What don’t we know? Understanding Security Vulnerabilities in SNARKs

Stefanos Chaliasos

ZKSecurity, Imperial College London

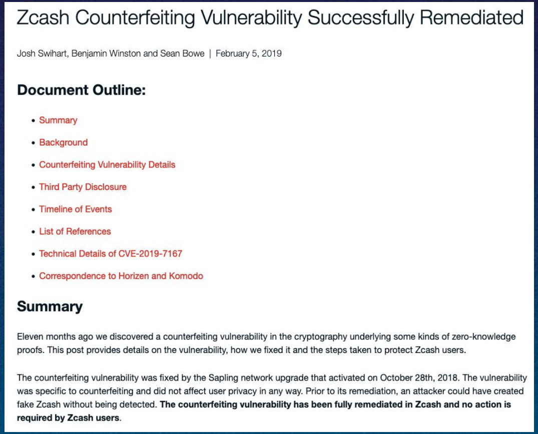
Jens Ernstberger - *Technical University of Munich*
David Theodore - *Ethereum Foundation*
David Wong - *ZKSecurity*
Mohammad Jahanara - *Scroll Foundation*
Benjamin Livshits - *Imperial College London*

The state of ZKP applications

- ➔ zk Rollups: > $4b
- ➔ Zcash
- ➔ zk apps
  - ◆ zkLogin
  - ◆ zkemail
  - ◆ Zk-bridges
  - ◆ Private payments / Pools
- ➔ Private Programmable L1s/L2s
- ➔ off-chain apps

https://l2beat.com/scaling/summary

https://defillama.com/



Zcash Counterfeiting Vulnerability Successfully Remediated

Josh Swihart, Benjamin Winston and Sean Bowe | February 5, 2019

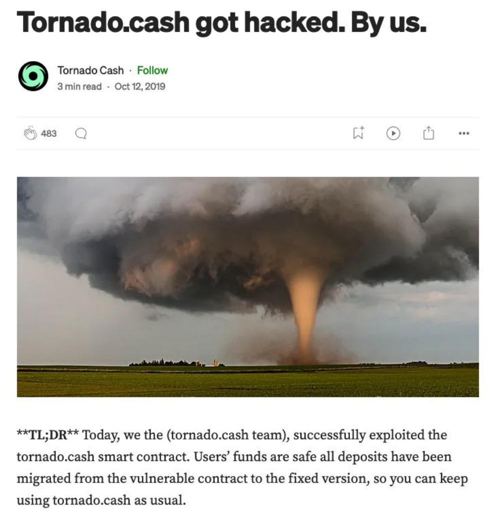
Document Outline:

- Summary
- Background
- Counterfeiting Vulnerability Details
- Third Party Disclosure
- Timeline of Events
- List of References
- Technical Details of CVE-2019-7167
- Correspondence to Horizen and Komodo

Summary

Eleven months ago we discovered a counterfeiting vulnerability in the cryptography underlying some kinds of zero-knowledge proofs. This post provides details on the vulnerability, how we fixed it and the steps taken to protect Zcash users.

The counterfeiting vulnerability was fixed by the Sapling network upgrade that activated on October 28th, 2018. The vulnerability was specific to counterfeiting and did not affect user privacy in any way. Prior to its remediation, an attacker could have created fake Zcash without being detected. **The counterfeiting vulnerability has been fully remediated in Zcash and no action is required by Zcash users**.

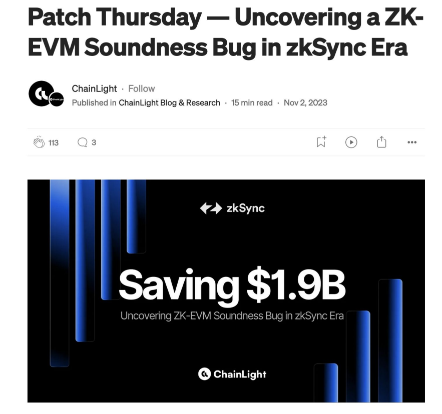


Tornado.cash got hacked. By us.

Tornado Cash · Follow
3 min read · Oct 12, 2019

483

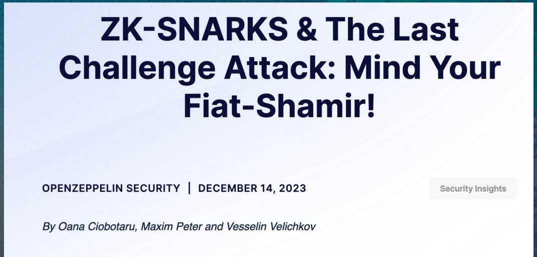
TL;DR Today, we the (tornado.cash team), successfully exploited the tornado.cash smart contract. Users' funds are safe all deposits have been migrated from the vulnerable contract to the fixed version, so you can keep using tornado.cash as usual.



Patch Thursday — Uncovering a ZK-EVM Soundness Bug in zkSync Era

ChainLight · Follow
Published in ChainLight Blog & Research · 15 min read · Nov 2, 2023

113 3



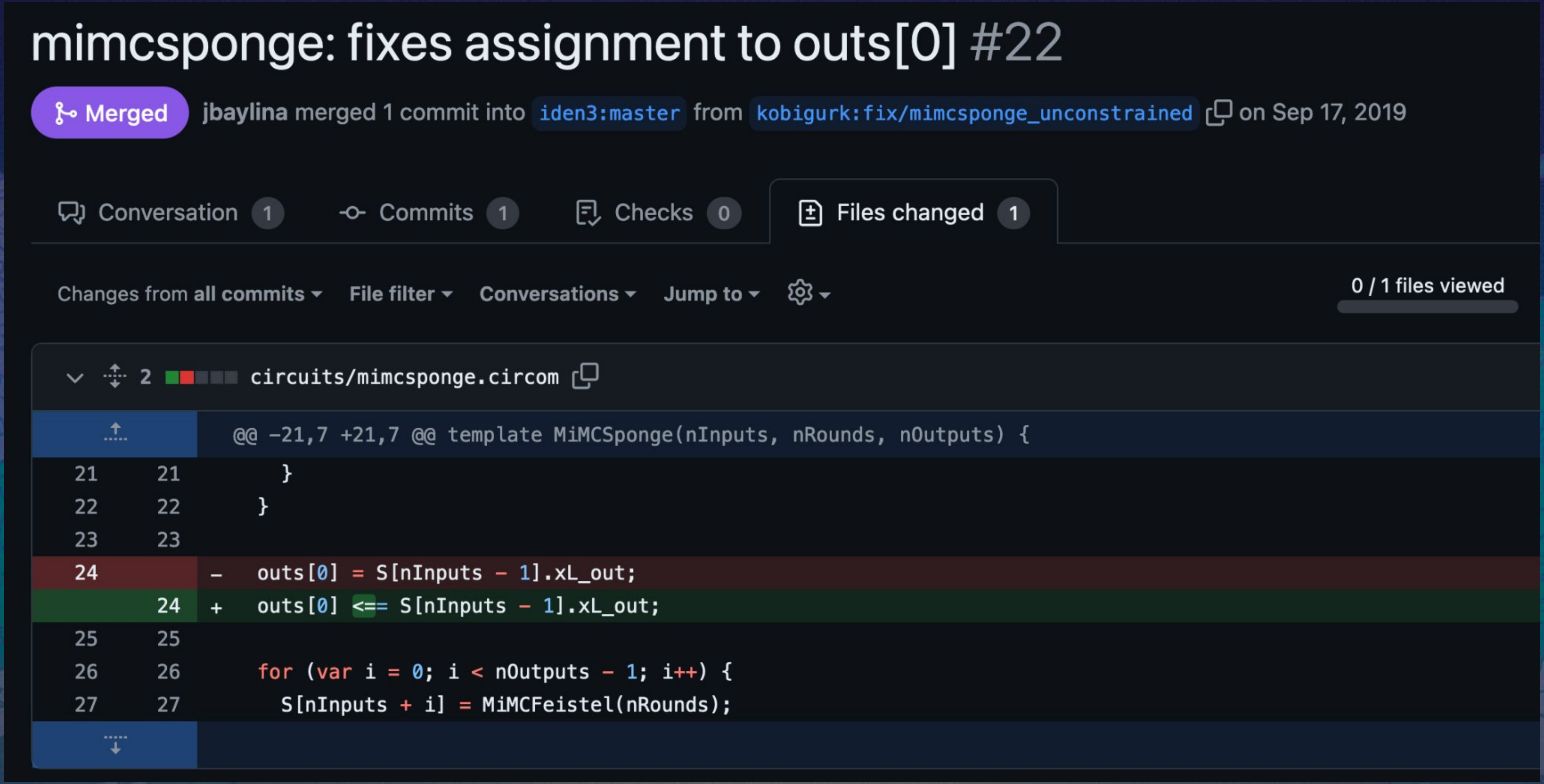
ZK-SNARKS & The Last Challenge Attack: Mind Your Fiat-Shamir!

OPENZEPPELIN SECURITY | DECEMBER 14, 2023

Security Insights

By Oana Ciobotaru, Maxim Peter and Vesselin Velichkov

Example Circuit Vulnerability



mimcsponge: fixes assignment to outs[0] #22

Merged jbaylina merged 1 commit into iden3:master from kobigurk:fix/mimcsponge_unconstrained on Sep 17, 2019

circuits/mimcsponge.circom

```
@@ -21,7 +21,7 @@ template MiMCSponge(nInputs, nRounds, nOutputs) {
      }
    }

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

    for (var i = 0; i < nOutputs - 1; i++) {
      S[nInputs + i] = MiMCFeistel(nRounds);
```

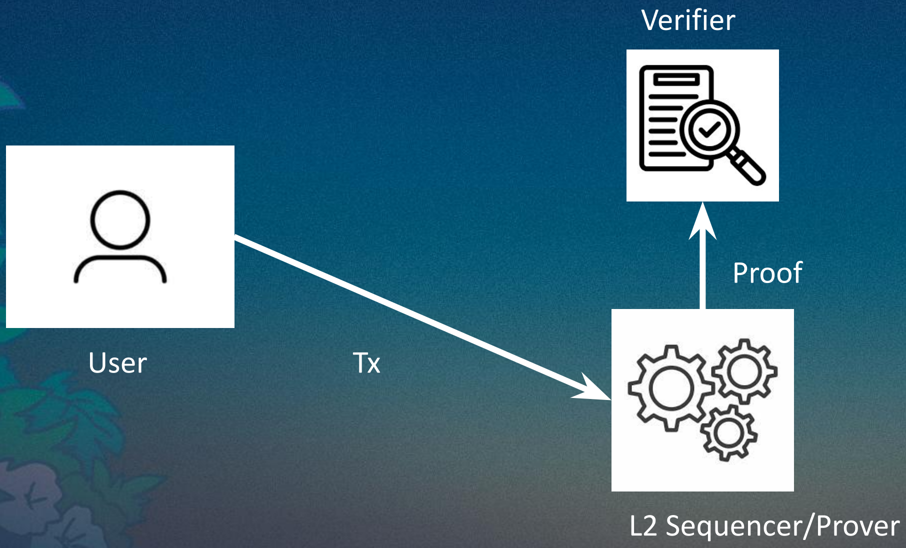
Towards Understanding implementation vulnerabilities in ZKPs

Properties

- **Knowledge Soundness**
  - A dishonest prover cannot convince the verifier of an invalid statement, except with negligible probability.
- **Perfect Completeness**
  - An honest prover can always convince the verifier of the correctness of a valid statement
- **Zero Knowledge**
  - The proof π reveals nothing about the witness *w*, beyond its existence

Threat Model – Adversaries

- ➔ **Network Adversary**: observe the system and its public values
- ➔ **Adversarial User**: submit inputs for proof generation to a non-malicious prover
- ➔ **Adversarial Prover**: ability to produce and submit proofs



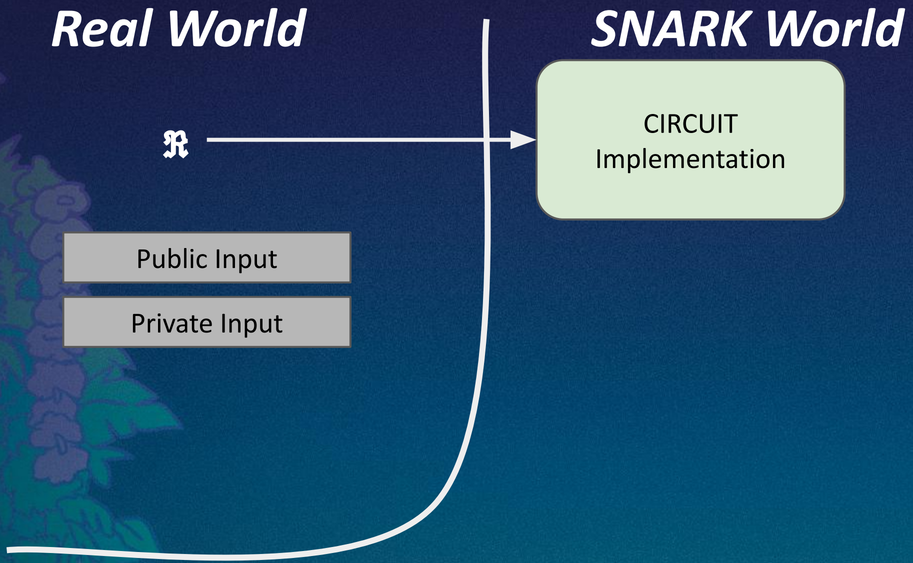
Threat Model – Vulnerability Impact

- **Breaking Soundness**
  - A prover can convince a verifier of a false statement
- **Breaking Completeness**
  - Cannot verify proofs for valid statements
- **Breaking Zero-Knowledge**
  - Information leakage about the private witness

Analyzing and Classify ZKP Vulnerabilities

- 141 Bugs
  - Audit Reports
  - Vulnerability Disclosures
  - Bug Tracker
- Goals:
  - Split vulnerabilities in Layers
  - Taxonomy of vulnerabilities

SNARKs Layers (Workflow)



Real World

SNARK World

$\mathfrak{R}$

CIRCUIT Implementation

Public Input

Private Input

Circuit Layer

```
inp1, inp2, tmp3, tmp4, out5


tmp3 = inp1 + inp2
tmp4 = inp2 * 4
out5 = tmp3 * tmp4
```

Computation

```
inp1, inp2, tmp3, tmp4, out5


tmp3 == inp1 + inp2
tmp4 == inp2 * 4
out5 == tmp3 * tmp4
```

Constraints

Circuit Layer – Vulnerabilities

→ Underconstrained Vulnerabilities

→ Overconstrained Vulnerabilities

→ Computation/Hints Errors

Circuit Layer – Root Causes

- Limited set of constraints -> Assigned but not Constrained
- Common usage of selectors -> Incorrect Custom Gates
- Field arithmetic -> Arithmetic Field Errors
- Configurations / lack of semantics -> Unsafe Reuse of Circuit

Different programming model

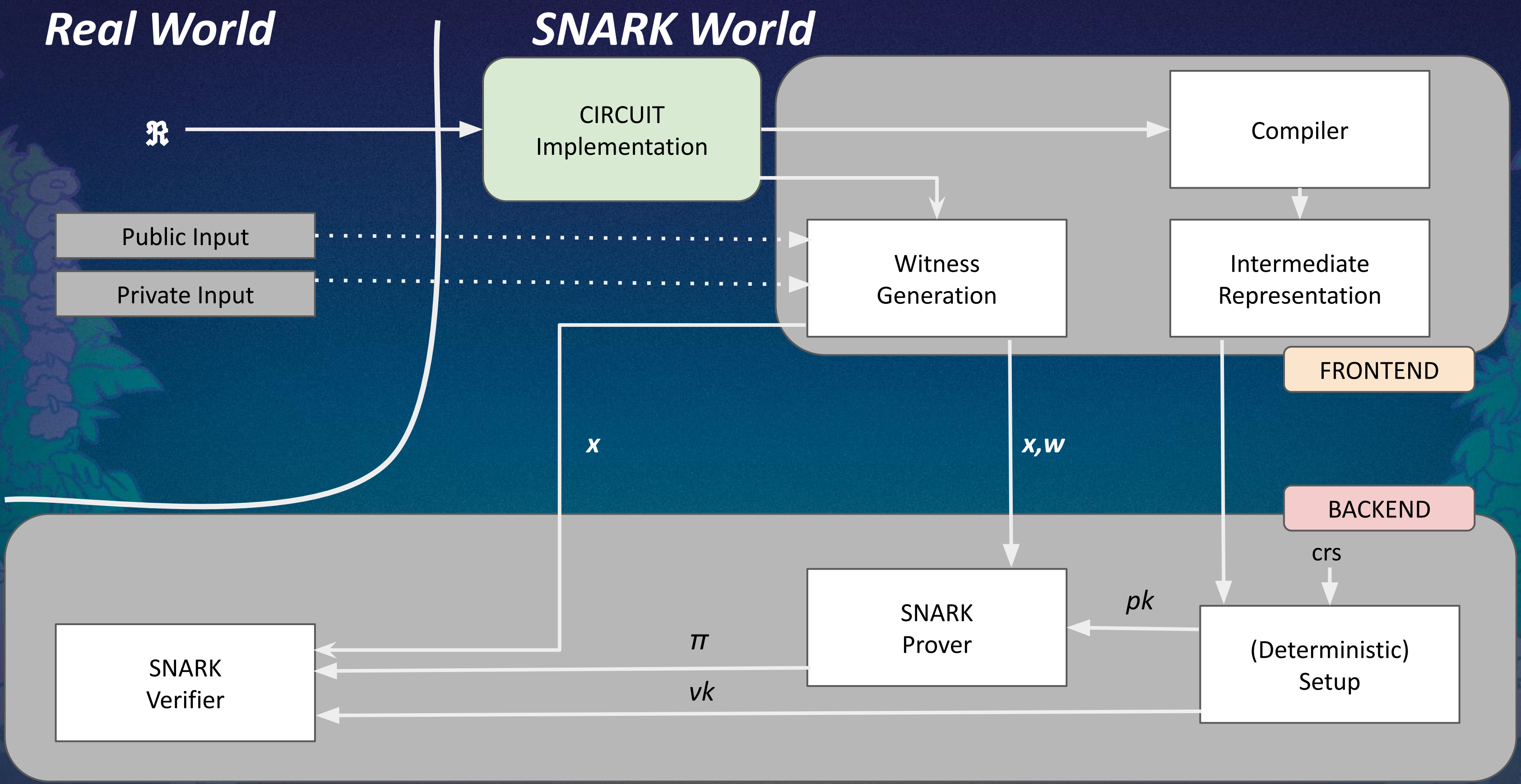
- Costs of constraints / Complexity ->
  - Missing Input Constraints
  - Wrong translation of logic into constraints
  - Out-of-circuit Computation Not Being Constrained

Optimizations /
Cryptography at the outer layer

- Specification issues -> Bad Circuit/Protocol Design
- Usual mistakes -> Other Programming Errors (e.g., API misuse, incorrect indexing in arrays)

Common Errors

SNARKs Layers (Workflow)



Frontend and Backend

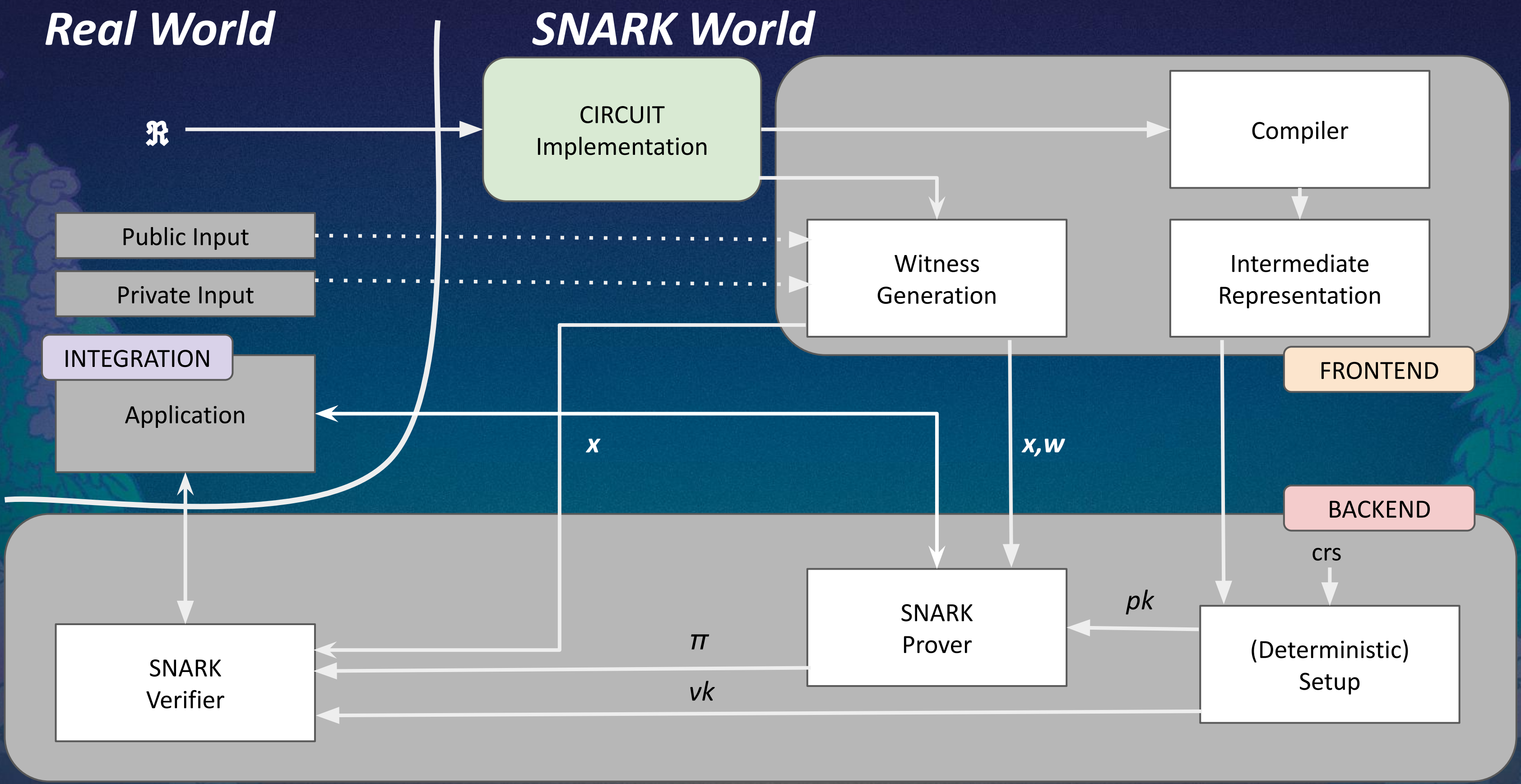
Frontend

- → Incorrect Constraint Compilation
- → Witness Generation Error

Backend

- → Setup Error
- → Prover Error
- → **Unsafe Verifier**

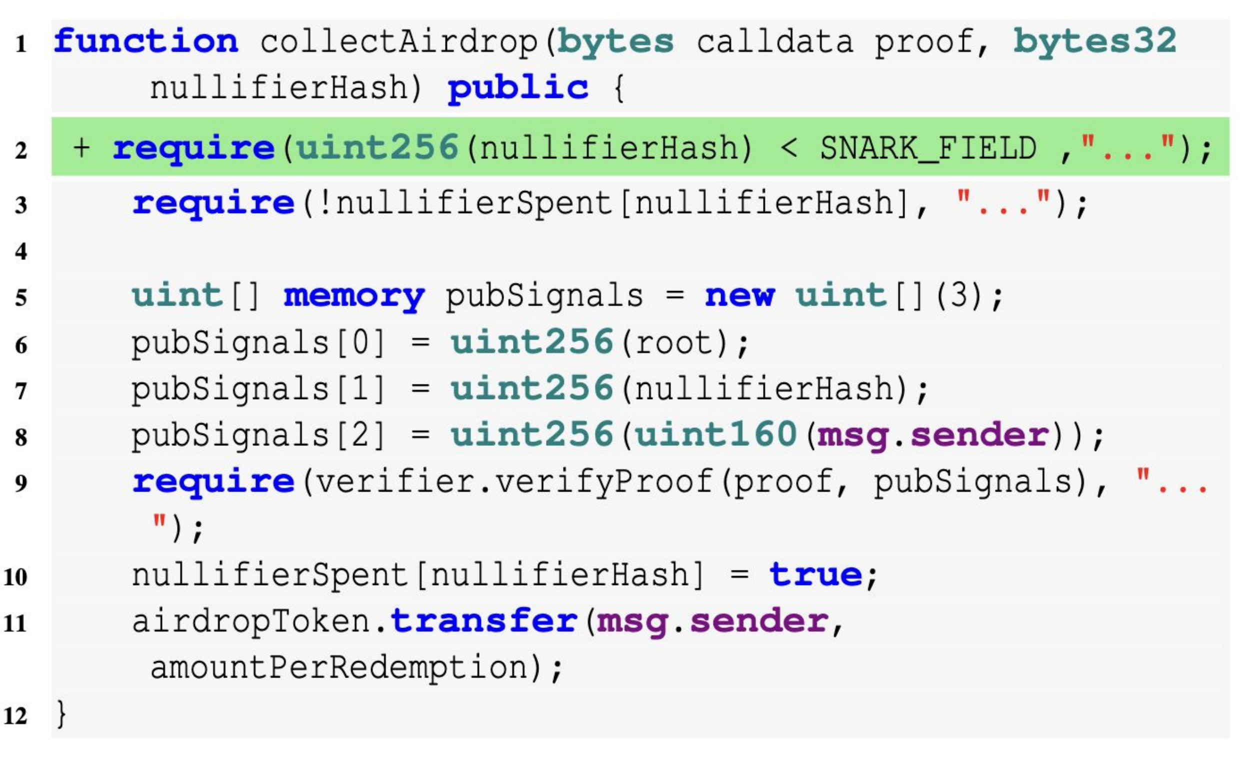
SNARKs Layers (Workflow)



Integration Layer

- → **Passing Unchecked Data**
- → Proof Delegation Error
- → Proof Composition Error
- → ZKP Complementary Logic Error

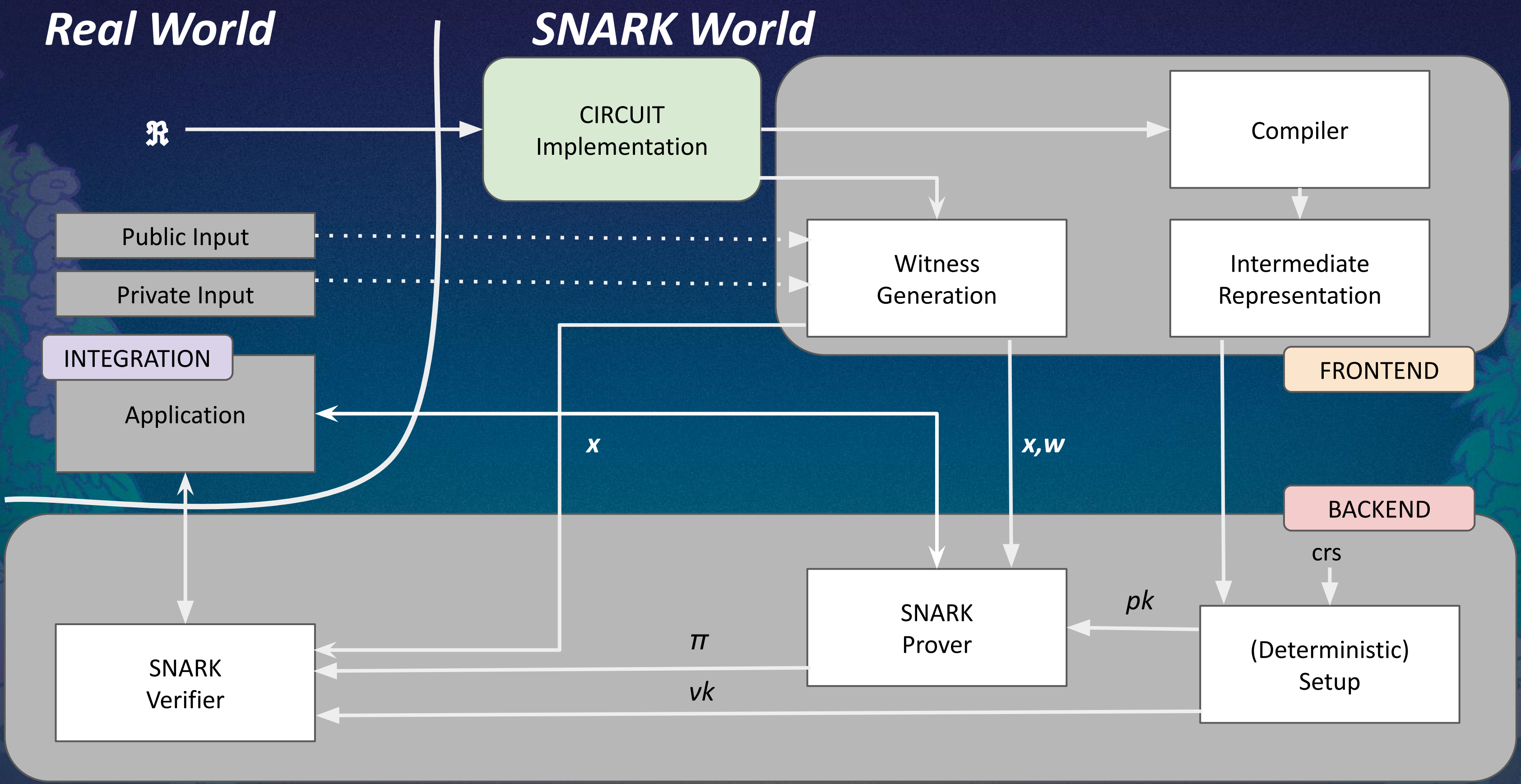
Integration Example (Missing Input Validation)



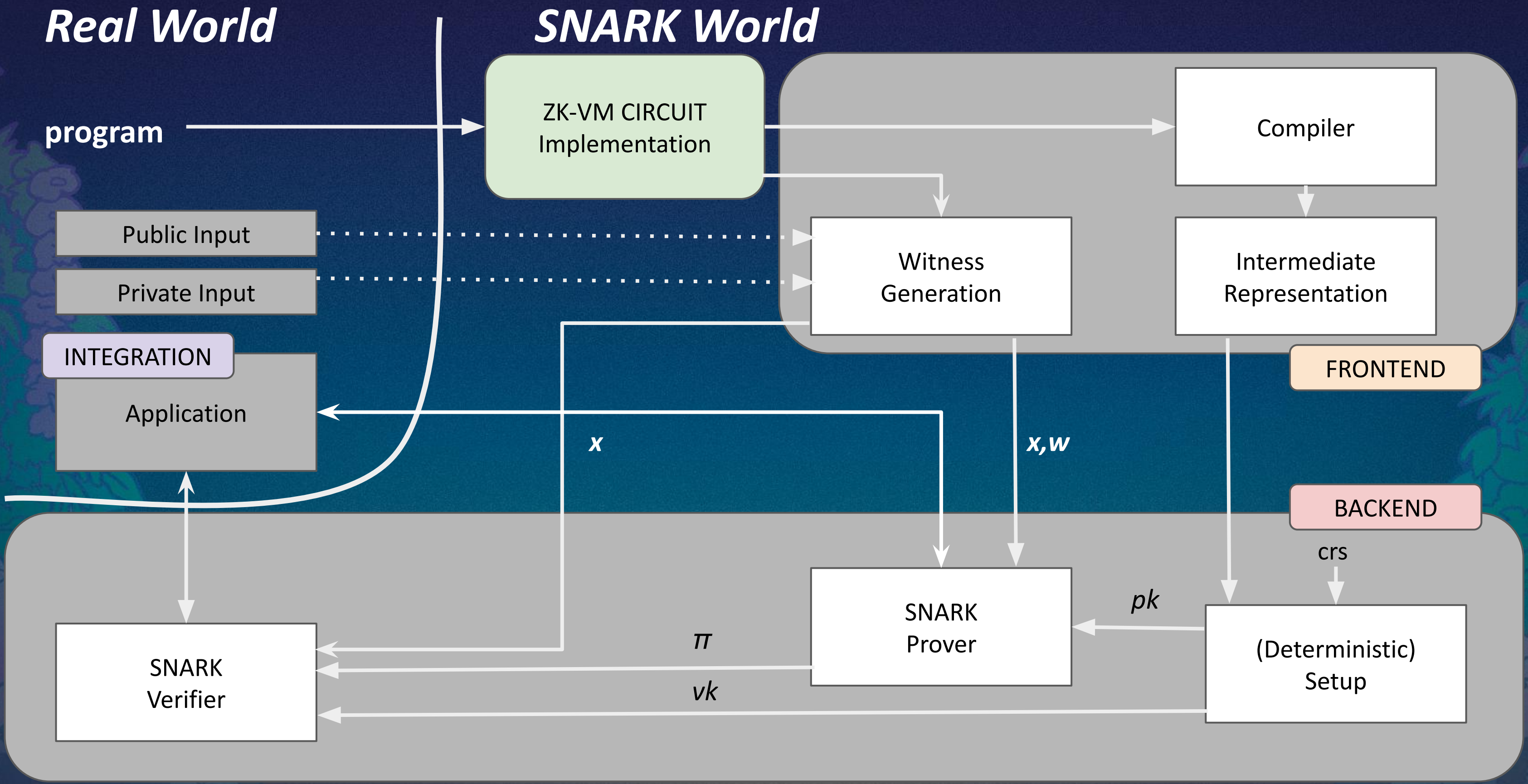
```
function collectAirdrop(bytes calldata proof, bytes32
     nullifierHash) public {
 + require(uint256(nullifierHash) < SNARK_FIELD ,"...");
    require(!nullifierSpent[nullifierHash], "...");

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

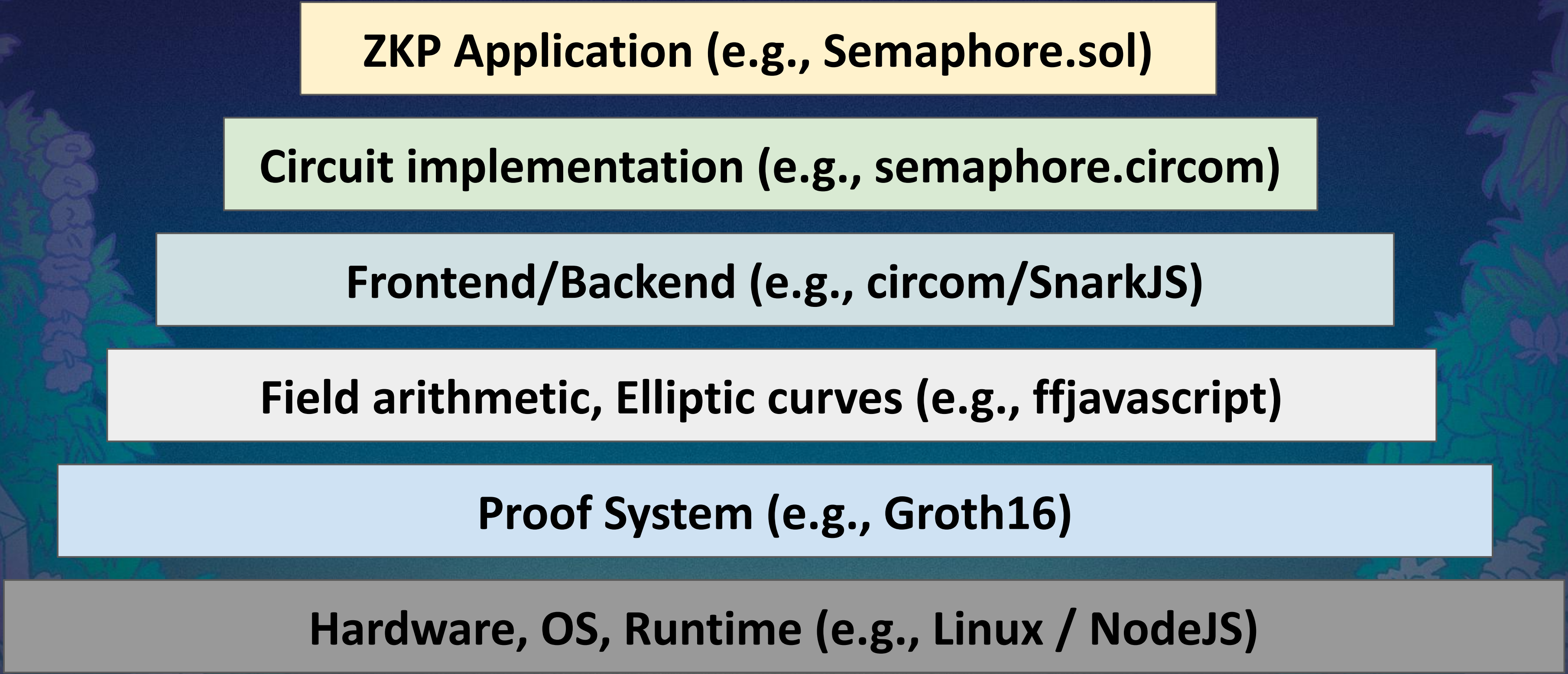
SNARKs Layers (Workflow)



SNARKs Layers (Workflow) – ZK-VMs



SNARKs Layers (Workflow) – Hierarchy



ZKP Application (e.g., Semaphore.sol)

Circuit implementation (e.g., semaphore.circom)

Frontend/Backend (e.g., circom/SnarkJS)

Field arithmetic, Elliptic curves (e.g., ffjavascript)

Proof System (e.g., Groth16)

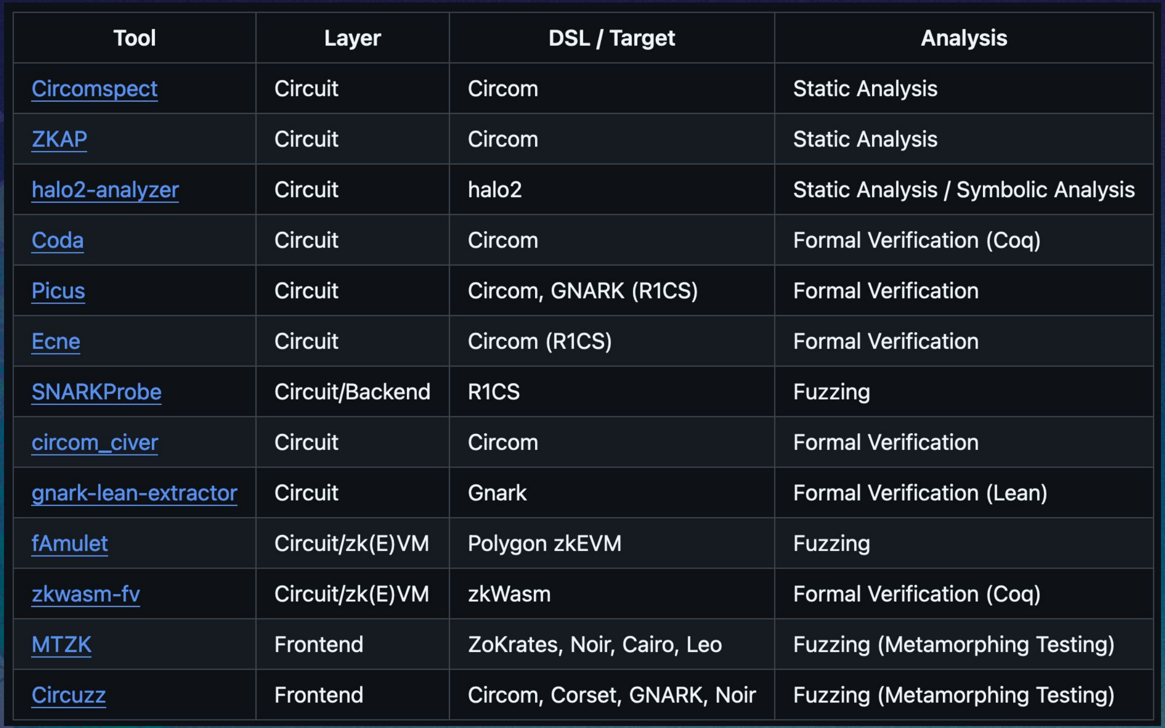
Hardware, OS, Runtime (e.g., Linux / NodeJS)

Analyzing and Classify ZKP Vulnerabilities

- 141 Bugs
  - Audit Reports
  - Vulnerability Disclosures
  - Bug Tracker

Layer	Soundness	Completeness	Zero Knowledge
Integration	11	2	0
Circuit	94	5	0
Frontend	2	4	0
Backend	17	3	3
Total	124	14	3

Security Tooling



Tool	Layer	DSL / Target	Analysis
Circomspect	Circuit	Circom	Static Analysis
ZKAP	Circuit	Circom	Static Analysis
halo2-analyzer	Circuit	halo2	Static Analysis / Symbolic Analysis
Coda	Circuit	Circom	Formal Verification (Coq)
Picus	Circuit	Circom, GNARK (R1CS)	Formal Verification
Ecne	Circuit	Circom (R1CS)	Formal Verification
SNARKProbe	Circuit/Backend	R1CS	Fuzzing
circom_civer	Circuit	Circom	Formal Verification
gnark-lean-extractor	Circuit	Gnark	Formal Verification (Lean)
fAmulet	Circuit/zk(E)VM	Polygon zkEVM	Fuzzing
zkwasm-fv	Circuit/zk(E)VM	zkWasm	Formal Verification (Coq)
MTZK	Frontend	ZoKrates, Noir, Cairo, Leo	Fuzzing (Metamorphing Testing)
Circuzz	Frontend	Circom, Corset, GNARK, Noir	Fuzzing (Metamorphing Testing)

https://github.com/StefanosChaliasos/Awesome-ZKP-Security

- → Targeting specific DSLs / vulnerability classes (Circuits)
- → Scalability issues

Conclusions (1/2)

- Why do we have bugs?
  - “not just maths”
    - Bugs in the implementations can break all the properties
  - “the poor user is given enough rope with which to hang himself”
    - Exposing cryptography to the outer layers
    - Missing of fundamental abstractions
    - Complexity / Different Threat Model
  - Lack of specifications

Conclusions (2/2)

➔ What can we do?

- ◆ More learning resources
- ◆ Specifications
- ◆ Easier and more secure programming languages
- ◆ Better testing/security tooling (from testing frameworks to FV)

SoK: What don't we know? Understanding Security Vulnerabilities in SNARKs

Stefanos Chaliasos
Imperial College London

Jens Ernstberger

David Theodore
Ethereum Foundation

David Wong
zkSecurity

Mohammad Jahanara
Scroll Foundation

Benjamin Livshits
Imperial College London & Matter Labs

Abstract

Zero-knowledge proofs (ZKPs) have evolved from being a theoretical concept providing privacy and verifiability to having practical, real-world implementations, with SNARKs (Succinct Non-Interactive Argument of Knowledge) emerging as one of the most significant innovations. Prior work has mainly focused on designing more efficient SNARK systems and providing security proofs for them. Many think of SNARKs as "just math," implying that what is proven to be correct and secure *is* correct in practice. In contrast, this paper focuses on assessing end-to-end security properties of real-life SNARK *implementations*. We start by building foundations with a system model and by establishing threat models and defining adversarial roles for systems that use SNARKs. Our study encompasses an extensive analysis of 141 actual vulnerabilities in SNARK implementations, providing a detailed taxonomy to aid developers and security researchers in understanding the security threats in systems employing SNARKs. Finally, we evaluate existing defense mechanisms and offer recommendations for enhancing the security of SNARK-based systems, paving the way for more robust and reliable implementations in the future.

1 Introduction

Zero-Knowledge Proofs (ZKPs) have undergone a remarkable evolution from their conceptual origins in the realm of complexity theory and cryptography [50, 51] to their current role as fundamental components that enable a wide array of practical applications [35]. Originally conceptualized as an interactive protocol where an untrusted prover could convince a verifier of the correctness of a computation without revealing any other information (zero-knowledge) [50], ZKPs have, over the past decade, transitioned from theory to practical widely used implementation [14, 16, 30, 69, 76, 84, 89, 93].

On the forefront of the practical application of *general-purpose* ZKPs are Succinct Non-interactive Argument of Knowledge (SNARKs) [25, 43, 47, 52, 82]. SNARKs are non-interactive protocols that allow the prover to generate a succinct proof. The proof is efficiently checked by the verifier, while maintaining three crucial properties: completeness, soundness, and zero-knowledge. What makes SNARKs particularly appealing is their general-purpose nature, allowing any computational statement represented as a *circuit* to be proven and efficiently verified. Typically, SNARKs are used to prove that for a given function f and a public input x, the prover knows a (private) witness w, such as $f(x,w) = y$. This capability allows SNARKs to be used in various applications, including ensuring data storage integrity [89], enhancing privacy in digital asset transfers [69, 93] and program execution [14, 16], as well as scaling blockchain infrastructure [62, 85, 86, 96]. Their versatility also extends to non-blockchain uses, such as in secure communication protocols [64, 92, 107] and in efforts to combat disinformation [31, 57, 59]. Unfortunately, developing and deploying systems that use SNARKs safely is a challenging task.

In this paper, we undertake a comprehensive analysis of publicly disclosed vulnerabilities in SNARK systems. Despite the existence of multiple security reports affecting such systems, the information tends to be scattered. Additionally, the complexity of SNARK-based systems and the unique programming model required for writing ZK circuits make it difficult to obtain a comprehensive understanding of the prevailing vulnerabilities and overall security properties of these systems. Traditional taxonomies for software vulnerabilities do not apply in the case of SNARKs; hence, we provide the seminal work that addresses this gap by providing a holistic taxonomy that highlights pitfalls in developing and using SNARKs. Specifically, we analyzed 141 vulnerability reports spanning nearly 6 years, from 2018 until 2024. Our study spans the entire SNARK stack, encompassing the theoretical foundations, frameworks used for writing and compiling circuits, circuit programs, and system deployments. We systematically categorize and investigate a wide array of vulnerabilities, uncovering multiple insights about the extent and causes of existing vulnerabilities, and potential mitigations.

Contributions.

Thank you!

https://arxiv.org/pdf/2402.15293

https://www.zksecurity.xyz/blog/