

Assessing and Benchmarking zkVMs: Insights into Performance and Scalability

Lawrence Lim, Nihar Shah

2024-05-13

Presentation Structure

- ▶ Introduction
 - ▶ What is a zkVM?
 - ▶ Table 1 - zkVM Architecture
 - ▶ The zkVM Landscape
 - ▶ Why RISC-V?
- ▶ zkVM Technical Overview
 - ▶ Frontends and Backends of zkVMs
 - ▶ Why do we care?
 - ▶ How to optimize zkVM performance
- ▶ zkVM Comparison
 - ▶ Table 2 - Component Comparison
 - ▶ Design Tradeoff Comparison
 - ▶ Benchmarking Rationale
 - ▶ Benchmarking Results
- ▶ Conclusion
- ▶ References

Introduction

Our project is an analysis and evaluation of zkVM construction and performance, benchmarking how the performance of different zkVMs scales with the memory usage of applications.

What is a zkVM?

A zkVM, is simply a VM implemented as a circuit for a zero-knowledge proof (zkp) system. So, instead of proving the execution of a program, as one would normally do in zkp systems, you prove the execution of the bytecode of a given Instruction Set Architecture (ISA).

There are a few types of zkVMs available on the market targeting different ISAs with various practical tradeoffs.

Table 1 - zkVM Architecture

	Existing Expertise / Tooling	Blockchain Focused	Performant
Mainstream ISAs RISC-V, WASM, MIPS	Lots	No	Maybe
EVM-Equivalent EVM Bytecode	Some	Yes	No
ZK-optimized New Instruction Set	No	Yes	Yes

The zkVM Landscape

EVM Equivalent:

- ▶ Type 1: Taiko
- ▶ Type 2-3: Scroll, Polygon zkEVM
- ▶ Type 4: zkSync

Mainstream ISAs

- ▶ RISC-V: Succinct's SP1, a16z's Jolt, RISC-0
- ▶ WASM: zkWASM
- ▶ MIPS: zkMIPS

ZK Optimized

- ▶ Polygon Miden, Starknet Cairo

Why RISC-V?

Extremely popular compile target for many programming languages (Rust, C++, LLVM). Open sourced. RISC vs. CISC → RISC has less instructions and is therefore easier to arithmetize and prove than x86 assembly for example.

zkVM Technical Overview

Now let's take a closer look at the frontend and backend components of zkVMs.

Frontends and Backends of zkVMs

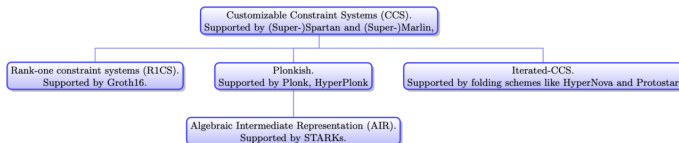


Figure 1: Intermediate representations (i.e., kinds of circuits) and the back-ends that can prove statements about them. The top-most box is the most general kind of circuit amongst those depicted, and each child is a special case of its parent. A subtlety not portrayed in the depicted taxonomy is that some SNARKs (such as Spartan and STARKs) can avoid having an honest party pre-process circuits if the circuits have repeated structure, while others cannot (Groth16, Marlin, Plonk).

Figure 1: Untitled

Frontend - Arithmetization Scheme

In general, arithmetization cannot be done manually except for elementary programs. Besides, the use of naïve arithmetization can lead to significant overhead. To deal with this, dedicated compilers accepting high-level programming languages have been developed.

Frontend - Precompiles

- ▶ Performance Issue: zkVMs operate significantly slower compared to running programs without proving overhead.
- ▶ Use of Precompiles: To improve efficiency, deployed zkVMs utilize “precompiles” — hand-optimized protocols for frequently used computations like hashing and signature verification.
- ▶ Risks of Overreliance: Relying heavily on precompiles can be problematic as designing these optimized protocols is the exact labor-intensive and error-prone process zkVMs aim to eliminate.

Backend - PCS & PIOP

The backend for proof system involves what we have learned in class, composed of two components: a PCS and a PIOP. Something notable about the interaction between the frontend and backend: most SNARKs can be easily tweaked to support both Plonkish and AIR with the exception of Groth16 which can only support R1CS.

Backend - Field Sizes

- ▶ Field Size Trade-offs: Operations in smaller fields are generally faster than in larger fields.
- ▶ Small vs. Large Fields: Using fields slightly smaller than 256 bits, like Goldilocks, can complicate operations with 256-bit numbers, requiring two field elements per value and roughly doubling prover costs.
- ▶ R1CS and Field Size: Currently, the R1CS system is constrained to larger fields, limiting flexibility in field size choice.

Backend - FRI Expansion Factor

The FRI blowup factor is a tunable parameter that allows you to adjust the cost to be more on the proving side or verifying side. A relatively low blowup factor leads to less prover time with larger proofs and a larger blowup factor leads to high cost to prove with smaller proof size.

Backend - Lookup Arguments

- ▶ Precomputed Outputs: This technique involves precomputing outputs for all possible inputs of a bitwise instruction.
- ▶ Efficient Verification: In a zkVM, a cost-effective SNARK operation, known as “the lookup,” verifies that the current instruction matches an entry in the precomputed table.
- ▶ Reduced Costs: Using lookup arguments decreases the cost of proving the instruction.

Why do we care?

It's important to distinguish between the backend and frontends of SNARKs to make clear assertions of the performance tradeoffs between various arithmetization schemes and SNARK backends. Failure to distinguish between them can result in misconceptions about performance and other characteristics of SNARKs

How to optimize zkVM performance

By efficient, we are almost always referring to proof generation time. Verifier time is about the same because we can use recursion to quickly verify proofs. Here are the options:

- ▶ Lookup tables.
- ▶ SNARK-friendly cryptographic primitives (such as Rescue, SAVER or Poseidon).
- ▶ Concurrent proof generation.
- ▶ Hardware acceleration (such as using GPU or FPGA).

zkVM Comparison

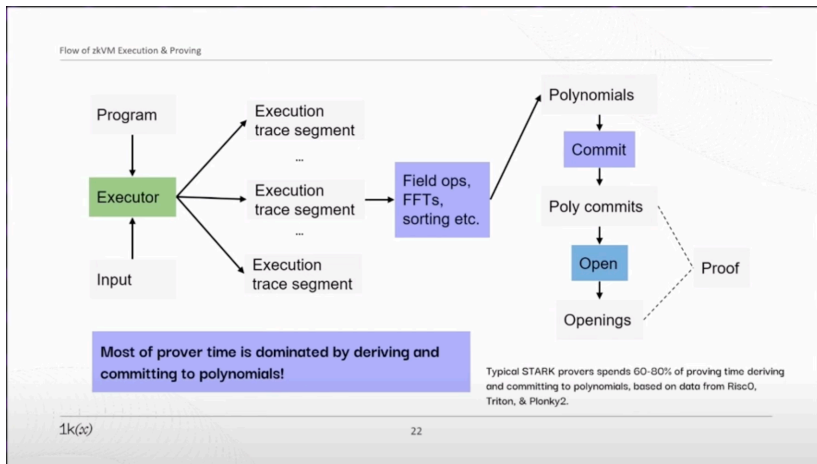


Figure 2: Untitled

Table 2 - Component Comparison

	RISC0	SP1	Jolt
PCS	FRI	FRI	Hyrax
Lookups	Plookup	Plookup?	Lasso
Field Size	~31-bit (baby bear)	~64-bit (goldilocks)?	~256-bit
Recursive Proofs	Yes	Yes	No
Precompiles	No?	Yes	No
Optimized for GPU	Yes	No	No
Arithmetization	AIR	AIR	R1CS
FRI Exp. Rate	4	2	N/A
SNARK Prover	Plonky2 STARK?	Plonky3 STARK	Spartan

Design Tradeoff Comparison

Jolt

In the frontend, Jolt uses Rank-1 Constraint System (R1CS). The backend employs Spartan and Hyrax, with Hyrax incurring larger prover costs. It operates over an approximately 256-bit field, although efforts are being made to use smaller fields. The system utilizes Spice-based memory checking and currently does not support recursion for aggregate proofs. For lookups, Jolt relies on Spice.

Design Tradeoff Comparison Continue

SP1

In the frontend, SP1 uses Algebraic Intermediate Representation (AIR) which requires expensive Fast Fourier Transforms (FFTs). The backend incorporates Plonky3 STARKs and Hyrax, with Hyrax incurring larger prover costs. It operates over smaller fields, either approximately 31-bit or 64-bit (Baby Bear, Goldilocks). SP1 supports recursion for aggregate proofs. The Fast Reed-Solomon Interactive Oracle Proof (FRI) blowup factor is 2, leading to faster proofs, larger proofs, and more expensive recursion. For lookups, SP1 uses Plookup. Notably, SP1 implements AIRs for each RISC-V instruction compatible with the Plonky3 prover and is optimized for CPU performance.

Design Tradeoff Comparison Continue

RISC0

In the frontend, RISC0 employs AIR, which also requires expensive FFTs. The backend utilizes Plonky3 STARKs and operates over a smaller, approximately 32-bit field (Baby Bear). RISC0 supports recursion for aggregate proofs, with a FRI blowup factor of 4, resulting in slower proofs, smaller proofs, and less expensive recursion. For lookups, RISC0 uses Plookup. This system is optimized for GPU performance.

Benchmarking Rationale

Jolt, RISC0, SP1 use different memory checking techniques which affects the proving time of programs as they scale with memory usage. The memory checking techniques are closely tied with the lookup arguments they use, with Jolt claiming to have improvements in this area. We need to breakdown the papers more to analyze the intricacies of these memory checks.

Benchmark Setup: The experimental setup, including hardware and software configurations, will be detailed, along with a description of the benchmarking methodology and specific memory-intensive operations tested.

Results: Findings from the benchmarks will be presented, focusing on memory usage, proving time, and verification time across different zkVM implementations.

Benchmarking Results

zkVM	Execution Time	Fibonacci Output	User Time	System Time	CPU Usage
RISC0	5.222s	n=10: 55	50.71s	0.32s	977%
RISC0	5.224s	n=100: 3594	50.75s	0.30s	977%
RISC0	10.128s	n=1000: 5965	103.04s	0.40s	1021%
RISC0	1:20.80	n=10000: 5721	812.55s	4.85s	1011%
Jolt	1.477s	n=10: 55	2.44s	0.85s	223%
Jolt	1.338s	n=100: 3594	3.42s	1.09s	337%

Benchmarking Results continue

zkVM	Execution Time	Fibonacci Output	User Time	System Time	CPU Usage
Jolt	2.678s	n=1000: 5965	10.80s	2.03s	478%
Jolt	22.720s	n=10000: 5721	117.09s	35.91s	673%
SP1	1.217s	n=10: 55	5.11s	0.67s	475%
SP1	1.221s	n=100: 3594	5.29s	0.64s	486%
SP1	1.877s	n=1000: 5965	10.79s	0.81s	618%
SP1	8.553s	n=10000: 5721	68.06s	1.74s	816%

Conclusion

- ▶ Benchmarks are misleading
- ▶ Precompiles are complicated
- ▶ Naming is hard
- ▶ Recursion is helpful

References

1. SP1 book. Introduction. (n.d.).
<https://succinctlabs.github.io/sp1/>
2. Jolt: Snarks for Virtual Machines . (n.d.).
<https://eprint.iacr.org/2023/1217.pdf>
3. Unlocking the lookup singularity with lasso. (n.d.-c).
<https://people.cs.georgetown.edu/jthaler/Lasso-paper.pdf>
4. RISC Zero zkVM: Scalable, transparent arguments of RISC-v integrity. (n.d.-b).
<https://dev.risczero.com/proof-system-in-detail.pdf>
5. LambdaClass. (2023, May 16). Arithmetization schemes for ZK-SNARKs. LambdaClass Blog.
<https://blog.lambdaclass.com/arithmetization-schemes-for-zk-snarks/>