The Ouroboros of ZK: Why Verifying the Verifier Unlocks Longer-Term ZK Innovation

Dr. Ben Livshits
VP of Research at Matter Labs

ABOUT ME

- VP of Research at Matter Labs/ZKSync
- Professor at Imperial College London

- Work on the intersection of programming languages, security, and privacy
- The last 4-5 years I have focused on blockchain
- DeFi, fee mechanisms, effectiveness of audits and contract analysis tools, etc.

THE PATH OF ZK

- Move away from wasteful execution of most L1s

- ZK proof production –
  - efficiency problem
  - reliability and correctness problem

DECENTRALIZATION OF PROOF GENERATION

Training wheels

- centralized sequencer
- centralized prover setup
- powerful machines in a cloud setting

Decentralized setup

- decentralized or even shared sequencing
- prover pools
- prover markets and mechanism design

CAN WE DECENTRALIZE PROVING?

- Who the h*ll not? We see a number of new prover marketplaces popping up: Gevulot, Succinct, Snarketplace, etc.
- What is the prover becomes malicious?
- Oh, no worry, this is ZK, we have the verifier to save us

A lot of concerns about verifier correctness

Project	Verifier URL	LOC
Era	Verifier.sol	1,710
Polygon Hermez	FflonkVerifier.sol	1,244
Taiko	PlonkVerifier.yulp	1,835
Linea	PlonkVerifier.sol	948

Figure 1: Comparison on deployed verifiers, in terms of LOC. Links in the table are clickable.

ZK IS NOT “JUST MATH”

```
// Roots
// S_0 = roots_8(xi) = { h_0, h_0w_8, h_0w_8^2,
// h_0w_8^3, h_0w_8^4, h_0w_8^5, h_0w_8^6, h_0w_8^7 }
uint16 constant pH0w8_0 = 224;
uint16 constant pH0w8_1 = 256;
uint16 constant pH0w8_2 = 288;
uint16 constant pH0w8_3 = 320;
uint16 constant pH0w8_4 = 352;
uint16 constant pH0w8_5 = 384;
uint16 constant pH0w8_6 = 416;
uint16 constant pH0w8_7 = 448;

// S_1 = roots_4(xi) = { h_1, h_1w_4, h_1w_4^2, h_1w_4^3 }
uint16 constant pH1w4_0 = 480;
uint16 constant pH1w4_1 = 512;
uint16 constant pH1w4_2 = 544;
uint16 constant pH1w4_3 = 576;

// S_2 = roots_3(xi) U roots_3(xi omega)
// roots_3(xi) = { h_2, h_2w_3, h_2w_3^2 }
uint16 constant pH2w3_0 = 608;
uint16 constant pH2w3_1 = 640;
uint16 constant pH2w3_2 = 672;
// roots_3(xi omega) = { h_3, h_3w_3, h_3w_3^2 }
uint16 constant pH3w3_0 = 704;
uint16 constant pH3w3_1 = 736;
uint16 constant pH3w3_2 = 768;

uint16 constant pPi    = 800; // PI(xi)
uint16 constant pR0    = 832; // r0(y)
uint16 constant pR1    = 864; // r1(y)
uint16 constant pR2    = 896; // r2(y)
```

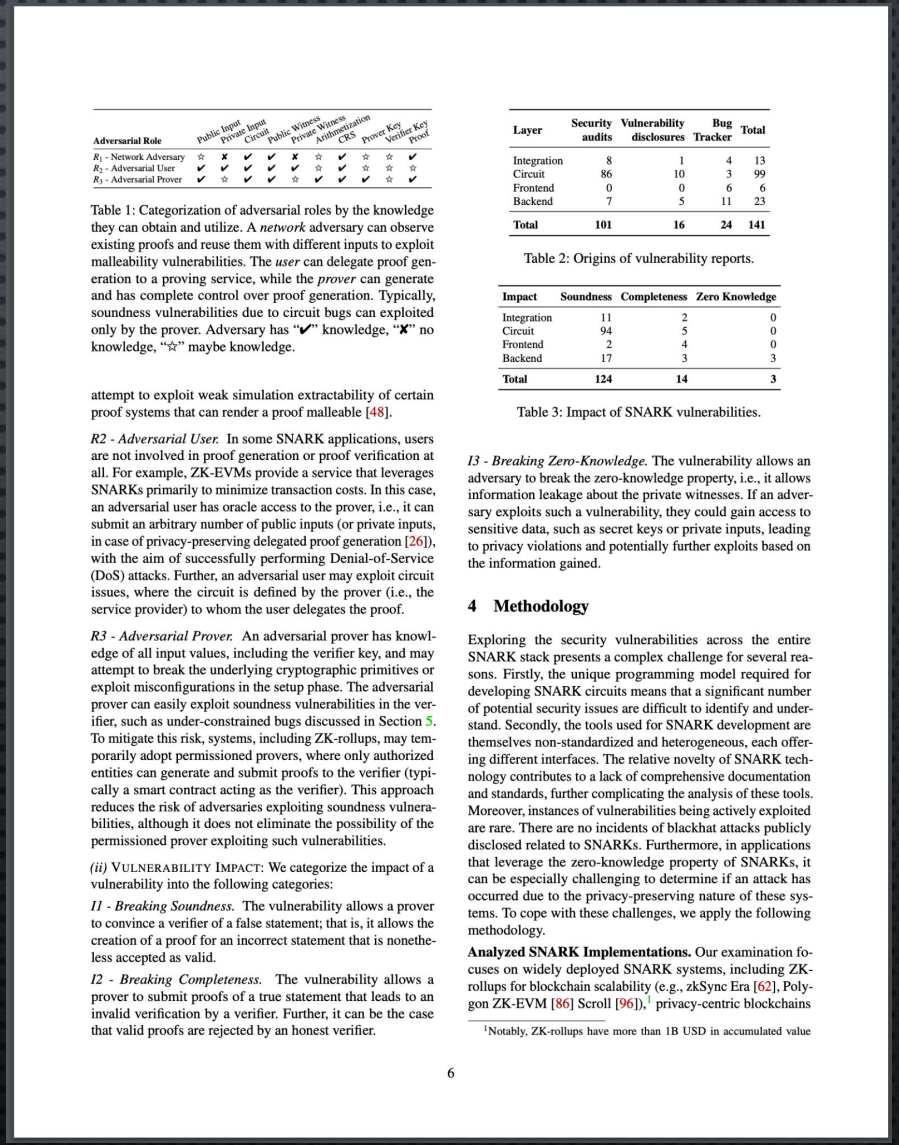
From Polygon

```
let y := calldataload(0x2a0)
    mstore(0x2c0, y)
    success := and(validate_ec_point(x, y), success)
}

{
    let x := calldataload(0x2c0)
    mstore(0x2e0, x)
    let y := calldataload(0x2e0)
    mstore(0x300, y)
    success := and(validate_ec_point(x, y), success)
}

{
    let x := calldataload(0x300)
    mstore(0x320, x)
    let y := calldataload(0x320)
    mstore(0x340, y)
    success := and(validate_ec_point(x, y), success)
}

{
    let x := calldataload(0x340)
    mstore(0x360, x)
    let y := calldataload(0x360)
    mstore(0x380, y)
    success := and(validate_ec_point(x, y), success)
}
mstore(0x3a0, keccak256(0x0, 928))
{
    let hash := mload(0x3a0)
    mstore(0x3c0, mod(hash, f_q))
    mstore(0x3e0, hash)
}
```

From Taiko

BUGS IN ZK

- Recent work has demonstrated that mathematical properties are sometimes at odds with the actual implementation found in the code.
- It highlights more than 140 bugs that largely come from security audit reports. If we are serious about the long-term future of ZK-based techniques, we should apply the same level of scrutiny to the ZK implementation as one applies to hardware.
- While in blockchain, TEEs like SGX are often criticized due to their vulnerabilities, failing to learn from the experience of many SGX bugs may will place us in the same position where theoretical claims are at odds with what their implementations actually provide.



Adversarial Role	Public Input	Private Input	Circuit	Public Witness	Private Witness	Arithmetization	CRS	Prover Key	Verifier Key	Proof
R_1 - Network Adversary	☆	✘	✔	✔	✘	☆	✔	☆	☆	✔
R_2 - Adversarial User	✔	✔	✔	✔	✔	☆	✔	☆	☆	☆
R_3 - Adversarial Prover	✔	☆	✔	✔	☆	✔	✔	✔	☆	✔

Table 1: Categorization of adversarial roles by the knowledge they can obtain and utilize. A *network* adversary can observe existing proofs and reuse them with different inputs to exploit malleability vulnerabilities. The *user* can delegate proof generation to a proving service, while the *prover* can generate and has complete control over proof generation. Typically, soundness vulnerabilities due to circuit bugs can exploited only by the prover. Adversary has "✔" knowledge, "✘" no knowledge, "☆" maybe knowledge.

attempt to exploit weak simulation extractability of certain proof systems that can render a proof malleable [48].

R2 - Adversarial User. In some SNARK applications, users are not involved in proof generation or proof verification at all. For example, ZK-EVMs provide a service that leverages SNARKs primarily to minimize transaction costs. In this case, an adversarial user has oracle access to the prover, i.e., it can submit an arbitrary number of public inputs (or private inputs, in case of privacy-preserving delegated proof generation [26]), with the aim of successfully performing Denial-of-Service (DoS) attacks. Further, an adversarial user may exploit circuit issues, where the circuit is defined by the prover (i.e., the service provider) to whom the user delegates the proof.

R3 - Adversarial Prover. An adversarial prover has knowledge of all input values, including the verifier key, and may attempt to break the underlying cryptographic primitives or exploit misconfigurations in the setup phase. The adversarial prover can easily exploit soundness vulnerabilities in the verifier, such as under-constrained bugs discussed in Section 5. To mitigate this risk, systems, including ZK-rollups, may temporarily adopt permissioned provers, where only authorized entities can generate and submit proofs to the verifier (typically a smart contract acting as the verifier). This approach reduces the risk of adversaries exploiting soundness vulnerabilities, although it does not eliminate the possibility of the permissioned prover exploiting such vulnerabilities.

(ii) VULNERABILITY IMPACT: We categorize the impact of a vulnerability into the following categories:

I1 - Breaking Soundness. The vulnerability allows a prover to convince a verifier of a false statement; that is, it allows the creation of a proof for an incorrect statement that is nonetheless accepted as valid.

I2 - Breaking Completeness. The vulnerability allows a prover to submit proofs of a true statement that leads to an invalid verification by a verifier. Further, it can be the case that valid proofs are rejected by an honest verifier.

Layer	Security audits	Vulnerability disclosures	Bug Tracker	Total
Integration	8	1	4	13
Circuit	86	10	3	99
Frontend	0	0	6	6
Backend	7	5	11	23
Total	**101**	**16**	**24**	**141**

Table 2: Origins of vulnerability reports.

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

Table 3: Impact of SNARK vulnerabilities.

I3 - Breaking Zero-Knowledge. The vulnerability allows an adversary to break the zero-knowledge property, i.e., it allows information leakage about the private witnesses. If an adversary exploits such a vulnerability, they could gain access to sensitive data, such as secret keys or private inputs, leading to privacy violations and potentially further exploits based on the information gained.

4 Methodology

Exploring the security vulnerabilities across the entire SNARK stack presents a complex challenge for several reasons. Firstly, the unique programming model required for developing SNARK circuits means that a significant number of potential security issues are difficult to identify and understand. Secondly, the tools used for SNARK development are themselves non-standardized and heterogeneous, each offering different interfaces. The relative novelty of SNARK technology contributes to a lack of comprehensive documentation and standards, further complicating the analysis of these tools. Moreover, instances of vulnerabilities being actively exploited are rare. There are no incidents of blackhat attacks publicly disclosed related to SNARKs. Furthermore, in applications that leverage the zero-knowledge property of SNARKs, it can be especially challenging to determine if an attack has occurred due to the privacy-preserving nature of these systems. To cope with these challenges, we apply the following methodology.

Analyzed SNARK Implementations. Our examination focuses on widely deployed SNARK systems, including ZK-rollups for blockchain scalability (e.g., zkSync Era [62], Polygon ZK-EVM [86] Scroll [96]),[1] privacy-centric blockchains

[1]Notably, ZK-rollups have more than 1B USD in accumulated value

6

BACKGROUND ON SGX BUGS

Security Vulnerabilities of SGX and Countermeasures: A Survey

SHUFAN FEI, The State Key Lab of ISN, School of Cyber Engineering, Xidian University, China
ZHENG YAN, The State Key Lab of ISN, School of Cyber Engineering, Xidian University, China and Department of Communications and Networking, Aalto University, Finland
WENXIU DING and HAOMENG XIE, The State Key Lab of ISN, School of Cyber Engineering, Xidian University, China

Trusted Execution Environments (TEEs) have been widely used in many security-critical applications. The popularity of TEEs derives from its high security and trustworthiness supported by secure hardware. Intel Software Guard Extensions (SGX) is one of the most representative TEEs that creates an isolated environment on an untrusted operating system, thus providing run-time protection for the execution of security-critical code and data. However, Intel SGX is far from the acme of perfection. It has become a target of various attacks due to its security vulnerabilities. Researchers and practitioners have paid attention to the security vulnerabilities of SGX and investigated optimization solutions in real applications. Unfortunately, existing literature lacks a thorough review of security vulnerabilities of SGX and their countermeasures. In this article, we fill this gap. Specifically, we propose two sets of criteria for estimating security risks of existing attacks and evaluating defense effects brought by attack countermeasures. Furthermore, we propose a taxonomy of SGX security vulnerabilities and shed light on corresponding attack vectors. After that, we review published attacks and existing countermeasures, as well as evaluate them by employing our proposed criteria. At last, on the strength of our survey, we propose some open challenges and future directions in the research of SGX security.

CCS Concepts: • **Security and privacy → Trusted computing**; *Mobile platform security*;

Additional Key Words and Phrases: Trusted execution environment, side-channel attacks, security, trustworthiness

ACM Reference format:
Shufan Fei, Zheng Yan, Wenxiu Ding, and Haomeng Xie. 2021. Security Vulnerabilities of SGX and Countermeasures: A Survey. *ACM Comput. Surv.* 54, 6, Article 126 (July 2021), 36 pages.
https://doi.org/10.1145/3456631

This work was supported in part by the National Natural Science Foundation of China under Grant 62072351 and Grant 61802293; in part by the National Postdoctoral Program for Innovative Talents under Grant BX20180238; in part by the Project funded by China Postdoctoral Science Foundation under Grant 2018M633461; in part by the Academy of Finland under Grant 308087 and Grant 335262; in part by the Shaanxi Innovation Team Project under Grant 2018TD-007; and in part by the 111 Project under Grant B16037.
Authors' addresses: S. F. Fei, W. X. Ding, and H. M. Xie, The State Key Lab of ISN, School of Cyber Engineering, Xidian University, 266 Xinglong Section of Xifeng Road, Xi'an, Shaanxi 710126, China; emails: shufanfei@gmail.com, wxding@xidian.edu.cn, 752099702@qq.com; Z. Yan (corresponding author), The State Key Lab of ISN, School of Cyber Engineering, Xidian University, Xi'an, Shaanxi, China and Department of Communications and Networking, School of Electrical Engineering, Aalto University, Konemiehentie 2, P.O.Box 15400, Espoo 02150, Finland; emails: zyan@xidian.edu.cn, zheng.yan@aalto.fi.

0360-0300/2021/07-ART126 $15.00
https://doi.org/10.1145/3456631

Table 3. Comparison of Published Attacks against SGX Security

Vulnerabilities	Type	Ref	Attacks: Attack surfaces PT	SE	CC	DRAM	BTB	PHT	RSB	EI	Attacks: Necessary conditions AVC	COI	CAI	MS	Attacks: Direct outcomes PAP	CAP	INT	MC	EIIP	Attacks: Compromised objectives CO	AS	US	Countermeasures: References
Address translation	PF	[99]	●	○	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	●	○	○	[1, 4, 10, 84, 90, 91]
		[91]	●	○	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	●	○	○	[1, 4, 10, 84, 90, 91]
	FL	[95]	●	○	○	○	○	○	○	○	●	○	○	○	●	○	○	○	○	●	○	○	[1, 4]
		[53]	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	●	○	○	[1, 4]
		[37]	●	○	●	●	○	○	○	○	●	●	○	○	●	●	○	○	○	●	○	○	[1, 4]
	SE	[39]	○	●	○	○	○	○	○	○	●	○	○	○	●	○	●	○	○	●	○	○	[1, 4]
CPU cache	CC	[31]	○	○	●	○	○	○	○	○	●	●	●	●	○	●	○	○	○	●	○	○	[1, 4, 33]
		[69]	○	○	●	○	○	○	○	○	●	●	●	○	○	●	○	○	○	●	○	○	[1, 4, 33]
		[82]	○	○	●	●	○	○	○	○	●	○	○	○	○	●	○	○	○	●	○	○	[1, 4, 33]
		[5]	○	○	●	○	○	○	○	○	●	●	○	○	○	●	○	○	○	●	○	○	[1, 4, 33]
		[17]	○	○	●	○	○	○	○	○	●	●	●	○	○	●	○	○	○	○	●	○	[1, 4, 33]
Branch prediction	BTB	[59]	○	○	○	○	●	○	○	○	●	○	○	○	○	○	●	○	○	●	○	○	[1]
		[9]	○	○	●	○	●	○	○	○	○	○	○	○	○	●	○	●	○	●	●	○	[1]
	PHT	[21]	○	○	○	○	○	●	○	○	○	●	○	○	○	○	●	○	○	●	○	○	[1]
		[44]	○	○	○	○	○	●	○	○	○	○	○	○	○	○	●	○	○	●	○	○	[1]
	RSB	[56]	○	○	●	○	○	○	●	○	○	○	○	○	○	●	○	●	○	●	○	○	[1]
	OTE	[94]	○	○	●	○	○	○	○	○	○	○	○	○	○	●	○	●	○	●	○	○	[1]
Enclave software	ROP	[58]	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●	○	[84]
	EI	[97]	○	○	○	○	○	○	○	●	●	●	○	○	○	○	○	○	●	●	○	○	\
Hardware	DDoS	[50]	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	●	\

● denotes a corresponding criterion is satisfied in a specific attack; ○ denotes a corresponding criterion is not satisfied in a specific attack; \ denotes there is no existing countermeasure could thwart a specific attack; PT: Page Table; SE: Segmentation; CC: CPU Cache; BTB: Branch Target Buffer; PHT: Page History Table; RSB: Return Stack Buffer; EI: Enclave Interface; AVC: Analysis of Victim Code; COI: COre Isolation; CAI: CAche Isolation; MS: Memory Sharing; PAP: Page Access Pattern leakage; CAP: Cache Access Pattern leakage; INT: Instruction Trace leakage; MC: Memory Content leakage; EIIP: Enclave Interface Innovation Pattern leakage; CO: Confidentiality; AS: Attestation Security; US: Usability.

ZK AUDITS

LINEA 2023

Critical Severity

Incorrect Randomness Computation Allows Proof Forgery

The `PlonkVerifier` contract is called by the rollup to verify the validity proofs associated with blocks of transactions. The `Verify` function can be called with the proof and the public inputs, and outputs a boolean indicating that the proof is valid for the received public inputs. To do so, the verifier notably samples a random value `u` and does a batch evaluation to verify that all the openings match their respective commitments.

However, `u` is not sampled randomly (or in practice as the hash of the full transcript), resulting in the possibility of forged proofs passing the verification. More specifically, `u` does not depend on $[W_\zeta]_1$ and $[W_{\zeta\omega}]_1$, meaning that it is possible for a malicious prover to change $[W_\zeta]_1$ and $[W_{\zeta\omega}]_1$ after seeing the value of `u`. Here is an example of an attack based on this observation:

1. The malicious prover `P` extracts $A = [W_\zeta]_1 + u*[W_{\zeta\omega}]_1$ and $B = z*[W_\zeta]_1 + u\zeta\omega*[W_{\zeta\omega}]_1 + [F]_1 - [E]_1$ obtained when any valid proof is submitted. `A` and `B` are by construction points on the elliptic curve for which $e(-A, [x]_2) * e(B, [1]_2) == 1$.
2. `P` sets the public input (or any proof commitment) to an arbitrary value. Any value beside $t(\zeta)$, $[W_\zeta]_1$ and $[W_{\zeta\omega}]_1$ can be changed.
3. `P` computes $T = [\ r(\zeta) + PI(\zeta) - ((a(\zeta) + \beta s_{\sigma 1}(\zeta) + \gamma)(b(\zeta) + \beta s_{\sigma 2}(\zeta) + \gamma)(c(\zeta) + \gamma)z_\omega(\zeta))\alpha - L1(\zeta)\alpha^2\ ]\ /\ Z_H(\zeta)$ following step 8 of the PLONK verifier algorithm. The prover sets $t(\zeta) = T$ in the forged proof. This step is required to pass this check in the code.
4. `P` computes `u`, ζ and ω as done by the code.
5. `P` solves the equations $X + u*Y = A$ and $\zeta*X + \zeta\omega u*Y = C + B$ obtained by denoting $X = [W_\zeta]_1$ and $Y = [W_{\zeta\omega}]_1$ taken from step 12 of the verifier algorithm. This system has for solutions $[W_\zeta]_1 = X = (-u)*Y + A$ and $[W_{\zeta\omega}]_1 = Y = 1/(\zeta u(\omega - 1)) * (C + B - \zeta A)$.
6. `P` submits the proof to the verifier, replacing $t(\zeta)$ by `T`, and $[W_\zeta]_1$, $[W_{\zeta\omega}]_1$ by the values computed in step 5.
7. The verifier computes $e(-A, [x]_2) * e(B, [1]_2) == 1$ and accepts the proof.

Project	When (clickable)
Linea	November 2023
ZKSync	February 2023
Scroll	September 2023
Hermez zkEVM	March 2024

Figure 4: Recent audits of ZK EVMs.

https://blog.openzeppelin.com/linea-verifier-audit-1

ZK AUDITS

AZTEC 2021

Vulnerabilities patched in Aztec 2.0

Aztec Labs · Follow
Published in The Aztec Labs Blog · 7 min read · Sep 16, 2021

67

In March 2021, we launched Aztec 2.0, which enables users to shield and send funds privately through Aztec private rollups. Aztec 2.0 utilizes our state of the art zkSNARK proving system, PLONK, developed in-house for the express purpose of scaling Ethereum with strong user privacy guarantees.

Aztec 2.0 is built with bleeding-edge cryptography and it is critical to promptly address any bugs. Aztec is in a continual state of audit internally and externally, incentivized by a bug bounty with Immunefi. Our core team discovered two security vulnerabilities as part of our internal efforts, with special thanks to our chief scientist Ariel Gabizon. Community members Sean Bowe and Daira Hopwood also highlighted vulnerabilities.

Bug: Recursive proof verification

When aggregating private transactions in our rollup circuit, we use the following circuit structure:

Join-split Circuit: Executes a private transaction; generated by the user locally on their device.

Rollup Circuit: Verifies the correctness of **28** join-split circuit proofs and performs database updates into the rollup's Merkle trees.

Root Rollup Circuit: Verifies the correctness of **4** rollup circuit proofs.

When verifying a Plonk proof inside one of our circuits, partial verification and proof aggregation occurs.

Each proof is verified up to the point that a bilinear pairing check is required. The Plonk verification algorithm's bilinear pairing check is structured such that both G2 group elements are fixed and do not vary between different proofs.

Instead of performing this pairing check inside our root rollup circuit, these two group elements are defined to be *public inputs* of the root rollup circuit. i.e. they are broadcasted on-chain as part of the root rollup proof.

The *verifier smart contract* will then extract the two group elements and aggregate them into the pairing check computed by the smart contract.

The bug was that, when performing the proof aggregation step in the root rollup circuit, we were aggregating only the rollup proofs, *but not the join-split proofs*.

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

HOW DO WE DO BETTER?

ZK PROPERTIES

Completeness. Completeness ensures the correct operation of the protocol if both prover and verifier follow the protocol honestly (in other words, exactly as prescribed by the description of the protocol).

Soundness. Soundness states that if the provable statement is false then a cheating prover cannot convince an honest verifier that it is true, except with some small probability.

Proof-of-knowledge. Proof-of-knowledge guarantees that any prover that successfully convinces the honest verifier actually knows a witness (and not only abstractly that it exists).

Lean

- Deep embedding of YUL syntax and semantics
- Nethermind developed a formal specification of YUL in Lean – excellent opportunities to verify functional properties of YUL contracts
- Lean can be used to reason about the functional correctness of the verifier.
- However, it is hard to prove cryptographic properties without specialized logics designed specifically to address probabilistic programs – no good approach to that

KEVM (K~Semantics of the Ethereum Virtual Machine)

- Guarantees of soundness of the EVM model and great degree of automation when it comes to derivation of functional properties of YUL code
- Cannot be directly used to derive cryptographic properties in the probabilistic setting

EasyCrypt

- EasyCrypt is a great fit for proving cryptographic properties, such as soundness and proof-of-knowledge.
- Has specialized logics to prove program equivalence as well as built-in support for imperative modules, which could be used for shallow embedding of simple functions from YUL (similar approach was already demonstrated to be successful with Jasmin/EasyCrypt toolchain

Given that we need to go beyond just functional correctness, where Lean could have been an easy and widely-used option, EasyCrypt is a good choice for targeting soundness and proof-of-knowledge properties of ZK verifiers.

OUR APPROACH

1. We extract Verifier.sol to EasyCrypt theorem prover preserving the semantics and the low-level structure of the code.
2. We use EasyCrypt abstractions to define a mathematical specification of the verification function.
3. We use program logics (e.g., Hoare logic and probabilistic relational Hoare logic) to prove equivalence between the low and high-level implementations of the verification function.
4. By relying on mathematical abstractions, we derive security for high-level implementation of the verifier.
5. We use the derived equivalence between high-level and low-level verifiers to establish security for low-level (YUL) implementation of the verifier

REASONING ABOUT THE VERIFIER (1)

$$L_i(z) = \frac{\omega^i(z^N - 1)}{N(z - \omega^i)}.$$

```
op lagrangePoly (i : int) (z : F) =
   let omegaPower = OMEGA ^ i in
          (z ^ N - 1) * omegaPower * inv ((z - omegaPower) * N).
```

REASONING ABOUT THE VERIFIER (2)

```
function evaluateLagrangePolyOutOfDomain(i : u256, z : u256) -> res {
    let omegaPower := 1
    if i {
        omegaPower := modexp(OMEGA, i)
    }

    res := addmod(modexp(z, N), sub(R_MOD, 1), R_MOD)

    if iszero(res) {
        revertWithMessage(28, "invalid vanishing polynomial")
    }

    res := mulmod(res, omegaPower, R_MOD)
    let denominator := addmod(z, sub(R_MOD, omegaPower), R_MOD)
    denominator := mulmod(denominator, N, R_MOD)
    denominator := modexp(denominator, sub(R_MOD, 2))
    res := mulmod(res, denominator, R_MOD)
}
```

```
module Verifier {
... // remaining code of the Verifier

 proc evaluateLagrangePolyOutOfDomain(i : u256, z : u256) : u256 = {
    var result, omegaPower, denominator;
    omegaPower <- 1;
    if (0 < i) {
        omegaPower <- modexp(OMEGA, i);
    }

    result <- addmod(modexp(z, N), sub(R_MOD, 1), R_MOD);

    if (iszero(result)) {
        Syscall.revert();
    }

    result      <- mulmod(result, omegaPower, R_MOD);
    denominator <- addmod(z, sub(R_MOD, omegaPower), R_MOD);
    denominator <- mulmod(denominator, N, R_MOD);
    denominator <- modexp(denominator, sub(R_MOD, 2));
    result      <- mulmod(result, denominator, R_MOD);
    return result;
 }
}
```

REASONING ABOUT THE VERIFIER (3)

```
lemma lagrange_eq (i z : u256) :
 hoare [ Verifier.evaluateLagrangePolyOutOfDomain
  : arg = (i,z) /\ {z}^n - 1 != 0
  ==>
       {res} = lagrangePoly [i] {z}].
```

The statement above is a Hoare triple that states that

- for any binary words i and z, the result of computation of evaluateLagrangePolyOutOfDomain is "equivalent" to the result of computation of lagrangePoly.

- Functions [_] and {_} convert a 256-bit word into an integer and a field element, respectively.
- Under the assumption that translations (YUL to EasyCrypt and math to EasyCrypt) are both sound, we have illustrated that the low-level YUL function implements high-level math function Li(z) correctly.

YUL-to-EasyCrypt TRANSLATION CHALLENGES

- **YUL memory state**. In EasyCrypt we give an explicit axiomatisation of a YUL memory state. The memory itself is represented by a value m of abstract datatype memory. Operations which depend on the memory state take memory value as an explicit argument. Then the axioms specify the relationship among properties of such operations, for example, mload((mstore(m,k,v), k) = v.

- **YUL datatypes**. We specify the binary model of YUL datatypes with the respective ops. For example, our model specifies the function z <- addmod(x,y,m) on binary words, with the corresponding correctness property stating that z = x + y %% m; where a converts the binary word a to its corresponding integer value.

- **YUL side-effects**. We do not model any computational side-effect beyond reverting. The YUL function revertWithMessage is translated to EasyCrypt as Syscall.revert(); which sets the boolean flag Syscall.reverted to true. This allows us to reason about under which conditions the contract reverts.

- **Gas**. We do not presently model gas explicitly; instead we prove the properties of the verifier under assumption that there is enough gas for the contract execution.

WHAT TO VERIFY

Circuits – a number of projects and active audits

Prover – a number of audits and languages for e2e correctness

Verifier – this work

The Ouroboros of ZK: Why Verifying the Verifier Unlocks Longer-Term ZK Innovation

Denis Firsov[1] and Benjamin Livshits[2]

[1]Matter Labs
[2]Imperial College London

Abstract

Verifying the verifier in the context of zero-knowledge proof is an essential part of ensuring the long-term integrity of the zero-knowledge ecosystem. This is vital for both zero-knowledge rollups and also other industrial applications of ZK. In addition to further minimizing the required trust and reducing the trusted computing base (TCB), having a verified verifier opens the door to decentralized proof generation by potentially untrusted parties. We outline a research program and justify the need for more work at the intersection of ZK and formal verification research.

1 Introduction

Zero-knowledge[1] (ZK) proofs give us a powerful way to move away from the relatively wasteful multi-execution model of most blockchains, where verifiers execute the same code and agree on the outcome. Zero-knowledge rollups in recent years have shown that highly optimized ZK provers provide an effective alternative to multi-execution. However, implementation bugs in both provers and verifiers can easily compromise the security of the overall system. Specifications (we can think of circuits as a form of specification), provers, and verifiers can all be buggy.

Buggy circuits. In the contemporary ZK proof systems the relations are typically specified by arithmetic circuits. The bugs in the circuits could not be directly attributed to neither verifiers nor provers, but can be thought as specification bugs [1, 2].

Buggy or malicious provers. Bugs in the provers led to the creation of bug taxonomies[2]. However, these are not the only kind of bugs possible in provers. Indeed, a malicious prover — a real possibility given that prover code can be changed by the attacker — can forge a ZK proof that will pass the verification. Malice is not the only source of trouble; for instance, insecure implementations of the Fiat-Shamir transformation can also allow attackers to successfully forge

[1]It is worth noting that throughout the text *ZK* and zero-knowledge terms stem for conventional reasons. In the context of blockchain rollups the cryptographic zero-knowledge property is not essential and is sometimes undesirable for regulation reasons.
[2]ZK Bug Tracker https://github.com/0xPARC/zk-bug-tracker

The Ouroboros of ZK: Why Verifying the Verifier Unlocks Longer-Term ZK Innovation

Denis Firsov[1] and Benjamin Livshits[2]

[1]Matter Labs
[2]Imperial College London

Abstract

Verifying the verifier in the context of zero-knowledge proof is an essential part of ensuring the long-term integrity of the zero-knowledge ecosystem. This is vital for both zero-knowledge rollups and also other industrial applications of ZK. In addition to further minimizing the required trust and reducing the trusted computing base (TCB), having a verified verifier opens the door to decentralized proof generation by potentially untrusted parties. We outline a research program and justify the need for more work at the intersection of ZK and formal verification research.

1 Introduction

Zero-knowledge[1] (ZK) proofs give us a powerful way to move away from the relatively wasteful multi-execution model of most blockchains, where verifiers execute the same code and agree on the outcome. Zero-knowledge rollups in recent years have shown that highly optimized ZK provers provide an effective alternative to multi-execution. However, implementation bugs in both provers and verifiers can easily compromise the security of the overall system. Specifications (we can think of circuits as a form of specification), provers, and verifiers can all be buggy.

Buggy circuits. In the contemporary ZK proof systems the relations are typically specified by arithmetic circuits. The bugs in the circuits could not be directly attributed to neither verifiers nor provers, but can be thought as specification bugs [1, 2].

Buggy or malicious provers. Bugs in the provers led to the creation of bug taxonomies[2]. However, these are not the only kind of bugs possible in provers. Indeed, a malicious prover — a real possibility given that prover code can be changed by the attacker — can forge a ZK proof that will pass the verification. Malice is not the only source of trouble; for instance, insecure implementations of the Fiat-Shamir transformation can also allow attackers to successfully forge

[1]It is worth noting that throughout the text *ZK* and zero-knowledge terms stem for conventional reasons. In the context of blockchain rollups the cryptographic zero-knowledge property is not essential and is sometimes undesirable for regulation reasons.
[2]ZK Bug Tracker https://github.com/0xPARC/zk-bug-tracker

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.

Mechanism Design for ZK-Rollup Prover Markets

Wenhao Wang* Lulu Zhou* Aviv Yaish†‡ Fan Zhang* Ben Fisch*
Benjamin Livshits‡§

April 10th, 2024

Abstract

In ZK-Rollups, provers spend significant computational resources to generate validity proofs. Their costs should be compensated properly, so a sustainable prover market can form over time. Existing transaction fee mechanisms (TFMs) such as EIP-1559, however, do not work in this setting, as EIP-1559 only generates negligible revenue because of burning, while provers often create or purchase specialized hardware in hopes of creating long-term revenue from proving, somewhat reminiscent of proof-of-work miners in the case of chains like Bitcoin. In this paper, we explore the design of transaction fee mechanisms for prover markets. The desiderata for such mechanisms include efficiency (social welfare is maximized), incentive compatibility (it is rational to bid honestly), collusion resistance (no profitable collusion among provers exists), and off-chain agreement proofness (no profitable collusion between users and provers exists). To demonstrate the difficulties of our new setting, we put forward several simple strawman mechanisms, and show they suffer from notable deficiencies.

1 Introduction

ZK-Rollup systems rely on provers to generate zero-knowledge proofs (ZKPs) to finalize ZK-Rollup blocks. Despite recent efficiency improvement [24, 8, 6], generating ZKPs for ZK-Rollups remains a demanding job that requires significant computational resources [12]. Currently, ZK-Rollup systems typically run their own centralized provers [56]. However, previous works on similar centralized approaches find they are susceptible to censorship [51] and may increase costs for users, while allowing free entry for service providers can increase competitiveness and lower costs [19]. An appealing alternative is to build an open and permissionless marketplace where provers compete to generate ZKPs at low costs. In a prover market, user fees should at the very least cover the cost of provers and, ideally, also provide some sustainable profit margins. This, together with a series of design goals, calls for a properly designed *transaction fee mechanism* for prover markets.

Prover markets differ from classical fee markets (e.g., that in Bitcoin and Ethereum) in three ways. Firstly, a prover market is a two-sided market where users demand proof capacity, and provers

*Yale University
†Hebrew University of Jerusalem
‡Matter Labs
§Imperial College London

References

[1] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, "SoK: What don't we know? understanding security vulnerabilities in SNARKs," 2024.

[2] H. Wen, J. Stephens, Y. Chen, K. Ferles, S. Pailoor, K. Charbonnet, I. Dillig, and Y. Feng, "Practical security analysis of Zero-Knowledge Proof Circuits," 2024.

[3] O. Ciobotaru, M. Peter, and V. Velichkov, "The last challenge attack: Exploiting a vulnerable implementation of the fiat-shamir transform in a KZG-based SNARK," Cryptology ePrint Archive, Paper 2024/398, 2024. [Online]. Available: https://eprint.iacr.org/2024/398

[4] Z. Ye, U. Misra, J. Cheng, W. Zhou, and D. Song, "Specular: Towards secure, trust-minimized optimistic blockchain execution," 2024.

[5] S. Fei, Z. Yan, W. Ding, and H. Xie, "Security vulnerabilities of SGX and countermeasures: A survey," *ACM Comput. Surv.*, vol. 54, no. 6, jul 2021.

[6] J. Liu, I. Kretz, H. Liu, B. Tan, J. Wang, Y. Sun, L. Pearson, A. Miltner, I. Dillig, and Y. Feng, "Certifying Zero-Knowledge Circuits with Refinement Types," *IACR Cryptol. ePrint Arch.*, 2023.

[7] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez-Núñez, J. V. Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, "Automated Detection of Under-Constrained Circuits in Zero-Knowledge Proofs," *Proc. ACM Program. Lang.*, vol. 7, no. PLDI, 2023.

[8] W. Wang, L. Zhou, A. Yaish, F. Zhang, B. Fisch, and B. Livshits, "Mechanism design for ZK-rollup prover markets," 2024.

[9] J. B. Almeida, D. Firsov, T. Oliveira, and D. Unruh, "Schnorr protocol in Jasmin," Cryptology ePrint Archive, Paper 2023/752, 2023. [Online]. Available: https://eprint.iacr.org/2023/752

[10] D. Firsov and D. Unruh, "Zero-knowledge in EasyCrypt," in *IEEE Computer Security Foundations Symposium (CSF)*, 2023.

[11] J. B. Almeida, M. Barbosa, M. L. Correia, K. Eldefrawy, S. Graham-Lengrand, H. Pacheco, and V. Pereira, "Machine-checked zkp for np-relations: Formally verified security proofs and implementations of mpc-in-the-head," Cryptology ePrint Archive, Paper 2021/1149, 2021. [Online]. Available: https://eprint.iacr.org/2021/1149

[12] G. Barthe, D. Hedin, S. Z. Béguelin, B. Grégoire, and S. Heraud, "A machine-checked formalization of sigma-protocols," in *IEEE Computer Security Foundations Symposium*. IEEE, 2010.

[13] D. Butler, A. Lochbihler, D. Aspinall, and A. Gascón, "Formalising σ-protocols and commitment schemes using crypthol," *Journal of Automated Reasoning*, vol. 65, no. 4, 2021.

[14] B. Bailey and A. Miller, "Formalizing Soundness Proofs of SNARKs."

[15] J. Bacelar Almeida, M. Barbosa, E. Bangerter, G. Barthe, S. Krenn, and S. Zanella Béguelin, "Full proof cryptography: verifiable compilation of efficient zero-knowledge protocols," in *Proceedings of the Conference on Computer and Communications Security*, 2012.

[16] A. Coglio, E. McCarthy, and E. W. Smith, "Formal Verification of Zero-Knowledge Circuits," *Electronic Proceedings in Theoretical Computer Science*, vol. 393, nov 2023.

[17] C. Chin, H. Wu, R. Chu, A. Coglio, E. McCarthy, and E. Smith, "Leo: A Programming Language for Formally Verified, Zero-Knowledge Applications," *IACR Cryptol. ePrint Arch.*, 2021.

[18] L. Goldberg, S. Papini, and M. Riabzev, "Cairo – a Turing-complete STARK-friendly CPU architecture," 2021.

[19] L. d. Moura and S. Ullrich, "The Lean 4 theorem prover and programming language," in *Automated Deduction (CADE)*, A. Platzer and G. Sutcliffe, Eds. Cham: Springer International Publishing, 2021.

[20] Nethermind, "YUL IR specification." [Online]. Available: https://github.com/NethermindEth/Yul-Specification

[21] E. Hildenbrandt, M. Saxena, N. Rodrigues, X. Zhu, P. Daian, D. Guth, B. Moore, D. Park, Y. Zhang, A. Stefanescu, and G. Rosu, "KEVM: A complete formal semantics of the Ethereum Virtual Machine," in *IEEE Computer Security Foundations Symposium (CSF)*, 2018.

[22] J. B. Almeida, M. Barbosa, G. Barthe, A. Blot, B. Grégoire, V. Laporte, T. Oliveira, H. Pacheco, B. Schmidt, and P.-Y. Strub, "Jasmin: High-assurance and high-speed cryptography," in *Proceedings of the Conference on Computer and Communications Security*, 2017.

Take-away Messages

- Verifiers are mission-critical yet small pieces of software
- Bugs in verifiers can be devastating and can bring down modern rollups with hundreds of millions in TVL, if exploited
- Verifying the verifier unlocks the possibility of truly decentralized proving
- Combining our knowledge of verification and ZK cryptography puts us in a very comfortable position to address these challenges before they play our in practice

Call to arms

1. people who do ZK in the industry should embrace formal verification

2. verification community should find exciting and relevant problems in the ZK space