Valence ZK Demo

by Timewave Computer

Valence

- Architecture
- Cross-chain Message Application
- Cross-chain Rate calculation
- Benchmarks

1.1. Architecture: Modularity

What is an interface?

- An interface defines shared functionality in a codebase and improves extensibility / simplifies integration processes.

- New chains can be added to Valence ZK, simply by implementing the set of methods defined in our interfaces.

What is a method?

- A method is a single function that is part of an interface.

- An interface is just a composition of methods.

Valence

1.1. Architecture: Modularity

1.1.1. Merkle Library:

2 common *interfaces*

MerkleClient interface

- *get_proof*() -> returns a merkle proof for the chain

MerkleVerifiable interface

- *verify*() -> verify a merkle proof

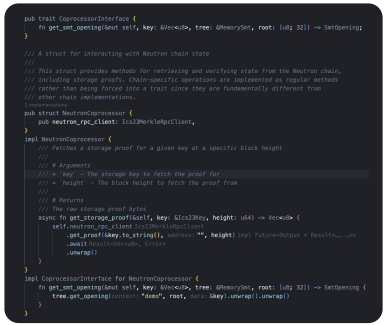
```rust
pub trait MerkleVerifiable {
    /// Verifies the proof against the expected Merkle root.
    ///
    /// # Arguments
    /// * `root` - The expected Merkle root to verify against
    ///
    /// # Returns
    /// A boolean indicating whether the proof is valid for the given root
    ///
    /// # Note
    /// The verification process should check that:
    /// 1. The proof nodes form a valid path from the leaf to the root
    /// 2. The leaf node contains the expected key-value pair
    /// 3. The root hash matches the expected root
    fn verify(&self, root: &[u8]) -> Result<bool>;
}
```

1.1.2 Coprocessor:

1 new *interface* per supported Chain

CoprocessorInterface

- *get_opening*() -> get a merkle proof against the Valence SMT
- *get_storage_proof*() -> get a storage proof for the target chain



1.1.3. ZK Light-Client:

1 new *interface* per ZK Light-Client

LightClientInterface

- get_proof() -> returns a zk light-client proof

This proof object consists of:

- zk proof with public values (previous chain state)
- chain merkle root

We verify the correctness of the proof against the previous merkle root of the chain and add the new merkle root to our Coprocessor.

This is a step-by-step process, where we always verify the new state of the supported chain against the last known state and publish the results in the form of a commitment. The app developer does not need to worry about this, everything happens under the hood!

Valence

1.2. Architecture: SMT Abstraction

In order to verify the state / values stored on different blockchains, we need to cryptographically verify the integrity of the merkle roots. This is usually done through one ZK Light-Client per chain.

Valence abstracts away the complexity of verifying state against multiple ZK Light-Clients by combining all these different clients into a single data structure, where we can cryptographically verify state with ease - the app developers don’t need to interact with ZK Light-Clients directly.

In the end all the developer needs to do is call a single verification function that is always the same, regardless of the chain they’re on:

```
for ethereum_proof in inputs.ethereum_messages_openings {
    assert!(MemorySmt::verify(
        "demo",
        &inputs.coprocessor_root.try_into().unwrap(),
        &ethereum_proof,
    ));
    messages.push(deserialize_ethereum_proof_value_as_string(ethereum_proof));
}
```

1.3. Architecture: ZK Light-Client Integrations

When integrating new ZK Light-Clients, which is an essential step towards production-grade application development on Valence, one simply needs to implement the respective interface, as mentioned in 1.1. Architecture: Modularity.

We are already working on solutions for

- Cosmos ICS23 (Tendermint)
- Ethereum and the EVM ecosystem

and will soon support a set of ZK Light-Clients by default.

Please note that we are moving fast and the state of ZK Light-Client integrations might have advanced in the meantime!

```rust
impl MockNeutronLightClientInterface for MockLightClient {
    /// Implementation of the Neutron light client interface
    ///
    /// Connects to the Neutron chain via RPC and retrieves the latest block's app hash and height
    ///
    /// # Returns
    /// A tuple containing:
    /// - The app hash as a byte vector
    /// - The block height as a u64
    async fn get_latest_neutron_root_and_height(&self) -> (Vec<u8>, u64) {
        let tendermint_client = tendermint_rpc::HttpClient::new(
            TendermintUrl::from_str(&self.neutron_light_client.rpc_url).unwrap(),
        )
        .unwrap();
        let latest_block = tendermint_client.latest_block().await.unwrap();
        let height = latest_block.block.header.height.value() - 1;
        let app_hash = base64::engine::general_purpose::STANDARD
            .encode(hex::decode(latest_block.block.header.app_hash.to_string()).unwrap());
        (
            base64::engine::general_purpose::STANDARD
                .decode(app_hash)
                .unwrap(),
            height,
        )
    }
}
```

Valence

2. Demo: Cross-chain messaging Application

For the scope of this Demo, we have built a cross-chain messaging application that was successfully deployed to Sepolia (Ethereum testnet) and Pion-1 (Neutron testnet).

Our cross-chain messaging Smart Contract stores messages by their index and increases a counter every time a message is sent.

We initialized it with a “HelloEthereum” and “HelloNeutron” message on the counterparty chains.

Valence

2. Demo: Cross-chain messaging Application

The entire Valence application (aside from the deployed contracts) is extremely simple, as we only need to verify inclusion proofs against our Coprocessor state.

The code on the right is all you need to write in order to use our Valence stack that under the hood verifies all of the cryptographic commitments on all the supported chains (in this case Neutron, Ethereum).

No deep knowledge in blockchain state proofs or ZK required!

```rust
#![no_main]

use types::{
    MailboxApplicationCircuitInputs, MailboxApplicationCircuitOutputs,
    deserialize_ethereum_proof_value_as_string, deserialize_neutron_proof_value_as_string,
};
use valence_smt::MemorySmt;

sp1_zkvm::entrypoint!(main);
fn main() {
    let inputs: MailboxApplicationCircuitInputs = borsh::from_slice(&sp1_zkvm::io::read_vec())
        .expect("Failed to deserialize MailboxApplicationCircuitInputs");
    let mut messages: Vec<String> = Vec::new();
    for ethereum_proof in inputs.ethereum_messages_openings {
        assert!(MemorySmt::verify(
            "demo",
            &inputs.coprocessor_root.try_into().unwrap(),
            &ethereum_proof,
        ));
        messages.push(deserialize_ethereum_proof_value_as_string(ethereum_proof));
    }
    for neutron_proof in inputs.neutron_messages_openings {
        assert!(MemorySmt::verify(
            "demo",
            &inputs.coprocessor_root.try_into().unwrap(),
            &neutron_proof,
        ));
        messages.push(deserialize_neutron_proof_value_as_string(neutron_proof));
    }
    let outputs = MailboxApplicationCircuitOutputs { messages };
    sp1_zkvm::io::commit_slice(&borsh::to_vec(&outputs).unwrap());
}
```

Valence

3. Demo: Cross-chain Vault Rate Calculation

In addition to the simple cross-chain messaging application, we would like to share an example where a cross-chain LP rate is calculated based on values stored on different chains.

- We define the storage keys of the values that we are interested in (in this case the minted LP shares, LP token balance on each chain => a total of 4 values).
- We calculate the cross-chain rate based on those values

That's all, we again don't need to worry about many cross-chain complexities, as all of these values can be verified against our Valance abstraction tree (Coprocessor).

```rust
sp1_zkvm::entrypoint!(main);
fn main() {
    let inputs: RateApplicationCircuitInputs = borsh::from_slice(&sp1_zkvm::io::read_vec())
        .expect("Failed to deserialize RateApplicationCircuitInputs");
    let neutron_balance =
        deserialize_neutron_proof_value_as_u256(inputs.neutron_vault_balance_opening.clone());
    let neutron_shares =
        deserialize_neutron_proof_value_as_u256(inputs.neutron_vault_shares_opening.clone());
    let ethereum_balance =
        deserialize_ethereum_proof_value_as_u256(inputs.ethereum_vault_balance_opening.clone());
    let ethereum_shares =
        deserialize_ethereum_proof_value_as_u256(inputs.ethereum_vault_shares_opening.clone());

    // verify the SMT opening proofs against the root
    assert!(MemorySmt::verify(
        "demo",
        &inputs.coprocessor_root.try_into().unwrap(),
        &inputs.ethereum_vault_balance_opening,
    ));
    assert!(MemorySmt::verify(
        "demo",
        &inputs.coprocessor_root.try_into().unwrap(),
        &inputs.ethereum_vault_shares_opening,
    ));
    assert!(MemorySmt::verify(
        "demo",
        &inputs.coprocessor_root.try_into().unwrap(),
        &inputs.neutron_vault_balance_opening,
    ));
    assert!(MemorySmt::verify(
        "demo",
        &inputs.coprocessor_root.try_into().unwrap(),
        &inputs.neutron_vault_shares_opening,
    ));
    // commit the rate as a public output
    sp1_zkvm::io::commit_slice(
        &borsh::to_vec(&RateApplicationCircuitOutputs {
            rate: ((neutron_balance + ethereum_balance) / (neutron_shares + ethereum_shares))
                .try_into()
                .unwrap(),
        })
        .unwrap(),
    );
}
```

Valence

4. Performance and Benchmarks

We have some initial benchmarks for our SP1 example applications (the mailbox and rate calculation examples).

These benchmarks will certainly change in the near future, but they are a good indication of how the first version of Valence ZK will perform in production.

We are actively exploring alternative proving systems and our modular architecture enables a high degree of flexibility even on an infrastructure/ ZK level.

M3 macbook pro with 64 GB ram

test name	elapsed time
rate	149.5s
rate + coprocessor	300.5s
mailbox	147.1s
mailbox + coprocessor	288.6s

SP1 prover network

test name	elapsed time
rate	35s
rate + coprocessor	65.1s
mailbox	29.2s
mailbox + coprocessor	54.6s

Valence