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: 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 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 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.

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:

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

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 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 $|w|$:

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

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 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:

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

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 $|\pi| = O_\lambda(\text{polylog}(N))$ and $T_{\mathcal{V}} = O_\lambda(\text{polylog}(N))$ 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.

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

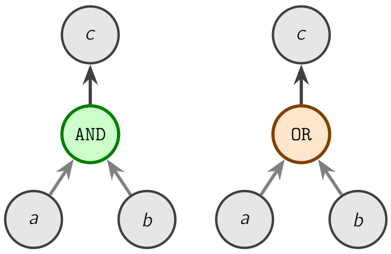

Figure 0.1: Boolean `AND` and `OR` Gates

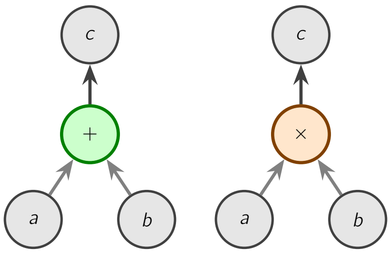

Figure 0.2: Addition and Multiplication Gates

A	B	A AND B
0	0	0
0	1	0
1	0	0
1	1	1

Table 1: `AND` Gate Truth Table

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.

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 0.3, calculating $(a \texttt{ AND } b) \texttt{ OR } c$.

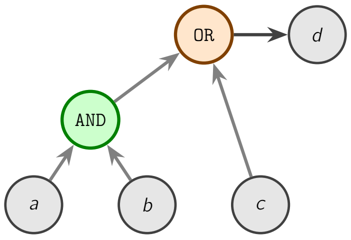

Figure 0.3: Example of a circuit evaluating $d = (a \texttt{ AND } b) \texttt{ OR } c$.

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 $\{0, 1\}$.

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

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 0.5. 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.

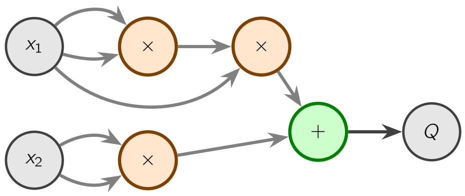

Figure 0.4: Example of a circuit evaluating $x_1^3 + x_2^2$.

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:

```python
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.

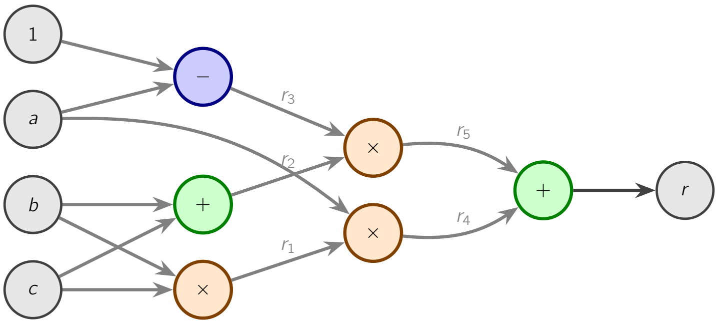

Figure 0.5: Example of a circuit evaluating the `if` statement logic.

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 $C : \mathbb{F}^n \to \mathbb{F}$ 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 $1, x_1, x_2, \dots, x_n$. 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 0.5. 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 0.7. 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.

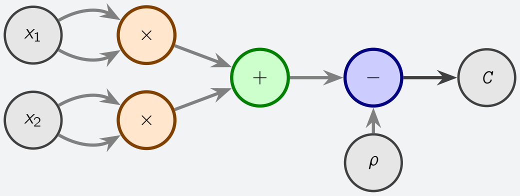

Illustration: *Arithmetic circuit for the equation* $x_1^2 + x_2^2 - \rho$.

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 0.8 (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 $T_{\mathcal{V}} = O(|C|)$ (since 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.

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.

0.3.1 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 0.9. 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:

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

For every constraint we need the coefficients vectors a_i, b_i, c_i, but all of them have the same witness vector **w**.

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

The coefficients vectors:

$$\begin{array}{lll} \mathbf{a}_1 = (0,0,1,0,0,0,0) & \mathbf{b}_1 = (0,0,1,0,0,0,0) & \mathbf{c}_1 = (0,0,1,0,0,0,0) \\ \mathbf{a}_2 = (0,0,0,1,0,0,0) & \mathbf{b}_2 = (0,0,0,0,1,0,0) & \mathbf{c}_2 = (0,0,0,0,0,1,0) \\ \mathbf{a}_3 = (0,0,1,0,0,0,0) & \mathbf{b}_3 = (0,0,0,0,0,1,0) & \mathbf{c}_3 = (0,0,0,0,0,0,1) \\ \mathbf{a}_4 = (1,0,-1,0,0,0,0) & \mathbf{b}_4 = (0,0,0,1,1,0,0) & \mathbf{c}_4 = (0,1,0,0,0,0,-1) \end{array}$$

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 **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.

0.3.2 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 0.10. 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.