0.1 What the zk-SNARK is?

Let's first discuss what zk-SNARK is.

Definition 0.1. zk-SNARK – Zero-Knowledge Succinct Non-interactive ARgument of Knowledge.

But 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 know data that resolves a certain problem, and this knowledge can be verified.
- **Succinct** - the proof size is relatively small and does not depend on the size of the data or statement. This will be explained with examples later.
- **Non-interactive** - 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.

From the above, you may also find the presence of two parties:

- **Prover** - the party who knows the data that can resolve the given problem.
- **Verifier** - the party that wants to verify the given proof.

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 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 don't have any background. Let's take a look at the example.

Example. Imagine you're 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.
- Succinct: 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-interactive: 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.

0.2 Arithmetic Circuits

The cryptographic tools we've learned in the previous sections work with numbers or certain primitives above them, so the first question is: how do we convert a program into mathematical language? Additionally, we need to do this in a way that is succinct and allows us to prove something about it.

The **Arithmetic circuits** can help us. Similar to boolean circuits, they have gates and wires, but instead of the operations AND, OR, and NOT, only multiplication and addition are allowed. Additionally, arithmetic circuits manipulate numbers directly, which are often elements of a prime field.

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

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

Table 1: AND Gate Truth Table

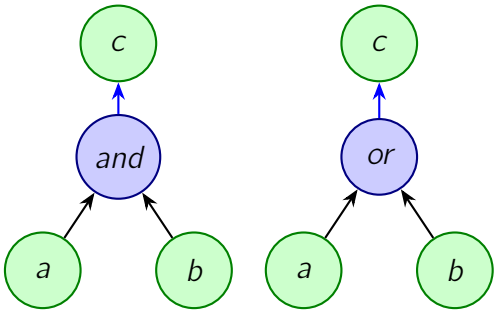

Figure 1: Boolean AND and OR Gates

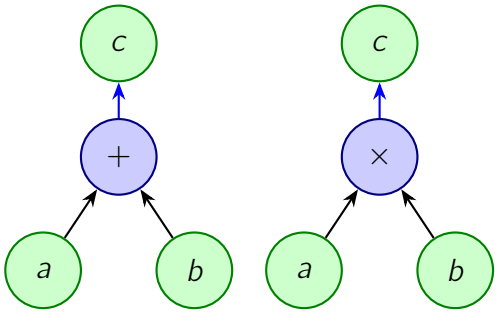

Figure 2: Addition and Multiplication Gates

the statement.

We can do the same with **arithmetic circuits** to verify computations without excessive verbosity because of binary arithmetic.

Let's take a look at some examples of programs and how can we translate them to the arithmetic circuits.

Very simple program with a multiplication.

```
return a * b
```

This can be represented as a circuit with only one gate:

$$r = a \times b$$

The witness vector (solution vector) is $w = (r, a, b)$, 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 don't resolve the circuit.

How can we translate an if statement?

```
if (a) {
    return b * c
} else {
    return b + c
}
```

We can express this logic in mathematical terms as follows: "If a is true, compute $b \times c$; otherwise, compute $b + c$." Only numerical expressions are allowed. Knowing that true := 1 and false := 0, we can transform it as follows:

$$r = a \times (b \times c) + (1 - a) \times (b + c)$$

Where the witness vector is: $w = (r, a, b, c)$: $(6, 1, 2, 3)$, $(5, 0, 2, 3)$.

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:

$$
\begin{aligned}
r_1 &= b \times c \\
r_2 &= b + c \\
r_3 &= 1 - a \\
r_4 &= a \times r_1 \\
r_5 &= r_3 \times r_2 \\
r &= r_4 + r_5
\end{aligned}
$$

With witness vector: $w = (r, r_1, r_2, r_3, r_4, r_5, a, b, c)$: $(6, 6, 5, 0, 6, 0, 1, 2, 3)$.

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 0.2. Let $\mathbb{V}$ be a vector space over the field $\mathbb{F}$. An **inner product** on $\mathbb{V}$ is a function:

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

defined for $u, v \in \mathbb{V}$ as a linear combination:

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

With the following properties $\forall c \in \mathbb{F}, \forall u, v, w \in \mathbb{V}$:

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

Example. Let u, v are vectors over the real number $\mathbb{R}$, where

$$u = (1, 2, 3), \quad v = (2, 4, 3)$$

Then:

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

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 0.3. Each **constraint** in the Rank-1 Constraint System must be in the form:

$$\langle a, w \rangle \times \langle b, w \rangle = \langle c, w \rangle$$

Where w is a vector containing all the *input*, *output*, and *intermediate* variables involved in the computation. The vectors a, b, and c are vectors of coefficients corresponding to these variables, and they define the relationship between the linear combinations of 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: $w = (r, x_1, x_2)$, $a = (0, 1, 0)$, $b = (0, 0, 1)$, $c = (1, 0, 0)$. 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 w.

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

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 0.4. Given two vectors $u \in \mathbb{F}^n$, $v \in \mathbb{F}^m$ 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:

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

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

- Transpose: $(u \otimes v) = (v \otimes u)^{\mathsf{T}}$
- Distributivity: $u \otimes (v + w) = u \otimes v + u \otimes w$
- Scalar Multiplication: $c(v \otimes u) = (cv) \otimes u = v \otimes (cu)$
- Rank: the outer product $u \otimes v$ is a rank-1 matrix if u and v are non-zero vectors

Example. Let u, v are vectors over the real number $\mathbb{R}$, where

$$u = (1, 2, 3), \quad v = (2, 4, 3)$$

Then:

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

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

$$w^T A w + c^T w = 0$$

Where A is the outer product of vectors a, b (denoted as $a \otimes b$), consequently a **rank-1** matrix.

Lemma proof. Suppose there is a constraint $\langle a, w \rangle \times \langle b, w \rangle = \langle c, w \rangle$. Where vectors

$a, b, c, w \in \mathbb{F}^n$. 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:

$$w^T A w + c^T w = 0$$

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

0.4 Quadratic Arithmetic Program

The Rank-1 Constraint System is not succinct. The number of constraints directly depends on 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.