

# Introduction to zk-SNARK. R1CS

Distributed Lab

Sep 12, 2024



# Plan

- 1 What is zk-SNARK?
- 2 Boolean Circuits
- 3 Arithmetic Circuits
- 4 Linear Algebra Preliminaries
- 5 Rank-1 Constraint System

# 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 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 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 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...

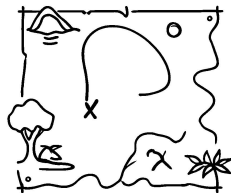
...and you've found a hidden treasure chest...



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

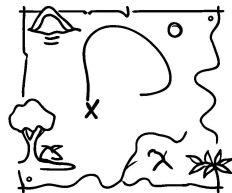
# Still didn't get who is Snark...

**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:

**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!

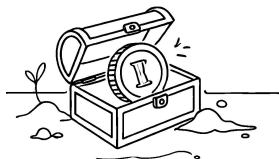
Here is how we can apply the zk-SNARK to our problem:

- 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!

# The First Question To Resolve

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

# The First Question To Resolve

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" ...

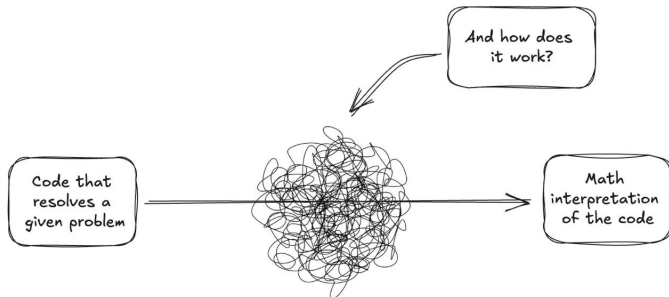
# The First Question To Resolve

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" ...

## Question?

How do we convert a program into a mathematical language?



# Boolean Circuits

# Boolean Circuits

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

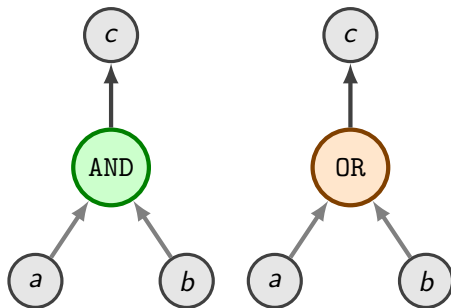


Figure: Boolean AND and OR Gates



# Boolean Circuits

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

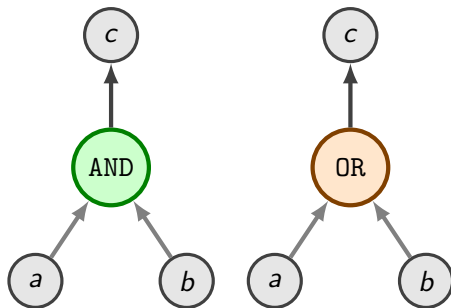


Figure: Boolean AND and OR Gates

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

# Boolean Circuit Example

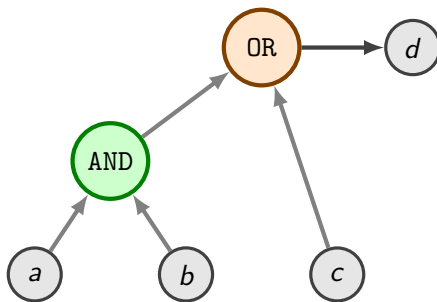


Figure: Example of a circuit evaluating  $d = (a \text{ AND } b) \text{ OR } c$ .

# Boolean Circuit Example

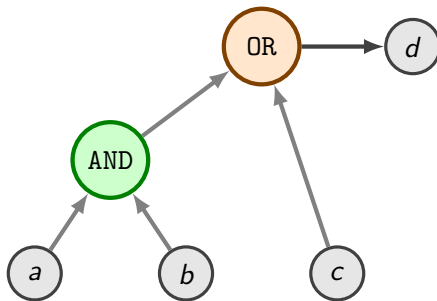


Figure: Example of a circuit evaluating  $d = (a \text{ AND } b) \text{ OR } c$ .

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
<a href="#">sha256Final.txt</a>	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.

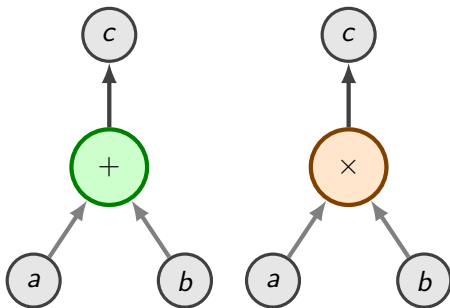


Figure: Addition and Multiplication Gates

# Arithmetic Circuits Example I

## Example

```
def multiply(a: F, b: F)  $\rightarrow$  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)  $\rightarrow$  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.

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

# Arithmetic Circuits Example II

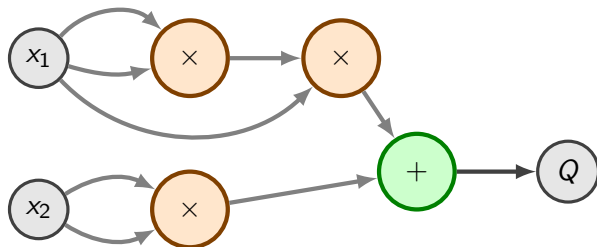


Figure: Example of a circuit evaluating  $x_1^3 + x_2^2$ .

# 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)  $\rightarrow$  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)  $\rightarrow$  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)  $\rightarrow$  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:

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

# Arithmetic Circuits Example III

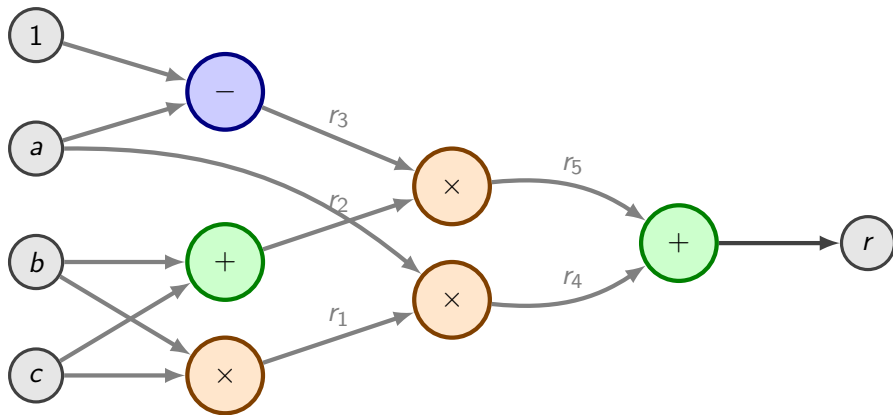


Figure: Example of a circuit evaluating the if statement logic.



# Circuit Satisfiability Problem

## Definition

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

# Circuit Satisfiability Problem

## Definition

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

## Definition

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

$$\mathcal{R}_C = \{(x, w) \in \mathbb{F}^n \times \mathbb{F}^m \mid C(x, w) = 0\},$$

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

# Linear Algebra Preliminaries

## Definition

A **vector space**  $V$  over the field  $\mathbb{F}$  is an abelian group for addition “+” together with a scalar multiplication operation “ $\cdot$ ” from  $\mathbb{F} \times V$  to  $V$ , sending  $(\lambda, x) \mapsto \lambda x$  and such that for any  $\mathbf{v}, \mathbf{u} \in V$  and  $\lambda, \mu \in \mathbb{F}$  we have:

- $\lambda(\mathbf{u} + \mathbf{v}) = \lambda\mathbf{u} + \lambda\mathbf{v}$
- $(\lambda + \mu)\mathbf{v} = \lambda\mathbf{v} + \mu\mathbf{v}$
- $(\lambda\mu)\mathbf{v} = \lambda(\mu\mathbf{v})$
- $1\mathbf{v} = \mathbf{v}$

Any element  $\mathbf{v} \in V$  is called a **vector**, and any element  $\lambda \in \mathbb{F}$  is called a **scalar**. We also mark vector elements in boldface.

# Matrix

The matrix is a rectangular array of numbers, symbols, or expressions, arranged in rows and columns. For example, the matrix  $A$  with  $m$  rows and  $n$  columns, consisting of elements from the finite field  $\mathbb{F}$  is denoted as  $A \in \mathbb{F}^{m \times n}$ .

## Definition

Let  $A, B$  be two matrices over the field  $\mathbb{F}$ . The following operations are defined:

- **Matrix addition/subtraction:**  $A \pm B = \{a_{i,j} \pm b_{i,j}\}_{i,j=1}^{m \times n}$ . The matrices  $A$  and  $B$  must have the same size  $m \times n$ .
- **Scalar multiplication:**  $\lambda A = \{\lambda a_{i,j}\}_{1 \leq i,j \leq n}$  for any  $\lambda \in \mathbb{F}$ .
- **Matrix multiplication:**  $C = AB$  is a matrix  $C \in \mathbb{F}^{m \times p}$  with elements  $c_{i,j} = \sum_{\ell=1}^n a_{i,\ell} b_{\ell,j}$ . The number of columns in  $A$  must be equal to the number of rows in  $B$ , that is  $A \in \mathbb{F}^{m \times n}$  and  $B \in \mathbb{F}^{n \times p}$ .

# Matrix Multiplication

## Example

Consider

$$A = \begin{bmatrix} 1 & 1 & 2 \\ 2 & 2 & 1 \end{bmatrix} \in \mathbb{R}^{2 \times 3}, \quad B = \begin{bmatrix} 2 & 1 \\ 1 & 3 \\ 1 & 1 \end{bmatrix} \in \mathbb{R}^{3 \times 2}$$

We cannot add  $A$  and  $B$  since they have different sizes. However, we can multiply them:

$$AB = \begin{bmatrix} 5 & 6 \\ 7 & 9 \end{bmatrix}, \quad BA = \begin{bmatrix} 4 & 4 & 5 \\ 7 & 7 & 5 \\ 3 & 3 & 3 \end{bmatrix}$$

To see why, for example, the upper left element of  $AB$  is 5, we can calculate it as  $\sum_{\ell=1}^3 a_{1,\ell} b_{\ell,1} = 1 \times 2 + 1 \times 1 + 2 \times 1 = 5$ .

# Vector As A Matrix

## Note

It just so happens that when working with vectors, we usually assume that they are **column vectors**. This means that the vector  $v = (v_1, v_2, \dots, v_n)$  is represented as a matrix:

$$\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \\ \vdots \\ v_n \end{bmatrix}$$

This is a common convention in linear algebra, and we will use it in the following sections.

# Matrix Transpose

## Definition (Transposition)

Given a matrix  $A \in \mathbb{F}^{m \times n}$ , the **transpose** of  $A$  is a matrix  $A^T \in \mathbb{F}^{n \times m}$  with elements  $A_{ij}^T = A_{ji}$ .

## Example

$$A = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \quad A^T = \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix}$$

$$B = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}, \quad B^T = \begin{bmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{bmatrix}$$

$$\mathbf{v} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \quad \mathbf{v}^T = [1, 2, 3]$$



# Inner Product

## Definition

Consider the vector space  $\mathbb{V}$  over the finite field  $\mathbb{F}_p$ . The **inner product** is a function  $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}_p$  satisfying the following conditions for all  $\mathbf{u}, \mathbf{v}, \mathbf{w} \in \mathbb{V}$ :

- $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$ .
- $\langle \mathbf{u}, \mathbf{v} + \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle + \langle \mathbf{u}, \mathbf{w} \rangle$ .
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$  for all  $\mathbf{u} \in \mathbb{V}$  iff  $\mathbf{v} = \mathbf{0}$ .
- $\langle \mathbf{u}, \mathbf{v} \rangle = 0$  for all  $\mathbf{v} \in \mathbb{V}$  iff  $\mathbf{u} = \mathbf{0}$ .

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

Consider the vector space  $\mathbb{V}$  over the finite field  $\mathbb{F}_p$ . The **dot product** on  $\mathbb{V}$  is a function  $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}$ , defined for every  $\mathbf{u}, \mathbf{v} \in \mathbb{V}$  as follows:

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

# Dot Product

## Definition

Consider the vector space  $\mathbb{V}$  over the finite field  $\mathbb{F}_p$ . The **dot product** on  $\mathbb{V}$  is a function  $\langle \cdot, \cdot \rangle : \mathbb{V} \times \mathbb{V} \rightarrow \mathbb{F}$ , defined for every  $\mathbf{u}, \mathbf{v} \in \mathbb{V}$  as follows:

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

## 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:

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

# Hadamard Product

## Definition

Suppose  $A, B \in \mathbb{F}^{m \times n}$ . The **Hadamard product**  $A \odot B$  gives a matrix  $C$  such that  $C_{i,j} = A_{i,j}B_{i,j}$ . Essentially, we multiply elements elementwise.

## Example

Consider  $A = \begin{bmatrix} 1 & 1 & 2 \\ 3 & 0 & 3 \end{bmatrix}$ ,  $B = \begin{bmatrix} 3 & 2 & 1 \\ 0 & 2 & 1 \end{bmatrix}$ . Then, the Hadamard product:

$$A \odot B = \begin{bmatrix} 1 \cdot 3 & 1 \cdot 2 & 2 \cdot 1 \\ 3 \cdot 0 & 0 \cdot 2 & 3 \cdot 1 \end{bmatrix} = \begin{bmatrix} 3 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}$$

# Outer Product

## Definition

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:

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

# Outer Product

## Lemma (Properties of outer product)

For any scalar  $c \in \mathbb{F}$  and  $(\mathbf{u}, \mathbf{v}, \mathbf{w}) \in \mathbb{F}^n \times \mathbb{F}^m \times \mathbb{F}^p$ :

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

# Outer 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:

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



# Outer 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:

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

The rows/columns number 2 and 3 in the result matrix can be represented as a linear combination of the first row/column, specifically by multiplying it by 2 and 3, respectively.

# Rank-1 Constraint System

# Constraint Definition

## 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.

# Constraint Example

## 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:

$$\mathbf{w} = (r, x_1, x_2), \quad \mathbf{a} = (0, 1, 0), \quad \mathbf{b} = (0, 0, 1), \quad \mathbf{c} = (1, 0, 0).$$

Therefore, our expression above reduces to:

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

$$w_2 \times w_3 = w_1$$

$$x_1 \times x_2 = r$$

# Constraint System Example

Now, let us consider a more complex example.

```
def r(x1: bool, x2: F, x3: F)  $\rightarrow$  F:  
    return x2 * x3 if x1 else x2 + x3
```

That can be expressed as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3)$$

# Constraint System Example

Now, let us consider a more complex example.

```
def r(x1: bool, x2: F, x3: F)  $\rightarrow$  F:  
    return x2 * x3 if x1 else x2 + x3
```

That can be expressed as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3)$$

We need a boolean restriction for  $x_1$ :

$$x_1 \times (1 - x_1) = 0$$

# Constraint System Example

Now, let us consider a more complex example.

```
def r(x1: bool, x2: F, x3: F) → F:  
    return x2 * x3 if x1 else x2 + x3
```

That can be expressed as:

$$r = x_1 \times (x_2 \times x_3) + (1 - x_1) \times (x_2 + x_3)$$

We need a boolean restriction for  $x_1$ :

$$x_1 \times (1 - x_1) = 0$$

Thus, the next constraints can be build:

$$x_1 \times x_1 = x_1 \quad (\text{binary check}) \quad (1)$$

$$x_2 \times x_3 = \text{mult} \quad (2)$$

$$x_1 \times \text{mult} = \text{selectMult} \quad (3)$$

$$(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \quad (4)$$

# Constraint System Example

The witness vector:  $\mathbf{w} = (1, r, x_1, x_2, x_3, \text{mult}, \text{selectMult})$ .

The coefficients vectors:

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

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  $(1 - x_1) \times (x_2 + x_3) = r - \text{selectMult} \quad (0 \times 7 = 12 - 12)$



# Why Rank-1?

## Lemma

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  $\mathbf{a}$ ,  $\mathbf{b}$ , so a **rank-1** matrix.

**Lemma proof.** Consider  $\mathbf{a}, \mathbf{b}, \mathbf{c}, \mathbf{w} \in \mathbb{F}^n$ .

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

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:

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

*Thanks for your attention!*