ZK Applications

rkm0959

Head of Security @ KALOS

March 2nd

Outline

Table of Contents

Tornado Cash: Introduction

Tornado Cash provides anonymous transactions for fixed size of ETH.

You deposit ETH with your public address, then you prove in ZK that you are one of the depositors, while not revealing which of the depositors.

When you withdraw, you give up some of the ETH as a gas fee for a *relayer*, so your recipient does not need to hold ETH.

Tornado Cash uses ZK

Tornado Cash is based on Groth16 + Powers of Tau.

- MPC Ceremony : https://ceremony.tornado.cash/
- MPC Ceremony Announcement
- Post MPC Ceremony Announcement
- Tornado Cash becomes trustless

zkSNARKs on Ethereum

From EIP196/197, EVM supports precompiled contracts for BN254 curve. This elliptic curve supports pairing operations and is suitable for FFT.

ETH Yellowpaper :
`https://ethereum.github.io/yellowpaper/paper.pdf`
EIP196 : `https://eips.ethereum.org/EIPS/eip-196`
EIP197 : `https://eips.ethereum.org/EIPS/eip-197`

Tornado Cash: Sketch of Internals

Prepare a **Merkle Tree** $\mathcal{T}$. We will do the following.

- **Deposit** : Add a hashed value as a leaf to $\mathcal{T}$ then recalculate the root
- **Withdraw** : Prove we know the preimage of the leaf and Merkle Proof

Tornado Cash: Details - Deposit

Let $H_1 : \{0,1\}^\star \to \mathbb{F}_p$ and $H_2 : (\mathbb{F}_p, \mathbb{F}_p) \to \mathbb{F}_p$ be hash functions.
Let $\mathcal{T}$ be a Merkle Tree of height 20 using H_2 as the hash.

To deposit N ETH, (where N is a supported amount of ETH in Tornado)

- Generate two random $k, r \in \{0,1\}^{248}$.
- Compute $H_1(k||r)$ and add this value as a leaf to $\mathcal{T}$

Note that $\mathcal{T}$ itself is public and computable from previous transactions.

Tornado Cash: Details - Withdraw

The goal is to prove the following statement in ZK.

Statement

The following values are public, i.e. directly sent as a part of transaction.

- R, the root of Merkle Tree $\mathcal{T}$
- h, the *nullifier hash*

We claim that we **know** $k, r,$ **Proof** where

- $h = H_1(k)$
- $H_1(k||r)$ exists as a leaf in $\mathcal{T}$
- **Proof** is a Merkle Proof for $H_1(k||r)$

Tornado Cash: Details - Withdraw

There are some very nice tricks here we have to discuss -

- **Preventing Double Spending** : The contract keeps track of $h = H_1(k)$ that was used for withdrawals, so double spending can be prevented unless some serious hash collisions occur
- **Preventing Frontrunning** : We can consider the case that an adversary in the mempool front-run your proofs. To prevent this, the circuit actually adds some dummy constraints on the recipient address, relayer address, and fee. These values are actually used during the ZK proof, so the proof is binding and cannot be front-run. To prevent frontrunners from stopping withdrawals via depositing and changing the Merkle Root, the contract stores 100 recent Merkle Roots.

On Groth16 Malleability

The dummy constraints and how they prevent malleability is a important topic in ZK security. We will take a deeper look here in a later lecture. For more details, consult the great article by Geometry Research.

https://geometryresearch.xyz/notebook/groth16-malleability

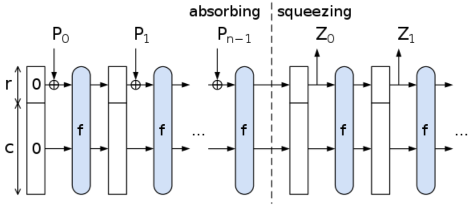
Tornado Cash: Choosing the Hash

H_1 is the Pedersen hash function (also used by dYdX for signatures!)
H_2 is the MiMC hash function via sponge + Feistel mode.

ABDK, the auditor, notes that H_1 is unoptimized for zkSNARKs.
H_2, on the other hand, is a hash specifically designed for zkSNARKS.

Optional: MiMC Hash

Sponge Construction is a method of creating a secure hash via pseudoranom permutation. It's also used by keccak256.



We use this with the MiMC permutation, which performs

$$(x, y) \to (y, x + (y + RK)^5)$$

sufficient times - this is where the name *Feistel* mode comes in.

Conclusion: Security

- No double spending (i.e. double withdrawals) is possible
- You must know both k, r to withdraw coins
- The proof is in ZK, so anonymity is preserved
- The proof is binding, so no frontrunning is possible
- 126-bit security, except the BN254 curve is now 100-bit secure

The Moral Character of Cryptographic Work

https://web.cs.ucdavis.edu/ rogaway/papers/moral-fn.pdf

Taken from Abstract

Cryptography rearranges power: it configures who can do what, from what. This makes cryptography an inherently political tool, and it confers on the field an intrinsically moral dimension. The Snowden revelations motivate a reassessment of the political and moral positioning of cryptography.

Table of Contents

Brainstorming

So far, our SNARKs can naturally handle some addition and multiplication over $\mathbb{F}_p$, with a prime p. However, this isn't what we usually use in our day to day programming in blockchains. What do we usually use, anyway?

Brainstorming Result

- 256-bit integer arithmetic
- Hashing Objects (SHA256, keccak256)
- Elliptic Curve (ECDSA, BLS, Pairings)

Let's take a brief look at each. These may not be state of the art.

Preparation

Circom (0xPARC): https://github.com/iden3/circomlib
Halo2 (Axiom): https://github.com/axiom-crypto/halo2-lib

Before that, we need to start with the basic of the basics.

- Zero Checking
- Range Checking
- Multiplexers

Zero Checking

To test if $a = 0$, set $b = a^{-1}$ if $a \neq 0$ and $b = 0$ if $a = 0$.
We add the constraints on t so that

$$at = 0, \quad t = 1 - ab$$

If $a = 0$, then immediately $t = 1$. If $a \neq 0$, then $t = 0$ immediately.

Range Checking

To test if $0 \leq a < 2^n$, we constrain

$$a = \sum_{i=0}^{n} b_i 2^i, \quad b_i(b_i - 1) = 0$$

This gives us b_i's as a bit representation, so bit operations can be done.

Since we now have lookup arguments as well, it can be used here.

Multiplexers

To select either a or b depending on whether c is 0 or 1,

$$r = ca + (1 - c)b$$

constraint gives us the result r accordingly. This can be generalized.

Integer Arithmetic: Brief Look

Circom (0xPARC): https://github.com/0xPARC/circom-ecdsa
Halo2 (Axiom): https://github.com/axiom-crypto/halo2-lib

The main issue is that we need to work with integers larger than p.
Therefore, we use three registers to handle 86 bits each.

Integer Arithmetic: Comparison & Addition & Subtraction

If we have two integers

$$a_0 + a_1 \cdot 2^{86} + a_2 \cdot 2^{172}, \quad b_0 + b_1 \cdot 2^{86} + b_2 \cdot 2^{172}$$

the comparison can be done by something like $a < b$ iff

$$(a_2 < b_2) \vee ((a_1 < b_1) \wedge (a_2 = b_2)) \vee ((a_0 < b_0) \wedge (a_1 = b_1) \wedge (a_2 = b_2))$$

For adding and subtracting, basic carry handling will work.

Integer Arithmetic: Multiplication

To compute the product of $\sum_{i=0}^{k-1} a_i N^i$ and $\sum_{i=0}^{k-1} b_i N^i$, first compute the polynomial multiplication result $\sum_{i=0}^{2k-2} c_i N^i$. Then, constrain

$$\sum_{i=0}^{k-1} a_i U^i \cdot \sum_{i=0}^{k-1} b_i U^i = \sum_{i=0}^{2k-2} c_i U^i$$

for some $2k-1$ different U's. This is enough due to polynomial properties.

Then, we handle the carries from the lowest chunk. This makes the representation "proper", in terms of base N representation.

Integer Arithmetic: Division

To compute a/b and $a \pmod{b}$, directly compute q, r and constrain

$$a = bq + r, \quad r < b$$

Hashing: Brief Look

This is *the* hard one - so hard that the **zkEVM meta-game** almost *revolves* around this. This is because keccak256 and SHA256 is very hard to implement in ZK. Therefore, replacements like MiMC, Poseidon and the like are one of the important sub-topics in ZK.

`github.com/rkm0959/rkm0959_presents/blob/main/ZKHash.pdf`

Quick Notes on zkEVM

https://vitalik.ca/general/2022/08/04/zkevm.html

- Type 1: Full ETH Equivalence
- Type 2: Full EVM Equivalence - but data structures are different
- Type 3: Nearly EVM Equivalent - but lacking a few stuff
- Type 4: High-Level Language Equivalent

The differences from Type 1 to Type 3 really revolve around hashes.

Elliptic Curve: Brief Look

It's a good mix of standard Elliptic Curve tricks and ZK circuit tricks.
Circom (Pairing): https://github.com/yi-sun/circom-pairing
Circom (ECDSA): https://github.com/0xPARC/circom-ecdsa
Halo2 (Axiom): https://github.com/axiom-crypto/halo2-lib

Here, $\mathbb{F}_p$ multiplication is hard as division now, so no Jacobian.
Circom-ECDSA uses a windowed method with cache for fixed-base case.

Table of Contents

Block Hash

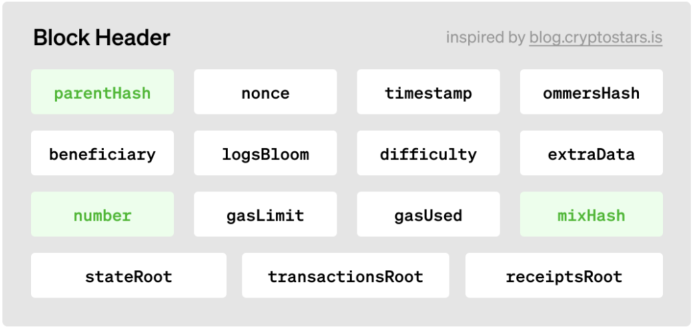
Our goal here will be to implement a *trustless* block hash oracle.

The easiest way to fetch a block hash is to use the BLOCKHASH opcode. However, the issue here is that you can only get the recent 256 blocks.

Therefore, the issue would be

- You need someone to call the oracle contract once every 256 blocks.
- You cannot get the block hash for older blocks trustlessly.

Block Hash


Block Header
inspired by blog.cryptostars.is
parentHash
nonce
timestamp
ommersHash
beneficiary
logsBloom
difficulty
extraData
number
gasLimit
gasUsed
mixHash
stateRoot
transactionsRoot
receiptsRoot

Block Hash via SNARKs

The power of SNARKs can definitely help here. With a carefully constructed circuit, one can prove 2^n consecutive block hashes at once, and output a merkle root along with its proof. The smart contract can verify that SNARK, and now use that merkle root freely.

This can be continued up to the very first block. Now a block hash can be easily used via submitting a merkle proof: we now have a block hash oracle.

Block Hash via SNARKs

The circuit design is of great importance for obvious reasons.

The currently optimal design choice seems to be a recursive, binary-tree like design. To prove the block hashes of blocks in range $[s, e)$, you

- Prove the block hashes of blocks in range $[s, m)$ and $[m, e)$.
- Take their proofs and outputs, i.e. first parent blockhash, last blockhash, and merkle root of the entire block range.
- Verify their proofs in a circuit, checks the "links", and outputs the first parent blockhash, last blockhash, and the total merkle root of $[s, e)$.

Table of Contents

Storage Proofs

The important data structures here are state trie and the storage trie.

- State Trie: A Merkle-Patricia Tree, which maps keccak256(address) to RLP([nonce, balance, storageRoot, codeHash])
- Storage Trie: A MPT which maps keccak256(slot) to RLP(value)

Storage Proofs

Therefore, we can do something like

- Prove State Root from Block Hash (Block Hash Formula)
- Prove Storage Root from State Root (MPT Proof)
- Prove Storage Value from Storage Root (MPT Proof)

Storage Proofs: Application

We now have *trustless* verification of a storage value of a given slot at a given address at *any block*. Generalization to code stored at an address should be possible as well. How can we use this in applications?

Storage Proofs: Application

Axiom is currently building this block hash / storage proof project.
They are doing an Early Partner Program, so go build there if you want.

There are some immediate applications -

- Fetch RANDAO value, although this isn't a storage proof really
- Fetch cumulative time-weighted price at a certain point, for TWAP computation. This method is checkpoint-free, which is great.

Table of Contents

Secure On-Chain Randomness

`https://www.paradigm.xyz/2023/01/eth-rng`

Secure on-chain randomness is a *very* important problem - solving it has great implications in Web3 gaming, NFTs, and practically every non-deterministic action going on inside the blockchain.

Secure On-Chain Randomness

We want the following properties from our randomness

- Unpredictable: No one knows the result before-hand
- Unbiased: No one cannot bias the result non-negligibly
- Verifiable: The generation of randomness can be verified
- Liveness: Can be fetched without any trust or privilege

So how do we generate randomness right now?

On-Chain Randomness - Current Options

Option 1: You generate it off-chain and send it on-chain.

- If you do this, then there isn't really a good reason to do it in Web3.

Option 2: You use the block hash.

- It can be manipulated somewhat via the block proposer.

On-Chain Randomness - Current Options

Option 3: You use the RANDAO value.
`eth2book.info/bellatrix/part2/building_blocks/randomness/`

- It gets mixed by many block proposers, so it's better.
- You need a high lookahead here. See EIP4399.

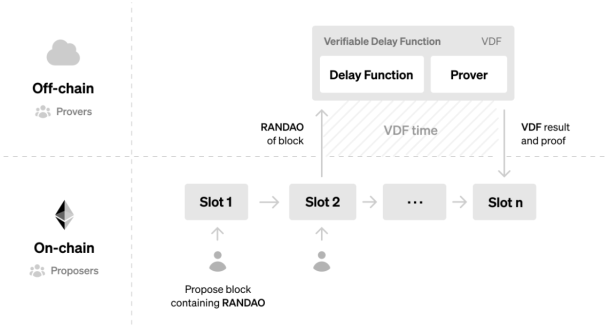
Option 4: You use Chainlink VRF.

- It's verifiable, and it's pretty random. It does cost you some LINK.
- Can be biased via request censorship. Liveness depends on operators.
- Issue may be fixed via implementing slashing. A solid option.

On-Chain Randomness - RANDAO + VDF

We now work on removing the bias from the RANDAO construction.

To do so, we apply a *Verifiable Delay Function* on the RANDAO value. This will make it impossible for the validators to actually compute the resulting random value in time. Therefore, no bias can be created.



Verifiable Delay Functions

A VDF consists of the following three algorithms -

- Setup: Takes difficulty t and security parameter λ, returns the keys.
- Eval: Takes input x, computes an output y and proof π.
- Verify: Takes x, y, π, returns whether the computation is correct.

VDF Properties

A VDF has the following properties -

- Correctness: If evaluated correctly, the proof verifies correctly.
- Soundness: If evaluated incorrectly, the proof doesn't verify.
- Sequentiality: (p, σ)-sequential, if you can't run Eval correctly in $\sigma(t)$ time with $p(t)$ processors. In another words, parallelization fails.

VDFs via Groups of Unknown Order [BBF18]

Take an unknown order group G, and make them evaluate

$$h = g^{2^T}$$

for some large T. Repeated squaring is the only way.

Two variants are popular. For details see [BBF18].

- Pietrzak's: Depends on the Low Order Assumption
- Wesolowski's: Depends on the Adaptive Root Assumption

VDFs via Groups of Unknown Order [BBF18]

Selecting the unknown order group G is also important.

- RSA Group: Requires a trusted setup, or extremely large keys.
- Class Group: Doesn't require a trusted setup, a good choice.
- Hyperelliptic Curve's Jacobian: Doesn't require a trusted setup.

For the latter two, note that square roots can be done easily.

VDFs via SNARKs on Iterative Computation

We first define a (t, ϵ)-sequential function.

Sequential Function

A (t, ϵ)-sequential function is $f : X \to Y$ such that

- For all $x \in X$, you can compute $f(x)$ in parallel time t, with $\text{poly}(\log t, \lambda)$ processors. Here, λ is the security parameter.
- Even with $\text{poly}(t, \lambda)$ processors, the probability you can compute $f(x)$ correctly for a random $x \in X$ in $(1 - \epsilon)t$ time is negligible.

VDFs via SNARKs on Iterative Computation

We then define an iterative sequential function.

Iterative Sequential Function

Let $g : X \to X$ be a (t, ϵ)-sequential function.
An $f(k, x) = g^{(k)}(x)$ is an iterative sequential function if for all $k = 2^{o(\lambda)}$, $h(x) = f(k, x)$ is a $(k \cdot t, \epsilon)$-sequential function as defined before.

VDFs via SNARKs on Iterative Computation

Assume $f(k, x)$ runs in time t, and its SNARK prover runs in time αt with some reasonable parallelism. We'll prove our $f(k, x)$ as follows.

- Calculate $f(k, x)$ for $\frac{t}{\alpha+1}$ time.
- Start to prove that amount of computation in parallel.
- Repeat the same for the $\frac{1}{\alpha+1}$ of the remaining computation.
- After l steps, $\left(\frac{\alpha}{\alpha+1}\right)^l$ of the computation remains.
- Therefore, use $n = \log_{1+1/\alpha}(k)$ steps to prove the entire computation.

To verify, simply SNARK verify the each chunk.

VDFs via SNARKs on Iterative Computation

This can be shown to be a VDF.

- Soundness: SNARK verification is done, so it's sound.
- Sequentiality: Due to parallelism, the time spent is equal to the time computing $f(k, x)$ - and due to iterative sequentiality of f, the proof is done. Note that the parallelism used here is poly-logarithmic.

MinRoot

`https://github.com/mmaller/vdf_snark`
Now we need a candidate for the iterative function. An immediate one is hashing, but that's way too painful. A better option is MinRoot.

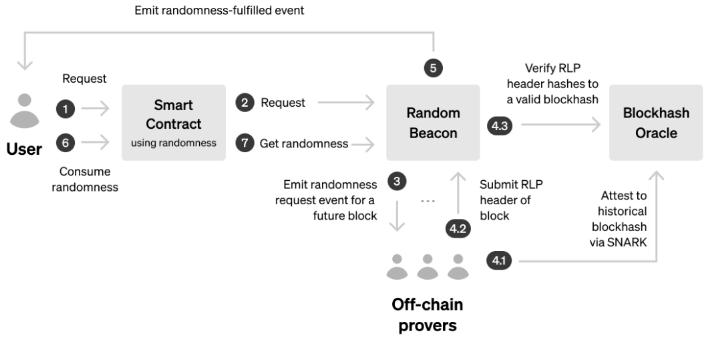
MinRoot Round Function

We work with the round function

$$(x_{i+1}, y_{i+1}) \leftarrow ((x_i + y_i)^{(2p-1)/5}, x_i + i)$$

The goal is to keep a low-degree polynomial relation $G(S, S') = 0$.

On-Chain Randomness: The End-Game


Emit randomness-fulfilled event
5
Request
1
User
6
Consume randomness
Smart Contract
using randomness
2 Request
7 Get randomness
Random Beacon
Verify RLP header hashes to a valid blockhash
4.3
Blockhash Oracle
Emit randomness request event for a future block
3
...
Submit RLP header of block
4.2
Attest to historical blockhash via SNARK
4.1
Off-chain provers