IACR Communications in Cryptology
Vol. 0, No. 0, 24 pages.

Who Verifies the Verifiers?

Lessons Learned From Formally Verified Line-Point Zero-Knowledge

Sabine Oechsner[1], Vitor Pereira[2] and Peter Scholl[3]

[1] Vrije Universiteit Amsterdam, The Netherlands
[2] Independent Researcher, Portugal
[3] Aarhus University, Denmark

Abstract. Computer-aided cryptography, with particular emphasis on formal verification, promises an interesting avenue to establish strong guarantees about cryptographic primitives. The appeal of formal verification is to *replace* the error-prone pen-and-paper proofs with a proof that was checked by a computer and, therefore, does not need to be checked by a human. In this paper, we ask the question of how reliable are these machine-checked proofs by analyzing a formally verified implementation of the Line-Point Zero-Knowledge (LPZK) protocol (Dittmer, Eldefrawy, Graham-Lengrand, Lu, Ostrovsky and Pereira, *CCS 2023*). The implementation was developed in EasyCrypt and compiled into OCaml code that was claimed to be *high-assurance*, i.e., that offers the formal guarantees of guarantees of completeness, soundness, and zero knowledge.

We show that despite these formal claims, the EasyCrypt model was flawed, and the implementation (supposed to be high-assurance) had critical security vulnerabilities. Concretely, we demonstrate that: 1) the EasyCrypt soundness proof was incorrectly done, allowing an attack on the scheme that leads honest verifiers into accepting false statements; and 2) the EasyCrypt formalization inherited a deficient model of zero knowledge for a class of non-interactive zero knowledge protocols that also allows the verifier to recover the witness. In addition, we demonstrate 3) a gap in the proof of the perfect zero knowledge property of the LPZK variant of Dittmer, Ishai, Lu and Ostrovsky (*CCS 2022*) that the EasyCrypt proof is based, which, depending on the interpretation of the protocol and security claim, could allow a malicious verifier to learn the witness.

Our findings highlight the importance of scrutinizing machine-checked proofs, including their models and assumptions. We offer lessons learned for both users and reviewers of tools like EasyCrypt, aimed at improving the transparency, rigor, and accessibility of machine-checked proofs. By sharing our methodology and challenges, we hope to foster a culture of deeper engagement with formal verification in the cryptographic community.

1 Introduction

Provable security is a cornerstone of modern cryptography. The idea is to carefully model security as an abstraction of the real world, in a way that allows to derive formal guarantees about the security of the cryptographic construction in question (within the limitations of the model). Such security *proofs* are usually presented in written form (typically dubbed *pen-and-paper* proofs) for academic papers. To verify such a proof, the reader has to carefully examine all parts of the security statement, including the description of the construction as well as the security model, and then check the soundness of all

E-mail: s.a.oechsner@vu.nl (Sabine Oechsner), vitor.pereira@algoritmosapiente.com (Vitor Pereira), peter.scholl@cs.au.dk (Peter Scholl)

This work is licensed under a "CC BY 4.0" license.

presented proof steps in their head. Cryptographers have developed various strategies and proof writing styles to support this process: for example through modular definition and reasoning styles [Can01, Mau10, BDF+18] or a more rigorous treatment of proof steps in the context of game-based security [Sho04, BR06, Hal05]. Still, conventional wisdom says that proof writing and especially security modeling is challenging, and both models and proofs should be checked carefully before adoption.

More recently, the help of computers in this process has been explored in the form of computer-aided cryptography [BBB+21]. Proof assistants such as EasyCrypt [BGHZ11] and Isabelle/CryptHOL [BLS20] can help here by allowing to machine-check the (very detailed) proof a human wrote in a specific language. While the cost of *writing* such proof is much higher, once a machine-checked proof exists, it relieves the reader of the burden of *verifying* the proof: all that is left to do for humans at this point is to check the security statement.

The cryptographic community seems largely in favor of the development of computer-aided cryptography in all its shapes. Since security proofs in this space require considerable expertise outside of traditional cryptographic research, they are currently mostly relegated to experts with specialist training. In this work, we argue that while computer-aided cryptography greatly improves the confidence in security proofs, it does not relieve the burden of checking the security models. While we agree that *writing a proof* requires significant training and hence a division of labor is a good idea to some extent, we want to point out that *reading definitions and lemmas* should be relatively straightforward for researchers unfamiliar with the formal tools. We hope that our work encourages more cryptographers to start reading machine-assisted cryptographic formalizations, and to present their security proofs in a way that simplifies the process of developing a formally verified version.

To support our claim that examining security statements is a crucial part of reviewing really any security proof, we discuss a recent work with seemingly contradicting security properties. At *CCS 2023*, Dittmer, Eldefrawy, Graham-Lengrand, Lu, Ostrovsky and Pereira [DEGL+23] presented a so-called high-assurance OCaml implementation[1] of Line-Point Zero-Knowledge (LPZK) [DIO21, DILO22a], a non-interactive zero-knowledge protocol for circuit satisfiability. To obtain their OCaml implementation, the authors first developed an implementation and security proof in the EasyCrypt proof assistant, and then extracted a verified OCaml implementation from it. (The authors also showed various code optimizations which will not be relevant to our work.) However, we found several modeling issues in the EasyCrypt model, related to attacks that are well within traditional security models, including a simple way to convince an honest verifier of any statement. The authors also claimed a formally verified proof of the zero knowledge property, following the pen-and-paper proof of Dittmer, Ishai, Lu and Ostrovsky [DILO22a], yet we identified an issue in the pen-and-paper proof. (Again, this seeming contradition is due to a modeling issue in the EasyCrypt model.) We emphasize that, while LPZK provides an interesting case study that demonstrates the practical relevance of the attacks presented in this paper, our contributions extend beyond this protocol and the specific work of [DEGL+23].

1.1 Our Contributions

This work provides contributions on two levels: On the level of LPZK itself, where we present three observations about the security of LPZK (implementations), and more generally at the level of computer-aided cryptography and its interaction with the wider cryptographic community. In summary:

- We describe an attack on and a flaw of the high-assurance OCaml LPZK implementation of Dittmer, Eldefrawy, Graham-Lengrand, Lu, Ostrovsky and Pereira [DEGL+23].

[1] https://github.com/SRI-CSL/high-assurance-crypto/tree/main/high-assurance-zk/lpzk

The former is a simple attack on soundness (malicious prover can convince verifier of any statement) while the latter affects zero knowledge (malicious verifier can extract witness). We also discuss how both are related to modeling issues, i.e. the EasyCrypt proofs themselves are of course "correct", but the proved lemmas do not match the ones on paper. (Sections 3 and 5)

- We also uncover an issue in the pen-and-paper security analysis of the LPZK variant of Dittmer, Ishai, Lu and Ostrovsky [DILO22a] that means that zero knowledge does not hold as stated. Since the formally verified security proofs of Dittmer, Eldefrawy, Graham-Lengrand, Lu, Ostrovsky and Pereira [DEGL+23] claim to match the pen-and-paper guarantees of Dittmer, Ishai, Lu and Ostrovsky [DILO22a], we also argue how the EasyCrypt modeling does in fact differ in crucial ways. (Section 4)

- We present lessons learned about the challenges of computer-aided cryptography and interacting with the proofs it produces. (Section 6)

We stress that the identified vulnerabilities are specific to this particular implementation. Nevertheless, we believe that the insights gained from identifying and exposing these vulnerabilities are transferable to similar contexts. Moreover, while the discussion about modeling choices in the EasyCrypt formalization might give the impression of an issue with the tool itself, we remark that this is not the case: The real issue is how it was used to model the concrete protocol and security notions, and insights are transferable to other tools and proof assistants.

Disclosure The vulnerabilities and attacks described in our paper have been reported to and acknowledged by the authors of [DEGL+23] and [DILO22a]. Since the disclosure, only the soundness proof vulnerability identified in [DEGL+23] (Section 3) has been fixed. We discussed the amount of work involved in fixing the zero knowledge issue (Section 5) with the authors of [DEGL+23], and it is our understanding that a fix would include redoing the whole modeling (and proof) of the zero knowledge property. For the issue in the proof of zero knowledge of [DILO22a, DILO22b] (Section 4), we have included a sketch of how to fix the pen-and-paper proof.

1.2 Technical Overview

Here we go into slightly more detail about our observations for LPZK and its implementations. Our main focus was the work of Dittmer, Eldefrawy, Graham-Lengrand, Lu, Ostrovsky and Pereira [DEGL+23], called LPZK-DEGLOP23 from here on. In this work, the authors implemented a version of LPZK in the EasyCrypt proof assistant and provided formally verified security proofs for completeness, soundness and zero knowledge. From this EasyCrypt model, an OCaml implementation was extracted. The implemented LPZK protocol is essentially the original protocol of Dittmer, Ishai and Ostrovsky [DIO21] (LPZK-DIO21) with a tweak due to Dittmer, Ishai, Lu and Ostrovsky [DILO22a] (LPZK-DILO22) that encodes the witness in the constant term $\vec{b}$ of the line $\vec{a}X + \vec{b}$ rather than in $\vec{a}$. LPZK-DEGLOP23 leave the generation of the correlated randomness $(\vec{a}, \vec{b})$ for the prover and $(\alpha, \vec{v} = \vec{a}\alpha + \vec{b})$ for the verifier to a trusted dealer.

Soundness of the LPZK-DEGLOP23 protocol The first observation concerns the soundness claim of LPZK-DEGLOP23. Despite the promise of formally verified security proofs, we found an attack on soundness that is well within the security model, and that allows a malicious prover to convince an honest verifier of *any* statement. What happened is that the work had both an implementation bug (the verifier algorithm was missing a

crucial consistency check) and a modeling bug (the soundness lemma was too restrictive). The latter turned out to mask the implementation bug.

Concretely, soundness was shown with respect to an assumption on the behaviour of the malicious prover, stated as an axiom. Malicious provers were assumed to only output proofs that follow a certain pattern, where the proof could plausibly be explained as executing the honest prover algorithm on *some* w (not necessarily a valid witness). This axiom was not acknowledged in the CCS paper, and we only found it by chance in the EasyCrypt code when trying to understand the reported implausible soundness error: The CCS paper (wrongly) claimed to have achieved a soundness error of $1/|\mathbb{F}|$ which would have contradicted the $2/|\mathbb{F}|$ soundness error in the original LPZK papers [DIO21, DILO22a].

Zero Knowledge of the LPZK-DILO22 and LPZK-DEGLOP23 protocols Our second observation concerns the zero knowledge guarantee of LPZK protocols that follow the LPZK-DILO22 tweak of encoding the witness in $\vec{b}$. LPZK-DILO22 claims to achieve perfect zero knowledge against malicious verifiers for this construction. We argue that their security proof is missing a corner case: If the verifier's evaluation point α is 0, then the corresponding $\vec{v} = \vec{a}\alpha + \vec{b}$ that the verifier receives will be equal to $\vec{b}$, and thus leak the witness. Since the simulator is asked to simulate $\vec{v}$, simulation must fail for this case. What this means is that LPZK-DILO22 either achieves only statistical honest verifier zero knowledge, or the dealer needs to be restricted to enforce $\alpha \neq 0$, reducing the soundness error slightly to $2/(|\mathbb{F}| - 1)$.

Interestingly, LPZK-DEGLOP23 claimed to have a formally verified proof of perfect zero knowledge for a similar construction that also encodes the witness in $\vec{b}$. However, we find that this claim is due to another modeling issue: The combination of zero knowledge notion and trusted dealer is expressed in such a way that the simulation can be independent of the prover's correlated randomness, and hence the simulation succeeds even if $\alpha = 0$.

A note on the LPZK-DEGLOP23 OCaml implementation Our third observation concerns the OCaml implementation of LPZK-DEGLOP23 that is extracted from the EasyCrypt model. In an ideal world, it should inherit all the security guarantees of the underlying EasyCrypt model. Unfortunately, the authors decided to combine the verifier with part of the trusted dealer in such a way that the OCaml verifier code takes the prover's randomness $(\vec{a}, \vec{b})$ as input. With this knowledge, even a semi-honest verifier could extract the witness. Note that this vulnerability is not induced by the code extraction mechanism itself, but rather due to the need to combine it with manual code that can itself induce security flaws.

1.3 Related Work

This work is concerned with verifying security proofs (and more generally the verifiability of security proofs). We are obviously not the first to think about this issue. In 2006 (and again in 2019), Koblitz and Menezes's [KM06, KM19] provided a very interesting and detailed critique of security proofs, where they point to existing *deficient protocol descriptions*, *inadequate definitions*, *bodacious assumptions* and *problematic proofs*. As an example, the authors list an inadequacy on the well established chosen-ciphertext attack (IND-CCA) security notion for public key encryption that took years to uncover [BHK15]. In terms of actual proofs, the authors list ten provable security claims that have falacies in the proofs, ranging from public key ecryption padding to random oracles and ideal ciphers.

Perhaps the first large-scale work to analyze a formally verified implementation of a cryptographic protocol was that of [miT25], which demonstrated both the potential and the limitations of formal verification in this domain. On the one hand, the authors uncovered TLS vulnerabilities such as SMACK, FREAK, and Logjam, which were subsequently

avoided in the resulting miTLS implementation. On the other hand, they also discussed the limitations of their methodology, identifying attack vectors beyond the scope of the miTLS analysis, such as side-channel vulnerabilities and weaknesses in unverified external libraries. Even during the recent NIST post-quantum standardization effort, Kudinov *et. al* [KKF20] identified a flaw in the security proofs of the SPHINCS+ hash-based scheme [HBD+20] during submission round three. Concretely, the flaw was found on the security proof of the underlying WOTS+ scheme, where a claim made for the case of a single hash function query was incorrectly extended to the case of a chain of hash functions. Similar issues have been found on hardware-related assumptions in formally verified cryprography. Athalye *et. at* [AKZT23] demonstrate that real hardware can break simple leakage model assumptions used in libraries like HACL⋆ [ZBPB17, PBP+20] or Fiat-Crypto [EPG+19].

Finally, we mention some works that study attacks on zero knowledge schemes: Garms and Livesey [GL25] also addresses a soundness error on a ZK scheme, similarly to what we are targeting with this paper. The work focuses on an optimization of the Gemini protocol [BCHO22] for which is possible to define an attack that will convince a verifier to accept an incorrect evaluation of a multilinear polynomial for any input. Note that the attack is not possible on the original (non-optimized) version of Gemini. Celi, Levin and Rowell [CLR24] describe gaps in the special soundness proofs of the NIZKPoK by Agrawal, Ganesh and Mohassel [AGM18] and its variant ZKAttest [FLM22] as well as a practical attack on ZKAttest. Gabizon [Gab19] reports a flaw in the description of the BCTV zk-SNARK [BCTV14], a variant of Pinocchio [PHGR13], that allows a malicious prover to prove knowledge for any public input given a valid proof for some public input. The weak Fiat-Shamir attack [BPW12, HLPT20, DMWG23] is another example of how difficult it can be to get the security of zero knowledge implementations right.

2 Preliminaries

We provide in this section the formal cryptographic definitions that we use in our formalization, which are all standard, and an explanation of the LPZK protocol. We then include a short overview of EasyCrypt and of formal verification in the context of CAC, before concluding with a recap of the LPZK-DEGLOP23 paper.

2.1 Zero-Knowledge Definitions

We adopt the same nomenclature and follow closely the definitions provided in the ZKProof Community Reference document [ZKP22]. A ZK protocol is a cryptographic protocol run between a prover $\mathcal{P}$ and a verifier $\mathcal{V}$, where $\mathcal{P}$ tries to convince $\mathcal{V}$ that some statement is true without revealing any confidential information about it. The prover's private input is named the *witness*, w, while the *statement* refers to the public information held by both parties, which consists of the *instance* x and a *relation* $\mathcal{R}(x, w)$, which is a circuit that specifies the relation between the instance and the witness. That is, a successful evaluation of $\mathcal{R}$ means that x is a valid instance for the relation. For some statement $(x, \mathcal{R})$, if there exists a witness for which $\mathcal{R}(x, w)$ is true, then the statement is part of the *language* L.

A ZK proof for an NP relation $\mathcal{R}(x, w)$ is a (possibly interactive) protocol between a prover, $\mathcal{P}$, and verifier, $\mathcal{V}$, where at the end, the verifier outputs 1 or 0 indicating whether it accepts or rejects the proof. In ZK protocols, $\mathcal{P}$ takes as input the statement and the witness w; $\mathcal{V}$ is only given the statement.

Informally, a ZK proof should satisfy the following three properties:

- **Completeness:** if $\mathcal{R}(x, w)$ is valid, then in an honest protocol execution, the verifier accepts.

- **Soundness:** if the statement is false, a malicious prover cannot make the verifier accept, except with low probability.
- **Zero-Knowledge:** even if the verifier is malicious, it cannot learn any information about the witness w, beyond the fact that the statement is true.

The zero-knowledge property is often relaxed to *honest-verifier* zero-knowledge, where the verifier is assumed to follow the protocol. A common strengthening of the soundness property is *knowledge soundness*, which ensures that if the verifier accepts then the prover must know a valid witness w for the statement.

2.1.1 Line-Point Zero-Knowledge

Line-point zero-knowledge (LPZK) is a variant of a standard zero-knowledge protocol for proving arithmetic circuit satisfiability over a finite field $\mathbb{F}$, where the prover and verifier are first provided with secret, *correlated randomness* before running the protocol. The correlated randomness for the prover consists of a set of random lines (over $\mathbb{F}$), while the verifier receives a random scalar, $\alpha \in \mathbb{F}$, and the evaluation of each of the lines at the point α.

We focus on *non-interactive* LPZK protocols where, after the correlated randomness setup, only one message is sent from the prover to the verifier. Below, we give the formal syntax and security properties of LPZK. The definition is parametrised by the choice of finite field, $\mathbb{F}$, which is implicitly a function of the security parameter, λ. The class of relations being proven is satisfiability of an arithmetic circuit C with $n_p + n_s$ input wires, that is, the prover wants to show that $C(x, w) = 0$ for some public input $x \in \mathbb{F}^{n_p}$ and secret witness $w \in \mathbb{F}^{n_s}$.

Definition 1 (Line-Point Zero-Knowledge protocol)**.** An LPZK protocol is a tuple of polynomial-time algorithms $(\mathsf{Dealer}, \mathsf{Prove}, \mathsf{Verify})$, such that:

- $\mathsf{Dealer}(1^\lambda, C)$ is a probabilistic algorithm that on input the security parameter 1^λ and an arithmetic circuit C, with $n_p + n_s$ inputs and m multiplication gates, samples the prover's randomness $R_p := (\vec{a}, \vec{b})$, where $\vec{a}, \vec{b} \leftarrow \mathbb{F}^{n_s+2m}$, and the verifier's randomness $R_v := (\alpha, \vec{v})$, where $\alpha \leftarrow \mathbb{F}$ and $\vec{v} := \alpha\vec{a} + \vec{b}$.
- $\mathsf{Prove}(1^\lambda, C, x, w, R_p)$ is a deterministic algorithm run by the prover that takes as input the circuit C, public input $x \in \mathbb{F}^{n_p}$, secret witness $w \in \mathbb{F}^{n_s}$, and prover randomness R_p, and outputs a *proof* π to be sent to the verifier.
- $\mathsf{Verify}(1^\lambda, C, x, R_v, \pi)$ is a deterministic algorithm run by the verifier that takes as input the statement, randomness R_v and the proof π and produces a decision bit b attesting whether the verifier accepts the proof.

The security properties are defined as follows:

- Completeness: in an honest protocol execution, if $C(x, w) = 0$, then Verify outputs 1 with probability 1.
- Soundness: Let $\mathcal{P}^*$ be a malicious and computationally unbounded prover algorithm, which outputs a proof π^* as well as its own correlated randomness R_p^*. The LPZK protocol has soundness error $\varepsilon = \varepsilon(\lambda)$ if, for any C, x such that $C(x, w) \neq 0$ for all $w \in \mathbb{F}^{n_p}$:

$$\Pr\left[\mathsf{Verify}(1^\lambda, C, x, (\alpha, \vec{v}), \pi^*) = 1 \;:\; \begin{array}{r}((\vec{a}^*, \vec{b}^*), \pi) \leftarrow \mathcal{P}^*(C, x),\\ \alpha \leftarrow \mathbb{F}, \vec{v} = \alpha\vec{a}^* + \vec{b}^*\end{array}\right] \leq \varepsilon$$

- Zero-Knowledge: For any malicious ppt verifier $\mathcal{V}^*$, there exists a ppt simulator $\mathcal{S}$, such that the view of $\mathcal{V}^*$ is statistically indistinguishable from the output of $\mathcal{S}$ on input the statement and the prover's randomness. Formally, there exists a negligible function $\mu(\lambda)$ such that for all circuits C and inputs (x, w) for which $C(x, w) = 0$, and for all (potentially unbounded) distinguishers $\mathcal{D}$:

$$\Pr[\mathcal{D}(C, x, w, (R_v, \pi)) = 1] - \Pr[\mathcal{D}(C, x, w, \mathcal{S}(C, x, R_p)) = 1] \leq \mu(\lambda)$$

  where the probabilities are taken over the randomness of $(R_v, R_p) \leftarrow \mathsf{Dealer}(C, x)$, $\pi \leftarrow \mathsf{Prove}(C, x, w, R_p)$, the simulator $\mathcal{S}$ and the distinguisher $\mathcal{D}$.

The Dealer algorithm that samples the correlated randomness is a form of trusted setup, which for the purposes of this paper, we assume is performed by a trusted dealer. In practice, when implementing LPZK and related protocols, the dealer can be replaced by a vector oblivious linear evaluation (VOLE) protocol, to obtain a standard (designated-verifier) ZK protocol with no trusted setup.

For the zero-knowledge property, if $\mu(\lambda) = 0$ then we say the protocol has *perfect* zero-knowledge, otherwise it is *statistical*. Relaxing the runtime of the distinguisher to be polynomial-time leads to *computational* zero-knowledge. The line-point zero-knowledge protocol we study in this paper satisfies statistical zero-knowledge against a malicious verifier, as well as the stronger knowledge soundness property. However, knowledge soundness was not formalised in EasyCrypt in [DEGL+23], so we do not discuss it further.

2.2 Line-Point Zero-Knowledge (LPZK) Protocol

Circuit format. As mentioned above, we assume an arithmetic circuit over a finite field $\mathbb{F}$, consisting of fan-in two addition and multiplication gates. There are $n = n_s + n_p$ inputs, m multiplication gates indexed $\{1, \ldots, m\}$, and W wires in total, indexed $\{1, \ldots, W\}$, which may have arbitrary fan-out. Without loss of generality, we assume wires $\{1, \ldots, n_s\}$ are the wire indices associated with the witness input w, while wires $\{n_s + 1, \ldots, n_s + n_p\}$ are associated with the public input x.

The protocol. The LPZK protocol comes in several different variants. We detail the "IT-LPZK-v1" version used in [DEGL+23], which is essentially the information-theoretic construction from [DIO21] together with a small tweak from [DILO22a]. For more discussion on the differences between the LPZK variants, see Section 2.2.1.

The idea is that the prover will encode elements from its witness into the $\vec{b}$ components of the line, which defines *commitments* to those values. This is done by sending a correction value $d_i = w_i - b_i$, for the i-th witness element w_i, and then using the line $f'_i(X) = a_i(X) + w_i$ for these components of $\vec{f}$, instead of $f_i(X)$. The verifier then adds d_i to its evaluation v_i, obtaining $f'_i(\alpha) = a_i\alpha + w_i$. We will refer to these evaluations held by the verifier as *tags*, denoted q_i.

Then, the prover does a similar thing for the output wire of every (fan-in two) multiplication gate in the arithmetic circuit. That is, for the j-th multiplication gate with input values x and y, the prover sends the correction $xy - b_{j+n}$. The prover and verifier then apply the correction as before, so the prover obtains a new line $f'_w(x) = a_{j+n}x + xy$ and the verifier holds $q_w := f'_w(\alpha)$, where w is the index of the gate output wire.

In order to show satisfiability of the circuit, the parties can then homomorphically evaluate addition and multiplication gates on committed values. They start by evaluating every addition gate in the circuit, by simply having the prover and verifier add their respective lines or evaluations. Now, the parties hold a line commitment to every wire in the circuit. It remains to prove that the outputs of the multiplication gates were committed to correctly by the prover. For a multiplication gate with inputs b_l, b_r and output b_o, there

Dealer$(1^\lambda, C)$:

1. Let $N = n_s + 2m$
2. Sample $\vec{a}, \vec{b} \leftarrow \mathbb{F}^N$, $\alpha \leftarrow \mathbb{F}$ and let $\vec{v} = \vec{a}\alpha + \vec{b}$
3. Output $R_p = (\vec{a}, \vec{b})$ to $\mathcal{P}$ and $R_v = (\alpha, \vec{v})$ to $\mathcal{V}$

Prove$(1^\lambda, C, x, w, (\vec{a}, \vec{b}))$:

1. For $i \in \{1, \ldots, n_s\}$, let $d_i = w_i - b_i$. Define the commitment polynomial $f_i(X) = a_i X + w_i$
2. For $i \in \{1, \ldots, n_p\}$, define the constant commitment polynomial $f_{n_s+i}(X) = x_i$
3. For each gate in the circuit with input wire indices (l, r) and output wire index o, let $f_l(X) = a_l X + b_l$, $f_r(X) = a_r X + b_r$ be the commitments to the input wires. Define the output commitment $f_o(X)$ as follows:
   - Addition gate: let $f_o(X) = (a_l + a_r)X + b_l + b_r$
   - For the j-th multiplication gate (where $j \in \{1, \ldots, m\}$):
     - Let (a_o, b_o) and (a', b') be the $n_s + j$-th and $n_s + j + 1$-th pairs from R_p
     - Let $f_o(X) = a_o X + b_l b_r$ be the commitment to the output wire
     - Define
     $$z_{3j+1} = b_l b_r - b_o$$
     $$z_{3j+2} = a_l a_r - a'$$
     $$z_{3j+3} = a_l b_r + a_r b_l - a_o - b'$$
4. Let $\hat{a}, \hat{b}$ be the coefficients of the commitment to the output wire, $f_W(X) = \hat{a}X + \hat{b}$ (where $\hat{b} = 0$ if $C(x, w) = 0$).
5. Send the proof $\pi := (d_1, \ldots, d_{n_s}, z_1, \ldots, z_{3m}, \hat{a})$

Figure 1: Trusted dealer and prover algorithms for the LPZK protocol

are line components $f_l(X) = a_l X + b_l$, $f_r(X) = a_r X + b_r$ and $f_o(X) = a_o X + b_o$, which commit the prover to b_l, b_r and b_o.

The prover can then define the degree-2 polynomial:

$$g(X) := f_l(X) f_r(X) - f_o(X) = a_l a_r X^2 + (a_l b_r + b_r a_l - a_o)X + b_l b_r - b_o$$

which can be seen as a form of degree-2 commitment to $b_l b_r - b_o$, where the verifier holds an evaluation of g at α. To verify that the gate is correct, the prover needs to show that $b_l b_r = b_o$, that is, the constant coefficient is zero. To do this, the prover takes another random VOLE pair $f'(x) = a'x + b'$, and sends the coefficients of:

$$f_l(X) f_r(X) - f_o(X) - X f'(X) = (a_l a_r - a')X^2 + (a_l b_r + a_r b_l - a_o - b')X + (b_l b_r - b_o)$$

Since the constant coefficient should be zero, the prover only needs to send the degree-1 and -2 coefficients. For the j-th multiplication gate, these are denoted z_{3j+2}, z_{3j+3} in Figure 1. Note that the additional random commitment $f'(x)$ is needed to ensure that any sensitive information in these coefficients is perfectly masked.

Verify$(1^\lambda, C, x, (\alpha, \vec{v}), \pi = (d_1, \ldots, d_{n_s}, z_1, \ldots, z_{3m}, \hat{a}))$

1. For $i \in \{1, \ldots, n_s\}$, compute the wire tag $q_i = v_i + \alpha d_i$.
2. For $i \in \{1, \ldots, n_p\}$, compute the wire tag $q_{n_s+i} = x_i$
3. For each gate in the circuit, with input wire indices (l, r) and output wire index o, let the input tags be q_l, q_r, and define the output tag q_o as follows:
   - Addition gate: let $q_o = q_l + q_r$
   - For the j-th multiplication gate:
     - Let q, q' be the $n_s + j$-th and $n_s + j + 1$-th entries from $\vec{v}$
     - Let $q_o = q + \alpha z_{3j+1}$
     - Let $e' = q' + \alpha z_{3j+2}$
     - Check that $q_l q_r - q_o - \alpha e' = z_{3j+2}\alpha$
4. For the circuit output wire, with tag $\hat{q}$, check that $\hat{q} = \hat{a}\alpha$
5. If all multiplication checks and the output check passed, output 1, otherwise output 0.

Figure 2: Verifier algorithm for the LPZK protocol

The verifier can then evaluate this polynomial at α and check that this is consistent with the evaluation computable from its own VOLE outputs. If the prover is malicious and the multiplication gate was incorrect, then the constant coefficient would be non-zero. The verification equation would then only pass if the evaluation at α of the polynomial sent by the prover happens to equal the evaluation of the original degree-2 polynomial which the verifier holds. Since these polynomials are not equal, their difference is non-zero and has at most 2 roots, meaning the verifier will accept with probability at most $2/|\mathbb{F}|$.

In addition to the multiplication checks, the prover sends the non-zero coefficient of the commitment to the final output wire, which we denote as $\hat{\alpha}$, to allow the verifier to check the circuit evaluates to zero.

The original information-theoretic LPZK protocol [DIO21] uses an optimization to the above, which batches several multiplication checks together. Instead of sending two coefficients of the polynomial above for each multiplication gate, the prover sends only the degree-1 coefficients. Then, the prover takes all of the constant coefficients in the batch, multiplies them together and sends over the resulting constant to the verifier. The verifier performs analogous operations on its evaluations and checks that it obtains the same constant. For a batch size of t, this gives a soundness error of $2t/|\mathbb{F}|$, since in the malicious case the final polynomial could have degree up to $2t$.

2.2.1 Comparison with Prior LPZK Specifications

LPZK Protocol. The protocol specification in Section 2.1.1 is a slightly more detailed version of the protocol from [DEGL+23, Section 2], which was used as the basis for the EasyCrypt formalisation. This protocol is essentially the same as information-theoretic construction from [DIO21], with one tweak due to [DILO22a], which is to encode the witness information in the constant coefficients of the f_i polynomials instead of the degree-1 coefficients. This change was made to slightly reduce the computational overhead of the verifier, however, as we shall see in Section 5, it also turns out to affect the security properties.

LPZK Definition. One difference between Definition 1, and prior definitions from [DIO21, DILO22a], is the treatment of the trusted dealer. Here, following [DEGL+23], we include an explicit dealer algorithm that models a trusted party who generates correlated randomness.

Other LPZK works [DIO21, DILO22a], on the other hand, use definitions where the dealer is left implicit. More precisely, in e.g. [DILO22a], the prover algorithm only takes as input the statement and the witness, and outputs a pair of vectors $(\vec{a}, \vec{b})$. The verifier algorithm then takes as input α and the evaluation $\vec{v}(\alpha) = \vec{a}\alpha + \vec{b}$, and outputs a decision bit. Thus, the proof is directly incorporated into the verifier's $\vec{v}(\alpha)$ value, rather than being sent by the prover.

While this approach cleanly captures the information-theoretic object at the heart of LPZK protocols, it requires an extra translation step to obtain a secure protocol that can be implemented. One option is to extract a trusted dealer algorithm that can distribute correlated randomness to the parties (and tweak the prover/verifier algorithms accordingly). Another approach, discussed in [DIO21], is to compile the LPZK protocol into a standard ZK protocol using a random vector oblivious linear evaluation protocol, run between the prover and verifier.

This difference in handling of the trusted dealer also changes the presentation of the security properties. For example, when defining the zero-knowledge property in [DIO21, DILO22a], the definition quantifies over all $\alpha \in \mathbb{F}$, and the simulator needs to simulate the verifier's input $\vec{v}(\alpha)$. In contrast, we use a more traditional, real/ideal definition, where the simulator needs to produce the view of the malicious verifier, which includes its correlated randomness as well as the proof.

2.3 Formal Verification and EasyCrypt

CAC [BBB+21, BBB+19] is a research area that aims at applying formal, machine-checkable approaches to the design, analysis and implementation of cryptography with the broader goal of addressing the challenges of developing secure cryptographic software and trying to catch potential vulnerabilities before software is deployed.

Cryptographic formal verification is a crucial aspect of CAC, utilizing mathematical methods to rigorously prove the correctness and security of cryptographic algorithms and protocols. By using dedicated tools and techniques, formal verification allows the establishement of rigorous security bounds for cryptographic primitives, thereby increasing the level of assurance that can be placed on concrete cryptographic implementations. If some implementation is verified using some formal verification technique, such implementation is typically considered *verified* or *high-assurance* since it has been mathematically proved, with the help of a machine, that the system is correct and matches the desired security properties.

When discussing formal verification, there are usually two approaches that can be followed: (i) the *automatic* approach, where specifications (or even concrete programs) are annotated with pre-conditions, post-conditions and invariants that are automatically analysed by some tool [Bla07, BCG+13]; and (ii) the *interactive* approach, where the user is responsible for guiding the proof, providing the appropriate tactics to discharge some proof goal [BGZB10, BLS20, BGHZ11, SHK+16, PM14]. The advantages of the automatic approach lie mostly in the automation provided by the underlying tools, that provide a faster pathway to establish some result. However, these tools tend to scale badly with the increase of the proof complexity or with the size of program being analysed. This is where interactive proof assistants are the better approach, since the user can guide the proof and provide *hints* that the interactive proof assistant can use to prove some goal. Interactive proof assistants lack the atractiveness of automatic tools, but they allow users to prove a wider range of proof goals, even if it means a bigger human/time effort.

EasyCrypt is an interactive proof assistant tailored for cryptography. It adopts the

code-based game-playing approach [Sho04, BR06, Hal05], in which primitives, security goals and hardness assumptions are expressed as probabilistic programs. The EasyCrypt proof system incorporates an ambient logic –for first-order and higher-order logic reasoning through a set of interactive proof tactics –and a probabilistic relational Hoare logic (pRHL) – for reasoning about programs written in a simple imperative language.

2.4 LPZK-DEGLOP23 overview

The goal of LPZK-DEGLOP23, as stated by its authors, was to explore new CAC approaches that could produce faster verified cryptographic implementations, reducing the formal verification penalty that was observed in other EasyCrypt-based CAC works.

The authors started by developing a *reference* LPZK specification in EasyCrypt, for which the completeness, soundness, and zero knowledge results were established. The idea behind this reference implementation was to reduce the proof effort as much as possible, without taking into account the performance of an extracted implementation. Then, optimization steps were iteratively introduced to the reference implementation to increase its performance. For each optimization, the authors proved that the resulting new implementation had the same *observable behavior* as the reference implementation, thus achieving the same level of security. For each version of the formalization, OCaml code was synthesized using an improved adaptation of the EasyCrypt to OCaml code extraction tool of [EP19].

Diving into technical details, every optimization step did not involve re-doing the security (completeness, soundness, and zero-knowledge) proofs. Instead, the security proof was done for the reference implementation, and each optimized version's security was reduced to the reference implementation security. This formalization choice allowed the authors to reduce the EasyCrypt proof effort, but it also puts a lot of emphasis on the reference implementation proof, implying that if the reference implementation proof was incorrect, then the rest of the formalization would also be incorrect. As we demonstrate in this paper, the reference implementation was vulnerable, implying that all the subsequent optimizations are also vulnerable.

3 Soundness of LPZK-DEGLOP23 Protocol

Our first observation is that the LPZK-DEGLOP23 implementation does not satisfy soundness, contradicting the corresponding claim in the paper [DEGL+23, Thm. 3.2]. The issue here is that the implemented verifier algorithm is incomplete, which allows a malicious prover to convince the honest verifier of *any* statement. The attack itself relies on an implementation bug in the LPZK-DEGLOP23 verifier. To explain our observation, we first recap the LPZK-DEGLOP23 version of the verifier algorithm and point out its shortcomings, then demonstrate how to exploit these flaws, and finally explain what the LPZK-DEGLOP23 "soundness" property really means (and why it allowed to establish "security" of a flawed implementation). The CCS version of the LPZK-DEGLOP23 implementation (i.e. without the fix) can be found here [rep25]. To explain our observation, we first recap the LPZK-DEGLOP23 version of the verifier algorithm and point out its shortcomings, then demonstrate how to exploit these flaws, and finally explain what the LPZK-DEGLOP23 "soundness" property really means (and why it allowed to establish "security" of a flawed implementation).

3.1 The LPZK-DEGLOP23 Verifier algorithm

We have already seen the LPZK-DIO21 verifier algorithm in Figure 2. As we will see, the LPZK-DEGLOP23 version deviates from it: It is missing the batched multiplication check

that ensures consistency between multiplication gates.

High-level version On input a circuit C and proof π, the LPZK-DEGLOP23 verifier algorithm performs the following checks:

1. Upon receiving π, the verifier first checks if π is a syntactically valid proof for C, i.e. if it is a proof for circuit C, including its topology.

2. Next, the verifier performs the computation in steps 1–3 of Figure 2, and encodes the resulting tags for each wire of the circuit in a recursive data structure. However, it does not perform the checks for each multiplication gate, from step 3.

3. It then checks if the commitment to the output wire of C is a commitment to zero (step 4 of Figure 2).

If any of the steps fails, the verifier rejects. Else, the verifier accepts.

EasyCrypt version For the sake of transparency, we include the EasyCrypt code snippet that captures the LPZK-DEGLOP23 verification algorithm (named `prove`) in Figure 3. `prove` takes as input the verifier's correlated randomness `r`, the public input `x`, as well as the proof from the prover, referred to as a commitment `c`.

```
1  op prove (r : verifier_rand_t) (x : verifier_input_t) (c : commitment_t) : bool =
2    let (z, z') = c in
3    if valid_z z (add_final_mul (fst x)) then
4      let f = gen_f r z in
5      let n = z' in
6      n ≠ fzero ∧ get_e f = fmul n r.‘alpha
7    else false.
```

Figure 3: Verifier proof checking algorithm

In line 2 of Figure 3, the verifier decomposes `c`, which is divided into the pair (z, z'), where z' contains the last field element, named $\hat{a}$ in Figure 1. Then, it checks that the message it received is consistent with the original circuit (which is also checked for validity), meaning that it will check whether the prover produced a commitment for the actual circuit being evaluated. This is done using the `valid_z` operator of line 3. If the commitment is valid, the verifier builds the `f` data structure using the `gen_f` operator of line 4. In short, `f` will hold the values of the tags q of Figure 2 for each circuit gate. Finally, in line 6, the verifier makes two checks: i. first, it checks that the value $\hat{\alpha}$ — represented by the `n` term in EasyCrypt— is different from zero; and ii. second, it performs the check given in line 4 of Figure 2 by asserting that the output wire tag $\hat{q}$ — represented by the `get_e f` term in EasyCrypt— is equal to $\hat{a}\alpha$ — represented by the `fmul n r.‘alpha` term in EasyCrypt.

3.2 Convincing the LPZK-DEGLOP23 Verifier of Any Statement

The flawed verifier algorithm, without the multiplication checks, can now be exploited in a straightforward way, allowing a malicious prover to convince the honest verifier of *any* statement. (We emphasise that this attack is specific to the LPZK-DEGLOP23 implementation and does not apply to the original LPZK protocol, nor other LPZK implementations.)

Attack idea Since the verifier only checks the final output wire, the prover does not need to commit to consistent wire values for the output of the multiplication gates. A simple way to construct a malicious prover that generates an accepting proof for any

circuit C and statement x — even if there exists no valid witness — is to just run the honest prover algorithm while modifying the commitment to the final multiplication gate, as follows (using notation as in Figure 1):

1. Choose an arbitrary "witness" w^*, e.g. $w^* = 1$.

2. Run the honest prover algorithm on C and w^* to obtain a "proof" π^*. Note that π^* would *not* be accepted by an honest verifier if $C(x, w^*) \neq 0$.

3. Let w be the output wire in the final multiplication gate of the circuit. Instead of committing to w, commit to $w - C(x, w^*)$ by replacing z_{3m+1} in the proof string π with $z^*_{3m+1} := z_{3m+1} + C(x, w^*)$, to obtain π^*.

At this point, π^* is a syntactically valid proof for (C, x) since π was. Moreover, the commitment to the last multiplication gate has been offset by $-C(x, w^*)$. Since the circuit output is obtained by (possibly) adding other wire values to the last multiplication, this leads to the commitment to the final output wire being a commitment to $C(x, w^*) - C(x, w^*) = 0$. Since the LPZK-DEGLOP23 verifier does not perform any additional checks on the remaining commitments in the proof or their consistency, the verifier will accept π^* as a valid proof for C and x.

Implementation Since the LPZK-DEGLOP23 verifier is ultimately compiled to an OCaml implementation, we were able to implement a malicious prover in OCaml and ran it against the LPZK-DEGLOP23 verifier algorithm. Figure 7 shows the simple modification to the honest prover algorithm, that simply adds one to the final commitment, meaning that the verifier will accept any statement where $C(x, w) = 1$ instead of $C(x, w) = 0$. This could easily be extended to make the verifier accept regardless of the output of $C(x, w)$, as detailed above.

```
let malicious_z = modify_z z {
  m = Z.add zi.m Z.one;
  m' = zi.m';
  c2 = zi.c2;
} in
let z' = let result = get_a1 r gg3 in (result) in (malicious_z, z')
```

Figure 4: Modification to z value in prover algorithm that leads to the attack on soundness. The helper function `modify_z` simply replaces the z values for one multiplication gate with the triple of values passed as input.

3.3 LPZK-DEGLOP23 Soundness Holds Only Against Restricted Provers

We now find ourselves in a situation where two seemingly contradictory statements hold: a formally verified soundness proof for LPZK exists, yet soundness was shown for a clearly insecure verifier algorithm. It is only natural to wonder what went wrong. In this case, the answer is both simple and subtle. While at first glance, the soundness notion as it was modeled in EasyCrypt appears to match the one typically used by cryptographers, on closer inspection, there is a fundamental difference in the modeled power of the malicious prover, compared with standard soundness notions, including that of LPZK-DIO21. Moreover, it does not match the notion presented in the body of the original LPZK-DEGLOP23 paper [DEGL+23].

The soundness game To understand the issue, we first need to understand how soundness is defined. Since EasyCrypt's proof system is based on game-hopping proofs, we first give an equivalent, game-based definition of soundness, in the style of pen-and-paper proofs.

Definition 2 (Game-based soundness)**.** Let (Dealer, Prove, Verify) be an LPZK protocol for a relation R. Let MP be a malicious prover who is given access to the ZK statement C, x, in the following experiment. (Note that the soundness game itself checks that $C(x, w) \neq 0$.)

Soundness((Dealer, Prove), MP, C, x)

1 : $(R_p, R_v) \leftarrow\$ \text{Dealer}(1^\lambda, C)$
2 : $\pi \leftarrow \text{MP}(R_p, C, x)$
3 : $b \leftarrow \text{Verify}(R_v, x, \pi)$
4 : **return** $\neg\mathcal{R}(C, x, w) \wedge b$

Then (Dealer, Prove, Verify) is considered *sound* if for all malicious provers MP and for all C and x,

$$\Pr[\text{Soundness}((\text{Dealer}, \text{Prove}), \text{MP}, C, x) = 1] \leq \mathsf{negl}(x).$$

Soundness lemma In EasyCrypt, the corresponding soundness lemma is written as follows.

```
lemma soundness : ∀ (MP <: MProver_t) (x : statement_t) (rp : prover_rand_t),
  Pr [ Soundness(RV, MP).main(rp, x) : res ] ≤ Q
```

This is to be read as: Let `RV` be the part of the dealer that generates the verifier's randomness (defined elsewhere in the file). Then for any malicious provers `MP`, statement `x`[2], and prover randomness `rp`, the probability that the soundness game against `MP` on input `rp` and `x` outputs `res` = 1 is less or equal than some `Q`. For technical reasons, the lemma also quantifies over memories `&m` which we omitted for the sake of readability.

The axiom So far, the soundness notion seems as expected. Upon further inspection, however, we found that the EasyCrypt formalization had added an axiom about the malicious prover that was not mentioned anywhere in the paper itself. The axiom in question restricts the class of malicious provers against the soundness game:

```
axiom mp_commit_exec (rp_ : prover_rand_t) (x_ : statement_t) :
     hoare [ MP.commit : rp_ = rp ∧ x_ = x ⇒
               ∃ w, res.`1 = gen_z rp_ (add_final_mul x_.`1).`gates x_.`2 w].
```

which can be found on line 530 at the LPZK.ec file at [rep25] [3].

Intuitively, this code snippet is a statement that relates the input of an arbitrary malicious prover to their output. The axiom itself is called `mp_commit_exec` and parameterized by `rp_` and `x_`. `MP` denotes the malicious prover. We don't know `MP`'s concrete implementation, but we know its interface: a single procedure `commit`, which (for an honest prover) implements the Prove algorithm. The keyword `hoare` indicates that this is a statement in Hoare logic, i.e. a statement about a single procedure (`MP.commit`) that relates the procedure's input to its output. The condition about the input is called precondition, the output condition is the postcondition, and the relation between the two is an implication.

[2] This includes the circuit C as well.

[3] https://github.com/SRI-CSL/high-assurance-crypto/blob/fa893a3ebf85081e43da3e0bc2d9ed7c1a558a3f/high-assurance-zk/lpzk/proof/LPZK/LPZK.ec#L530

The precondition is simple and more of a technicality: It states that the inputs `rp` and `x` to `MP.commit` are indeed the parameters `rp_` and `x_` of the axiom.

The postcondition is more involved. First, we need to know that the keyword `res` refers to the output of `MP.commit`, denoted by (z, z') in the rest of this work, and `res.`‘`1` refers to the first projection, i.e. z. Note that what is called `x_` in the code is a tuple (C, x) consisting of the circuit C and the public input x. Hence `x_.`‘`1` is the circuit C, and `x_.`‘`2` the public input x. The postcondition states that there must exist a witness `w` such that an equality holds: The left hand side is simply the output `res.`‘`1` of `MP.commit`. On the right hand side, `gen_z` is how the honest prover generates z, taking as input the prover randomness $rp_$, the circuit, public input, and the witness w. Finally, `add_final_mul` is an artifact of the LPZK-DEGLOP23 formalization that forces the circuit C to have a multiplication as final gate, by adding such a gate to the circuit.

In a style that is closer to pen-and-paper definitions, the axiom can be summarized as follows:

Axiom 1 (Malicious prover axiom ([DEGL+23], EasyCrypt code))**.** *For all malicious provers* `MP` *that output a proof* (z, z')*, there exists* w *such that* z *can be explained as running the honest prover algorithm with* w*.*

Note that the axiom does not require w to be a valid witness for `x`. It is sufficient for the proof to be the result of running the honest prover algorithm on *some* w.

The actual LPZK-DEGLOP23 soundness guarantee An axiom in EasyCrypt in this context behaves like a mathematical axiom, meaning that any subsequent lemma is shown in a world where the axiom holds. Given this axiom, LPZK-DEGLOP23 hence only showed soundness in a world where all provers follow the protocol specification, apart from possibly changing their prescribed input. This rules out the class of provers that violate consistency between commitments for different parts of the circuit, in particular, excluding our attacker from Section 3.2.

3.4 Fixing the EasyCrypt formalization

We end this section by briefly reporting on how the soundness proof was fixed. After we notified the authors of LPZK-DEGLOP23 of the missing check in the verifier algorithm and the issues around the malicious prover axiom Axiom 1, they added the check, removed the axiom and reproved soundness. The soundness proof gains an additional case for a malicious prover that does *not* respect the prescribed relations between internal gates. In this case, a malicious prover can only trick the verifier into accepting if they manage to forge the linear expression, which leads to the expected soundness error of $2/|\mathbb{F}|$.

4 Zero Knowledge of LPZK-DILO22 and LPZK-DEGLOP23 Protocols

Our second observation concerns the claim of achieving perfect zero knowledge for the tweaked LPZK of LPZK-DILO22 [DILO22a, Thm. 3.1]. As we will discuss, the LPZK-DILO22 construction as described in the paper does not achieve perfect zero knowledge. Since LPZK-DEGLOP23 claims to have a formally verified proof of perfect zero knowledge for this construction, we will also investigate this proof in more detail.

4.1 The Perfect Zero Knowledge Claim of LPZK-DILO22

We are going to revisit the LPZK-DILO22 security proof from the full version [DILO22b, Appendix A.2]. Recall that the LPZK-DILO22 model leaves the trusted dealer implicit. As

discussed in Section 2.2.1, the zero knowledge property is informally stated as follows:

Definition 3 (Perfect zero knowledge [DILO22a], informal)**.** There exists a PPT simulator $\mathcal{S}$ such that for any circuit C with public input x, any witness w such that $C(x, w) = 0$, and any $\alpha \in \mathbb{F}$, $\mathcal{S}(C, \alpha)$ is distributed identically to $\vec{v}(\alpha)$, where $\vec{v}$ is produced by the prover algorithm.

The security proof then needs to construct a simulator that simulates $\vec{v}(\alpha)$, without access to the prover's correlated randomness $(\vec{a}, \vec{b})$. The LPZK-DILO22 simulator simply samples α and $\vec{v}$ uniformly at random[4]. However, remember that LPZK-DILO22 encodes the witness in the $\vec{b}$ parts of the line $\vec{v}(x) = \vec{a}x + \vec{b}$. If α happens to be 0, then $\vec{v}$ contains the witness w in the clear. The simulator cannot simulate this situation. Moreover, if we consider security against a malicious verifier who can choose α (as indicated by the universally quantified α), then the verifier can now force the prover to leak the witness!

Discussion As we have seen above, the LPZK-DILO22 construction does not achieve perfect zero knowledge as is. The security model considers a malicious verifier that can choose α (since α is universally quantified), meaning they can always choose $\alpha = 0$ and learn the witness. A possible fix would be a relaxation to honest verifier zero knowledge where α is chosen uniformly at random. In that case, the zero knowledge guarantee would be statistical. Alternatively, one could work with a trusted dealer that excludes the case $\alpha = 0$, which would slightly increase the soundness error to $2/(|\mathbb{F}| - 1)$.

4.2 The Perfect Zero Knowledge Claim of LPZK-DEGLOP23

Now that we know that the LPZK-DILO22 zero knowledge guarantee is flawed, how is it possible for LPZK-DEGLOP23 to have a formally verified proof of perfect zero knowledge for a similar LPZK version? Like in the previous section, the answer is a mismatch between the intended zero knowledge notion of LPZK-DILO22 and what was formalized by LPZK-DEGLOP23.

To understand this modeling issue, we need to understand that LPZK-DEGLOP23 modeled the zero knowledge property as a hybrid between the two approaches we discussed in Section 2.2.1: While LPZK-DILO22 worked with an implicit dealer that is baked into the zero knowledge notion, the LPZK-DEGLOP23 model tries to make the dealer explicit. But instead of working with a single fully separate dealer as in our Definition 1, the authors went with a "split" approach, where the dealer parts generating the prover and verifier randomness appear in different places of the zero knowledge game.

The Zero Knowledge game LPZK-DEGLOP23 define the zero knowledge property in EasyCrypt as the game shown in Figure 5, where a distinguisher `D` is asked to distinguish a real proof (generated by the honest prover algorithm) from an ideal proof (generated by a simulator). That is, in contrast to the definitions discussed in Section 2.2.1, the LPZK-DEGLOP23 model introduces an explicit distinguisher algorithm.

The `ZK` game is parameterized by the distinguisher `D`, a prover randomness generator `RP` and an "evaluator" `E`. There will be two evaluators: A real one that runs the honest prover algorithm to generate a proof `ctr` and forwards it to the malicious verifier (without generating any correlated randomness for the verifier!). And an ideal one that instead asks the simulator to generate the proof `ctr`. The game starts by sampling honest correlated randomness `rp` for the prover using algorithm `RP.gen` (line 12), before calling the evaluator to obtain the proof `ctr` which is then provided to the distinguisher (line 13). The distinguisher takes as input the statement `x` and witness `w`, as well as the proof `ctr`

[4]We assume that $\mathcal{S}$ sampling α is a typo. The described proof strategy works just as well when the simulator receives α as input.

that the prover outputs, and returns a bit (line 14). The crux here is that the proof `ctr` that the distinguisher sees does not contain any of the correlated randomness (denoted by `rp` and `rv`, respectively). As a result, the simulator is not faced with the impossible task of simulating the witness when $\alpha = 0$.

```
1  module type Distinguisher_t = {
2    proc guess(_ : witness_t * statement_t * trace_t option) : bool
3  }.
4
5  module type Evaluator_t = {
6    proc eval(w : witness_t, x : statement_t, rp : prover_rand_t) : trace_t option
7  }
8
9  module ZK (D : Distinguisher_t) (RP : RandP_t) (E : Evaluator_t) = {
10   proc main(w : witness_t, x : statement_t) : bool = {
11     var ctr, rp, b';
12     rp <@ RP.gen(w,x);
13     ctr <@ E.eval(w,x,rp);
14     b' <@ D.guess(w,x,ctr);
15     return b';
16   }
17   }.
```

Figure 5: Zero knowledge security game for DVNIZK protocols

Discussion We can only speculate, but is seems plausible that this flawed definition is the result of a translation error: When translating the LPZK-DILO22 model without an explicit dealer into a version *with* an explicit dealer, the simulator's output needs to be adjusted. In that case, the distinguisher needs to be given not only the proof π (called `ctr` here), but also the verifier's correlated randomness $(\alpha, \vec{v})$ generated by the trusted dealer.

5 Zero Knowledge of the LPZK-DEGLOP23 OCaml Implementation

So far, we have only looked into the EasyCrypt model and implementation, since any implementation flaw on the EasyCrypt side will also show up in the extracted OCaml implementation. Our third observation is about the OCaml implementation of the verifier itself that was generated from the EasyCrypt model, or more generally about limitations of the workflow that produced that code. What we found is that the equivalent of the verifier code, the decide function, is a strange hybrid of the verifier and the evaluator (i.e. part of the trusted dealer) from the EasyCrypt model. In particular, decide gets as input the prover randomness rp. From here, it is trivial to extract the witness that was used since the randomness encodes the witness. This flaw can in principle be seen as an attack, though we think it is more appropriate to consider it as evidence of the pitfalls of the workflow: It indicates that working with a trusted dealer in this context can be tricky.

5.1 The LPZK-DEGLOP23 Verifier Algorithm, Extracted to OCaml

The authors define a wrapper that invokes LPZK code extracted from the EasyCrypt model discussed in the previous sections [5]. This wrapper discloses two procedures, corresponding to the prover and verifier, respectively: i. commitment: that takes as input the witness and the statement, runs the LPZK prover and outputs the proof; and ii. decide: that takes as input the statement `statement`, the proof `cs` produced by commitment and the

[5] https://github.com/SRI-CSL/high-assurance-crypto/blob/main/high-assurance-zk/lpzk/extraction/array/src/lpzk_exec.ml

prover randomness `rp`, generates the verifier correlated randomness `rv` and then invokes the LPZK verifier to accept or reject the proof. Figure 6 shows the relevant sections of decide. The part of the trusted dealer that generates the verifier's randomness is partly folded into the verifier. If one were to execute decide as if it was a verifier implementation, then it would be trivial to extract the witness from `rp`.

```
let decide statement cs rp =
    (*...*)
  Format.printf "Generating␣verifier␣randomness...@." ;
  let rv = get_verifier_rand topo !q rp in
    (*...*)
  Format.printf "Checking␣proof...@." ;
    (* check proof using rv *)
    (*...*)
```

Figure 6: Verifier implementation in OCaml.

5.2 Extracting the Witness from the Verifier's Input

For completeness, we include our code for extracting the witness from the input to decide in Figure 7. The idea behind the "attack" is straighforward: given the commitment `z` and the randomness that originated it `u`, we can transverse the commitment structure and, when we find a commitment for a secret input gate, we compute the original witness by doing $w \leftarrow u[wid].b + c.m$, where wid is the wire ID, b is a projection to the random value and m is a projection to the commitment data, as specified by the LPZK code.

```
let w = ref Array.make w_len Z.zero

let rec reverse_gen_z (u: ui_t array) (gg: gates_t) (inst: Z.t array) (z: z_t) : Z.t
     array =
  match z with
  | PInputZ _ -> ()
  | SInputZ (wid, c) ->
    let b = nth def_ui1 u wid in
    xs.(wid) <- c.‘m + b
  | ConstantZ _ -> ()
  | AdditionZ (_, _, zl, zr) ->
    reverse_gen_z u gg xp zl ; reverse_gen_z u wr xp zr
  | MultiplicationZ (_, _, zl, zr) ->
    reverse_gen_z u gg xp zl ; reverse_gen_z u gg xp zr
```

Figure 7: Zero knowledge attack by a malicious verifier

5.3 Insufficient Separation of Trusted Dealer and Verifier

Again, we can ask how such an obviously flawed verifier implementation can have a formally verified security proof. In this case, the answer is simple: In the EasyCrypt model, the verifier and its randomness generation are treated as separate entities, as they should. For whatever reason though, the OCaml implementation combined the two into one algorithm, meaning there is no separate verifier algorithm anymore.

6 Discussion

We end the paper with a discussion of lessons learned over the course of this work.

Modeling issues in formal proofs, proof issues in pen-and-paper proofs. This work uncovered both modeling and proof issues. All the issues we found with LPZK-DEGLOP23 can be traced back to modeling issues. The only exception we found in this work is the pen-and-paper proof of the zero knowledge property of LPZK-DILO22. We remark that the corresponding EasyCrypt proof had the potential to catch the missing case of $\alpha = 0$ described in Section 4, had it not made a modeling mistake itself.

In summary, we saw a neat divide though between proof issues in the pen-and-paper proofs, and (solely) modeling issues in the formal proofs. This is not surprising, as formal proofs in proof assistants like EasyCrypt are meant to increase trust in the proofs themselves. The only part that is left to be checked by humans (and hence the primary source of mistakes) are then the security models.

Gaps between pen-and-paper and formal models The biggest source of issues was clearly the signifant gaps between the pen-and-paper LPZK models and what was intended to be their EasyCrypt counterparts. We found gaps in the soundness and zero knowledge notions (Sections 3.3 and 4.2) as well as the verifier algorithm itself (Section 3.1). The latter was probably a bug rather than a conscious choice though.

For the future, we suggest to make the mapping between pen-and-paper and formal model even more explicit, and to document all choices that deviate from the model on paper. Formal proofs will inevitably have to fill in gaps left on paper, in proofs and in models, and we acknowledge that it can be a challenge to distinguish between filling in a modeling gap and making a deliberate modeling choice that deviates from the original model. Nevertheless, documenting everything will help readers and reviewers to spot issues. For example, the LPZK-DEGLOP23 dealer modeling differs drastically from that of LPZK-DILO22. However, this could have been (partially) a reaction to filling in a perceived gap in the pen-and-paper modeling, where the dealer is not explicitly defined.

Gaps between code description and actual code In Section 3, we saw the impact that an undisclosed axiom can have on the formalized soundness notion. More generally, there can be significant gaps between the actual code and how it is presented in a paper. While this is in principle a known issue, it is unclear how to best address it: Since it is on the reviewers to ultimately decide about a submission, it makes sense to ask the reviewers to inspect the code. However, reviewers are typically not obliged to review any supplementary material beyond the actual submission text. Adding such a requirement would add a substantial burden to the already high review load for a PC member. It might then be tempting to assign responsibility to the artifact evaluation process that more and more conferences implement. However, this process in its current form is definitely not suitable either, as artifact evaluation happens only *after* acceptance.

Reading proofs vs reading models This final claim is only backed by anecdotal evidence: We found that the EasyCrypt code (not the proofs) was decently understandable to someone with programming experience, but who is not familiar with EasyCrypt, if that person got a bit of support.

Acknowledgments

The authors would like to thank the anonymous CiC reviewers for the interesting discussion about review practices and where the responsibility for inspecting proof code lies.

The authors also thank Sam Dittmer, Steve Lu and Rafail Ostrovsky for helpful discussions regarding the soundness flaw and the subsequent fix.

References

[AGM18] Shashank Agrawal, Chaya Ganesh, and Payman Mohassel. Non-interactive zero-knowledge proofs for composite statements. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part III*, volume 10993 of *Lecture Notes in Computer Science*, pages 643–673. Springer, Cham, August 2018. doi:10.1007/978-3-319-96878-0_22.

[AKZT23] Anish Athalye, M. Frans Kaashoek, Nickolai Zeldovich, and Joseph Tassarotti. Leakage models are a leaky abstraction: the case for cycle-level verification of constant-time cryptography. In *Programming Languages for Architecture (PLARCH), co-located with PLDI 2023*, June 2023. Preprint available at MIT CSAIL PDOS. URL: https://people.csail.mit.edu/nickolai/papers/athalye-chroniton.pdf, doi:10.1145/3585000.3585008.

[BBB+19] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. Cryptology ePrint Archive, Report 2019/1393, 2019. URL: https://eprint.iacr.org/2019/1393.

[BBB+21] 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*, pages 777–795. IEEE Computer Society Press, May 2021. doi:10.1109/SP40001.2021.00008.

[BCG+13] Gilles Barthe, Juan Manuel Crespo, Benjamin Grégoire, César Kunz, Yassine Lakhnech, Benedikt Schmidt, and Santiago Zanella Béguelin. Fully automated analysis of padding-based encryption in the computational model. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013: 20th Conference on Computer and Communications Security*, pages 1247–1260. ACM Press, November 2013. doi:10.1145/2508859.2516663.

[BCHO22] Jonathan Bootle, Alessandro Chiesa, Yuncong Hu, and Michele Orrù. Gemini: Elastic SNARKs for diverse environments. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 427–457. Springer, Cham, May / June 2022. doi:10.1007/978-3-031-07085-3_15.

[BCTV14] Eli Ben-Sasson, Alessandro Chiesa, Eran Tromer, and Madars Virza. Succinct non-interactive zero knowledge for a von neumann architecture. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014: 23rd USENIX Security Symposium*, pages 781–796. USENIX Association, August 2014. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/ben-sasson.

[BDF+18] Chris Brzuska, Antoine Delignat-Lavaud, Cédric Fournet, Konrad Kohbrok, and Markulf Kohlweiss. State separation for code-based game-playing proofs. In Thomas Peyrin and Steven Galbraith, editors, *Advances in Cryptology – ASIACRYPT 2018, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 222–249. Springer, Cham, December 2018. doi:10.1007/978-3-030-03332-3_9.

[BGHZ11] Gilles Barthe, Benjamin Grégoire, Sylvain Heraud, and Santiago Zanella Béguelin. Computer-aided security proofs for the working cryptographer. In Phillip Rogaway, editor, *Advances in Cryptology – CRYPTO 2011*, volume

6841 of *Lecture Notes in Computer Science*, pages 71–90. Springer, Berlin, Heidelberg, August 2011. doi:10.1007/978-3-642-22792-9_5.

[BGZB10] Gilles Barthe, Benjamin Grégoire, and Santiago Zanella Béguelin. Programming language techniques for cryptographic proofs. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving*, pages 115–130, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. doi:10.1007/978-3-642-14052-5_10.

[BHK15] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of IND-CCA: When and how should challenge decryption be disallowed? *Journal of Cryptology*, 28(1):29–48, January 2015. doi:10.1007/s00145-013-9167-4.

[Bla07] Bruno Blanchet. Computationally sound mechanized proofs of correspondence assertions. In Andrei Sabelfeld, editor, *CSF 2007: IEEE 20th Computer Security Foundations Symposium*, pages 97–111. IEEE Computer Society Press, 2007. doi:10.1109/CSF.2007.16.

[BLS20] David A. Basin, Andreas Lochbihler, and S. Reza Sefidgar. CryptHOL: Game-based proofs in higher-order logic. *Journal of Cryptology*, 33(2):494–566, April 2020. doi:10.1007/s00145-019-09341-z.

[BPW12] David Bernhard, Olivier Pereira, and Bogdan Warinschi. How not to prove yourself: Pitfalls of the Fiat-Shamir heuristic and applications to Helios. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology – ASIACRYPT 2012*, volume 7658 of *Lecture Notes in Computer Science*, pages 626–643. Springer, Berlin, Heidelberg, December 2012. doi:10.1007/978-3-642-34961-4_38.

[BR06] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *Advances in Cryptology – EUROCRYPT 2006*, volume 4004 of *Lecture Notes in Computer Science*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006. doi:10.1007/11761679_25.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145. IEEE Computer Society Press, October 2001. doi:10.1109/SFCS.2001.959888.

[CLR24] Sofía Celi, Shai Levin, and Joe Rowell. CDLS: Proving knowledge of committed discrete logarithms with soundness. In Serge Vaudenay and Christophe Petit, editors, *AFRICACRYPT 24: 15th International Conference on Cryptology in Africa*, volume 14861 of *Lecture Notes in Computer Science*, pages 69–93. Springer, Cham, July 2024. doi:10.1007/978-3-031-64381-1_4.

[DEGL+23] Samuel Dittmer, Karim Eldefrawy, Stéphane Graham-Lengrand, Steve Lu, Rafail Ostrovsky, and Vitor Pereira. Boosting the performance of high-assurance cryptography: Parallel execution and optimizing memory access in formally-verified line-point zero-knowledge. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 2098–2112. ACM Press, November 2023. doi:10.1145/3576915.3616583.

[DILO22a] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 829–841. ACM Press, November 2022. doi:10.1145/3548606.3559385.

[DILO22b] Samuel Dittmer, Yuval Ishai, Steve Lu, and Rafail Ostrovsky. Improving line-point zero knowledge: Two multiplications for the price of one. Cryptology ePrint Archive, Report 2022/552, 2022. URL: https://eprint.iacr.org/2022/552.

[DIO21] Samuel Dittmer, Yuval Ishai, and Rafail Ostrovsky. Line-point zero knowledge and its applications. In Stefano Tessaro, editor, *ITC 2021: 2nd Conference on Information-Theoretic Cryptography*, volume 199 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:24. Schloss Dagstuhl, July 2021. doi:10.4230/LIPIcs.ITC.2021.5.

[DMWG23] Quang Dao, Jim Miller, Opal Wright, and Paul Grubbs. Weak fiat-shamir attacks on modern proof systems. In *2023 IEEE Symposium on Security and Privacy*, pages 199–216. IEEE Computer Society Press, May 2023. doi:10.1109/SP46215.2023.10179408.

[EP19] Karim Eldefrawy and Vitor Pereira. A high-assurance evaluator for machine-checked secure multiparty computation. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019: 26th Conference on Computer and Communications Security*, pages 851–868. ACM Press, November 2019. doi:10.1145/3319535.3354205.

[EPG+19] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. Simple high-level code for cryptographic arithmetic - with proofs, without compromises. In *2019 IEEE Symposium on Security and Privacy*, pages 1202–1219. IEEE Computer Society Press, May 2019. doi:10.1109/SP.2019.00005.

[FLM22] Armando Faz-Hernández, Watson Ladd, and Deepak Maram. ZKAttest: Ring and group signatures for existing ECDSA keys. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021*, volume 13203 of *Lecture Notes in Computer Science*, pages 68–83. Springer, Cham, September / October 2022. doi:10.1007/978-3-030-99277-4_4.

[Gab19] Ariel Gabizon. On the security of the BCTV pinocchio zk-SNARK variant. Cryptology ePrint Archive, Report 2019/119, 2019. URL: https://eprint.iacr.org/2019/119.

[GL25] Lydia Garms and Michael Livesey. Attacking soundness for an optimization of the gemini polynomial commitment scheme. Cryptology ePrint Archive, Report 2025/565, 2025. URL: https://eprint.iacr.org/2025/565.

[Hal05] Shai Halevi. A plausible approach to computer-aided cryptographic proofs. Cryptology ePrint Archive, Report 2005/181, 2005. URL: https://eprint.iacr.org/2005/181.

[HBD+20] Andreas Hülsing, Daniel J. Bernstein, Christoph Dobraunig, Maria Eichlseder, Scott Fluhrer, Stefan-Lukas Gazdag, Panos Kampanakis, Stefan Kölbl, Tanja Lange, Martin M. Lauridsen, Florian Mendel, Ruben Niederhagen, Christian Rechberger, Joost Rijneveld, Peter Schwabe, Jean-Philippe Aumasson, Bas

Westerbaan, and Ward Beullens. SPHINCS$^+$. Technical report, National Institute of Standards and Technology, 2020. available at https://csrc.nist.gov/projects/post-quantum-cryptography/post-quantum-cryptography-standardization/round-3-submissions.

[HLPT20] Thomas Haines, Sarah Jamie Lewis, Olivier Pereira, and Vanessa Teague. How not to prove your election outcome. In *2020 IEEE Symposium on Security and Privacy*, pages 644–660. IEEE Computer Society Press, May 2020. doi:10.1109/SP40000.2020.00048.

[KKF20] Mikhail Kudinov, Evgeniy Kiktenko, and Aleksey Fedorov. Round 3 official comment: Sphincs+. NIST PQC Forum (via pqc-forum@nist.gov), July 2020. Comment on SPHINCS+ security proof, available as PDF at NIST CSRC. URL: https://csrc.nist.gov/CSRC/media/Projects/post-quantum-cryptography/documents/round-3/official-comments/Sphincs-Plus-round3-official-comment.pdf.

[KM06] Neal Koblitz and Alfred Menezes. Another look at "provable security". II. (invited talk). In Rana Barua and Tanja Lange, editors, *Progress in Cryptology - INDOCRYPT 2006: 7th International Conference in Cryptology in India*, volume 4329 of *Lecture Notes in Computer Science*, pages 148–175. Springer, Berlin, Heidelberg, December 2006. doi:10.1007/11941378_12.

[KM19] Neal Koblitz and Alfred Menezes. Critical perspectives on provable security: Fifteen years of "another look" papers. Cryptology ePrint Archive, Report 2019/1336, 2019. URL: https://eprint.iacr.org/2019/1336.

[Mau10] Ueli Maurer. Constructive cryptography - a primer (invited paper). In Radu Sion, editor, *FC 2010: 14th International Conference on Financial Cryptography and Data Security*, volume 6052 of *Lecture Notes in Computer Science*, page 1. Springer, Berlin, Heidelberg, January 2010. doi:10.1007/978-3-642-14577-3_1.

[miT25] miTLS. A zoo of tls attacks. https://www.mitls.org/pages/attacks, 2025. Accessed: 2025-07-07.

[PBP+20] Marina Polubelova, Karthikeyan Bhargavan, Jonathan Protzenko, Benjamin Beurdouche, Aymeric Fromherz, Natalia Kulatova, and Santiago Zanella-Béguelin. HACLxN: Verified generic SIMD crypto (for all your favourite platforms). In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 899–918. ACM Press, November 2020. doi:10.1145/3372297.3423352.

[PHGR13] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE Computer Society Press, May 2013. doi:10.1109/SP.2013.47.

[PM14] Adam Petcher and Greg Morrisett. The foundational cryptography framework. *CoRR*, abs/1410.3735, 2014. URL: http://arxiv.org/abs/1410.3735, arXiv:1410.3735.

[rep25] SRI-CSL/high-assurance-crypto repository. Commit 'fa893a3ebf85081e43da3e0bc2d9ed7c1a558a3f'. https://github.com/SRI-CSL/high-assurance-crypto/tree/fa893a3ebf85081e43da3e0bc2d

9ed7c1a558a3f/high-assurance-zk/lpzk/proof, July 2025. Accessed: 2025-07-07.

[SHK+16] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in f*. *SIGPLAN Not.*, 51(1):256–270, January 2016. doi:10.1145/2914770.2837655.

[Sho04] Victor Shoup. Sequences of games: a tool for taming complexity in security proofs. Cryptology ePrint Archive, Report 2004/332, 2004. URL: https://eprint.iacr.org/2004/332.

[ZBPB17] Jean Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. HACL*: A verified modern cryptographic library. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1789–1806. ACM Press, October / November 2017. doi:10.1145/3133956.3134043.

[ZKP22] ZKProof Community. Zkproof community reference. Technical report, ZKProof, July 2022. Accessed: 2025-07-07. URL: https://docs.zkproof.org/reference.pdf.