Assessing and Benchmarking zkVMs: Insights into Performance and Scalability

Lawrence Lim, Nihar Shah

2024-05-13

Table of Contents

- Introduction
  - What is a zkVM?
  - Table 1 - zkVM Architecture
  - The zkVM Landscape
  - Why RISC-V?
- zkVM Technical Overview
  - Frontends and Backends of zkVMs
    - Frontend
    - Backend
  - 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 programing languages (Rust, C++, LLVM). Open sourced. RISC vs. CISC → RISC has less instructions and is therefore easier to prove than x86 assembly for example.

zkVM Technical Overview

ZK Design Differences: The paper will compare the underlying zero-knowledge proof systems, such as SNARKs and STARKs, and examine arithmetization schemes like AIR and R1CS, focusing on their implications for memory usage.

Memory Management Differences: Different approaches to memory management across selected zkVMs will be highlighted, with a focus on strategies like lookup tables and state management.

Frontends and Backends of zkVMs

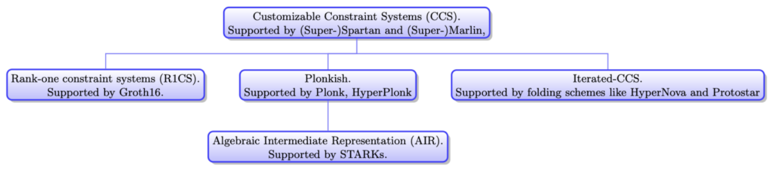

Customizable Constraint Systems (CCS).
Supported by (Super-)Spartan and (Super-)Marlin,
Rank-one constraint systems (R1CS).
Supported by Groth16.
Plonkish.
Supported by Plonk, HyperPlonk
Iterated-CCS.
Supported by folding schemes like HyperNova and Protostar
Algebraic Intermediate Representation (AIR).
Supported by STARKs.

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.

Precompiles

Compared to running a program without any proving overhead, zkVMs are incredibly slow. This poor performance is why all deployed zkVMs today use "precompiles," or hand-optimized protocols for specific computations that come up over and over again, like SHA2 or Keccak evaluations. But overreliance on precompiles is dangerous: designing pre-complies is *exactly* the error-prone and time-intensive process that zkVMs are meant to obviate.

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.

Field Sizes

There are additional tradeoffs to consider here such as field size and and FRI expansion rate (or blowup factor) for FRI based SNARKs. Some considerations when choosing a field size is that field operations over a small group are substantially faster than field operations over a large group. It's generally good to have the option to use a small field, but some operations like operating over 256 bit numbers get very annoying over small fields just under 256 bits like Goldilocks. You need to allocate two field elements for one value and it roughly doubles the prover costs. R1CS is constrained to larger fields (for now).

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.

Lookup Arguments

So to address this problem, researchers often use a technique known as a "lookup argument". Rather than compute the bitwise instruction directly via additions and multiplications, lookup arguments precompute the outputs of the bitwise instruction on all possible inputs. Then, a zkVM applies a relatively cheap SNARK operation – aka "the lookup" – to verify that the current instruction lives within the pre-computed table. Doing so decreases the cost of the instruction.

Why do we care?

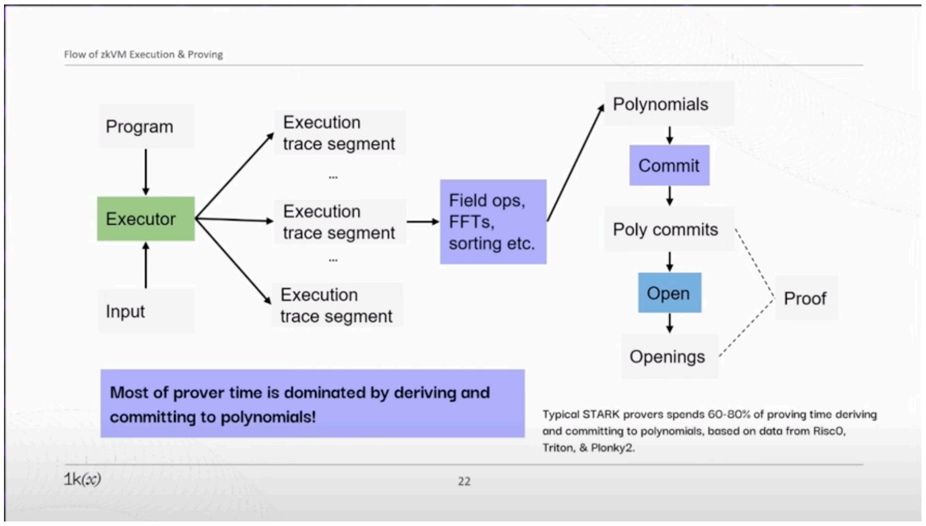
It's important to distinguish between the backend and frontends of SNARKs to make clear assertions of the performance tradeoffs between various arithmetization schemes and SNARK backends. Failure to distinguish between them can results in misconceptions about performance and other characteristics of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



Most of prover time is dominated by deriving and committing to polynomials!

Typical STARK provers spends 60-80% of proving time deriving and committing to polynomials, based on data from RiscO, Triton, & Plonky2.

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-big (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.

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.

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

...

Conclusion

It would be quite interesting to compare the proof

Benchmarking Challenges: The section will discuss the challenges in creating fair and comprehensive benchmarks for zkVMs, especially with varying memory loads, and mention the limitations of the Rust package and its impact on benchmark accuracy. Interpretation of Results: How the results should be interpreted given the limitations will be explained, suggesting areas for further research to refine the benchmarks.

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

References