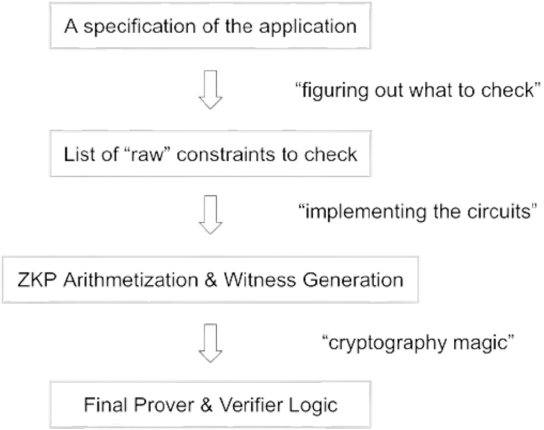
Opinionated Survey of ZKP Security

Gyumin “rkm0959” Roh

Head of Security @ KALOS

June 4th

ZKP Development



Step 0: Specification

$\boxed{\text{A specification of the application}} \longrightarrow$ sanity check

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

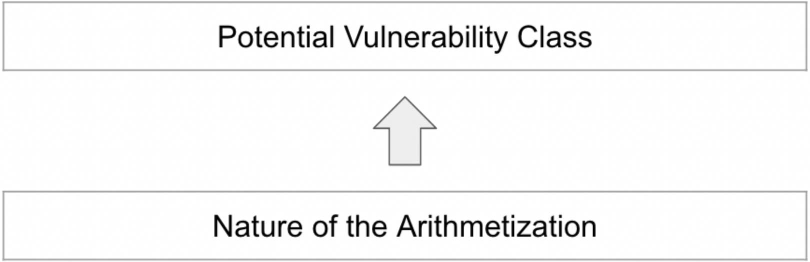
1. Under-constrained Circuits
2. Nondeterministic Circuits
3. Arithmetic Over/Under Flows
4. Mismatching Bit Lengths
5. Unused Public Inputs Optimized Out
6. Frozen Heart: Forging of Zero Knowledge Proofs
7. Trusted Setup Leak
8. Assigned but not Constrained

The Nature of the Arithmetization

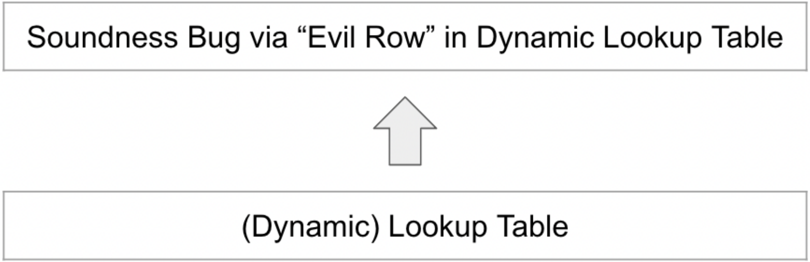
Overflows, Underflows, Range Checks, Bit Length Issues

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.

$$q \cdot expr \in T, q \neq 0 \not\Rightarrow expr \in T$$

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 (multiplication) of

- A fixed column q_{enable} is turned on
- A fixed column q_{last} is turned off
- The "tag" of the row (which is expected to be 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 $pushSize = 1$. This passes the lookup argument!

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

Step 3: The Final Cryptography

Vulnerabilities in the “Final Cryptography”

Implementation Issue in the “Backend”

Developer’s Misuse of the Framework

Trusted Setup Leak, Frozen Heart

Optimized Out Public Inputs

Step 3: The Final Cryptography

Some vulnerabilities can be handled on the backend

```
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]);
}
```

Step 3: The Final Cryptography

...and the same vulnerabilities can be handled on the developer side.

```
// 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;
```

Step 3: The Final Cryptography

ZKP techniques have improved a lot - how about their secure usage?

- How to aggregate SNARKs securely?
- How to apply folding schemes securely?

Conclusion: study on secure usage of new ZKP techniques will be an interesting topic - and the solution may be in the backend/tooling side or the developer side, or maybe even both.

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!