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

Denis Firsov[1] and Benjamin Livshits[2]

[1]Matter Labs
[2]Imperial College London

Abstract

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

1 Introduction

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

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

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

[1]It is worth noting that throughout the text *ZK* and zero-knowledge terms stem for conventional reasons. In the context of blockchain rollups the cryptographic zero-knowledge property is not essential and is sometimes undesirable for regulation reasons.

[2]ZK Bug Tracker https://github.com/0xPARC/zk-bug-tracker

proofs. These types of issues have been dubbed as "Frozen Heart" vulnerabilities by the TrailOfBits team [3][3].

> We've dubbed this class of vulnerabilities Frozen Heart. The word frozen is an acronym for FoRging Of ZEro kNowledge proofs, and the Fiat-Shamir transformation is at the heart of most proof systems: it's vital for their practical use, and it's generally located centrally in protocols. We hope that a catchy moniker will help raise awareness of these issues in the cryptography and wider technology communities.
>
> – Jim Miller, TrailOfBits

Their blog goes on to explain that the idea behind Fiat-Shamir transformation is to compute challenges from the transcript of the protocol execution "so far." As a result, the prover can compute the challenges by itself without interaction with the verifier. The random oracle in this case is typically modeled by a strong hash function. This allows the communication to happen within a single round — the prover generates the proof and sends it to the verifier. The verifier then answers with accept or reject. As explained by TrailOfBits, these bugs are widespread due to the general lack of guidance around implementing the Fiat-Shamir transformation for different protocols.

Buggy verifiers. Bugs in ZK verifiers are potentially even more troublesome, in that they can allow an adversary to forge a malicious proof and get it accepted. In today's rollups, the prover is centralized, which connects the reputation of the prover to that of the operating company. However, as proof production decentralizes, this situation is likely to change, leaving the verifier as the only barrier between malicious provers and rendering the resulting system unsound. In the rest of the paper, we argue why *verifying the verifier* is both a necessary stepping stone to permissionless prover pool decentralization and an excellent self-contained problem for verification researchers to work on. Verifying the verifier also has the effect or reducing [4] the trusted computing base (TCB), an important goal for rollups that want to inherit the security of underlying layer-1 blockchains such as Ethereum.

Figure 1 summarizes information for the verifiers for the commonly used ZK EVMs that are running in production, together with the code sizes. Except for case of Linea, these are largely implemented in YUL, an intermediate language that can be compiled to EVM bytecode. The YUL code of these verifiers, specifically, is very low-level; we give a glimpse of some of the YUL code from the Hermez and Taiko verifiers in Figure 2.

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

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

1.1 ZK Is Not "Just Math"

Recent work [1] has demonstrated that the ZK properties outlined below are sometimes at odds with the actual implementation found in the code; it highlights more than 140 bugs that largely come

[3]https://blog.trailofbits.com/2022/04/13/part-1-coordinated-disclosure-of-vulnerabilities-affecting-girault-bulletproofs-and-plonk/

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

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

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

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

```

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

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

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

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

```

Figure 2: Low-level YUL code from to Polygon Hermez verifier and the Taiko verifier.

Project	**When (clickable)**	**Vulnerability summary**
Aztec	Sept. 2021	The bug occured in the proof aggregation step. More specifically, only the rollup proofs were aggregated but not the join-split proofs.
Plonk KZG	Oct. 2023	The vulnerability allowed a third party to derive a valid proof from a valid initial tuple $\langle proof, public\_inputs \rangle$, corresponding to the same public inputs as the initial proof. It is due to a randomness being generated using a small part of the scratch memory describing the state, allowing for degrees of freedom in the transcript. Note that the impact is limited to the PlonK verifier smart contract.
Plonk C++ verifier	Dec. 2021	A critical issue in a cutting-edge ZKP PLONK C++ implementation which allowed an attacker to create a forged proof that all verifiers will accept.

Figure 3: Verifier vulnerabilities.

from security audit reports. If we are serious about the long-term future of ZK-based techniques, we should apply the same level of scrutiny to the ZK implementation as one applies to hardware. While in the blockchain communities, TEEs like SGX are often criticized due to their vulnerabilities, failing to learn from the experience of many SGX bugs may will place us in the same position where theoretical claims are at odds with what their implementations actually provide [5][4].

1.2 Focus on the Verifier

In the context of ZK rollups a verifier is usually a publicly visible contract which is transparently deployed on the Ethereum blockchain. Because of this, we assume that verifier is not intentionally malicious, but it might be buggy. An honest verifier is associated with the following fundamental properties of proof systems: completeness, soundness, and proof-of-knowledge.

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

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

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

Out of these properties we identify *proof-of-knowledge* as the **most important** to establish in the context of blockchains as it protects users against invalid state transitions. Also, *soundness* is typically derivable from proof-of-knowledge. Proving completeness however, involves reasoning about both the honest prover and the honest verifier. Unlike the verifier, the prover itself is often a monumental piece of engineering, so we have seen approaches from compiler and bug finding literature applied to this task [6, 7]. We consider this to be an important but relatively orthogonal direction of work to reasoning about the verifier in isolation.

[4] https://sgx.fail/

1.3 Known Audits of Verifier Code

Most, if not all deployed zero-knowledge projects have undergone some form of auditing. To illustrate this point, we summarize some recent audits of ZK EVMs in Figure 3. Notably, the Linea code audit from OpenZeppelin identified a critical vulnerability in its November 2023 audit.

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

However, `u` is not sampled randomly (or in practice as the hash of the full transcript), allowing the forged proofs to pass. More specifically, not all relevant values from the transcript participated in the computation of `u` which made it possible for a malicious prover to attack the protocol.

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

Figure 4: Recent audits of ZK EVMs.

The Linea issue is not the only one discovered in verifiers over the last several years. Figure 4 shows some of the other issues in Aztec, Plonk KZG, and the C++ Plonk verifiers. This further highlights the need for verification when it comes to verifier code.

1.4 Deploying Decentralized Provers

There is a growing level of interest in distributed or decentralized proving, as exemplified by the proliferation of proof markets, from Snarketplace[5] to Gevulot[6], Succinct[7], and others. All of these rely on the off-the-shelf verifier but allow provers — permissioned or permissionless — to participate in the network using a variety of incentive mechanisms [8]. Given that these provers run on untrusted, potentially custom-designed hardware, and can run a modified version of the prover software without attestation, verifying the verifier is essential.

2 Comparing Verification Approaches

2.1 Related Work

Reasoning about complex cryptography. The closest piece of work to ours is Almeida et. al. [9]; they implement the Schnorr protocol using a low-level Jasmin programming language. Then the Jasmin implementation is automatically extracted into EasyCrypt. At the same time, authors use EasyCrypt's ZK framework [10] to specify high-level version of the Schnorr protocol, and semi-automatically prove it safe and correct. Finally, the low-level version of the Schnorr protocol, which was extracted from Jasmin code, is proved equivalent to the high-level version of the protocol. At the end, authors are able to use the proved equivalence to carry over the security proofs from the high-level protocol to the low-level version of it. The task of verifying the verifier is very similar to

[5]`https://minaexplorer.com/snarketplace`
[6]`https://gevulot.com/`
[7]`https://blog.succinct.xyz/succinct-network/`

what we are attempting to do. The main difference is that in our case the verifier is implemented in YUL and there is no available transpiler from YUL to EasyCrypt. Also note that PlonK proof system is conceptually significantly more complex than the Schnorr protocol.

Almeida et. al. [11] analyze a zero-knowledge proof system in EasyCrypt, and parts of that protocol are implemented in Jasmin. The protocol is different from the one considered here, it is the MPC-in-the-head (MitH) protocol. They prove in EasyCrypt that MitH is a restricted variant of zero-knowledge, and that it has soundness. They also aim at end-to-end verification of the protocol but with a different approach than we do here: They implement the protocol in EasyCrypt and translate the EasyCrypt code to OCaml (using code extraction). We recognize that similar top-to-bottom approach could also be used in the context of ZK rollup verifiers. However, ad-hoc top-down translation might lead to computationally less efficient verifiers which could push back the actual deployment of these verifiers.

In Barthe et. al. [12] authors formalize a special class of Σ-protocols in EasyCrypt's predecessor, CertiCrypt. Also, proofs of security and composability are derived. Similar results are re-derived in CryptoHOL [13].

In Bailey et. al.[14], authors implement a formal framework in the Lean 3 theorem prover for representing a subclass of SNARKs based on linear PCPs. More specifically, authors define a generic class of proof systems in the Algebraic Group Model (AGM) and instantiate these for various constructions, including Groth'16. In their definition of soundness authors assume ideal extractor that must succeed for every successful run of the prover. We believe that this approach constitutes a simplification and does not capture the probabilistic nature of the security offered by proof systems; please see 2.2.

Another important formal verification project for zero-knowledge is Almeida et. al. [15], where the authors implement a verification framework for developing ZK proofs. The framework encompasses a non-verified optimizing ZK compiler that translates high-level ZK proof goals to C or Java implementations, and a verified compiler that generates a reference implementation. Authors formally prove that, for any goal, the reference implementation satisfies the ZK properties and that the optimized implementation has the same observable behavior as the reference implementation.

Circuits. Reasoning about circuits is intricately related to verifying the verifier; if the relation defined by the circuit is buggy, we are ultimately failing to accomplish the overall end-to-end goal. Recently developed proof-systems target general purpose computations (as compared to the sigma protocols which historically were developed for specific relations). In most cases computations are represented as arithmetic circuits. At the same time manual implementation of circuits is highly error-prone, often leading to under- or over-constrained circuits. If a circuit is over-constrained then the prover may fail in generating a zero-knowledge proof which in the context of blockchains might result in liveness issues. When a circuit is under-constrained then the verifier might accept more witnesses than it should which in terms of blockchains might result in "soundness" issues.

In the last two years, much effort went into developing tools for automatic detection and elimination of under and over-constrained bugs in circuits [2, 16, 6]. Specialized languages like Leo [17] and Cairo [18] attempt to give end-to-end security properties to circuits written in those languages.

2.2 Choice of Tooling

Below we compare some of the more popular tool chains as options for reasoning about verifiers written in YUL.

Lean. The advantages of Lean [19] include the possibility to create a deep embedding of YUL syntax and semantics. Lean can be used to reason about the functional correctness of the verifier. However, it is hard to prove cryptographic properties without specialized logics designed specifically to address probabilistic programs. For example, Nethermind developed a formal specification of YUL in Lean [20]. We believe that such efforts could provide excellent opportunities to verify functional properties of YUL contracts, however, currently there are no good approaches for deriving cryptographic security properties.

KEVM. The KEVM [21] (K Semantics of the Ethereum Virtual Machine) describes a model of the EVM in the K-framework. Their approach provides high-guarantees of soundness of the EVM model and great degree of automation when it comes to derivation of functional properties of YUL code. However, the downside is that K-framework cannot be directly used to derive cryptographic properties in the probabilistic setting.

Why EasyCrypt. EasyCrypt is a great fit for proving cryptographic properties, such as soundness and proof-of-knowledge. It has specialized logics to prove program equivalence as well as built-in support for imperative modules, which could be used for shallow embedding of simple functions from YUL (similar approach was already demonstrated to be successful with Jasmin/EasyCrypt toolchain [22]). However, one needs to be careful to represent the YUL programs soundly as EasyCrypt modules (see more in Sec. 3.2). Given that we need to go beyond just functional correctness, where Lean could have been an easy and widely-used option, EasyCrypt is a strong choice for targeting soundness and proof-of-knowledge properties of ZK verifiers.

3 Our Approach

Let us review our approach to the verification of zkSync Era verifier, which is deployed on Ethereum blockchain as a contract `Verifier.sol` (`https://github.com/matter-labs/era-contracts/blob/f3630fcb01ad8b6e2e423a6f313abefe8502c3a2/l1-contracts/contracts/zksync/Verifier.sol`). The `Verifier.sol` contract is almost entirely implemented using YUL inline assembly. The contract has simple structure, but is massive in the number of lines of code and the low-level details.

We propose the following approach to formal verification of the verifier:

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

3.1 Example: Lagrange Polynomial Evaluation

In this section we present a walk-through a motivating example which gives some intuition for our approach. In this example we address evaluation of i^{th} term in the Lagrange polynomials, which

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

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

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

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

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

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

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

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

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

Figure 5: YUL and EasyCrypt code.

is used in the verifier. In particular, at different stages of the proof verification the verifier must evaluate the following function:

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

In `Verifier.sol`, this is implemented as function `evaluateLagrangePolyOutOfDomain` shown in Figure 5 (the left column). Observe that the original YUL implementation is non-trivially different from the original mathematical function; e.g., the high-level concepts like inverse are computed through the use of modular arithmetic at the level of bit-words. Also, the YUL function is *partial* — if the vanishing polynomial is equal to zero at point `z` then the verification is immediately aborted which signals that verification of a proof has failed.

In Figure 5 (the right column), we present the result of automatic translation of function `evaluateLagrangePolyOutOfDomain` from YUL to EasyCrypt.

Next, we manually translate the specification of function $L_i(z)$ to EasyCrypt. At this stage it is important to preserve the high-level abstraction of this function to match its mathematical description:

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

The function above is implemented in terms of operations on integers and abstract field elements (e.g., `_*_` is a binary field operation, `inv` returns the inverse, etc.). To ensure that YUL implementation is free of low-level implementation mistakes we prove that it is a refinement of high-level description:

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

The statement above is a Hoare triple that states that for any binary words `i` and `z`, the result of computation of `evaluateLagrangePolyOutOfDomain` is "equivalent" to the result of computation of `lagrangePoly`. Functions `[_]` and `{_}` convert a 256-bit word into an integer and a field element, respectively. Under the assumption that translations (YUL to EasyCrypt and math to EasyCrypt) are both sound, we have illustrated that the low-level YUL function implements high-level math function $L_i(z)$ correctly.

In this section we briefly outlined our approach to the verification of a verifier, however, it must be understood that the exposition is partial. To complete the verification process, we must apply this technique to about 30 functions of different levels of conceptual complexity. By the end of this process, we will be able to create a high-level specification of the verifier that comes out of combining these different lemmas. The next step would be to prove that the high-level verifier has the proof-of-knowledge property. Because of the equivalence mentioned above, the security proof applies to the low-level implementation.

3.2 YUL-to-EasyCrypt Translation Challenges

We use a custom-built YUL2EasyCrypt transpiler that allows us to automatically convert YUL code to the EasyCrypt modules.[8] In fact, a significant subset of YUL needs only minor syntactical adjustments to be accepted as an EasyCrypt module, with the result illustrated in Figure 5 (right-hand column). At the same time, we must be careful to ensure soundness of such translation. Below we mention some considerations regarding a transpiler from YUL to EasyCrypt:

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

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

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

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

[8]At this stage the transpiler is only able to handle a relatively simple subset of YUL which is, however, sufficient for verifier implementation.

4 Conclusions

In this paper we advocate for the use of formal reasoning to ensure strong mathematical properties for zero-knowledge verifiers. These serve as the security cornerstone for today's actively used zero-knowledge rollups and, as such, must be considered part of mission-critical infrastructure. We advocate for a specific tool-chain that involves translation of the existing YUL implementation of the verifier to EasyCrypt. We then use a series of reduction steps to formally prove security properties of the original YUL verifier. At the same time, we recognize that alternative top-to-bottom approaches [11] also provide viable alternatives to the end-to-end verification of verifiers.

In addition to minimizing the required trust and effectively reducing the trusted computing base [4] of zero-knowledge systems including rollups, this work also sets the stage for a not-so-distant future in which proving will be increasingly decentralized, making verification of the verifier a vital part of ensuring the end-to-end security of the rollup ecosystem.

Acknowledgement

We would like to thank Igor Zhirkov, Antonio Locascio, Joonatan Saarhelo, Alex Ozdemir, Matteo Campanelli, and Julian Sutherland for discussions and insights that helped us to clearly present the motivation and outline the approach for verifying the verifier.

References

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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