Lesson 22

Week 6

Lesson 21 - Formal Verification

Lesson 22 - Risc Zero

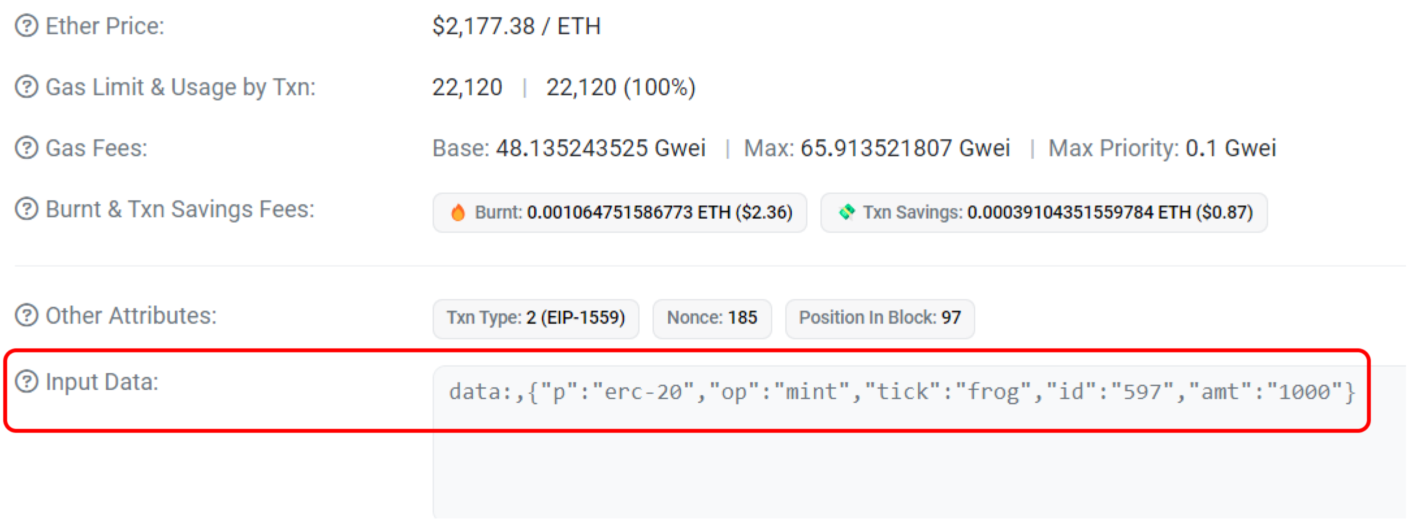
Lesson 23 - Advanced Plonk / Scroll

Lesson 24 - Review

Today's topics

- Impact of Inscriptions on L2 chains
- Risc Zero review
- Powdr and Polygon Miden review
- Walkthrough of example code

> # Inscriptions Introduction
>
> Inscriptions are a means to add digital assets on the blockchain. Traditionally the pattern used has been to store a link to a decentralised file system containing the asset.
> The mechanism was first seen on Bitcoin, where assets were added to satoshis, the mechanism is slightly different on non UTXO chains, where the asset data is added to a transaction.
>
> For example in an Ethereum transaction this looks like



Ether Price: $2,177.38 / ETH

Gas Limit & Usage by Txn: 22,120 | 22,120 (100%)

Gas Fees: Base: 48.135243525 Gwei | Max: 65.913521807 Gwei | Max Priority: 0.1 Gwei

Burnt & Txn Savings Fees: Burnt: 0.001064751586773 ETH ($2.36) Txn Savings: 0.00039104351559784 ETH ($0.87)

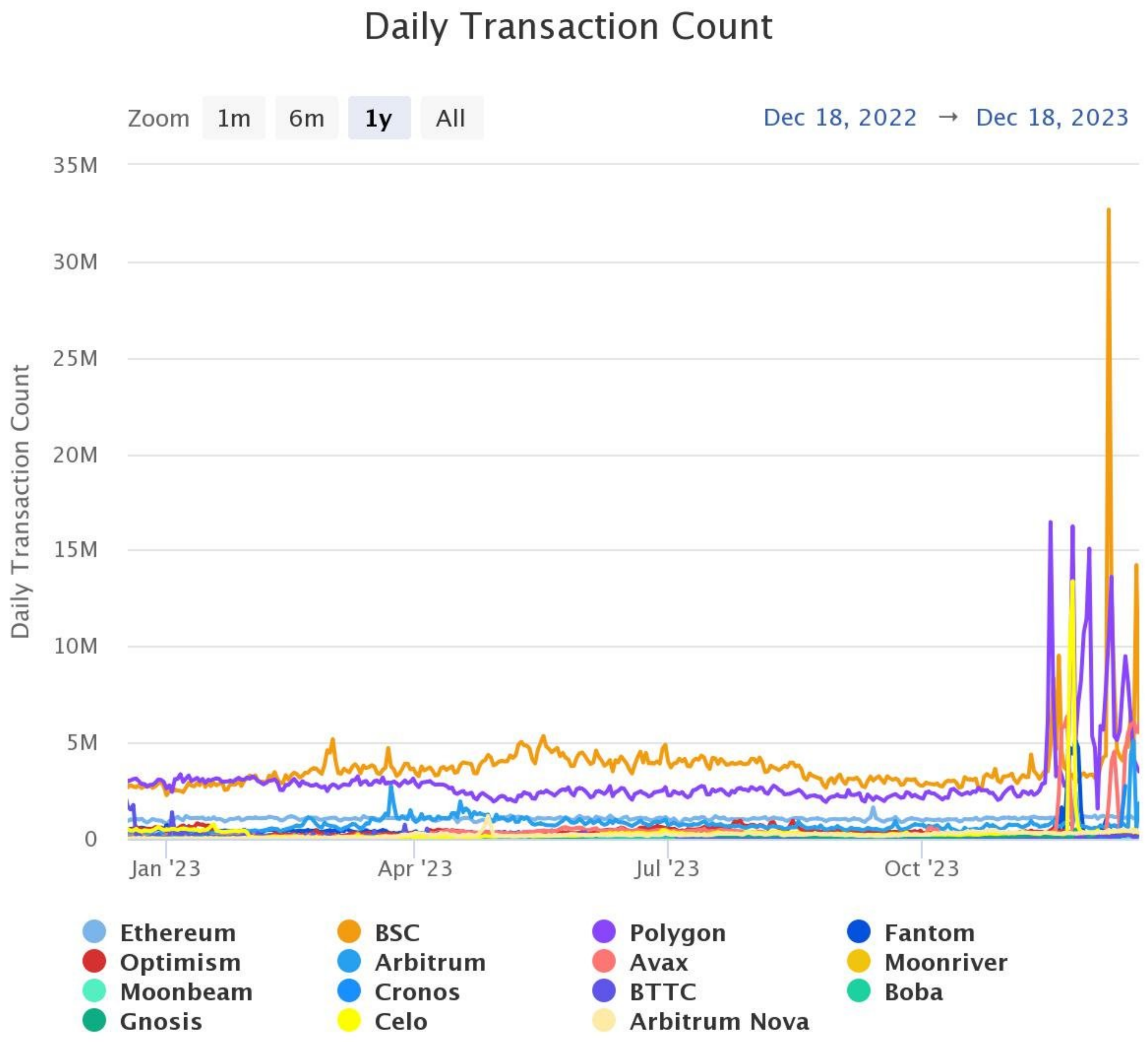
Other Attributes: Txn Type: 2 (EIP-1559) Nonce: 185 Position In Block: 97

Input Data:

```
data:,{"p":"erc-20","op":"mint","tick":"frog","id":"597","amt":"1000"}
```

The volume of inscriptions quickly became significant, here is a chart from Etherescan.

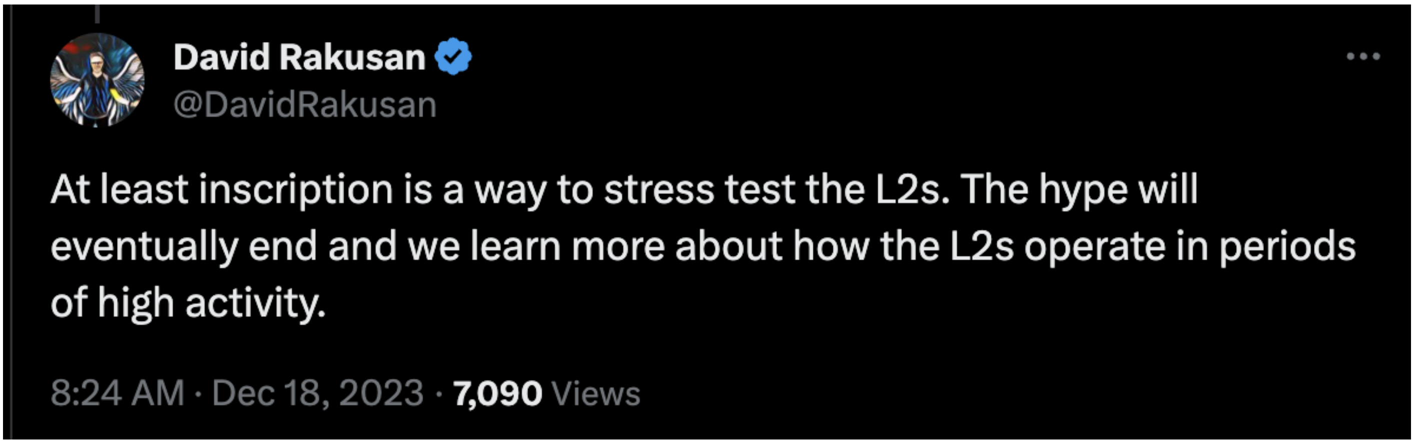
Post



Where transaction data is used he mechanism relies on third parties to index the data, and apply the rules you would see in a smart contract.

Effect of Inscriptions on networks

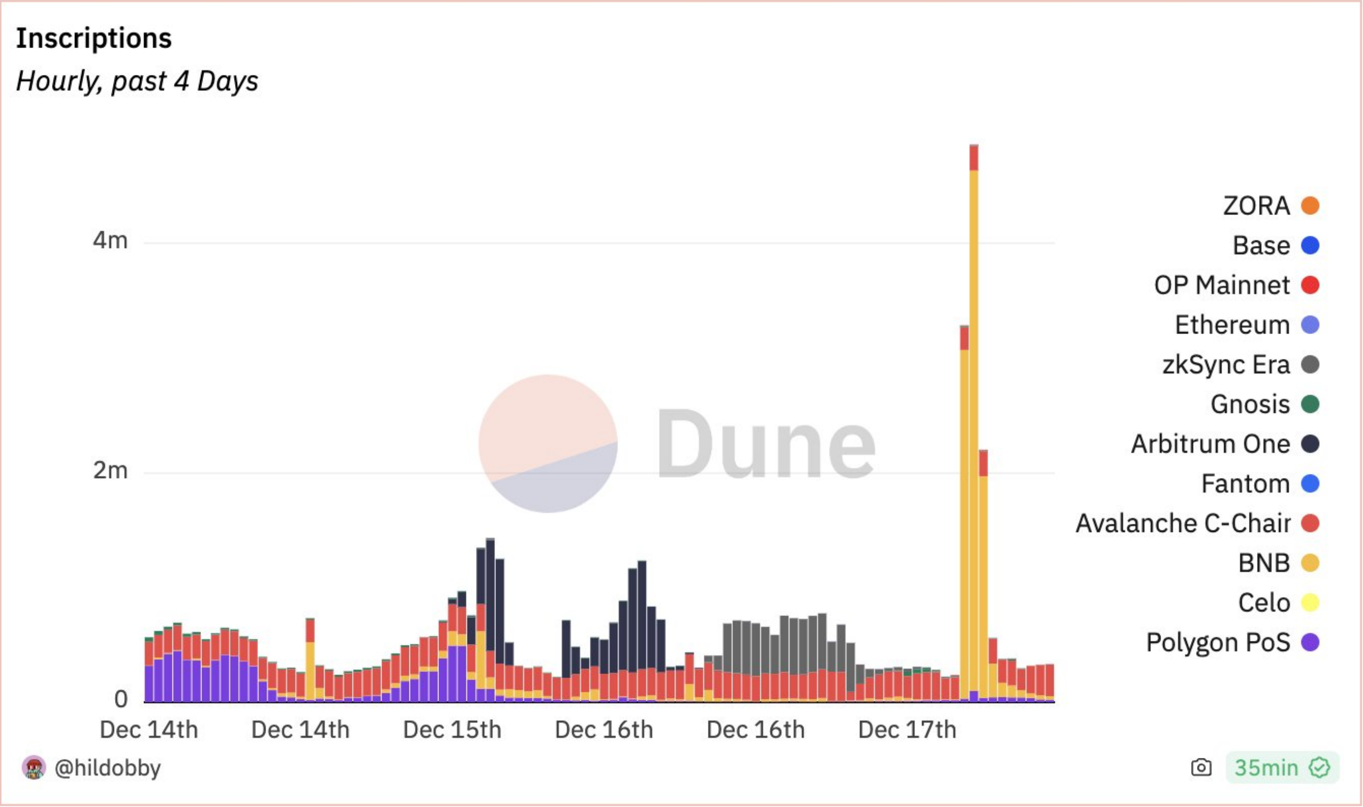
See article
From Coin Telegraph article
Arbitrum,Avalanche, Cronos, zkSync and The Open Network have all experienced partial or full outages recently due to inscriptions, with modular data availability network Celestia the latest to cave according to industry researchers
See post



David Rakusan
@DavidRakusan

At least inscription is a way to stress test the L2s. The hype will eventually end and we learn more about how the L2s operate in periods of high activity.

8:24 AM · Dec 18, 2023 · 7,090 Views



Arbitrum outage

See article

The Arbitrum One Sequencer experienced an outage on December 15th.

The Arbitrum batch poster is responsible for posting transaction data to Ethereum, and was unable to keep up with an Ethereum consensus client (which had a bug), exacerbated by large volumes of inscriptions being minted.

This lead to a failure in the feed component of the sequencer, which further lead to failures in RPC feeds.

The effect of this was out of date chain data seen by clients, whose transactions therefore failed.

Pricing mechanism impact

> The pricing mechanism received out-of-date information and undercharged for fees.
> Once batches began posting to L1 again after the sequencer restarted the result was a deficit of fees collected to pay for L1 data compared to the amount spent posting data to L1.
> The pricing mechanism then raised gas fees to try and compensate for the deficit, making transacting on the chain too expensive for most users.

Risc Zero in detail

Proof System

When a the RISC Zero zkVM executes, it produces a computational receipt that consists of:

- a journal which contains the public outputs of the computation, and
- a seal which is a zkSTARK

Given a `receipt` and an `Image ID` a skeptical third party can verify the purported output of the computation.

Further details

This includes

- Code to emulate RISC-V, including decoding RISC-V instructions and constructing the execution trace.
- Code to evaluate the constraint polynomials that check the execution trace.

The image ID is the SHA-2 hash of the image of the initial zkVM memory state.

Checking the Image ID

The image ID can be determined from the compiled ELF source code. Someone wishing to confirm that a receipt corresponds to Rust source code can compile that code targeting the RISC Zero zkVM and verify that the image ID resulting from this compilation matches the image ID in the receipt.

Like other zk-STARKs, RISC Zero’s implementation makes it cryptographically infeasible to generate an invalid receipt:

- If the binary is modified, then the receipt’s seal will not match the image ID of the expected binary.
- If the execution is modified, then the execution trace will be invalid.
- If the output is modified, then the journal’s hash will not match the hash recorded in the receipt.

Once the proof receipt has been generated it can be sent via side channel to the verifier.
The verifier does not need access to the host code, but they do need the image ID.

Structure of a zkVM program

In typical use cases, a RISC Zero zkVM program will actually be structured with three components:

- Source code for the *guest*,
- Code that *builds* the guest's source code into executable methods, and
- Source code for the *host*, which will call these built methods.

The code for each of these components uses its own associated RISC Zero crate or module:

- The *guest* code uses the `guest` module of the `risc0-zkvm` crate
- The *build* code for building guest methods uses the `risc0-build` crate
- The *host* code uses the `risc0-zkvm` crate

Example program

In the repo we have 3 example programs.

Password checker

Overall process

Alice wants to create a password, that complies with Bob's password requirements without revealing her password.
The program is divided between a host driver that runs the zkVM code and a guest program that executes on the zkVM.
Alice can run a password validity check and her password never needs to leave her local machine.
The process is as follows :

- Alice's `host driver program` shares a password and salt with the `guest zkVM` and initiates the guest program execution.
- The `guest zkVM program` checks Alice's password against a set of validity requirements, such as minimum length, inclusion of special characters.
- If the password is valid, it is hashed with the provided salt using SHA-256. If not, the program panics and no computational receipt is generated.

- The guest program generates a salted hash of Alice's password and commits it to a `journal`, part of a computational `receipt`.
- Alice sends the receipt to Bob's Identity Service.

The `image ID` and the `journal` on the receipt provide Bob assurance that:

The program Alice executed within the zkVM was actually Bob's Password Checker, and
Bob's Password Checker approved Alice's password

It demonstrates that the guest zkVM program has executed, which tells Bob `his password requirements were met`.
It also provides a tamper-proof `journal` for public outputs, the integrity of which tells Bob that **the shared outputs are the result of running the password program**.

Application design

We split the code between guest and host, if there is a part of the computation that doesn't need to be performed securely, then it can be run outside the zkVM.
For optimisation, when using cryptographic operations, it is possible to build ‘accelerator’ circuits such as the Risc zero implementation of SHA26.
Fast cryptography is sufficient to support many ‘DeFi’ applications. For many other applications, it is possible to perform most computation on the host (outside the zkVM) and then verify the results in the zkVM.
In terms of program size, although there is a theoretical maximum size is 128 MB, with the current implementation programs should be kept to at most ~ 1MB.

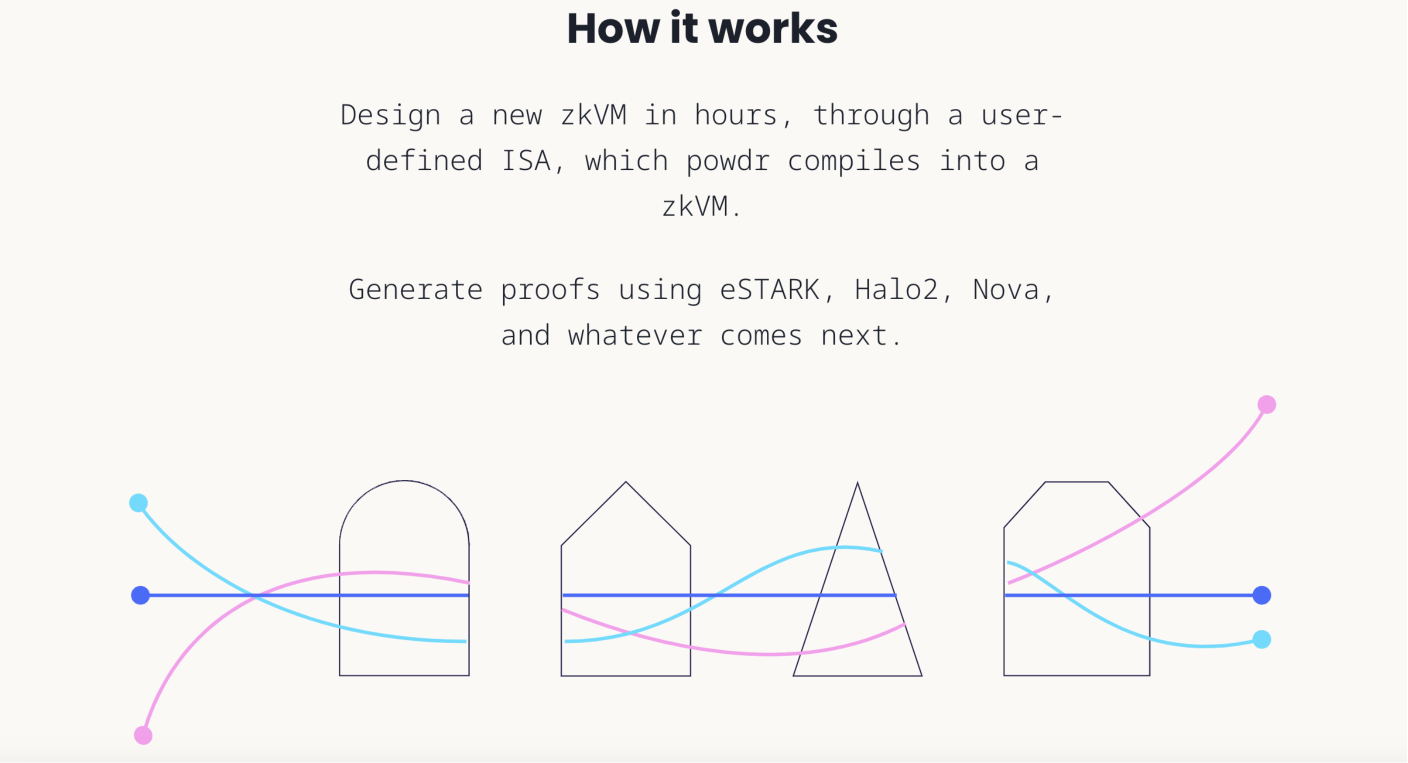
Proving the validity of two RISC Zero receipts currently takes ~10 seconds.

Rust compatibility

Risc zero are working to include the rust standard library, until that is complete it is advised to use the `no_std` option

Powdr

Powdr brings modularity, flexibility, security and excellent developer experience to zkVMs.



How it works

Design a new zkVM in hours, through a user-defined ISA, which powdr compiles into a zkVM.

Generate proofs using eSTARK, Halo2, Nova, and whatever comes next.

See Docs

powdr is a modular compiler stack to build zkVMs. It is

> ideal for implementing existing VMs and experimenting with new designs with minimal boilerplate.
>
> - Domain specific languages are used to specify the VM and its underlying constraints, not low level Rust code
> - Automated witness generation
> - Support for multiple provers as well as aggregation schemes
> - Support for hand-optimized co-processors when performance is critical
> - Built in Rust 🦀
>
> See video from Christian Reitwiessner
>
> ## Installation
>
> Rust is a prerequisite
>
> Instructions are here
>
> ## Front Ends
>
> A Risc V frontend is available and others are under development.
>
> ## Backends
>
> Halo 2 and eSTARK are supported

> # Polygon Miden overview
>
> Polygon Miden is a general-purpose, STARK-based ZK rollup with EVM compatibility.
> They have emphasised zk friendliness over EVM compatibility.
>
> From their documentation
> "Polygon Miden can process up to 5,000 transactions in a single block, with new blocks produced every five seconds. Although this ZK rollup exists as a prototype for now, it is expected to boost throughput to over 1,000 transactions per second (TPS) at launch."

Miden and Risc Zero benchmarks