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*

https://www.usenix.org/conference/usenixsecurity24/presentation/chaliasos

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

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 [47, 48] to their current role as fundamental components that enable a wide array of practical applications [36]. 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) [47], ZKPs have, over the past decade, transitioned from theory to practical widely used implementation [15, 17, 31, 67, 70, 77, 82, 84].

On the forefront of the practical application of *general-purpose* ZKPs are Succinct Non-interactive Arguments of Knowledge (SNARKs) [26, 40, 44, 50, 75]. 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 [82], enhancing privacy in digital asset transfers [67, 84] and program execution [15, 17], as well as scaling blockchain infrastructure [60, 78, 79, 85]. Their versatility also extends to non-blockchain uses, such as in secure communication protocols [61, 96] and in efforts to combat disinformation [54, 56]. 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.

- **SNARKs system and threat models:** We provide the first framework for reasoning about systems built using SNARKs, analyzing interactions between different components, defining adversaries and their knowledge, and discussing potential implementation-level vulnerabilities and their impact.
- **Study of vulnerabilities:** We present the first systematic study of known vulnerabilities in systems using SNARKs. We gathered 141 vulnerabilities from 107 audit reports, 16 vulnerability disclosures, as well as a number of bug trackers of popular SNARK projects. When it comes to SNARKs, this is the first study of this scale in the literature. Further, because of the breadth of our coverage, we believe our findings to be representative of the entire SNARK space.
- **Vulnerabilities taxonomy:** We introduce a taxonomy for classifying vulnerabilities in SNARKs, highlighting unique vulnerabilities and common pitfalls in the SNARK stack that help researchers and practitioners better understand important threats in the SNARK ecosystem.
- **Analyzing defenses:** We analyze the main defense techniques proposed by the research and practitioner communities and highlight some notable gaps.

Key Findings. We find that developers seem to struggle in correctly implementing arithmetic circuits that are free of vulnerabilities, especially due to most tools exposing a low-level programming interface that can easily lead to misuse without extensive domain knowledge in cryptography. In detail, we find the following flaws to be most pressing: *(i) Implementation bugs across SNARK systems' layers*, including classic vulnerabilities like input validation errors and over/underflows. These can undermine SNARKs' core properties: completeness, soundness, and zero-knowledge. *(ii) The unique programming model for SNARK circuits poses challenges, often leading to under-constrained circuits*. This category emerges from overlooking constraints or misinterpreting logic into circuits. The low-level nature of SNARK Domain Specific Languages (DSLs), such as Circom, exacerbates vulnerabilities due to a lack of common high-level programming features such as basic types. *(iii) Design and implementation errors in proof systems* are critical yet often overlooked vulnerabilities. These errors may originate from the frameworks implementing the proof systems, like an implementation error in Gnark's Plonk verifier, or from the theoretical foundations themselves. An example is the "Frozen Heart" vulnerability, attributed to incomplete descriptions in the original proof system papers, leading to significant implementation errors [32].

It is important to highlight that ZK systems are *not* "just math" — they are complex, "compositional" systems where cross-layer interactions can introduce complex vulnerabilities. This paper attempts to cover the entire gamut of erroneous possibilities in the ZKP space.

2 Background on SNARKs

A ZKP enables an entity to prove that a statement is true, without disclosing anything besides the veracity of the statement. A ZKP is termed a ZK-SNARK if the proof size and verification time are sublinear in the statement to be proven, the communication between prover and verifier is non-interactive, and security holds against a computationally bounded prover. Common ZK-SNARKs are targeting the problem of *circuit satisfiability*, where the statement is represented as an arithmetic circuit. Hence, general-purpose SNARKs prove a fixed NP relation $\mathfrak{R}$, and allow the prover to convince a verifier that for the public input x they know a witness w such that $(x,w) \in \mathfrak{R}$. *Pre-processing SNARKs* [26], which are the focus of this work, additionally introduce a *setup* phase that encodes the relation being proven into a succinct representation.

In summary, a SNARK is composed of three algorithms:
$Setup(pp) \rightarrow (pk, vk)$. Given public parameters pp as input, output proving and verification keys pk and vk.
$Prove(pk, x, w) \rightarrow \pi$. Given the proving key pk, the public input x, and the witness w, output a proof π.
$Verify(vk, x, \pi) \rightarrow \{0,1\}$. Given the verification key vk, the public input x, and the proof π, output 1 if the proof is valid and 0 otherwise.

A ZK-SNARK satisfies the following security 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 veracity of a valid statement.
Zero Knowledge. The proof π reveals nothing about the witness w, beyond its existence.

For an in-depth introduction of ZKPs and SNARKs from a theoretical perspective, we refer the reader to [88].

3 System and Threat Models for SNARKs

We introduce a four-layer system model, showing how SNARKs are implemented in practice. Based on our system model, we provide a holistic threat model defining SNARK vulnerabilities, adversaries, and their potential impact.

3.1 System Model

Figure 1 depicts the system architecture of an application based on SNARKs for circuit satisfiability, i.e., argument systems that let the prover show that given a public input x, it knows a witness w such that the circuit satisfies $C(x,w) = y$. In our system model, we refer to four distinct layers. First,

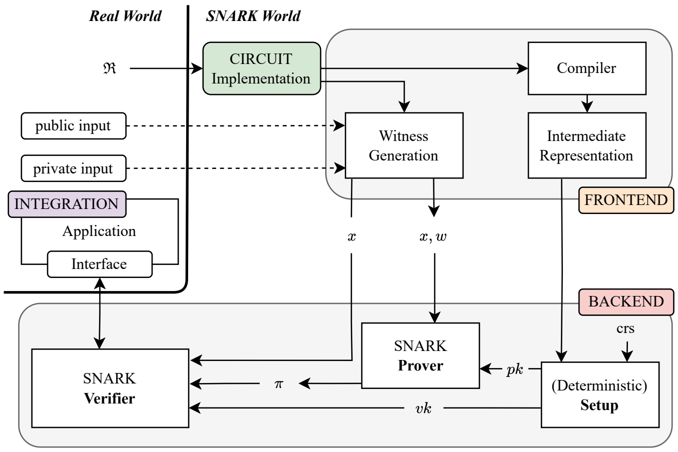

Figure 1: System model of an application based on SNARKs.

```
type Circuit struct {
   X frontend.Variable `gnark:",private"`
   C frontend.Variable `gnark:",public"`
   Y frontend.Variable `gnark:",public"`
}
func (circuit *Circuit) Define(api frontend.API) error {
  outputs := api.Compiler().NewHint(hint.SqrtHint, 1, circuit.X)
  squareRoot := outputs[0]
  api.AssertIsEqual(api.Mul(squareRoot, squareRoot), circuit.X)
  result := api.Mul(squareRoot, circuit.Const)
  api.AssertIsEqual(circuit.Y, result)
  return nil
}
```

Figure 2: Example circuit written in the eDSL gnark [16]. Lines highlighted in gray add constraints in the circuit.

the developer specifies the circuit in the *Circuit Layer*, according to the specification of the statement to be proven. The *Frontend Layer* enables the compilation of the circuit to a SNARK-friendly representation. The *Backend Layer* consumes this representation, and provides the concrete implementation of the proof system. The *Integration Layer* is the application logic that interacts with the proof system. We proceed to introduce the key components in each layer.

(i) CIRCUIT LAYER. SNARKs targeting circuit satisfiability require the computational statement to be represented as an arithmetic circuit. Most SNARKs work over elements in a finite field $\mathbb{F}_p$, and hence each wire in the arithmetic circuit is represented as an element in $\mathbb{F}_p$. Note, that there are also SNARKs employing Ring arithmetic [42], with the purpose of achieving more efficient operations over $\mathbb{Z}_{2^{64}}$ to match native execution on standard CPUs. When developing a ZK system, there are several considerations at different levels of the stack that must be taken into account by developers: *(i)* determining what inputs and outputs of the computation are exposed publicly, *(ii)* encoding the application logic as an arithmetic circuit in $\mathbb{F}_p$, and *(iii)* specifying how applications should compose with the circuit (e.g. by ensuring that public inputs are well-formed before verifying a proof). Specifying circuits correctly is unintuitive, as arithmetic circuits do not natively support non-arithmetic operations. Moreover, developers face challenges in translating conventional programming logic into circuit formats while simultaneously addressing the dual requirements of value assignment for witnesses and applying constraints to check the validity of solutions. The prover is responsible for assigning these values (i.e., witness assignments in the circuit), but the verifier has to check that these assignments adhere to the constraints (i.e., constraints in the circuit). Crucially, developers must remember that a malicious prover might manipulate witness values, bypassing circuit logic. Therefore, it is vital to rigorously constrain the witness within the circuit to only validate legitimate solutions. If the constraints are not sound, then there is the risk of exploitation (see Section 5). For example, expressing $X \neq 0$ is trivial in a "normal" programming language, whereas encoding it in a custom circuit is non-trivial. By Fermat's Little Theorem, one can check $X^{p-1} = 1$, but this would cost $O(logp)$ constraints [71]. An alternative is to have the prover provide a *hint* $H = X^{-1}$, the multiplicative inverse of X, to check $X \cdot H = 1$. Since all nonzero elements in $\mathbb{F}_p$ have a multiplicative inverse, this constrains X to be nonzero by deduction [57]. We provide a concrete example of a circuit leveraging a hint in gnark for efficiently computing the square root in Figure 2. In practice, developer tools employ differing approaches to ease circuit specification in high-level programming interfaces:

Circuit Domain Specific Languages. Domain Specific Languages (DSLs) are specialized programming languages designed to address specific problem domains. In the case of SNARKs, they offer a tailored syntax to efficiently express constraints in an arithmetic circuit [3,11,28,33,71,75]. Learning a DSL might be challenging, especially for developers not familiar with SNARKs.

Circuit Embedded Domain Specific Languages. An Embedded Domain Specific Language (eDSL) is a type of domain-specific language that is embedded within a host general-purpose programming language. For developing SNARKs, several eDSLs have emerged in recent years, embedded as libraries in Golang [16], Rust [37,81,95], and TypeScript [69]. eDSLs can seamlessly interact with other code written in the host language, allowing easy integration with existing libraries. At the same time, they require developers to actively distinguish between in-circuit and out-of-circuit operations, which requires domain expertise to ensure correct implementation.

ZK Virtual Machines. Circuit DSLs and eDSLs allow developers to specify circuits in a manner similar to *application-specific integrated circuits* (ASIC). Zero-Knowledge Virtual Machines (ZK-VMs) follow a different programming model, where the arithmetic circuit represents the loop of fetching instructions from memory and successively executing them (similar to the fetch-decode-execute cycle as observed in a general-purpose CPU). Common ZK-VMs target existing Instruction Set Architectures [4,18,60,79,85]. For example, ZK-EVMs target the primitive instructions (i.e., "opcodes") of the Ethereum Virtual Machine [60, 79, 85], and RISC Zero [18] and Jolt [4] operate on the RISC-V instruction set. Similarly, there are ZK-VMs for custom ISAs that are optimized for proving in a SNARK [46, 78]. Some ZK-VMs, such as the Cairo ZK-VM [46], provide a DSL to specify programs for the ZK-VM. ZK-VMs can be beneficial as they operate on existing instruction sets and can leverage existing tooling. Writing circuits directly, while more error-prone, can be more efficient due to access to low-level optimizations.

(ii) FRONTEND LAYER: The frontend of a SNARK for circuit-satisfiability compiles the high-level program written by the developer to a representation (i.e., *arithmetization*) which is amenable for the proof system in a backend. Frontends are agnostic to proof systems, as in they can provide a compilation of high-level programs to differing arithmetizations. At runtime, the frontend assigns all intermediate wires in the circuit in order to generate the public and private parts of the *witness*. We introduce the components in detail as follows:

Arithmetization. An arithmetization is a representation of the constraint system in a mathematical form, typically a set of algebraic equations, that can be efficiently processed by the argument system, i.e., the backend of a SNARK. Common arithmetizations include rank-1 constraint systems (R1CS) [44], variations of Plonkish arithmetization [40], Arithmetic Intermediate Representations (AIR) [13] and Customizable Constraint Systems (CCS) [86]. Different arithmetizations result in a different cost profile for the respective proof system that relies on them, and notably, they introduce different limitations with respect to how the circuit is defined. For example, R1CS does not support constraints that have a polynomial degree larger than two, and AIR requires circuits to be uniform. Notably, Setty et al. provided CCS, a constraint system that generalizes Plonkish, AIR, and R1CS arithmetizations [86].

Circuit Compiler. The circuit compiler in a SNARK implementation compiles the circuit specified in a high-level DSL or eDSL to the respective arithmetization. Note, that not all compilers target the compilation of high-level languages to specific arithmetizations. For example, high-level programs can first be compiled to VampIR [1] and ACIR [2], which are intermediate representations that allow compilation to, e.g., R1CS or some form of Plonkish arithmetization and aim to ease the support of multiple backends. Similarly, zkLLVM provides a compiler from the LLVM IR to different arithmetizations, allowing users to prove the execution of, e.g., native Rust, Golang, or C++ [3].

Witness Generation. The main task of the witness generator is to calculate the intermediate wires for a given circuit $\mathcal{C}$, given the assignment of public and private inputs. Note that the concept of a separate witness generator is not necessarily implemented by every frontend. Often, the circuit code can be executed to either produce constraints at compile time (e.g., the line highlighted in grey in Figure 2) or to produce witness values at runtime/proving (e.g., the hint in Figure 2). When an auxiliary "witness generator" exists, instructions are generated on how to fill a symbolic witness table and the witness generator is a binary generated at compile time.

(iii) BACKEND LAYER. Given the arithmetic circuit and witness, the backend specifies the algorithmic implementation of proof systems (e.g., Groth16 [50], Plonk [40], Stark [13], Marlin [26]). At a high level, the backend algorithms follow the API as in Section 2.

Setup. In pre-processing SNARKs, the *setup* algorithm generates the prover and verifier keys (*pk* and *vk*) by encoding the circuit relation, with the primary goal of succinct verification and optimized proving. However, different SNARKs have vastly differing properties with regard to trust assumptions, which in turn also impact their performance. Some SNARKs require a *trusted setup*, i.e., a randomized setup phase that introduces a trapdoor either per circuit (i.e., non-universal) [50] or in a universal setup phase [40]. The *randomized* part of the setup involves public parameters and random input to generate a common reference string (CRS) [68], which can be used to derive *pk* and *vk* in the deterministic part of the setup for a concrete circuit instantiation. The trapdoor needs to be discarded after the setup proccess is done, as anyone who has knowledge of the trapdoor can forge proofs. SNARKs that do not employ a trapdoor in the setup phase are commonly coined *transparent* [13]. Note, that this introduces a trade-off in verifier efficiency — transparent SNARKs only have polylogarithmic-time verification as compared to constant-time for SNARKs with trusted setup [88].

Prover & Verifier. The prover consumes the values generated by the frontend, where the circuit is represented in a specific arithmetization. At a high level, the prover of a pre-processing SNARK (aside from Groth16 [50]) utilizes a *polynomial commitment* to commit to the satisfying witness, and successively engages with the verifier in an interactive protocol to evaluate the correctness of assignments with regard to the circuit relation. This "recipe" leads to a SNARK if the verifier is public-coin [76], and the prover can hash the transcript of the interactive proof to render the protocol *non-interactive* (a process commonly denoted as applying the "Fiat-Shamir heuristic"). Different proof systems choose different combinations of polynomial commitments and interactive proofs,

[1] https://github.com/anoma/vamp-ir
[2] https://github.com/noir-lang/acvm
[3] https://github.com/NilFoundation/zkLLVM

resulting in differing properties for the respective target applications. The most popular approaches include Plonk [40], which employs a constant-round polynomial interactive oracle proof (IOP) with the KZG commitment [55]. Halo2 [95] uses the Plonk IOP with the Bulletproofs polynomial commitment [19]. Another class of protocol combines a polynomial commitment based on Merkle hashes with a *low-degree test*, resulting in a protocol that does not demand for a trusted setup or operations in elliptic curve groups [12], at the cost of non-constant verification time and a larger proof size [35]. Plonky2 [81] uses the FRI-based polynomial commitment scheme and combines it with the Plonk IOP. Groth16 [50], the SNARK with the shortest proof size, is based on probabilistically checkable proofs (PCP) [44] and pairings over elliptic curves for proof verification. For an in-depth taxonomy, we refer readers to [88]. Some applications leverage one of the above SNARKs without the *zero-knowledge* property, solely relying on non-interactivity and succinctness (e.g zk-EVMs [79, 85]). In most SNARKs, zero-knowledge can be obtained cheaply with minor modifications.

(iv) INTEGRATION LAYER. In our system model, we collectively term any application-specific implementation that doesn't directly relate to the circuit layer, frontend layer, or backend layer, the *integration layer*. This includes source code interacting with any of the other layers, application logic that may impact the overall security of employing SNARKs, as well as composition and aggregation of proofs.

Code Interacting with SNARK Components. A SNARK application requires code that interacts with the functions exposed by the prior layers. Typically, developers use Solidity, JavaScript, or any other language to perform API calls to the frontend or backend. For performance reasons, many applications reduce circuits to their minimal size to fulfill a specific functionality, outsourcing operations, like range-checks for certain input values, to native application code.

Complementary ZKP Logic. An application might employ complementary logic beyond the SNARK itself to ensure its security properties. For example, nullifiers are commonly used as private values that "nullify" a specific object upon its use to ensure that specific operations are only executed once. Consider the use of nullifiers in Tornado Cash [77]. A user deposits a token in a smart contract and associates a secret nullifier with it. When a user wants to withdraw, the contract checks that the nullifier has not been used before to prevent double-spending. The user additionally generates a SNARK proof, which proves that the nullifier is associated with a deposit, without revealing the exact deposit to achieve anonymity.

Proof Delegation, Aggregation, Recursion & Composition. In some applications, proofs can be too computationally expensive to be generated on end-user devices. To achieve proof generation on computationally restricted devices, one may delegate the proof generation to an untrusted party [64]. Similarly, one may require a fast prover (linear in the size of the statement) *and* a fast verifier (constant in the size of the circuit). In this case, common projects employ a recursive composition of proofs, i.e., they first leverage a SNARK with a fast prover and successively prove the verification in a SNARK with a fast verifier. For example, Polygon ZK-EVM [79] composes a FRI-based SNARK with a Groth16 to obtain the succinct proof size and verification for cheap verification of proofs in a smart contract, whilst benefiting from the faster prover time and decreased size of the reference string required for Groth16. Note that in this case, the trust assumption reduces to the weakest component, i.e., Groth16 still requires a trusted setup. There are several other works that employ a similar strategy to obtain efficient SNARK constructions through aggregation or recursive composition [34, 41, 59, 83, 92, 93].

3.2 Threat Model Taxonomy

We consider any usage of a SNARK that violates the intended behavior of one of the previously discussed layers as a SNARK vulnerability. In this section, we first define the scope of vulnerabilities at each layer, outline the roles an adversary can take that may lead to a specific adversarial impact throughout our system model.

Scope of Vulnerabilities. In the *(i) circuit layer*, vulnerabilities may result from coding mistakes on the implementation of circuits that can lead to having inconsistent or weak constraints that break the soundness and/or completeness of the system. A primary reason for this is mistranslation due to developers not being used to writing in a differing programming model that introduces surprising pitfalls. In the circuit layer, we exclude logic bugs that are not necessarily associated with writing SNARK circuits, and focus on issues that arise from the use of SNARK software specifically. In the *(ii) frontend layer*, vulnerabilities may primarily arise due to bugs in compiling from high-level source code to a specific arithmetization. In the *(iii) backend layer*, vulnerabilities may occur in the prover, where adversarial provers attempt proof forgery. For SNARKs requiring a trusted setup, applications might use an MPC-ceremony for generating the reference string trustlessly. Vulnerabilities in the MPC-ceremony are considered out of scope, as they do not directly pertain to SNARKs. Vulnerabilities in the *(iv) integration layer* resemble any issue in the software components that interact with a SNARK. For example, circuits can have implicit constraints that ought to be checked in the integration layer. We exclude traditional vulnerabilities, such as reentrancy for smart contracts, as they are not unique to systems using SNARKs.

(i) ADVERSARIAL ROLES: Throughout this work, we assume that adversaries are rational agents aiming to attack systems utilizing SNARKs. We outline the knowledge an adversary can obtain given its role in our system model in Table 1:

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.

R1 - Network Adversary. A network adversary can observe the public values transmitted between participants in the SNARK application. Hence, it can obtain knowledge about the public witness x, the public common reference string, and the proof π sent from the prover to the verifier. A network adversary may attempt to exploit weak simulation extractability of certain proof systems that can render a proof malleable [45].

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 [27]), 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.

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.

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.

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 [60], Polygon ZK-EVM [79] Scroll [85]),[4] privacy-centric blockchains (e.g., Zcash [84], Aztec [6], and decentralized privacy applications (e.g., Tornado Cash [77]). We also assess key circuit libraries and SNARK frameworks (e.g., Circom and halo2) [35]. Our analysis further extends to inspecting the underlying proof systems for known bugs. This comprehensive approach ensures that we cover a broad spectrum of real-world SNARK applications and the various layers of technology they rely on.

Data Sources. In our comprehensive study of security vulnerabilities related to SNARK systems, we have utilized an extensive range of data sources to ensure a thorough analysis. Our primary source includes critical and high-impact vulnerabilities identified in 107 security audit reports of systems employing SNARKs. Among these, 75 reports specifically highlighted vulnerabilities directly related to SNARKs, while the remaining reports detailed other types of vulnerabilities, such as common smart contract deficits. Additionally, we have incorporated vulnerabilities from public bug bounty programs and vulnerability disclosures. Our search for disclosures included those from prominent projects like Zcash, Aztec Connect, and TornadoCash, disclosures by security firms and researchers specializing in ZKPs, and comprehensive web searches targeting ZKP vulnerabilities. We reviewed Web3 bug bounty programs like Immunefi, though we found no ZKP-related disclosures there. Additionally, we consulted closely with top audit firms focusing on ZKPs to ensure we didn't miss any critical disclosures. These sources provide real-world examples of how vulnerabilities have impacted operational systems as well as the measures taken to address them. Furthermore, we delved into bug trackers of popular GitHub projects related to SNARKs. Our focus here has been specifically on security-related bugs, filtering out non-security-related issues. Table 2 in our paper depicts these varied data sources, collectively forming the foundation of our study of SNARK security.

Classification. We classify vulnerabilities by determining the layer that they occur in with respect to our system model (cf. Section 3.1). Successively, we asses the vulnerability type (e.g., over-constrained), identify its root cause, and assess whether it impacts soundness, completeness, or zero-knowledge when exploited (cf. Table 3). We provide a full classification of all vulnerabilities as an open-source dataset as a reference for practitioners.[5]

Threats to validity. Our investigation into vulnerabilities within SNARK systems faces several challenges that could affect the validity of our findings. Firstly, the absence of proof-of-concept exploits for many bugs raises questions about their actual exploitability. We try to mitigate this by excluding non-security vulnerabilities from our dataset. Secondly, while our classification of vulnerabilities was rigorously reviewed by multiple researchers and iteratively refined, the possibility of misclassification cannot be entirely ruled out. Our reliance on publicly available reports means our analysis may be skewed towards more commonly detected vulnerabilities, potentially overlooking others that are less apparent but equally prevalent.

[4]Notably, ZK-rollups have more than 1B USD in accumulated value locked in them according to L2BEAT https://l2beat.com/scaling/summary

[5]https://docs.google.com/spreadsheets/d/1E97ulMufitGSKo_Dy09KYGv-aBcLPXtlN5QUpwyv66A

Root Cause	**UC**	**OC**	**CE**	**Total**
Assigned but Unconstrained	14	0	0	14
Missing Input Constraints	25	0	0	25
Unsafe Reuse of Circuit	9	0	0	9
Wrong translation of logic into constraints	32	0	2	34
Incorrect Custom Gates	1	0	0	1
Out-of-Circuit Computation Not Being Constrained	1	0	0	1
Arithmetic Field Errors	8	0	0	8
Bad Circuit/Protocol Design	4	0	0	4
Other Programming Errors	1	1	1	3
Total	**95**	**1**	**3**	**99**

Table 4: Circuit vulnerabilities. UC: Under-Constrained, OC: Over-Constrained, CI, Computational/Hints Error.

5 Circuit Issues

Vulnerabilities at the circuit layer represent the most prevalent threat to systems using SNARKs (c.f. Table 2). The primary challenge for developers lies in adapting to a different level of abstraction, coupled with the need to optimize circuits for efficiency, as they significantly influence the cost of using a SNARK. This section initially highlights the three primary vulnerabilities encountered in the circuit layer, followed by outlining the root causes that lead to these vulnerabilities (see Table 4). In total, 95 circuit vulnerabilities led to soundness issues and 4 led to completeness issues (c.f. Table 3).

5.1 Vulnerabilities

V1. Under-Constrained: The most frequent vulnerability in ZK circuits arises from insufficient constraints. This deficiency causes the verifier to mistakenly accept invalid proofs, thus undermining the system's soundness or completeness.

V2. Over-Constrained: Although less common than under-constrained issues, circuits can be over-constrained, leading to the rejection of valid witnesses by honest provers or benign proofs from honest verifiers. This issue stems from extra constraints in the circuit, where legitimate solutions cannot be proven or verified, leading to DoS issues. Nevertheless, over-constrained bugs should not be confused with redundant constraints that add no additional value but do not lead to any issues other than introducing computational inefficiencies.

V3. Computational/Hints Error: Occurs when the computational part of a circuit is erroneous, often leading to completeness issues where for correct inputs, the witness generation either fails or produces wrong results. Note that completeness issues can often be transient, meaning that you can fix the underlying issue without having to update the circuit and recompute the prover and/or the verifier keys. Computational issues may also result in soundness issues if the constraints are applied using the same erroneous logic.

5.2 Root Causes

R1. Assigned but not Constrained: A frequent issue in ZK circuit design lies in distinguishing between assignments and constraints. While constraints are mathematical equations that must be satisfied for any given witness for the proof to be valid, assignments allocate values to variables during the witness generation process. The problem arises when variables are assigned values based on correct computations but lack corresponding constraints. This oversight can lead the verifier to erroneously accept any value for these variables.

R2. Missing Input Constraints: Developers sometimes neglect to apply constraints on input variables in reusable circuits. This omission occurs either (i) unintentionally or (ii) because they anticipate these constraints will be enforced at a different interface level (i.e., caller circuit or integration layer), thus omitting the constraints for optimization reasons. However, the absence of clear documentation or fully constrained inputs in these circuits can lead to severe vulnerabilities. Note that this issue is particularly common in low-level circuit DSLs, such as Circom, which lack user-defined types.

R3. Unsafe Reuse of Circuit: In ZK circuit design, particularly when using DSLs like Circom, the practice of reusing circuits (such as templates in Circom or gadgets in halo2) can introduce vulnerabilities if not handled correctly. This primarily occurs in two scenarios: *(i) Implicit Constraints in Sub-Circuits*, occurred when circuits are reused without appropriately constraining their inputs or outputs based on the assumption that the user will apply these constraints on call-site. *(ii) Insecure Circuits Instantiation* when circuits are meant to be used for specific setups (e.g., specific curves). An example is the Sign template from circomlib, which was designed solely for the BN254 curve field.[6]

R4. Wrong translation of logic into constraints: Translating computations for ZK circuits presents a unique set of challenges, primarily due to the distinct programming model of ZK circuits compared to traditional CPU-targeted code. A significant issue arises when translating logic that involves types and operations available in conventional programming languages but are either absent or must be re-implemented in ZK frameworks (e.g., fixed-point arithmetic). This often leads to the need for creative but error-prone solutions, such as using multiplexers for conditional logic. Developers might inadvertently omit essential constraints or simplify the logic to reduce the number of constraints, potentially missing critical corner cases. This can leave variables under-constrained, allowing them to accept multiple or any values under certain conditions, thus deviating from the developer's intent and introducing vulnerabilities.

[6] https://github.com/iden3/circomlib/blob/master/circuits/sign.circom#L23

R5. Incorrect Custom Gates: In implementations following the TurboPLONK model, such as Halo2, circuit constraints are defined by using custom gates applied to specific rows and cells of a table that constitute the Plonkish representation of the circuit. However, this approach can lead to bugs when custom gates are incorrectly handled. Errors may arise from inaccurately determining the appropriate offsets, resulting in misalignment with the intended behavior of the circuit.

R6. Out-of-Circuit Computation Not Being Constrained: Out-of-circuit computation refers to computations within the code that do not directly impact the witness generation yet play a crucial role in the overall functionality. In DSLs like Circom, certain functions operate outside the circuit logic. Similarly, in eDSLs, standard code (e.g., vanilla Rust) is used for various computations that do not affect witness generation. For instance, computations like division, are typically performed out-of-circuit to optimize circuit efficiency. This method involves executing the computation externally and then witnessing the result back into the circuit where it is constrained to ensure correctness. Issues arise when these out-of-circuit computations lead to missing assignments or when constraints necessary for the circuit's integrity are overlooked. A specific manifestation of this issue is the *boomerang issue*, where a variable, initially constrained within the circuit, is temporarily moved out-of-circuit and then reintegrated without the necessary reapplication of constraints.

R7. Arithmetic Field Error: Working with field arithmetic in ZK circuits can be challenging, especially as developers might overlook the nuances of computations within a finite field. The most common issues in this context are arithmetic overflows and underflows. We categorize the primary types of overflows and underflows in ZK circuits as follows: (i) *Native Field Arithmetic Over/Underflow*, occurs when circuit computations exceed the finite field's limits, causing values to loop back within the field's range due to modulo arithmetic. (ii) *Overflows in Transformed Formats*, risks of overflows arise when numbers are transformed into bit representations for specific operations (e.g., range checks) or for emulating non-native arithmetics like fixed-point arithmetics. This is particularly problematic when multiple bit representations remain within the field's overflow limits, leading to under-constrained vulnerabilities.

R8. Bad Circuit/Protocol Design: Circuit design issues in SNARKs often stem from fundamental flaws in how circuits are conceptualized, potentially leading to unintended behaviors or the violation of protocol properties. A common manifestation of this problem is the incorrect categorization of

Root Cause	PDE	PCE	PUD	ZKPCLE	Total
Integration Design Error	1	2	0	0	3
Missing Validation Input	0	0	9	1	10
Total	**1**	**2**	**9**	**1**	**13**

Table 5: Integration vulnerabilities. PDE: Proof Delegation Error, PCE: Proof Composition Error, PUD: Passing Unchecked Data, ZKPCLE: ZKP Complementary Logic Error

variables – such as designating a variable as private when it should be public. These issues can significantly impact the functionality and security of the protocol.

R9. Other Programming Errors: This category encompasses a range of common programming errors that do not fit neatly into other specific vulnerability categories but still have significant implications for the integrity of SNARK systems. These errors include, but are not limited to, API misuse, incorrect indexing in arrays, and logical errors within the computational parts (i.e., witness generation) of the circuit that are not directly related to constraint application.

Countermeasures. To address circuit layer vulnerabilities in SNARK systems, especially the common issue of under-constrained vulnerabilities, several straightforward yet effective strategies are recommended. Firstly, adding missing constraints, particularly range checks, is crucial to ensure the integrity and robustness of the circuits. In-depth documentation of circuit design and SNARK system specifications can significantly aid in preventing misunderstandings and oversights during development and auditing. Additionally, adopting DSLs, whenever possible, that support modern programming features, such as abstractions and basic types, enhances the development process by providing clearer guidelines and reducing the likelihood of errors. Lastly, the use of specialized security tools designed for ZK circuits, as detailed in Section 8, can detect some vulnerabilities during the developing phase, further securing SNARK systems against vulnerabilities.

6 Integration Layer Issues

Besides circuit-related problems, numerous vulnerabilities in systems employing SNARKs originate from the integration layer. These vulnerabilities often stem from improper interactions between the application/system and the prover or verifier, or from design flaws within this layer that compromise the inherent properties of SNARKs. Table 5 categorizes all analyzed bugs into distinct categories and root causes.

6.1 Vulnerabilities

V4. Passing Unchecked Data: This vulnerability manifests when implicit constraints on the public inputs expected by the circuit are not enforced by the application's verifier. It can result in both soundness and completeness issues. While unchecked data can be input in the circuit directly or indirectly (by hashing it), the failure to enforce these implicit constraints at the integration layer can compromise the integrity of the entire system.

V5. Proof Delegation Error: In scenarios where proof generation is delegated to an untrusted prover, there's a risk of malicious activities. For example, a bad design could lead to the leaking of personal or secret data. This issue underlines the need for secure and trusted channels in proof delegation and decentralized proving services [43].

V6. Proof Composition Error: This issue arises when the logic is distributed across multiple proofs, but the intended behavior is not adequately enforced by the verifier who is expected to glue parts of certain proofs with parts of others. The lack of coherence and coordination between different proofs can lead to undefined behaviors in the application.

V7. ZKP Complementary Logic Error: This category encompasses vulnerabilities in the integration layer arising from the flawed implementation of logic that operates in conjunction with ZKPs. An illustrative example of such a flaw is the poor management of protocol-specific mechanisms like nullifiers. Consider a privacy-preserving application like Tornado Cash (TC), where users aim to dissociate deposits from withdrawals. TC employs ZKPs alongside a nullifier scheme; each deposit involves generating a nullifier and a secret, which, when hashed, are appended in a Merkle tree in TC.[7] Upon withdrawal, the user submits the nullifier and employs a SNARK to prove knowledge of the corresponding secret without revealing it, thus unlinking the withdrawal from the deposit. However, a critical check within the integration layer is required to ensure the unique use of each nullifier. Without this check, the system is vulnerable to exploitation, allowing a user to repeatedly withdraw funds, thus draining TC. This class of bugs, crucially, is not due to a flaw in the circuit's design or implementation but stems from an oversight in auxiliary mechanisms that are integral to the secure and correct application of ZKPs.

6.2 Root Causes

R10. Missing Validation Input: This root cause involves inadequate validation checks on data before its input into the circuit or before computations that affect data used later by the circuit. It arises when the integration layer does not properly validate input data or when the circuit itself lacks necessary checks. This oversight can lead to the acceptance of inappropriate data, potentially getting an application stuck, or worse, compromising the circuit's computations.

[7] For more information on how TC works we refer the reader to the official documentation https://docs.tornadoeth.cash/generals/how-does-tornado-cash-work

Layer	Vulnerability	# Bugs
Frontend	Incorrect Constraint Compilation	3
	Witness Generation Error	3
	Total	**6**
Backend	Setup Error	1
	Prover Error	6
	Unsafe Verifier	16
	Total	**23**

Table 6: Summary of frontend and backend bugs.

R11. Integration Design Error: Design flaws in the integration code that lead to undesired behavior or break protocol properties, such as not checking for duplicate nullifiers, are critical. Furthermore, designing privacy-enhanced applications poses a significant challenge. It is not straightforward to create ZK applications with privacy as a core property, and developers may inadvertently leak information without realizing it. This issue isn't about the proof system or constraints; it is about the inherent difficulty in designing an application that doesn't subtly leak data. This underscores the complexity and importance of carefully considering privacy aspects in the design phase of ZK applications. Fixes often require a high-level redesign rather than a simple programming correction.

7 Frontend/Backend Issues

The compilation of ZK circuits (frontend), along with proving a witness and verifying it (backend), rely on the correctness of the underlying software that implements the compilation process as well as the algorithms that underpin the proof system. Table 6 provides an overview of vulnerabilities that can occur within these components. Vulnerabilities in either the frontend or backend layers pose a significant risk, even if the circuits have been formally verified or if the proof system is theoretically secure; any defects in these layers could render the entire system insecure for end-users.

7.1 Frontend Vulnerabilities

V8. Incorrect Constraint Compilation: This type of vulnerability emerges from deficiencies in the compilation phase, specifically in the enforcement of constraints defined in the ZK circuit. Specifically, such issues may stem from the arithmetization process or overly aggressive optimization routines that unintentionally excise vital constraints. Any missteps in these processes can result in an inaccurate or incomplete translation of the original constraints at the circuit layer. Consequently, this could permit a verifier to erroneously accept an invalid proof or, conversely, to reject a legitimate one. At its core, this vulnerability mirrors a classical compiler bug, wherein the compiler erroneously interprets high-level DSL code.

V9. Witness Generation Error: This issue arises during the witness generation phase, where errors in the compilation can produce invalid witnesses or cause crashes, potentially leading to a denial of service. Typically, such errors occur due to misinterpretation of the circuit code, or due to the generation of bogus witness generators in target languages.

7.2 Backend Vulnerabilities

V10. Setup Error: This vulnerability arises during the setup phase, where the generation of public parameters occurs. Incorrect or easily compromised parameters can significantly undermine the system's integrity.

V11. Prover Error: Issues within the prover of the proof system fall into this category. These vulnerabilities can lead to the prover mistakenly rejecting valid witnesses, accepting invalid ones, or breaking the zero-knowledge property.

V12. Unsafe Verifier: This category includes vulnerabilities resulting from inadequate checks on the verifier's inputs, missing checks during verification, or bogus mathematical operations, risking the proof system's integrity. Vulnerabilities can emerge regardless of whether the verifier is manually implemented or generated by a framework.

7.3 Root Causes

Frontend and backend vulnerabilities can stem from a variety of root causes, ranging from basic programming errors to configuration issues or deviations from the specifications of proof systems. These can be as simple as missing validation checks for proofs or as complex as informational leaks that expose sensitive data. Additionally, the use of poor-quality or predictable randomness sources can compromise crucial ZKP properties. Adherence to proof system specifications is critical in preventing vulnerabilities. A common issue is the incorrect implementation of the Fiat-Shamir transformation, where a critical component is omitted during the hashing process of the transcript.

8 Defenses For SNARKs

This section presents an overview of defense mechanisms aimed at mitigating the SNARK vulnerabilities detailed in the previous sections. Table 7 compiles, to our best knowledge, all publications and tools associated with SNARK security. A filled bullet in the table indicates that the referenced technique provides at least one defensive measure against the issues indicated by the corresponding row's header. Notably, in addition to SNARK-specific strategies, conventional security tools (e.g., fuzzing) have been utilized in audits, although they typically fall short in preventing most vulnerabilities due to the oracle problem [9]. For example, fuzzing might be

	Vulnerability	Defenses										
		Circomspect [89]	ZKAP [91]	Korrekt [87]	Coda [63]	Ecne [90]	Picus [74]	Aleo [29,30]	OWBB23 [73]	SnarkProbe [38]	CIVER [53]	Trad Sec*
Circuit	Under-Constrained	●	●	●	●	●	●	●	○	●	●	○
	Over-Constrained	○	○	●	●	○	○	●	○	●	○	○
	Computational Error	○	○	○	●	○	○	●	○	○	○	●
Frontend	Incorrect Constraint Compilation	○	○	○	○	○	○	○	●	○	○	○
	Witness Generation Error	○	○	○	○	○	○	○	○	○	○	●
Backend	Setup Error	○	○	○	○	○	○	○	○	●	○	○
	Prover Error	○	○	○	○	○	○	○	○	○	○	●
	Unsafe Verifier	○	○	○	○	○	○	○	○	○	○	●

Table 7: An overview of papers and tools offering defense mechanisms for addressing vulnerabilities in SNARK-based systems. While numerous techniques are concentrated on circuit-layer security, they often face challenges in scalability or possess limited functionality tied to specific DSLs. The last column represents traditional security tools (e.g., AFL and property-based testing) employed by auditors in audit reports. Note that these tools typically have limited capabilities for detecting SNARK-related bugs. We excluded the integration layer from the table as no defense detects bugs in that layer.

useful in detecting completeness bugs, but it often falls short of finding soundness issues. In the following, we delve into the current defensive measures for each layer, suggest future steps for preventing SNARK vulnerabilities, and review the tools employed in the audits we examined.

Circuit Layer Defenses. The circuit layer emerges as the most prominent layer for SNARK vulnerabilities, leading to the development of various techniques like static analysis and symbolic execution, particularly targeting under-constrained bugs. Tools such as Circomspect [89] and ZKAP [91] employ static analysis using predefined rules to identify under-constrained issues in Circom circuits. Similarly, Korrect [87] utilizes static analysis and SMT solvers to spot common pitfalls in Halo2 circuits. SNARKProbe [38] leverages fuzzing and SMT solvers to test circuits written using R1CS-based libraries. Picus [74], on the other hand, adopts a more advanced symbolic execution approach to verify that Circom circuits are not under-constrained. CIVER [53] uses a modular technique based on the application of transformation and deduction rules to verify properties of Circom circuits using pre- and post-conditions. In a different approach, DSLs like Coda [64] and Leo [29, 30] support formal verification of circuits, aiding developers in creating more secure circuits.

Despite these initial strides in tool development for ZK circuits, significant limitations remain. Tools relying on SMT solvers, like Picus and Korrect, face challenges due to limited support and efficiency in handling finite field arithmetic, leading to performance bottlenecks [51, 72]. Static analysis tools are often restricted in the range of vulnerabilities they can detect and tend to be specific to certain languages. CirC [71] compiler represents a step towards language-agnostic tooling, allowing various DSLs to compile into an Intermediate Representation, which can then be analyzed for potential vulnerabilities. Another promising direction is the creation of more secure DSLs, like Aleo [28], noname [8], and Noir [9], or eDSLs such as o1js [69], which incorporate strong typing and improved abstractions to prevent common issues.

Additionally, differential testing [65] against a reference implementation could be a viable method to identify computational issues. We anticipate that proper compilation of DSL to an IR could enable more general static analysis to uncover SNARK vulnerabilities in a DSL-agnostic way. Furthermore, applying hardening techniques to automatically patch circuits with missing range checks could enhance security. Complementary to existing approaches, compositional verification techniques could be applied to verify ZK circuits. Lastly, advanced fuzzing techniques that generate slightly incorrect witnesses might prove effective in detecting under-constrained vulnerabilities, while providing counterexamples. Finally, there is a growing demand for more advanced testing frameworks in the field. Such frameworks would greatly assist developers in creating their own unit and integration tests, which could be instrumental in identifying and rectifying many easily detectable vulnerabilities prior to deployment.

Integration Layer Defenses. Table 7 highlights a notable gap: there are currently no specific defenses developed for vulnerabilities in the integration layer of SNARK systems. However, traditional security methods like property-based fuzzing or formal verification could be applicable if there's a comprehensive specification detailing the expected system behavior and its interactions with SNARK components. A promising direction for future development is the creation of

[8] https://github.com/zksecurity/noname
[9] https://aztec.network/aztec-nr/

security frameworks designed to rigorously test the integration between client-side code (such as Solidity and JavaScript) and circuit code. We suggest exploring multi-layer testing and verification techniques targeting both these components simultaneously. An example could be a combined testing of Circom and Solidity applications. Additionally, having a detailed specification of the entire system, including how different components interact, can significantly enhance the effectiveness of manual code inspection.

Frontend/Backend Layers Defenses. The infrastructure layer, comprising the frontend and backend of SNARK systems, has largely been overlooked by security researchers. Ozdemir et al. [73] acknowledged that a flaw in a ZKP compiler could undermine the integrity of a ZKP, and took initial steps to mitigate such risks by partially verifying a key compiler pass in the CirC compiler [71]. SNARKProve [38] implements a fuzzing framework that can be configured with custom files (i.e., *ideal files*) to test specific cryptographic properties of the backends and frontends. In its initial version, it has been successful in detecting issues in the Setup phase of the backend. Moreover, practitioners have employed fuzzing tools like AFL to identify bugs in these layers. However, these tools are somewhat limited in scope, typically only capable of detecting crashes, potentially missing more subtle but critical vulnerabilities such as miscompilations.

As the field progresses with more complex optimizations in the compilation stage being enabled and the introduction of more advanced proof systems, it is likely that more vulnerabilities will emerge at the infrastructure layer of SNARKs. Consequently, there is a pressing need for the development of more sophisticated testing methodologies for the infrastructure layer. Overcoming the oracle problem and generating effective test cases are key challenges in this area. Insights from the extensive body of work on testing conventional compilers [23, 24, 62, 94] could be valuable in devising strategies for more effectively testing SNARK compilers.

Tools Applied on Audits. Our analysis of the 75 audit reports we reviewed revealed that SNARK-related tools were utilized in only 5 instances. Specifically, Picus was employed in 4 audits, while ZKAP, Ecne, and Circomspect were each used once. Interestingly, one audit incorporated differential testing against a reference implementation to identify computational issues in the circuits. Moreover, traditional security tools like AFL, Semgrep, and property testing were used in 10 audits, primarily for detecting bugs at the circuit and infrastructure levels. This highlights a significant need for enhanced tooling for both the frontend and backend layers of SNARK systems.

Multi-Provers as a Defense Mechanism. SNARK proofs are often used in safety-critical contexts, yet, with the current state of technology, guaranteeing that a complex SNARK system is bug-free and safe remains an elusive goal. The multi-prover design enhances safety by introducing redundancy at the proof system level. A multi-prover system utilizes multiple proof systems besides the primary SNARK, such as alternative SNARK proofs or trusted-execution environment (TEE) based proofs [21], and primary proofs are accepted only if the secondary ones agree with it. This design trades off liveness with more safety, as a dispute will potentially bring the system to a halt. Such systems require a dispute resolution mechanism to handle potential disagreements. In practice, some ZK-Rollup projects [85], arguably one of the most complex categories of SNARK systems today, are already experimenting with multi-provers.[10]

9 Issues in Proof Systems

While the focus of earlier sections was on implementation and design vulnerabilities across the four layers introduced in Section 3.1, vulnerabilities can also exist within the theoretical foundations of SNARKs' proof systems. This includes the formal descriptions and security proofs of protocols. Such vulnerabilities are less common but carry significant implications, potentially affecting all implementations of a given proof system. For example, the "Frozen Heart" vulnerability within the Plonk proof system (c.f. Appendix **??**) led to standard implementations of Plonk being compromised. The discovery of such flaws could severely impact the SNARK ecosystem, especially protocols that depend on these systems. Vulnerabilities in the proof system generally stem from two main sources: errors in the original proof system description, including missing or incorrect security proofs or incomplete descriptions that could lead developers to introduce significant vulnerabilities during implementation.

The primary method for evaluating proof systems currently is peer review, where researchers assess security proofs, supplemented by occasional manual audits. Beyond cryptanalysis, employing tools like EasyCrypt [10] and other computer-aided cryptographic proof software [8] offers a promising avenue for formalizing and verifying the security properties of zero-knowledge protocols, as exemplified by Firsov et al.'s work [39]. It is crucial for every proof system to be accompanied by exhaustive security proofs and to undergo rigorous review before production use to prevent potential bugs.

Universal Composability. The universal composability (UC) framework by Canetti [20] is widely considered as the "gold standard" for proving the security of cryptographic primitives. The reason for the popularity of the UC model is that it can guarantee strong security against adaptive adversaries and further allows for modular reusability of cryptographic primitives in greater, high-level protocol designs. Arguing UC security for a SNARK, or a SNARK-based protocol, is non-trivial, as SNARKs commonly use techniques that are not realizable in the UC model [49], and result in SNARKs that are not formally non-malleable. Recent work aims to close this gap by studying compilers that render common

[10] https://scroll.io/blog/scaling-security

SNARKs UC-secure by encrypting the witness and including it in the argument, which results in an overhead that renders the SNARK non-succinct due to the unbounded size of the witness [1, 2, 7, 58]. Most recent work proves that initial SNARK constructions [14, 66] that *do* utilize random oracles are UC-secure [25].

In practice, we observe that UC security is often not considered, and non-UC-secure protocols, like Groth16 [50], are applied without further ado. However, we are not aware of any protocol that claims UC-security and successively got exploited due to erroneous analysis, i.e., an underspecified ideal functionality or incorrectly claiming that the real-life model can be simulated in the ideal process model.

10 Discussion

In the following, we extract insights on the current state of SNARK security, highlight key findings, discuss their implications and make recommendations for future research.

1) Insight – Under-constrained bugs pose a significant threat to SNARK deployments. Under-constrained bugs emerge as the most prevalent vulnerability class within ZK circuits. Unlike typical vulnerabilities, their root causes span from straightforward programming errors to challenges inherent in SNARK DSLs, including the complexity of translating logic to constraints efficiently. As developers navigate these peculiarities, ensuring circuits are thoroughly constrained remains crucial for maintaining system integrity and security.
RQ: *What tools and methodologies can be developed to better identify and mitigate under-constrained bugs in ZK circuits? Which techniques are the most efficient in detecting such bugs?*

2) Insight – Soundness bugs affecting SNARK verifiers lead on average to high severity bugs. Particularly when soundness bugs are exploitable, typically due to under-constrained circuits, they can lead to significant security breaches. Their potential to compromise system security has led to suboptimal strategies, such as employing multi-provers and permissioned provers, alongside significant bug bounties, sometimes reaching up to $500k for a single vulnerability. [11] These measures underscore the critical nature of soundness in maintaining the trustworthiness of SNARK-based systems.
RQ: *What are effective strategies to detect and prevent soundness bugs in SNARKs? Can we formally verify the verifiers, as their scope is limited?*

3) Insight – Low-level circuit DSLs are easy-to-misuse leading to many vulnerabilities. Crafting efficient ZK circuits often necessitates using low-level ZK DSLs such as Circom and Gnark, reminiscent of early assembly and C programming, where common high-level programming features such as abstractions and basic types are absent. This complexity not only steepens the learning curve for developers but also increases the likelihood of introducing vulnerabilities into the circuits. The prevalent use of such DSLs highlights a regression to an era with bugs due to the absence of modern programming safeguards, underscoring an urgent need for more user-friendly DSLs that offer better abstractions and safety features, thereby mitigating the main shortcomings and reducing the vulnerability surface of SNARK applications.
RQ: *How can more user-friendly DSLs be designed to reduce the vulnerability surface of SNARK applications?*

4) Challenge – The added complexity of SNARKs present challenges for developers and auditors. The inherent complexity of SNARKs introduces significant challenges for both developers and auditors, compounded by the abstraction levels of ZK circuits and the low-level intricacies of most DSLs. Developers often find themselves navigating the task of integrating critical cryptographic operations—such as digital signatures, commitment schemes, and Merkle trees—within the ZK circuits. This combination of complex cryptographic code with the unique paradigm of SNARKs places a considerable burden on ensuring accuracy and security, particularly when such code underpins the most vital or privacy-sensitive parts of an application. Hence, not only must developers acquire the technical depth of cryptographic programming within these new frameworks, but auditors must also adapt their methodologies to effectively scrutinize these sophisticated systems.
RQ: *What educational and tooling resources can assist developers and auditors in navigating the complexities of using SNARKs? How useful can specifications (formal or informal) be towards detecting vulnerabilities in systems using ZKPs?*

5) Insight – Compiler and proof system implementation bugs can undermine major protocols. Infrastructure bugs in compilers and proof system implementations can critically undermine the security of major SNARKs. Even when circuits are verified and audited, vulnerabilities in the underlying infrastructure or proof systems can jeopardize the entire application, emphasizing the importance of holistic testing approaches for the infrastructure layer of the SNARKs.
RQ: *How can the reliability and security of compilers and proof system implementations for SNARKs be ensured? Which testing or verification techniques can be applied?*

6) Challenge – Preliminary security tools show promising results but also limitations. Recent developments in tools for securing ZK circuits show promise but face scalability issues and are often limited to specific DSLs or types of vulnerabilities. The complexity of SNARK systems makes manual code inspection necessary, pointing to a significant need for better security tools and educational resources. This combination of advanced tooling and increased knowledge is crucial for improving the security of the SNARK ecosystem.
RQ: *What improvements are needed in security tools to effectively scale and cover a broader range of vulnerabilities? What are the limitations of formal verification tools?*

7) Insight – Insecure Proof System Instantiation. The selection of cryptographic curves and fields for SNARK instantia-

[11] https://aztec.network/blog/aztec-connect-postmortem/

tion is a critical decision that can significantly impact system security. Insecure choices can expose the system to potential brute force attacks similarly to other cryptographic protocols, underscoring the necessity of careful selection to ensure the overall security and integrity of SNARK applications.
RQ: *What is the security of SNARKs instantiation given specific curves and fields? How many bits of security does each SNARK have when using specific configurations?*
8) Challenge – SNARK undetectable exploits. In privacy-preserving blockchains, such as ZCash, coins enter a privacy pool (a.k.a. shielded value pool) where they can be transferred without revealing the amount or recipient address. The transaction validation mechanism relies on SNARKs and a soundness bug can enable the adversary to print infinite coins inside the privacy pool. Note that such attacks can remain undetected as long as, at any point, the amount of coins that exited the pool is smaller or equal to coins that have entered the pool previously. A partial defense against these attacks is *turnstile enforcement*, adding a consensus check to reject blocks violating pool balance, ensuring exits don't exceed entries. This highlights the need for privacy-focused SNARK systems with exploit detection and prevention mechanisms.
RQ: *How can we develop detection mechanisms or heuristics for exploits in privacy-preserving SNARK systems?*

11 Related Work

SNARK Vulnerabilities. This paper introduces a four-layer system model, defines threat models along with a detailed taxonomy of vulnerabilities and root causes across each layer in SNARK systems. Prior works have identified specific vulnerabilities within SNARKs; for example, Wen et al. [91] highlighted common vulnerabilities in Circom circuits, while others [38, 53, 74, 87, 90] have focused mainly on under-constrained vulnerabilities and proposed countermeasures. Ozdemir et al. [73] shed light on potential issues during the SNARK compilation phase. For security tools related to SNARK vulnerabilities, Table 7 offers a comprehensive overview. Additionally, the community maintains a bug tracker dedicated to ZKP-related vulnerabilities.[12] Our research complements these efforts by systematizing knowledge from the examination of 141 vulnerabilities and enriching the understanding of SNARK security.

Security of Integrity-Preserving Technologies. Integrity-preserving computation encompasses a variety of technologies, each presenting unique security challenges. Similar to our work, Cerdeira et al. [21] examined 124 CVEs within TEE-based systems and proposed a vulnerability taxonomy for TrustZone-assisted TEE Systems. Blockchain smart contracts represent another avenue for integrity-preserving computation; Praitheeshan et al. [80] identified common software and Ethereum smart contract vulnerabilities, focusing on issues prevalent at the smart contract layer. Homoliak et al. [52] introduced a multi-layered security model, systematically addressing vulnerabilities, threats, and countermeasures for blockchains. Atzei et al. [5] explored Ethereum's security vulnerabilities, offering a classification of common programming pitfalls. Zhou et al. [97] developed a five-layer model and a comprehensive taxonomy of threat models to analyze and compare incidents in DeFi. Further, Chaliasos et al. [22] used the dataset from [97] to evaluate state-of-the-practice smart contract security tools against real-world vulnerabilities. In a similar way, future work could leverage our findings to assess the efficacy of emerging security tools for SNARKs against the vulnerabilities detailed in our dataset.

[12] https://github.com/0xPARC/zk-bug-tracker

12 Conclusions

In this work, we present comprehensive system and threat models for SNARK systems' security, a detailed study, and a taxonomy of 141 vulnerabilities, demonstrating that security breaches can affect every layer of systems employing SNARKs, jeopardizing completeness, soundness, and zero-knowledge properties. Our work reveals the intricate and unique security challenges inherent to SNARKs, indicating that defense mechanisms focusing on SNARK security have significant limitations. By highlighting key insights and potential advancements in security practices, we underscore the urgency for continued research and enhanced defenses within the SNARK ecosystem. As SNARKs become increasingly pivotal in cryptographic applications, our study emphasizes the necessity for progressive and fortified security measures to ensure the robustness of these systems.

Acknowledgments

We would like to thank our shepherd and the anonymous USENIX reviewers. We also thank Shankara Pailoor, Alex Ozdemir, Jean-Philippe Aumasson, Oana Ciobotaru, and Justin Traglia for their insightful feedback. Finally, we thank the Ethereum Foundation and 0xPARC for supporting this work.

References

[1] Behzad Abdolmaleki, Noemi Glaeser, Sebastian Ramacher, and Daniel Slamanig. Universally composable nizks: Circuit-succinct, non-malleable and crs-updatable. *IACR Cryptol. ePrint Arch.*, 2023, 2023.

[2] Behzad Abdolmaleki, Sebastian Ramacher, and Daniel Slamanig. Lift-and-shift: obtaining simulation extractable subversion and updatable snarks generically. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[3] Nada Amin, John Burnham, François Garillot, Rosario Gennaro, Daniel Rogozin, Cameron Wong, et al. Lurk: Lambda, the ultimate recursive knowledge. *Cryptology ePrint Archive*, 2023.

[4] Arasu Arun, Srinath Setty, and Justin Thaler. Jolt: Snarks for virtual machines via lookups. *Cryptology ePrint Archive*, 2023.

[5] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In *Principles of Security and Trust: 6th International Conference, POST 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 6*. Springer, 2017.

[6] Aztec. Aztec network, 2023. https://docs.aztec.network/.

[7] Karim Baghery and Mahdi Sedaghat. Tiramisu: black-box simulation extractable nizks in the updatable crs model. In *International Conference on Cryptology and Network Security*. Springer, 2021.

[8] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. Sok: Computer-aided cryptography. In *2021 IEEE symposium on security and privacy (SP)*. IEEE, 2021.

[9] Earl T Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *IEEE transactions on software engineering*, 41(5), 2014.

[10] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. *International School on Foundations of Security Analysis and Design*, 2012.

[11] Marta Bellés-Muñoz, Miguel Isabel, Jose Luis Muñoz-Tapia, Albert Rubio, and Jordi Baylina. Circom: A circuit description language for building zero-knowledge applications. *IEEE Transactions on Dependable and Secure Computing*, 2022.

[12] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Fast reed-solomon interactive oracle proofs of proximity. In *45th international colloquium on automata, languages, and programming*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[13] Eli Ben-Sasson, Iddo Bentov, Yinon Horesh, and Michael Riabzev. Scalable zero knowledge with no trusted setup. In *Advances in Cryptology–CRYPTO 2019: 39th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 18–22, 2019, Proceedings, Part III 39*. Springer, 2019.

[14] Eli Ben-Sasson, Alessandro Chiesa, and Nicholas Spooner. Interactive oracle proofs. In *Theory of Cryptography: 14th International Conference, TCC*. Springer, 2016.

[15] Joseph Bonneau, Izaak Meckler, Vanishree Rao, and Evan Shapiro. Mina: Decentralized cryptocurrency at scale. *Whitepaper*, 2020.

[16] Gautam Botrel, Thomas Piellard, Youssef El Housni, Ivo Kubjas, and Arya Tabaie. Consensys/gnark: v0.8.0, 2023.

[17] Sean Bowe, Alessandro Chiesa, Matthew Green, Ian Miers, Pratyush Mishra, and Howard Wu. Zexe: Enabling decentralized private computation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[18] Jeremy Bruestle, Paul Gafni, and the RISC Zero Team. Risc zero zkvm: Scalable, transparent arguments of risc-v integrity. https://dev.risczero.com/proof-system-in-detail.pdf, 2023.

[19] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Greg Maxwell. Bulletproofs: Short proofs for confidential transactions and more. In *2018 IEEE symposium on security and privacy (SP)*. IEEE, 2018.

[20] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001.

[21] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020.

[22] Stefanos Chaliasos, Marcos Antonios Charalambous, Liyi Zhou, Rafaila Galanopoulou, Arthur Gervais, Dimitris Mitropoulos, and Benjamin Livshits. Smart contract and defi security tools: Do they meet the needs of practitioners? In *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, 2023.

[23] Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, and Dimitris Mitropoulos. Finding typing compiler bugs. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, 2022.

[24] Yang Chen, Alex Groce, Chaoqiang Zhang, Weng-Keen Wong, Xiaoli Fern, Eric Eide, and John Regehr. Taming compiler fuzzers. In *Proceedings of the 34th ACM*

SIGPLAN conference on Programming language design and implementation, 2013.

[25] Alessandro Chiesa and Giacomo Fenzi. zksnarks in the rom with unconditional uc-security. *Cryptology ePrint Archive*, 2024.

[26] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Noah Vesely, and Nicholas Ward. Marlin: Preprocessing zksnarks with universal and updatable srs. In *Advances in Cryptology–EUROCRYPT: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020.

[27] Alessandro Chiesa, Ryan Lehmkuhl, Pratyush Mishra, and Yinuo Zhang. Eos: Efficient private delegation of zksnark provers. In *USENIX Security Symposium. USENIX Association*, 2023.

[28] Collin Chin, Howard Wu, Raymond Chu, Alessandro Coglio, Eric McCarthy, and Eric Smith. Leo: A programming language for formally verified, zero-knowledge applications. *Cryptology ePrint Archive*, 2021.

[29] Alessandro Coglio, Eric McCarthy, Eric Smith, Collin Chin, Pranav Gaddamadugu, and Michel Dellepere. Compositional formal verification of zero-knowledge circuits. *Cryptology ePrint Archive*, 2023.

[30] Alessandro Coglio, Eric McCarthy, and Eric W Smith. Formal verification of zero-knowledge circuits. *arXiv preprint arXiv:2311.08858*, 2023.

[31] World Wide Web Consortium et al. Verifiable credentials data model 1.0: expressing verifiable information on the web. *Specification*, 2019.

[32] Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. Weak fiat-shamir attacks on modern proof systems. Cryptology ePrint Archive, Paper 2023/691, 2023. https://eprint.iacr.org/2023/691.

[33] Jacob Eberhardt and Stefan Tai. Zokrates-scalable privacy-preserving off-chain computations. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*. IEEE, 2018.

[34] Youssef El Housni and Aurore Guillevic. Families of snark-friendly 2-chains of elliptic curves. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2022.

[35] Jens Ernstberger, Stefanos Chaliasos, George Kadianakis, Sebastian Steinhorst, Philipp Jovanovic, Arthur Gervais, Benjamin Livshits, and Michele Orrù. zk-bench: A toolset for comparative evaluation and performance benchmarking of snarks. *Cryptology ePrint Archive*, 2023.

[36] Jens Ernstberger, Stefanos Chaliasos, Liyi Zhou, Philipp Jovanovic, and Arthur Gervais. Do you need a zero knowledge proof? Cryptology ePrint Archive, Paper 2024/050, 2024. https://eprint.iacr.org/2024/050.

[37] Privacy & Scaling Explorations. halo2 community edition, 2023.

[38] Yongming Fan, Yuquan Xu, and Christina Garman. Snarkprobe: An automated security analysis framework for zksnark implementations. *International Conference on Applied Cryptography and Network Security*, 2024.

[39] Denis Firsov and Dominique Unruh. Zero-knowledge in easycrypt. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)*. IEEE, 2023.

[40] Ariel Gabizon, Zachary J Williamson, and Oana Ciobotaru. Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge. *Cryptology ePrint Archive*, 2019.

[41] Nicolas Gailly, Mary Maller, and Anca Nitulescu. Snarkpack: Practical snark aggregation. In *International Conference on Financial Cryptography and Data Security*. Springer, 2022.

[42] Chaya Ganesh, Anca Nitulescu, and Eduardo Soria-Vazquez. Rinocchio: Snarks for ring arithmetic. *Journal of Cryptology*, 36(4), 2023.

[43] Sanjam Garg, Aarushi Goel, Abhishek Jain, Guru-Vamsi Policharla, and Sruthi Sekar. zksaas: Zero-knowledge snarks as a service. *Cryptology ePrint Archive*, 2023.

[44] Rosario Gennaro, Craig Gentry, Bryan Parno, and Mariana Raykova. Quadratic span programs and succinct nizks without pcps. In *Advances in Cryptology–EUROCRYPT: 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2013.

[45] Geometry. Groth16 malleability. https://geometry.xyz/notebook/groth16-malleability, 2023.

[46] Lior Goldberg, Shahar Papini, and Michael Riabzev. Cairo–a turing-complete stark-friendly cpu architecture. *Cryptology ePrint Archive*, 2021.

[47] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC ’85. Association for Computing Machinery, 1985.

[48] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof systems. *Siam J. Comput*, 18(1), 1989.

[49] Jens Groth. Simulation-sound nizk proofs for a practical language and constant size group signatures. In *Advances in Cryptology–ASIACRYPT: 12th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2006.

[50] Jens Groth. On the size of pairing-based non-interactive arguments. In *Advances in Cryptology–EUROCRYPT: 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2016.

[51] Thomas Hader, Daniela Kaufmann, and Laura Kovács. Smt solving over finite field arithmetic. *arXiv preprint arXiv:2305.00028*, 2023.

[52] Ivan Homoliak, Sarad Venugopalan, Daniël Reijsbergen, Qingze Hum, Richard Schumi, and Pawel Szalachowski. The security reference architecture for blockchains: Toward a standardized model for studying vulnerabilities, threats, and defenses. *IEEE Communications Surveys & Tutorials*, 23(1), 2020.

[53] Miguel Isabel, Clara Rodríguez-Núñez, and Albert Rubio. Scalable verification of zero-knowledge protocols. In *2024 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2024.

[54] Daniel Kang, Tatsunori Hashimoto, Ion Stoica, and Yi Sun. Zk-img: Attested images via zero-knowledge proofs to fight disinformation, 2022.

[55] Aniket Kate, Gregory M Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. In *Advances in Cryptology-ASIACRYPT: 16th International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2010.

[56] Hankyung Ko, Ingeun Lee, Seunghwa Lee, Jihye Kim, and Hyunok Oh. Efficient verifiable image redacting based on zk-snarks. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, ASIA CCS '21. Association for Computing Machinery, 2021.

[57] Ahmed Kosba, Charalampos Papamanthou, and Elaine Shi. xjsnark: A framework for efficient verifiable computation. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[58] Ahmed Kosba, Zhichao Zhao, Andrew Miller, Yi Qian, Hubert Chan, Charalampos Papamanthou, Rafael Pass, Elaine Shi, et al. C0c0: A framework for building composable zero-knowledge proofs. *Cryptology ePrint Archive*, 2015.

[59] Abhiram Kothapalli, Srinath Setty, and Ioanna Tzialla. Nova: Recursive zero-knowledge arguments from folding schemes. In *Annual International Cryptology Conference*. Springer, 2022.

[60] Matter Labs. zksync era, 2023. https://era.zksync.io/.

[61] Jan Lauinger, Jens Ernstberger, Andreas Finkenzeller, and Sebastian Steinhorst. Janus: Fast privacy-preserving data provenance for tls 1.3. *Cryptology ePrint Archive*, 2023.

[62] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. *ACM Sigplan Notices*, 49(6), 2014.

[63] Junrui Liu, Ian Kretz, Hanzhi Liu, Bryan Tan, Jonathan Wang, Yi Sun, Luke Pearson, Anders Miltner, Işıl Dillig, and Yu Feng. Certifying zero-knowledge circuits with refinement types. *arXiv preprint arXiv:2304.07648*, 2023.

[64] Tianyi Liu, Tiancheng Xie, Jiaheng Zhang, Dawn Song, and Yupeng Zhang. Pianist: Scalable zkrollups via fully distributed zero-knowledge proofs. *Cryptology ePrint Archive*, 2023.

[65] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 10(1), 1998.

[66] Silvio Micali. Computationally sound proofs. *SIAM Journal on Computing*, 30(4), 2000.

[67] Ian Miers, Christina Garman, Matthew Green, and Aviel D Rubin. Zerocoin: Anonymous distributed e-cash from bitcoin. In *2013 IEEE Symposium on Security and Privacy*. IEEE, 2013.

[68] Valeria Nikolaenko, Sam Ragsdale, Joseph Bonneau, and Dan Boneh. Powers-of-tau to the people: Decentralizing setup ceremonies. *Cryptology ePrint Archive*, 2022.

[69] o1 labs. Typescript framework for zk-snarks and zkapps. https://github.com/o1-labs/o1js, 2021.

[70] Alex Ozdemir and Dan Boneh. Experimenting with collaborative {zk-SNARKs}:{Zero-Knowledge} proofs for distributed secrets. In *31st USENIX Security Symposium (USENIX Security 22)*, 2022.

[71] Alex Ozdemir, Fraser Brown, and Riad S Wahby. Circ: Compiler infrastructure for proof systems, software verification, and more. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022.

[72] Alex Ozdemir, Gereon Kremer, Cesare Tinelli, and Clark Barrett. Satisfiability modulo finite fields. *Cryptology ePrint Archive*, 2023.

[73] Alex Ozdemir, Riad S Wahby, Fraser Brown, and Clark Barrett. Bounded verification for finite-field-blasting (in a compiler for zero knowledge proofs). *Cryptology ePrint Archive*, 2023.

[74] Shankara Pailoor, Yanju Chen, Franklyn Wang, Clara Rodríguez, Jacob Van Geffen, Jason Morton, Michael Chu, Brian Gu, Yu Feng, and Işıl Dillig. Automated detection of under-constrained circuits in zero-knowledge proofs. *Proceedings of the ACM on Programming Languages*, 7(PLDI), 2023.

[75] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. *Communications of the ACM*, 59(2), 2016.

[76] Rafael Pass, Wei-Lung Dustin Tseng, and Douglas Wikström. On the composition of public-coin zero-knowledge protocols. *SIAM Journal on Computing*, 40(6), 2011.

[77] Alexey Pertsev, Roman Semenov, and Roman Storm. Tornado cash privacy solution version 1.4. *Tornado cash privacy solution version*, 2019.

[78] Polygon. Miden vm. https://polygon.technology/polygon-miden, 2023.

[79] Polygon. Polygon zkevm, 2023. https://polygon.technology/polygon-zkevm.

[80] Purathani Praitheeshan, Lei Pan, Jiangshan Yu, Joseph Liu, and Robin Doss. Security analysis methods on ethereum smart contract vulnerabilities: a survey. *arXiv preprint arXiv:1908.08605*, 2019.

[81] Mir Protocol. Plonky2, 2023.

[82] Protocol Labs. Filecoin: A decentralized storage network. https://filecoin.io/filecoin.pdf, 2023.

[83] Deevashwer Rathee, Guru Vamsi Policharla, Tiancheng Xie, Ryan Cottone, and Dawn Song. Zebra: Anonymous credentials with practical on-chain verification and applications to kyc in defi. *Cryptology ePrint Archive*, 2022.

[84] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE symposium on security and privacy*. IEEE, 2014.

[85] Scroll. Scroll zkevm, 2023. https://scroll.io/.

[86] Srinath Setty, Justin Thaler, and Riad Wahby. Customizable constraint systems for succinct arguments. *Cryptology ePrint Archive*, 2023.

[87] Fatemeh Heidari Soureshjani, Mathias Hall-Andersen, MohammadMahdi Jahanara, Jeffrey Kam, Jan Gorzny, and Mohsen Ahmadvand. Automated analysis of halo2 circuits. *Cryptology ePrint Archive*, 2023.

[88] Justin Thaler et al. Proofs, arguments, and zero-knowledge. *Foundations and Trends® in Privacy and Security*, 4(2–4), 2022.

[89] Trail of Bits. circomspect: A static analyzer for circom 2 circuits. https://github.com/trailofbits/circomspect, 2023. Accessed: 2024-01-31.

[90] Franklyn Wang. Ecne. https://github.com/franklynwang/EcneProject, 2022.

[91] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. *Cryptology ePrint Archive*, 2023.

[92] Tiancheng Xie, Jiaheng Zhang, Zerui Cheng, Fan Zhang, Yupeng Zhang, Yongzheng Jia, Dan Boneh, and Dawn Song. zkbridge: Trustless cross-chain bridges made practical. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, 2022.

[93] Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. In *Annual International Cryptology Conference*. Springer, 2022.

[94] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[95] ZCash. halo2, 2023.

[96] Fan Zhang, Deepak Maram, Harjasleen Malvai, Steven Goldfeder, and Ari Juels. Deco: Liberating web data using decentralized oracles for tls. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020.

[97] Liyi Zhou, Xihan Xiong, Jens Ernstberger, Stefanos Chaliasos, Zhipeng Wang, Ye Wang, Kaihua Qin, Roger Wattenhofer, Dawn Song, and Arthur Gervais. Sok: Decentralized finance (defi) attacks. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023.