A Survey of ZKP Security (2024)

rkm0959

Security Researcher, KALOS

February 5th

Introduction

- Security Researcher at KALOS (2022-)
- Audited Scroll zkEVM, Axiom (with Zellic/KALOS)
- Audited Succinct Labs's prover curta
- Paradigm CTF 2023 2nd Place with KALOS++
- Top CTF placements with Super Guesser
- Math Olympiad / Competitive Programming Background

Outline

The target audience for this talk is auditors with

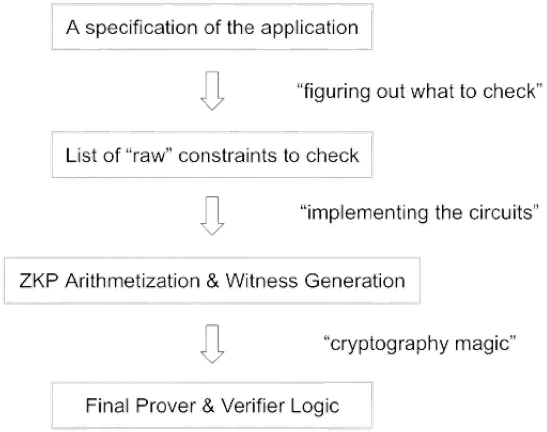
- strong security background
- high-level understanding of ZKP
- good understanding of ZKP related programming
- wants to expand knowledge into ZKP security

Outline

The goal for this talk is to discuss

- various attack vectors in ZKP Security, with examples
- discuss the fundamental reason of why such vectors exist
- discuss prerequisites to start ZKP Security
- especially, a key question here is “how much math do we actually need”

ZKP Development


A specification of the application
"figuring out what to check"
List of "raw" constraints to check
"implementing the circuits"
ZKP Arithmetization & Witness Generation
"cryptography magic"
Final Prover & Verifier Logic

Attack 1: Mistakes at Specification

You really need to know what you need to check inside the circuit.

This is more about understanding your application. You need to know the EVM to audit zkEVM. You need to know EdDSA to audit ZKP of EdDSA.

You need to know recursive SNARKs properly in order to audit it.

Attack 1: Mistakes at Specification

Also important: what's verified in-circuit and what's verified outside.

Integrating different codebases into a single project is a delicate process.

Constraints into Circuit-Friendly Constraints

The biggest part: the main differences are

- all witnesses live in a finite field $\mathbb{F}_p$
- limited set of constraints and tricks, some of them cryptographic
- the witnesses are written in a “fixed rectangular board” (PLONKish)

Welcome to Finite Fields

All witnesses are in $\mathbb{F}_p$, with p prime. These lead to

- overflows and underflows
- $p \neq 2^{256}$ leads to double spend ideas

Attack 2: Overflows and Underflows

In finite field $\mathbb{F}_p$, $p = 0$. All equalities are over modulo p.

Therefore, there's a discrepancy between

- equality over $\mathbb{Z}$ (usually what we desire)
- equality over $\mathbb{F}_p$ (what we can test)

Attack 2: Overflows and Underflows

Example: to test $a \pmod{q} = r$, have witness b and check

- $a = bq + r$ computed in $\mathbb{F}_p$
- $0 \leq r < q$ where r, q are considered in $\mathbb{Z}$
- $0 \leq a, bq + r < p$ considered in $\mathbb{Z}$

Attack 3: Into the L1: Double Spending

In the circuit, x and $x + p$ are the same. In the L1 (out-of-circuit) they are not. Using this trick on nullifiers leads to possible double spending. This is due to both x and $x + p$ being in the usual uint256 range.

Plenty of examples in the zk-bug-tracker repository.

Limited Set of Constraints and Tricks

We have a very limited, but improving set of constraints

- addition and multiplication over $\mathbb{F}_p$
- lookup arguments, possibly dynamic and vector-valued
- randomness sampling via Fiat-Shamir
- offline memory checking algorithms

Limited Set of Constraints and Tricks

These lead to

- assigned but not constrained vulnerabilities
- additional missing constraints
- incorrect handling of assumptions specifications
- incorrect usage of cryptographic tricks

Attack 4: Assigned, not Constrained

The limited nature of constraints lead us to assign variables via out-of-circuit computation and then constrain them afterwards.

Consider 64-bit bitwise decomposition

$$a = \sum_{i=0}^{63} b_i 2^i$$

Attack 4: Assigned, not Constrained

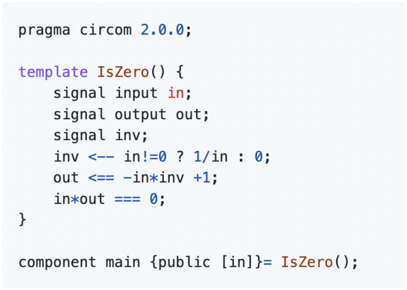
It would be hard to compute b_i via direct constraints. The efficient way is to assign them directly as witnesses and then constrain that

$$a = \sum_{i=0}^{63} b_i 2^i, \quad b_i(b_i - 1) = 0$$

It is crucial that assigning b_i are not enough, as ZKP is about constraints.

Attack 4: Assigned, not Constrained

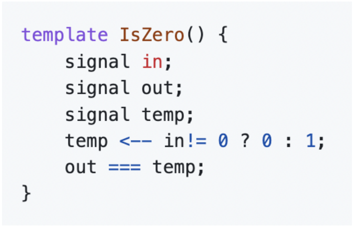
One needs to be really careful about how the DSL handles this.



```
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;
}
```

Attack 4: Assigned, not Constrained

Consider a copy constraint in Halo2.

- Correct: place a copy constraint, connecting two cells
- Incorrect: read the witness value, then assign it to the other

Attack 5: Additional Missing Constraints

Mistakes happen while converting a constraint into ZKP-friendly one.

To constrain that $r = (a \leq b)$ with $0 \leq a, b < 2^n$ known, you need

$$2^n + b - a = 2^n \cdot r + c$$

with $r \in [0, 1]$ and $c \in [0, 2^n)$.

Indeed, this conversion is tricky and opens up many vectors for mistakes.

Attack 6: Assumptions Handling

Recall that the circuit above has $0 \leq a, b < 2^n$ as an assumption.
Without it, bad cases like $a = p - 1, b = 0, r = 1$.

Incorrect handling of these assumptions is critical.

Attack 6: Assumptions Handling

Q: Why can't we just check all assumptions everywhere?

- some constraints are very expensive (range check)
- performance optimization is important

Documentations are very important to prevent this!

Attack 7: Incorrect Cryptographic Tricks

Each of the cryptographic tricks within the arithmetization

- lookup arguments
- fiat-shamir tricks
- offline memory checking

have their own rules that need to be checked.

Attack 7-1: Lookup Arguments

Dynamic Lookups: In this case, one needs to make sure that the entire lookup column is constrained appropriately. Unconstrained rows bad.

Vector Lookups: It's very important to combine the vectors via Fiat-Shamir randomness, not some fixed number.

Attack 7-2: Fiat-Shamir Tricks

In Halo2, a "Phase" exists to handle multiple rounds and their relevant randomness. Even if explicit phases are not used, implicit ones are possible.

A good intuitive understanding of Fiat-Shamir is important.

Attack 7-2: Fiat-Shamir Tricks

Consider a classic RLC application - to prove c is the concat of a and b, where a, b, c are strings of fixed length n_a, n_b, n_c, one checks that

$$\mathrm{RLC}(a) \cdot \gamma^{n_b} + \mathrm{RLC}(b) = \mathrm{RLC}(c)$$

where we define RLC as, with randomness γ,

$$\mathrm{RLC}(a_0, \cdots, a_{n-1}) = \sum_{i=0}^{n-1} a_i \cdot \gamma^{n-1-i}$$

Attack 7-2: Fiat-Shamir Tricks

How does one sample γ? The important part is that γ should be a hash with a preimage that includes a, b, c somehow, to prevent attacks.

How to attack weak Fiat-Shamir? If $\gamma = H(a, b)$, then one fixes a, b, γ and works around incorrect values of c that satisfy the constraint.

Intuition on Fiat-Shamir and (maybe) Schwartz-Zippel recommended.

If you need a PoC with higher severity, knowledge of LLL recommended.

Attack 7-3: Offline Memory Checking

The “memory-in-the-head” technique and its variants allow the tuple

$$(\text{name}, \text{index}, \text{time}, \text{value}, \text{multiplicity})$$

to be added and read - and shows all added values are read at some point.

Attack 7-3: Offline Memory Checking

In a single array, consider that we do, in order,

- add $(1, t_1, v_1, 1)$, read $(1, t_2, v_2, 1)$
- add $(1, t_3, v_3, 1)$, read $(1, t_4, v_4, 1)$

Intuition is $(t_1, v_1) = (t_2, v_2)$ and $(t_3, v_3) = (t_4, v_4)$.

In reality, the constraints we placed is

$$\{(t_1, v_1), (t_3, v_3)\} = \{(t_2, v_2), (t_4, v_4)\}$$

There's not really a single fixed spec for this technique, so depends.

The fixed rectangular board

In PLONKish, the witnesses are organized on a fixed rectangular board.

This leads to

- incorrect handling of variable length objects
- incorrect handling of boundary / transition constraints

Attack 8: Variable Length Objects

One direct consequence of the “fixed rectangular board” is that you can’t deal with variable length arrays at all. The only way is to have a fixed length array of “MAX LENGTH”, and assign a variable for its length.

Therefore, via prefix/suffix zeros, one could have

$$\mathrm{RLC}(a) = \mathrm{RLC}(b), \quad a \neq b$$

Attack 9: Boundary/Transition Constraints

With the rectangular structure, a common technique is to have

- A boundary constraint, conditioning a given row
- A transition constraint, conditioning two consecutive rows

For each region, be careful about the first/last row!

Attack 9: Boundary/Transition Constraints

Selector	Value	Accumulator
1	5	5
1	8	58
0	7	587

$$sel \cdot (10 \cdot accu + val' - accu') = 0$$

Attack 10: Cryptographic Backend + Misc.

More difficult cryptography vulnerabilities are possible -

- Frozen Heart (prover-side weak Fiat-Shamir)
- Weak Randomness (weak RNG being used)
- Weak Blinding (losing zero-knowledge)

There's also some weird ones like compiler optimizations, where unused public inputs are optimized out, leading to malleability.

The prerequisites for ZKP Security

Q: How much cryptography/math is exposed to the surface?

- The prerequisites will be “enough math to understand the above”

Let’s take a look at some examples.

The prerequisites for ZKP Security

Case 1: I have to audit the entire prover and verifier logic

- You need to know the entirety - down to the paper level
- Understand the protocol's soundness and zero-knowledge-ness proof

The prerequisites for ZKP Security

Case 2: I have to audit the usual Halo2 circuit

- You need to know the relevant math for the application specs
- If there are chips for mathematical operations (comparing, range checking, bitwise decomposition, etc) - need to be able to prove soundness and completeness for the ZKP circuit implementation
- Strong intuitive understanding of Fiat-Shamir
- Strong understanding of the underlying DSL/library (here, Halo2)

The prerequisites for ZKP Security

Case 3: It's Halo2, but all mathy things are already audited

- You need to know the relevant math for the application specs
- Strong understanding of the underlying DSL/library (here Halo2)

The prerequisites for ZKP Security

Case 4: It's in a DSL/zkVM that does the ZKP under the hood

- You need to know the relevant math for the application specs
- Strong understanding of the underlying DSL/library

The math prerequisites for Smart Contract Security

I'm actually auditing a smart contract

- You need to know the relevant math for the application specs
- Strong understanding of the underlying DSL/library

I think I see a pattern - do you?

Potential Endgame

If a strong DSL/library/VM for ZKP circuit writing comes with

- **all common mathematical functions**
- **all cryptographic tricks under the hood**

then ZKP security math (for circuits at least) will be much easier.

In that era, compiler related stuff could be more important, and more fundamental security skills and adversarial mindset will be more important.