

ZKScan: A Static Analyzer for Smart Contract Based Zero-Knowledge Proof Verifiers on Blockchains

Anik Tahabilder
Department of Computer Science
Wayne State University
Detroit, MI, USA
tahabilderanik@wayne.edu

M M Imran
Department of Computer Science
Wayne State University
Detroit, MI, USA
mmimran@wayne.edu

Changxin Bai
Department of Computer Science
Kettering University
Flint, MI, USA
cbail@kettering.edu

Qiang Zhu
Department of Computer and Information Science
University of Michigan-Dearborn
Dearborn, MI, USA
qzhu@umich.edu

Shiyong Lu
Department of Computer Science
Wayne State University
Detroit, MI, USA
shiyong@wayne.edu

Abstract—Zero-knowledge (ZK) proof systems on Ethereum use on-chain verifier contracts to check cryptographic proofs for Layer 2 rollups, privacy protocols, and identity systems. A bug in the verifier may cause an invalid proof to be accepted and can lead to loss of funds. Existing ZK security tools mainly analyze prover circuits, while general smart contract analyzers focus on conventional contract vulnerabilities. Neither approach targets verifier-specific logic such as pairing operations, Fiat-Shamir transcripts, and inline Yul assembly. We present ZKScan, a static analyzer that checks verifier contracts before deployment. ZKScan organizes ten detection rules into five vulnerability classes for Groth16, PLONK, and FFLONK verifiers. For precompile calls, ZKScan uses intraprocedural Yul dataflow analysis to identify unchecked return values. We evaluate ZKScan on 74 contracts across three datasets: a synthetic dataset for detector validation, an audit-derived dataset to measure detection effectiveness, and a mainnet dataset combined with the audited contracts to evaluate real-world alert volume. On the audit-derived dataset, ZKScan achieves 94% recall and 77% precision. In contrast, Slither and Mythril detect none of the injected ZK-specific vulnerabilities.

Keywords—zero-knowledge proof, verifier contract, smart contract vulnerability, static analysis, Groth16, Ethereum

I. INTRODUCTION

Every major Ethereum zero-knowledge (ZK) rollup (e.g., zkSync Era, Polygon zkEVM, Scroll, Linea, Taiko) relies on an on-chain verifier contract to check cryptographic proofs before the corresponding state transition is finalized on the Ethereum base layer [1]. The verifier forms the final trust boundary for these proofs. If it accepts a forged proof, an attacker will be able to fabricate state transitions and drain funds. ZK rollups secure billions of dollars in total value locked (TVL) on Ethereum [2], making the security of their verifier contracts increasingly important.

As shown in Fig. 1, vulnerabilities in ZK systems can occur at four layers: circuit constraints, compiler/library code, on-chain verifier logic, and application logic. Existing audit tools

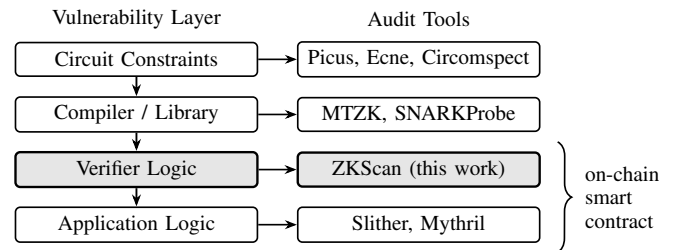


Fig. 1. ZK vulnerability layers and audit tool coverage.

address circuit, library, and application vulnerabilities, but lack dedicated coverage for on-chain verifier contracts.

This gap matters in practice because verifier vulnerabilities have already led to real-world exploits. In 2022, the Frozen Heart vulnerability [3], [4] could let attackers forge proofs in multiple PLONK implementations. More recently, the FOOMCASH and Veil Cash exploits [5] drained \$2.26M from protocols that shared the same broken verifier, where `gamma2 == delta2` broke the pairing soundness condition. The 0xPARC ZK Bug Tracker [6] also documents dozens of ZK vulnerabilities, including verifier-side flaws that existing tools do not detect.

We therefore present **ZKScan**, a static analyzer for checking on-chain ZK proof verifiers. The source code is available at <https://github.com/atahabilder1/ZKScan>. Our contributions are:

- 1) A **vulnerability taxonomy** for Solidity/Yul verifier contracts with five classes derived from published audits and exploit disclosures.
- 2) **ZKScan**, a static analyzer with ten detector rules (D1–D10) for Groth16, PLONK, and FFLONK. Detector D6 uses intraprocedural Yul dataflow analysis to identify unchecked precompile return values.
- 3) An evaluation suite of 74 contracts across three datasets:

an 18-contract **synthetic dataset**, a 20-contract **audit-derived dataset**, and a 36-contract in-the-wild **mainnet dataset**.

II. BACKGROUND

The correctness of a verifier contract depends on three factors, namely the cryptographic proof system it implements, the Ethereum Virtual Machine (EVM) precompiles it invokes, and the adversarial inputs it must handle.

A. ZK Proof Systems

On-chain verifiers primarily implement Groth16, PLONK-family, or FFLONK proof systems [7], each with distinct verification requirements and failure modes.

Groth16 [8] is a pairing-based proof system widely used for its minimal proof size (3 group elements: $A \in G_1, B \in G_2, C \in G_1$) and $O(1)$ verification. Verification requires evaluating the pairing equation:

$$e(A, B) = e(\alpha, \beta) \cdot e\left(\sum_{i=0}^{\ell} a_i L_i, \gamma\right) \cdot e(C, \delta) \quad (1)$$

where $\alpha, \beta, \gamma, \delta$ come from the verification key, L_i are input commitments, and a_i are public inputs. Omitting any of these terms may break soundness. Groth16 verifiers generated by tools such as snarkjs [9] have been widely deployed in systems such as Tornado Cash and Semaphore.

PLONK [10] offers a universal setup and derives verification challenges via Fiat-Shamir [11]. The verifier reconstructs successive challenges $(\beta, \gamma, \alpha, \zeta, v, u)$ by hashing the proof transcript. Omitting or reordering commitments breaks soundness. On Ethereum, PLONK-family verifiers encompass standard PLONK (e.g., Linea [12], Aztec) and PLONKish/Halo2 systems.

FFLONK [13] batches polynomial openings to reduce proof size and gas cost and has been deployed in Polygon zkEVM verifier contracts.

B. EVM Precompiles

On-chain verification relies on three EVM precompiles for BN254 arithmetic defined by Ethereum Improvement Proposals (EIPs) EIP-196 [14] and EIP-197 [15]: `ecAdd` (0x06), `ecMul` (0x07), and `ecPairing` (0x08), invoked via `staticcall`. Using these precompiles introduces three main security concerns. First, a failed precompile invocation returns a zero `staticcall` success flag and may leave stale output memory. Second, raw BN254 field multiplication can overflow 256-bit EVM words without suitable modular arithmetic. Third, confusing the base field prime p with the scalar field order r allows scalar aliasing.

C. Threat Model

We assume an adversary who controls proof elements (e.g., (A, B, C) in Groth16) and public inputs $\{a_i\}$ submitted as calldata. The verification key (VK) is assumed trustworthy unless its handling is vulnerable (D5). The adversary seeks a *soundness violation*, attempting to induce the verifier to accept a proof for an invalid statement.

TABLE I
VERIFIER TAXONOMY: 5 CLASSES, 10 DETECTOR RULES

Class	Rules	Sev.	Grounding
C1: Input sanitization	D3, D9	High	Linea [12]
C2: Field/curve arith.	D2, D7	Crit.	ABDK [24]
C3: VK integrity	D5, D10	Crit.	FOOMCASH [5]
C4: Fiat-Shamir binding	D4	Crit.	Frozen Heart [3]
C5: Pairing/precompiles	D1, D6, D8	Crit.	Aztec-ToB [25]

III. RELATED WORK

General smart contract analyzers such as Slither [16] and Mythril [17] detect issues such as reentrancy and arithmetic bugs. They do not model ZK-specific pairing or transcript semantics. Other tools focus on the prover side. Picus [18] checks circuit constraints, Circomspect [19] analyzes Circom source, SNARKProbe [20] fuzzes rank-1 constraint system (R1CS) libraries, and MTZK [21] tests ZK compilers. These tools do not provide dedicated coverage of the verifier-specific vulnerabilities considered here.

Chaliasos et al. [22] catalogue 141 vulnerabilities in succinct non-interactive arguments of knowledge (SNARKs) and place 16 on-chain verifier defects in an *Unsafe Verifier* class. Their study records no existing defense for this class. ZKScan focuses on these verifier-side problems, and part of our audit-derived dataset comes from their study. Circomspect [19] is a related example of syntactic pattern matching, although it operates on Circom circuits rather than on-chain verifiers. ZKScan additionally uses intraprocedural dataflow analysis for Yul assembly because verifier values are often passed through memory before they are checked, which makes simple pattern matching insufficient.

The Frozen Heart disclosure [3] showed that the same Fiat-Shamir error can appear across multiple codebases. Professional audits [12], [23] do identify these classes of issues, but they require manual review. ZKScan is intended as an automated pre-audit check for similar verifier-specific vulnerabilities.

IV. VULNERABILITY TAXONOMY

Based on historical exploits, audits [12], [23], and the ZK Bug Tracker [6], we group verifier vulnerabilities into five classes. Table I shows these classes and the ten corresponding detector rules. Severities follow audit standards (Consensys, OpenZeppelin) and reflect the potential impact of each class, while individual detector rules assign finding-specific severities.

C1: Input and Calldata Sanitization (D3, D9). Public inputs enter via $\sum a_i \cdot L_i$. Because G_1 has order r , noncanonical inputs ($a_i \geq r$) induce scalar aliasing, where a_i and $a_i + r$ yield the same curve point, enabling input malleability and, depending on application semantics, double-spending. Entry points that accept dynamic arrays without adequate length validation may similarly admit malformed calldata. Both Linea audits [12], [23] flagged this class.

C2: Finite Field and Curve Arithmetic (D2, D7). Multiplying two 254-bit BN254 field elements can require up to 508 bits, exceeding the 256-bit EVM word size. Furthermore, in custom

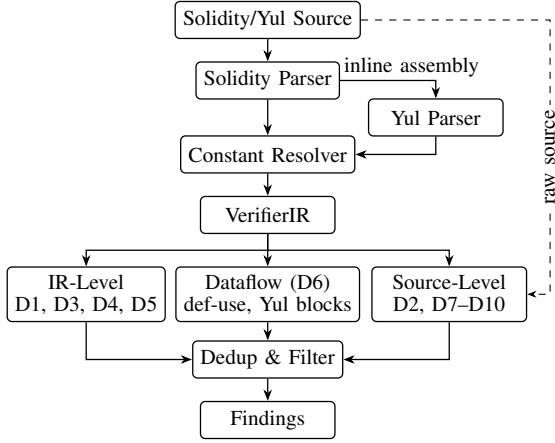


Fig. 2. ZKScan multi-stage analysis pipeline and detector routing.

elliptic-curve (EC) arithmetic routines that do not delegate to precompiles, omitting on-curve ($y^2 = x^3 + 3 \bmod p$) or G_2 subgroup checks (cofactor $h \neq 1$) can produce incorrect results or permit invalid points. Both defects were observed in the Linea [23] and ABDK [24] audits.

C3: Verification Key and Setup Integrity (D5, D10). The verification key determines *which circuit* is verified. An unauthenticated calldata-supplied key, a missing Groth16 component ($\alpha, \beta, \gamma, \delta$, input-coefficient (IC) points), or duplicate G_2 components ($\gamma_2 = \delta_2$) can break this binding; the last flaw was exploited in FOOMCASH/Veil Cash [5]. Zero-valued VK coordinates or scalar-field/base-field mismatches can have the same effect.

C4: Transcript and Fiat-Shamir Binding (D4). PLONK verifiers derive challenges $\beta, \gamma, \alpha, \zeta, v, u$ by hashing successive rounds of the proof transcript. Each challenge must bind to prior commitments. Omitting required transcript data such as public inputs, as in Frozen Heart [3], lets the prover manipulate challenges to satisfy false statements.

C5: Pairing Logic and Precompile Handling (D1, D6, D8). Implementations evaluate the pairing relation in (1) as a single product tested against unity. Dropping any pairing term breaks soundness, as in the Aztec Swap Validator [25]. When invoking precompiles (EIP-197 [15]), the verifier must check both call success and the pairing output word: an unchecked failed call can leave stale memory, while an unchecked output word lets a false pairing result pass. Linea audits [12] report both variants.

V. SYSTEM DESIGN

As shown in Fig. 2, ZKScan analyzes Solidity contracts containing inline Yul assembly in three stages: source parsing, intermediate representation (IR) construction, and vulnerability detection.

A. Architecture Overview

A **Solidity parser** extracts function signatures, constants, and `require` statements, while a **Yul parser** extracts `staticcall` targets, `let` bindings, and `mstore/mload`

operations from inline assembly blocks. Both parsers use regex extraction and operate without `solc`, avoiding compiler dependencies across different Solidity versions. A **constant resolver** maps named constants to values (e.g., `r_mod` to the BN254 scalar field order), letting downstream stages recognize precompile calls, field bounds, and VK components across `snarkjs`, `gnark` [26], and custom code.

Parsing produces a structured **VerifierIR** that records proof-system type, pairing terms, field operations, input range checks, Fiat-Shamir challenges, verification-key components, and precompile return-check status. The parser produces a nonempty VerifierIR for 50 of the 56 evaluated real-world contracts, resolving 342 precompile calls.

The ten detectors operate across three tiers (Fig. 2), spanning IR-level checks (D1, D3, D4, D5), source-level rules (D2, D7–D10), and Yul dataflow analysis (D6). A proof-system-aware registry ensures that detectors run only on applicable contracts (e.g., D4 is restricted to PLONK/FFLONK). Several detectors combine IR and source checks. The resulting findings are then deduplicated.

B. Detection Techniques

IR-level analysis (D1, D3, D4, D5). Four detectors reason over VerifierIR. D1 checks pairing input lengths against heuristic term-count thresholds (6×32 bytes per pair). D3 checks that public inputs are bounded ($a_i < r$) to prevent scalar aliasing. D5 inspects verification key sourcing and component completeness, flagging calldata-supplied keys or duplicate G_2 points. For PLONK and FFLONK verifiers, D4 checks extracted challenge names and order ($\beta, \gamma, \alpha, \zeta, v, u$ for PLONK; $\beta, \gamma, \xi, \alpha, y$ for FFLONK), resolves aliases (e.g., `pBeta`), and checks source-level hash call counts across `keccak256` and `SHA256`.

Intraprocedural dataflow analysis (D6). Since gas-optimized verifiers often defer return checks through memory or accumulators, D6 tracks precompile `staticcall` return values through variables, memory slots, and accumulators to identify return-check patterns in inline Yul assembly and Solidity code.

Algorithm 1 handles five patterns found in production verifiers: immediate branch checks, switch-case dispatch, boolean accumulators, cross-block memory tracking (e.g., Aztec UltraPlonk), and function-local validation helpers. Guard sets represent syntactically recognized checks; the analysis does not fully account for variable scope or establish whether recognized checks lead to rejection. Solidity-level guards (e.g., `require` or branch conditions outside assembly) are mapped directly to these checks.

Source-level and syntactic analysis (D2, D7–D10). Five detectors evaluate contract source directly. D2 flags raw field multiplication lacking recognized modular reduction. D7 flags custom EC routines and missing on-curve validation ($y^2 = x^3 + 3 \bmod p$) while recognizing precompile wrappers. D8 flags pairing calls whose `staticcall` success appears checked but whose output lacks a recognized output-use or validation pattern. D9 flags dynamic proof/public-input

Algorithm 1 Pattern-Based Precompile Return Check (D6)**Input:** Verifier source S (Solidity with inline Yul blocks)**Output:** Set of potentially unchecked precompile staticcall sites U

```

1:  $U \leftarrow \emptyset, C \leftarrow \emptyset$ 
2: for each precompile invocation  $s \in S$  do
3:   if IsWrapped( $s$ , pop) then
4:      $U \leftarrow U \cup \{s\}$   $\triangleright$  Return value discarded
5:   else if  $\neg$ IsWrapped( $s$ , iszero) then
6:      $V_s \leftarrow \text{CaptureVars}(s)$ 
7:      $C \leftarrow C \cup \{(s, V_s)\}$   $\triangleright$  Captured for pattern matching
8:   end if
9: end for
10:  $M \leftarrow \{slot \mapsto v \mid \text{mstore}(slot, v) \in S \wedge \exists s : v \in V_s\}$ 
11:  $G_v \leftarrow \text{GuardVars}(S), G_m \leftarrow \text{GuardSlots}(S)$ 
12: for each  $(s, V_s) \in C$  do
13:   if  $V_s \cap G_v = \emptyset \wedge \forall v \in V_s \forall slot (slot \mapsto v \in M \Rightarrow slot \notin G_m)$  then
14:     if  $\neg \text{AccumCheckPattern}(V_s) \wedge \neg \text{FnScopeGuard}(s)$  then
15:        $U \leftarrow U \cup \{s\}$ 
16:     end if
17:   end if
18: end for
19: return  $U$ 

```

parameters without recognized length-check patterns. Finally, D10 validates constant sanity, flagging zero-valued coordinates and base-vs-scalar prime confusion.

VI. EVALUATION

We evaluate ZKScan on three datasets: a synthetic dataset, an audit-derived dataset, and a mainnet dataset.

A. Setup

ZKScan is implemented in Python 3.12, and experiments were conducted on Ubuntu 24.04 with an Intel Core i7 processor and 64 GB of RAM. The tool analyzes Solidity source directly without compiler or chain dependencies and processes a contract in under 500 ms (averaging 54 ms).

Our evaluation suite contains 74 contracts across three datasets: an 18-contract **synthetic dataset** with 11 vulnerable and 7 safe contracts, a 20-contract **audit-derived dataset** reproducing published audit findings [12], [22], [23], [25], and a 36-contract in-the-wild **mainnet dataset**. For the audit-derived dataset, 15 contracts derive ground truth from published audits, and five use canonical `snarkjs` templates. Together, the audit-derived and mainnet datasets comprise 56 real-world contracts spanning Groth16 (28), PLONK (8), FFLONK (5), STARK (1), and 14 thin wrappers or interface stubs without inline pairing logic (e.g., Scroll, Loopring, Taiko SGX). We test historical code and do not claim active zero-day vulnerabilities.

B. Results

Alert distribution. Across the 56 real-world verifiers, ZKScan reports 133 raw alerts (Table II) as signals for review, not verified exploits. On the 15 audit-backed contracts alone, ZKScan achieves 80.0% precision and 94.1% recall, close to the full audit-derived dataset result. Table III shows per-class precision, recall, and true-positive (TP), false-positive (FP), and false-negative (FN) counts. Of the 18 ground-truth positives, 11 occur in the Linea pre-audit verifier, making recall especially sensitive to that contract.

TABLE II
ALERT DISTRIBUTION ACROSS 56 CONTRACTS BY PROOF SYSTEM
(MERGED AUDIT-DERIVED AND MAINNET CONTRACTS)

Group	C1	C2	C3	C4	C5	Tot.
Groth16 (28)	14	16	7	0	16	53
PLONK (8)	20	12	11	6	12	61
FFLONK (5)	2	0	5	0	0	7
STARK (1)	2	1	1	0	0	4
Other (14)	2	1	2	0	3	8
Total	40	30	26	6	31	133

TABLE III
EVALUATION ON SYNTHETIC AND AUDIT-DERIVED DATASETS

Class	Synthetic		Audit-derived		
	Prec.	Rec.	Prec.	Rec.	TP/FP/FN
C1: Input sanitization	1.00	1.00	1.00	1.00	7/0/0
C2: Field & curve arith.	1.00	1.00	0.80	1.00	4/1/0
C3: VK integrity	1.00	1.00	0.50	0.50	1/1/1
C4: Fiat-Shamir binding	1.00	1.00	1.00	1.00	1/0/0
C5: Pairing/precompiles	1.00	1.00	0.57	1.00	4/3/0
Overall (C1–C5)	1.00	1.00	0.77	0.94	17/5/1

TABLE IV
ABLATION OF YUL DATAFLOW ANALYSIS ON D6

Configuration	Raw alerts on merged audit-derived and mainnet dataset		Detection effectiveness on audit-derived dataset			
	Alerts	Contracts	TP	FP	Prec.	Rec.
Regex-only	366	42	3	25	0.11	1.00
Full ZKScan	25	3	3	3	0.50	1.00

Detection effectiveness. Overall, ZKScan achieves 94% recall (17/18) and 77% precision (17/22) on the audit-derived dataset ($F1=0.85$), and 100% precision and recall on the synthetic dataset. Given the small number of ground-truth findings, Wilson 95% intervals span 74.2–99.0% for recall and 56.6–89.9% for precision. On the synthetic dataset, ZKScan identifies all injected defects across all five classes. On the audit-derived dataset, C1, C2, C4, and C5 achieve high recall (1.00). The single false negative stems from a trusted-setup governance issue outside code-level scope (1 C3 FN). The five false positives occur in the Linea pre-audit contract (3), Nightfall (1), and Worldcoin (1); no false positives were observed in the evaluated Tornado Cash or Polygon zkEVM contracts.

Dataflow ablation. Table IV evaluates the empirical necessity of the Yul dataflow analysis (Algorithm 1) against a regex-only baseline that does not track how return values propagate. Without dataflow tracking, regex pattern matching triggers 366 raw D6 alerts across 42 contracts, including 79 false alarms on Aztec’s UltraPlonk verifier due to deferred return checks of precompile calls. Full ZKScan suppresses 341 of these alerts (a 93.2% alert-volume reduction), isolating 25 alerts across three contracts. Crucially, on the audit-derived dataset, this alert reduction preserves recall: D6 precision rises from 0.11 to 0.50 while recall remains 1.00, retaining all three expected true positives.

TABLE V
DETECTION COVERAGE BY VULNERABILITY CLASS

Vulnerability Class	ZKScan	Slither	Mythril
C1: Input and calldata sanitization	●	○	○
C2: Field and curve arithmetic	●	○	○
C3: VK and setup integrity	●	○	○
C4: Fiat-Shamir transcript binding	●	○	○
C5: Pairing logic and precompiles	●	○	○

● = class detected ○ = no applicable detection rule

Baseline comparison. We ran Slither [16] and Mythril [17] on all 18 synthetic dataset contracts. Slither completed 17 contracts and Mythril 14; the remainder failed to compile under available `solc` versions. Slither reported 225 *Informational* findings and Mythril reported one generic *Low* finding. None identified any of the eleven injected defects (Table V). Their rules do not reach inside inline assembly; for the missing on-curve check, Slither calls the validation routine dead code that “should be removed”. Neither tool models pairing or transcript semantics, so the key evaluation of our analysis design is the ablation in Table IV.

Practical impact and case study. At 77% precision on the audit-derived dataset, approximately three in four alerts are true positives. Across 56 real-world verifiers, ZKScan reports 2.4 alerts per contract on average (median 1; 17 produce none), suggesting a manageable triage burden. Beyond the low triage overhead, ZKScan demonstrates utility on production rollups. On the pre-audit Linea verifier, ZKScan completed analysis in 200 ms, detecting auditor-confirmed bugs [12], [23] in four of five classes while missing only the trusted-setup issue (C3). On the remediated version, ZKScan reports no alerts, which is consistent with the fixes.

VII. CONCLUSION

We presented ZKScan, a static analyzer for on-chain ZK verifier contracts with ten detector rules across five vulnerability classes and intraprocedural Yul dataflow analysis for the precompile return-value check (D6). On a 20-contract audit-derived dataset, ZKScan achieves 94% recall and 77% precision, detecting auditor-confirmed vulnerabilities in historical contract versions. It reduces false alarms compared to a regex-only baseline, particularly on optimized verifiers such as Aztec’s UltraPlonk. Slither and Mythril detect none of the injected ZK-specific defects in the synthetic dataset. ZKScan has limitations: it cannot track values across Solidity function boundaries or verify the mathematical correctness of pairing equations or prover-side constraints. Future work will add interprocedural taint analysis, a lightweight parsing expression grammar (PEG) parser for hand-written Yul, detectors for Cairo and Rust, and a larger dataset as more audit reports become public.

REFERENCES

- [1] V. Buterin, “An incomplete guide to rollups,” vitalik.eth.limo, 2021, [Online]. Available: <https://vitalik.eth.limo/general/2021/01/05/rollup.html>. Accessed: Oct. 7, 2026.
- [2] L2Beat, “Value secured by layer 2 scaling projects,” [Online]. Available: <https://l2beat.com/layer2s/tvs>. Accessed: Oct. 7, 2026.
- [3] Trail of Bits, “The Frozen Heart vulnerability in PlonK,” Trail of Bits Blog, 2022, [Online]. Available: <https://blog.trailofbits.com/2022/04/18/the-frozen-heart-vulnerability-in-plonk/>. Accessed: Oct. 7, 2026.
- [4] Q. Dao, J. Miller, O. Wright, and P. Grubbs, “Weak Fiat-Shamir attacks on modern proof systems,” in *Proc. IEEE S&P*. IEEE, 2023, pp. 199–216, doi: 10.1109/SP46215.2023.10179408.
- [5] Rekt News, “The unfinished proof: FOOMCASH and Veil Cash Groth16 verifier exploits,” Rekt News, 2026, [Online]. Available: <https://rekt.news/the-unfinished-proof>. Accessed: Oct. 7, 2026.
- [6] 0xPARC, “ZK bug tracker,” GitHub repository, 2023, [Online]. Available: <https://github.com/0xPARC/zk-bug-tracker>. Accessed: Oct. 7, 2026.
- [7] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, “Succinct non-interactive zero knowledge for a von Neumann architecture,” in *Proc. USENIX Security*, 2014, pp. 781–796.
- [8] J. Groth, “On the size of pairing-based non-interactive arguments,” in *Proc. EUROCRYPT*, ser. LNCS, vol. 9666. Springer, 2016, pp. 305–326, doi: 10.1007/978-3-662-49896-5_11.
- [9] J. Baylina and M. Bellés-Muñoz, “snarkjs,” GitHub repository, 2018, [Online]. Available: <https://github.com/iden3/snarkjs>. Accessed: Oct. 7, 2026.
- [10] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, “PLONK: Permutations over Lagrange-bases for oecumenical noninteractive arguments of knowledge,” IACR Cryptol. ePrint Arch., Rep. 2019/953, 2019, [Online]. Available: <https://eprint.iacr.org/2019/953>.
- [11] A. Fiat and A. Shamir, “How to prove yourself: Practical solutions to identification and signature problems,” in *Proc. CRYPTO*, 1986, pp. 186–194, doi: 10.1007/3-540-47721-7_12.
- [12] Consensus Diligence, “Linea Plonk verifier,” 2023, [Online]. Available: <https://diligence.security/audits/2023/06/linea-plonk-verifier/>. Accessed: Oct. 7, 2026.
- [13] A. Gabizon and Z. J. Williamson, “fflonk: A fast-Fourier inspired verifier efficient version of PlonK,” IACR Cryptol. ePrint Arch., Rep. 2021/1167, 2021, [Online]. Available: <https://eprint.iacr.org/2021/1167>.
- [14] C. Reitwiesner, “EIP-196: Precompiled contracts for alt_bn128 elliptic curve operations,” Ethereum Improvement Proposals, 2017, [Online]. Available: <https://eips.ethereum.org/EIPS/eip-196>. Accessed: Oct. 7, 2026.
- [15] V. Buterin and C. Reitwiesner, “EIP-197: Precompiled contracts for optimal Ate pairing check on alt_bn128,” Ethereum Improvement Proposals, 2017, [Online]. Available: <https://eips.ethereum.org/EIPS/eip-197>. Accessed: Oct. 7, 2026.
- [16] J. Feist, G. Grieco, and A. Groce, “Slither: A static analysis framework for smart contracts,” in *Proc. WETSEB*. IEEE, 2019, pp. 8–15, doi: 10.1109/WETSEB.2019.00008.
- [17] B. Mueller, “Mythril,” GitHub repository, 2018, [Online]. Available: <https://github.com/Consensus/mythril>. Accessed: Oct. 7, 2026.
- [18] S. Pailoor, Y. Chen, F. Wang, C. Rodríguez-Núñez, J. Van Geffen, J. Morton, M. Chu, B. Gu, Y. Feng, and I. Dillig, “Automated detection of under-constrained circuits in zero-knowledge proofs,” in *Proc. PLDI*. ACM, 2023, pp. 1510–1532, doi: 10.1145/3591282.
- [19] Trail of Bits, “Circomspect,” GitHub repository, 2022, [Online]. Available: <https://github.com/trailofbits/circomspect>. Accessed: Oct. 7, 2026.
- [20] Y. Fan, Y. Xu, and C. Garman, “SNARKProbe: An automated security analysis framework for zkSNARK implementations,” in *Proc. ACNS*. Springer, 2024, pp. 340–372, doi: 10.1007/978-3-031-54773-7_14.
- [21] D. Xiao, Z. Liu, Y. Peng, and S. Wang, “MTZK: Testing and exploring bugs in zero-knowledge (ZK) compilers,” in *Proc. NDSS*, 2025, doi: 10.14722/ndss.2025.230530.
- [22] S. Chaliasos, J. Ernstberger, D. Theodore, D. Wong, M. Jahanara, and B. Livshits, “SoK: What don’t we know? Understanding security vulnerabilities in SNARKs,” in *Proc. USENIX Security*, 2024, pp. 3855–3872.
- [23] OpenZeppelin, “Linea verifier audit,” 2023, [Online]. Available: <https://www.openzeppelin.com/news/linea-verifier-audit-1>. Accessed: Oct. 7, 2026.
- [24] ABDK Consulting, “Term Structure Labs audit, part 2,” ZK Security Reviews archive, 2023, [Online]. Available: <https://github.com/nullity00/zk-security-reviews>. Accessed: Oct. 7, 2026.
- [25] Trail of Bits, “AZTEC security assessment,” Trail of Bits Publications, 2019, [Online]. Available: <https://github.com/trailofbits/publications/blob/master/reviews/aztec.pdf>. Accessed: Oct. 7, 2026.
- [26] Consensus, “gnark,” GitHub repository, 2020, doi: 10.5281/zenodo.5819104. [Online]. Available: <https://github.com/Consensus/gnark>. Accessed: Oct. 7, 2026.