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.
- **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, so 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 or-

ganizer 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.

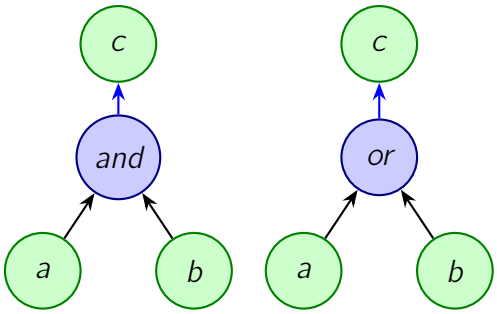

Figure 1: Boolean AND and OR Gates

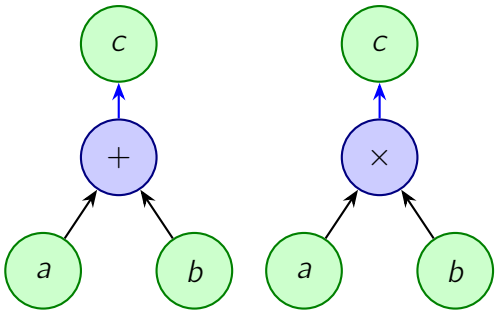

Figure 2: Addition and Multiplication Gates

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.

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

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

Table 1: AND Gate Truth Table

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:

$$r = a \times b \tag{1}$$

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:

$$r = a \times (b \times c) + (1 - a) \times (b + c) \tag{2}$$

$w = \{r, a, b, c\}$: $\{6, 1, 2, 3\}$, $\{5, 0, 2, 3\}$.

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:

$$\begin{aligned} r_1 &= b \times c \\ r_2 &= b + c \\ r_3 &= 1 - a \\ r_4 &= a \times r_1 \\ r_5 &= r_3 \times r_2 \\ r &= r_4 + r_5 \end{aligned} \tag{3}$$

$w = \{r, r_1, r_2, r_3, r_4, r_5, a, b, c\}$: $\{6, 6, 5, 0, 6, 0, 1, 2, 3\}$

0.3 Rank-1 Constraint System

For the ZK proof we need just a little bit different format - **Rank-1 Constrain System**. Where the simpliest term is **constraint**.