Zero-knowledge proofs security, in practice

JP Aumasson
@veorq

CSO @ taurushq.com

NIZK arguments security, in practice

JP Aumasson
@veorq

CSO @ taurushq.com

This talk

Focus on **zkSNARKs**, a class of NIZK arguments

- *Fully succinct* = O(1) proof size and O(circuit size) verification time

Based on my experience looking for bugs in systems using

- **Groth16**, used in Zcash, Filecoin, and many others
- **Marlin**, a universal zkSNARK, used in Aleo

Most of the content applies to other systems (Plonk, SONIC, etc.) and STARKs

Why study zkSNARKs security?

For blockchain projects: **A major risk**:

- Complexity + Novelty => Non-trivial **bugs**
- A lot **at stake** ($$$ and user data/privacy)

Why study zkSNARKs security?

For blockchain projects: **A major risk**:

- Complexity + Novelty => Non-trivial **bugs**
- A lot **at stake** ($$$ and user data/privacy)

As a cryptographer: **The most interesting** crypto today:

- Solving real-world problems and deployed at scale (good papers + good code!)
- Intricate constructions with non-trivial components
- “Simple but complex" (non-interactive, many moving parts)
- “Multidimensional" way to reason about security

What is zkSNARKs security?

Soundness, often the *highest risk* in practice:

- Invalid proofs should always be rejected
- Forging, altering, replaying valid proofs should be impossible

What is zkSNARKs security?

Soundness, often the *highest risk* in practice:

- Invalid proofs should always be rejected
- Forging, altering, replaying valid proofs should be impossible

Zero-knowledge: Proofs should not leak witness information (private variables)

- In practice succinct proofs of large programs can leak only little data

What is zkSNARKs security?

Soundness, often the *highest risk* in practice:

- Invalid proofs should always be rejected
- Forging, altering, replaying valid proofs should be impossible

Zero-knowledge: Proofs should not leak witness information (private variables)

- In practice succinct proofs of large programs can leak only little data

Completeness, often a DoS/usability risk that may be further exploited:

- Valid proofs should always be accepted
- All programs/circuits supported should be correctly processed

Bug hunting challenges

Practical zkSNARKs are recent, thus auditors often have

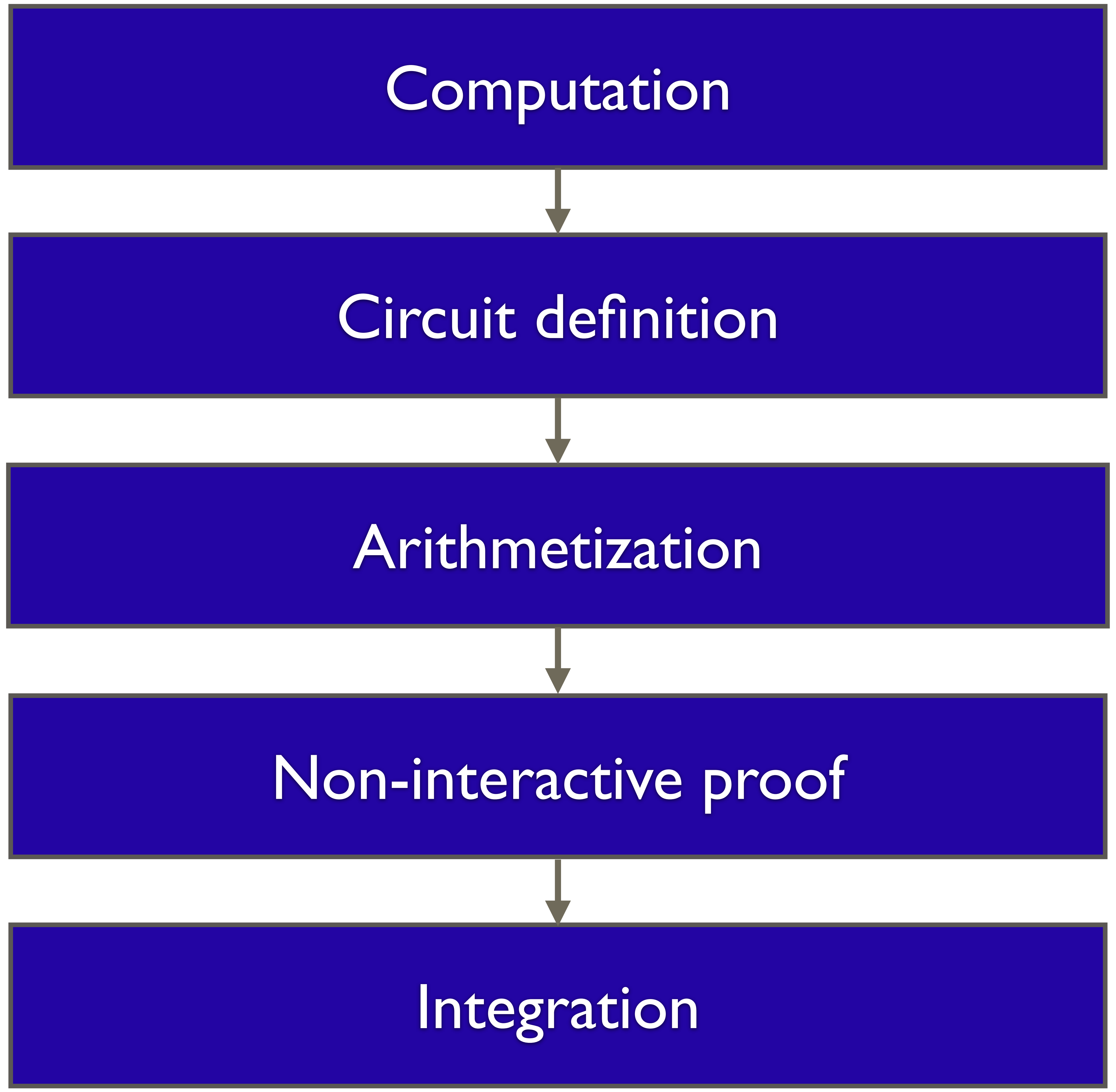
- Limited **experience** auditing ZKPs
- Limited **knowledge** of the theory, of implementations' tricks
- Limited **“checklist"** of bugs and bug classes
- Limited **tooling** and methodology

Most bugs found internally or by teams of similar projects

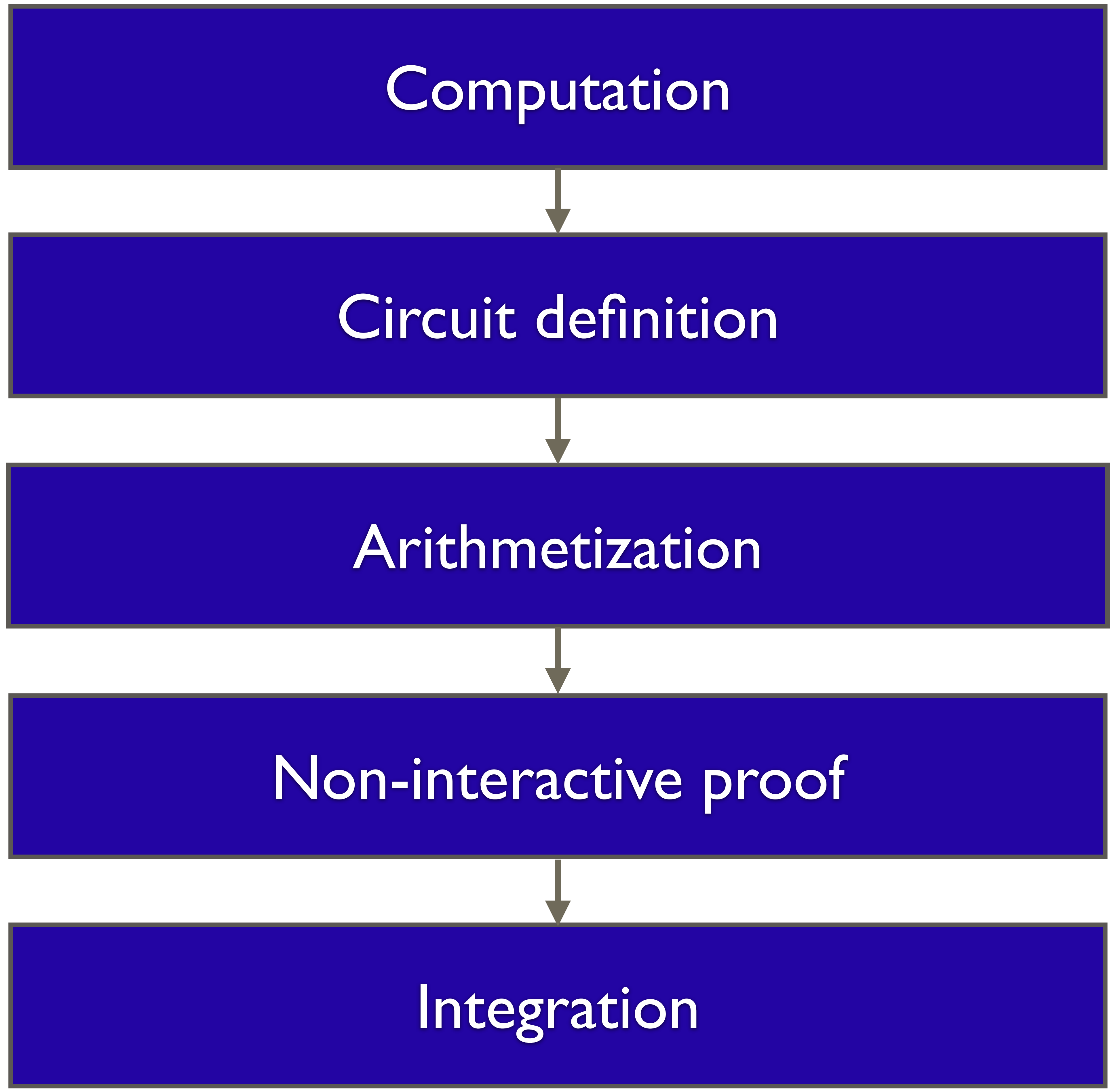
New crypto, new approach

- More **collaboration** with the devs/designers (joint review sessions, Q&As, etc.)
- More **threat analysis**, to understand the application's unique/novel risks
- Practical **experience**: writing PoCs, circuits, proof systems, etc.
- Learn **previous failures**, for example from…
  - Public disclosures and exploits
  - Other audit reports
  - Issue trackers / PRs
  - Community

General workflow, and failure *examples*



General workflow, and failure *examples*



The program's logic is not secure

The circuit is not equivalent to the program

The CS fails to enforce an operation a constraint

Bad choice of internal commitment schemes wrt hiding or binding properties

The application allows replays of previous proofs

How to break zkSNARKs security? (1/2)

Break soundness, for example by exploiting

- Constraint system not effectively enforcing certain constraints
- Insecure generation or protection of proving keys

How to break zkSNARKs security? (1/2)

Break soundness, for example by exploiting

- Constraint system not effectively enforcing certain constraints
- Insecure generation or protection of proving keys

Break zero-knowledge, for example by exploiting

- Private data treated as public variables
- Protocol-level “metadata attacks”

How to break zkSNARKs security? (1/2)

Break soundness, for example by exploiting

- Constraint system not effectively enforcing certain constraints
- Insecure generation or protection of proving keys

Break zero-knowledge, for example by exploiting

- Private data treated as public variables
- Protocol-level “metadata attacks”

Break completeness, for example by exploiting

- Incorrect R1CS synthesis behaviour on edge cases (e.g. wrt number of private vars)
- Gadget composition failure caused by type mismatch between gadget i/o values

How to break zkSNARKs security? (2/2)

Break (off-chain) software, via any bug leading to

- Leakage of data, including via side channels, encodings (“ZK execution”)
- Any form in insecure state (code execution, DoS)

Compromise the supply-chain, via

- Trusted setup's code and execution
- Build and release process integrity
- Software dependencies

How to break zkSNARKs security? (2/2)

Break (off-chain) software, via any bug leading to

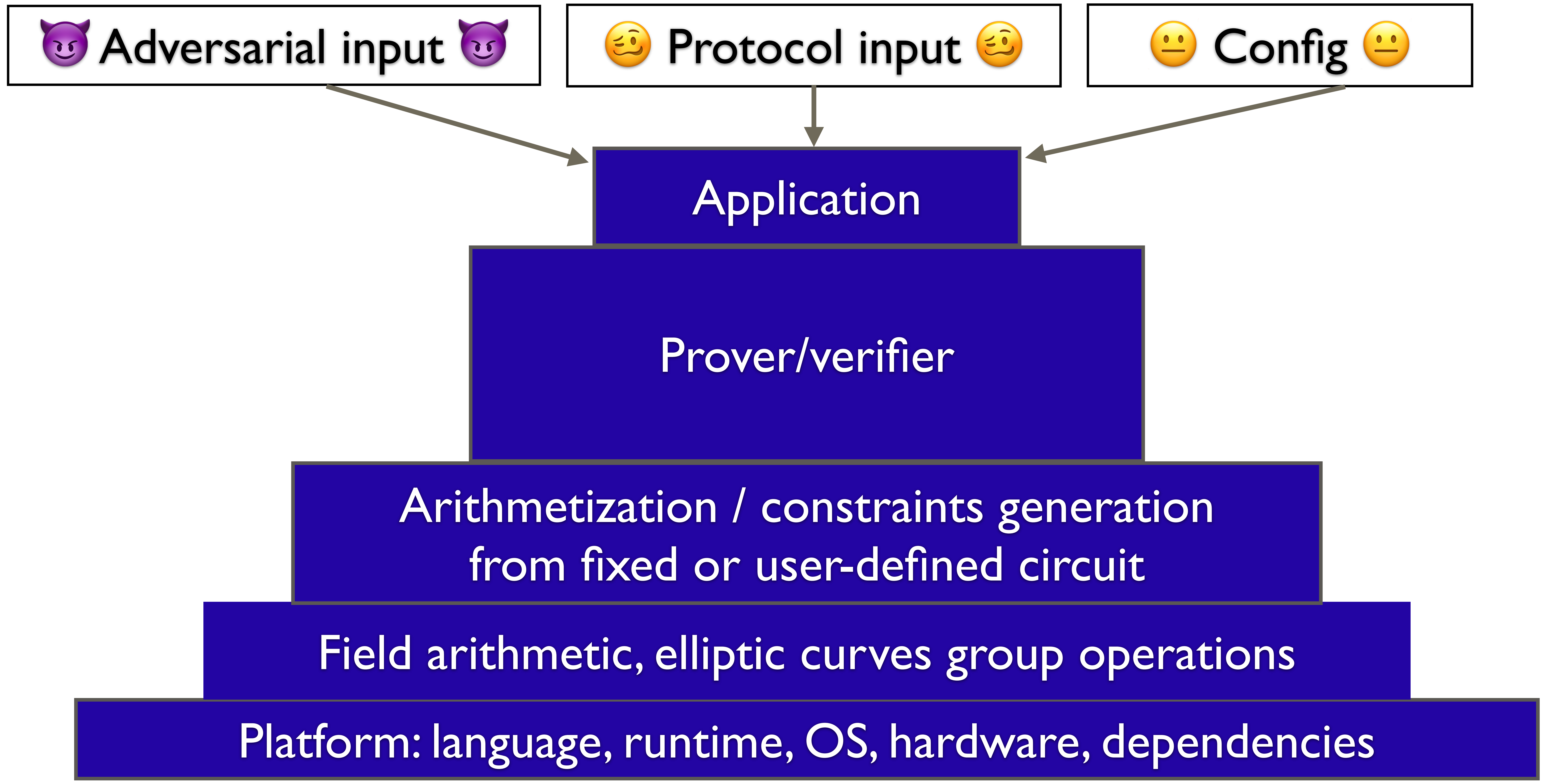
- Leakage of data, including via side channels, encodings (“ZK execution”)
- Any form in insecure state (code execution, DoS)

Compromise the supply-chain, via

- Trusted setup's code and execution
- Build and release process integrity
- Software dependencies
- **Break (on-chain) software** (incl. verifier) via smart contract bugs, logic flaws, etc.

Multiple layers

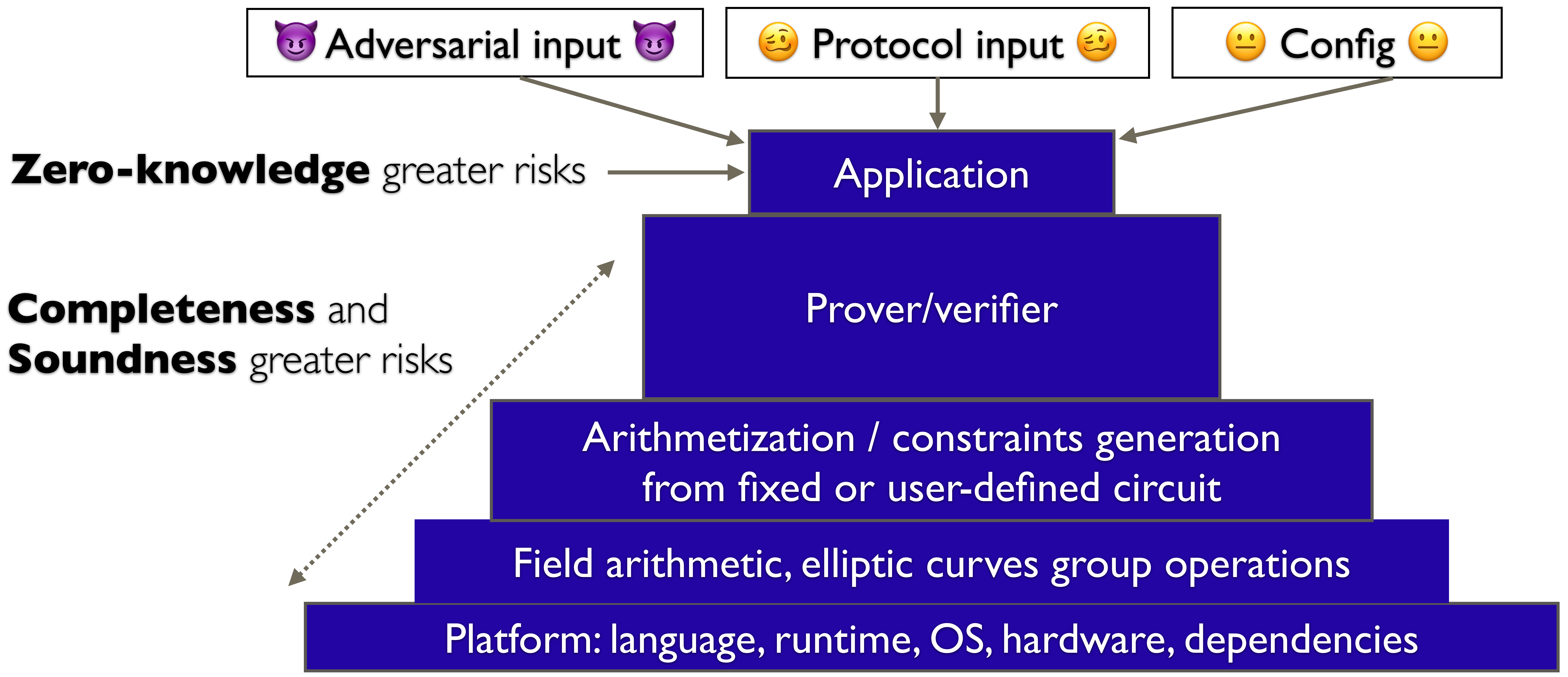
- A failure in a **lower layer** can jeopardise the security of all upper layers



😈 Adversarial input 😈
🥴 Protocol input 🥴
😐 Config 😐
Application
Prover/verifier
Arithmetization / constraints generation from fixed or user-defined circuit
Field arithmetic, elliptic curves group operations
Platform: language, runtime, OS, hardware, dependencies

Multiple layers

- A failure in a **lower layer** can jeopardise the security of all upper layers


😈 Adversarial input 😈
🥴 Protocol input 🥴
😐 Config 😐
Zero-knowledge greater risks
Application
Completeness and **Soundness** greater risks
Prover/verifier
Arithmetization / constraints generation from fixed or user-defined circuit
Field arithmetic, elliptic curves group operations
Platform: language, runtime, OS, hardware, dependencies

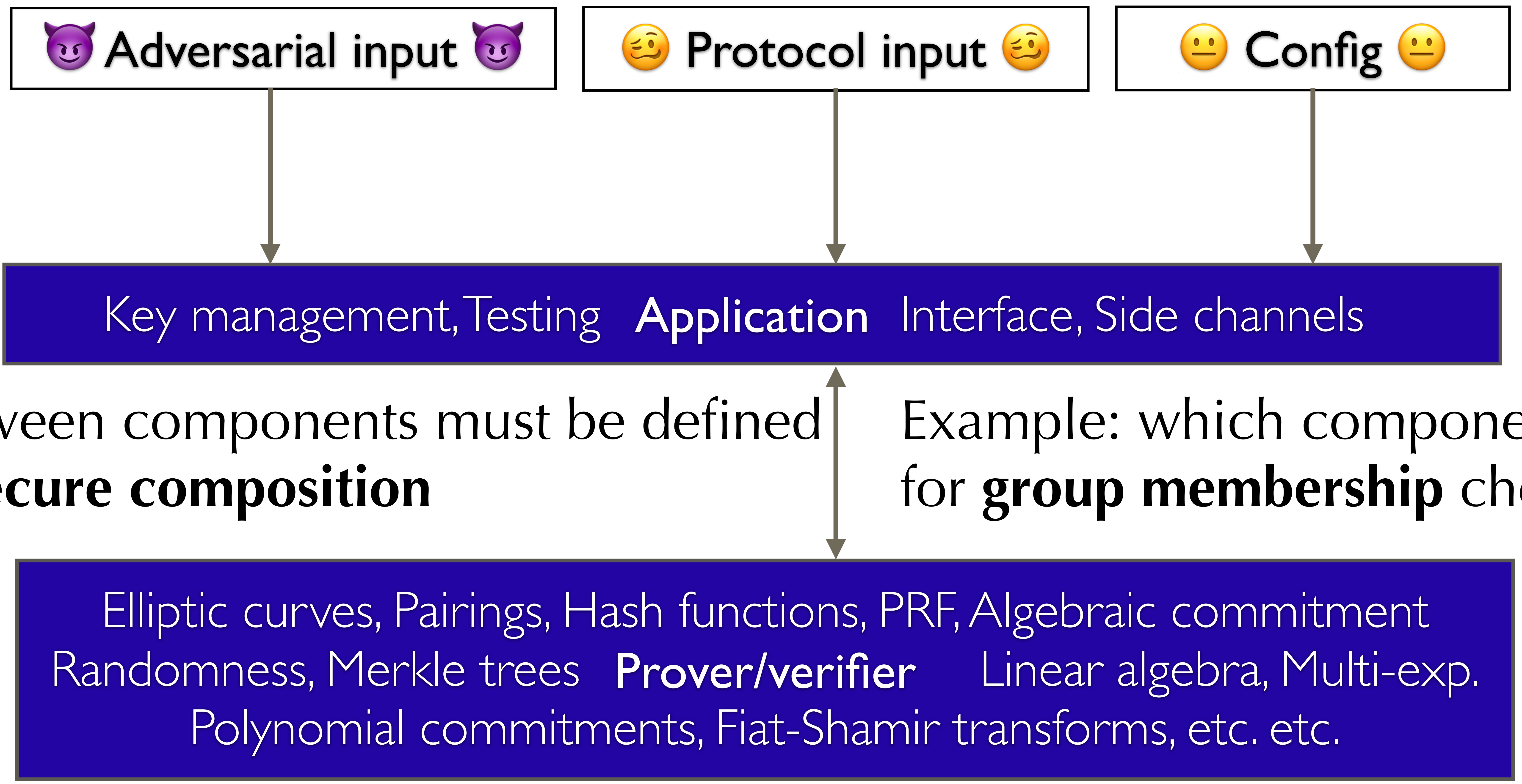
Multiple layers

- A failure in a **subcomponent** can jeopardise the security of all upper layers

😈 Adversarial input 😈
🥴 Protocol input 🥴
😐 Config 😐
Application
Key/nonce management, Testing
Interface, Side channels, Replays
Prover/verifier
Hashing, PRF, Algebraic commitment, Randomness, Merkle trees, …
Fiat-Shamir, Polynomial commitments, Hash-to-curve, linear algebra, …
Arithmetization / constraints generation from fixed or user-defined circuit
R1CS, AIR, polynomials, …
Field arithmetic, elliptic curves group operations
Fast operations, multi-exp, …
Platform: language, runtime, OS, hardware, dependencies
RNG, …

Multiple layers

- Security 101: **Input validation** must be defined, implemented, and tested

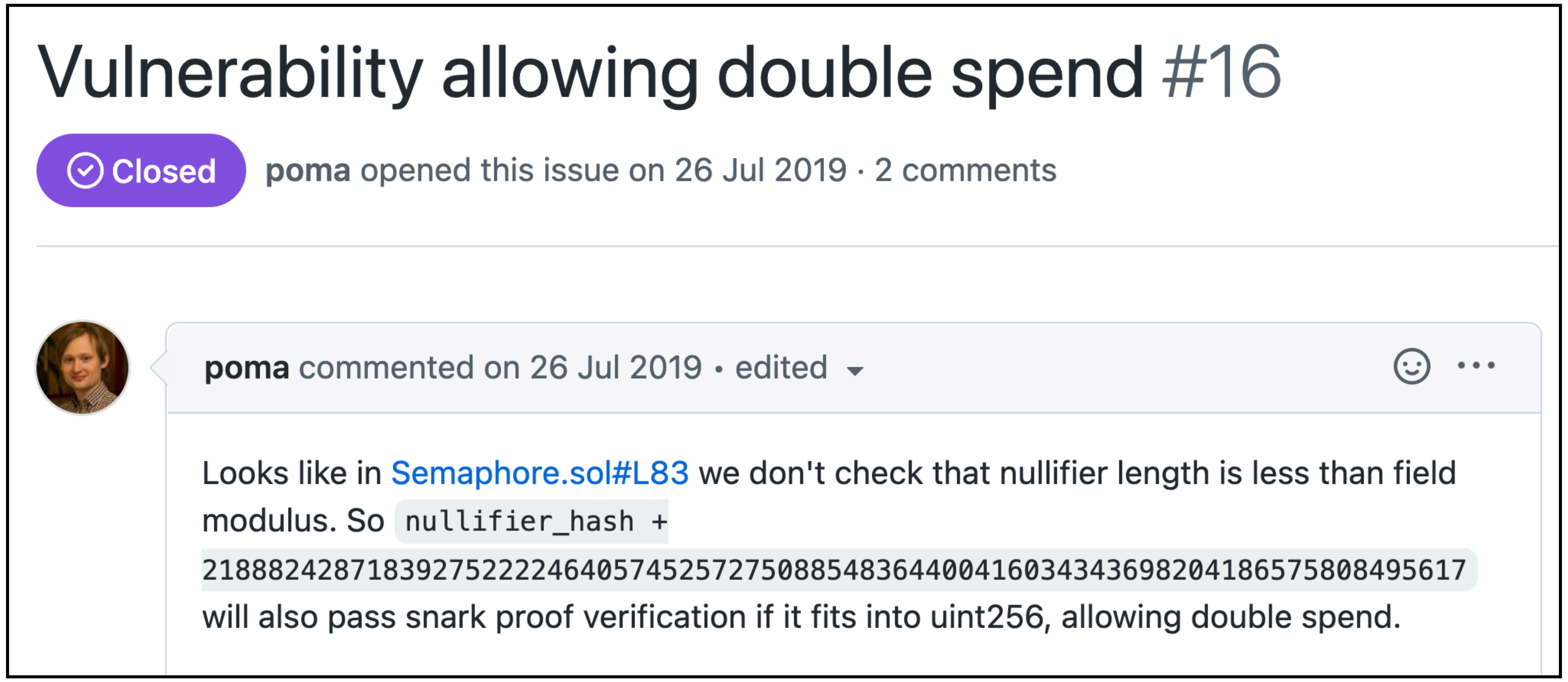

😈 Adversarial input 😈
🥴 Protocol input 🥴
😐 Config 😐
Key management, Testing **Application** Interface, Side channels
Elliptic curves, Pairings, Hash functions, PRF, Algebraic commitment
Randomness, Merkle trees **Prover/verifier** Linear algebra, Multi-exp.
Polynomial commitments, Fiat-Shamir transforms, etc. etc.

Contracts between components must be defined to prevent **insecure composition**

Example: which component is responsible for **group membership** checks?

NATURAL HISTORY MUSEUM
REAL-LIFE BUGS & Insects
NATIONAL GEOGRAPHIC
Collect them if you dare!
NATIONAL GEOGRAPHIC
Published Weekly
ISSUE 29
REAL-LIFE
BUGS
& Insects
DISCOVER THEIR
AMAZING
SECRET LIVES

Soundness – Field arithmetic



Vulnerability allowing double spend #16

Closed poma opened this issue on 26 Jul 2019 · 2 comments

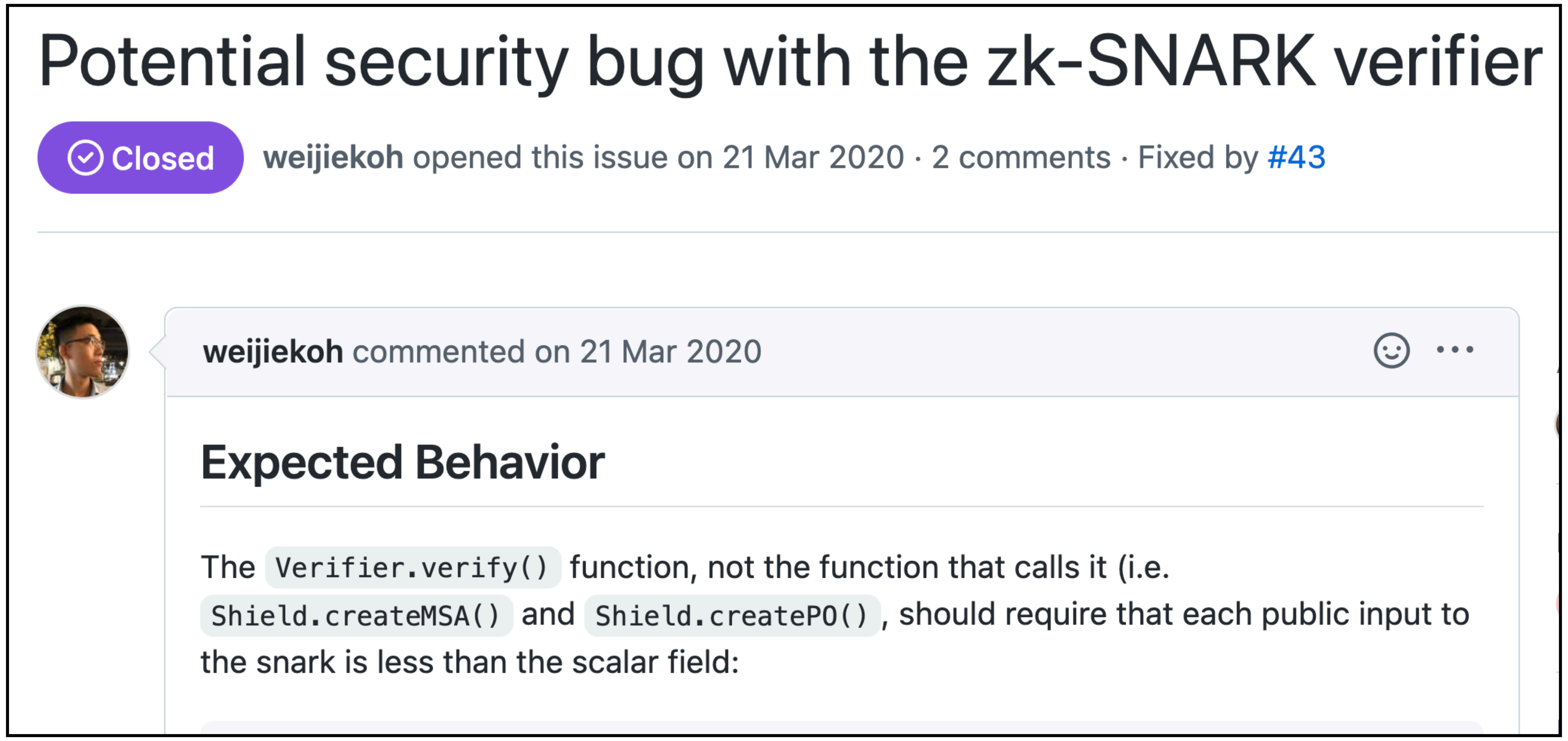
poma commented on 26 Jul 2019 • edited

Looks like in Semaphore.sol#L83 we don't check that nullifier length is less than field modulus. So `nullifier_hash + 21888242871839275222246405745257275088548364400416034343698204186575808495617` will also pass snark proof verification if it fits into uint256, allowing double spend.

Root cause: Missing overflow check (of a nullifier ~ unique ID of a shielded payment)

https://github.com/appliedzkp/semaphore/issues/16

Soundness – Field arithmetic



Potential security bug with the zk-SNARK verifier

Closed weijiekoh opened this issue on 21 Mar 2020 · 2 comments · Fixed by #43

weijiekoh commented on 21 Mar 2020

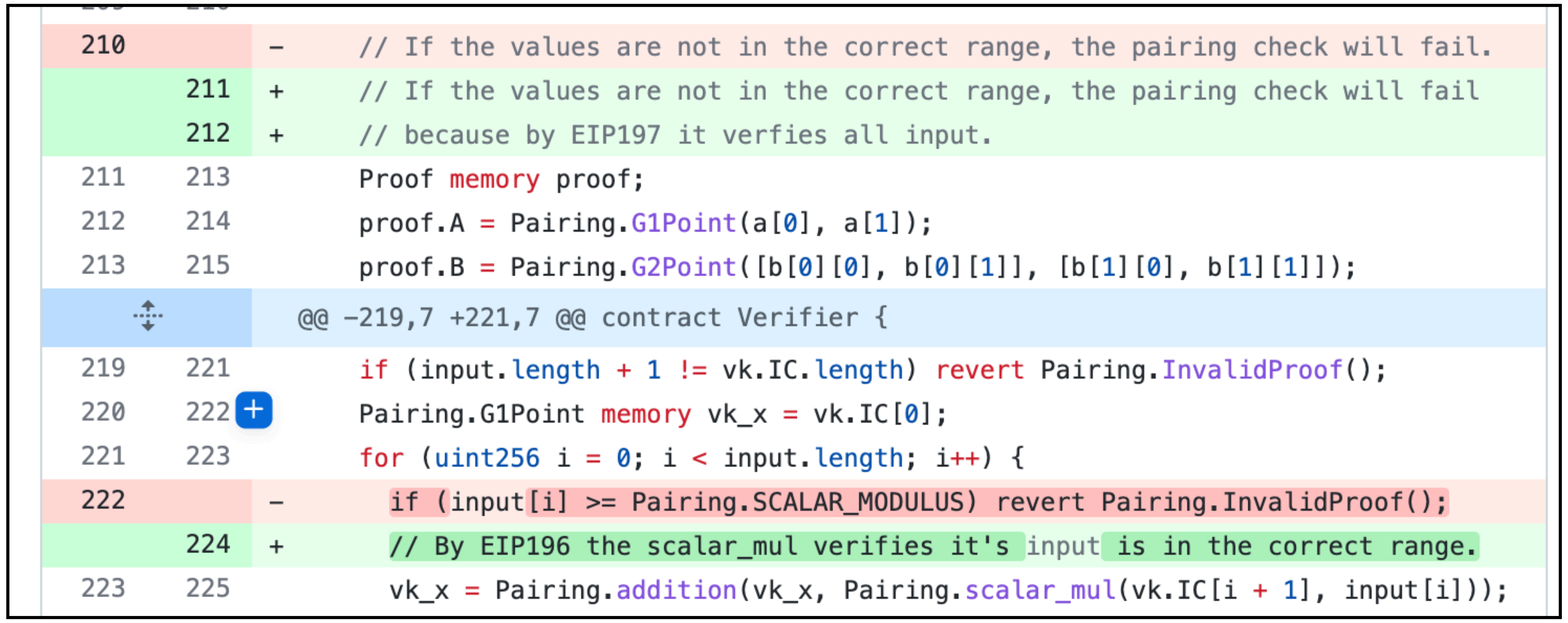
Expected Behavior

The `Verifier.verify()` function, not the function that calls it (i.e. `Shield.createMSA()` and `Shield.createPO()`, should require that each public input to the snark is less than the scalar field:

Missing overflow check (of a public input)

https://github.com/eea-oasis/baseline/issues/34

Soundness – Field arithmetic

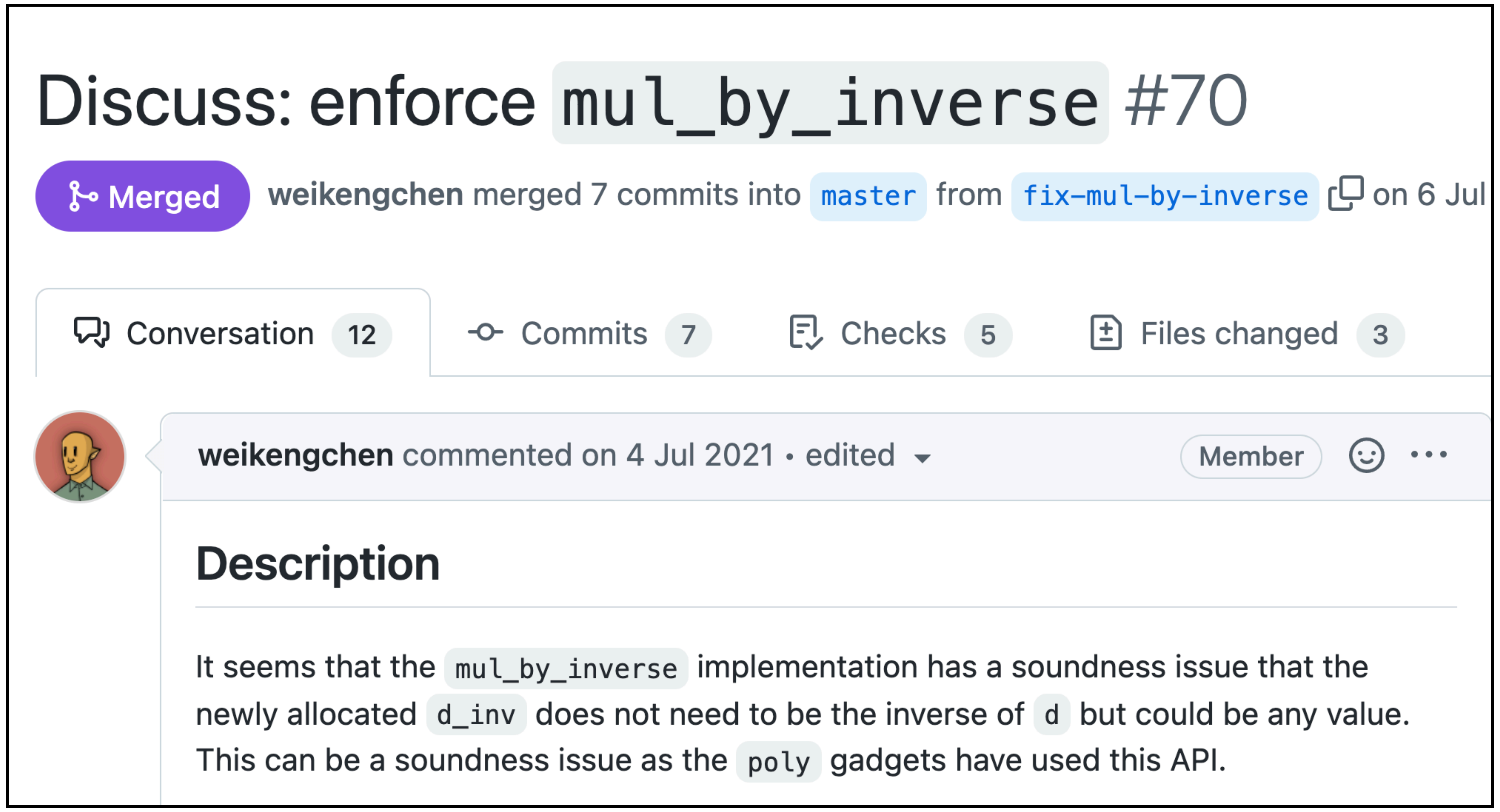


```diff
-    // If the values are not in the correct range, the pairing check will fail.
+    // If the values are not in the correct range, the pairing check will fail
+    // because by EIP197 it verfies all input.
     Proof memory proof;
     proof.A = Pairing.G1Point(a[0], a[1]);
     proof.B = Pairing.G2Point([b[0][0], b[0][1]], [b[1][0], b[1][1]]);
@@ -219,7 +221,7 @@ contract Verifier {
     if (input.length + 1 != vk.IC.length) revert Pairing.InvalidProof();
     Pairing.G1Point memory vk_x = vk.IC[0];
     for (uint256 i = 0; i < input.length; i++) {
-      if (input[i] >= Pairing.SCALAR_MODULUS) revert Pairing.InvalidProof();
+      // By EIP196 the scalar_mul verifies it's input is in the correct range.
       vk_x = Pairing.addition(vk_x, Pairing.scalar_mul(vk.IC[i + 1], input[i]));
```

Missing overflow check (of a public input)

https://github.com/appliedzkp/semaphore/pull/96/

Soundness – R1CS


Discuss: enforce `mul_by_inverse` #70

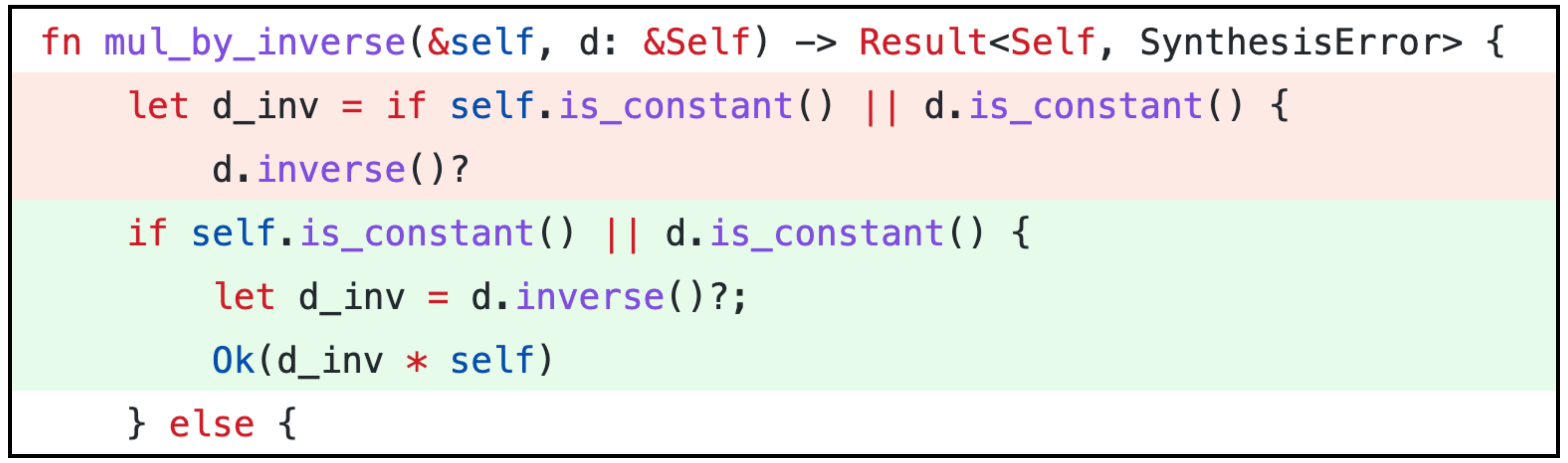
Merged — weikengchen merged 7 commits into `master` from `fix-mul-by-inverse` on 6 Jul

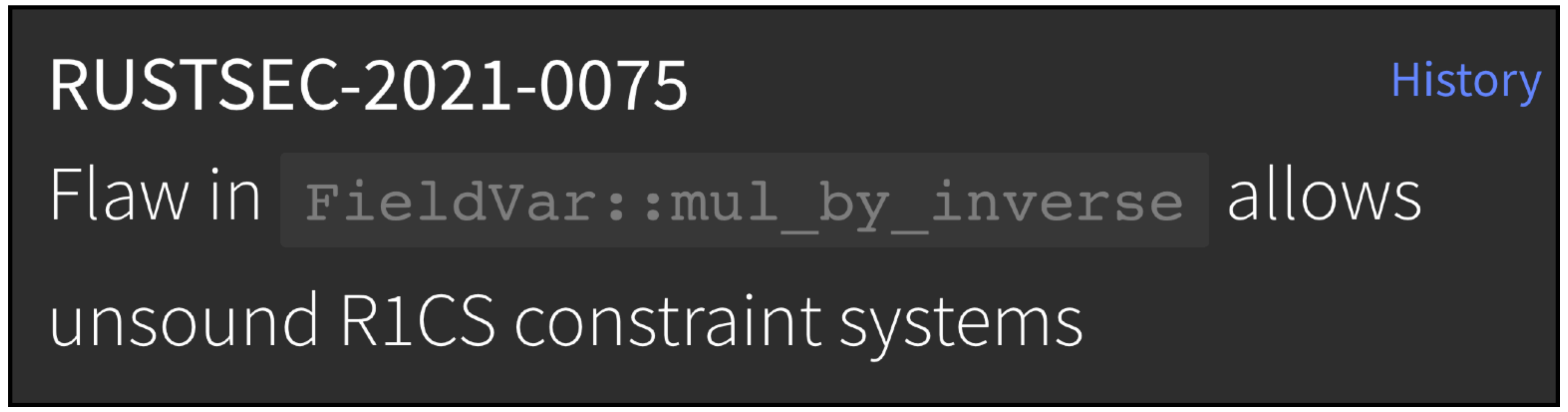
Conversation 12 · Commits 7 · Checks 5 · Files changed 3

weikengchen commented on 4 Jul 2021 • edited — Member

Description

It seems that the `mul_by_inverse` implementation has a soundness issue that the newly allocated `d_inv` does not need to be the inverse of `d` but could be any value. This can be a soundness issue as the `poly` gadgets have used this API.


```
fn mul_by_inverse(&self, d: &Self) -> Result<Self, SynthesisError> {
    let d_inv = if self.is_constant() || d.is_constant() {
        d.inverse()?
    if self.is_constant() || d.is_constant() {
        let d_inv = d.inverse()?;
        Ok(d_inv * self)
    } else {
```

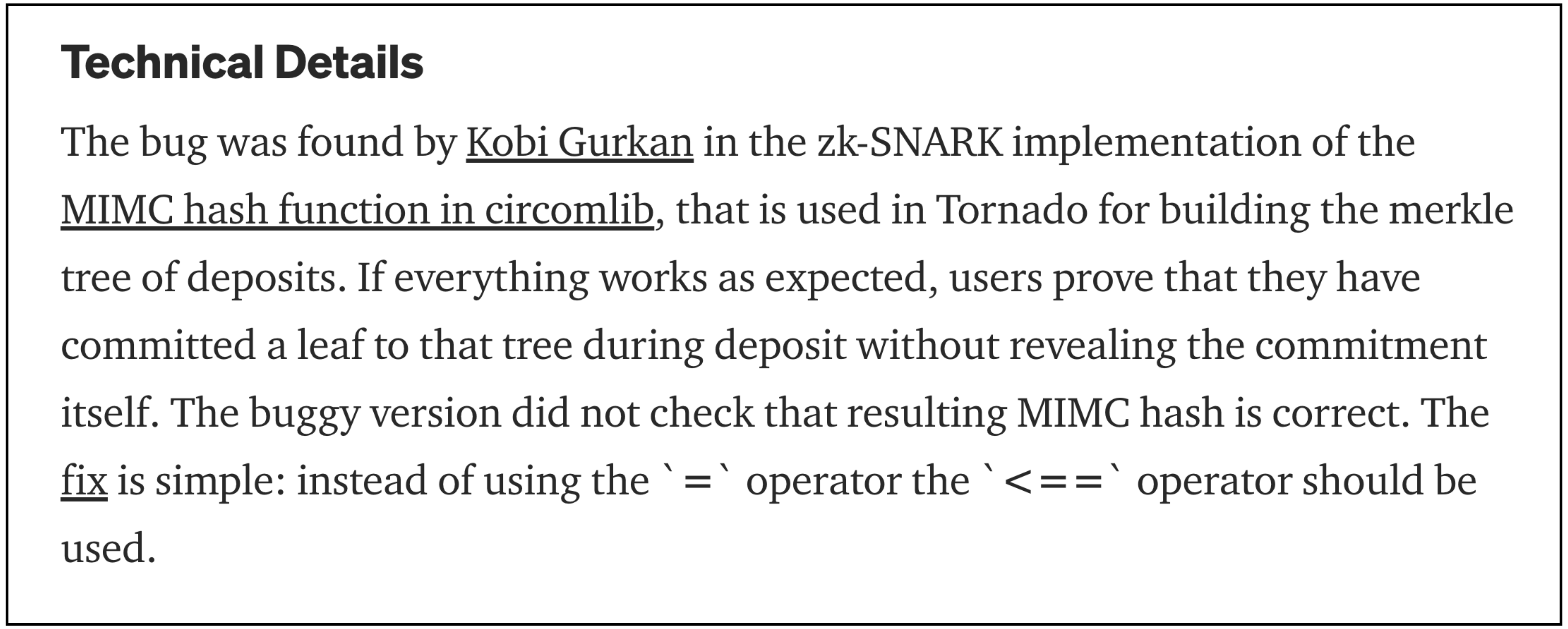

RUSTSEC-2021-0075 — History

Flaw in `FieldVar::mul_by_inverse` allows unsound R1CS constraint systems

Field element inverse property not enforced by the constraint system

https://github.com/arkworks-rs/r1cs-std/pull/70

Soundness – Hash validation



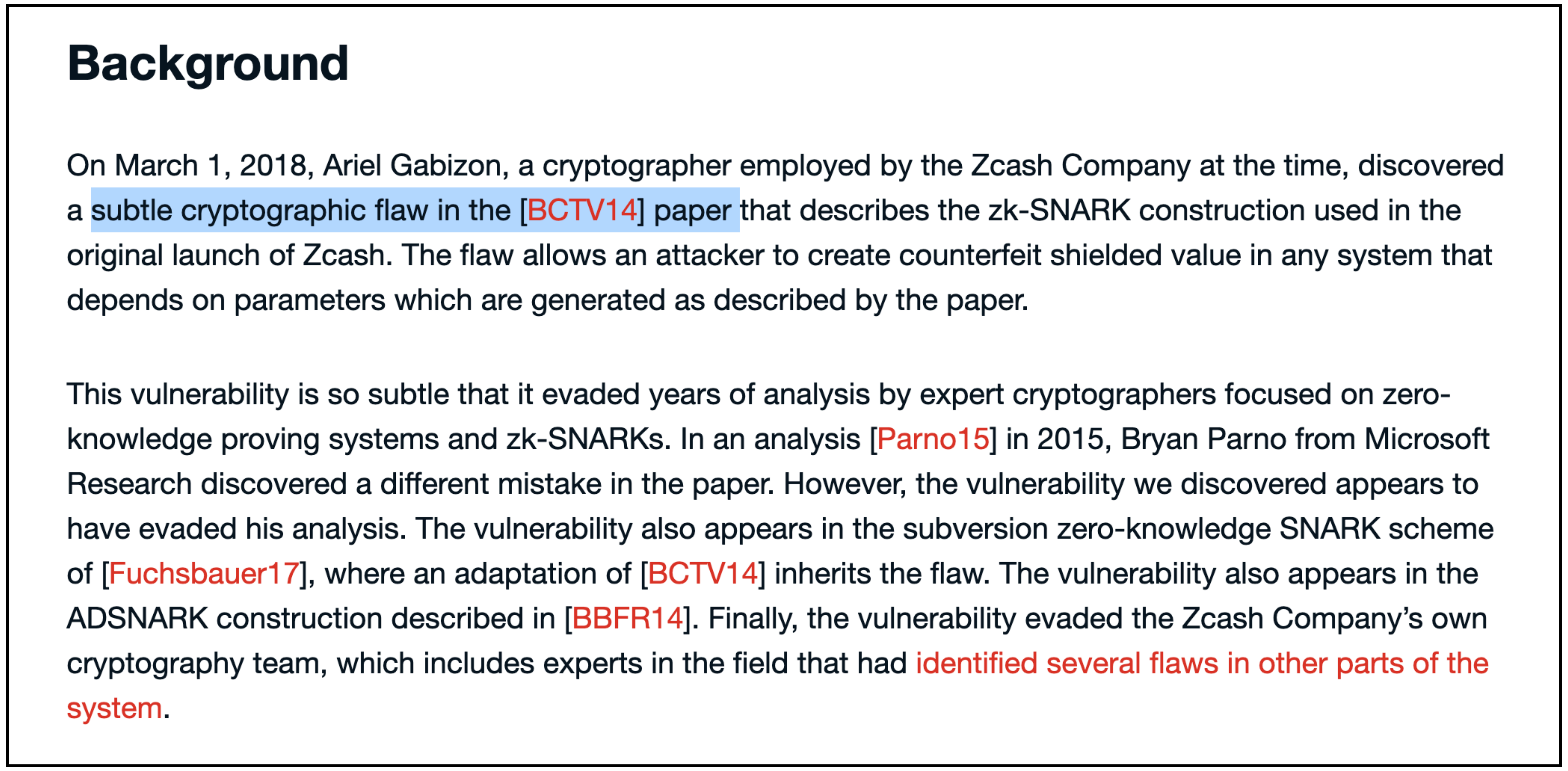
Technical Details

The bug was found by Kobi Gurkan in the zk-SNARK implementation of the MIMC hash function in circomlib, that is used in Tornado for building the merkle tree of deposits. If everything works as expected, users prove that they have committed a leaf to that tree during deposit without revealing the commitment itself. The buggy version did not check that resulting MIMC hash is correct. The fix is simple: instead of using the `=` operator the `<==` operator should be used.

Coding error, allowing to fake the witness' Merkel root and forge proofs

https://tornado-cash.medium.com/tornado-cash-got-hacked-by-us-b1e012a3c9a8

Soundness – Paper / Setup



Background

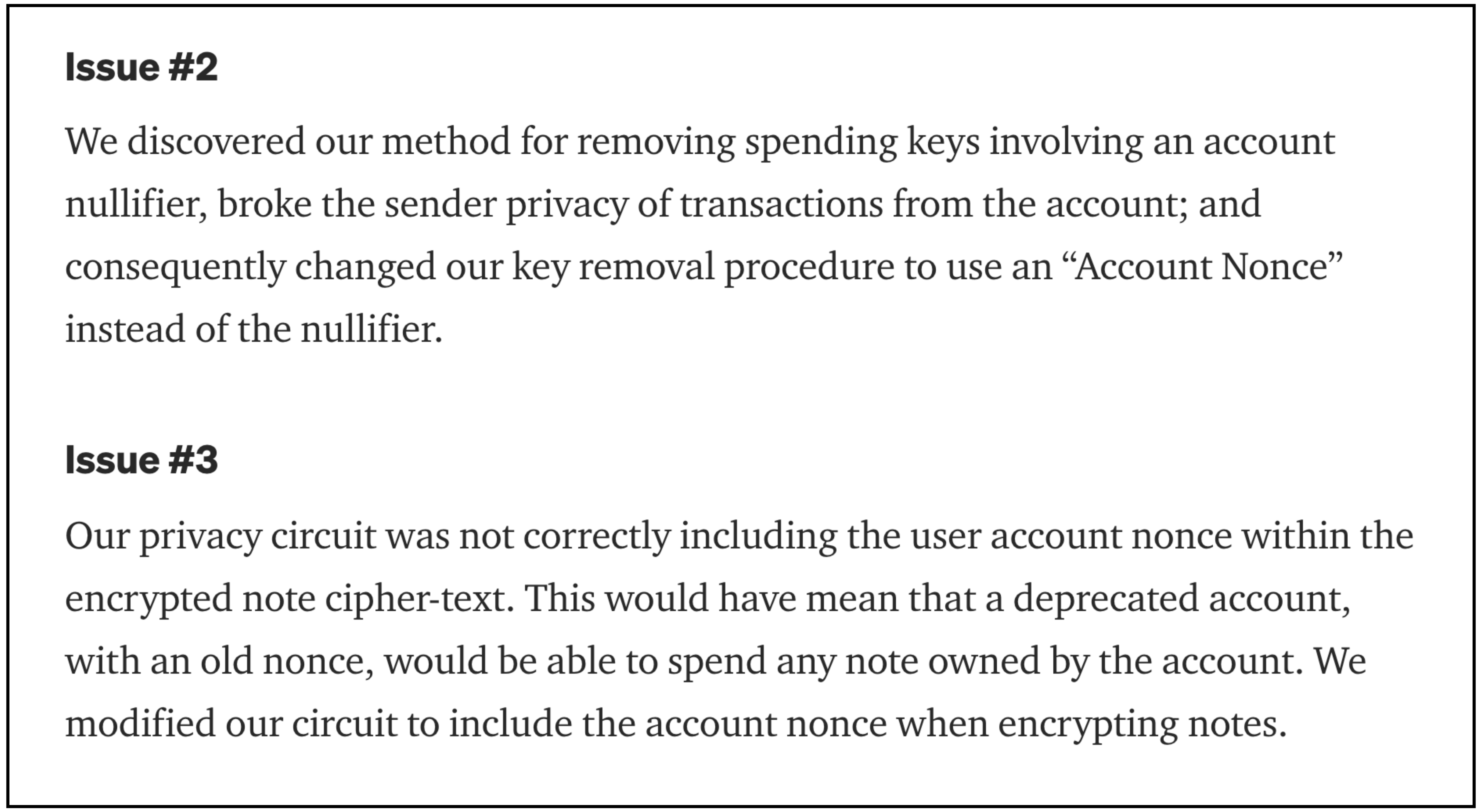
On March 1, 2018, Ariel Gabizon, a cryptographer employed by the Zcash Company at the time, discovered a subtle cryptographic flaw in the [BCTV14] paper that describes the zk-SNARK construction used in the original launch of Zcash. The flaw allows an attacker to create counterfeit shielded value in any system that depends on parameters which are generated as described by the paper.

This vulnerability is so subtle that it evaded years of analysis by expert cryptographers focused on zero-knowledge proving systems and zk-SNARKs. In an analysis [Parno15] in 2015, Bryan Parno from Microsoft Research discovered a different mistake in the paper. However, the vulnerability we discovered appears to have evaded his analysis. The vulnerability also appears in the subversion zero-knowledge SNARK scheme of [Fuchsbauer17], where an adaptation of [BCTV14] inherits the flaw. The vulnerability also appears in the ADSNARK construction described in [BBFR14]. Finally, the vulnerability evaded the Zcash Company's own cryptography team, which includes experts in the field that had identified several flaws in other parts of the system.

Theoretical flaw in the paper's setup description (sensitive values not cleared)

https://electriccoin.co/blog/zcash-counterfeiting-vulnerability-successfully-remediated/

Zero-knowledge – Application (Aztec)



Issue #2

We discovered our method for removing spending keys involving an account nullifier, broke the sender privacy of transactions from the account; and consequently changed our key removal procedure to use an "Account Nonce" instead of the nullifier.

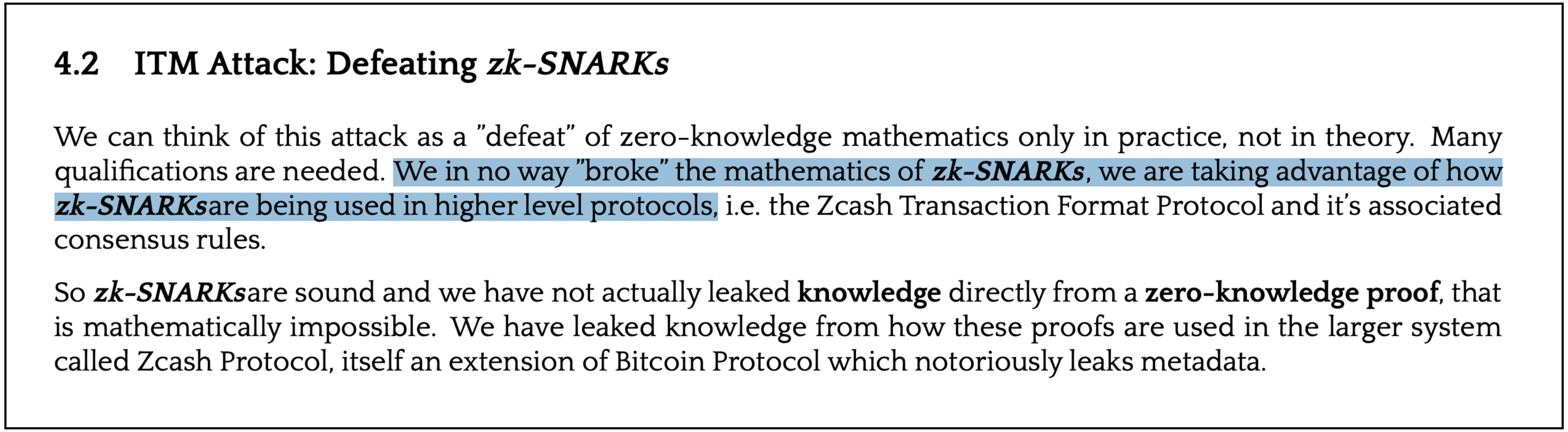
Issue #3

Our privacy circuit was not correctly including the user account nonce within the encrypted note cipher-text. This would have mean that a deprecated account, with an old nonce, would be able to spend any note owned by the account. We modified our circuit to include the account nonce when encrypting notes.

Nonces not correctly set by the application, breaking privacy

https://medium.com/@jaosef/54dff729a24f (Aztec 2.0 Pre-Launch Notes)

Zero-knowledge – Application (Zcash)



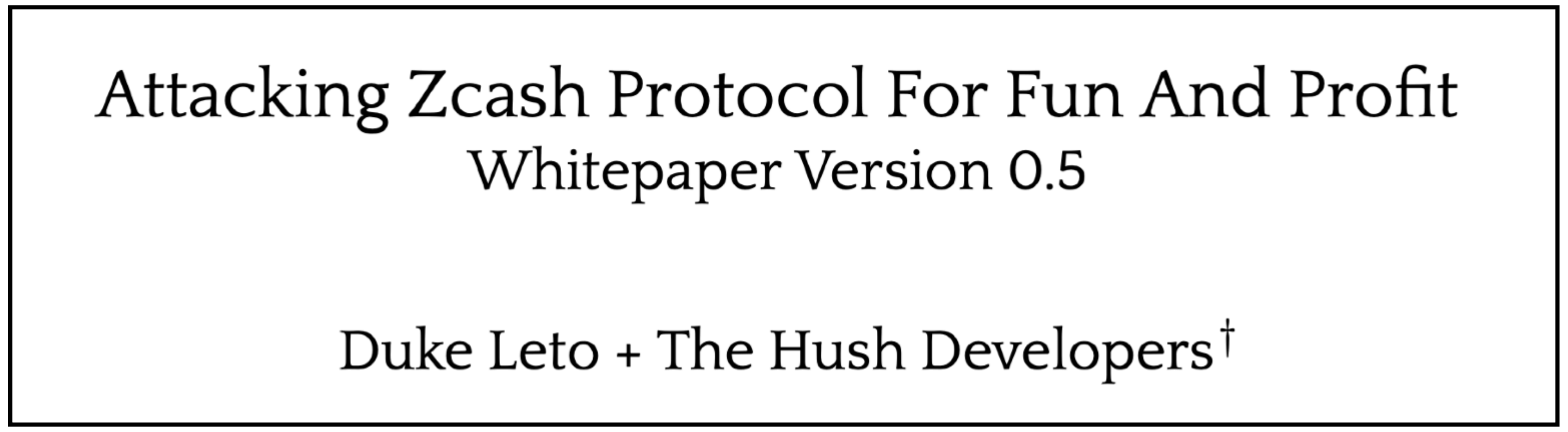
4.2 ITM Attack: Defeating *zk-SNARKs*

We can think of this attack as a ”defeat” of zero-knowledge mathematics only in practice, not in theory. Many qualifications are needed. We in no way ”broke” the mathematics of ***zk-SNARKs***, we are taking advantage of how ***zk-SNARKs***are being used in higher level protocols, i.e. the Zcash Transaction Format Protocol and it’s associated consensus rules.

So ***zk-SNARKs***are sound and we have not actually leaked **knowledge** directly from a **zero-knowledge proof**, that is mathematically impossible. We have leaked knowledge from how these proofs are used in the larger system called Zcash Protocol, itself an extension of Bitcoin Protocol which notoriously leaks metadata.

Correlations between (shielded) transactions leaking exploitable information

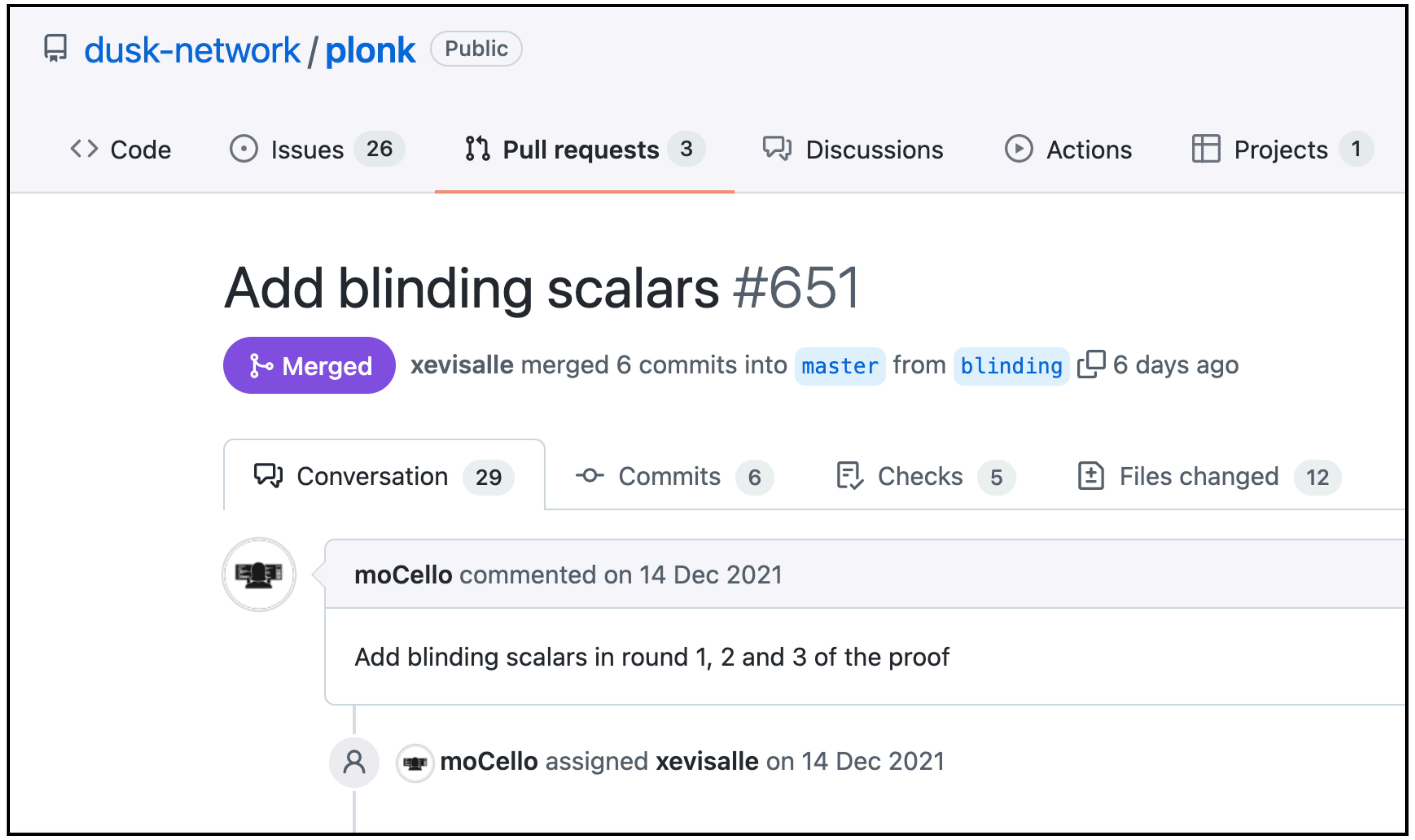
https://eprint.iacr.org/2020/627.pdf



Attacking Zcash Protocol For Fun And Profit
Whitepaper Version 0.5

Duke Leto + The Hush Developers[†]

Zero-knowledge – Prover (Plonkup)


dusk-network / plonk Public

Code · Issues 26 · Pull requests 3 · Discussions · Actions · Projects 1

Add blinding scalars #651

Merged xevisalle merged 6 commits into master from blinding 6 days ago

Conversation 29 · Commits 6 · Checks 5 · Files changed 12

moCello commented on 14 Dec 2021

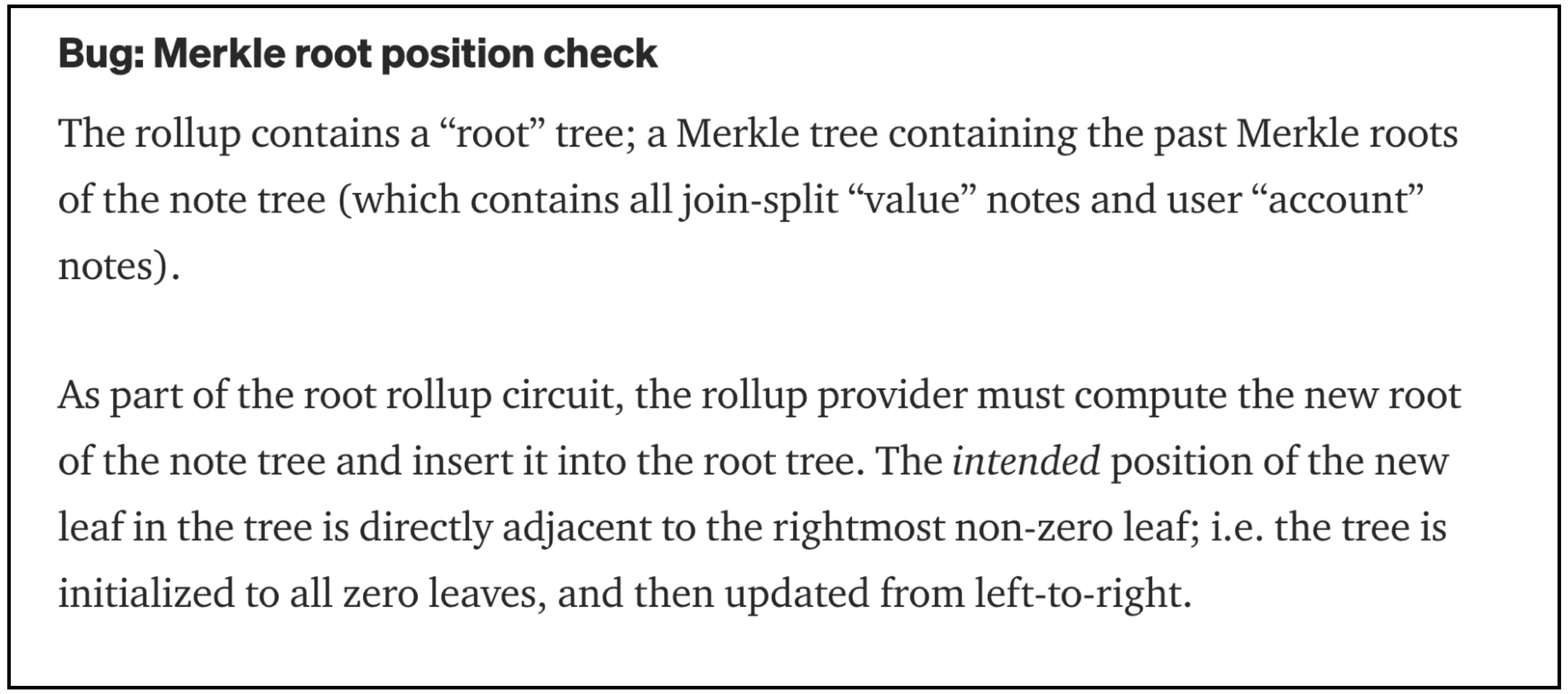
Add blinding scalars in round 1, 2 and 3 of the proof

moCello assigned xevisalle on 14 Dec 2021

Missing (randomized) blinding to hide private inputs – *potential* ZK loss

https://github.com/dusk-network/plonk/pull/651

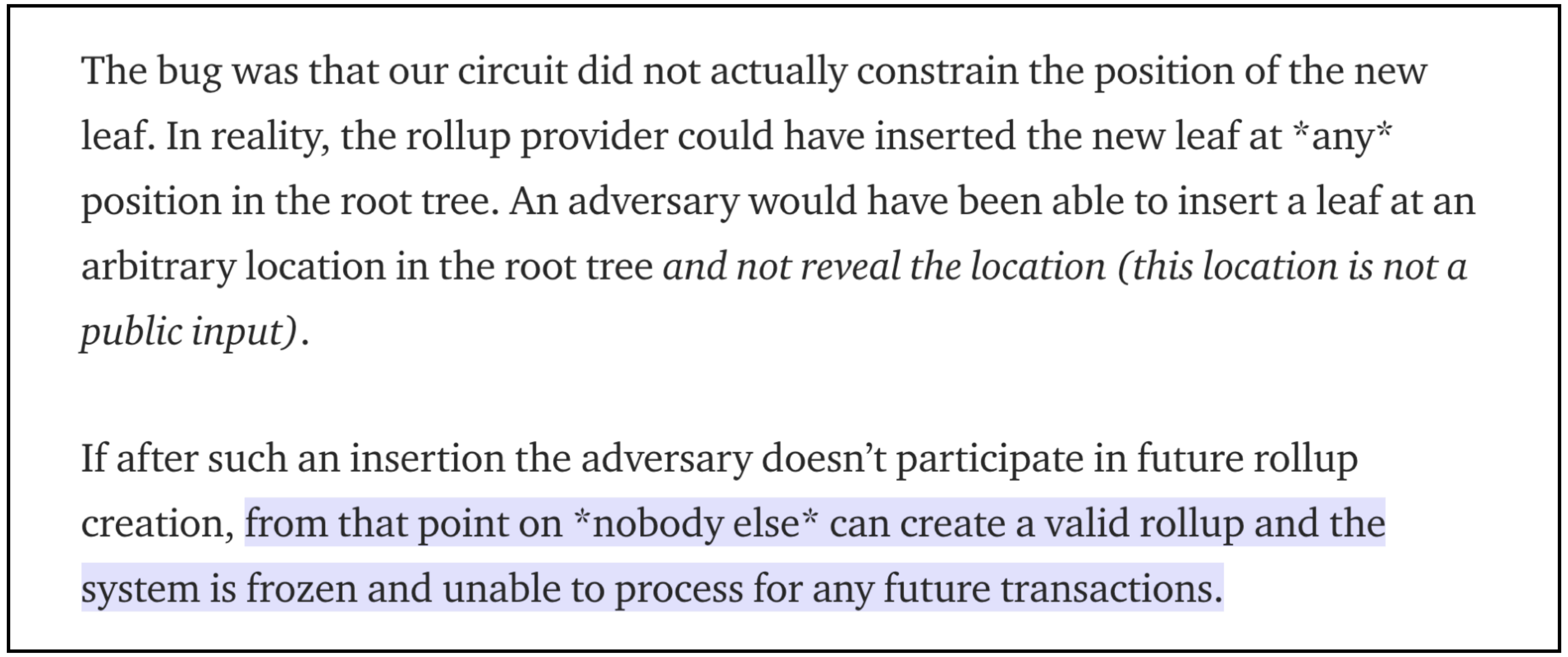
DoS – Merkle tree



Bug: Merkle root position check

The rollup contains a "root" tree; a Merkle tree containing the past Merkle roots of the note tree (which contains all join-split "value" notes and user "account" notes).

As part of the root rollup circuit, the rollup provider must compute the new root of the note tree and insert it into the root tree. The *intended* position of the new leaf in the tree is directly adjacent to the rightmost non-zero leaf; i.e. the tree is initialized to all zero leaves, and then updated from left-to-right.



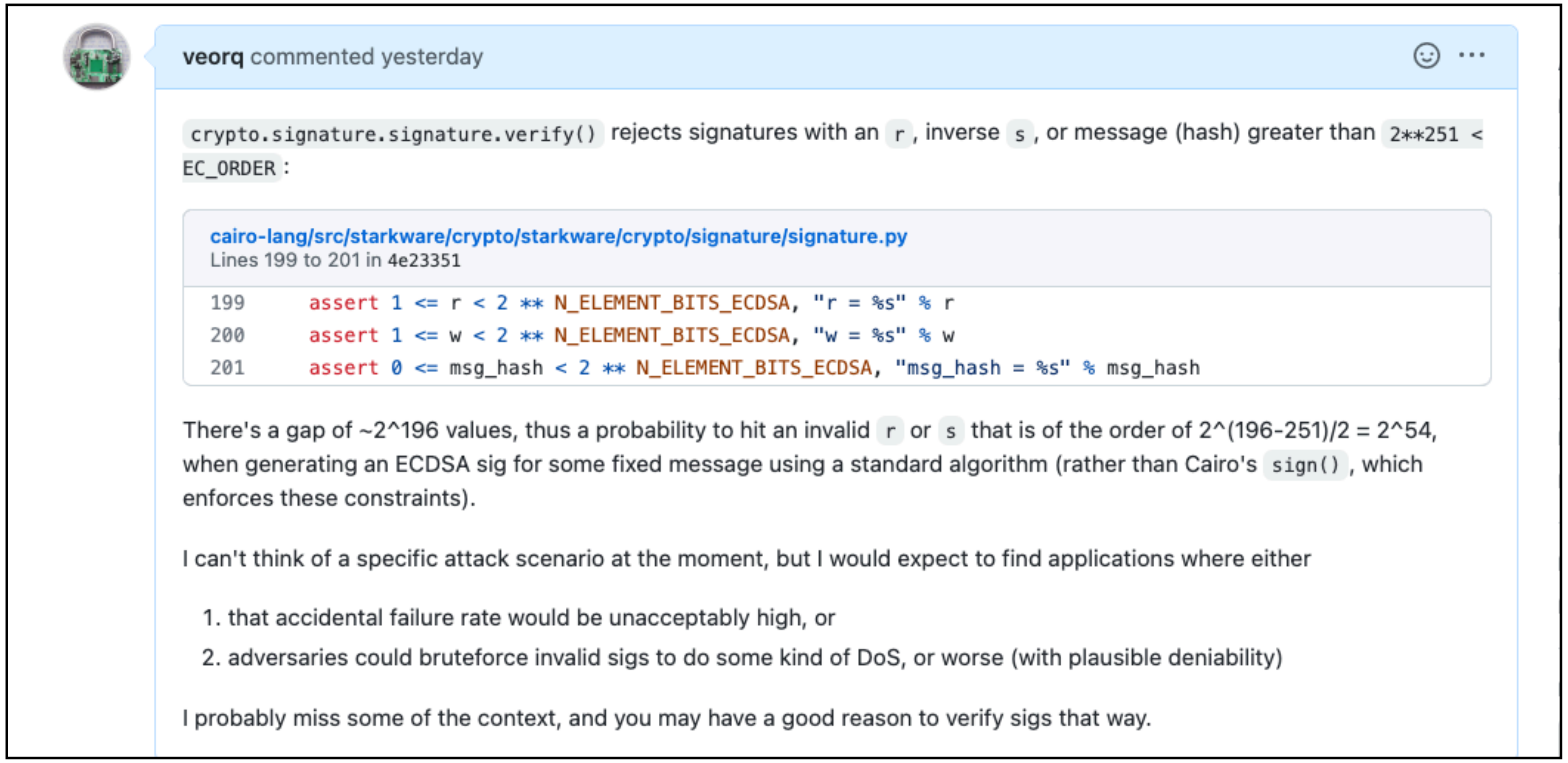
The bug was that our circuit did not actually constrain the position of the new leaf. In reality, the rollup provider could have inserted the new leaf at *any* position in the root tree. An adversary would have been able to insert a leaf at an arbitrary location in the root tree *and not reveal the location (this location is not a public input)*.

If after such an insertion the adversary doesn't participate in future rollup creation, from that point on *nobody else* can create a valid rollup and the system is frozen and unable to process for any future transactions.

Incomplete tree constraints, leading to a freeze of the rollup validation

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

DoS / Completeness? – DSL / Signatures



veorq commented yesterday

`crypto.signature.signature.verify()` rejects signatures with an `r`, inverse `s`, or message (hash) greater than `2**251 < EC_ORDER`:

cairo-lang/src/starkware/crypto/starkware/crypto/signature/signature.py
Lines 199 to 201 in 4e23351

```
    assert 1 <= r < 2 ** N_ELEMENT_BITS_ECDSA, "r = %s" % r
    assert 1 <= w < 2 ** N_ELEMENT_BITS_ECDSA, "w = %s" % w
    assert 0 <= msg_hash < 2 ** N_ELEMENT_BITS_ECDSA, "msg_hash = %s" % msg_hash
```

There's a gap of ~2^196 values, thus a probability to hit an invalid `r` or `s` that is of the order of 2^(196-251)/2 = 2^54, when generating an ECDSA sig for some fixed message using a standard algorithm (rather than Cairo's `sign()`, which enforces these constraints).

I can't think of a specific attack scenario at the moment, but I would expect to find applications where either

1. that accidental failure rate would be unacceptably high, or
2. adversaries could bruteforce invalid sigs to do some kind of DoS, or worse (with plausible deniability)

I probably miss some of the context, and you may have a good reason to verify sigs that way.

Valid signatures rejected, risk initially deemed negligible

https://github.com/starkware-libs/cairo-lang/issues/39

Other types of bugs

- **Crypto** issues such as:
  - Pedersen bases generation/uniqueness
  - Padding scheme in algebraic hashes and commitments
  - Non-conform implementations of crypto schemes (e.g. Poseidon algebra bugs)
  - Insufficient data being “Fiat-Shamir'd" from the transcript
- **Composability**: unsafe interactions between nested proof systems
- **Side channels**: Non-ct code, RAM leakage, speculative execution leaks

Conclusions

😌 **Why not be too scared?**

- Robust code and frameworks (e.g. Rust projects such as arkworks and zkcrypto)
- DSLs (Cairo, Leo, etc.) make it easier to write safe code
- Relatively narrow attack surface in practice

Conclusions

😌 Why not be too scared?

- Robust code and frameworks (e.g. Rust projects such as arkworks and zkcrypto)
- DSLs (Cairo, Leo, etc.) make it easier to write safe code
- Relatively narrow attack surface in practice

😱 Why be scared?

- Few people understand zkSNARKs, even fewer can find bugs
- Lack of tooling (wrt testing, fuzzing, verification)
- More ZKPs used => more $$$ at stake => greater RoI for vuln researchers

Thank you!

JP Aumasson
@veorq

CSO @ taurushq.com

Thanks *to Aleo, Protocol Labs, Kobi Gurkan, Adrian Hamelink, Lúcás Meier, Mathilde Raynal*