Zero Knowledge Proofs

High-level primer

@blockdeveth

Privacy & Scaling Explorations

June 20, 2024

Table of Contents

Related

Context: Zero-knowledge cryptography, Zero-knowledge proofs, privacy, scalability.

The Setup

- A function $y = f(x_1, x_2, ..., x_n)$.
- Some inputs are *private*: $\{x_1, ..., x_i\}$, rest are public $\{x_{i+1}, ..., x_n, y\}$.
- *Private* inputs mean that the input values are only known to the *Prover*.
- *Public* inputs mean that the *Prover* will send the input values to the *Verifier*.

The Goal

- *Prover* has to convince the *Verifier* that it has correctly computed the following while keeping $\{x_1, ..., x_i\}$ private:

$$y = f(x_1, ..., x_i, x_{i+1}, ..., x_n)$$

- In other words, *Verifier* has to be convinced that given $\{x_{i+1}, ..., x_n, y\}$:
  1. *Prover* knows values for $\{x_1, ..., x_i\}$ (witness) such that

$$y = f(x_1, ..., x_i, x_{i+1}, ..., x_n)$$

- *Verifier* should only learn about the truth of the statement, and nothing else (private inputs, intermediate values), hence Zero-knowledge.

Example

The Setup

- Our function: $y = \text{SHA}(x)$.
- *Private* input: x
- *Public* input: y

The Goal

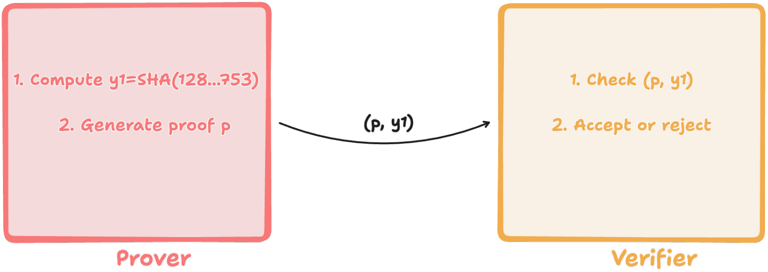
- *Verifier* has to be convinced that *Prover* knows some x such that $y = \text{SHA}(x)$.

One specific instance

- *Prover* computes $y_1 = \mathrm{SHA}(12834992849219753)$.
- It generates a proof p (a long string of bytes), and sends (p, y_1) to *Verifier*.
- *Verifier* runs its program on (p, y_1).
- *Verifier* is convinced iff the program returns `true`.

Application: Password verification. Server stores the password hash, and user generates a ZK-proof to prove it knows the related password.

One specific instance



1. Compute y1=SHA(128...753)
2. Generate proof p

Prover

(p, y1)

1. Check (p, y1)
2. Accept or reject

Verifier

zkEVM Example

The Setup

- Our function: `new_state = EVM(state, transaction)`.
- No *Private* input.
- All inputs are public.

The Goal

- *Verifier* has to be convinced that the transaction was executed correctly by the *Prover*.

Assumptions as application devs

- Verifier is honest.
- Prover is dishonest.

What is a ZK proof system?

Given a function $y = f({\color{red}x_1, \ldots, x_i}, x_{i+1}, \ldots, x_n)$, a ZK proof system defines how to write a Verifier program `V`, and a Prover program `P`:

- Honest prover can run `P` to generate proof, and send data to verifier.
- Verifier can input received data into `V`, and accept or reject the proof.
- Groth16, Plonk, STARK.

Example

- `V` can be deployed as a smart contract.
- `P` can be run on user devices, and the proof and public inputs can be sent with the transaction.

Properties of a ZK proof system

- **Completeness**: Prover can convince Verifier of true statements.
- **Soundness**: Malicious Prover cannot convince Verifier of false statements.
- **Zero-knowledge**: Verifier should not learn anything except the validity of the statement.
  - Enables privacy (e.g. MACI).
- Ideally **Succintness**: Small proof, fast to verify
  - Enables scaling (e.g. Rollups, Mina).

Parameters to evaluate ZK proof systems

- Proof size: Size (in B, KB) of the generated proof.
- Proof generation time: Time required to generate ZK proof by an honest prover.
- Proof verification time: Time required to verify the proof by an honest verifier.
- Quantum resistance: Does quantum computing break any property of the ZK proof system?
- Need for trusted setup.

Trusted Setup

Remember “The Setup” from before?

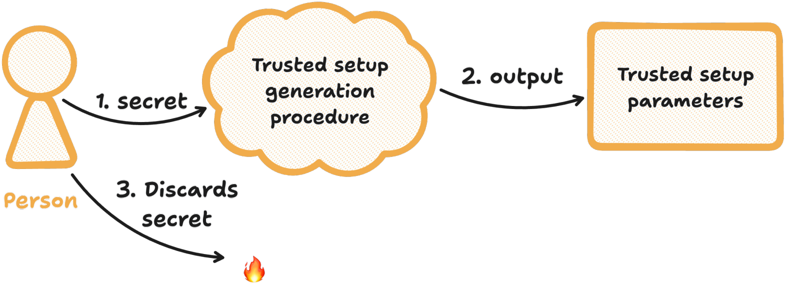
- You have a function $y = f(x_1, x_2, ..., x_n)$.
- Some inputs are *private*. Let’s say the first i inputs are private: $\{x_1, ..., x_i\}$ and the rest are public $\{x_{i+1}, ..., x_n, y\}$.
- *Private* inputs mean that the values of these inputs are only known to the *Prover*.
- *Public* inputs mean that the *Prover* will send the values of these inputs to the *Verifier*.

Trusted Setup

Remember “The Setup” from before?
Some ZK proof systems require pre-processing to generate parameters for the setup phase. These parameters are used every time the proof system protocol is run.

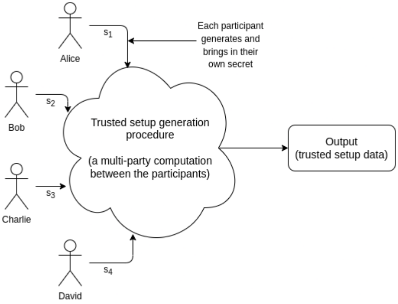
- These parameters are generated via *Trusted Setup ceremony*.
- A random secret number is required to generate these parameters.
- Access to this `secret` allows generating fake proofs and cheating the verifier.
- This `secret` has to be discarded after the trusted setup ceremony is complete. Hence, “toxic waste”.
- Hence the name “Trusted” setup ceremony, since we are trusting (or assuming) `secret` is discarded by the generator.

Trusted Setup



Distributed Trusted Setup ceremony

- What if `secret` isn't discarded? Hence, distributed ceremony.
- Multiple people individually generate `secret`, and all these values together generate the trusted setup parameter.
- If at least one person discards their secret, then noone can cheat the verifier.

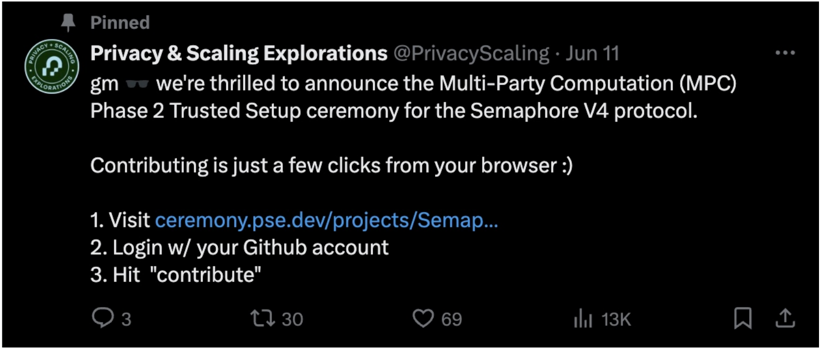


from Vitalik's post

Need for trusted setup

- STARKs do not need a trusted setup, and are quantum resistant.
- Plonk requires a “universal” trusted setup ceremony: output of one ceremoney can be used for setups.
- Protocols like Groth16 require circuit (function) specific ceremony: output of one ceremony isn’t usable if the setup is changed. However, it boasts of the smallest proof size and verification time.

Need for trusted setup



Pinned

Privacy & Scaling Explorations @PrivacyScaling · Jun 11

gm 🕶 we're thrilled to announce the Multi-Party Computation (MPC) Phase 2 Trusted Setup ceremony for the Semaphore V4 protocol.

Contributing is just a few clicks from your browser :)

1. Visit ceremony.pse.dev/projects/Semap...
2. Login w/ your Github account
3. Hit "contribute"

@PrivacyScaling

SNARK vs STARK

- S(uccint) N(on-interactive) AR(gument of) K(nowledge).
- S(calable) T(ransparent) AR(gument of) K(nowledge).

See 17 misconceptions about SNARKs.

From theory to application

- Underlying theory: ZK proof systems. Groth16, Plonk, STARK.
- Implementing theory: Frameworks or languages to generate prover and verifier program. Circom, Halo2, Plonky, Cairo.
- Applications: Write programs (the function f) in these langaguges: Dark forest, Rollups, zkVMs, Semaphore, MACI, zkEmail. These programs are also called circuits.

From theory to application

Stack
Applications written in frameworks
Frameworks or languages implementing proof systems
ZK proof systems

Feeling ZK!

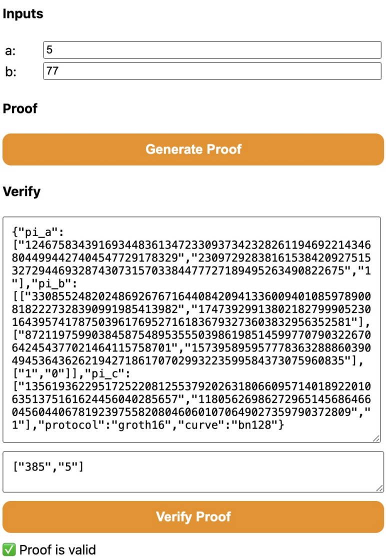
A Circom program:

```
template Example () {
    signal input x_1;
    signal input x_2;
    signal output y;

    y <== x_1 * x_2;
}

component main { public [ x_2 ] } = Example();
```

$y = f(x_1, \dots, x_i, x_{i+1} \dots, x_n)$: $y = x_1 * x_2$



Inputs

a: 5
b: 77

Proof

Verify

```
{"pi_a":
["1246758343916934483613472330937342328261194692214346
8044994427404547729178329","23097292838161538420927515
32729446932874307315703384477727189495263490822675","1
"],"pi_b":
[["330855248202486926767164408420941336009401085978900
8182227328390991985413982","17473929913802182799905230
164395741787503961769527161836793273603832956352581"],
["8721197599038458754895355503986198514599770790322670
642454377021464115758701","157395895957778363288860390
49453643626219427186170702993223599584373075960835"],
["1","0"]],"pi_c":
["1356193622951725220812553792026318066095714018922010
6351375161624456040285657","11805626986272965145686466
045604406781923975582080460601070649027359790372809","
1"],"protocol":"groth16","curve":"bn128"}
```

```
["385","5"]
```

✅ Proof is valid

Feeling ZK!

A Circom program:

```
template Example () {
    signal input x_1;
    signal input x_2;
    signal output y;

    y <-- x_1 * x_2;
    y === x_1 * x_2;
}
```

Mental Model to understand Circom circuits

A Circom program is split into two programs for verifier and honest Prover.
Prover program:

```
template Example () {
    ...
    y <-- x_1 * x_2; // compute y
}
```

Verifier program (just a mental model):

```
template Example () {
    ...
    y === x_1 * x_2; // verify y matches x_1 * x_2
}
```

Buggy code

A Circom program:

```
template Example () {
    signal input a;
    signal input b;
    signal output q;

    q <-- a \ b;
}

component main { public [ a ] } = Example();
```

Buggy code

Prover program:

```
template Example () {
    ...
    q <-- a\b; // compute q
}
```

Verifier program (just a mental model):

```
template Example () {
    ...
    // nothing: Any (a, b, q) is accepted
}
```

Scaling Ethereum

Validity Rollup: Starknet (STARK), Scroll (Halo2), zkSync (STARK then SNARK), Polygon zkEVM (STARK then Groth16).

- Uses ZK only for scalability.

ZK Rollup: Aztec.

- Uses ZK for privacy and scalability. Hence, can be called the true `ZK` rollup.

zkML

Proving execution of an AI model. Input can be kept public or private. Some use cases:

- To prove that a service provider has actually run the model.
- Proving you know the input for a particular output. You may not want to reveal what prompt you are using.
- It's currently impractical to generate a ZK proof for large models (Llama, GPT etc.).
- Modulus labs, Giza.

Private Voting: MACI

- **Correct execution**: Ensures counting process is correct.
- **Censorship resistance**: Anyone eligible to vote should be able to vote how they choose, and every vote should be counted.
- **Privacy**: you should not be able to tell which candidate someone specific voted for, or even if they voted at all.
- **Coercion resistance**: you should not be able to prove to someone else how you voted, even if you want to.
- MACI. Other PSE projects: `https://pse.dev/en/projects`.

Resources for application devs

- Circom docs: `https://docs.circom.io/`
- Circom course from 0xPARC: `https://learn.0xparc.org/`
- Rareskills ZK book:
  `https://www.rareskills.io/zk-book`,
  `https://www.rareskills.io/post/circom-tutorial`
- Tornado Cash 101: Mirror article

Resources for theory

- Why and How zk-SNARK Works: Definitive Explanation by Maksym Petkus.
- STARK @ home by StarkWare
- Proofs, Arguments, and Zero-Knowledge by Justin Thaler.
- Moonmath Manual by Least Authority