An Overview of ZKP Security

rkm0959

Head of Security @ KALOS

June 29th

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*.

PLONK: A Quick Explainer

PLONK allows three types of constraints over $\mathbb{F}_p$.

Polynomial Constraints

$$q_a \cdot a + q_b \cdot b + q_m \cdot a \cdot b + q_c = 0$$

Equality Constraints

$$[a_1, \cdots, a_n, b_1, \cdots, b_n, c_1 \cdots, c_n] = \sigma([a_1, \cdots, a_n, b_1, \cdots, b_n, c_1 \cdots, c_n])$$

Lookup Constraints

$$a_i \in T, \quad [a_i, b_i, c_i] \in T$$

PLONK: An Example

Selector1	Selector2	val1	val2	val3
1	1	3	5	8
1	1	5	8	13
1	0	8	13	21
1	0	2	6	8

$$sel_1 \cdot (val_1 + val_2 - val_3) = 0$$
$$sel_2 \cdot (val'_1 - val_2) = 0$$
$$sel_2 \cdot (val'_2 - val_3) = 0$$
$$val_2[1] - val_3[3] = 0$$

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 the former.

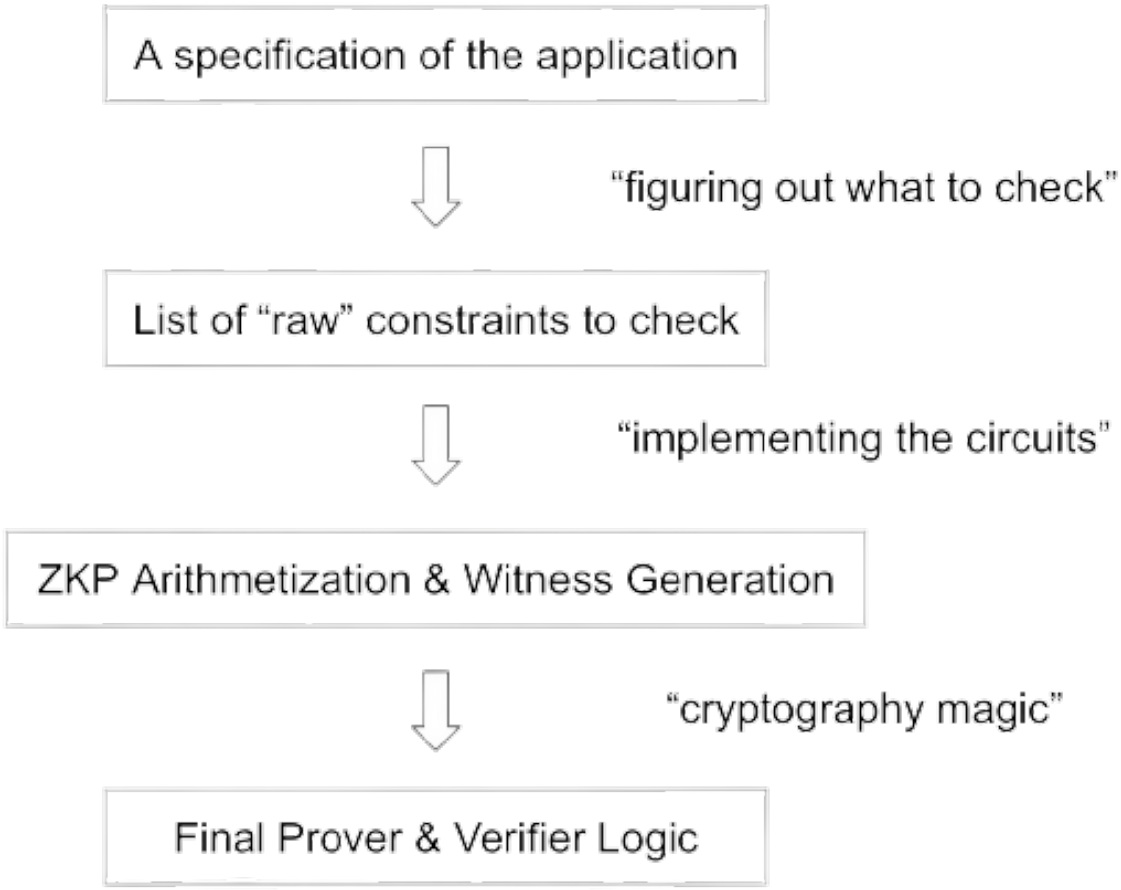
Quick Vulnerability Example

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

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

Can you find the missing piece?

ZKP Development



Step 1: What to Check?

	Matches Specification	Doesn't Match Specification
Passes the Checks	OK	Underconstrained
Fails the Checks	Overconstrained	OK

Step 1: What to Check?

"Overconstraining" is not exactly intuitive - but it's still important.

	Matches Specification	Doesn't Match Specification
Passes the Checks	OK	Underconstrained
Fails the Checks	**Overconstrained**	OK

Step 1: What to Check?

"Underconstraining" can also be caused by simply forgetting a check.

	Matches Specification	**Doesn't Match Specification**
Passes the Checks	OK	**Underconstrained**
Fails the Checks	Overconstrained	OK

Conclusion: Security fundamentals are still very relevant.

Step 2: Circuit Implementation

Clearly, the main dish of the ZKP security.

Common Vulnerabilities

1. Under-constrained Circuits
2. Nondeterministic Circuits
3. Arithmetic Over/Under Flows
4. Mismatching Bit Lengths
5. Unused Public Inputs Optimized Out
6. Frozen Heart: Forging of Zero Knowledge Proofs
7. Trusted Setup Leak
8. Assigned but not Constrained

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.

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: 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 $n = 3, k = 2^{255}, r = 0, a = 2^{255}$.

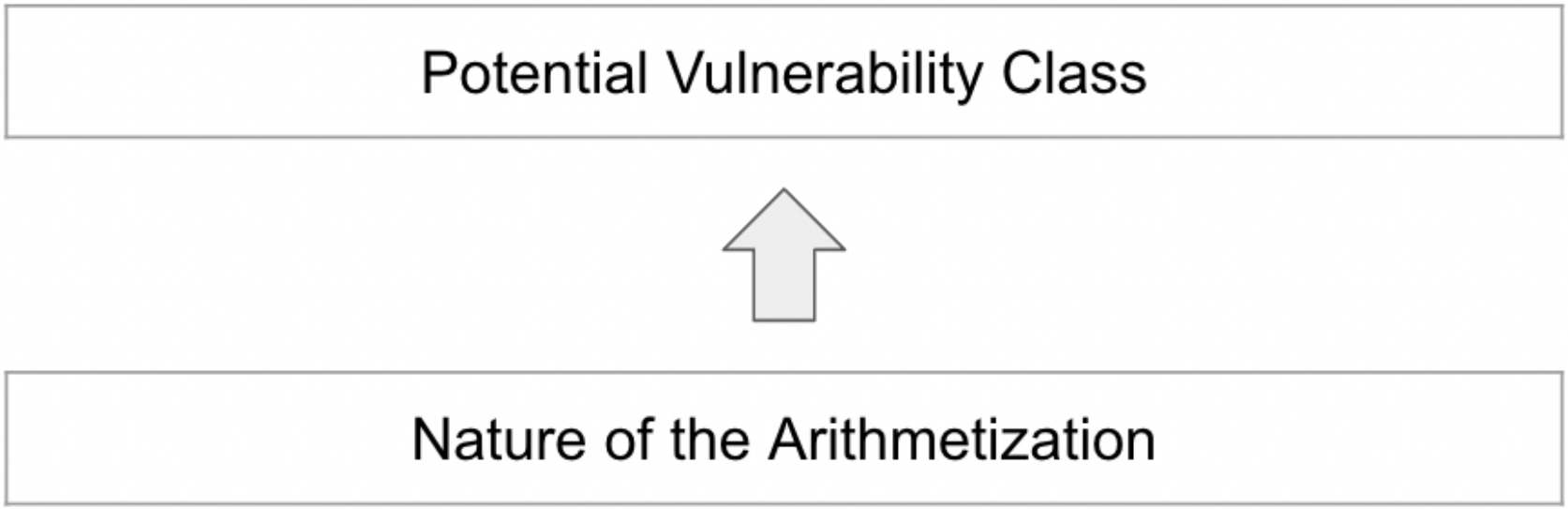
```
// Constrain k * n + r no overflow
cb.add_constraint("overflow == 0 for k * n + r", mul_add_words.overflow());
```

The Nature of the Arithmetization

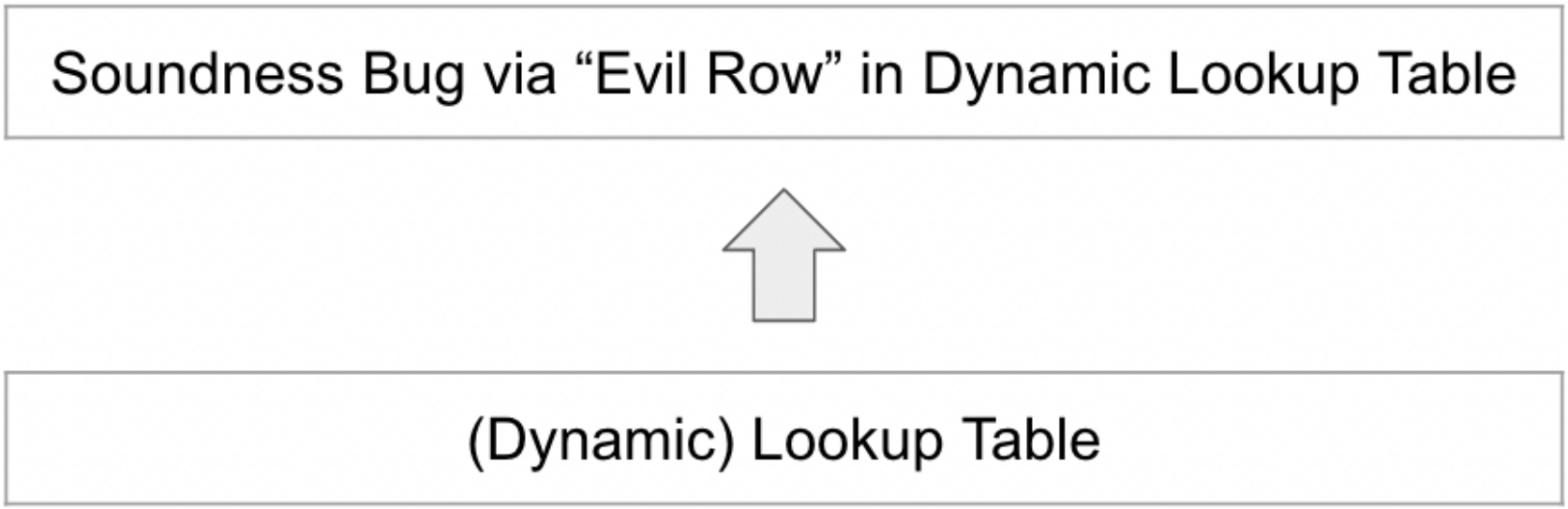
Overflows, Underflows, Range Checks, Bit Length Issues

Arithmetization over Fp, 254 bit prime p (BN254 PLONK)

The Nature of the Arithmetization



The Nature of the Arithmetization: Lookups



The Nature of the Arithmetization: Lookups

soundness problem of advice lookup #866

Open **lispc** opened this issue on Oct 31, 2022 · 10 comments

lispc commented on Oct 31, 2022

Advice Column for Lookups
Constrained
Constrained
Not Constrained

The Nature of the Arithmetization: Selectors

There are various ways where selectors could lead to vulnerabilities.

- Incorrect selector enabling logic
- Misuse of "Selectors" that aren't actually fixed columns

Example: Scroll's Poseidon Circuit

```
let (states_in, states_out) = layouter.assign_region(
    || "hash table",
    |mut region| {
        let offset = self.fill_hash_tbl_custom(&mut region)?;
        self.fill_hash_tbl_body(&mut region, offset)
    },
)?;
```

- 1 or 2 custom rows based on whether "mpt_only" is true
- main hash table body rows

Example: Scroll's Poseidon Circuit

```
config.s_custom.enable(region, 1)?;
if self.mpt_only {
    return Ok(1);
}
```

A part of custom row logic. Incorrectly enables the selector "s_custom" in offset 1 even when "mpt_only" is true, then returns 1 as the offset.

Example: Scroll's Poseidon Circuit

```
fn fill_hash_tbl_body(
    &self,
    region: &mut Region<'_, Fp>,
    begin_offset: usize,
) -> Result<PermutedStatePair<PC::Word>, Error> {
```

The table body is filled in starting with the offset 1.

Example: Scroll's Poseidon Circuit

```
meta.create_gate("custom row", |meta| {
    let s_enable = meta.query_selector(s_custom);

    vec![
        s_enable.clone() * meta.query_advice(hash_inp[0], Rotation::cur()),
        s_enable.clone() * meta.query_advice(hash_inp[1], Rotation::cur()),
        s_enable * meta.query_advice(control, Rotation::cur()),
    ]
});
```

This leads to overconstrain on the hash inputs.

The Nature of the Arithmetization: Selectors

Sometimes we "select" constraints based on expressions/columns that are not actually public or fixed. For example, we could take the **IsZero** expression and use it as a "selector" for a gate.

In some cases, selectors are actually defined as advice columns.

The Nature of the Arithmetization: Selectors

```rust
/// The config for poseidon hash circuit
#[derive(Clone, Debug)]
pub struct SpongeConfig<Fp: FieldExt, PC: Chip<Fp> + Clone + DebugT> {
    permute_config: PC::Config,
    hash_table: [Column<Advice>; 5],
    hash_table_aux: [Column<Advice>; 6],
    control_aux: Column<Advice>,
    s_sponge_continue: Column<Advice>,
    control_step_range: TableColumn,
    s_table: Selector,
    s_custom: Selector,
    /// the configured step in var-len mode, i.e (`input_width * bytes in each field`)
    pub step: usize,
}
```

The Nature of the Arithmetization: Selectors

Sometimes, it doesn't matter if selectors are boolean.

$$q \cdot expr = 0, q \neq 0 \implies expr = 0$$

The Nature of the Arithmetization: Selectors + Lookups

Sometimes, it does matter if selectors are boolean.

$$q \cdot expr \in T, q \neq 0 \implies expr \in T$$

A "Close Call" at zkEVM (Scroll/PSE)

The bytecode circuit in zkEVM checks that $(opcode, pushSize)$ is valid by utilizing a lookup table. For this lookup check, the selector is an AND of

- A fixed column q_{enable} is turned on
- A fixed column q_{last} is turned off
- The "tag" of the row (which is 1 for "Byte" and 0 for "Header")

A "Close Call" at zkEVM (Scroll/PSE)

```
meta.lookup_any(
    "push_data_size_table_lookup(cur.value, cur.push_data_size)",
    |meta| {
        let enable = and::expr(vec![
            meta.query_fixed(q_enable, Rotation::cur()),
            not::expr(meta.query_fixed(q_last, Rotation::cur())),
            is_byte(meta),
        ]);

        let lookup_columns = vec![value, push_data_size];

        let mut constraints = vec![];

        for i in 0..PUSH_TABLE_WIDTH {
            constraints.push((
                enable.clone() * meta.query_advice(lookup_columns[i], Rotation::cur()),
                meta.query_fixed(push_table[i], Rotation::cur()),
            ))
        }
        constraints
    },
);
```

A "Close Call" at zkEVM (Scroll/PSE)

The issue here is that

- The "tag" is actually an advice column.
- The "tag" is never directly constrained to be boolean.

Therefore, with (PUSH5 = 0x64 = 100, 5) inside the table, we can actually try $tag = 5$, $opcode = 20$, and $pushSize = 1$.

Conclusion: with an arithmetization, we can think of how certain properties can lead to vulnerabilities. More bug classes will be fruitful!

Credits

Thanks to

- Kyle Charbonnet for zk-bug-tracker and nice discussions
- 0xPARC and their security group for great information
- Scroll for allowing me to share some findings

Contact me at rkm0959@gmail.com or allen@kalos.xyz!