Applied Learnings for Adverserial Distributed Systems

Methods, Explorations for everything blockchains

BY

Supragya Raj

PUBLISHED IN THE WILD

Contents

Chapter 1

ZKVMs: Zero Knowledge Virtual Machines

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 The general design of ZKVMs

1.3 Writing constraints for these instructions

1.4 Self-modifying code and dealing with it

1.5 Identifying the program

1.6 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

1.7 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

Version Control using Git

2.1 Using Git via SSH

We can use GIT via a specific SSH command override, useful for when we want to use a specific key for `git clone` or `git push`. This can be done via:

```
GIT_SSH_COMMAND='ssh -i ~/.ssh/your_private_key' git push
```

Chapter 3

Ethereum

3.1 Legacy Ethereum (Ethereum 1.0)

3.1.1 Accessing Historical Block Hashes: EIP: 2935

For various applications, it is necessary to access the historical block hashes of the Ethereum blockchain. One such reasons is that it provides a good source of randomness. Another good reason is to build an "introspection engine" where contracts on Ethereum can introspect its own history.

Block hashes of the current block can not be known during the context of execution of a transaction. One of the reasons for this is that during execution of a transaction, the block's receipt hash is not built yet, since that in turn depends on all the transactions in the block combined, each transaction cannot have the block-hash.

Ethereum however, has an opcode called `BLOCKHASH` which allows contracts to access the block hash of the 256 most recent blocks. This opcode is useful for applications that require randomness, but it is not enough for applications that require access to the entire history of block hashes.

But what limits the access to historical block hashes more than 256 blocks old? There hash been two separate known attempts at solving this. One by Vitalik himself via EIP-210 and another by Vitalik, Tomasz (Netermind) etc via EIP-2935. As of writing, EIP-210 is marked as stagnant and EIP-2935 is in "Review" state since 4 years.

The primary idea of EIP-2935 was to create a new storage trie that stores the block hashes of all blocks. This would have been located at a special location: `0xffffffff_ffffffff_ffffffff_ffffffff_fffffffe`. This address would essentially be just a state without a bytecode, making it different from all other accounts. Hence, a new opcode `STOP` could

3.2 Ethereum after Merge (Ethereum 2.0)

3.2.1 The Proof-of-Stake Block

In every slot (spaced twelve seconds apart) a validator is randomly selected to be the block proposer. They bundle transactions together, execute them and determine a new 'state'. They wrap this information into a **block** and pass it around to other validators.

Other validators re-execute and validate: Validators who hear about the new block re-execute the transactions to ensure they agree with the proposed change to the global state. Assuming the block is valid, they add it to their own database.

If a validator hears about two conflicting blocks for the same slot they use their fork-choice algorithm to pick the one supported by the most staked ETH.

Previously, the Ethereum's Proof-of-Work Block (PoW Block) only contained execution related information. Post-Merge, the new Proof-of-Stake Block (PoS Block) will contain ACE information:

- **A**dministration (the metadata of the block)
- **C**onsensus (Beacon Chain co-ordination)
- **E**xecution (Block data)

This makes PoW Block more or less a subset of the new PoS Block. See 3.1.

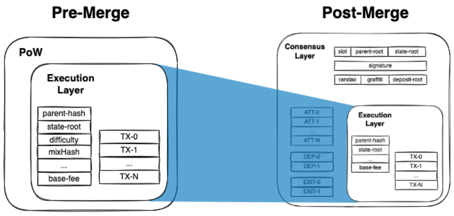

Figure 3.1: The Proof-of-Stake Block

3.2.2 Finality Metalayer: Casper FFG

Talk about:

- Plausible Liveness
- Accountable Safety

Chapter 4

Solana

The 8 advents of Solana

4.1 The Transaction Processing Unit

4.2 Proof of History: VDF based transaction ordering

4.3 Entries, Slots, Bank

Chapter 5

General Problems with Adversarial Distributed Systems

5.1 The ever growing state, state expiry

State expiry refers to removing state from individual nodes if it hasn't been accessed recently. There are several ways this could be implemented, including:

Expire by rent: charging "rent" to accounts and expiring them when their rent reaches zero. This design is implemented in Solana. **Expire by time**: making accounts inactive if there are no reading/writing to that account for some amount of time

5.2 Rebuilding the chain

Chapter 6

Layer 2 Scaling Solutions

6.1 General classification

We classify Layer 2 scaling solution according to their setup:

- **Plasma**: A tree of sidechains (chains of chains) where each user is asked to ensure the part of state it owns is available. Here, significant interaction with L1 is only required in case of dispute resolution. This came much before other solutions (proposed by Joseph Poon and Vitalik Buterin in 2017) but it was too complex to build a useful plasma.
- **Rollup**: An ecosystem running in parallel to a Layer 1 solution that has it's own state transition function (STF). After a bunch of transactions and blocks are created, the rollup posts transaction data / state-diff on chain and proves the combined state transition via opmistic or succinct zk proof.
- **Validium**: Similar to a rollup, just that the data-availability of the transaction is kept not on L1, but a separate *data-availability committee* or the validium operator themselves. Validium saves on recurring cost to make data available on L1, but at the cost of security.

6.2 Where sits the enourmous data?

Whenever a layer 2 scaling solution is built, the layer 1 contract is provided with the root commitment of the updated state. But a commitment is useless unless it can be opened. A *witness* links a leaf to the commitment and it requires a view of the state / transaction data for things like disputes and withdrawals.

Where does this data exists for generating a witness to do anything useful? A lot of answers can come up:

- Let the data sit on the Layer 1 itself, the rollup route. This is the simplest of the solutions but seldom the most practical. Layer 1s are expensive to post data to.
- Let the data sit on some other special data-availability layer which has some presence on the Layer 1 concerned. For example, posting on celestia. This is the validium route. Here you still pay to post on Celestia albeit much less.
- Let the data sit on a data-availability layer even cheaper than celestia, maybe with a smaller quorum. This is an even cheaper validium route. But the trade-off is security, as the data-availability layer may not be as secure as for example celestia.
- Let the data sit with the users themselves. Let they maintain the data and post it when required. This is the plasma route. This is the most decentralized of the solutions but also the most complex to build. Also, it requires the users to be online all the time. A pretty bad UX.

6.3 Plasma: Each one save one

The setup of Plasma entails that if each part of the state has a well-defined owner, then the owner can be made responsible for keeping their part of the system available.

Essentially, all Plasma chains have an **Operator** who keeps the chain moving. The system is setup optimistically with an intinsic level of trust on this operator. If the operator misbehaves, it remains the responsibility of the users to exit the chain.

Plasma chains are multi-layered chains where every "Parent Chain" settles for its child chains. The topmost "Parent Chain", the root chain settles on a well-known settlement platform and calls it the Layer 1. Each parent chain is responsible for three basic events through it's contract:

- **New Deposit**: Whenever anyone submits a new token deposit of the form (`depositAddress, amount, blockHeight`).
- **Block Progression**: Only submit the root-hash of the block, *not the data contained within it*. The data within hash to be kept in a good condition (and available) by the user. This takes the form (`blockHeight, rootHash`).
- **New Withdrawal**: If the operator misbehaves, the users are supposed to exit the plasma chain. This request is posted directly on Layer 1, as it may be assumed that the operator is not available / misbehaving. This takes the form (`withdrawaladdress, amount, blockheight, txIndex, ouputIndex`).

6.3.1 Operator misbehaviors and exits

6.3.2 Plasma Cash

In **Plasma Cash**, since users are only keeping track of their own tokens, they don't know who owns any of the other tokens. When a user wants to send their token to another user, they need to prove that they actually own that token! This creates a problem where each user is expected to be *perennially online* to ensure they can generate a witness against the current state-root of plasma chain.

In such a setup however, other inefficiencies also arise, because of the non-fungible nature of the tokens involved. In a ledger model of accounting, spending 0.1 Token or spending 20 Token out of a 50 Token balance are the same complexity-wise. However, the same is much harder with NFTs given we have to establish the denominations beforehand for all possible transfers in the future, or deal with splitting and merging of tokens.

6.4 Bottlenecked by DA

Layer 2 rollups have been around for a while. The first Layer 2 scaling solution was Plasma, proposed by Joseph Poon and Vitalik Buterin in 2017. The idea was to create a tree of sidechains, each of which would be able to process transactions in parallel. The root chain would be responsible for the security of the system, while the sidechains would be responsible for processing transactions.

Rollups however needs to guarantee that the state of rollup is **data-available**. Otherwise, if state is not available and the rollup-operator stops working, users cannot withdraw. As, even though the root commitment to the entire state is available, the opening are not. This makes rollups constrained by data-availability capacity of the Layer 1 chain.

Ethereum provides 375 `KB` blobs every 12 seconds.

The following question hence often comes us: *Can we eliminate need for state availability in L2 scalability solutions?*

6.5 Rollups and levels of confirmation

6.6 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.

The Interoperability problems is concerned with the *fragmentation* issue brought in by the advent of L2s.

In its most basic essence, Interoperability is concerned with a transaction that intends to affect state of two different L2 rollup chains. For example, a token transfer from A's account on rollup X to B's account on rollup Y is a transaction that affects the state of both X and Y.

[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 7

Golang

Golang things

Chapter 8

Rust

RUSTY things

8.1 Iterators

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

Chapter 9

Binary Search

9.1 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)**:

Chapter 10

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:

$$\begin{aligned} F(x) &= Max(F(x-1)+1, 1) \\ F(0) &= 0 \end{aligned} \tag{10.1}$$

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

On the other hand, the following relation:

$$\begin{aligned} F(x) &= Max(Max(nF(x-n)+1)\forall n \in [0,x), 1) \\ F(0) &= 0 \end{aligned} \tag{10.2}$$

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

The cleverness in the relation 10.2 that can potentially help it get to similar asymptotics as the 10.1 is **memoization**.

10.1 Memoization

Memoization is nothing but trade-off of space to gain in time. Naive implementation of 10.2 in code will yield results with almost constant space but exponential time. The time complexity leads to:

$$O(n) = \sum_{i=0}^{n-1} O(i) + 1 \tag{10.3}$$

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:

$$
\begin{aligned}
O(k) &= \sum_{i=0}^{k-1} O(i) + 1 \\
&= \sum_{i=0}^{k-1} O(1) + 1 \\
&= k + 1
\end{aligned}
\tag{10.4}
$$

We are going to use this idea quite generously.

10.2 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:

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