Zero-knowledge proofs of software vulnerabilities
or
How to prove I found a vulnerability without revealing it

Daniel Augot
LIX (Cryptography team at École polytechnique)

1 Disclosure of vulnerabilities and zero-knowledge

In the case of a *bug bounty*, the *white hat* programmer who discovers an attack and discloses it risks gaining no benefit from the *bug bounty* unless they make it public. But this would endanger the target.

Zero-knowledge proofs are an advanced cryptographic concept that allows one to prove that a statement, fact, claim, or computation result is true, without revealing any of the elements that make it true. In the context of *bug bounties*, the statement is "I know a software vulnerability in such and such software, here are its effects." A zero-knowledge proof of this claim will be made, demonstrating the effect without revealing the vulnerability.

The underlying technology is that of *ZK-snarks*, which have recently made significant progress in maturity and speed to reach this level of abstraction, generality, and performance. This is due to the enthusiasm of the blockchain world for these technologies, which are heavily funded and have seen an explosion of both scientific and software developments (references and pointers here). This internship does not deal with the cryptographic internals of snarks.

Snarks use advanced cryptography (elliptic curves, error-correcting codes, etc.), but software infrastructures like RISC-0 allow non-specialist programmers to use these technologies.

2 RISC-0

The RISC-0 software provides a host virtual machine that supports the execution of a guest RISC-V binary in ELF format. The RISC-0 software allows to

1. provide a *cryptographic proof* of the correct execution of the binary. This proof can then be verified by a cryptographic program called a verifier. This verification requires much fewer resources than direct execution. Anyone can verify it.
2. hide (*blind*) certain elements of the execution, which thus cannot be deduced from the proof.

3 CHESS Example

The chess example (`github`) below clearly shows what happens. Here is the guest program:

```
use chess_core::Inputs;
use risc0_zkvm::guest::env;
use shakmaty::{CastlingMode, Chess,
       FromSetup, Move, Position,
       Setup, fen::Fen, san::San};
fn main() {
    let inputs: Inputs = env::read();
    let mv: String = inputs.mv;
    let initial_state: String = inputs.board;
    env::commit(&initial_state);
    let setup = Setup::from(Fen::from_ascii(initial_state.as_bytes()).unwrap());
    let pos = Chess::from_setup(setup, CastlingMode::Standard).unwrap();
    let mv: Move = mv.parse::<San>().unwrap().to_move(&pos).unwrap();
    let pos = pos.play(&mv).unwrap();
    assert!(pos.is_checkmate());
}
```

Note that this is a standard Rust program.

We see the call `let inputs: Inputs = env::read();` which defines the state of the chessboard before checkmate, and the sequence of moves to play `mv`. This program is executed as a guest by the RISC0 VM, which provides its *inputs* (`initial_state`, which will be public, and `mv`, which will be private).

What is public is determined by the line `env::commit(&initial_state);` which allows writing to an execution *journal*. Only what is written in the journal will become public; all other data will remain hidden by default.

Then, after execution, a *receipt* is written (by the host), which contains the journal (thus `initial_state`) and the zero-knowledge "snark" proof. This proof can then be verified *a posteriori* by a verifier program (also provided by RISC-0).

In this example, we see that RISC-0 provides a whole *framework* and APIs that allow the programmer to ignore the underlying cryptographic techniques, and standard rust libraries like chess can be used with no effort.

4 Proof of smart contract vulnerabilities

RISC-0 has been used to prove the existence of vulnerabilities in Ethereum *smart contracts*. A *smart contract* is a program recorded on a blockchain, which can be called by an Ethereum transaction. These programs can have vulnerabilities that certain transactions can trigger.

With RISC0, it has been demonstrated that it is possible to prove the execution of a smart contract to an undesirable state, following a transaction and another attacking *smart contract*. The zero-knowledge property allows proving that this state is reached without revealing the transaction that triggers the attack.

See this blog post and the github repository. In this example, a reentrancy bug is exploited.

5 Internship goal: “de-blockchainize” the subject, study feasibility and performance

Here, the goal is to create an example of a vulnerability proof for a much more standard, non-blockchain context.

The vulnerability will be proven on a “toy” example, chosen by the intern, but must remain simple. For example, initially, one could draw inspiration from this blog post, which describes a *buffer overflow* in Rust, the native language of RISC-0.

A second step will be to produce a vulnerability proof for an ELF program written in a language other than Rust, such as C. Then, more realistic programs will be attempted. This idea of cryptographic vulnerability proofs has led to publications [1, 3, 2], but it seems that RISC-0 is more *programmer friendly*.

References

[1] Santiago Cuéllar, Bill Harris, James Parker, Stuart Pernsteiner, and Eran Tromer. “Cheesecloth: Zero-Knowledge Proofs of Real World Vulnerabilities”. In: *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023.* Ed. by Joseph A. Calandrino and Carmela Troncoso. USENIX Association, 2023, pp. 6525–6540. URL: `https://www.usenix.org/conference/usenixsecurity23/presentation/cuellar`.

[2] Santiago Cuéllar Gempeler, Bill Harris, James Parker, Stuart Pernsteiner, Ian Sweet, and Eran Tromer. “Cheesecloth: Zero-Knowledge Proofs of Real-World Vulnerabilities”. In: *ACM Trans. Priv. Sec.* (4 Sept. 2025), p. 35.

[3] Xueyan Tang, Lingzhi Shi, Xun Wang, Kyle Charbonnet, Shixiang Tang, and Shixiao Sun. *Zero-Knowledge Proof Vulnerability Analysis and Security Auditing.* Cryptology ePrint Archive, Paper 2024/514. 2024. URL: `https://eprint.iacr.org/2024/514`.