a_{47} \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

The columns of these matrices represent the mappings from constraint number i to the corresponding coefficient of the j element in the witness vector.

Example. Considering the witness from the previos 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:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Thus, for constraint number 4 ($i = 4$) the coefficient of x_1 is -1:

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

As we know from the previous chapters, such a mapping in math can be built using 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 follows:

$$\{(i, a_{ij}) \mid i \in \{1, 2, \ldots, m\}\}$$

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's used in the 1st, 3rd, and 4th constraints as the values of the coefficients aren't zero.
The Lagrange interpolation polynomial for this set of points can be built as follows:

$$\begin{aligned}
\ell_1(x) &= \frac{(x-2)(x-3)(x-4)}{(1-2)(1-3)(1-4)} = -\frac{(x-2)(x-3)(x-4)}{6}, \\
\ell_2(x) &= \frac{(x-1)(x-3)(x-4)}{(2-1)(2-3)(2-4)} = \frac{(x-1)(x-3)(x-4)}{2}, \\
\ell_3(x) &= \frac{(x-1)(x-2)(x-4)}{(3-1)(3-2)(3-4)} = -\frac{(x-1)(x-2)(x-4)}{2}, \\
\ell_4(x) &= \frac{(x-1)(x-2)(x-3)}{(4-1)(4-2)(4-3)} = \frac{(x-1)(x-2)(x-3)}{6}.
\end{aligned}$$

$$\begin{aligned}
A_1(x) &= 1 \cdot \ell_1(x) + 0 \cdot \ell_2(x) + 1 \cdot \ell_3(x) + (-1) \cdot \ell_4(x) \\
&= -\frac{(x-2)(x-3)(x-4)}{6} - \frac{(x-1)(x-2)(x-4)}{2} - \frac{(x-1)(x-2)(x-3)}{6} \\
&= -\frac{5}{6}x^3 + 6x^2 - \frac{79}{6}x + 9
\end{aligned}$$

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 Figure 6, 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.

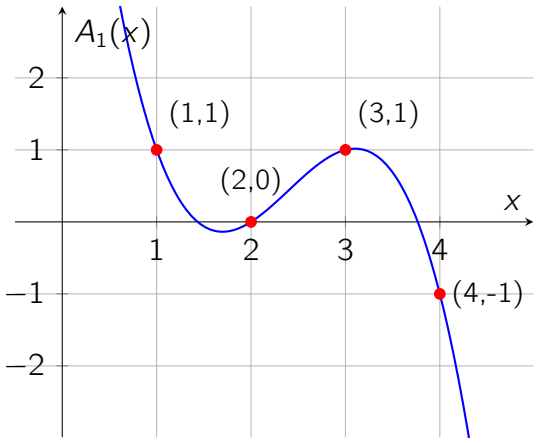

Figure 6: The Lagrange inteprolation polynomial for points $\{(1,1),(2,0),(3,1),(4,-1)\}$

The degree of coefficient polynomials doesn't exceed $m-1$.

Now, using coefficients encoded with polynomials, a constraint number $X \in \{1,\dots\ m\}$, from a constraint system with a witness vector **w** can be built in the next way:

$$(w_1A_1(X) + w_2A_2(X) + \cdots + w_nA_n(X)) \times (w_1B_1(X) + w_2B_2(X) + \cdots + w_nB_n(X)) = \\ = (w_1C_1(X) + w_2C_2(X) + \cdots + w_nC_n(X))$$

Or:

$$\left(\sum_{i=1}^{n} w_iA_i(X)\right) \times \left(\sum_{i=1}^{n} w_iB_i(X)\right) = \left(\sum_{i=1}^{n} w_iC_i(X)\right)$$

Remark. 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 7), the values at each point for the corresponding x are the sum of the initial polynomials' points.

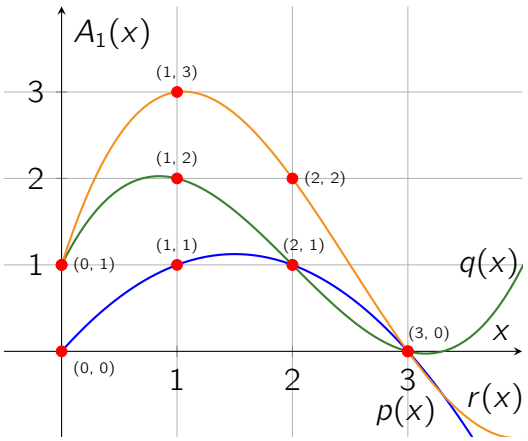

Figure 7: Addition of two polynomials

Now, back to the constraint defined using polynomials, let's define polynomials $A(x), B(x), C(x)$ as:

$$A(X) = \sum_{i=1}^{n} w_i A_i(X), \quad B(X) = \sum_{i=1}^{n} w_i B_i(X), \quad C(X) = \sum_{i=1}^{n} w_i C_i(X)$$

Thus:

$$A(X) + B(X) = C(X)$$

Therefore, as the polynomial $P(X) = A(X) + B(X) - C(X)$ has roots at $x = 1, \ldots, n$, it can be divide by $T(X) = \prod_{i=1}^{n}(x - i)$ without a remainder!

$$\frac{A(X) + B(X) - C(X)}{(X-1)(X-2)\ldots(X-n)} = \frac{P(X)}{T(X)} = H(X)$$

Where $T(X)$ is called the target polynomial.

This was our final step in representing a high-level programming language to some math primitive. We've managed to encode our computation to a single polynomial.