ZKGuard: A Zero-Knowledge Policy Engine for On-Chain Transactions

Santiago Galiñanes Arienti
University of Coimbra, CISUC/LASI
Portugal
sgalinanes4@gmail.com

Nuno Laranjeiro
University of Coimbra, CISUC/LASI, DEI
Portugal
cnl@dei.uc.pt

***Abstract*—Blockchain wallets enforce transaction authorization through off-chain policy engines, but these systems generally rely on trusted operators and do not bind policy checks cryptographically to on-chain execution. This paper proposes ZKGuard, a software tool for runtime policy enforcement in which the wallet executes an action only if a zero-knowledge proof confirms that the action satisfies a policy belonging to a previously committed set. The policy set is committed on-chain via a Merkle root, and the prover demonstrates both policy membership and compliance without revealing the policy contents. We implement ZKGuard across three proving stacks that cover distinct points in the design space: RISC Zero, Noir with UltraHonk, and Gnark with Groth16. An experimental evaluation across seven policy scenarios and three hardware profiles (2 cores/8 GiB, 4 cores/16 GiB, 8 cores/32 GiB), with 30 independent runs per configuration, together with a policy-set scaling study up to 4096 committed policies and on-chain cost measurements for the ZKGuard Safe module on the RISC Zero backend, shows that circuit-based stacks achieve sub-minute proving on well-provisioned hardware, Groth16 delivers the fastest verification, and Noir is the only stack viable on constrained hardware. RISC Zero offers the most general programming model at substantially higher proving cost. The main overhead introduced by ZKGuard is the proving step; in return, even a party controlling a valid signing key cannot authorize actions that violate the committed policy.**

***Index Terms*—zero-knowledge proofs, blockchain security, policy enforcement, smart-contract wallets, zkSNARK**

I. Introduction

Smart contracts on blockchains execute as deterministic, replicated state transitions [1], [2]. Once a transaction is finalized, it is irreversible, which makes the security of every post-deployment action a critical concern.

Most existing security practices, including audits, static analysis, and formal verification, target the pre-deployment phase, reducing the likelihood of vulnerabilities in the contract code itself. However, even a contract that has been thoroughly analyzed before deployment remains exposed to risks that materialize only at execution time, such as the compromise of a signing key or the submission of a malicious transaction by an authorized party. In production environments, this gap has been addressed in two main ways. On one hand, wallet and custody providers such as Fireblocks [3], Fordefi [4], and Turnkey [5] enforce operational constraints such as destination allowlists, multi-party approvals, and transfer thresholds through policy engines that operate off-chain. On the other hand, transaction-screening services such as Blockaid [6], OpenZeppelin Defender [7], and Forta [8] provide monitoring and anomaly detection over submitted transactions. Both categories are useful in practice, but they share a common limitation: they generally require trusting an off-chain operator, and they do not provide cryptographic on-chain enforcement that binds security checks to execution. In other words, a policy violation detected off-chain does not, by itself, prevent the corresponding on-chain action from being carried out.

This paper proposes ZKGuard [9], a software tool for runtime policy enforcement in which policy evaluation runs off-chain, but on-chain execution is conditioned on a zero-knowledge proof that the evaluation was executed correctly for the claimed action. Concretely, the smart-contract wallet verifies a Zero-Knowledge Succinct Non-Interactive Argument of Knowledge (zkSNARK) [10], [11] over two statements: (i) the selected policy belongs to a previously committed policy set, and (ii) the user action complies with that policy. The prover commits to a policy set, proves that one policy belongs to that committed set, and proves that the action to execute satisfies it, while revealing only the public commitments and proof validity. In our implementation, this policy-binding step is instantiated with a Merkle commitment over the policy set, together with backend-specific proof systems.

ZKGuard can be seen as a policy-enforcement construction built from existing zero-knowledge proving systems, rather than as a new one. Service providers and teams running business-critical contracts are usually more sensitive to safety failures than to moderate prover overhead, so ZKGuard moves heavy analysis off-chain while keeping on-chain verification succinct. The approach also keeps policy details private, since only proof validity is revealed on-chain, reducing the attack surface compared with existing on-chain guards that expose policy logic. ZKGuard therefore provides *off-chain policy checking* (policy logic is not limited by on-chain execution cost), *mandatory on-chain enforcement* (a wallet executes an action only if a valid proof is verified on-chain), and *policy privacy* (the verifier learns that the action satisfies a committed policy but not its contents).

We evaluate ZKGuard across three proving stacks that represent distinct points in the design space: RISC Zero [12], a general-purpose zkVM with a Rust programming model and

a STARK-to-Groth16 wrapper; Noir [13] with the UltraHonk backend, a domain-specific language (DSL) that avoids per-circuit trusted setup; and Gnark [14] with Groth16, a low-level circuit framework in Go that produces the most compact proofs at the cost of trusted-setup infrastructure. Each stack is benchmarked across seven policy scenarios and three hardware profiles (2 cores / 8 GiB, 4 cores / 16 GiB, and 8 cores / 32 GiB), using 30 independent runs per configuration.

Specifically, this paper aims to answer the following research questions:

- **RQ1:** How can off-chain policy evaluation be bound to on-chain wallet execution so that only policy-compliant actions are executable?
- **RQ2:** What security guarantees does ZKGuard provide regarding authorization soundness and policy privacy?
- **RQ3:** What trade-offs do different proving stacks introduce for ZKGuard in terms of proving latency, verification cost, memory usage, setup overhead, and developer practicality?

Our results show that circuit-based stacks (Noir and Gnark) achieve substantially lower proving latency than the zkVM-based stack (RISC Zero), which in turn offers a more general programming model at higher proving cost. Among the circuit stacks, Groth16-based proving yields the most efficient on-chain verification and the most compact proofs, while UltraHonk avoids the need for a per-circuit trusted setup. Noir exhibits the lowest memory footprint and is the only stack that completes on the most constrained hardware profile. The main cost introduced by ZKGuard, compared with an unguarded wallet, is the proving step; in return, even a party controlling a valid signing key cannot authorize actions that violate the committed policy.

The main contributions of this work are as follows:

- The design of ZKGuard as a runtime authorization layer for on-chain actions that enforces policy compliance via zkSNARK proofs while keeping policy contents private.
- The implementation of ZKGuard across three proving paradigms and its integration with the smart-contract wallet execution flow. The full tool, including the implementations under the three proving systems, an end-to-end smart-contract wallet integration, the raw measurement data, and the reproduction scripts, is permanently available [9].
- An experimental evaluation across seven policy scenarios and three hardware configurations, a policy-set scaling study up to 4096 committed policies, and on-chain cost measurements for the ZKGuard Safe module, identifying trade-offs among latency, memory, verification cost, setup overhead, and deployment model.

This paper is organized as follows. Section II presents the background and related work. Section III describes the threat model and design goals. Section IV details the ZKGuard architecture. Section V presents the experimental evaluation. Section VI discusses threats to validity, and Section VII concludes the paper.

II. Background and Related Work

Regarding **background concepts**, and specifically *smart-contract wallets*, a smart-contract wallet on Ethereum generalizes an externally owned account (EOA), which authorizes transactions with a single signing key, by replacing the fixed signing scheme with programmable validation logic that can enforce conditions such as multi-signature thresholds, spending limits, or time locks before execution [1], [2]. Account abstraction proposals such as ERC-4337 [15] standardize this pattern through a `UserOperation` mempool and an `EntryPoint` contract that delegates validation to each wallet's verification function. ZKGuard builds on this separation by requiring that the wallet's validation step include a zero-knowledge proof of policy compliance.

Concerning *zero-knowledge proofs and zkSNARKs*, a zero-knowledge proof (ZKP) lets a prover convince a verifier that a statement is true without revealing the witness, satisfying completeness, soundness, and zero-knowledge [16], [17]. Practical deployments use zkSNARKs, which are non-interactive, succinct, and computationally sound [18]. Representative families (Groth16, Plonk-derived systems such as UltraHonk, and zkVM-oriented constructions) trade off setup assumptions, proof size, and prover/verifier cost [11], [12], [19].

ZKGuard also relies on *cryptographic commitments and the commit-and-prove pattern*. A commitment binds a party to a value and, when hiding, conceals it; a Merkle tree is a hash-based commitment whose root commits to all leaves, with logarithmic-length membership proofs. In the commit-and-prove pattern, a public commitment fixes the data the proof may reason about, and the proof shows the data is consistent with it. In ZKGuard, the policy set is committed via a Merkle root stored on-chain, and the prover shows that a specific policy belongs to that set and that the user action satisfies it.

Finally, two *implementation models* for zero-knowledge applications are relevant. Circuit-DSL stacks such as Noir [13] and Circom [20] compile constrained programs into algebraic representations for a proving backend, whereas zkVMs fix a VM/ISA (often RISC-V) and prove a program's execution trace, improving developer familiarity at the cost of higher execution overhead [21]. The three stacks we evaluate span this space. Gnark uses Groth16 [19], with constant-size proofs and fast pairing-based verification but a per-circuit trusted setup; Noir uses UltraHonk from Barretenberg [22], avoiding per-circuit setup through a universal reference string at the cost of larger proofs; and RISC Zero [12] proves a RISC-V trace and wraps the resulting STARK in a Groth16 proof for succinct on-chain verification, at higher proving cost. Section V describes each stack in detail.

Regarding **related work**, institutional wallet infrastructure commonly uses policy-based authorization controls. Fireblocks, Fordefi, and Turnkey enforce operational constraints such as destination allowlists, multi-party approvals, and transfer thresholds through off-chain policy engines [3]–[5].

Blockaid and related providers additionally offer transaction screening and anomaly detection [6]. These systems keep policy details private and are widely deployed, but they rely on trusted off-chain operators and do not bind policy checks cryptographically to on-chain execution.

Previous research has explored direct on-chain monitoring and detection. ContractGuard embeds an intrusion-detection system in the contract itself [23], and BChainGuard trains ML classifiers off-chain and embeds only the decision function on-chain [24]. A separate line applies runtime verification to deployed contracts. ContractLarva instruments Solidity contracts with checks derived from formal specifications [25], Sereum extends the EVM client to detect and revert re-entrancy attacks [26], and TxSpector analyzes historical transactions for attack detection [27]. These approaches enforce properties after deployment but do not provide the proof-binding model of ZKGuard, and the on-chain guards among them expose their logic publicly and add gas overhead.

Recent work on private policy verification, such as zkAt/zkAt+, is conceptually the closest academic neighbor to ZKGuard, since it also uses zero-knowledge proofs for blockchain transaction authorization while hiding policy internals [28]. The distinction between the two is primarily architectural. zkAt encodes the authentication policy directly inside a zero-knowledge circuit, so that the verifier's public key reveals nothing about the underlying policy; zkAt+ extends this with oblivious policy updates through recursive proofs. ZKGuard takes a different approach by separating policy storage from policy proving: the policy set is first committed on-chain via a Merkle root, and the prover then demonstrates that a specific policy belongs to that committed set and that the user action satisfies it. This Merkle-based binding is what guarantees that the policy used during proving is the same one the wallet owner previously authorized. Table I contrasts ZKGuard with representative prior systems.

TABLE I
COMPARISON WITH REPRESENTATIVE RELATED SYSTEMS.

System	Trust basis	Checks	On-chain enforced	Policy	Proving System
Fireblocks	MPC + provider policy	Off-chain	No	Private	N/A
Turnkey	TEE / enclave-based	Off-chain	No	Private	N/A
ContractGuard	Public guard logic	On-chain	Yes	Public	N/A
BChainGuard	Off-chain ML + on-chain decision	Hybrid	Partial	Public	N/A
zkAt/zkAt+	ZK proofs	On-chain	Yes	Private	Modified
ZKGuard	ZK proofs + committed policy	On-chain	Yes	Private	Existing

These differences have practical consequences for an operator. Off-chain policy engines (Fireblocks, Fordefi, Turnkey) rely on a trusted operator, so a violation they detect does not by itself stop the on-chain action; in ZKGuard the action cannot execute without a valid on-chain proof, even when the attacker holds a valid signing key. On-chain guards such as ContractGuard expose their logic publicly and add on-chain cost per rule, whereas ZKGuard keeps the policy private and evaluates it off-chain. zkAt/zkAt+ binds one authentication policy to a verification key, whereas ZKGuard commits an entire policy set under a single on-chain root with explicit membership proofs, which the operator can update. Integrating ZKGuard requires only that the wallet call the verifier and compare commitments before executing.

III. System Model and Threat Model

This section presents the system model and threat environment that guide the design of ZKGuard.

In our **system model**, a deployed smart contract maintains committed state that includes a policy root and context (group and allowlist set) commitments. The contract receives a user action U and a proof π. Before any action is executed, the contract verifies π over a public statement and a private witness. The public statement is defined as

$$x = (C_U, C_p, C_G, C_A),$$

where C_U, C_p, C_G, and C_A are commitments to the action to be executed, the predefined policy set, the group set, and the allowlist set, respectively. The prover witness is defined as:

$$w = (p, \theta_p, G, A, U).$$

where p is a policy and θ_p is the evaluation or opening proof showing that p is the opening for C_p. G and A are the group set and allowlist set composed of addresses which commit to C_G and C_A respectively and U is the aforementioned user action.

The verifier accepts only if the relation $(x, w) \in \mathcal{R}$, where $\mathcal{R}$ encodes policy inclusion and policy compliance checks for the claimed action. The submitted user action U includes any execution parameters required by the concrete instantiation, including a freshness field used for replay protection. A detailed definition of the implemented User Action as well as the relation $\mathcal{R}$ can be seen in Section IV.

After verifying the proof, the system must check that the policy and context commitments in the public input x match the commitments previously stored on-chain. For the user action, the contract recomputes its commitment from the submitted action and verifies equality with the corresponding commitment in x. Finally, to prevent replay attacks, the contract performs an additional on-chain freshness check over the submitted action (e.g., by verifying that its nonce matches the current nonce stored in the smart contract).

Our **security objective** is authorization soundness at runtime: an adversary should not be able to execute an action that violates the committed policy while still passing on-chain verification. As discussed in the introduction, existing off-chain policy engines lack this guarantee because a policy violation detected off-chain does not, by itself, prevent the corresponding on-chain action.

Regarding **adversary capabilities**, we consider an adversary that can: (i) submit arbitrary transactions directly to the wallet entry point, bypassing any off-chain service; (ii) observe mempool and on-chain state; and (iii) adaptively choose payloads, signatures, and timing.

Our **cryptographic and trust assumptions** are the following:

- *Proof-system soundness and knowledge extraction.* For each backend, security inherits from the deployed proving system (RISC Zero, UltraHonk, Groth16) [12], [19], [22].
- *Non-interactive security model.* We rely on standard Fiat-Shamir-style assumptions in the random-oracle setting used by modern non-interactive proof systems [18].
- *Collision-resistant hashing.* All commitment and Merkle-tree constructions assume collision resistance of the underlying hash function. The concrete instantiation is backend-specific and described in Section IV.
- *Signature security.* Authorization signatures cannot be forged under the deployed signature scheme and key parameters.
- *Trusted setup where required.* For Groth16 deployments, a circuit-specific trusted setup is assumed to be generated correctly and the toxic waste is assumed to have been destroyed [19].
- *On-chain verifier correctness.* The verifier contract and wallet execution gate are correctly implemented and deployed.

We state the security arguments more explicitly. Let $S = (C_p, C_G, C_A, n)$ be the wallet's committed state. Let $\mathsf{Accept}_S(U, \pi)$ denote that the protected wallet entry point accepts proof π, commitment equality, and freshness for action U. Let $\mathsf{Compliant}_S(U)$ mean that there exist p, θ_p, G, A such that $(x, w) \in \mathcal{R}$, where $x = (C_U, C_p, C_G, C_A)$ and $w = (p, \theta_p, G, A, U)$.

Authorization soundness requires that, for every PPT adversary $\mathcal{A}$

$$\Pr\left[\begin{array}{c}\mathsf{Accept}_S(U,\pi)=1 \\ \wedge\ \mathsf{Compliant}_S(U)=0\end{array}\;\middle|\;(U,\pi)\leftarrow\mathcal{A}(1^\lambda,S)\right]\leq \mathsf{negl}(\lambda).$$

If such an event occurs, proof-system soundness implies a witness satisfying $\mathcal{R}$, while collision resistance binds that witness and action to the public commitments checked on-chain. Thus any accepted action is compliant unless a stated assumption fails.

Policy privacy requires that, for any two valid witnesses w_0, w_1 for the same public statement x, no PPT distinguisher $\mathcal{D}$ can determine which witness was used:

$$\left|\Pr\left[b'=b\;\middle|\;\begin{array}{l}b\leftarrow\{0,1\},\\ \pi\leftarrow\mathsf{Prove}(x,w_b),\\ b'\leftarrow\mathcal{D}(1^\lambda,x,\pi)\end{array}\right]-\frac{1}{2}\right|\leq\mathsf{negl}(\lambda).$$

Thus observers learn only the submitted action, public commitments, and proof validity, not the selected policy, Merkle path, or policy artifacts.

Several aspects fall **out of scope**. Full compromise of policy administrators or the verifier contract itself represents a different class of threat (privileged insider and smart-contract vulnerability, respectively) that falls outside the runtime authorization boundary targeted by ZKGuard. Chain-level failures such as consensus breaks and liveness failures are also excluded. Transaction-ordering effects, including front-running, do not affect proof soundness or policy enforcement for unauthorized actions, although they may cause authorized transactions to fail due to freshness or state-consistency checks, such as nonce or commitment mismatches. Economic MEV effects are outside the scope of this work.

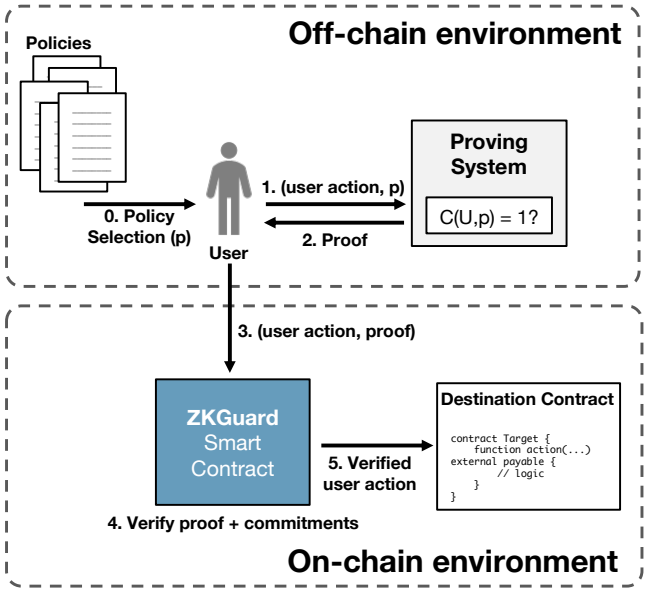

Fig. 1. Conceptual architecture of ZKGuard and transaction lifecycle. The user selects a policy and obtains an off-chain proof that the action is compliant. The action and proof are submitted to the ZKGuard contract, which verifies the proof and commitments before forwarding the action to its destination.

IV. ZKGuard Architecture

This section describes the main components of ZKGuard: the policy specification language, the commitment scheme that binds policies to on-chain state, the proof generation pipeline, and the on-chain verification and execution flow. Figure 1 illustrates the end-to-end transaction lifecycle.

At a high level, the flow proceeds as follows. The user prepares an action U (to, value, data, nonce) and selects a policy p together with its Merkle path θ_p. The off-chain policy engine checks compliance ($C(x, w) = 1$, where C is a circuit of algebraic constraints which is equal to 1 if and only if $(x, w) \in \mathcal{R}$) and generates a proof π covering both membership and compliance. The user then submits (U, π) on-chain, where the wallet verifier checks proof validity and equality of the public commitments (user action hash, policy root, group and allowlist hashes) against the values stored in the contract. If all checks pass, the wallet executes U; otherwise the transaction is rejected.

A. Policy Specification

A policy defines a set of constraints that must be checked against a user action. Table II summarizes the schema used in this work.

Listing 1 gives the *Amount Limits* policy from our evaluation set. It permits an ERC-20 transfer of at most

TABLE II
POLICY SCHEMA FIELDS.

Field	Values / description
`Id`	Policy identifier.
`TxType`	Either a token/native transfer or a smart-contract call.
`DestinationPattern`	Destination can be unrestricted or restricted to a named address set, such as a group of internal wallets or an allowlist of approved external contracts.
`SignerPattern`	Signer can be unrestricted, required to match one exact address, required to belong to a named signer group, or required to satisfy a threshold from a named signer group.
`AssetPattern`	Asset can be unrestricted or required to match one exact token contract address.
`AmountMax` (opt.)	Maximum transfer amount. If unset, no amount bound is enforced.
`FunctionSelector` (opt.)	Allowed contract-call function selector. If unset, no selector restriction is enforced.

10,000 USDC to a member of the `TeamWallets` group, signed by a designated address. The user selects this policy in step 0 of Figure 1, and the prover then establishes its membership in the committed set and the action's compliance with it.

Listing 1. A policy from the evaluation set (the *Amount Limits* scenario).

```
{ "id": 6,
  "tx_type": "Transfer",
  "destination": { "Group": "TeamWallets" },
  "signer":      { "Exact": "0xf39Fd6...fFb92266" },
  "asset":       { "Exact": "0xA0b869...3606eB48" },
  "amount_max": 10000000000,
  "function_selector": null }
```

The current policy specification is intentionally scoped to single-action constraints over destination, signer set, asset, amount, and function selector. Richer policy schemas, such as stateful or history-dependent constraints, external data feeds, or more complex conditions over calldata arguments, are possible extensions but are not covered in this work.

B. Policy, Groups, and Allowlists as Committed Artifacts

In our implementation, the policy engine consumes three configuration artifacts (stored as JSON files): (i) a policy file containing policies, (ii) a groups file mapping group names to address sets, and (iii) an allowlists file mapping allowlist names to address sets. Each proving stack maps these artifacts into its native witness representation. Policies, groups, and allowlists are written by editing these JSON files directly, following the schema of Table II (there is no separate configuration tool or API). To add or change a policy, the operator edits the policy file, recomputes the Merkle root, and submits the new commitment as a protected update, described later in this subsection.

The key property is not the file format but commitment binding. The prover computes commitments to these artifacts and these remain public. On-chain verification checks that the contract-stored wallet commitments match the claimed policy commitment, instantiated as a Merkle root over the policy set, and the claimed hash-based commitments to the group and allowlist artifacts.

A Merkle commitment is used for the policy set for two reasons. First, it lets the prover show that the policy used during compliance checking is an element of the committed policy set. Second, it keeps policy membership proofs scalable: the prover supplies only the selected policy together with a Merkle authentication path, so the work to establish inclusion grows with the depth of the tree (logarithmic in the number of policies) rather than with the total number of policies.

Hash-based commitments are computed over canonical byte encodings of the corresponding artifacts. For groups and allowlists, this requires canonicalizing set-valued data structures so that equivalent artifacts yield the same commitment independent of insertion order; the exact serialization is backend-specific and documented in the tool repository [9]. If an attacker modifies any local artifact off-chain (e.g., edits a group membership list or policy), the resulting commitment changes and will no longer match the stored on-chain state on verification.

An important consequence is that commitment updates can disrupt the system entirely. Therefore, these updates are also policy-governed operations. In our design, changing the policy, groups, or allowlist is modeled as a protected action that must itself satisfy an update policy and produce a valid proof. This means governance over security configuration is enforced by the same pipeline used for ordinary transactions: the system does not rely on a separate privileged private key path for these updates. We refer to this as a self-protecting configuration property. If its proof fails on-chain, the transaction reverts and no state changes. Because the on-chain commitments do not contain a recoverable representation of the artifacts, a policy cannot be reconstructed from its commitment alone; if the on-chain root ever diverges from the local artifacts, recovery is impossible, so deployments must retain the local policy, group, and allowlist files.

C. User Action and Wallet Abstraction

A *User Action* is the payload the smart-contract wallet is asked to execute, consisting of a destination address (`To`), native value (`Value`), calldata bytes (`Data`), a `Nonce` to prevent replay attacks, and an `OperationType` of `CALL` or `DELEGATECALL` (this work uses `CALL`).

We separate *transaction* (EOA-initiated envelope) from *user action* (wallet-executed payload). ZKGuard only requires that the wallet exposes a verification method before execution. It can be made compatible with multiple wallet constructions, including wallets supporting ERC-4337 [15]. Our reference integration is a complete, deployable Safe module [29] for the RISC Zero backend, rather than a proof-of-concept forwarder; it verifies the proof, checks commitment equality and the nonce, and then executes the user action. In its current form it is not an ERC-4337 wallet, though the wallet-agnostic design allows one with minimal changes. For the Noir and Gnark

backends we provide the proving pipeline but not the on-chain integration.

D. Proof Generation

Proof generation has three logical stages that are packaged into one proof:

a) Policy inclusion proof: Policies are committed in a Merkle tree with root R. The prover shows in zero knowledge that selected policy leaf $l = H(p)$ is included under R via path θ_p, without revealing p or θ_p publicly.

b) User action compliance proof: Given the selected private policy, the prover attests that the user action satisfies it. In the reference implementations, this check is performed over a canonicalized action representation: the action is first classified into the relevant execution path (e.g., transfer versus contract call), after which signer identities are recovered from submitted signatures and evaluated against the policy constraints. For threshold policies, the check counts unique valid signers. At a high level, compliance is encoded by Algorithm 1.

Algorithm 1 User Action Compliance Check

```
Require: policy, user_action
assert(policy.tx_type == user_action.tx_type)
match_destination(policy.destination,
user_action.destination_address)
match_signer(policy.signer, user_action.signatures)
match_asset(policy.asset, user_action.asset)
if policy.max_amount is set then
    check_max_amount(policy.max_amount,
user_action.amount)
end if
if policy.function_selector is set then
    check_function_selector(policy.selector,
user_action.selector)
end if
```

c) Commitment checks: After the user action is shown to be compliant, the proof must demonstrate that the user action, group set, allowlist set, and policy set all match the commitments declared in the public input. This guarantees that the data used for proof generation is cryptographically bound to the on-chain state.

E. On-Chain Verification and Execution

The on-chain verifier checks proof validity and equality of claimed public commitments against stored values. Replay protection is enforced by including nonce data in the committed user action. Execution occurs if and only if all checks pass. Each proving stack executes the witness before generating the proof, so an unsatisfied constraint or runtime error appears as a clear off-chain failure rather than an invalid proof. On-chain, proof verification returns only a binary result, while the other checks (commitment equality and nonce) revert with explicit error messages that distinguish a failed proof from a stale or mismatched action.

The binding between off-chain evaluation and on-chain execution relies on the fact that the proof is verified against the same public commitments stored by the wallet contract. The action, policy, group, and allowlist commitments are all part of the verified public statement. As a result, a proof generated for different local artifacts or for a different action cannot pass the on-chain equality check against the stored commitments.

Answer to RQ1: ZKGuard binds off-chain policy evaluation to on-chain wallet execution through commitment equality: the proof is valid only against the public commitments stored in the wallet contract, so a proof produced for different artifacts or a different user action cannot authorize execution.

Under the assumptions of Section III, an adversary cannot cause the wallet to execute an action that violates the committed policy while still passing verification. Doing so would require producing a proof for a false statement in the relation $\mathcal{R}$, which encodes both policy inclusion and the compliance checks of Algorithm 1. Authorization in ZKGuard therefore differs from traditional signature-only verification by additionally requiring a valid compliance proof. At the same time, policy privacy is preserved because the proof exposes only commitment values and proof validity, rather than the private policy or the rest of the committed policy set.

Answer to RQ2: Under the assumptions of Section III, ZKGuard achieves runtime authorization soundness (no policy-violating action can pass verification) while preserving policy privacy (only commitments and proof validity are revealed on-chain).

V. Experimental Evaluation

This section presents the experimental evaluation of ZKGuard across the three proving stacks.

A. Experimental Setup

All three stacks implement the same logical pipeline (policy artifacts are parsed from their JSON representation, the witness is constructed, and Algorithm 1 is executed to check compliance), but differ in how they realize it. *RISC Zero* [12] runs standard Rust inside a zkVM guest; it fixes a RISC-V circuit and proves the execution trace of a guest program, combining STARK proving with a Groth16 wrapper (STARK-to-SNARK) so that the final artifact is a Groth16 proof suitable for on-chain verification. *Noir* [13] expresses the logic as circuit constraints that compile to ACIR (Abstract Circuit Intermediate Representation), proven via Barretenberg's UltraHonk backend [22]. *Gnark* [14] expresses the logic directly as a constraint system and uses Groth16 [19] as its proving backend.

We evaluate each stack under three hardware configurations representing constrained, moderate, and well-provisioned prover environments. Table III summarizes both the hardware profiles and the software versions used across all runs.

Our *workloads* are seven policy/action scenarios (amount limits, supply lending, function-level controls, DeFi swaps, contributor payments, advanced signer policies, and dApp interaction), with identical policy and action semantics across all three stacks. The scenarios are constructed examples rather than captured production policies, but they model common wallet and custody operations such as bounded transfers to

TABLE III
EXPERIMENTAL ENVIRONMENT.

Item	Configuration
Hardware profiles (Intel Xeon Platinum, Cascade Lake)	
Low profile	2 cores, 8 GiB RAM
Mid profile	4 cores, 16 GiB RAM
High profile	8 cores, 32 GiB RAM
Software (common to all runs)	
OS	Ubuntu 22.04 LTS
Noir	`nargo 1.0.0-beta.13`, Barretenberg 0.87.0 (UltraHonk)
Gnark	v0.13.0 (Go 1.23, `gnark-crypto` v0.18.0, Groth16)
RISC Zero	`risc0-zkvm` v3.0.3 (Rust, STARK + Groth16 wrapper)

a team group, swaps and lending restricted to allowlisted protocols, and threshold-signed governance actions. Full per-scenario definitions are in the tool repository.

Our artifact bundle [9] contains the three proving-backend implementations (Gnark, Noir, RISC Zero) with their benchmark runner scripts, the raw per-sample measurements and plotting scripts, the full off-chain result tables, the policy-set scaling and RISC Zero Safe-module on-chain data, the negative-case evidence, and the ZKGuard Safe wallet module with its Foundry deployment scripts. Installation steps, dependencies, checksums, and the per-scenario policy, group, and allowlist definitions are documented in the repository READMEs and build manifests.

Our off-chain metrics follow the execution phases exposed by each proving stack. For Noir, we measure compilation, setup/preprocessing, witness generation, proving, verification, and peak RSS (Resident Set Size). For Gnark, we measure compilation (circuit construction), setup (proving/verification key generation), witness assignment, proving, verification, and peak RSS. For RISC Zero, we measure trace generation (guest execution), total cycles, proving, verification, and peak RSS.

For the policy-scaling experiment, we keep the action and selected policy semantics fixed and vary only the committed policy-set capacity from 8 to 4096 leaves on the high hardware profile, measuring the resulting proving, verification, and memory overhead. For the on-chain experiment, which is limited to the ZKGuard Safe module, we measure the deployed wallet path in terms of total transaction gas, ZKGuard-specific execution overhead, proof size, calldata size, verifier bytecode size, and deployment gas. Unless otherwise noted, we perform $n = 30$ independent repetitions per reported cell and report the median and standard deviation (σ) for every metric.

B. Off-Chain Proving Performance

The off-chain results show that the circuit-based stacks are substantially faster than the zkVM-based stack. On the high profile, Noir and Gnark achieve similar proving times, while RISC Zero incurs substantially higher latency because it proves a full RISC-V execution trace. Memory usage is primarily stack-dependent rather than scenario-dependent: Noir has the smallest footprint, followed by RISC Zero and Gnark. Verification is fastest for Groth16-based proofs, while

TABLE IV
OFF-CHAIN BENCHMARK SUMMARY ACROSS PROVING STACKS. VALUES ARE MEDIANS OVER THE $n = 30$ SCENARIO RUNS.

	Proving time (s)			Setup	Verify	Peak RSS
Stack	Low	Mid	High	**(s)**	**(ms)**	**(GiB)**
Noir (UltraHonk)	100.4	67.1	48.0	22.3	20.0	5.8
Gnark (Groth16)	–	70.4	49.5	295.7	2.1	10.3
RISC Zero (zkVM)	–	912.8	678.6	–	5.8	9.2

Noir trades higher verification cost for avoiding a per-circuit trusted setup.

To reduce space, we omit the full per-scenario off-chain tables from the main text and summarize the results through the cross-system figures and discussion below; the complete per-scenario tables are available in the artifact bundle [9].

C. Cross-System Perspective

We now compare the three stacks along the principal axes that determine deployment suitability. Figures 2 and 3 visualize proving time and the end-to-end breakdown on the high profile.

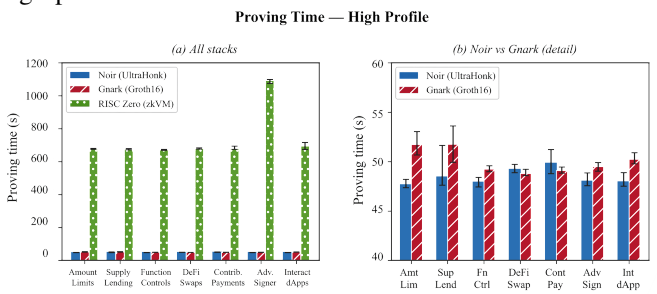

Fig. 2. Proving time on the high profile. (a) All three stacks; RISC Zero is 13–22× slower than the circuit-based stacks. (b) Detail view of Noir vs. Gnark, showing comparable proving times (≈48–52 s).

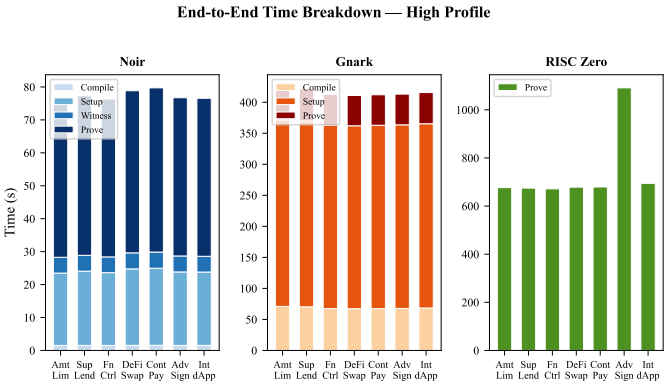

Fig. 3. End-to-end time breakdown on the high profile. Noir's time splits across compilation, setup, witness generation, and proving. Gnark is dominated by setup, and RISC Zero absorbs all overhead into proving.

a) Proving latency: On the high profile, Noir and Gnark are closely matched, with median proving times around 48–52 s across scenarios. RISC Zero is substantially slower, at roughly 11.2–11.6 minutes for single-segment scenarios and 18.2 minutes for the advanced signer case. This gap reflects

the cost of generality: RISC Zero proves execution of a full RISC-V trace, whereas Noir and Gnark prove only the application-specific constraint system. The advanced signer scenario, which doubles the RISC Zero guest cycle count, produces no measurable proving-time increase in the circuit-based stacks (the median proving time remains within the single-segment range) because signature verification is handled by optimized circuit gadgets rather than by general-purpose instruction emulation.

b) Verification: Figure 4 shows verification time per scenario and stack. Gnark and RISC Zero achieve the fastest verification since they end up verifying Groth16 proofs, while Noir's UltraHonk verification is slower. For deployment, this matters most in on-chain settings, where Groth16 benefits from mature EVM precompile support and therefore lower verification cost. Proof size is largely determined by the proof system and bounds on-chain calldata. Groth16 proofs (Gnark, and the RISC Zero on-chain wrapper) are compact and constant, whereas UltraHonk proofs are larger. For the ZKGuard Safe module we report the measured on-chain costs (proof size, calldata size, verifier-contract size, deployment gas, and per-action gas) in Tables VI–VII. Noir and Gnark are evaluated off-chain only, since we do not provide deployable on-chain verifiers for them.

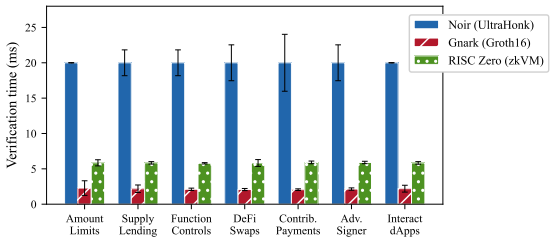

Fig. 4. Verification time in milliseconds for each proving system and scenario.

c) Peak memory: Noir has the smallest memory footprint (about 5.8 GiB), followed by RISC Zero (about 9.2 GiB) and Gnark (about 10.3 GiB). In all three stacks, memory usage is nearly constant across scenarios, which suggests that footprint is driven primarily by the proving system itself rather than by policy complexity. Operationally, this means memory provisioning depends more on stack choice than on the specific policy being proved.

d) Setup and operational overhead: Gnark has the heaviest explicit setup burden: key generation takes about 296 s on the high profile and requires per-circuit trusted-setup key management. Noir's setup is much smaller (about 22 s) and does not require a trusted setup. RISC Zero does not expose a comparable per-application setup phase in our benchmark, although its Groth16 wrapper ultimately relies on parameters that were generated in a trusted setup ceremony [30].

e) Scaling behavior: All three stacks show sublinear scaling when moving from the mid to the high profile: performance improves, but no stack approaches an ideal speedup from doubling cores and increasing memory. Noir's results across all three profiles reinforce the same pattern: scaling is meaningful, but clearly limited by overheads that do not parallelize perfectly.

Answer to RQ3: Noir and Gnark achieve sub-minute proving on the high profile, while RISC Zero offers the most general programming model at substantially higher proving cost. Groth16-based stacks (Gnark, RISC Zero wrapper) provide the lowest on-chain verification cost. Noir is the only stack viable on constrained hardware and avoids a per-circuit trusted setup. Increasing the committed policy set has only a sublinear effect on proving cost, and we quantify the on-chain enforcement cost of the ZKGuard Safe module.

D. Policy-Set Size Sensitivity

To isolate the cost of policy-set membership, we vary the committed policy-set capacity and provide results for 8, 64, 512, and 4096 policies, corresponding to Merkle depths 3, 6, 9, and 12. For each capacity, the selected policy and user action are semantically unchanged; only the Merkle authentication path length changes. This experiment therefore measures the cost of supporting larger committed policy sets, not a change in policy semantics.

TABLE V
POLICY-SET SIZE SENSITIVITY ON THE HIGH PROFILE. MEDIAN PROVING-TIME OVERHEAD VERSUS THE 8-POLICY BASELINE.

Stack	8 policies	64 policies	512 policies	4096 policies
Noir	1.00×	1.00×	1.01×	1.01×
Gnark	1.00×	1.02×	1.04×	1.05×
RISC Zero	1.00×	1.13×	1.18×	1.18×

Increasing the committed policy-set capacity increases the Merkle authentication path by one level per doubling of the capacity. Table V shows that the overhead grows sublinearly with the committed policy-set capacity, as expected from the Merkle-path membership design. The effect is negligible for Noir, small for Gnark, and more visible for RISC Zero, where the additional membership work is executed inside the zkVM trace.

E. ZKGuard Safe-Module On-Chain Costs

This experiment measures the on-chain cost of the only backend for which we provide a complete deployable wallet integration: the ZKGuard Safe module, instantiated with the RISC Zero backend. Noir and Gnark are therefore excluded from this subsection; their evaluation remains limited to the off-chain proving pipeline. The goal is to quantify the execution overhead of enforcing ZKGuard and understand the on-chain execution transaction properties.

Table VI reports the execution overhead of enforcing ZKGuard inside the Safe-module path. We measure the full transaction gas and separate the ZKGuard module component of it. The ZKGuard component captures proof verification, commitment and freshness checks. Across scenarios, this overhead is stable at approximately 256k gas and accounts for 77.2–85.1% of the protected transaction gas.

TABLE VI
ZKGUARD SAFE-MODULE EXECUTION OVERHEAD. MEDIAN $\pm\,\sigma$ OVER $n = 30$ RUNS PER SCENARIO.

Scenario	Transaction gas usage	ZKGuard gas usage	Share
Amount Limits	325332 ± 13	256078 ± 13	78.7%
Supply Lending	331766 ± 16	256078 ± 16	77.2%
Function Controls	301068 ± 15	256069 ± 15	85.1%
DeFi Swaps	301068 ± 14	256069 ± 14	85.1%
Contrib. Payments	325326 ± 12	256072 ± 12	78.7%
Adv. Signer	301068 ± 14	256069 ± 14	85.1%
Interact dApps	301068 ± 12	256069 ± 12	85.1%

TABLE VII
RISC ZERO VERIFIER SIZE, CALLDATA, PROOF SIZE, AND DEPLOYMENT COST.

Integration	Proof size	Calldata size	Verifier size	Deploy gas
ZKGuard Safe module	260 bytes	868–932 bytes	2,356 bytes	1,774,939

Table VII reports the fixed deployment footprint and per-transaction input overhead of the deployed verifier path. The proof size is constant at 260 bytes, while calldata ranges from 868 to 932 bytes across scenarios. The verifier bytecode is 2,356 bytes and the measured deployment path consumes 1,774,939 gas. Together, these measurements characterize the practical on-chain cost of using ZKGuard as a mandatory Safe-module authorization layer with the RISC Zero backend.

To confirm that policy-violating actions are rejected, we exercised negative cases by attempting to authorize actions that no committed policy permits. Such actions are rejected off-chain during proof generation, because an honest prover cannot produce a proof when no policy in the committed set is satisfied, i.e., $(x, w) \notin \mathcal{R}$. For instance, a transfer to an external address that belongs to no group and appears on no allowlist, a transfer of a token other than the permitted USDC or WETH, or an action signed by an unauthorized signer cannot be matched to any committed policy, so no proof is generated. Tampering and replay are rejected on-chain. A proof produced for altered local artifacts is internally valid, but its public commitment no longer matches the on-chain value, so the commitment-equality check fails and the Safe module reverts. A replayed action carries a stale nonce, which fails the on-chain freshness check and likewise reverts. These outcomes follow from the compliance checks of Algorithm 1 and the on-chain commitment and freshness checks, and are the empirical counterpart of the authorization-soundness argument in Section III. Scripts and Foundry revert tests reproducing these cases are in the artifact bundle [9].

F. Deployment Guidance

The evaluation results translate into the following guidance for operators choosing a proving stack for ZKGuard deployment, each keyed to a common deployment constraint.

(1) For *latency-sensitive deployments* (e.g., user-facing transaction authorization with sub-minute budgets), Noir and Gnark are both suitable, with high-profile proving times under one minute for all tested scenarios; RISC Zero's proving latency places it outside this operating regime without hardware acceleration or proof-generation parallelism.

(2) For *gas-sensitive deployments*, Groth16 is the preferred choice because it has the most compact proof format and lowest verifier gas consumption.

(3) For *memory-constrained environments*, Noir is the only stack that completed on the low profile and consistently uses the least memory, making it the best option where memory is scarce or costly.

(4) Where *developer experience and generality* matter most, RISC Zero's general-purpose Rust programming model is the most developer-friendly, with policy logic written as standard Rust code without circuit-level reasoning; this advantage is relevant in cases where more complex constraints are required.

(5) For deployments with low *trusted-setup tolerance*, Noir is the only transparent option, since Gnark (Groth16) requires a per-circuit trusted setup and key management for each application circuit, and RISC Zero also relies on a Groth16 trusted setup for its STARK-to-SNARK wrapper.

VI. THREATS TO VALIDITY

Regarding *construct validity*, the reported metrics (proving time, verification time, peak RSS, cycle count) are standard for zero-knowledge evaluation. Proving time was measured on virtualized cloud instances, so shared tenancy and hypervisor scheduling introduce variability; we mitigated this with 30 independent repetitions per configuration, reporting median $\pm\sigma$. Peak RSS approximates memory use but may miss short-lived allocations in constrained systems. On-chain gas is measured for the ZKGuard Safe module (RISC Zero backend) only (Tables VI–VII); Noir and Gnark are evaluated off-chain, since we provide no deployable on-chain verifiers for them. The RISC Zero gas figures are not a cross-backend comparison, and absolute values may depend on compiler settings, calldata encoding, the verifier, and the EVM environment.

Concerning *internal validity*, each stack required a separate implementation of the circuit logic (Rust for RISC Zero, Noir DSL for Noir, Go for Gnark), so implementation quality or compiler optimization could influence performance beyond each proof system's characteristics. RISC Zero and Gnark were not run on the low profile, since preliminary runs exceeded its resource budget, limiting cross-stack comparability on constrained hardware. All experiments used Intel Xeon Platinum (Cascade Lake); results may differ on other architectures or with GPU or FPGA acceleration.

Regarding *external validity*, the policy specification is scoped to single-action, stateless constraints; cross-transaction state, temporal conditions, or external data feeds would raise circuit complexity in ways not captured here. The seven scenarios cover the main schema aspects (transfer limits, function selectors, signer thresholds, group and allowlist checks) but not all configurations. Policy-set size was varied from 8 to 4096 committed leaves, but only on the high hardware profile; the number of signers and the size of allowlists were not varied, so their effect on proving cost is characterized only indirectly.

ZKGuard was integrated with a single wallet construction; the design is wallet-agnostic, but other implementations remain untested. Finally, proof-system toolchains evolve rapidly, so the results reflect the versions listed in Table III, and future releases of RISC Zero, Noir/Barretenberg, or Gnark may shift the observed trade-offs.

VII. Conclusion and Future Work

This paper presented ZKGuard, a runtime policy-enforcement tool for smart-contract wallets in which policy evaluation runs off-chain and on-chain execution is conditioned on a zero-knowledge proof that the selected policy belongs to a committed set and that the action satisfies it, achieving authorization soundness and policy privacy under standard cryptographic assumptions. Across three proving stacks, seven scenarios, and three hardware profiles, our evaluation maps the trade-offs among proving latency, verification cost, memory, and setup overhead, with circuit-based stacks proving in sub-minute time and Noir the only stack viable on constrained hardware. A policy-set scaling study up to 4096 committed policies shows only sublinear proving overhead, and on-chain measurements of the ZKGuard Safe module quantify the cost of mandatory on-chain enforcement.

Future work includes reducing prover latency through circuit engineering, compact witness encodings, and specialized Merkle gadgets; placing machine learning in the policy loop by committing to model parameters and proving that a model allowed an action, in line with recent zkML work on verifiable inference [31], [32]; and post-quantum compatibility via STARKs or lattice-based succinct arguments [33], [34], since Groth16 is vulnerable to quantum adversaries [19], [35].

Acknowledgments

This work is funded by national funds through FCT – Foundation for Science and Technology, I.P., within the scope of the research unit UID/00326 – Centre for Informatics and Systems of the University of Coimbra, https://doi.org/10.54499/UID/00326/2025.

References

[1] V. Buterin, "Ethereum whitepaper," 2014. [Online]. Available: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf.

[2] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," tech. rep., Ethereum, 2014. Yellow Paper. [Online]. Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[3] Fireblocks, "Fireblocks platform," 2025. [Online]. Available: https://www.fireblocks.com/.

[4] Fordefi, "Fordefi documentation," 2025. [Online]. Available: https://docs.fordefi.com/.

[5] Turnkey, "Turnkey whitepaper," 2025. [Online]. Available: https://whitepaper.turnkey.com/.

[6] Blockaid, "Blockaid platform," 2025. [Online]. Available: https://www.blockaid.io/.

[7] OpenZeppelin, "Defender," 2025. [Online]. Available: https://www.openzeppelin.com/defender.

[8] Forta Foundation, "Forta," 2025. [Online]. Available: https://forta.org/.

[9] S. Galiñanes Arienti and N. Laranjeiro, "Zkguard: A zero-knowledge policy engine for on-chain transactions – supplementary material," Apr. 2026. [Online]. Available: https://doi.org/10.5281/zenodo.19701985.

[10] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, pp. 238–252, IEEE, 2013.

[11] T. Chen, H. Lu, T. Kunpittaya, and A. Luo, "A review of zk-SNARKs," *arXiv preprint arXiv:2202.06877*, 2022.

[12] RISC Zero, "RISC Zero developer documentation," 2025. [Online]. Available: https://dev.risczero.com/api.

[13] Noir Team, "Noir documentation," 2025. [Online]. Available: https://noir-lang.org/docs/.

[14] ConsenSys, "gnark documentation," 2025. [Online]. Available: https://docs.gnark.consensys.io/.

[15] V. Buterin, Y. Weiss, D. Tirosh, S. Nacson, A. Forshtat, K. Gazso, and T. Hess, "EIP-4337: Account abstraction using alt mempool," 2021. [Online]. Available: https://eips.ethereum.org/EIPS/eip-4337.

[16] S. Goldwasser, S. Micali, and C. Rackoff, "The knowledge complexity of interactive proof systems," *SIAM Journal on Computing*, vol. 18, no. 1, pp. 186–208, 1989.

[17] O. Goldreich, "Zero-knowledge tutorial notes," 2002. [Online]. Available: https://www.wisdom.weizmann.ac.il/~oded/PSX/zk-tut02.pdf.

[18] J. Thaler, *Proofs, Arguments, and Zero-Knowledge*. Now Publishers, 2022.

[19] J. Groth, "On the size of pairing-based non-interactive arguments," in *Advances in Cryptology – EUROCRYPT 2016*, pp. 305–326, Springer, 2016.

[20] M. Bellés-Muñoz, M. Isabel, J. L. Muñoz-Tapia, A. Rubio, and J. Baylina, "Circom: A circuit description language for building zero-knowledge applications," *IEEE Transactions on Dependable and Secure Computing*, vol. 20, no. 6, pp. 4733–4751, 2023.

[21] T. Gassmann, S. Chaliasos, T. Sotiropoulos, and Z. Su, "Evaluating compiler optimization impacts on zkVM performance," *arXiv preprint arXiv:2508.17518*, 2025.

[22] Aztec/Barretenberg, "How to generate a solidity verifier," 2025. [Online]. Available: https://barretenberg.aztec.network/docs/how_to_guides/how-to-solidity-verifier/.

[23] X. Wang, J. He, Z. Xie, G. Zhao, and S.-C. Cheung, "ContractGuard: Defend Ethereum smart contracts with embedded intrusion detection," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 314–328, 2020.

[24] S. Aladhadh, H. Alwabli, T. Moulahi, and M. A. Asqah, "BChainGuard: A new framework for cyberthreats detection in blockchain using machine learning," *Applied Sciences*, vol. 12, no. 23, p. 12026, 2022.

[25] S. Azzopardi, J. Ellul, and G. J. Pace, "Monitoring smart contracts: ContractLarva and open challenges beyond," in *Runtime Verification (RV 2018)*, vol. 11237 of *Lecture Notes in Computer Science*, pp. 113–137, Springer, 2018.

[26] M. Rodler, W. Li, G. O. Karame, and L. Davi, "Sereum: Protecting existing smart contracts against re-entrancy attacks," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[27] M. Zhang, X. Zhang, Y. Zhang, and Z. Lin, "TXSPECTOR: Uncovering attacks in Ethereum from transactions," in *Proceedings of the 29th USENIX Security Symposium*, pp. 2775–2792, 2020.

[28] K. K. Chalkias, D. Maram, A. Roy, J. Wang, and A. Yadav, "Zero-knowledge authenticator for blockchain: Policy-private and obliviously updateable." IACR ePrint 2025/921, 2025.

[29] Safe, "Safe{Wallet}: Smart account modules," 2025. [Online]. Available: https://docs.safe.global/advanced/smart-account-modules.

[30] RISC Zero, "RISC Zero Groth16 ceremony," 2023. [Online]. Available: https://www.risczero.com/blog/ceremony.

[31] T. South, A. Camuto, S. Jain, S. Nguyen, R. Mahari, C. Paquin, J. Morton, and A. Pentland, "Verifiable evaluations of machine learning models using zkSNARKs." arXiv preprint arXiv:2402.02675, 2024.

[32] W. Benno, A. Centelles, A. Douchet, and K. Gibran, "Jolt atlas: Verifiable inference via lookup arguments in zero knowledge." arXiv preprint arXiv:2602.17452, 2026.

[33] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," in *Advances in Cryptology – CRYPTO 2019, Part III*, vol. 11694 of *Lecture Notes in Computer Science*, pp. 701–732, Springer, 2019.

[34] J. Bootle, A. Chiesa, and K. Sotiraki, "Lattice-based succinct arguments for NP with polylogarithmic-time verification." Cryptology ePrint Archive, Paper 2023/930, 2023.

[35] P. W. Shor, "Algorithms for quantum computation: Discrete logarithms and factoring," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*, pp. 124–134, IEEE, 1994.