Applied Learnings for Adverserial Distributed Systems

Methods, Explorations for everything blockchains

BY

Supragya Raj

PUBLISHED IN THE WILD

Contents

Chapter 1

Explorations in ZKVMs

This chapter intends to give a brief overview of the zero-knowledge virtual machines and their current state of usability.

1.1 RISC-V as a common target

As soon as we find ourselves exploring ZK-VMs, we quickly find ourselves with a bunch of projects such as RISC-Zero, Succinct's SP1, Mozak, Polygon Miden, Jolt etc, all of which for some reason are RISC-V compatible. What that means is that they expect input instructions to be composed of RISC-V instruction set.

1.1.1 Why RISC-V though?

Modern compiler infrastructure such as that of LLVM makes the code generation process modular. A lot of programming languages can be supported if we target RISC-V, for example C++ through Clang, Rust through RustC, Golang through TinyGo can all convert into LLVM bytecode. This can at the end be compiled down to a well knows small instruction set target such as RISC-V. RISC-V essentially is the 5th reduced-instruction-set-architecture that came out of UC Berkeley; hence the name. The main reason however that RISC-V becomes a compilation target of interest is because of two simple reasons apart from language-support: **minimalism** and **modularity**.

Minimalism in RISC-V: RISC-V is relatively simple ISA. It has 32 registers that can hold 32 bits each (`x0`-`x31`) alonside a special register `pc` that holds the address of currently executing instruction's address. The base 32-bit version of RISC-V dealing in integers is `riscvi` and has well-known simple instructions like `ADD`, `JAL`, `BNE` etc. All of these are relatively simple to describe.

Modularity in RISC-V: RISC-V provides a base, but also provides a bunch of useful extensions that one can opt-in to. For example, `+m` provides functionality such as multiplication (and division), `+f` provides floating point numbers, `+a` gives access to atomics among others. This modularity allows us to choose the tightest subset of features to implement for a ZKVM.

RISC-V compatible ZK-VMs are mostly targeted towards `riscvi+m`.

1.2 Writing constraints for these instructions

1.3 Self-modifying code and dealing with it

1.4 Identifying the program

1.5 Proof generation without pre-compiles

While the numbers described below present the situation as seen in December 2024, they still give good information on what we can expect in the signature verification regime. ED25519 is one of the signature schemes used quite commonly in blockchains like Solana, Tendermint-based chains like Cosmos as well as Polkadot.

Test Description	**Execution Trace Generation Time**	**RISC-0 Segments Count**	**Total Cycles**	**User Cycles**	**Proving Time**
ED25519 1 signature no-precompile 3-byte msg	91ms	4	3.67M	3.12M	130s
ED25519 1 signature no-precompile 3000-byte msg	91ms	4	4.19M	3.48M	147s
ED25519 2 signatures no-precompile 3-byte msg	171ms	7	6.81M	6.06M	242s
ED25519 2 signatures no-precompile 3000-byte msg	191ms	8	7.60M	6.78M	270s

Analysis of the above gives us the following key points:

- The proof generation procedure is undertaken in two stages: a simulation phase where the execution "trace" is generated and the proof generation phase where the execution trace is proven.
- Compared to the actual proving time in each which includes both the stages, the first stage (execution "trace" generation) takes miniscule amount of time. For example in the test running 1 ed25519 signature verification in zkvm, 91ms were taken in execution "trace" generation step while around 130,000ms were taken for proof generation.
- On average, on a very beefy machine without GPU, Risc-0 can clock in around **28,000 instruction / seconds** of proving.
- In large calculations, the verification is broken down into multiple smaller proving segments, limiting on average, proving time **30s / segment**, **8GB RAM usage / segment**. Maybe we can use parallelization of these segments for faster proving, but we need to see who "proves things". Also, segmentation of proof provides larger overhead in proof accumulation and continuation checks not accounted for in these benchmarks.

Another important test we run is for a hash function: SHA256. Alongside ED25519, SHA256 becomes the primary underlying algorithm used extensively in chains like Solana

Test Description	**Execution Trace Generation Time**	**RISC-0 Segments Count**	**Total Cycles**	**User Cycles**	**Proving Time**
SHA256 1 byte	3.21ms	1	131K	8.9K	4.5s
SHA256 10 byte	3.25ms	1	131K	8.9K	4.5s
SHA256 100 byte	3.39ms	1	131K	15.4K	4.62s
SHA256 1,000 byte	4.9ms	1	262K	100K	9.26s
SHA256 10,000 byte	20.5ms	1	1.05M	0.96M	37s
SHA256 100,000 byte	175.43ms	10	10.48M	9.5M	368s

1.6 Proof generation with pre-compiles

A few other things we learn from the above experiments:

- We find that zkVM needs to run at least 2^{17}**cycles** of instructions regardless of what our code does. If instructions are less, the rest of the rows are padded, which amounts to extra zkvm cycles.
- The growth of total cycles required for execution grows linearly with amount of bytes in question.

Chapter 2

Layer 2 Scaling Solutions

2.1 Faster Layer 2 Interoperability

In late 2020, Ethereum community started focusing on *rollup-centric roadmap*. The idea of this was to enable various Layer 2 scaling solutions that may potentially provide a whole lot more of execution capacity than the Ethereum (henceforth to be referred as Layer1 or L1) will be capable of.

However, Layer 2 come with following new concerns:

- **Security**: The security of Layer 2's is the concern of how easy it is to compromise a L2 in comparison to an L1. Generally, in a proof-of-stake system like ethereum, a lot of security is borrowed from stake at play. As of writing, this is around 33.8 million ETH staked / around 136.8 billion USD securing the ecosystem. Can the L2 match this level of security?

- **Decentralization**: Control of the system especially with chains like Ethereum is fairly decentralized. As of writing, around 1 million validators run the nodes for the main chain[1]. Layer 2 solutions hardly have decentralized sequencers and hence sit almost on the far end of decentralization argument. Ca the L2 match this level of decentralized controls?

- **Fragmentation**: Every L2 is almost like a separate "room", although situated within the same "house", the L1. This eventually leads to fragmentation of both users, and the liquidity. On a DEX, liquidity is directly tied to the quantity of tokens users have committed to the liquidity pools. If a crypto asset lacks sufficient liquidity, token holders may face difficulty selling their tokens when they wish. L2 brings in another dimenstion to this problem. Not only are tokens spread thin among various DeFi apps on the same chain, they are now spread among various DeFi apps on *various* chains.

[1]This number is a bit misleading given a lot is contributed by single institutions like stake.fish running multiple validator nodes given each validator can only stake 32 ETH per validator. The Ethereum Pectra Upgrade (EIP-7251) is supposed to fix it with increased `MAX_EFFECTIVE_BALANCE` the staking balance per validator, in hopes consolidating many such large validators into fewer entities

Chapter 3

Golang

Golang things

Chapter 4

Rust

RUSTY things

4.1 Iterators

IMPLEMENT RLE ITERATOR here: https://leetcode.com/problems/rle-iterator/description/

Chapter 5

Dynamic Programming

Dynamic Programming in it's crudest sense is nothing more than "clever bruteforce". What that means is while the solution eventually a large brute-force computation, the *cleverness* is in finding a sub-problem that helps solve for the larger problem instances. The key idea is "instance*s*", not a single instance.

Take the following relation for example:

$$\begin{aligned} F(x) &= Max(F(x-1)+1, 1) \\ F(0) &= 0 \end{aligned} \tag{5.1}$$

Here, $F(2)$ will only be used for calculating $F(3)$ and nowhere else.

On the other hand, the following relation:

$$\begin{aligned} F(x) &= Max(Max(nF(x-n)+1)\forall n \in [0, x), 1) \\ F(0) &= 0 \end{aligned} \tag{5.2}$$

uses $F(2)$ for all calculations where $x > 2$ such as $F(3)$, $F(4)$, etc.

The cleverness in the relation 5.2 that can potentially help it get to similar asymptotics as the 5.1 is **memoization**.

5.1 Memoization

Memoization is nothing but trade-off of space to gain in time. Naive implementation of 5.2 in code will yield results with almost constant space but exponential time. The time complexity leads to:

$$O(n) = \sum_{i=0}^{n-1} O(i) + 1 \tag{5.3}$$

If however, after every solution of an instance in range $[0, k)$, the solution is kept in some constant-time accessible storage M, the problem instance k dissolves into:

$$\begin{aligned} O(k) &= \sum_{i=0}^{k-1} O(i) + 1 \\ &= \sum_{i=0}^{k-1} O(1) + 1 \\ &= k + 1 \end{aligned} \tag{5.4}$$

We are going to use this idea quite generously.

5.2 Most important subproblems

Most of the times while we find ourselves looking at different problem statements, many-a-times at their core, they stem from one of the following few recursive relations:

- **0/1 Knapsack**:
- **Unbounded Knapsack**:
- **Longest Common Subsequence (LCS)**:
- **Longest Increasing Subsequence (LIS)**: