Implementing covenants and Circle STARK verifier with OP_CAT

Weikeng Chen and Pingzhou Yuan

L2IV through a grant from its LP

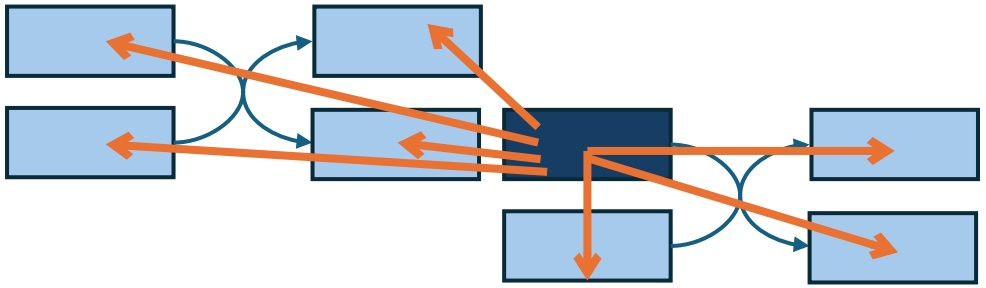
STARKWARE

Covenants

Covenants

Andrew Poelstra showed that with OP_CAT, the Bitcoin script can obtain a hash of the CheckSigVerify preimage, through a Schnorr signature over a dummy public key.

This allows us to:
- build covenants
- reflect previous outputs



Rijndael | BIP-420 🐱 🔵 🚁
@rot13maxi

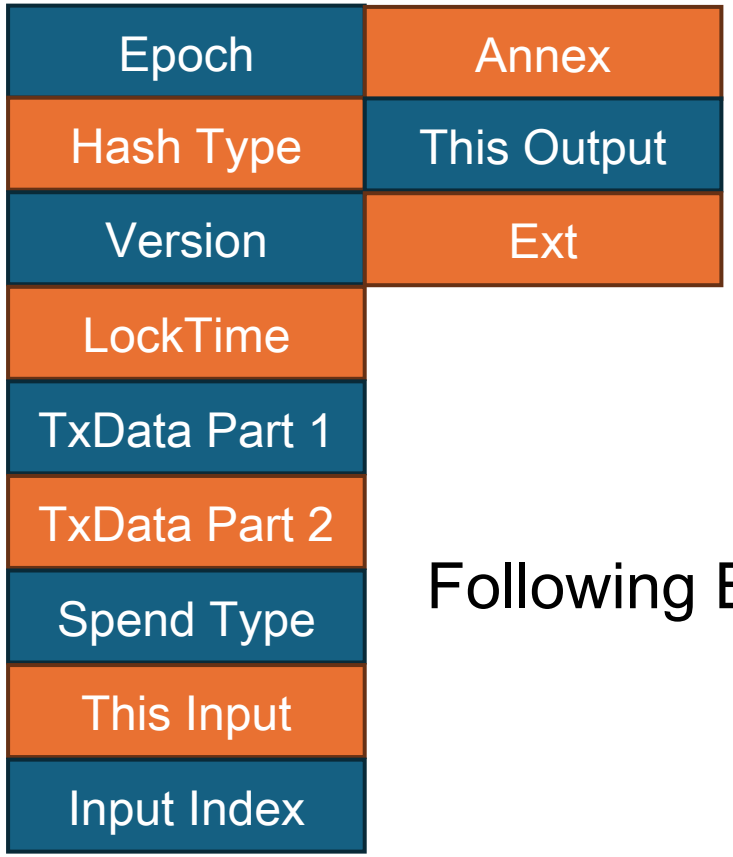
Wanna see a covenant with OP_CAT?

This script enforces that these coins can ONLY be spent if they send 0.999 BTC to bcrt1py9ccnmdrk9z4ylvgt68htyazmssvsz0cdzjcm3p3m75dsc0j203q37qzse. No other transaction with them is valid.

```
OP_TOALTSTACK OP_CAT OP_CAT OP_CAT OP_CAT de890a8209d796493ee7bac9a58b62fbced10ccb7311e24f26c461c079ead08c OP_SWAP OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT OP_CAT 5461705369676861736 8 OP_SHA256 OP_DUP OP_ROT OP_CAT OP_CAT OP_SHA256 4249503033430 2f6368616c6c656e6765 OP_SHA256 OP_DUP OP_ROT 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798 OP_DUP OP_DUP OP_TOALTSTACK 2 OP_ROLL OP_CAT OP_CAT OP_CAT OP_CAT OP_SHA256 OP_FROMALTSTACK OP_SWAP OP_CAT OP_FROMALTSTACK OP_DUP 1 OP_CAT OP_ROT OP_EQUALVERIFY 2 OP_CAT 79be667ef9dcbbac55a06295ce870b07029bfcdb2dce28d959f2815b16f81798 OP_CHECKSIG
```

The only opcode it uses that's not currently active on Bitcoin is CAT.

Two steps of building a covenant

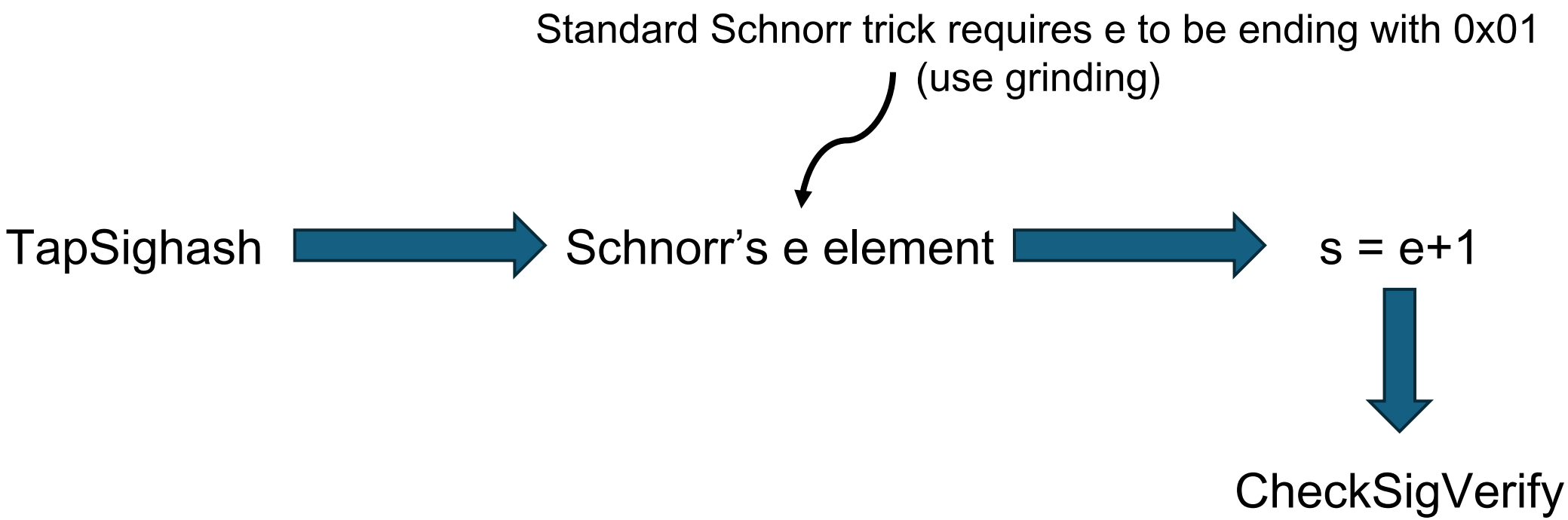


Following BIP-341

TapSighash

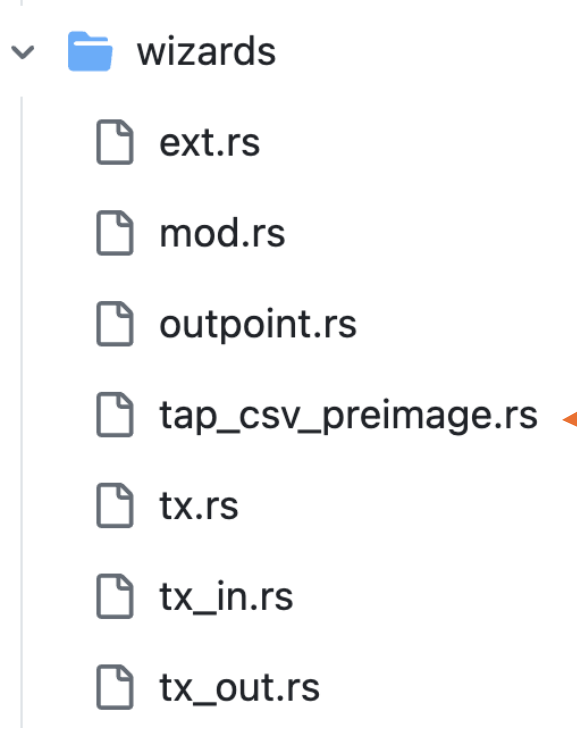
Assemble CheckSigVerify preimage

Two steps of building a covenant

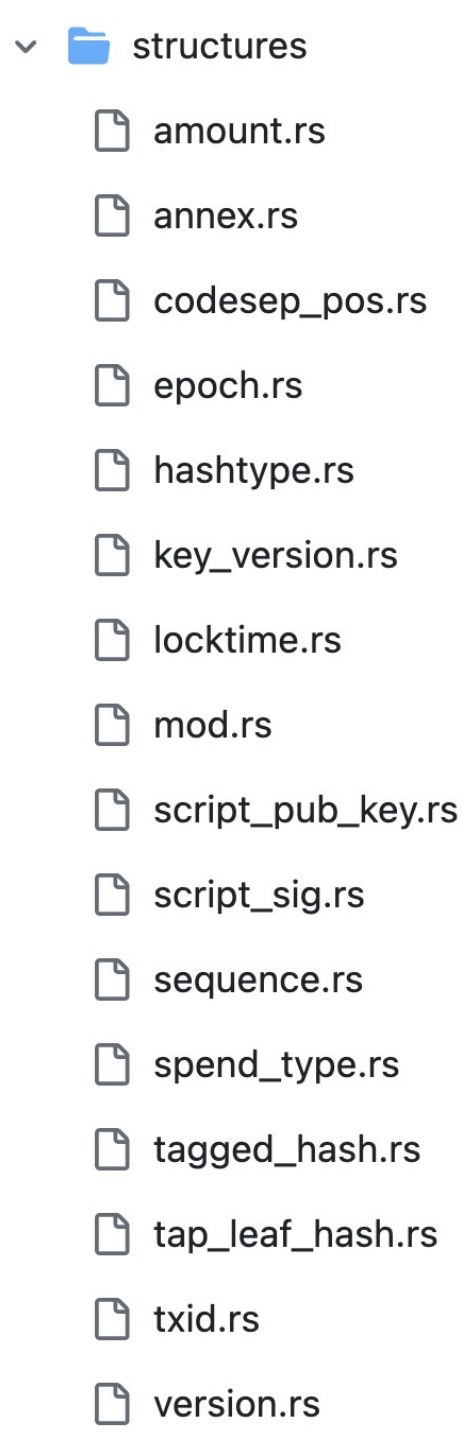


Implementation

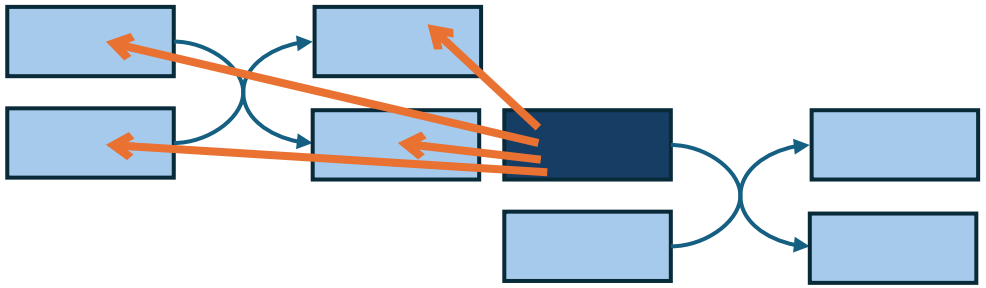
- https://github.com/Bitcoin-Wildlife-Sanctuary/covenants-gadgets



All the Bitcoin scripts needed for CheckSigVerify

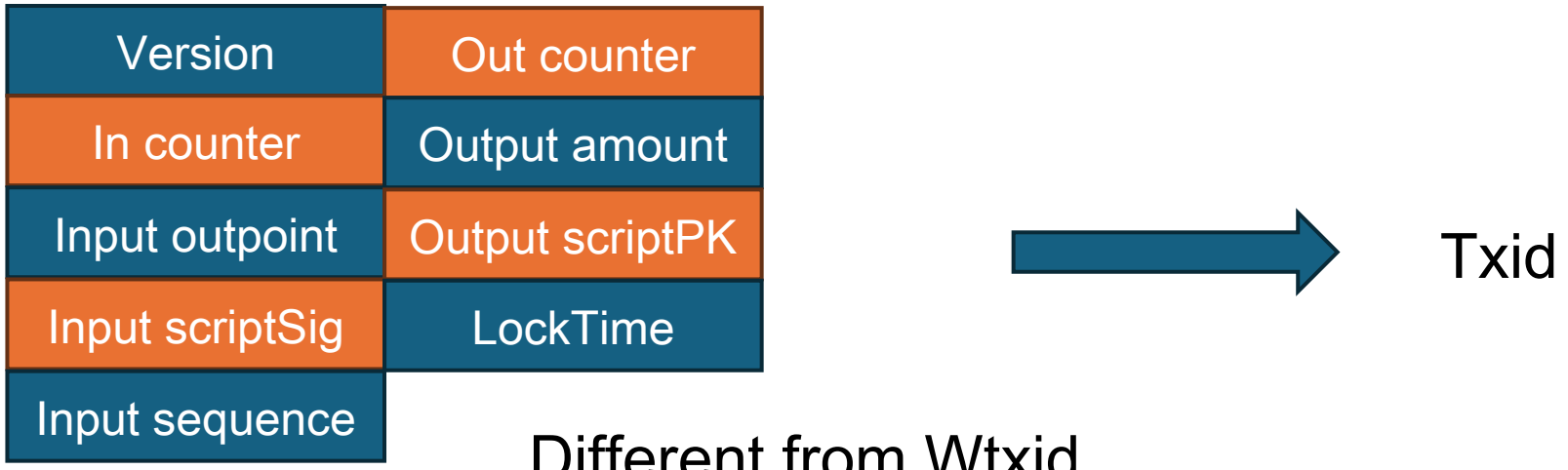


Two steps of reflecting a previous transaction



Through the TXID obtained from covenants

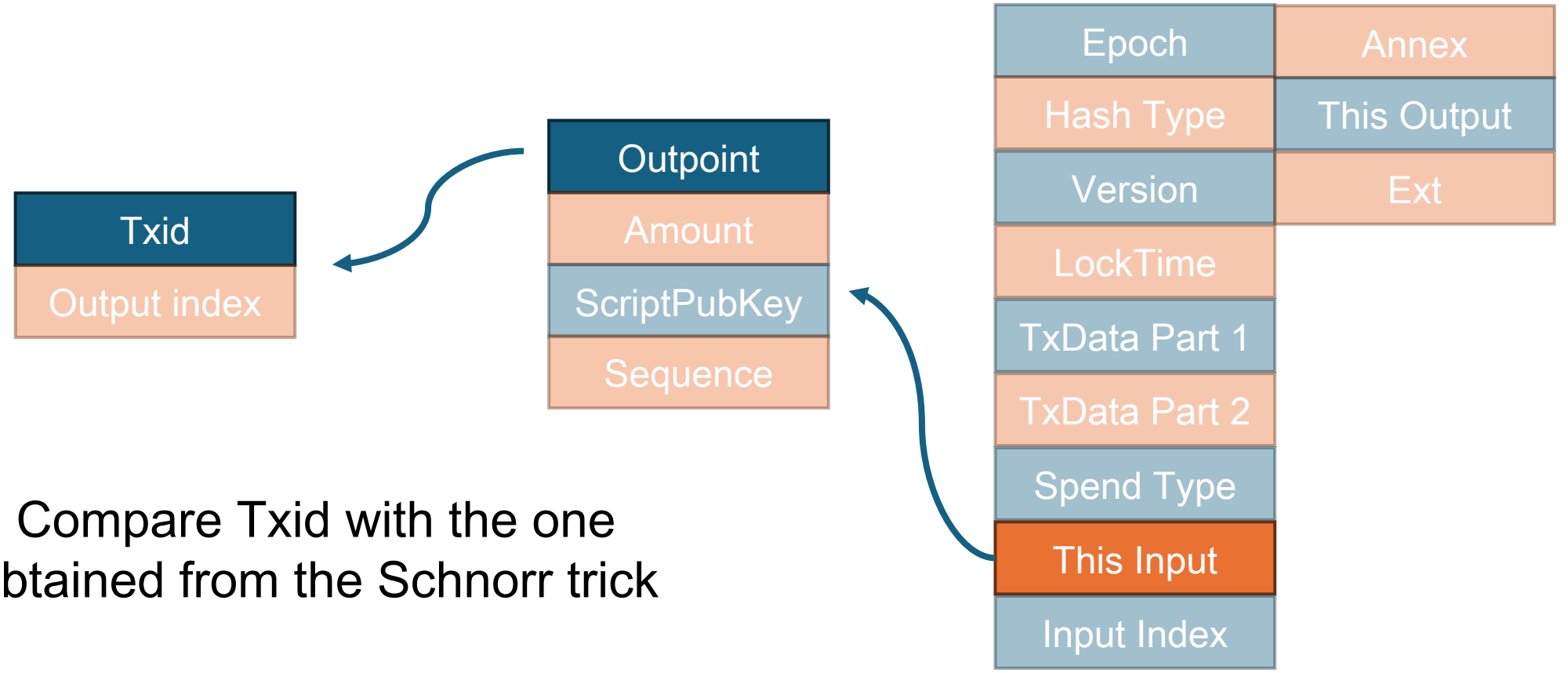
Two steps of reflecting a previous transaction



Different from Wtxid

Assemble Txid preimage

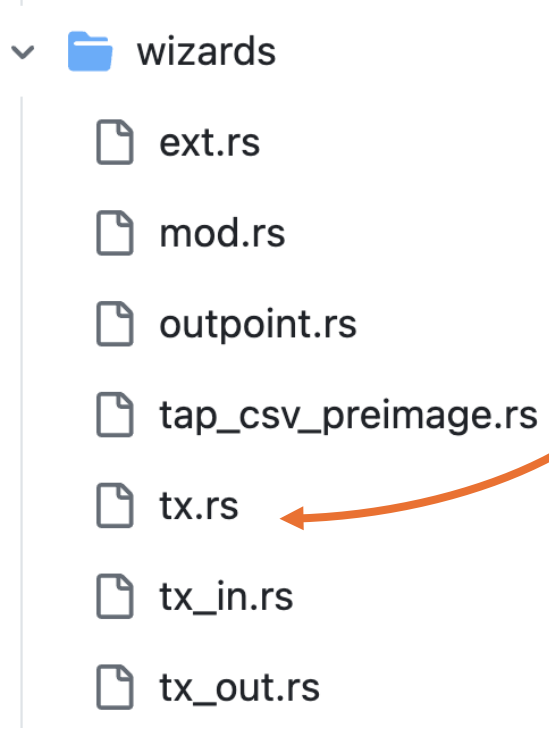
Two steps of reflecting a previous transaction



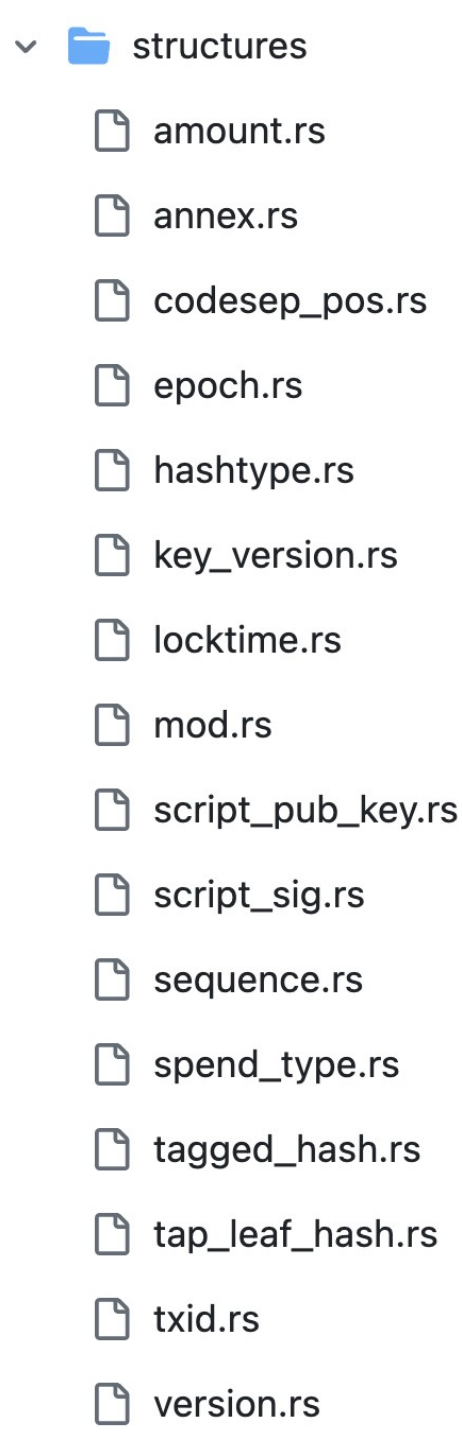
Compare Txid with the one obtained from the Schnorr trick

Implementation

- https://github.com/Bitcoin-Wildlife-Sanctuary/covenants-gadgets



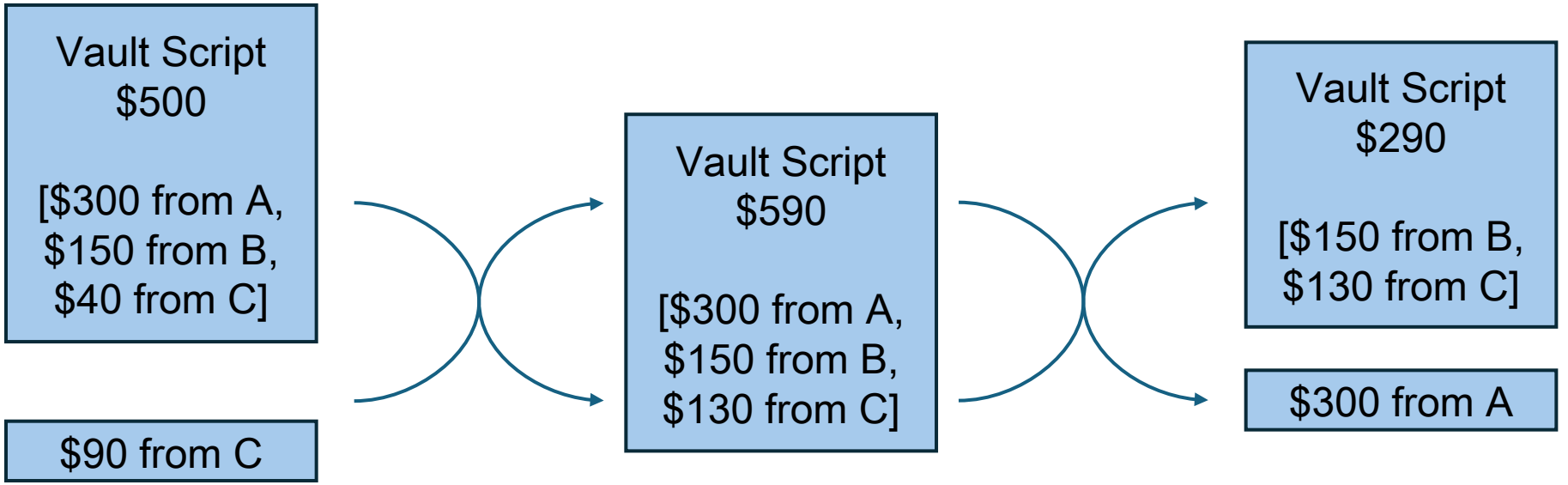
All the Bitcoin scripts needed for opening up txid



State-carrying UTXOs

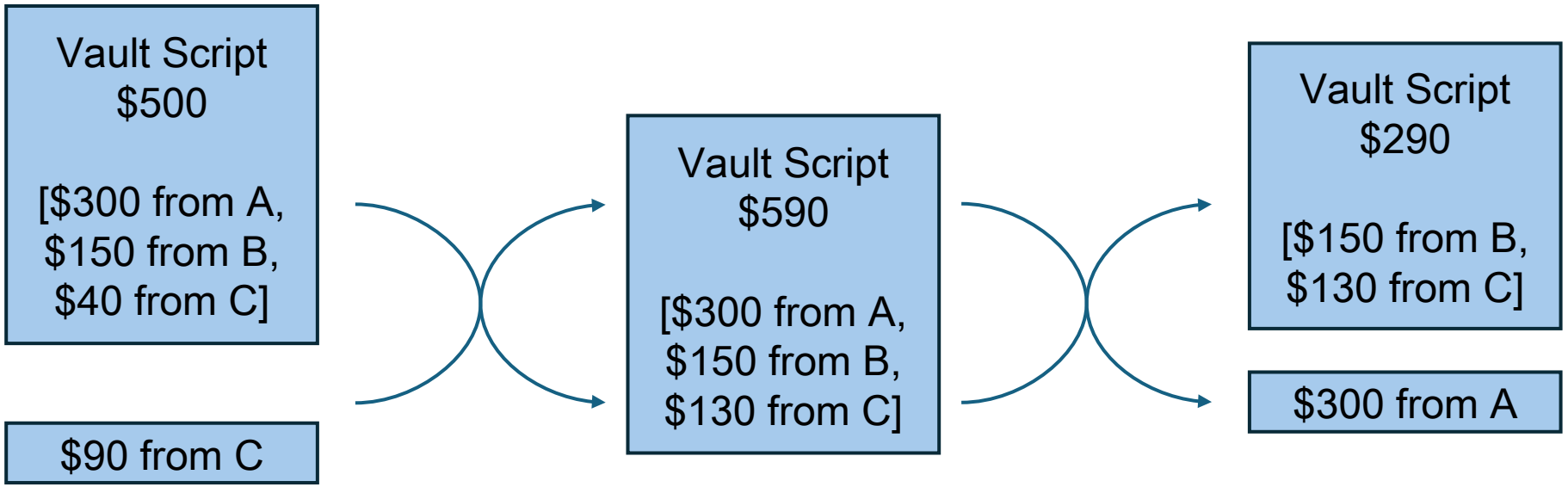
Passing data to the next program

To construct smart contracts on chain, we need to be able to carry data.



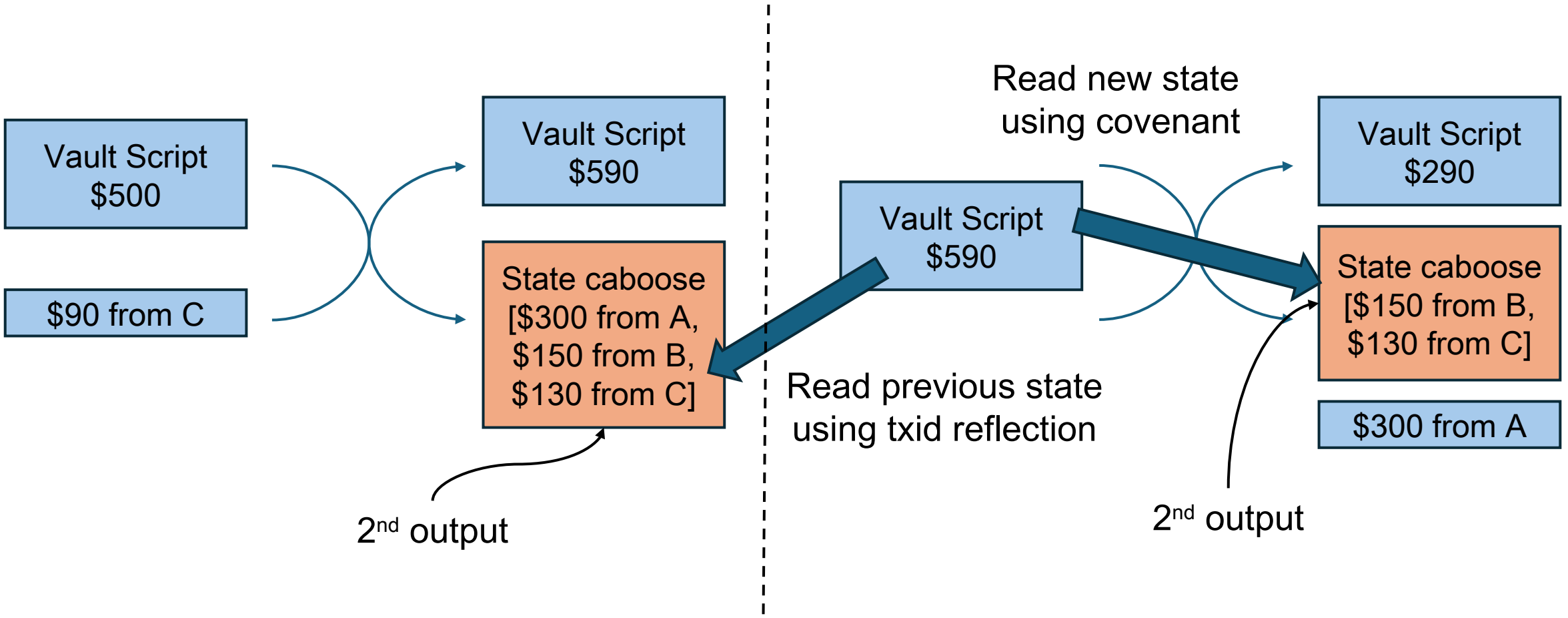
The naïve solution is to embed the state data in the new UTXO's vault script.

Naïve solution doesn't work well

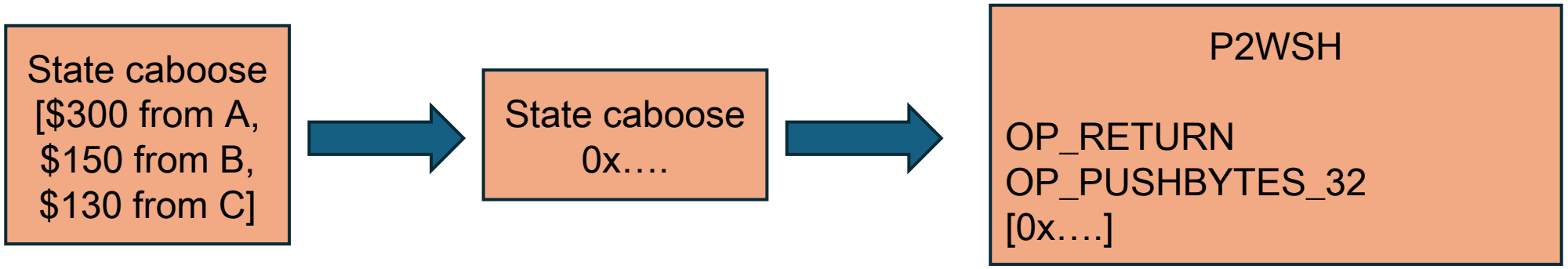


For P2TR, the scriptPubKey is an elliptic curve point tweaked by the script hash. Computing the new vault's scriptPubKey will be expensive, especially when the script exceeds the OP_CAT output limit (520 bytes).

Solution: state caboose

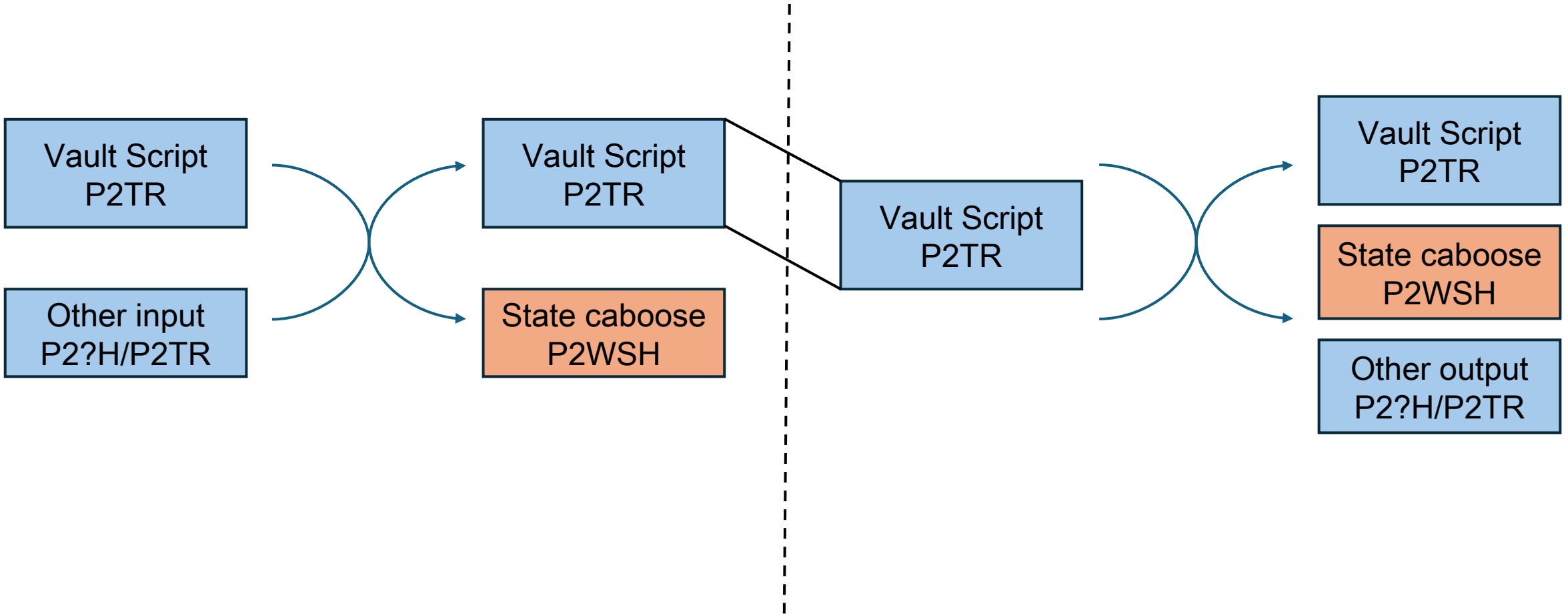


State caboose only stores a hash of the state



Vault script can ask the caller to include the state preimage in the unlocking script (scriptSig), so the state caboose can just keep a hash of the state for binding.

Solution: state caboose

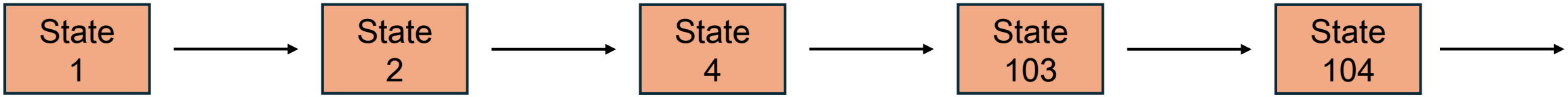


Example: a counter-carrying program

- https://github.com/Bitcoin-Wildlife-Sanctuary/covenants-examples

Count-update script

Three functions:
- Increase by 1
- Increase by 2
- Increase by a caller-provided number if it is <100



Implementation

```rust
/// Trait for a covenant program.
pub trait CovenantProgram {
    /// Type of the state for this covenant program.
    type State: Into<Script> + Debug + Clone;

    /// Type of input (could be an enum).
    type Input: Into<Script> + Clone;

    /// Unique name for caching.
    const CACHE_NAME: &'static str;

    /// Create an empty state.
    fn new() -> Self::State;

    /// Compute the state hash, which is application-specific.
    fn get_hash(state: &Self::State) -> Vec<u8>;

    /// Get all the scripts of this application.
    fn get_all_scripts() -> BTreeMap<usize, Script>;

    /// Get the common prefix script.
    fn get_common_prefix() -> Script;

    /// Run the program to move from the previous state to the new state.
    fn run(id: usize, old_state: &Self::State, input: &Self::Input) -> Result<Self::State>;
}
```

Implementation

```
// increase by a given number as long as it is smaller than 100
map.insert(
    456789,
    script! {
        // stack:
        // - old counter
        // - new counter

        OP_HINT
        OP_DUP 0 OP_GREATERTHANOREQUAL OP_VERIFY
        OP_DUP 100 OP_LESSTHAN OP_VERIFY
        OP_SUB OP_EQUAL
    },
);
```


```
fn get_all_scripts() -> BTreeMap<usize, Script> {
    let mut map = BTreeMap::new();
    // increase by 1
    map.insert(
        123456,
        script! {
            // stack:
            // - old counter
            // - new counter
            OP_1SUB OP_EQUAL
        },
    );
    // increase by 2
    map.insert(
        123457,
        script! {
            // stack:
            // - old counter
            // - new counter
            OP_1SUB OP_1SUB OP_EQUAL
        },
    );
```

```rust
fn run(id: usize, old_state: &Self::State, input: &Self::Input) -> Result<Self::State> {
    if id == 123456 {
        Ok(CounterState {
            counter: old_state.counter + 1,
        })
    } else if id == 123457 {
        Ok(CounterState {
            counter: old_state.counter + 2,
        })
    } else if id == 456789 {
        assert!(input.0.is_some());

        let input = input.0.unwrap();
        assert!(input < 100);

        Ok(CounterState {
            counter: old_state.counter + input,
        })
    } else {
        unimplemented!()
    }
}
```

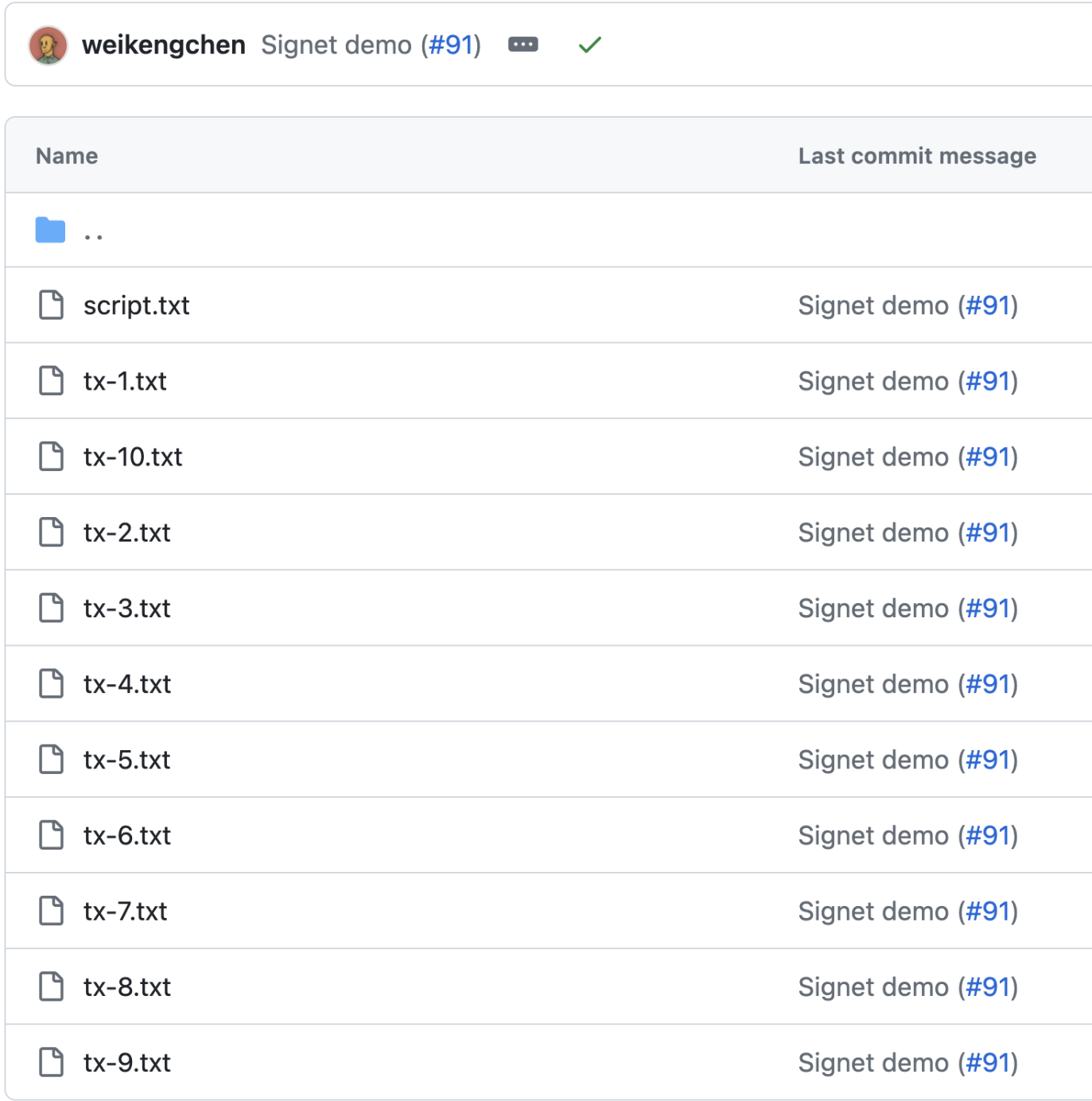
Implementation

Toy circle STARK verifier

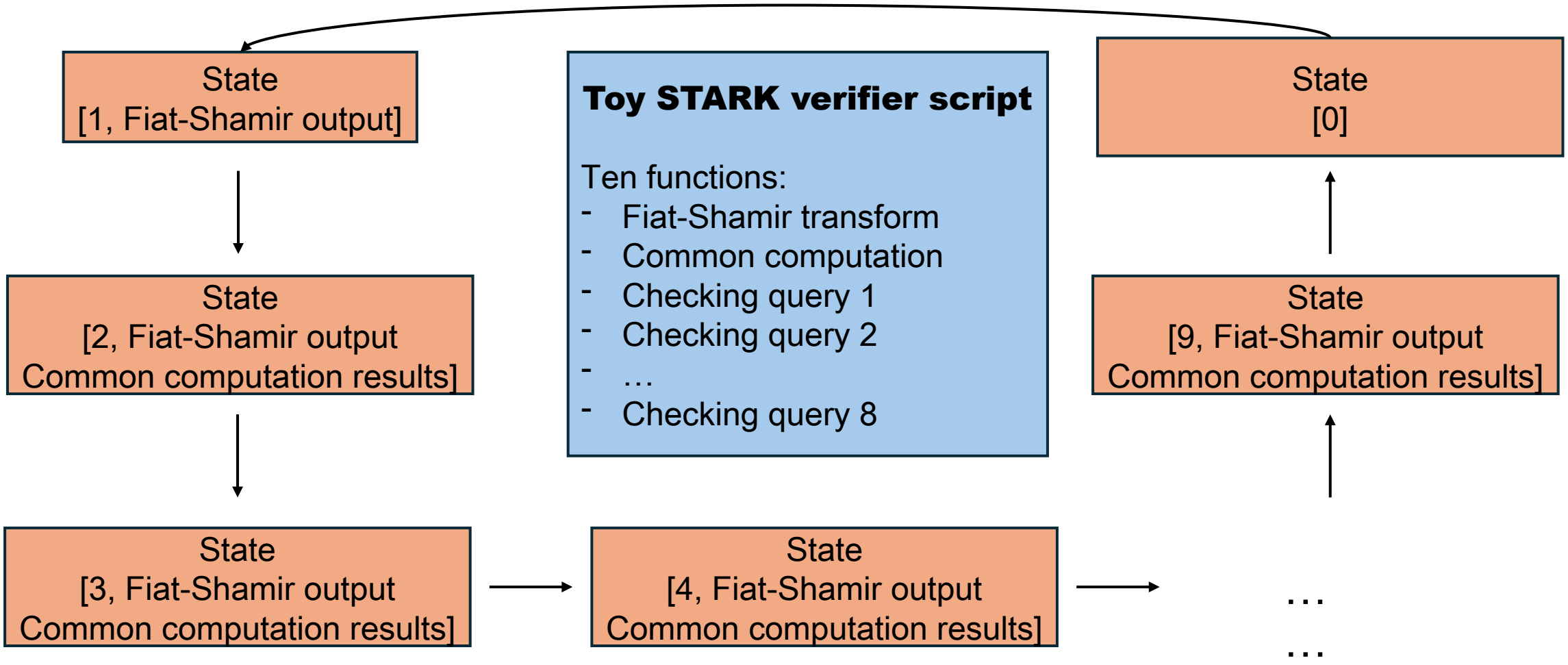
Script layout

We split the verifier into 10 transactions.

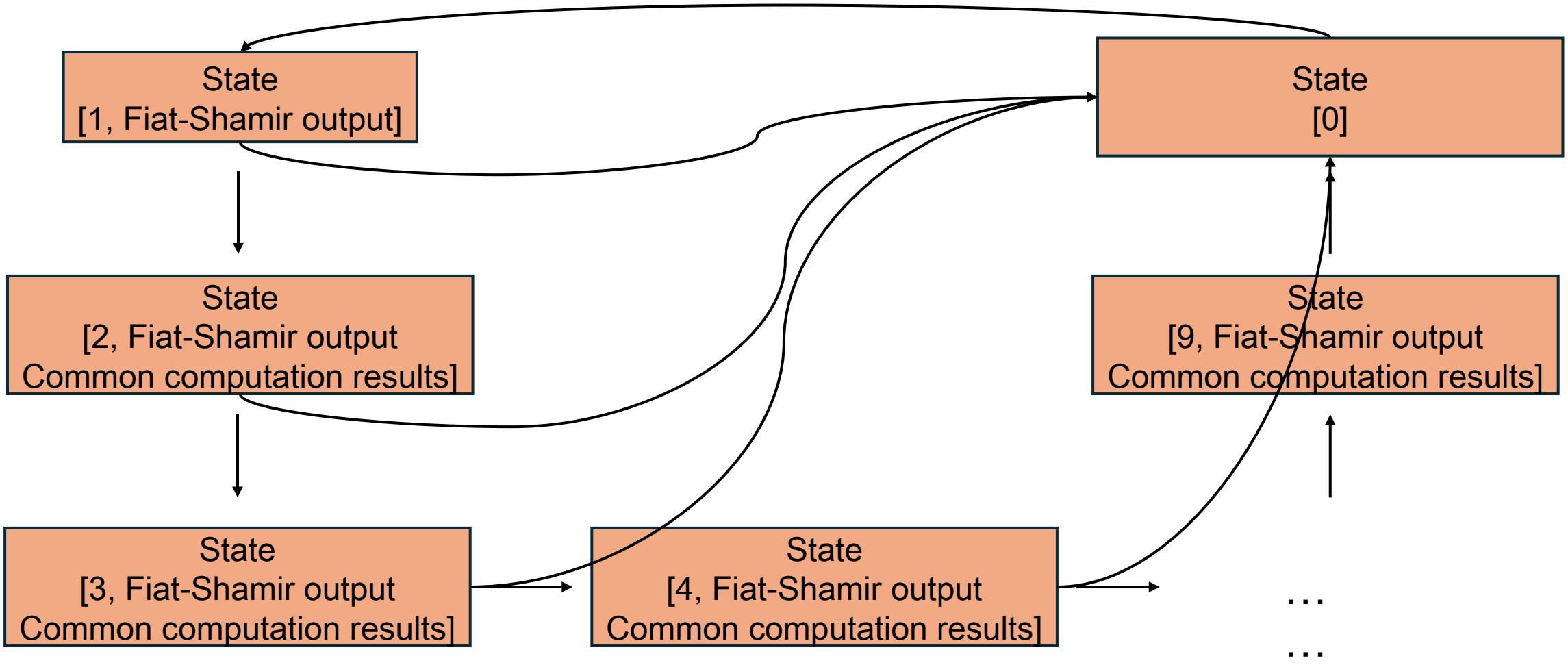
- One handling Fiat-Shamir transformation.
- One handling common computation.
- 8 more transactions, each handling one of the 8 FRI queries



Chaining them together



Dynamic control flow: RESET as an example

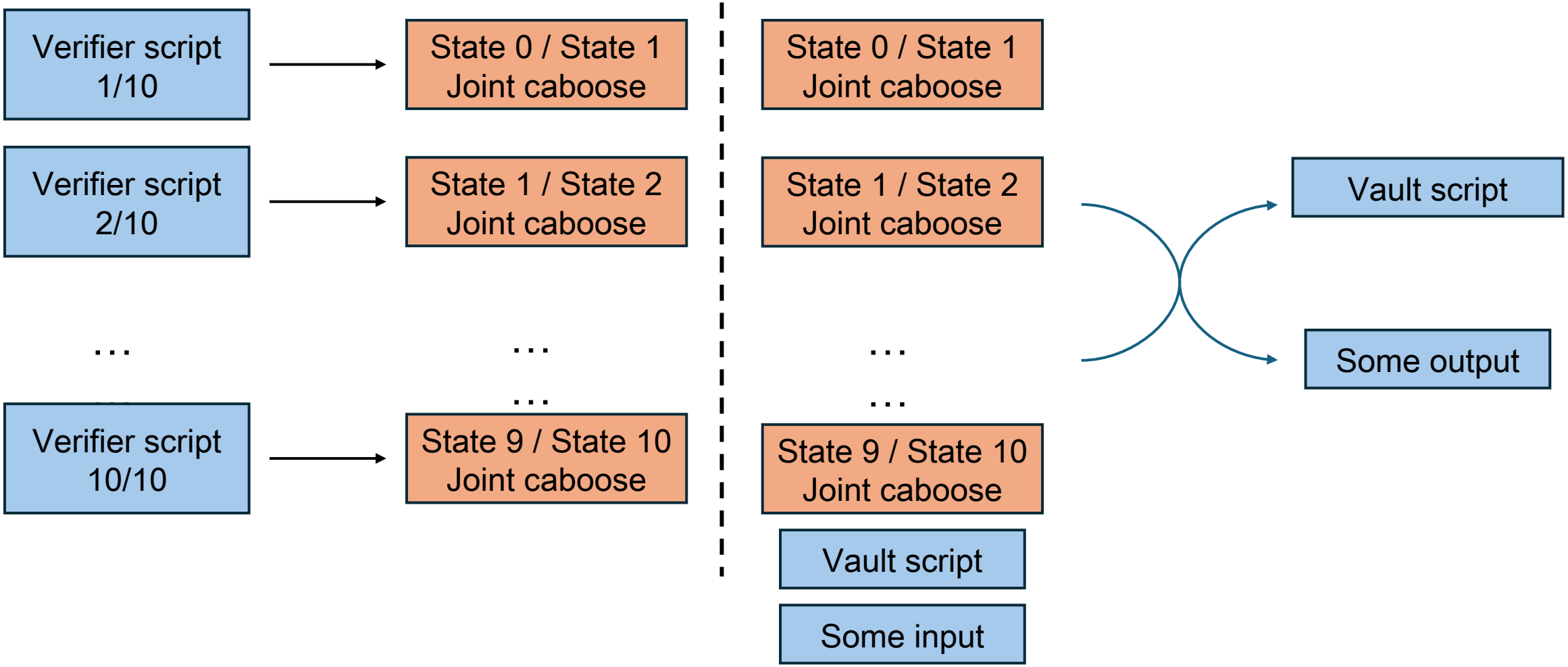


Next steps

Mempool-friendly transaction flow

- Although the ten transactions can, in theory, be submitted together and settled in 1 block, in our experiment, we can only push one transaction per block.
- Due to default mempool limits on limitancestorsize, limitdescendantsize.
- **Solution:** we are planning on a new transaction flow design that bypasses limitancestorsize and limitdescendantsize.

Mempool-friendly transaction flow



STARK for General Computation: mini-Plonk

- Mini-Plonk (from Starkware) is a simplified Plonk proof system that, compared with the toy example, provides general computation.

https://github.com/Bitcoin-Wildlife-Sanctuary/circle-plonk
https://github.com/Bitcoin-Wildlife-Sanctuary/r1cs-to-circle-plonk

Thank you