zkMIPS: What “Security” Means for Our zkVM’s Proofs (Part 2)

Now that we have described the broader questions of ZK proofs security in Part 1, let’s continue with Question 2.

In the analysis of a two-party protocol, there are three cases to be considered:

Case A: what happens if both the Prover and Verifier are honest? Case B: what happens if the Verifier is honest but the Prover isn’t? Case C: what happens if the Prover is honest but the Verifier isn’t?

(Obviously, we are not interested in the fourth potential case, as there is no honest party left whose security interests need to be protected; we don’t care what dishonest parties might do.)

Case A: completeness

The definition of completeness of a proof of knowledge corresponds to Case A, where both parties are honest. If indeed y=F(x), then we want the protocol to succeed ‘always’. This isn’t really a security condition, it’s just a way of saying that the protocol correctly implements what it is supposed to do: if parties are honest, everything works as it should. It is the same as saying in an identification protocol that the false rejection rate is 0, i.e. a legitimate user won’t get rejected.

Case B: soundness

The soundness of a protocol corresponds to Case B: if the Prover tries to fool the Verifier into believing that y’=F(x) when in reality this is false, then the Verifier should detect this with probability 1; that is, the Prover ‘never’ succeeds in fooling the Verifier.

[

ZKM - ANGEL DEVIL 2 (3)

1920×1080 181 KB

](https://ethresear.ch/uploads/default/original/2X/8/8a604c826f1145952271c6222052502825d2b9b6.jpeg)

Prover dishonest, Verifier honest

Cryptographers sometimes use terms such as ‘probability 1’, ‘always’ and ‘never’ in peculiar ways. Almost all practical cryptographic systems make some use of probabilism and allow for an astronomically small probability of error. So in the context of soundness, ‘always’ stands for probability 1-eps and ‘never’ for probability 0+eps, where eps = 2^ -100 or smaller. As an indication of how improbable this is, the chances of winning the Mega Millions jackpot once is about 2 ^-28.17, so a probability of 2 ^-90 is less than winning this jackpot three times in a row. Very unlikely indeed.

Case C: privacy

This case is the most interesting one. What does it mean for the Verifier to be dishonest in a proof of knowledge? In other words, what is the Prover’s interest in whether or not the Verifier might try to violate? Well, this depends on context.

The original notion of zero knowledge, as introduced by Goldwasser, Micali, and Rackoff in 1985, is a very specific definition of privacy, meaning that the Verifier cannot obtain useful computational information about the Prover’s witness T through the protocol transcript, defined as the set of messages exchanged. (To be more precise, the transcript should be simulatable.)

However over the past few years, people have started to use zero knowledge to refer to succinct proofs for verifiable computation. Strictly speaking, this is an abuse of terminology, but such is the power of buzzwords that it is no use preaching against them.

How does this translate to the specific setting of ZK rollups? Well, the Verifier is simply glad that the Prover computed y = F(x) for him, while the Prover has no particular reason to keep the execution trace T secret. The important part is that the result of the outsourced computation is correct. The manner in which the Prover actually found the answer y (using T) is not something that needs to be kept a secret.

In other words, ZK Rollups are an example of a setting with a focus on succinctness, not on privacy. All transactions are to be publicly posted on Layer 1 anyway; so there are no data that need to be kept confidential. Often, the Prover might as well publish T (since T is huge anyway), so the Verifier might not even be able to read and process it all. That’s exactly where the succinct proof Z comes in, to make the reading and processing by the Verifier feasible.

However, there are more general settings, beyond verifiably outsourced computation, in which the Prover might wish to keep some algorithm private. For instance, suppose the Prover claims he has developed a novel, very efficient algorithm F for some very difficult computational problem, and wants to convince the Verifier that this is indeed the case.

In this setting the Prover may wish to generate a proof which isn’t only succinct, but which also keeps F and T private. The Prover may actually choose to keep the output y private, thus convincing the Verifier that it knows an answer, without actually showing what it is. Zero knowledge can be mind-boggling sometimes.

So to summarize, the term zero knowledge can actually mean three different things:

(1) a private proof (its original meaning),

(2) a succinct proof (its popularized meaning), and

(3) a combination of both.

Indeed, there is a subtle distinction between SNARKs and zkSNARKs: the first corresponds to interpretation (2), whereas the second corresponds to interpretation (3). Same thing for STARKs and zkSTARKs.

Total transparency and total auditability

One may ask: how about third parties? To what extent are outside observers allowed to monitor the Prover and Verifier? Do the Prover and Verifier have to encrypt their communication?

The answer is simple. In the context of ZK Rollups and Verifiable Computation, there are no secrets. Everything we do is public for everyone to see, verify, and reproduce. No encryptions, no secrets. Which implies a very strong property: total transparency, total auditability.

Message Integrity and software correctness

But transparency still means we need to be vigilant against malicious modifications of messages and software.

As far as message integrity is concerned, it is simply good practice to use digital signatures on each and every communication, to guarantee their origin and contents, as well as to be able to attribute responsibilities. So that solves the messaging part.

A second concern is the correctness of the software. When looking at a protocol from a cryptographic point of view, we usually assume that no errors occurred during implementation of the software components. But we know that in practice such errors are very common. For instance, recall that Edward Snowden’s leaks showed the world that the underlying cryptography of a system almost never gets broken; the problems are the software vulnerabilities, bugs, and errors. Therefore ZKM is planning a severe audit of its source code before the system is put into production.

Focus on the Verifier only

From a cryptographic perspective we need only focus on the correctness of the Verifier for this audit, to avoid a situation where an incorrect proof gets accepted by mistake.

Software errors in the Prover might lead to less-than-perfect completeness, which is undesirable but not catastrophic. That’s because implementation errors will probably cause errors in the message the Prover sends to the Verifier, leading the latter to reject the proof. Or these errors might lead to loss of privacy (confidentiality), which isn’t really an issue in our setting, as we saw before.

So we see that, to guarantee the soundness property of our protocol, we only need to focus on the Verifier being correctly implemented. That’s good news for other settings, in which we don’t even have control over the Prover. (This is a topic for a future post.)

As a final point: note that we did not discuss the Fiat-Shamir transformation to render the final protocol non-interactive. This transformation makes the validity of our security analysis easier to understand, but is complicated enough to also deserve a separate post. Stay tuned!

—

Want to discuss this article and other ZK-related topics with our core ZKM team? Join our Discord channel: discord.gg/Bck7hdGcns

Zkm