Session 4

Navigator sessions

Date	Topics
15/12/23	o1js review
19/1/24	Development workflow, design approaches, techniques and useful patterns
26/1/24	Recursion
9/2/24	**Application storage solutions**
1/3/24	Utilising decentralisation
15/3/24	zkOracles and decentralised exchanges
5/4/24	Ensuring security
26/4/24	Review Session

Today's topics

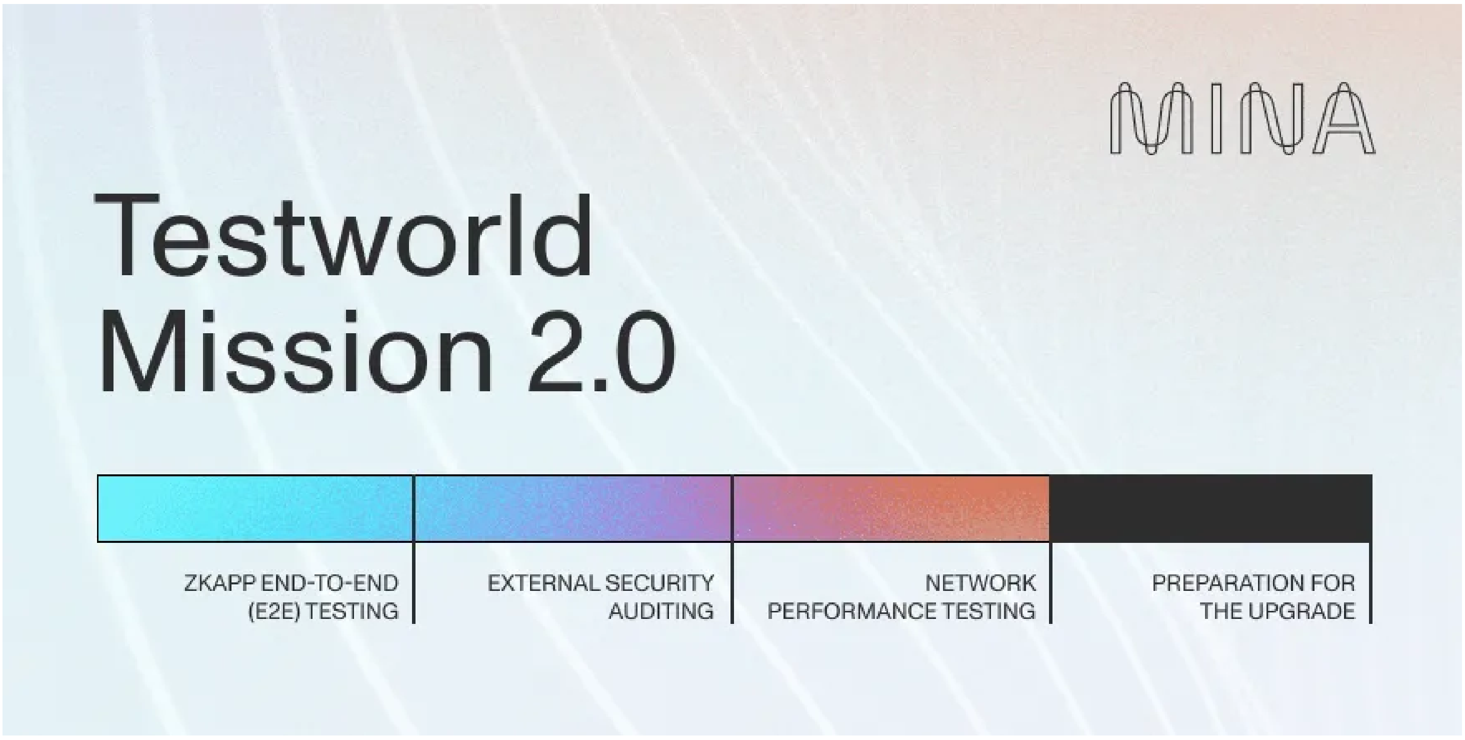
- What's new
- Offhchain storage
  - Simple techniques
  - Data structures
  - Other techniques
  - Concurrency Problems
    - Action Recucer
    - Alternatives
  - The bigger picture
  - Modular Blockchains
  - L2 architecture
  - L2 Projects on Mina
    - Anomix
    - Zeko
    - Protokit part 1

Slido Link

What's new

See Mina Blog

Testworld Mission


MINA
Testworld Mission 2.0
ZKAPP END-TO-END (E2E) TESTING
EXTERNAL SECURITY AUDITING
NETWORK PERFORMANCE TESTING
PREPARATION FOR THE UPGRADE

- The Testworld Mission 2.0: Protocol Performance Testing program received more than 3,305 applications from over 70 countries.
- 250+ experienced node operators were selected to provide the network backbone for Testworld Track 3.
- Operators ran more than 670 individual nodes and processed more than 29k blocks, 176k transactions, and 85k zkApp transactions over 10 rounds of load testing.
- Track 3 allowed for the testing of various loads and helped uncover issues which have since been resolved. As a result, an optimal configuration was identified, and the release candidate for the Mainnet Upgrade is ready.
- Next, there will be testing of all the processes and tooling for the upgrade mechanism as the final preparation step before the major Berkeley Upgrade.

The upgrade voted on in proposals (MIP1, MIP3, MIP4) will include :

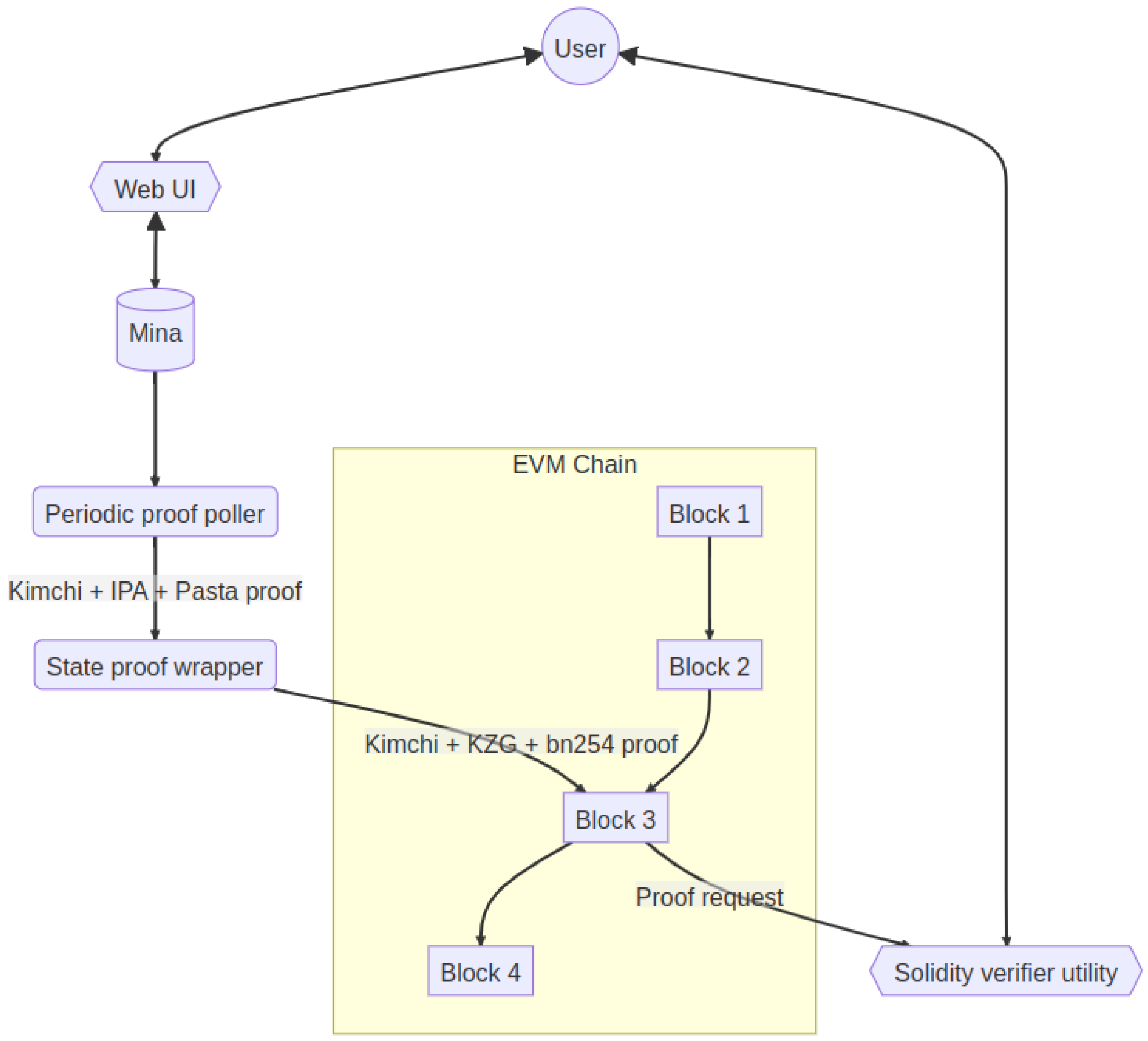
- **ZK programmability with o1js**, previously “SnarkyJS,” which is a TypeScript library for writing zkApps.
- **Enhanced security and efficiency from Kimchi**, Mina’s marketing-leading proof system that facilitates the creation of recursive ZK proofs.
- **Removal of supercharged rewards**, a temporary incentive to increase staking adoption at the earliest stages of mainnet.

Next Steps

The testnet for Mina's Berkeley Upgrade is split into four tracks, with the first three tracks including zkApp End-to-End (E2E) testing, external security auditing, and protocol performance testing now complete.
The fourth, which will begin in February 2024, will test the new features in a replicated network and engineer the upgrade mechanism, the final stage before the Mina mainnet upgrade.
Private testing with exchanges and custodians is already underway, and will continue to run throughout this track.

Mina to Ethereum Bridge

See Blog
See Repo



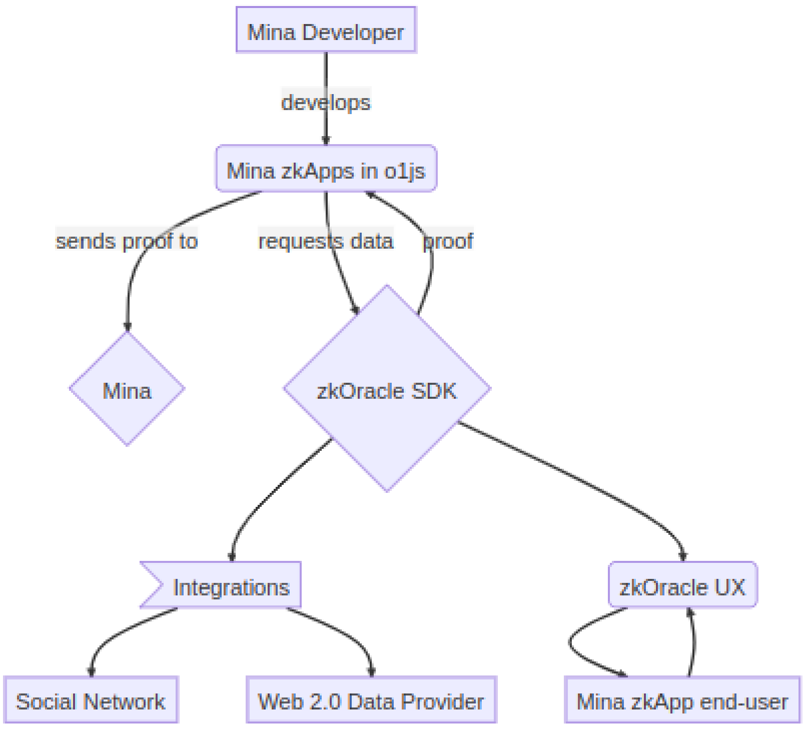
zkAppsUmstad

SeeRepo
zkApps Umstad AI Assistant, your expert guide in the world of MINA Protocol, o1js. This agent provides in-depth assistance with zkApps o1js which is specifically designed to assist developers working on zkApps development powered by the GPT language model. This project includes two main application:

- **Web Application Chatbot:** zkappsumstad.com
- **CLI Agent:** zkappumstad

Ecosystem Advancement (RFP): zkOracle Integration for o1js

See discussion
To integrate zkOracle functionality with o1js, enabling Mina zkApp developers to incorporate zk-oraclized data in their applications.



Off chain Storage

Defining the problem

- Limited amount of on chain storage
- Privacy

A small amount of chain state is one of the essential features of Mina.

The 'standard' approach

Store an onchain commitment to an off chain data structure.
See Tutorial

Problems

- Trust (data storage provider)
- More generally this speaks to Data availability problems (guarantees of availability , a malicious person could for example just repeatedly provide a merkle hash, but not keep the data).
- Decentralising a solution is costly / complex, it means replication and proving stored data. (See the techniques used by Filecoin)

Some solutions from the tutorial

1. A single-server storage solution
2. A multi-server storage solution that can be run by multiple parties for stronger trust guarantees.
3. A solution that leverages storage on modular blockchains.
4. A future hard fork to add purchasable on-chain data storage to Mina.
5. A future hard fork to add data-storage committees to Mina for horizontally scalable storage.

When designing these solutions the Mina Foundation provides the following suggestions

- Eliminate DDOS vulnerability by adding a token that limits storage requests.
- Do not store trees for a smart contract if that contract is misconfigured in a way to prevent cleaning up old data.
- Add support to the client library for connecting to multiple storage servers, enabling correctness under a majority-honest assumption.
- Switch the project to a more scalable database implementation (for example, Redis).
- Write an implementation for an automatically scalable service (for example, Cloudflare).

Design using a merkle tree

- Each leaf is either empty or stores a number (an o1js field), which is the data.
- Updates to the tree can update a leaf if the new number in the leaf is greater than the old number.
- The root of the tree is stored on-chain.
- The tree itself is stored on an off-chain storage server.

Suitable Data Structures - Merkle Trees / Maps

Points

- Why do we use merkle trees ?
- Remember this is just a data structure (albeit with useful features , but a cost of many hashes)

Merkle Trees

See Documentation
You can import the `MerkleTree` type and create one of a specific height, for height `h` you will get 2^{h-1} leaves
A common pattern is to store the root of the tree in our contract.
Checks for inclusion of a leaf, or updates to that leaf require details of the path through the tree to the leaf, such as specifying the index of the leaf.

Example update code

```
// initialize the zkapp
const zkApp = new BasicMerkleTreeContract(basicTreeZkAppAddress);
await BasicMerkleTreeContract.compile();

// create a new tree
const height = 20;
const tree = new MerkleTree(height);
class MerkleWitness20 extends MerkleWitness(height) {}

...

const incrementIndex = 522n;
const incrementAmount = Field(9);

// get the witness for the current tree
const witness = new MerkleWitness20(tree.getWitness(incrementIndex));

// update the leaf locally
tree.setLeaf(incrementIndex, incrementAmount);

// update the smart contract
const txn1 = await Mina.transaction(senderPublicKey, () => {
  zkApp.update(
    witness,
    Field(0), // leafs in new trees start at a state of 0
    incrementAmount
  );
});
await txn1.prove();
const pendingTx = await txn1.sign([senderPrivateKey, zkAppPrivateKey]).send();
await pendingTx.wait();

// compare the root of the smart contract tree to our local tree
console.log(
  `BasicMerkleTree: local tree root hash after send1: ${tree.getRoot()}`
);
console.log(
  `BasicMerkleTree: smart contract root hash after send1: ${zkApp.treeRoot.get()}`
);
```

In the contract we have

```


  @state(Field) treeRoot = State<Field>();

  @method initState(initialRoot: Field) {
    this.treeRoot.set(initialRoot);
  }

  @method update(
    leafWitness: MerkleWitness20,
    numberBefore: Field,
    incrementAmount: Field
  ) {
    const initialRoot = this.treeRoot.get();
    this.treeRoot.assertEquals(initialRoot);

    incrementAmount.assertLt(Field(10));

    // check the initial state matches what we expect
    const rootBefore = leafWitness.calculateRoot(numberBefore);
    rootBefore.assertEquals(initialRoot);

    // compute the root after incrementing
    const rootAfter = leafWitness.calculateRoot(
      numberBefore.add(incrementAmount)
    );

    // set the new root
    this.treeRoot.set(rootAfter);
  }
```

Merkle Map

See Docs
This works in a similar way to the merkle tree, but allows to to specify the leaf by a key rather than an index.
Example code

```
const map = new MerkleMap();

const rootBefore = map.getRoot();

const key = Field(100);

const witness = map.getWitness(key);

// update the smart contract
const txn1 = await Mina.transaction(deployerAccount, () => {
  zkapp.update(
    contract.update(
      witness,
      key,
      Field(50),
      Field(5)
    );
  );
});
```

Also see this library
"The library contains implementations of *Sparse Merkle Tree*, *Standard Merkle Tree* and *Compact Merkle Tree* based on o1js, which you can use in the **browser** or **node.js** env, and provides a corresponding set of verifiable utility methods that can be run in **circuits**. Besides, you could choose different persistence storage tools for each Merkle tree."

Techniques to help with on chain state.

- Pack the state that you have, but this only goes so far.
  -
    - **o1js-pack** A library for o1js that allows a zkApp developer to pack extra data into a single Field. GitHub and npm
- Use a multiple contract design
- Use the fields available in a token
  See this discussion.

The concurrency problem

If multiple users make updates to state off chain, how can we decide the ordering or resolves any conflicts.

Action / Reducer

See Docs

Motivation / Use case

A common pattern to handle off chain state is to have a merkle tree holding state, update the state in the tree, producing a new state root, and then publish the root on chain.

This runs into problems when we want concurrent state updates, and given Mina's block time, it is likely that any particular block will contain multiple updates.
This is an obvious concern for useful applications running off chain, trying to keep a consistent view of state without concurrency problems, such as the first update succeeding, and subsequent updates in the same block failing.
From Reducers are Monoids

"Manipulation of state in large applications quickly gets hairy. As an application grows, it becomes a real challenge to be sure that state mutations affect only the components you want it to. One mitigation is to centralise all of your state manipulation as best as you can — ideally to one function or one file or one module.
To do so, we can decouple an intention to change state (or an action) from the actual state change itself.
Reducers are one way to cleanly declare atomic chunks of state manipulation in response to these actions. Smaller reducers can be composed into bigger ones as our application's state management grows in scope."

To continue with our earlier example, we send a number of `pending actions` which show our intentions, then at a later time you roll these up using a `reducer` to apply the actions sequentially, then finally we would update the state on chain.

Events and actions are not stored in the ledger and exist only on the transaction.

Examples

You can see an example here

```
// compute the new counter and hash from pending actions
    let pendingActions = this.reducer.getActions({
      fromActionState: actionState,
    });

    let { state: newCounter, actionState: newActionState } =
      this.reducer.reduce(
        pendingActions,
        // state type
        Field,
        // function that says how to apply an action
        (state: Field, _action: Field) => {
          return state.add(1);
        },
        { state: counter, actionState }
      );

    // update on-chain state
    this.counter.set(newCounter);
    this.actionState.set(newActionState);
  }
}
```

Drawbacks to the Action / Reducer model

See Issue
This doesn't scale well, we need to have a reducer process continually running in order to rollup any actions that happen.
Also the reducer is limited in what it can due because of the circuit size limit, see comment

---

Workarounds / Alternatives

There is an interesting discussion here

Some thoughts from that

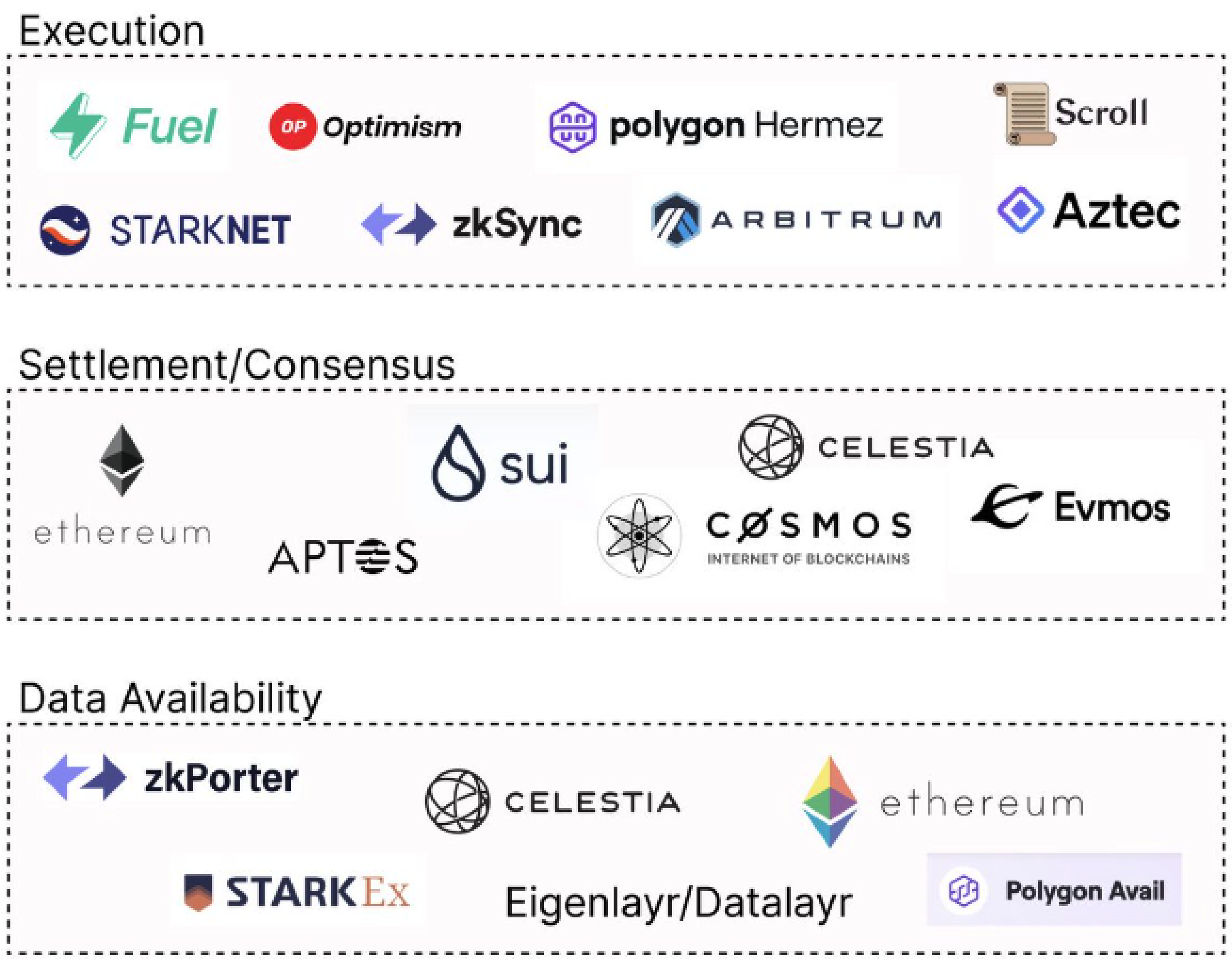
- The reducer has a recursive proof to change the state.
  The zkApp gets all pending actions from the archive node, reduce them recursively using ZkProgram, and feeds the proof to the reducer on-chain as an argument to change the state.

A proposal from Gregor is
to have two kinds of actions - normal ones and those big state-storing ones. The reducer would ignore the big state actions and for reading state we would ignore the normal actions

---

The bigger picture

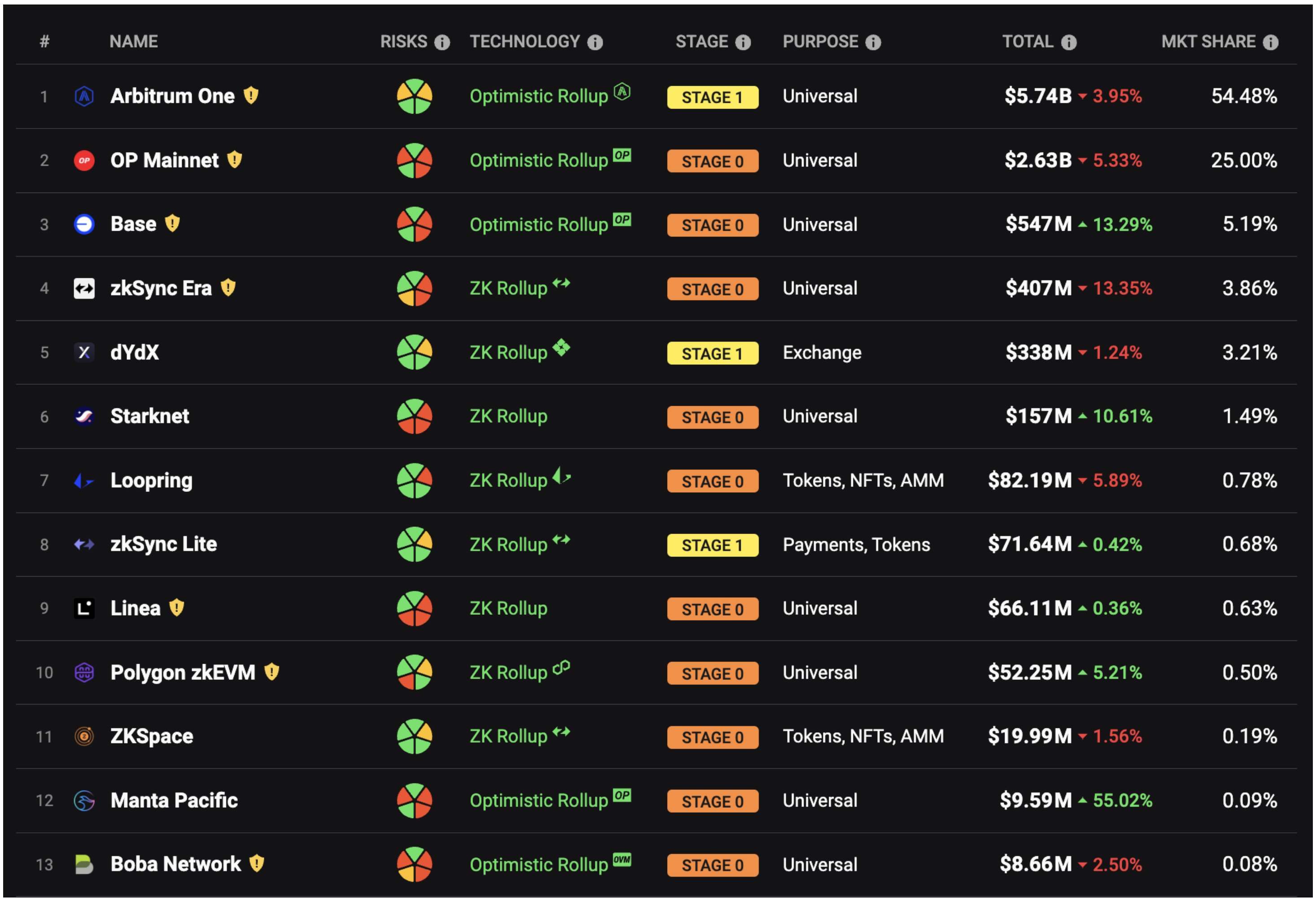
Modular Blockchains



If we follow the principle of separation of concerns, we can use a combination of blockchains to provide the functionality of a L1 and increase scalability.
For further details see Volt article

L2 architecture

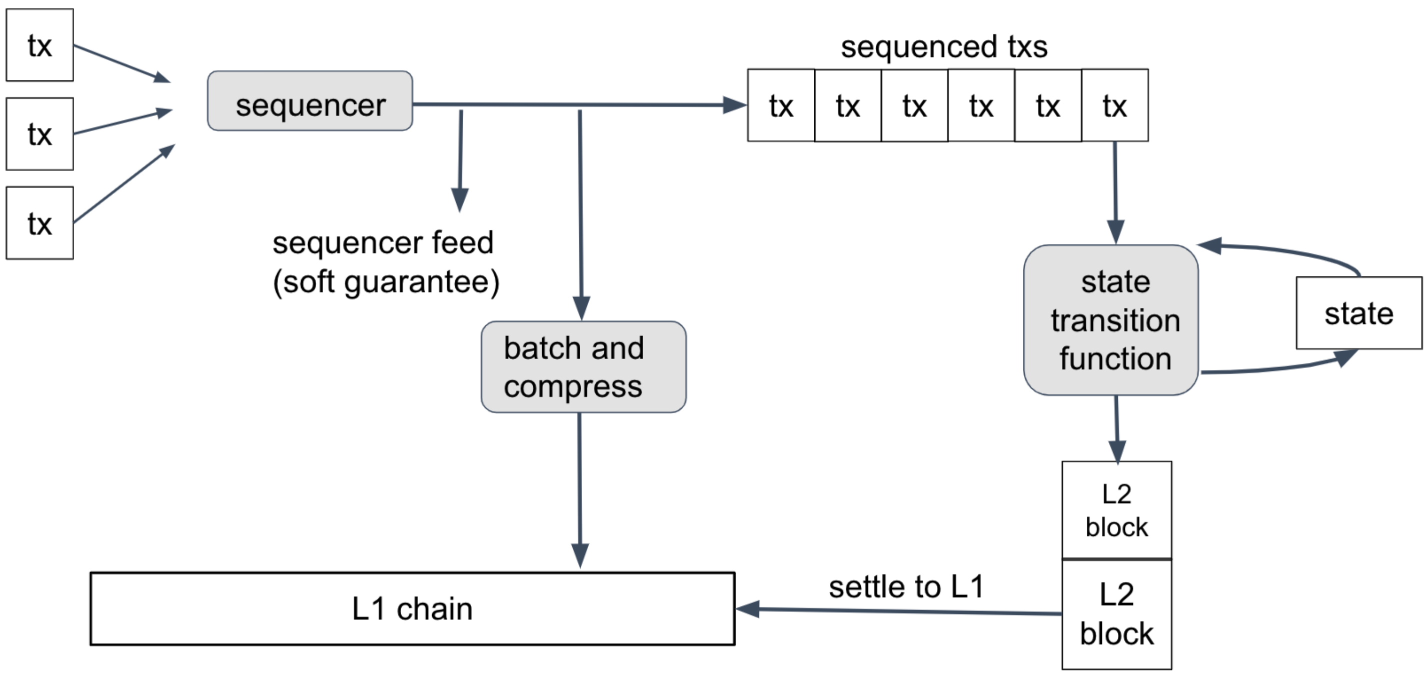
Layer 2 chains have become popular as a solution to scaling Ethereum.



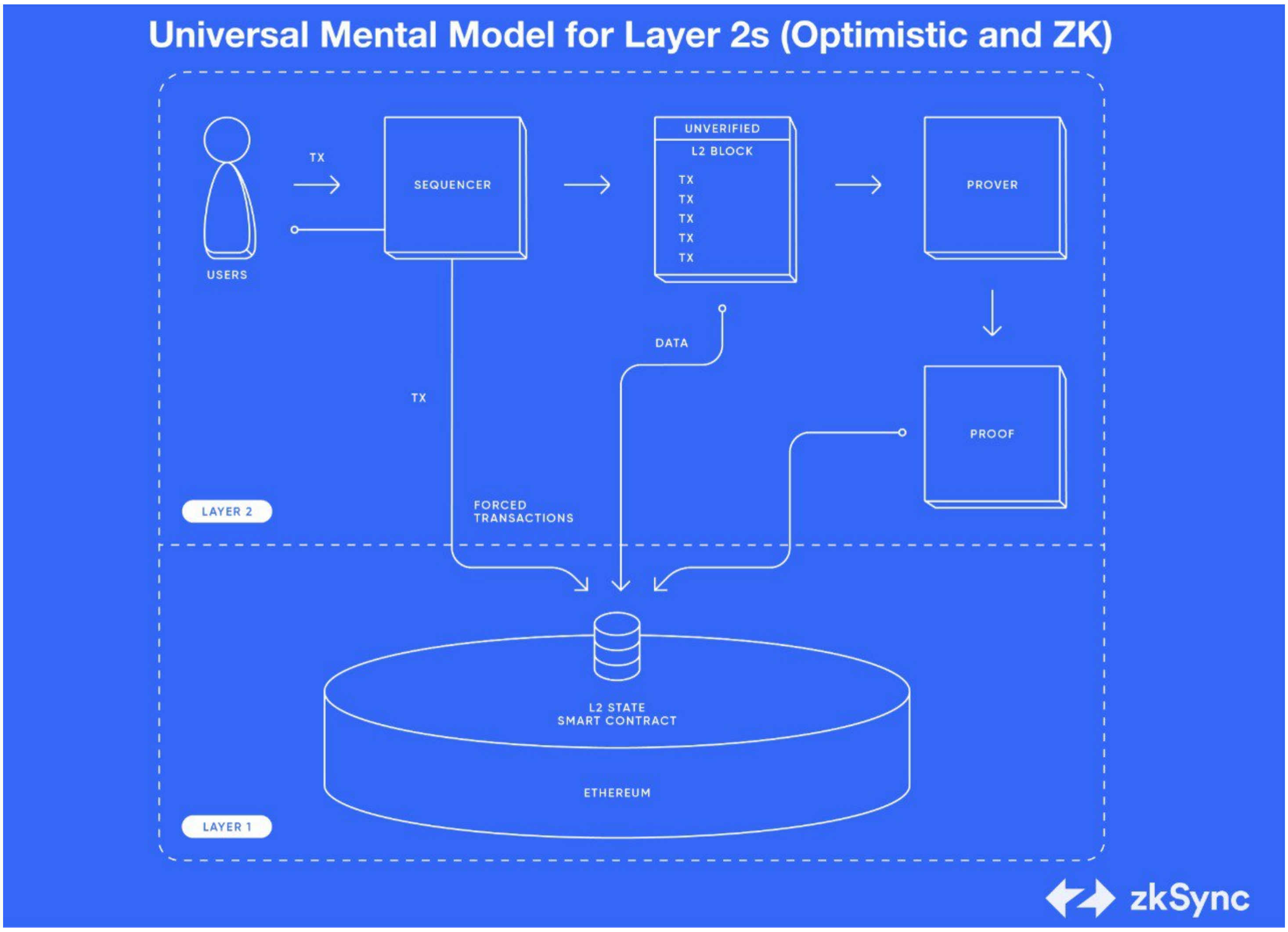
#	NAME	RISKS	TECHNOLOGY	STAGE	PURPOSE	TOTAL	MKT SHARE
1	Arbitrum One	●	Optimistic Rollup	STAGE 1	Universal	$5.74B ▾ 3.95%	54.48%
2	OP Mainnet	●	Optimistic Rollup OP	STAGE 0	Universal	$2.63B ▾ 5.33%	25.00%
3	Base	●	Optimistic Rollup OP	STAGE 0	Universal	$547M ▴ 13.29%	5.19%
4	zkSync Era	●	ZK Rollup	STAGE 0	Universal	$407M ▾ 13.35%	3.86%
5	dYdX	●	ZK Rollup	STAGE 1	Exchange	$338M ▾ 1.24%	3.21%
6	Starknet	●	ZK Rollup	STAGE 0	Universal	$157M ▴ 10.61%	1.49%
7	Loopring	●	ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$82.19M ▾ 5.89%	0.78%
8	zkSync Lite	●	ZK Rollup	STAGE 1	Payments, Tokens	$71.64M ▴ 0.42%	0.68%
9	Linea	●	ZK Rollup	STAGE 0	Universal	$66.11M ▴ 0.36%	0.63%
10	Polygon zkEVM	●	ZK Rollup	STAGE 0	Universal	$52.25M ▴ 5.21%	0.50%
11	ZKSpace	●	ZK Rollup	STAGE 0	Tokens, NFTs, AMM	$19.99M ▾ 1.56%	0.19%
12	Manta Pacific	●	Optimistic Rollup OP	STAGE 0	Universal	$9.59M ▴ 55.02%	0.09%
13	Boba Network	●	Optimistic Rollup OVM	STAGE 0	Universal	$8.66M ▾ 2.50%	0.08%

They can be seen more generally, as part of the modular blockchain approach.

At typical architecture of an L2 is shown in Arbitrum



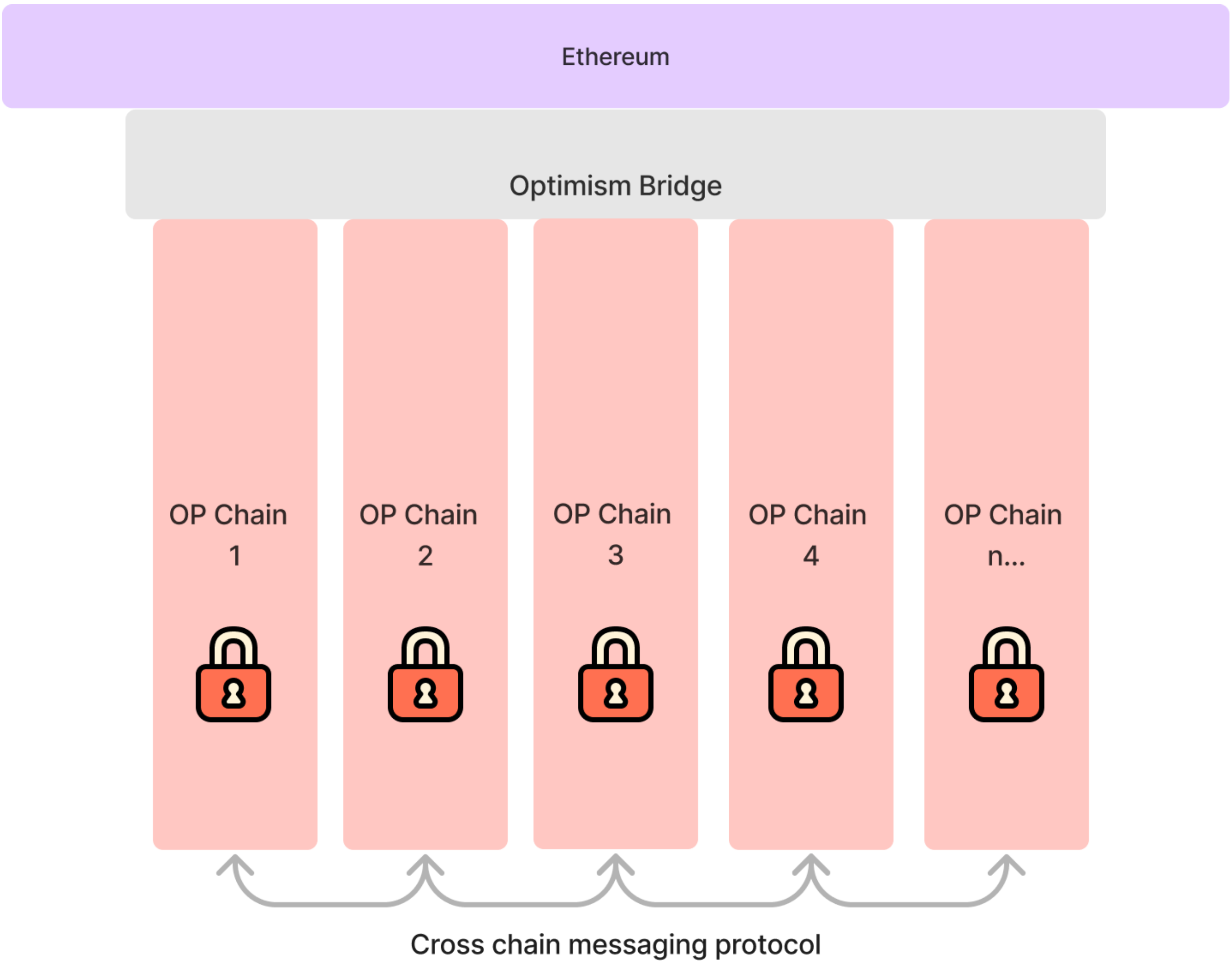
Here is a similar diagram from zkSync



Fractal Scaling

An ongoing trend is to build L3 / L4 ... using a similar idea to the L2, this is often done to produce specialised chains, we talk of app chains to mean a chain (L2 or otherwise) that has been designed for a specific application or type of application.

We also see the idea of horizontal scaling by spinning up similar chains on demand, such as with OP Stack to produce a 'superchain'



Important points to consider

- When do we have finality for a transaction ?
    - L2 acceptance / L1 acceptance
- How do we withdraw assets from L2 -> L1 ?
- Centralisation of the sequencer and censorship

Can this type of architecture help with the offchain storage problems ?
If so why do they help ?

L2 Projects on Mina

Anomix L2

Anomix Network

A zk-zkRollup layer2 solution on Mina
focusing on Privacy and Scalablility

Privacy first, easily protect your assets from tracking

Store & Send Mina at a lower cost than Mainnet transfers

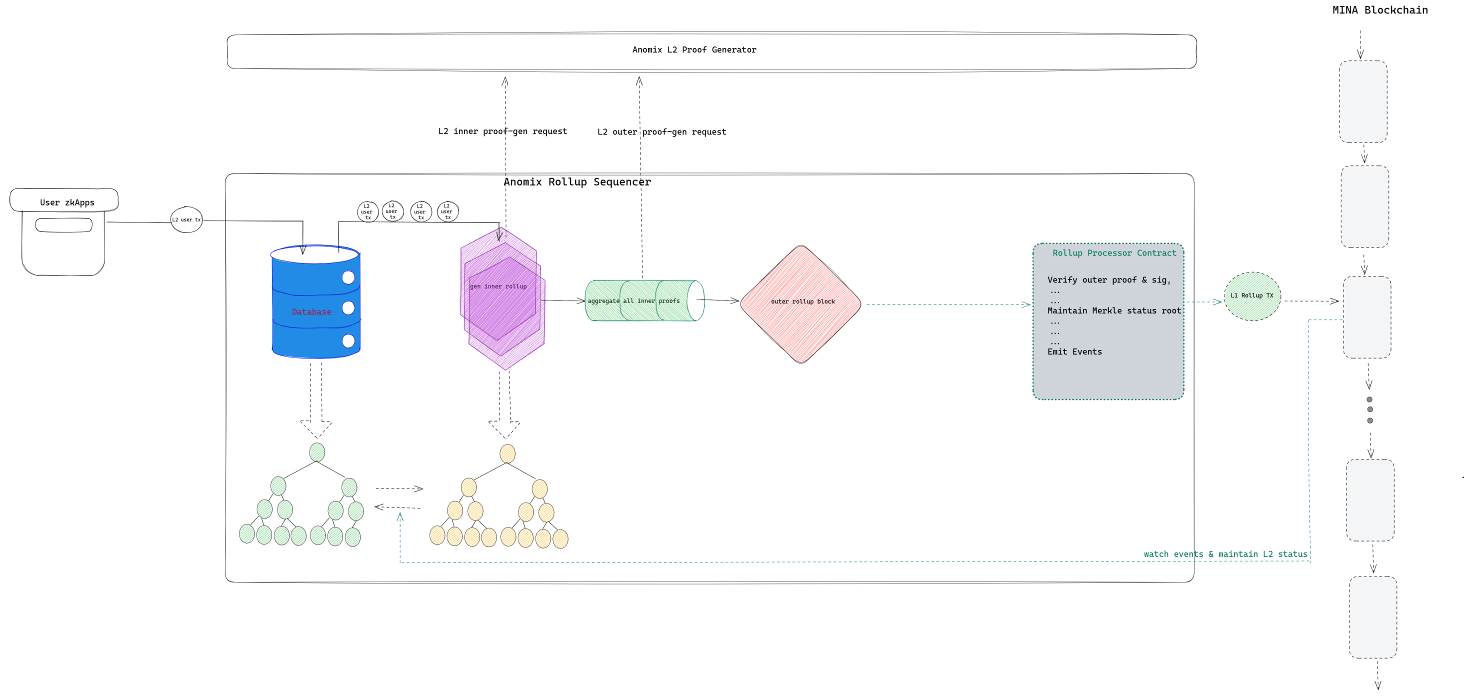
See [zkignite details](#)
[Repo](#)

Features

- Privacy of state
- Batching of transactions
- Based on UTXO model

From [Docs](#)
`Anomix Sequencer`

- receive, validate and store L2 User Tx (normally from `User-Oriented zkApps`)
- calculate recursive zkProof for latest L2 User Tx
  - Anomix L2 Inner Rollup Circuit
  - Anomix L2 Outer Rollup Circuit
- publish rollup block within a L1 tx to `Anomix Rollup Processor Contract`
- maintain L2 status(i.e. each merkle trees)
- listen for onchain events, .etc.


MINA Blockchain
Anomix L2 Proof Generator
L2 inner proof-gen request
L2 outer proof-gen request
Anomix Rollup Sequencer
User zkApps
L2 user tx
L2 user tx
L2 user tx
L2 user tx
L2 user tx
Database
gen inner rollup
aggregate all inner proofs
outer rollup block
Rollup Processor Contract
Verify outer proof & sig,
...
Maintain Merkle status root
...
Emit Events
L1 Rollup TX
watch events & maintain L2 status

Storage

All L2 tx(normally from `User-Oriented zkApps`) are persisted into Database by `Anomix Sequencer` after basical validation.

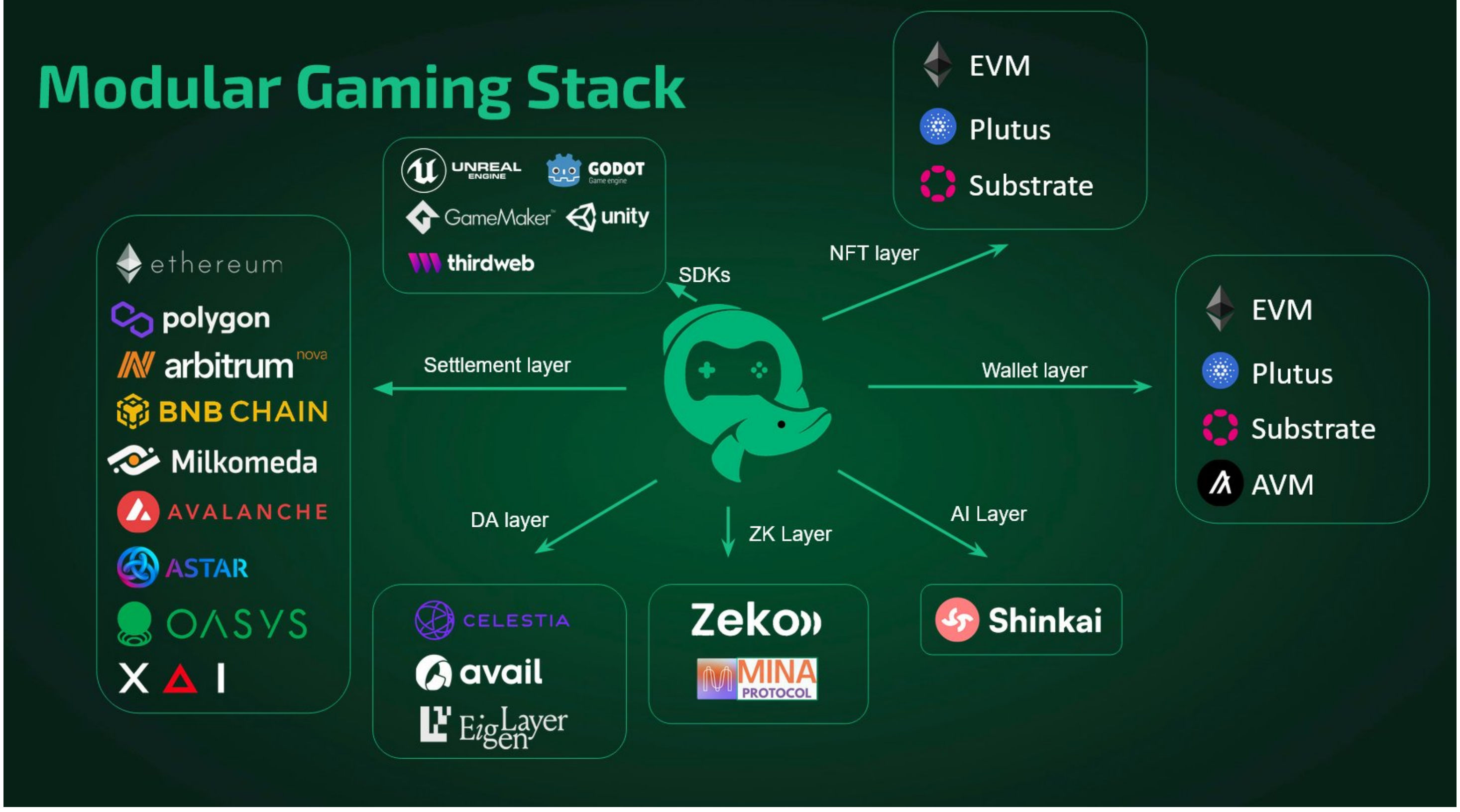
When Anomix Network starts, `Anomix Sequencer` constructs Merkle Trees based on all confirmed L2 tx set, etc. These Merkle trees represent the Latest status of the whole L2 Network.

- data tree
    - account note commitment
    - value note commitment
- nullifier tree
    - alias nullifier
    - account key nullifier
    - value note nullifier
- root tree
    - records all historical root of data tree.

Note: All leaf nodes is based on Hash of plain txt instead of encrypted notes.

Zeko

See Video
See litepaper



Tech details

Data availability

Initially built on Ethermint chain on Cosmos, giving reasonable finality (1-2s)
Validators will sign the batched data, the signatures will be aggregated and posted to Mina L1.
The validators in this case are some trusted organisations (O1 Labs / MF etc.)

Eventually it is likely to go to use Celestia instead.

Protokit

Came from 2 zkIgnite projects that merged.
Video introduction and this thread.

" Protokit is a framework that provides abstractions for writing zk-rollups, it separates business and state handling logic - the goal was to simplify everything down to a point where you can just focus on business logic - like if you're writing solidity"

"just persisting the merkle tree isn't enough, you need to separate data reads/writes into a separate circuit, and combine it with circuits for business logic, otherwise you'll have too many things at once in one circuit and parallelisation of proving wont be efficient - thats what protokit does for you"

The protokit server has a built in db and handles the merkle tree operations behind the scenes.
"There's a full SDK that uses a graphql api behind the scenes to fetch the latest state of your rollup"

Starter kit

See Repo

Setup

```
git clone https://github.com/proto-kit/starter-kit my-chain
cd my-chain

# ensures you have the right node.js version
nvm use
pnpm install
```

Running the sequencer & UI

```
# starts both UI and sequencer locally
pnpm dev

# starts UI only
pnpm dev -- --filter web
# starts sequencer only
pnpm dev -- --filter chain
```

Navigate to `localhost:3000` to see the example UI, or to `localhost:8080/graphql` to see the GQL interface of the locally running sequencer.