zk-SNARK

Distributed Lab

Sep 5, 2024

Plan

What is zk-SNARK?

What Is zk-SNARK?

Definition

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

What Is zk-SNARK?

Definition

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

- **Argument of Knowledge** — a proof that the prover knows the data (witness) that resolves a certain problem, and this knowledge can be "extracted".

What Is zk-SNARK?

Definition

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

- **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.

What Is zk-SNARK?

Definition

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

- **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.
- **Non-interactiveness** — to produce the proof, the prover does not need any interaction with the verifier.

What Is zk-SNARK?

Definition

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

- **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.
- **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.

Still didn't get who is Snark...

Well... Let's take a look at some example.

Still didn't get who is Snark...

Well... Let's take a look at some example.

Imagine you're part of a treasure hunt...

Still didn't get who is Snark...

Well... Let's take a look at some example.

Imagine you're part of a treasure hunt...

...and you've found a hidden treasure chest...

Still didn't get who is Snark...

Well... Let's take a look at some example.

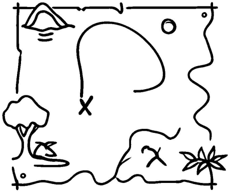
Imagine you're part of a treasure hunt...

...and you've found a hidden treasure chest...

...but how to prove that without revealing the chest location?

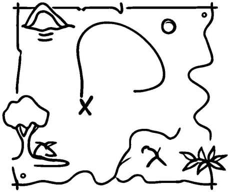
Still didn't get who is Snark...

The Problem: you have found a hidden treasure chest, and you want to prove to the organizer that you know its location without actually revealing that.



Still didn't get who is Snark...

The Problem: you have found a hidden treasure chest, and you want to prove to the organizer that you know its location without actually revealing that.



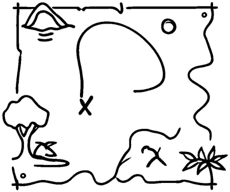
We can retrieve some information from that:

Question #81673

What is a secret data? Who is a prover and who is a verifier?

Still didn't get who is Snark...

The Problem: you have found a hidden treasure chest, and you want to prove to the organizer that you know its location without actually revealing that.



We can retrieve some information from that:

Question #81673

What is a secret data? Who is a prover and who is a verifier?

The Secret Data: the exact treasure location.

The Prover: you.

The Verifier: the treasure hunt organizer.

Ohh... Got it!

Here is how we can apply the zk-SNARK to our problem:

- Argument of Knowledge: You need to create a proof that demonstrates you know the chest is.

Ohh... Got it!

Here is how we can apply the zk-SNARK to our problem:

- Argument of Knowledge: You need to create a proof that demonstrates you know the chest is.
- 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.

Ohh... Got it!

Here is how we can apply the zk-SNARK to our problem:

- Argument of Knowledge: You need to create a proof that demonstrates you know the chest is.
- 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.
- Non-interactive: You don't need to have a back-and-forth conversation with the organizer to create this proof.

Ohh... Got it!

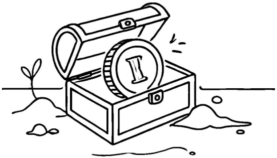
Here is how we can apply the zk-SNARK to our problem:

- Argument of Knowledge: You need to create a proof that demonstrates you know the chest is.
- 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.
- Non-interactive: You don't need to have a back-and-forth conversation with the organizer to create this proof.
- Zero-Knowledge: The proof doesn't reveal any information about the actual location of the treasure chest.

Ohh... Got it!

Here is how we can apply the zk-SNARK to our problem:

- Argument of Knowledge: You need to create a proof that demonstrates you know the chest is.
- 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.
- Non-interactive: You don't need to have a back-and-forth conversation with the organizer to create this proof.
- Zero-Knowledge: The proof doesn't reveal any information about the actual location of the treasure chest.

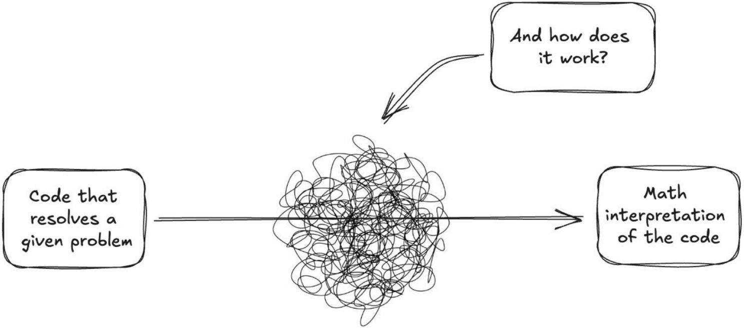


Well... The golden coin where the pirates' sign is engraved is our zk-SNARK proof!

But the problems that we usually want to solve are in a slightly different format.

But the problems that we usually want to solve are in a slightly different format.

When we need to prove that some element is in a merkle tree, we can't come to a verifier and give them a "coin"...



Arithmetic Circuits

The First Question To Resolve

The cryptographic tools we have learned in the previous lectures operate with numbers or certain primitives above them.

Question?

How do we convert a program into a mathematical language?

Do not forget about succintness!

Boolean Circuits

We can do that in a way like the computer does it - boolean circuits.

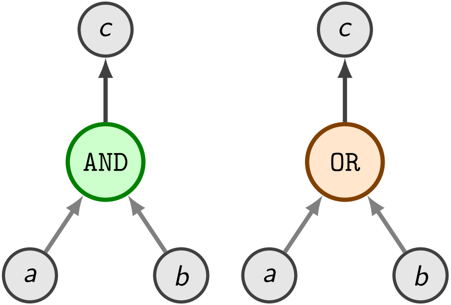


Figure: Boolean `AND` and `OR` Gates

Boolean Circuits

We can do that in a way like the computer does it - boolean circuits.

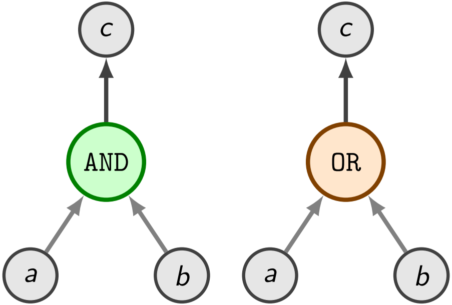


Figure: Boolean AND and OR Gates

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

Figure: AND Gate Truth Table

Note

With any of {AND, NOT} or {OR, NOT} gates sets one can build any possible logical circuit, they are called **functionally complete** sets.

Boolean Circuit Example

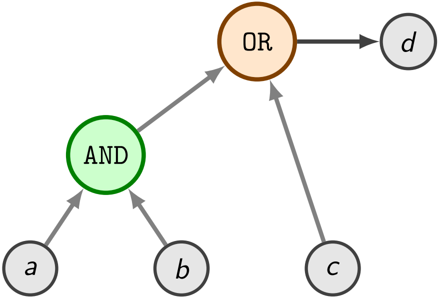


Figure: Example of a circuit evaluating $d = (a \texttt{ AND } b) \texttt{ OR } c$.

Boolean Circuit Example

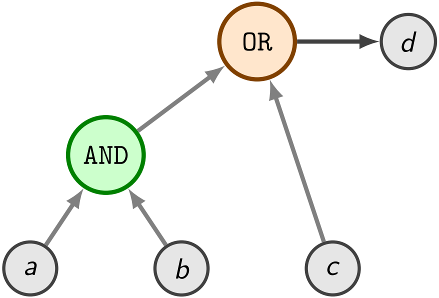


Figure: Example of a circuit evaluating $d = (a \text{ AND } b) \text{ OR } c$.

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.

The above circuit can be satisfied with the next values:

$$a = 1, \quad b = 1, \quad c = 0$$

SHA-256 Boolean circuit

File	No. ANDs	No. XORs	No. INVs
sha256Final.txt	22,272	91,780	2,194

Figure: Stats of a SHA256 boolean circuit implementation.

More than 100000 gates. Impressive, doesn't it?

But it also shows how inconvenient the boolean circuits are.

Arithmetic Circuits

Arithmetic Circuits

Similar to Boolean Circuits, the **Arithmetic circuits** consist of gates and wires.

- Wires: elements of some finite field $\mathbb{F}$.
- Gates: addition ($\oplus$) and multiplication ($\odot$) corresponding to the field.

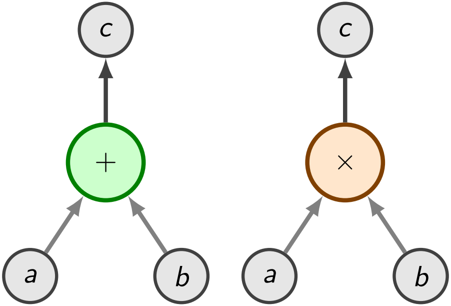


Figure: Addition and Multiplication Gates

Arithmetic Circuits Example I

Example

```
def multiply(a: F, b: F) -> F:
    return a * b
```

Arithmetic Circuits Example I

Example

```
def multiply(a: F, b: F) -> F:
    return a * b
```

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

$$r = a \times b$$

Arithmetic Circuits Example I

Example

```
def multiply(a: F, b: F) -> F:
    return a * b
```

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.

Arithmetic Circuits Example I

Example

```
def multiply(a: F, b: F) -> F:
    return a * b
```

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.

Note

We can think of the "$=$" in the gate as an assertion.

Arithmetic Circuits Example II

Example

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.

```
def evaluate(x1: F, x2: F) -> F:
    return x1**3 + x2**2
```

Looks easy, right? But the circuit is now much less trivial.

$$
\begin{aligned}
x_1^2 &= x_1 \times x_1 \\
x_1^3 &= x_1^2 \times x_1 \\
x_2^2 &= x_2 \times x_2 \\
Q &= x_1^3 + x_2^2
\end{aligned}
\quad \text{or} \quad
\begin{aligned}
r_1 &= x_1 \times x_1 \\
r_2 &= r_1 \times x_1 \\
r_3 &= x_2 \times x_2 \\
Q &= r_2 + r_3
\end{aligned}
$$

Arithmetic Circuits Example II

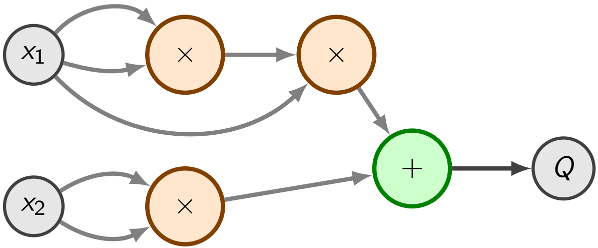


Figure: Example of a circuit evaluating $x_1^3 + x_2^2$.

Arithmetic Circuits Example III

Example

Well, it is quite clear how to represent any polynomial-like expressions. But how can we translate `if` statements?

```
def example(a: bool, b: F, c: F) -> F:
    if a:
        return b * c
    else:
        return b + c
```

Arithmetic Circuits Example III

Example

Well, it is quite clear how to represent any polynomial-like expressions. But how can we translate `if` statements?

```
def example(a: bool, b: F, c: F) -> F:
    if a:
        return b * c
    else:
        return b + c
```

We can transform such a function into the next expression:

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

Arithmetic Circuits Example III

Example

Well, it is quite clear how to represent any polynomial-like expressions. But how can we translate `if` statements?

```
def example(a: bool, b: F, c: F) -> F:
    if a:
        return b * c
    else:
        return b + c
```

We can transform such a function into the next expression:

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

Corresponding equations for the circuit are:

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

Arithmetic Circuits Example III

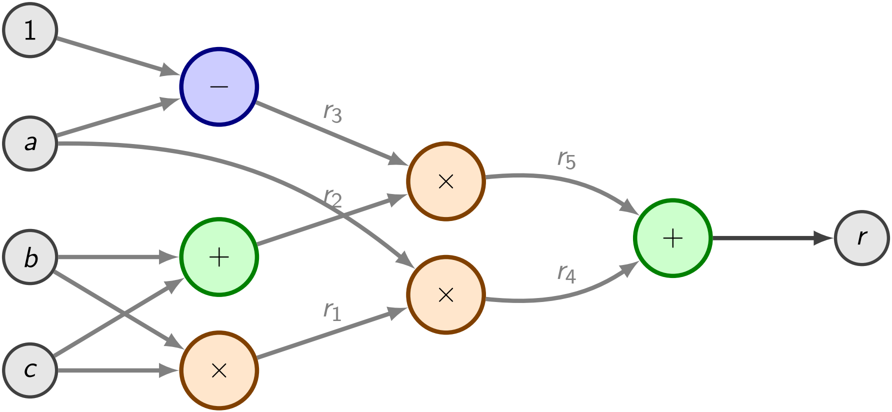


Figure: Example of a circuit evaluating the `if` statement logic.

Circuit Satisfability Problem

Definition

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.

Circuit Satisfability Problem

Definition

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 $1, x_1, x_2, \dots, x_n$. By $|\mathtt{C}|$ we denote the number of gates in the circuit.

Definition

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:

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

Rank-1 Constraint System

Inner Product

Definition

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**.

Dot Product

Definition

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:

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

Dot Product

Definition

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:

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

Note

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.

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:

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

Rank-1 Constraint System

With knowledge of the dot product of two vectors, we can now formulate a definition of the constraint in the context of the R1CS.

Definition

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.

Rationale Behind The Structure Of R1CS

Quadratic Arithmetic Program