ZK Applications

rkm0959

Head of Security @ KALOS

April 13th

Outline

Interesting News

KALOS is diving into ZKP Security! Announcements will follow.

- We'll be collaborating with a top-tier audit firm.
- Our first client will be a zkEVM team that we all know.

The ZKP audit taskforce is **very** strong - if you are a team that builds on ZK, contact us. We'll be happy to make friends and keep in touch!

If you are qualified, we want you to join us as a ZKP Security Researcher.

ZKP: A Brief Summary

The goal is to provide a proof of a statement. To do so,

- Circuit devs cook up a *constraint system* with a DSL that corresponds to the big statement. They also provide the *witness* for the proof.
- The *prover code* takes all the witnesses and outputs a succint proof. This can be thought as a *cryptographic backend*.

Initial Classification of Bugs

Therefore, we can initially divide the class of bugs -

- Bugs in the Constraint System
- Bugs in the Cryptographic Side

The former would be more interesting for circuit devs, while the latter will be more mathematical and theoretical in nature. We'll look at both.

Table of Contents

Assignments vs Constraints

You have to **actually add the constraints**. If you forget a constraint, then an invalid proof can be verified to be correct, a critical bug.

In **Circom**, assignment to a variable is different from adding constraints.

```
pragma circom 2.0.0;

template IsZero() {
    signal input in;
    signal output out;
    signal inv;
    inv <-- in!=0 ? 1/in : 0;
    out <== -in*inv +1;
    in*out === 0;
}

component main {public [in]}= IsZero();
```

```
template IsZero() {
    signal in;
    signal out;
    signal temp;
    temp <-- in!= 0 ? 0 : 1;
    out === temp;
}
```

Assignments vs Constraints

In **Halo2**, *constraint configuration* and *witness assignment* are done on separate functions. This makes it very easy to get confused.

The verification logic depends on the built constraint system. An adversary may create a proof while not strictly following the witness assignment logic provided in the prover code. **Constraint System** determines all.

Examples: MiMC Hash Vulnerability in Tornado Cash

https://github.com/iden3/circomlib/pull/22/files

```
-   outs[0] = S[nInputs - 1].xL_out;
+   outs[0] <== S[nInputs - 1].xL_out;
```

Self-explanatory bug - the buggy version allows arbitrary *outs*[0].

Since this is a bug in the hash that allows arbitrary hash results, a fake merkle proof can be forged, leading to a complete asset theft.

Examples: SHL/SHR Opcode in PSE zkEVM

Issue #1124 in PSE's zkEVM.

```rust
/// ShlShrGadget verifies opcode SHL and SHR.
/// For SHL, verify pop1 * (2^pop2) % 2^256 == push;
/// For SHR, verify pop1 / (2^pop2) % 2^256 == push;
/// when pop1, pop2, push are 256-bit words.
#[derive(Clone, Debug)]
pub(crate) struct ShlShrGadget<F> {
    same_context: SameContextGadget<F>,
    quotient: util::Word<F>,
    divisor: util::Word<F>,
    remainder: util::Word<F>,
    dividend: util::Word<F>,
    /// Shift word
    shift: util::Word<F>,
    /// First byte of shift word
    shf0: Cell<F>,
    /// Gadget that verifies quotient * divisor + remainder = dividend
    mul_add_words: MulAddWordsGadget<F>,
    /// Check if divisor is zero
    divisor_is_zero: IsZeroGadget<F>,
    /// Check if remainder is zero
    remainder_is_zero: IsZeroGadget<F>,
    /// Check if remainder < divisor when divisor != 0
    remainder_lt_divisor: LtWordGadget<F>,
}
```

Examples: SHL/SHR Opcode in PSE zkEVM

The idea between *shift* and *shf*0 is that only the last byte of *shift* matters - if *shift* ≥ 256 then the result of SHL/SHR will be zero anyway.

The issue here is that the fact that *shf*0 is the last byte of *shift* is only *assigned*, not *constrained*. This leads to a full break of SHL/SHR opcodes.

```
cb.require_zero(
    "shf0 == shift.cells[0]",
    shf0.expr() - shift.cells[0].expr(),
);
```

Under-constrained Circuits

Of course, there are bugs where the circuit dev does not add sufficient constraints to enforce the intended checks. This is another big vuln class.

To catch these types of vulnerabilities, a complete understanding of the business logic and required constraint system is required.

Examples: Circom-Pairing Circuit Output

PR #21 on circom-pairing. Here, the BigLessThan circuit is being utilized, but the output is never constrained. Therefore, no checks are being done.

```
component lt[10];
// check all len k input arrays are correctly formatted bigints < q (BigLessThan calls Num2Bits)
for(var i=0; i<10; i++){
    lt[i] = BigLessThan(n, k);
    for(var idx=0; idx<k; idx++)
        lt[i].b[idx] <== q[idx];
}
for(var idx=0; idx<k; idx++){
    lt[0].a[idx] <== pubkey[0][idx];
    lt[1].a[idx] <== pubkey[1][idx];
    lt[2].a[idx] <== signature[0][0][idx];
    lt[3].a[idx] <== signature[0][1][idx];
    lt[4].a[idx] <== signature[1][0][idx];
    lt[5].a[idx] <== signature[1][1][idx];
    lt[6].a[idx] <== hash[0][0][idx];
    lt[7].a[idx] <== hash[0][1][idx];
    lt[8].a[idx] <== hash[1][0][idx];
    lt[9].a[idx] <== hash[1][1][idx];
}

}
var r = 0;
for(var i=0; i<10; i++){
    r += lt[i].out;
}
r === 10;
```

Examples: Aztec Merkle Root Position

`https://medium.com/aztec-protocol/vulnerabilities-found-in-aztec-2-0-9b80c8bf416c`

Basically, there is a Merkle tree that is being utilized in the protocol. A user was supposed to add new elements to it at the very next leaf node - but there was no constraints on this leaf position. This could've been a Denial of Service vulnerability, as then no one could've done any transactions.

Range Checks & Bit Length Checks

A very common vulnerability case is a missing range check. This includes various bit length checks, overflows and underflows as well.

It is always important to note that **every computation is done** in $\mathbb{F}_p$.

Due to this fact, many protocols require a "range check" - a constraint that a certain variable is in $[a, b]$. Missing this could be a vuln.

Examples: Dark Forest RangeProof

Consider a circuit that constrains $|input| \leq MAX$.

```
// From darkforest-v0.3/circuits/range_proof/circuit.circom
template RangeProof(bits, max_abs_value) {
    signal input in;

    component lowerBound = LessThan(bits);
    component upperBound = LessThan(bits);

    lowerBound.in[0] <== max_abs_value + in;
    lowerBound.in[1] <== 0;
    lowerBound.out === 0

    upperBound.in[0] <== 2 * max_abs_value;
    upperBound.in[1] <== max_abs_value + in;
    upperBound.out === 0
}
```

Examples: Dark Forest RangeProof

The issue here is that the circuit LessThan has a parameter n in which both inputs are *assumed*, not *constrained* to be in n bits.

```
// From circomlib/circuits/comparators.circom
template LessThan(n) {
    assert(n <= 252);
    signal input in[2];
    signal output out;

    component n2b = Num2Bits(n+1);

    n2b.in <== in[0]+ (1<<n) - in[1];

    out <== 1-n2b.out[n];
}
```

Examples: Dark Forest RangeProof

Since *input* and *MAX* are never range checked, there are no constraints on the bit lengths of these values. This needs to be added to keep soundness.

```
template RangeProof(bits) {
    signal input in;
    signal input max_abs_value;

    /* check that both max and abs(in) are expressible in `bits` bits  */
    component n2b1 = Num2Bits(bits+1);
    n2b1.in <== in + (1 << bits);
    component n2b2 = Num2Bits(bits);
    n2b2.in <== max_abs_value;

    /* check that in + max is between 0 and 2*max */
    component lowerBound = LessThan(bits+1);
    component upperBound = LessThan(bits+1);

    lowerBound.in[0] <== max_abs_value + in;
    lowerBound.in[1] <== 0;
    lowerBound.out === 0

    upperBound.in[0] <== 2 * max_abs_value;
    upperBound.in[1] <== max_abs_value + in;
    upperBound.out === 0
}
```

Examples: Semaphore On-Chain Range Check

There are also vulnerabilities arising from the fact that p - the field we are working with in zkSNARKs, are usually less than 2^{256}.

In many cases, the verifier will constrain the public inputs to be within $[0, p)$. This means that relevant parameters must be validated to be in this range as well - instead of the entire uint256 range.

Examples: Semaphore On-Chain Range Check

Semaphore Issue #90.

For example, in Semaphore, the Group ID was a public input that was a uint256 variable on-chain. However, there was no validation that this was in $[0, p)$ - and if it was set to be larger than p, no proofs will be verified.

```
function _createGroup(
  uint256 groupId,
  uint8 depth,
  uint256 zeroValue
) internal virtual {
  require(groupId < SNARK_SCALAR_FIELD, "SemaphoreGroups: group id must be < SNARK_SCALAR_FIELD");
  require(getDepth(groupId) == 0, "SemaphoreGroups: group already exists");
```

Examples: Modulo Gadget in PSE zkEVM

Issue #996 in PSE zkEVM.

The goal of the modulo circuit is to show that $a \pmod{n} = r$, with $r = 0$ if $n = 0$. To do so, the prover supplies the witness k and shows

$$a = kn + r$$

However, the gadget used actually showed

$$a = kn + r \pmod{2^{256}}$$

which is fine since all values are within $[0, 2^{256})$.

Examples: Modulo Gadget in PSE zkEVM

However, the key idea is that while this doesn't matter in *witness generation*, this surely matters in *constraint system design*.

This is because $kn + r < 2^{256}$ is a constraint that is needed, but never actually added. If $kn + r \geq 2^{256}$, then $a \equiv kn + r \pmod{2^{256}}$ is a completely different statement from $a = kn + r$, which is what we want.

For a proof of concept, consider $n = 3, k = 2^{255}, r = 0, a = 2^{255}$.

```
// Constrain k * n + r no overflow
cb.add_constraint("overflow == 0 for k * n + r", mul_add_words.overflow());
```

Examples: Aztec Emulated Field Operations

We perform a modular multiplication in $\mathbb{F}_q$, where we natively work in $\mathbb{F}_p$.

To do so, the Aztec team wrote

$$a \cdot b = u \cdot q + r$$

and checked that each equation holds over $\pmod{p}$ and $\pmod{2^t}$.
Then, a range check to assert each sides are less than $M = p \cdot 2^t$ was done.

These constraints are enough to constrain $a \cdot b = u \cdot q + r$ via CRT.
However, $u \cdot q + r < M$ was never properly constrained. ($50K Bounty)

Issues with Nullifiers

Recall the Tornado Cash system design.

Nullifiers help ZK systems prevent double-spending while keeping anonymity. The ability to re-use such nullifiers are equivalent to a double spending, which is critical. A *unique* nullifier for use should be designed.

Examples: Aztec Nullifier Reuse via Missing Range Check

`https://hackmd.io/@aztec-network/disclosure-of-recent-vulnerabilities`

The computation of nullifier for Aztec uses an index, which is assumed to be a 32-bit integer. However, this range check was never actually done - so one could change the other bits of the index and re-use a note with a different nullifier. This leads to a double spending, practically a theft.

Examples: zkDrops Nullifier Reuse via Missing Range Check

PR #2 on zkDrops

If the nullifier is an element of $\mathbb{F}_p$, then the actual value on-chain should also be constrained to be in $[0, p)$. If not, then both x and $x + p$ can fit into uint256 and be a valid nullifier, but a distinct one.

```
function collectAirdrop(bytes calldata proof, bytes32 nullifierHash) public {
    require(uint256(nullifierHash) < SNARK_FIELD ,"Nullifier is not within the field");
    require(!nullifierSpent[nullifierHash], "Airdrop already redeemed");

    uint[] memory pubSignals = new uint[](3);
    pubSignals[0] = uint256(root);
    pubSignals[1] = uint256(nullifierHash);
    pubSignals[2] = uint256(uint160(msg.sender));
    require(verifier.verifyProof(proof, pubSignals), "Proof verification failed");

    nullifierSpent[nullifierHash] = true;
    airdropToken.transfer(msg.sender, amountPerRedemption);
}
```

Examples: StealthDrop ECDSA Nullifier

`https://twitter.com/0xPARC/status/1493705025385906179`

It's important to understand that nullifiers should be unique and cannot be spent twice. In 0xPARC's StealthDrop, an ECDSA signature was used as a nullifier. This is bad, as ECDSA signatures are not unique by design - a nonce can be used to create many valid signatures.

PLUME: Unique ECDSA Nullifier for zkSNARKs

https://blog.aayushg.com/posts/nullifier/
https://eprint.iacr.org/2022/1255.pdf

Aayush Gupta proposed a ECDSA-based nullifier scheme that satisfies

- uniqueness, deterministic
- verifiable with public key
- non-interactive

The posts above are fascinating and a recommended read.

Dynamic Lookup Tables

There are cases where a *dynamic* lookup tables are utilized. In other words, a lookup argument can look up at the **witnessed values** which can definitely change per instances. This is still an unstable feature.

Here, the constraint system needs to note that the dynamic advice columns to be used as a lookup table should be constrained very heavily - an unconstrained row can be filled with malicious values and then looked up.

Dynamic Lookup Tables

Solutions for this are still being discussed. Some of them are

- adding more tags for dynamic lookups
- force at least one column to be a fixed one
- enforcing selector/column lengths

See halo2's PR #715, PSE zkEVM's Issue #866 for discussions & dev.

Table of Contents

Groth16 Malleability

`https://geometry.xyz/notebook/groth16-malleability`

The big idea is to **attach a public input** to a proof. In other words, we want to make it impossible to modify a proof (without knowing the private witnesses) and modify the public input along with it.

- In other words, we want to prevent front-running in Tornado Cash.

Groth16 Malleability - Tornado Cash

Tornado Cash adds "dummy constraints" on public inputs to keep them involved in the constraint system. This makes it cryptographically difficult to change the public inputs - so front-running attacks will not work.

```
// Add hidden signals to make sure that tampering with recipient or fee will invalidate the snark proof
// Most likely it is not required, but it's better to stay on the safe side and it only takes 2 constraints
// Squares are used to prevent optimizer from removing those constraints
signal recipientSquare;
signal feeSquare;
signal relayerSquare;
signal refundSquare;
recipientSquare <== recipient * recipient;
feeSquare <== fee * fee;
relayerSquare <== relayer * relayer;
refundSquare <== refund * refund;
```

Groth16 Malleability - A Theoretical Look

We have the following values as CRS - here, $E = (E_1, E_2)$.

- $E_1(\alpha), E(\beta), E_2(\gamma), E(\delta)$
- $E(x^i)$ for $0 \leq i < n$
- $E_1\left(\frac{\beta u_i(x)+\alpha v_i(x)+w_i(x)}{\gamma}\right)$ for $0 \leq i \leq l$
- $E_1\left(\frac{\beta u_i(x)+\alpha v_i(x)+w_i(x)}{\delta}\right)$ for $l+1 \leq i \leq m$
- $E_1\left(\frac{x^i t(x)}{\delta}\right)$ for $0 \leq i \leq n-2$

Groth16 Malleability - A Theoretical Look

Assume we know $a_1, \cdots, a_m$. Our proof consists of

$$E_1(A) = E_1(\alpha) + \sum_{i=0}^{m} a_i E_1(u_i(x)) + rE_1(\delta)$$

$$E_2(B) = E_2(\beta) + \sum_{i=0}^{m} a_i E_2(v_i(x)) + sE_2(\delta)$$

$$E_1(C) = \sum_{i=l+1}^{m} a_i E_1\left(\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\delta}\right) + sE_1(A)$$
$$+ rE_1(B) - rsE_1(\delta) + E_1\left(\frac{h(x)t(x)}{\delta}\right)$$

with r, s selected random values from $\mathbb{F}$.

Groth16 Malleability - A Theoretical Look

The verifier, given $E_1(A), E_2(B), E_1(C)$, checks

$$e(E_1(A), E_2(B)) = e(E_1(\alpha), E_2(\beta)) + \sum_{i=0}^{l} e\left(E_1\left(\frac{\beta u_i(x) + \alpha v_i(x) + w_i(x)}{\gamma}\right), a_i E_2(\gamma)\right) + e(E_1(C), E_2(\delta))$$

Since we are in an attacker perspective, we focus on the verifier logic.
We want to modify A, B, C so that we can modify a public input a_i.
Of course, the proof should still pass the verification logic.

Groth16 Malleability - A Theoretical Look

First, the easy case, where a public input is not used even once.

If ith public input is not used at all, $u_i(x) = v_i(x) = w_i(x) = 0$ - so changing a_i doesn't matter in the verification logic at all.

Therefore, the exact same proof can be used with a different public input.

Groth16 Malleability - A Theoretical Look

https://eprint.iacr.org/2020/811.pdf.

Theorem 1. *Assume that* $\{u_i(x)\}_{i=0}^{l}$ *are linearly independent and* $\text{Span}\{u_i(x)\}_{i=0}^{l} \cap \text{Span}\{u_i(x)\}_{i=l+1}^{m} = \emptyset$. *Then* ***Groth16*** *achieves weak white-box SE against algebraic adversaries under the* $(2n-1, n-1)$***-dlog*** *assumption.*

Definition 1 (White-box Weak Simulation-Extractability, [KZM+15]). *We say that* NIZK *is* white-box weak SE *if for any PPT adversary* $\mathcal{A}$ *there exists a polynomial time extractor* $\mathcal{X}_{\mathcal{A}}$ *such that for* $\mathcal{R}_\lambda$,

$$\Pr\left[\begin{array}{c}(\boldsymbol{\sigma},\tau)\leftarrow \mathsf{Setup}(\mathcal{R}_\lambda);(\phi,\pi)\leftarrow \mathcal{A}^{\mathcal{S}_{\boldsymbol{\sigma},\tau}}(\boldsymbol{\sigma}); \\ w\leftarrow \mathcal{X}_{\mathcal{A}}(\mathsf{trans}_{\mathcal{A}})\end{array} : \begin{array}{c}\mathsf{Verify}(\boldsymbol{\sigma},\phi,\pi)=1\wedge \\ (\phi,w)\notin\mathcal{R}_\lambda\wedge\phi\notin Q\end{array}\right]=\text{negl}(\lambda),$$

where $\mathcal{S}_{\boldsymbol{\sigma},\tau}(\phi)$ *is a simulator oracle that calls* $\mathsf{Sim}(\boldsymbol{\sigma},\tau,\phi)$ *internally, and also records* ϕ *into* Q.

Groth16 Malleability - A Theoretical Look

The solution is to add $a_i \cdot 0 = 0$ on public inputs.

If the kth constraint is $a_i \cdot 0 = 0$, then $u_i(x) = L_k(x), v_i(x) = w_i(x) = 0$.
As the Lagrange polynomials form a basis, linear independence holds.

Of course, Tornado Cash's idea works as well.

Groth16 Malleability - Defenses

A defense is usually embedded in the prover by adding dummy constraints on public inputs. The below example is taken from SnarkJS.

```
for (let s = 0; s <= nPublic ; s++) {
    const l1t = TAU_G1;
    const l1 = sG1*(r1cs.nConstraints + s);
    const l2t = BETATAU_G1;
    const l2 = sG1*(r1cs.nConstraints + s);
    if (typeof A[s] === "undefined") A[s] = [];
    A[s].push([l1t, l1, -1]);
    if (typeof IC[s] === "undefined") IC[s] = [];
    IC[s].push([l2t, l2, -1]);
    coefs.push([0, r1cs.nConstraints + s, s, -1]);
}
```

Malleability in General

Malleability is a big concept in cryptography in general - and theoretical developments on it over various ZKP protocols are being done.

`https://eprint.iacr.org/2021/1393.pdf`
For example, in the Algebraic Group Model, it is shown that the BulletProofs used with the Fiat-Shamir heuristic is non-malleable.

Frozen Heart Vulnerability

The interactive variant of ZKP requires a verifier to send in random field element for the proof system to have soundness. Of course, this is not desirable in terms of practicality, so a Fiat-Shamir transform is used.

Here, the entire transcript must be hashed to compute the "random field element" - with the Random Oracle Model, this will keep the protocol sound without interactivity. What if the entire transcript is not used?

Frozen Heart Vulnerability

If the transcript so far is a, b, c, then the "random field element" via Fiat-Shamir should be $H(a, b, c)$ with a secure hash H.

The rationale here is that modifying even one of a, b, c will completely change $H(a, b, c)$ like a "random oracle" so the attacker can't control it.

Frozen Heart Vulnerability

If the element is $H(a, c)$, for example, then the situation may change.
Here, we can change b as we want while fixing the $H(a, c)$ value.
By maliciously changing b, one may be able to verify invalid proofs.
The methodology of attacks differ from protocols.

The fix for these is simple - just keep good track of the transcripts.

```
// PIs have to be part of the transcript
for pi in self.cs.to_dense_public_inputs().iter() {
    transcript.append_scalar(b"pi", pi);
}

// We fill with zeros up to the domain size, in order to match the
// length of the vector used by the verifier in his side in the
// implementation
for _ in 0..(domain.size() - self.cs.to_dense_public_inputs().len()) {
    transcript.append_scalar(b"pi", &BlsScalar::from(0u64));
}
```

"Zero" and other Elliptic Curve Attacks

There are various vulnerabilities that revolve around the elliptic curve.

There are attacks revolving around "Zero" - Nguyen Thoi Minh Quan is **the** expert on this subject, finding several bugs in this fashion. Basically, by utilizing zero and the point of infinity, the Aztec PLONK verifier accepts any proof. The value zero (or one, for that matter) and the point of infinity are very special ones, so a good consideration of them is a must.

"Zero" and other Elliptic Curve Attacks

There are more classical attacks as well -

- Invalid Curve Attack - where a point (x, y) is not tested to be on the actual elliptic curve. Testing subgroup membership is also important.
- Small Cofactor Attack - where the cofactor is utilized for an attack

In any case, a deep understanding on the cryptographic backend (proving scheme, various underlying components) is needed to audit it the prover.

Other Miscellaneous Attacks

- Blinding: To provide ZK, blinding techniques are used. If these are missing, then the privacy guarantees may be in question.
- Randomness: An unbiased, unpredictable source of randomness is always important in cryptography. Don't use Mersenne Twister.
- Trusted Setup Leak: Toxic waste could be leaked during the setup.
- Theoretical Vulnerabilities: For example, ZK-friendly hash functions could have some undesirable cryptographic attack vectors.

Table of Contents

Toolings & Techniques

- Manual Audit: self-explanatory
- Formal Verification: Computer-Assisted Proofs can be implemented with languages like Coq to show the constraints are sufficient.
- Fuzzing: Various fuzzing methods to test whether the constraints are enough. For example, a sound execution trace could be mutated to see if the verifier correctly disallows incorrect traces.

Toolings & Techniques - Determinism Check

`https://0xparc.org/blog/ecne` (Franklyn Wang)

The goal here is mostly weak/witness verification.

- **Weak Verification** wants to show that given the input variables, the output variables are uniquely determined from the constraints.
- **Witness Verification** wants to show that all the witness variables in all constraints are uniquely determined as well.

Toolings & Techniques - Determinism Check

The rough idea is to keep a state for each variable - whether they are unique or they have a lower/upper bound. Then, a graph traversal like algorithm is applied, repeating the following over and over.

- Take a constraint from the queue, and apply "rules" to make progress.
- If progress is made on a variable, push to queue all relevant equations.

The goal is to finish with all variables uniquely determined.

Toolings & Techniques - Determinism Check

The rulebook consists of various relatively trivial statements.

- If cx is known to be unique and constant $c \neq 0$, then x is unique.
- $(x - a) \cdot (x - b) = 0$ implies $x \in \{a, b\}$.
- $ax + b = 0$ with $a \neq 0$ uniquely determines x
- Facts on Binary Representation
- If $x = y$, then properties of y is copied to x and vice versa.
- $Ax = b$ with invertible, constant matrix A leads to unique x.
- We can add "trusted functions" to help verification.

Toolings & Techniques - Static Analysis

There are various static analysis tools. One of them is circomspect from Trail of Bits. The current iteration of circomspect can search for

- unconstrained signals
- unused variables or signals
- shadowing variable declarations
- unconstrained divisions
- missing bit length check in LessThan
- and even more!

Notes on Recent zkSync "Vulnerability"

It is important to note that zkEVM and EVM that we know may have different specs. There are tradeoffs for achieving zkEVM scaling - and not understanding these changes may lead to disaster for the developers working on those platforms. Recall the zkEVM "types".

In zkSync, a project raised 921 ETH via a token sale, but as the ETH transfers were done via transfer(), it got stuck. Later, the issue was resolved - some gas metering changes were needed.

Resources

- https://github.com/0xPARC/zk-bug-tracker
- https://www.aumasson.jp/data/talks/zksec_zk7.pdf
- ZKDocs from Trail of Bits
- This presentation, hopefully!
- There are some audit reports out now - those are nice.