Covenants and STARK proof verification with OP_CAT

Weikeng Chen

L2IV

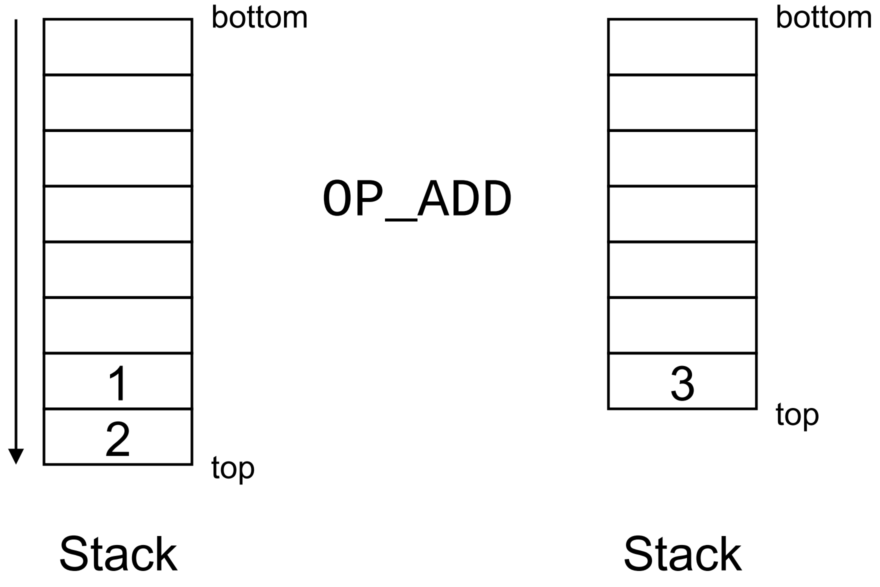
Bitcoin Programmability

- Bitcoin script and EVM are both stack-based virtual machines.

- EVM is more powerful than Bitcoin:
  - State
  - Function calls
  - Concurrency
  - Crypto accelerators
  - Send and receive money

Bitcoin script

Stack machine and opcode



Bitcoin has a lot of opcodes.

- If-else
  - `OP_IF OP_ELSE OP_ENDIF` …
- 32-bit integer math
  - `OP_ADD OP_SUB` …
- Hash
  - `OP_SHA256` …
- Signature verification
  - `OP_CHECKSIGVERIFY` …

Example of Bitcoin script

```
OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1b OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD
OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1b OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1b OP_ROLL OP_PUSHNUM_8
OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD
OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000
OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f
OP_ROLL OP_PUSHNUM_6 OP_PICK OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_7 OP_PICK OP_ROT OP_ADD OP_ADD OP_DUP
OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_7 OP_PICK OP_ROT OP_ADD OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP
OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_7 OP_PICK OP_ROT OP_ADD OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1
20 OP_ROLL OP_PUSHNUM_7 OP_PICK OP_ROT OP_ADD OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_7 OP_PICK OP_ROT OP_ADD
OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_15 OP_PICK OP_ROT OP_ADD OP_ADD OP_DUP OP_PUSHBYTES_2 8000 OP_ADD OP_PICK
OP_SWAP OP_PUSHBYTES_2 b000 OP_ADD OP_PICK OP_PUSHBYTES_1 20 OP_ROLL OP_PUSHNUM_15 OP_PICK OP_ROT OP_ADD OP_ADD OP_PUSHBYTES_1 7f OP_ADD OP_PICK OP_PUSHBYTES_1 1c OP_ROLL OP_PUSHNUM_8 OP_PICK
OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1c OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK
OP_PUSHBYTES_1 1c OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1c OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD
OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1c OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL
OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2
ef00 OP_ADD OP_PICK OP_PUSHBYTES_1 1f OP_ROLL OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 e000 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ef00 OP_ADD OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 0002 OP_ADD OP_PICK
OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP
OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD
OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK
OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD OP_PICK OP_PUSHNUM_6 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_6 OP_ROLL OP_PUSHBYTES_2 0002
OP_ADD OP_PICK OP_PUSHNUM_8 OP_PICK OP_PUSHBYTES_2 f101 OP_ADD OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHNUM_8 OP_ROLL OP_PUSHBYTES_2 0002 OP_ADD OP_PICK OP_PUSHNUM_8 OP_ROLL OP_PUSHBYTES_2 f001 OP_ADD
OP_PICK OP_SWAP OP_SWAP OP_ADD OP_PUSHBYTES_1 7f OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2 df00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ee00 OP_ADD OP_PICK OP_PUSHBYTES_1 7e OP_ROLL OP_PUSHBYTES_1 30
OP_ROLL OP_PUSHBYTES_2 de00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ed00 OP_ADD OP_PICK OP_PUSHBYTES_1 7d OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2 dd00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ec00
OP_ADD OP_PICK OP_PUSHBYTES_1 7c OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2 dc00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 eb00 OP_ADD OP_PICK OP_PUSHBYTES_1 7b OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL
OP_PUSHBYTES_2 db00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 ea00 OP_ADD OP_PICK OP_PUSHBYTES_1 7a OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2 da00 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e900 OP_ADD
OP_PICK OP_PUSHBYTES_1 79 OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2 d900 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e800 OP_ADD OP_PICK OP_PUSHBYTES_1 78 OP_ROLL OP_PUSHBYTES_1 30 OP_ROLL OP_PUSHBYTES_2
d800 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e700 OP_ADD OP_PICK OP_PUSHBYTES_1 5f OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d700 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e600 OP_ADD OP_PICK
OP_PUSHBYTES_1 5e OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d600 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e500 OP_ADD OP_PICK OP_PUSHBYTES_1 5d OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d500
OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e400 OP_ADD OP_PICK OP_PUSHBYTES_1 5c OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d400 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e300 OP_ADD OP_PICK OP_PUSHBYTES_1 5b
OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d300 OP_ADD OP_PICK OP_ADD OP_PUSHBYTES_2 e200 OP_ADD OP_PICK OP_PUSHBYTES_1 5a OP_ROLL OP_PUSHBYTES_1 18 OP_ROLL OP_PUSHBYTES_2 d200 OP_ADD OP_PICK OP_ADD
```

Bitcoin lacks certain opcodes

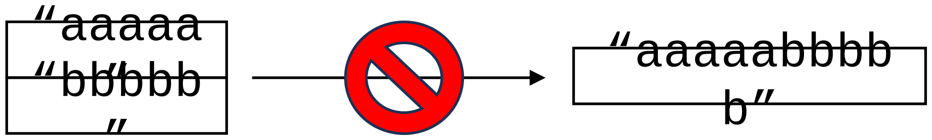
- No `OP_MUL` `OP_DIV`
  - Multiplication and division can be emulated using `OP_ADD` and others

- No `OP_XOR`
  - XOR can be emulated by lookup table

Bitcoin lacks certain opcodes

- No `OP_SEND_BTC OP_GET_INPUT OP_GET_OUTPUT`
  - The only opcodes that involve the transaction is signature verification.
  - Covenant is [provably] not possible.

- No `OP_CAT`
  - There is [provably] no way to concatenate two strings that are longer than 4 bytes.



Consequences

- Some operations are possible but slow
- Some operations are [provably] impossible

Merkle tree is possible, but...

- Without `OP_CAT`, one needs to implement a hash function in Bitcoin script (and cannot use the `OP_SHA256` opcode)

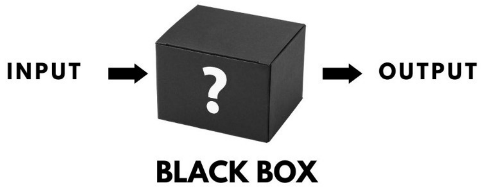
- Best result: Blake3, 46k script per 512 bits
  - Simulate 32-bit additions using `OP_ADD OP_SUB OP_GREATERTHAN`
  - Simulate XOR using a lookup table
- USD $14.95 per layer in the Merkle tree, assuming 2sat/vByte
- If we use Merkle tree for a 4GB memory (each leaf is 32-bit), writing a leaf costs USD $600.

- Wasting blockspace: doing the script 100 times needs to repeat the script 100 times

Covenant is impossible

- A transaction has inputs and outputs.

- There is [provably] no way for a script to constrain outputs other than going through signature verification.
- Signatures are “black-box” to the script.



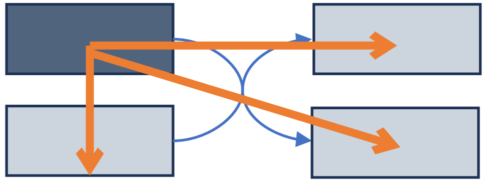
Covenant with OP_CAT 😺

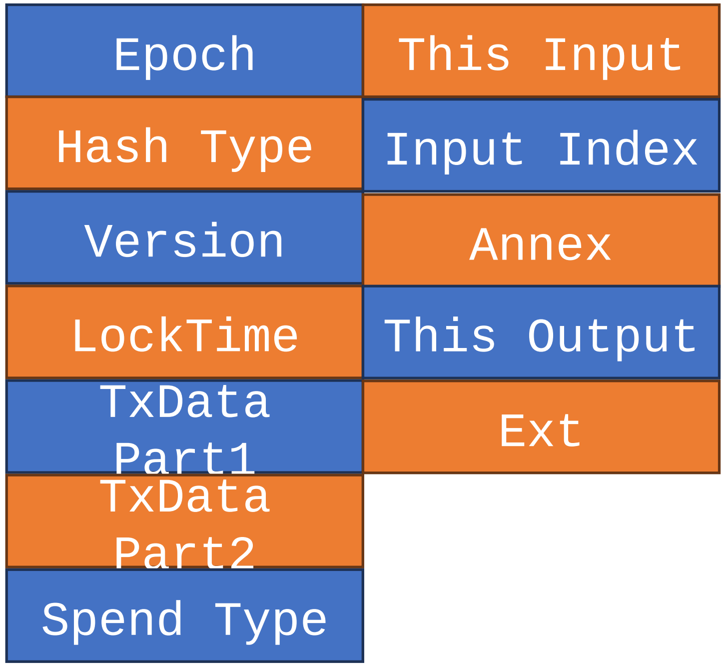
Schnorr trick

- With OP_CAT, there is a way to obtain a "hash" of the current transaction, through the Schnorr signature, proposed by Andrew Poelstra.
- Idea:
  - Compute a signature for the given transaction as follows:
    - Use a dummy secret key $sk = 1$
    - Use $k = 1$ for the randomizer
    - The signature is (R, s) where R is a known constant and $s = H + 1$, where H is a SigHash of the transaction
  - In the script,
    - Given the SigHash H, compute s
    - Check if (R, s) is a valid signature of the transaction under the dummy public key
      - To assemble the signature, OP_CAT is needed here and seems unavoidable

Open up SigHash with OP_CAT

- SigHash in Taproot is the hash of several components of the transaction.
- Given the transaction information, the script can reconstruct the transaction, compute its SigHash, and check if it is the same SigHash from the Schnorr trick.
    - The reconstruction requires OP_CAT and it seems impossible to bypass.
- This allows the script to constrain inputs and outputs.

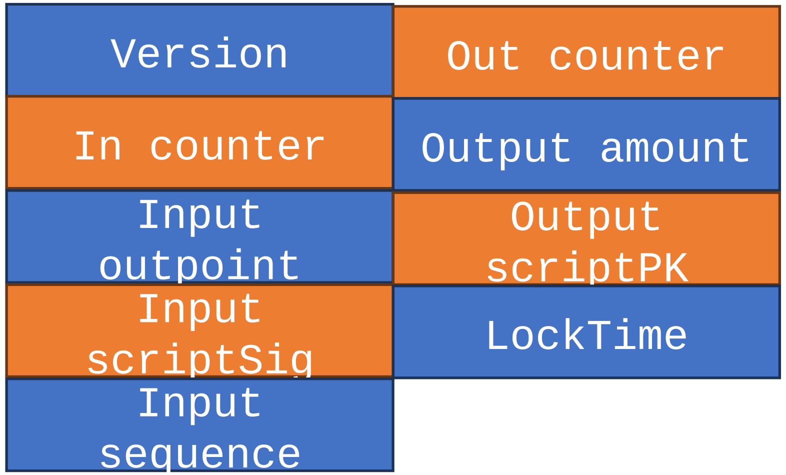


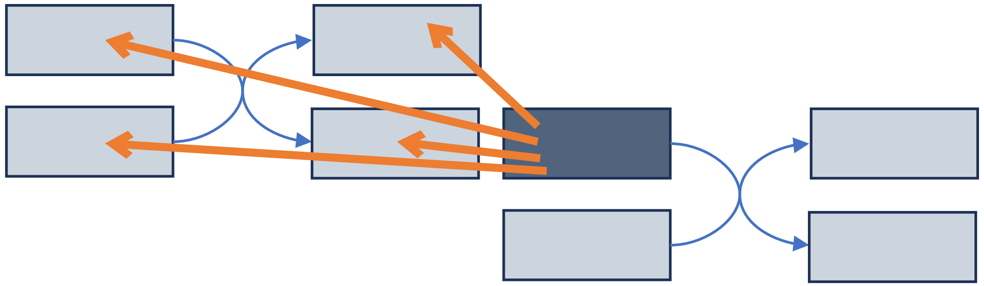


Txid reflection with OP_CAT

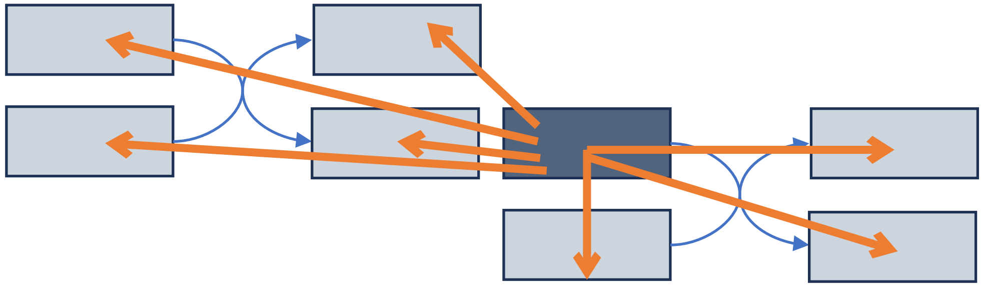
- SigHash can provide the txid of input UTXOs
- Txid is a hash of the transaction (with certain information removed)

- Given a transaction, the script can compute its txid and compare if it matches the txid of a certain input.
- This allows the script to “reflect” on the previous transaction.

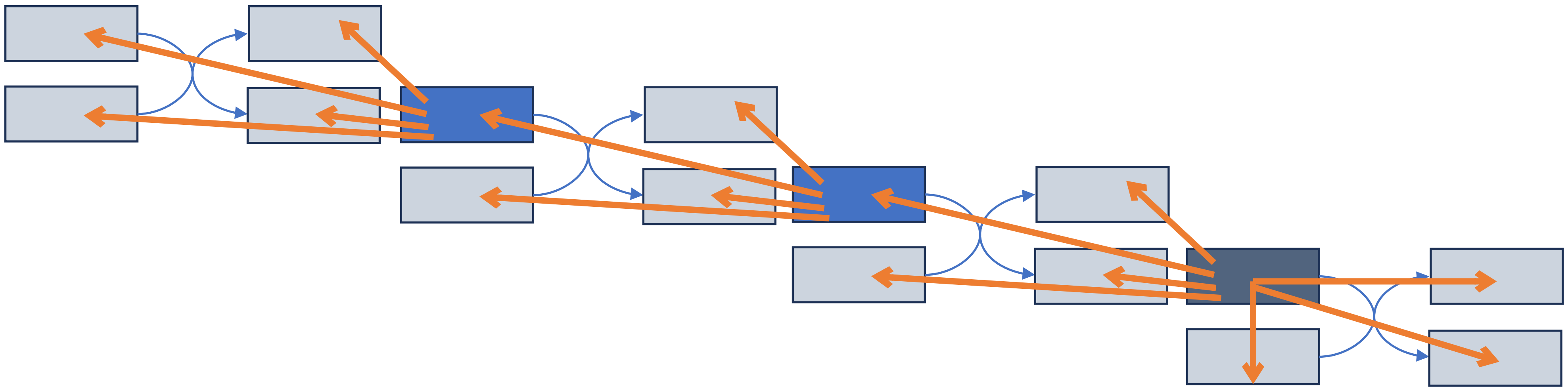




Covenant with OP_CAT



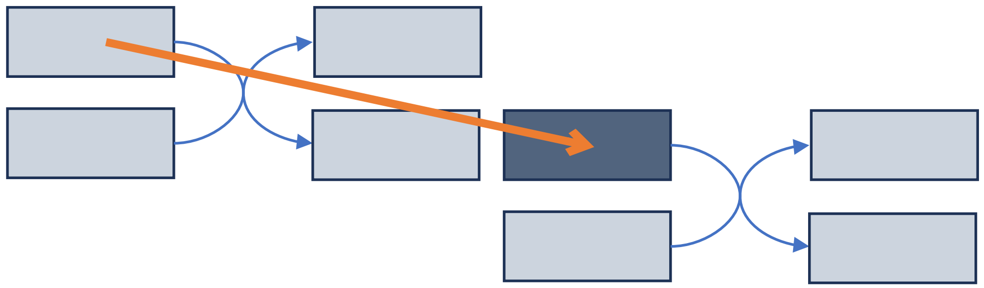
Covenant with OP_CAT



New tools from OP_CAT

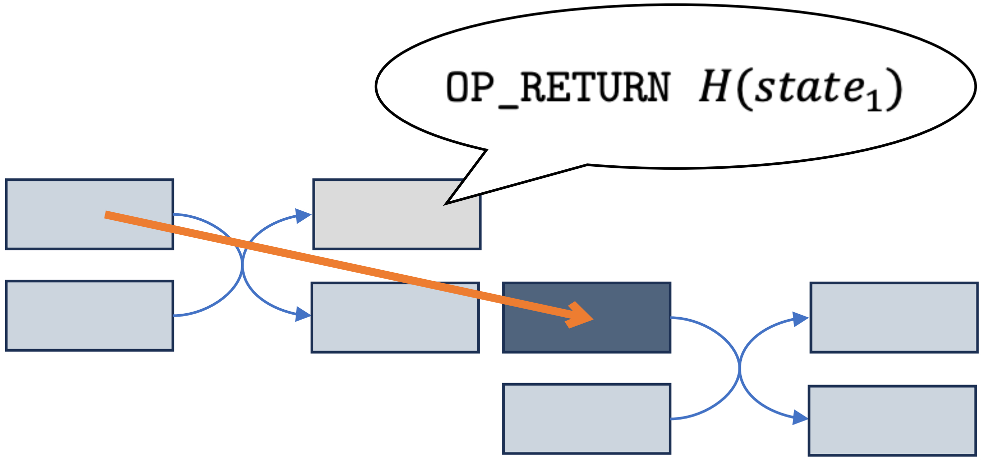
State

- Problem: how to pass the memory data, as a state, to the next execution?



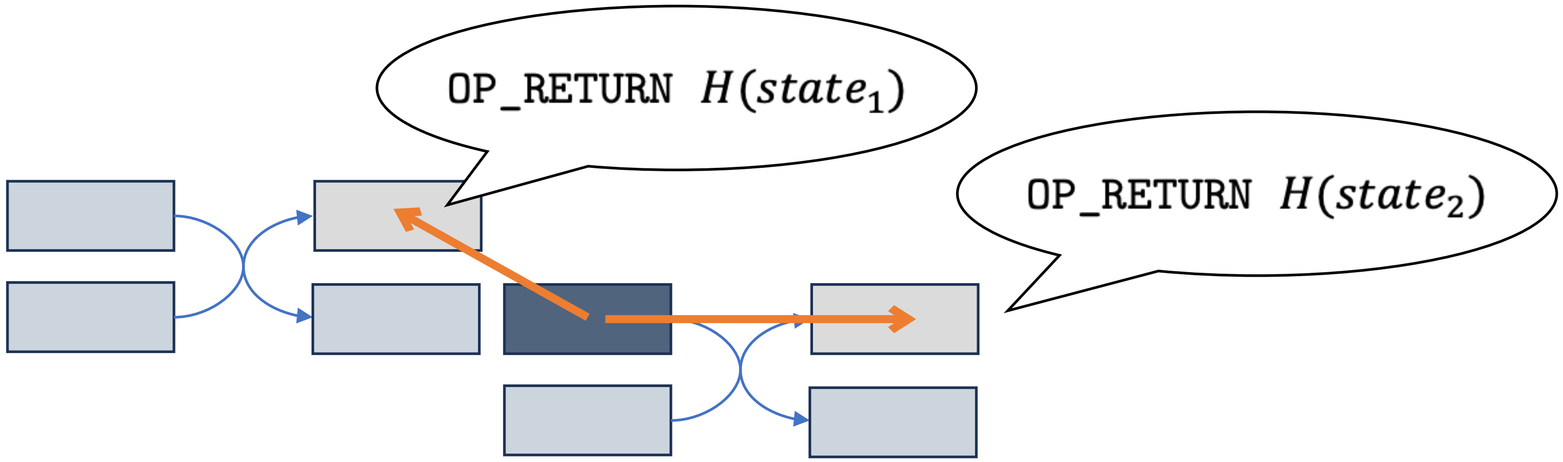
State caboose

- A design called “state caboose” or “state caboose hash” is a way to **commit** the state in the transaction.



State caboose

- The next computation can use txid reflection to read the state commitment and commit its new state similarly.



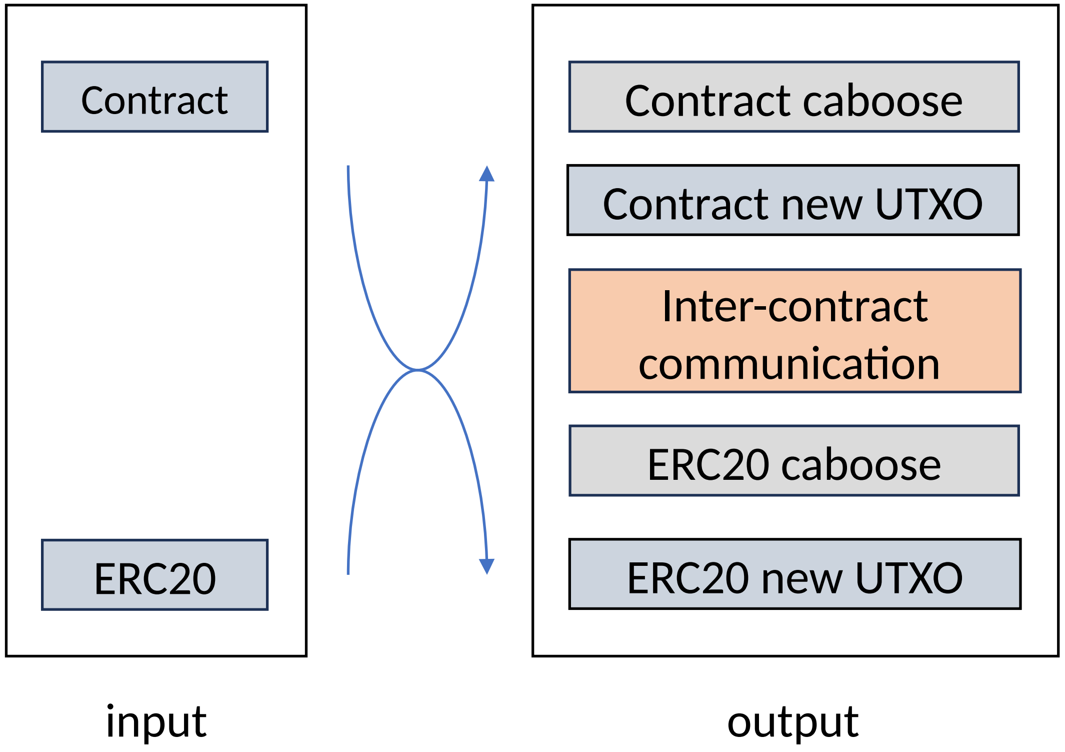
State enables ERC20

- With Merkle trees and state, one can implement ERC20.

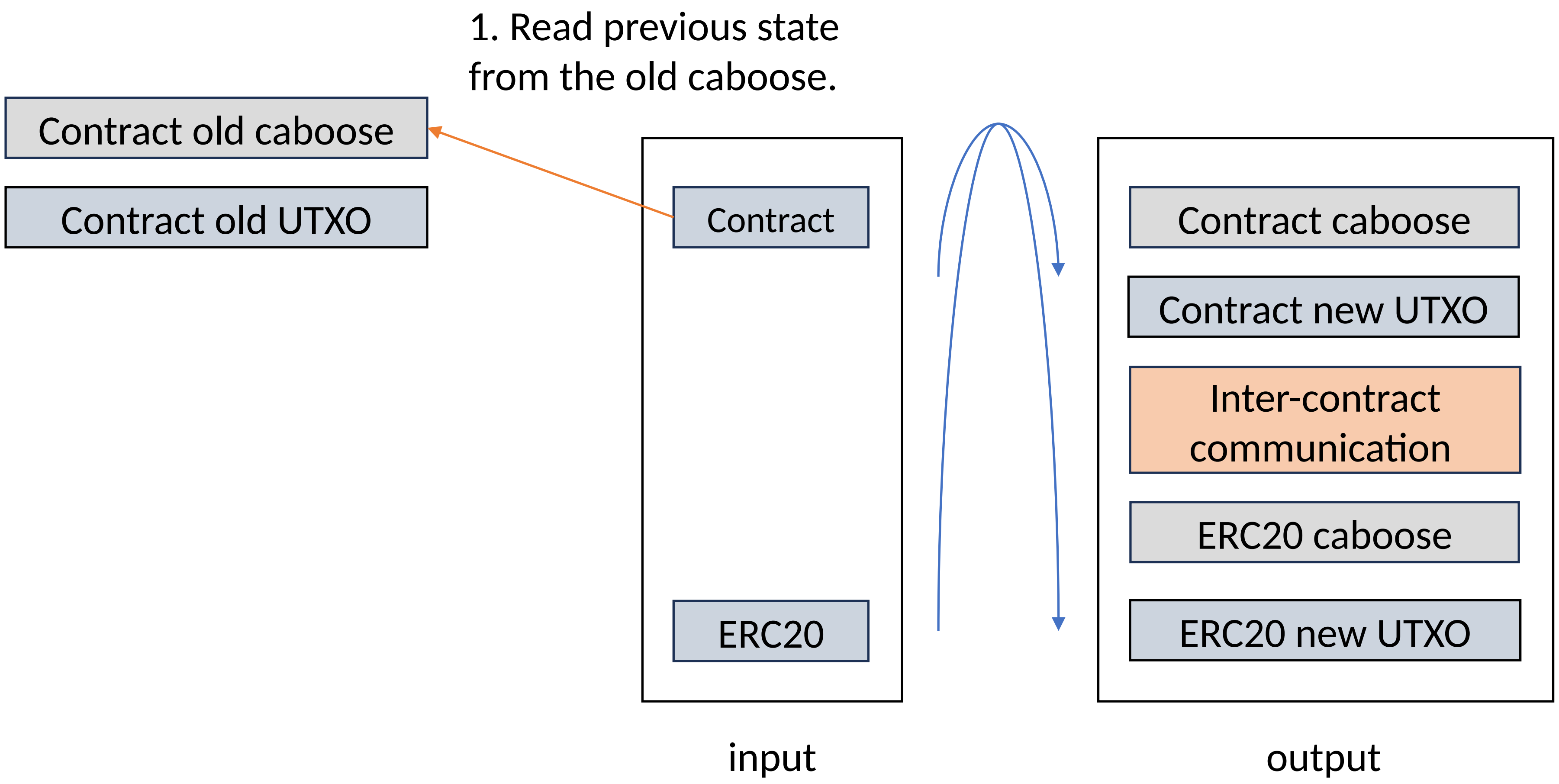
- Token transfer:
  - Check the sender has provided authorization (e.g., CheckSigVerify)
  - Subtract the number of tokens by N of the sender
  - Increase the number of tokens by N for the receiver

Function calls

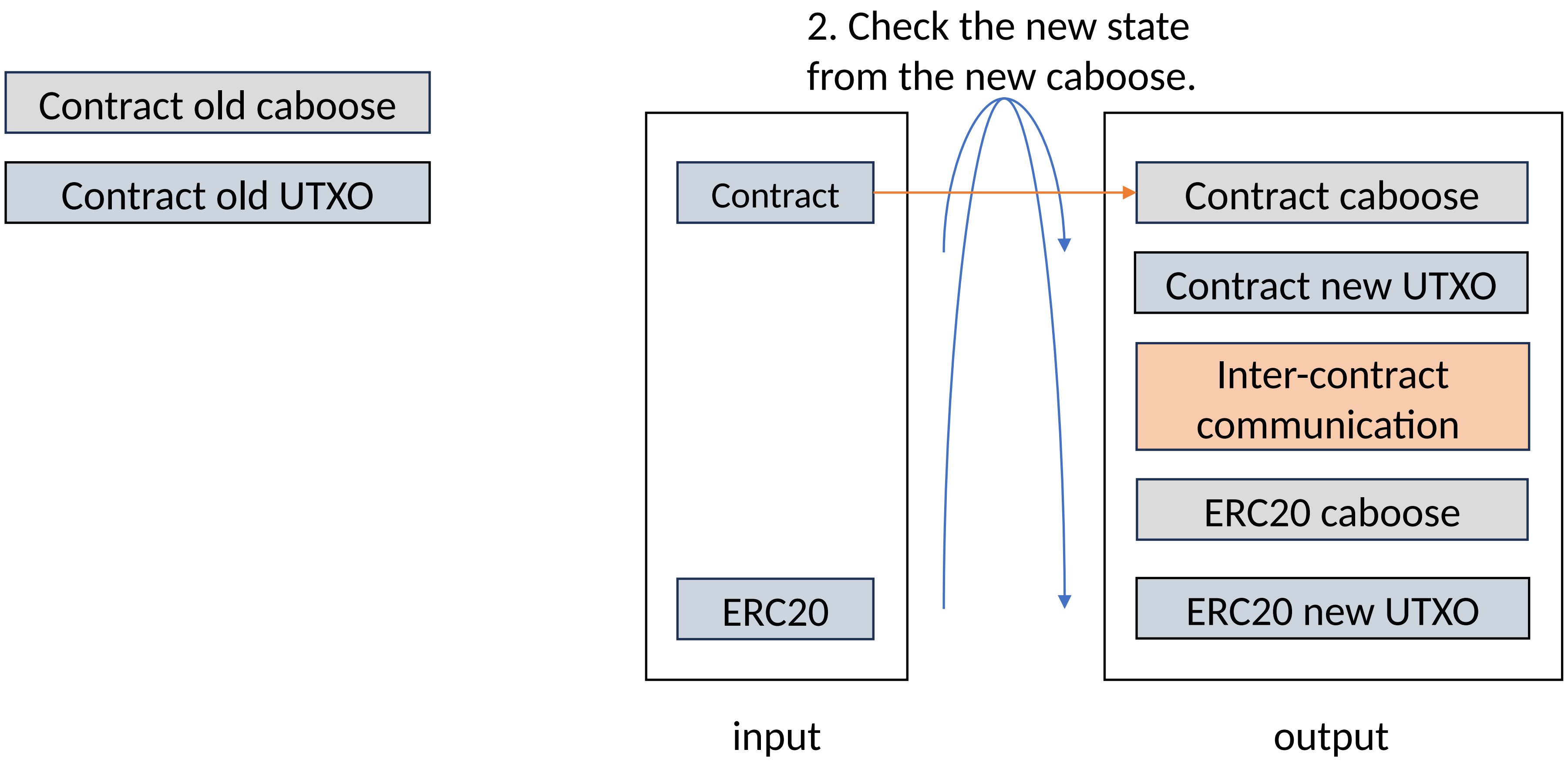
- Example: A contract invoking ERC20 contract to send K tokens to a user



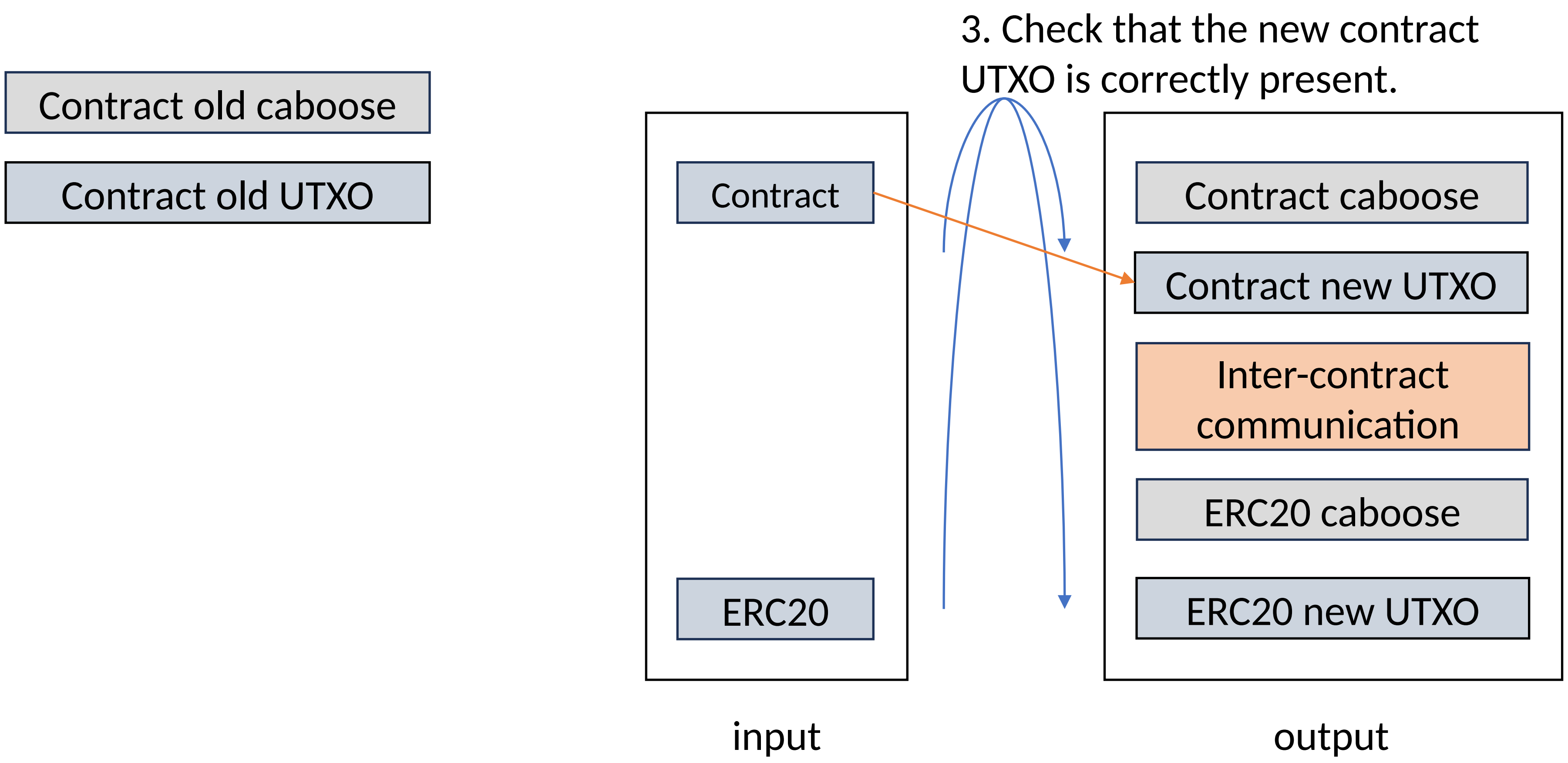
Function calls



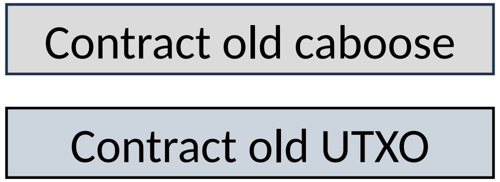
Function calls



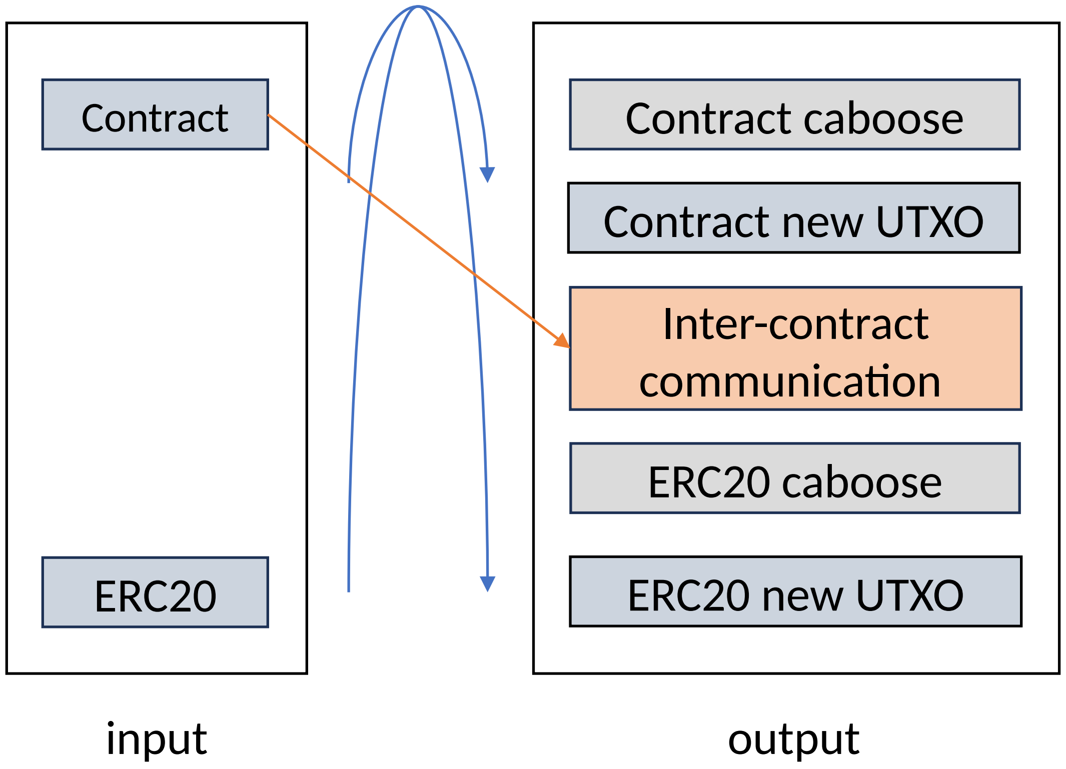
Function calls



Function calls



4. Check that the ERC20 token transfer request is correctly placed in the inter-contract communication.



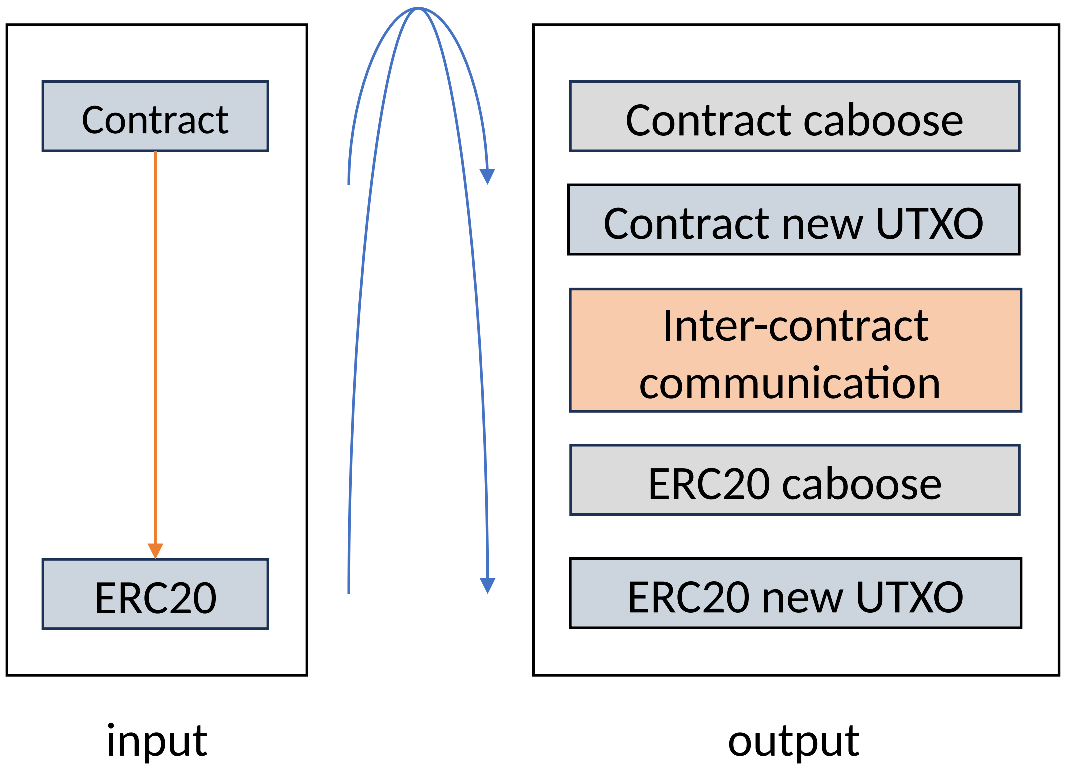
Function calls

Contract old caboose

Contract old UTXO

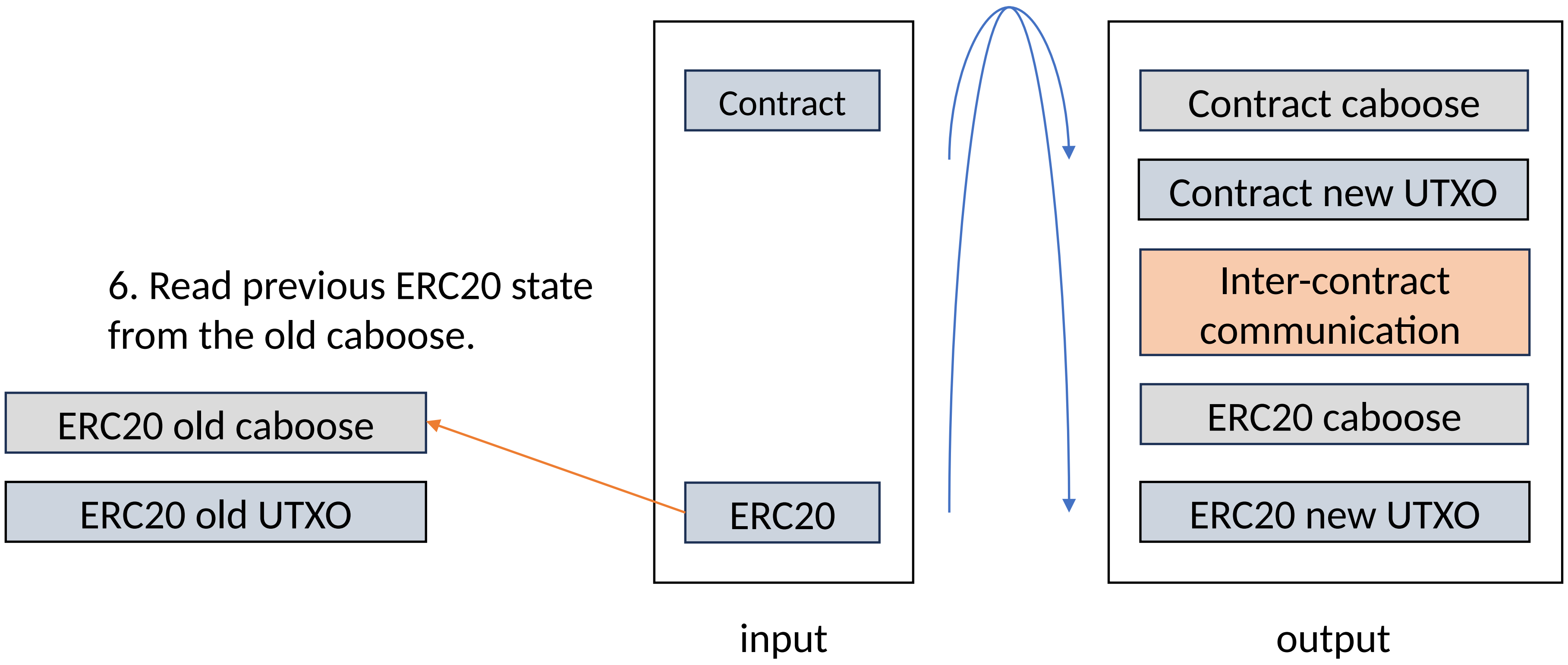
(use a technique called “account emulation” from the CAT20 protocol)

5. Check that the ERC20 contract is present.



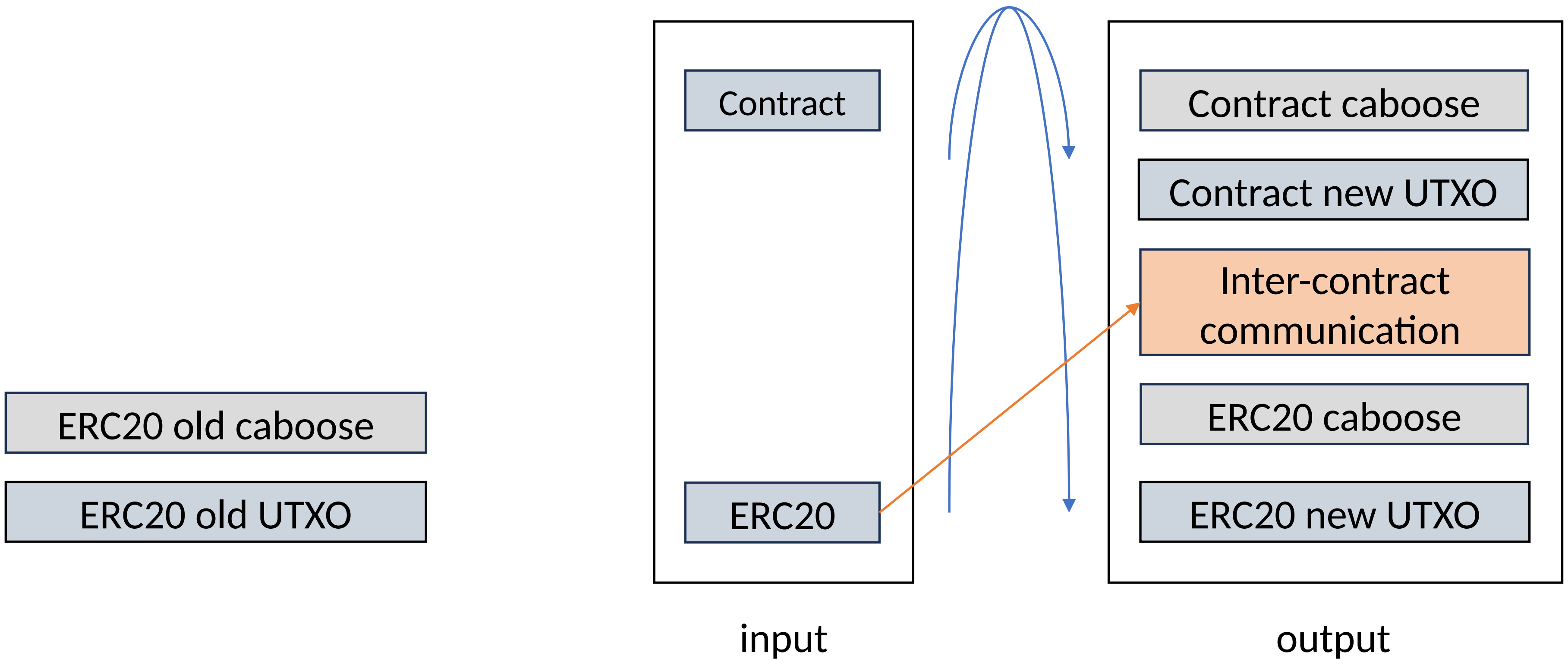
Function calls

6. Read previous ERC20 state from the old caboose.



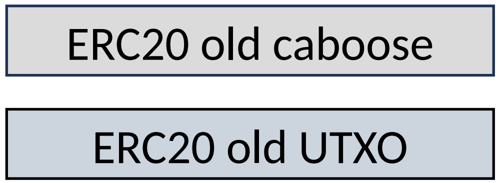
Function calls

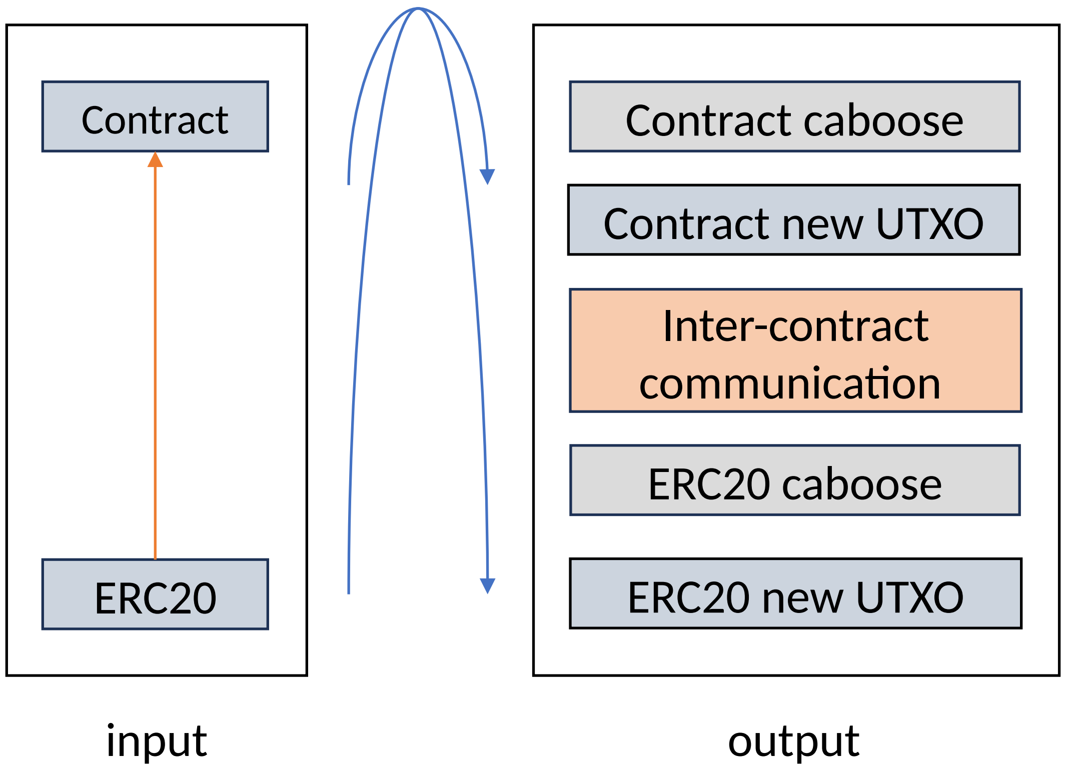
7. Read the inter-contract communication.



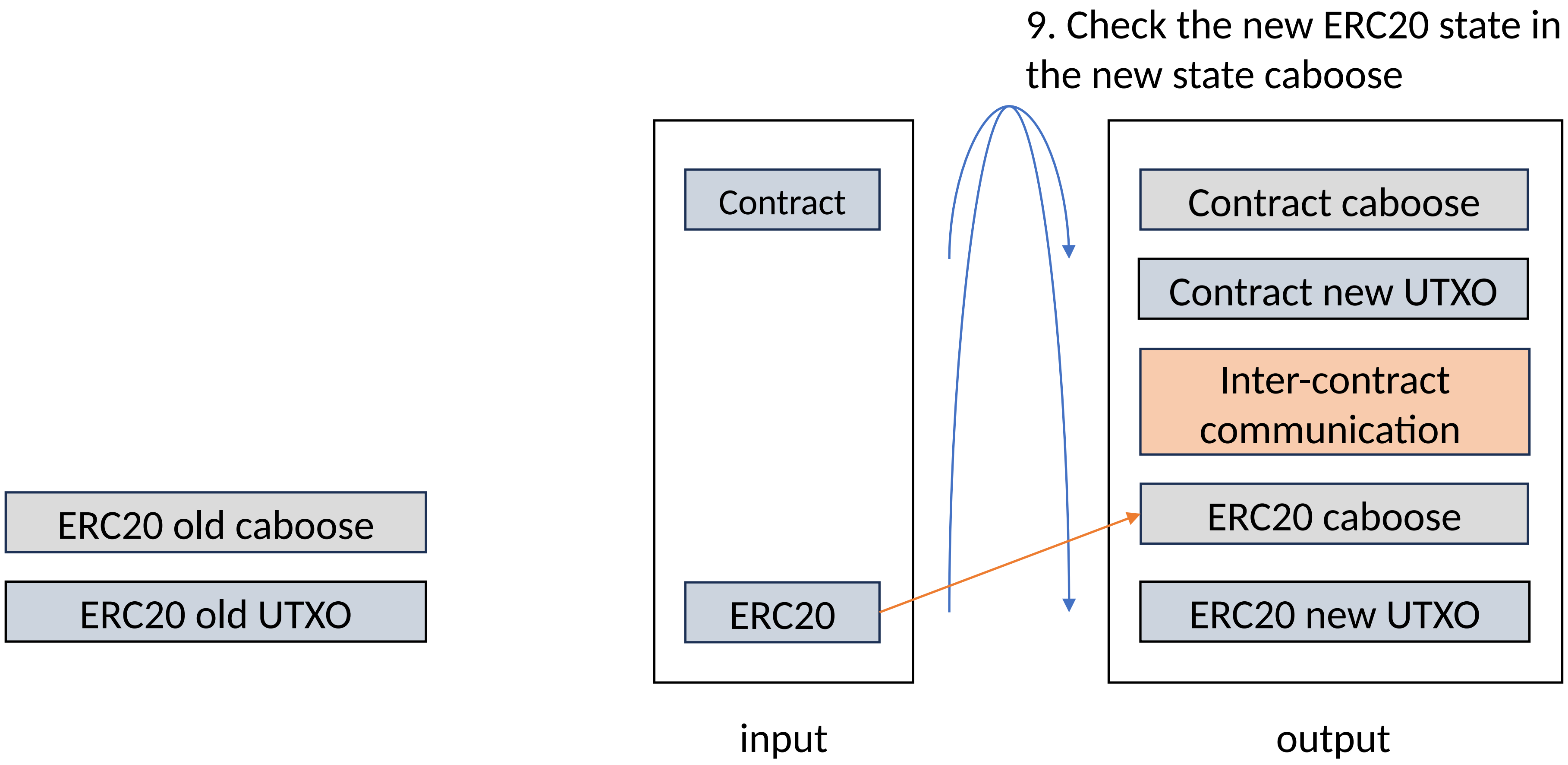
Function calls

8. Check that the contract (who is the sender) is present



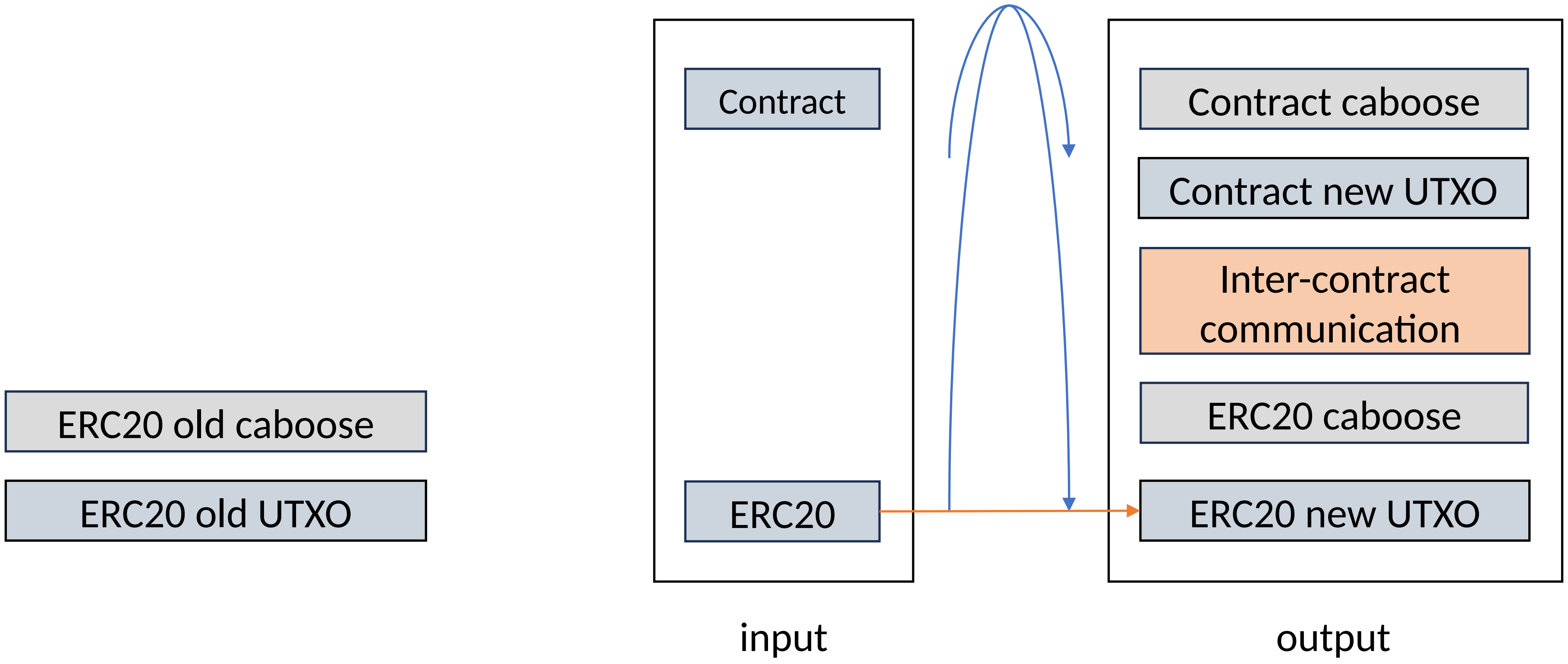


Function calls



Function calls

10. Check that the new ERC20 contract UTXO is correctly present.

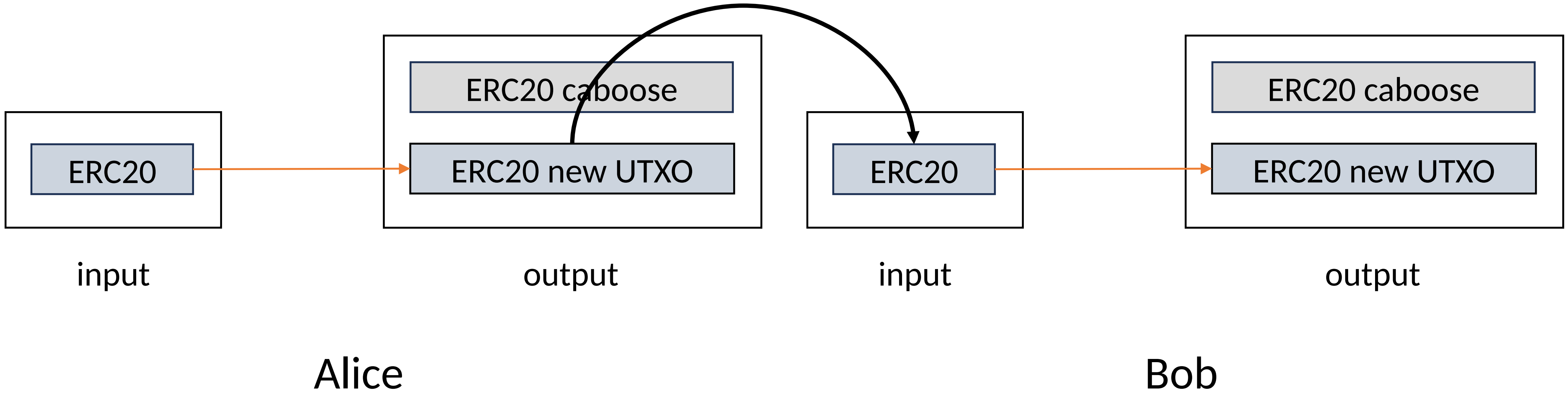


Function calls

- The example is a special case for two contracts to interact with each other.
- Needs standardization for such inter-contract communication that is simple, safe, scalable, and flexible.

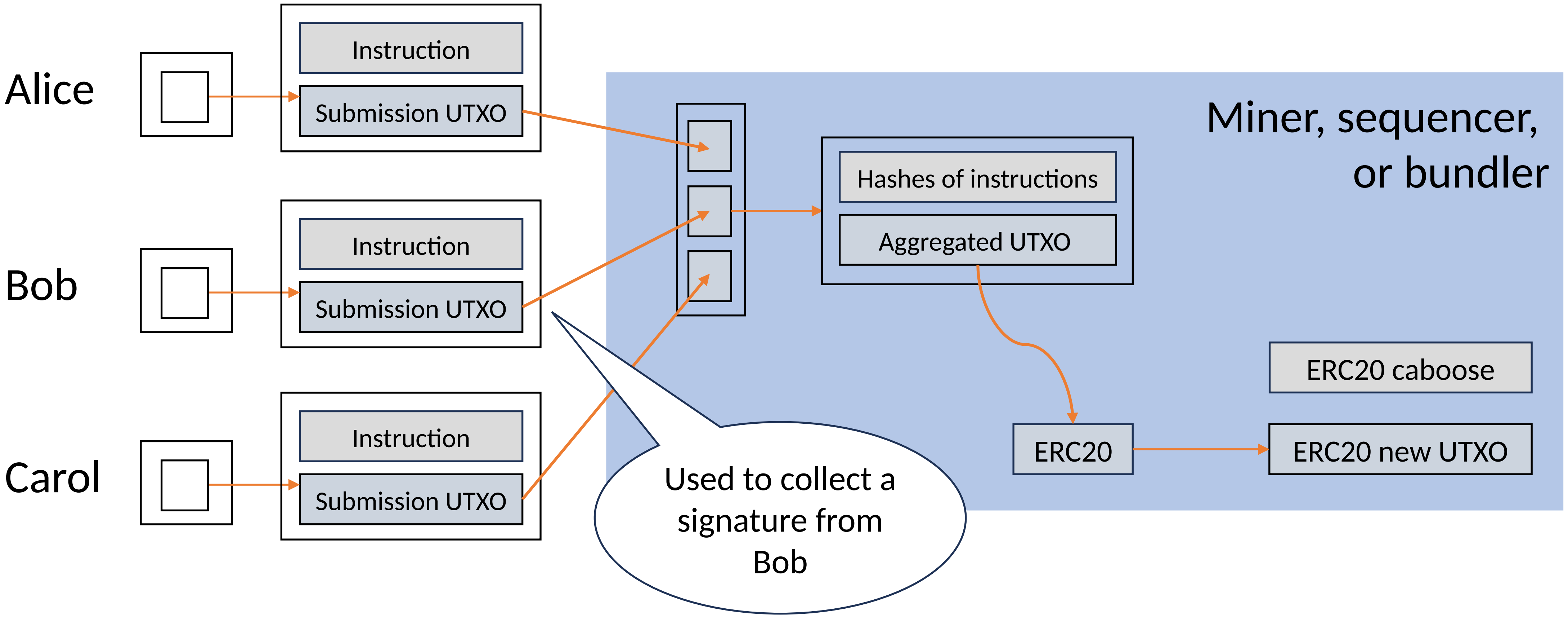
Concurrency

- A fundamental issue with function calls is that users may need to work together to schedule the transaction flow, to allow the same contract being invoked multiple times in the same block.



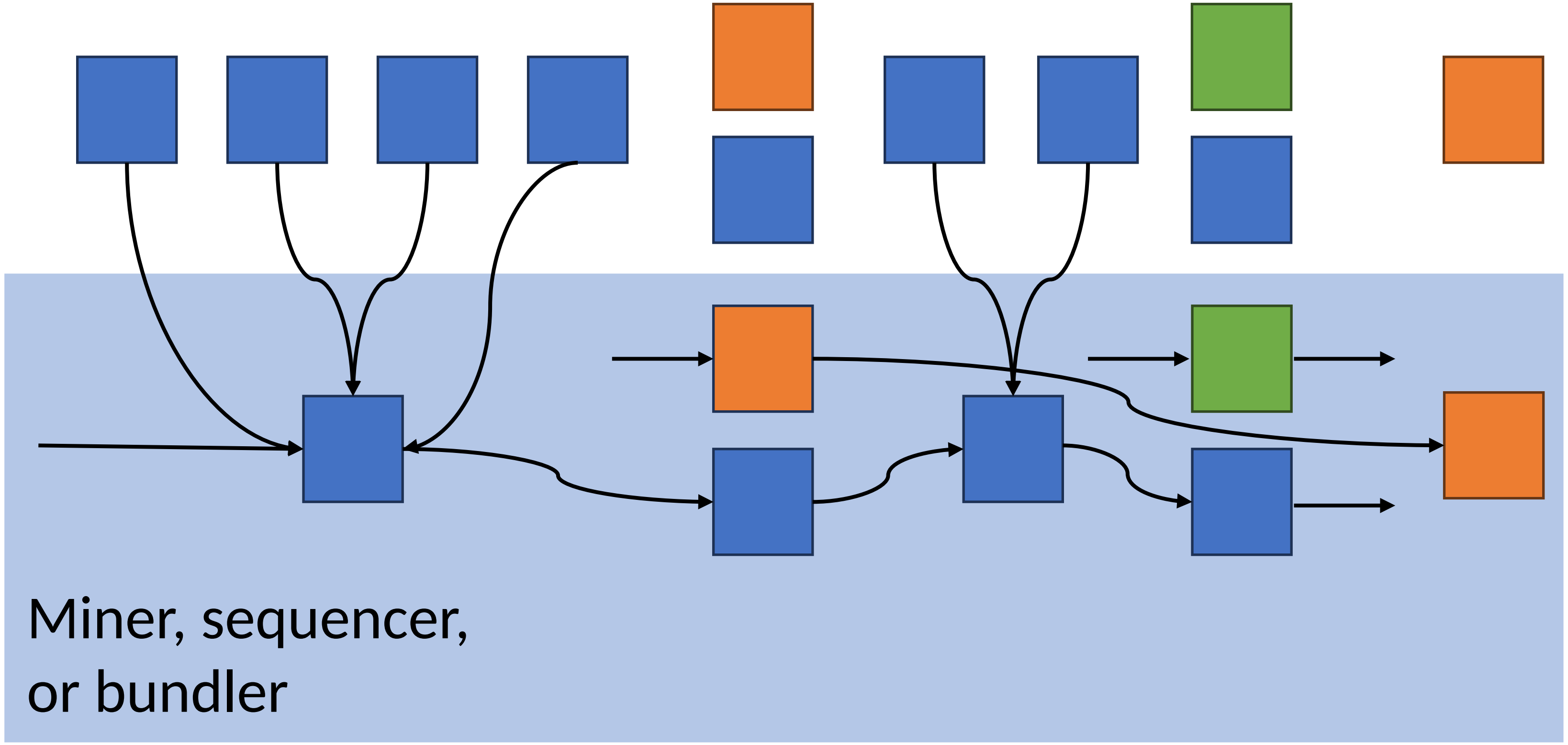
Concurrency

- Miner-helped sequencing (sCrypt uses this idea in CAT-based StarkNet bridge)



Concurrency

- Function calls would make it more complicated

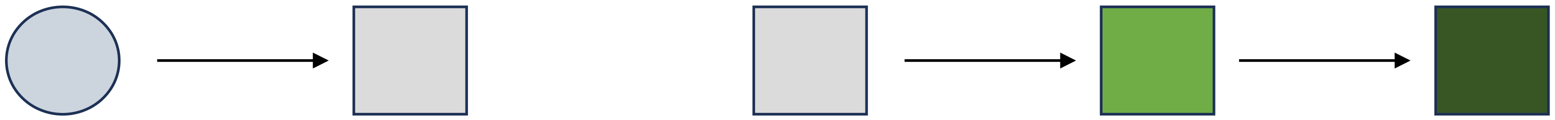


Concurrency

- Miners can decide the order and even change the order to reduce the overhead of aggregation -> MEV from miners or from submitters
- Solution:
  - **Decentralize the MEV:** Currently miners are the only one who decides the order, and miners will change. It is possible to create a role of "bundler" and rotate who is the bundler that is authorized to sort the transactions.
  - **Discourage MEV by PoW:** Complicated but useful in some use cases that are highly MEV sensitive. Could be application-specific.
    - Block n: build a list of submission, sort them by the txid, commit the sorted list
    - Block n+1: use a valid block header, shuffle the sorted list with the block header hash, execute the list exactly following the order in the lsit

Crypto accelerators

- Bitcoin has OP_SHA256, but without OP_CAT it can only do two things:
  - Hash a non-hash element and make it a hash
  - Hash a hash element again

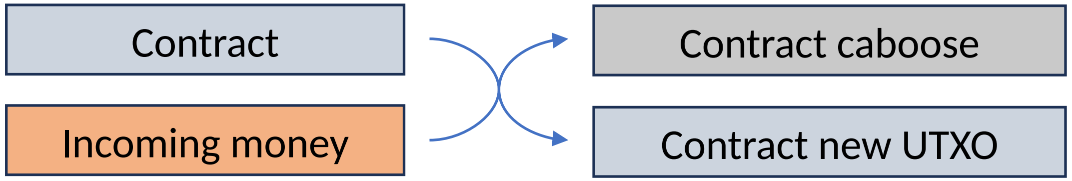


Enough for Lamport and Winternitz signatures, but not for Merkle trees

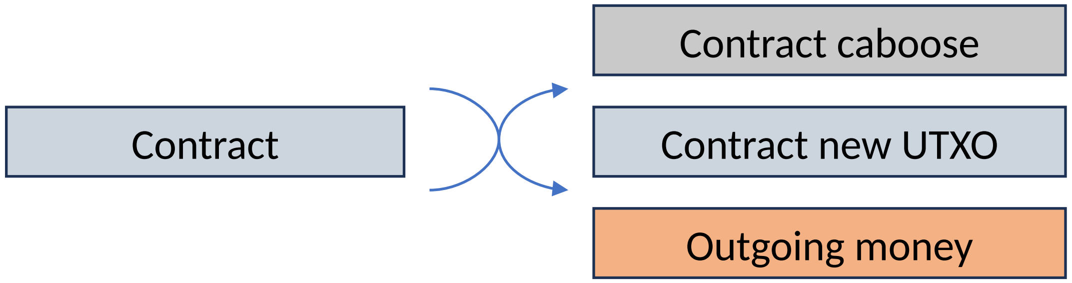
- With OP_CAT, the opcode OP_SHA256 becomes a general-purpose crypto accelerator for hashing up to 520 bytes at once.

Merkle trees

Send and receive money



Receive money



Send money

Status of OP_CAT

Using OP_CAT today

- Liquid Network (TVL: USD $251m)
  - Native token is L-BTC, 1-1 peg with BTC
  - Deviates from Bitcoin by having more opcodes and protocol changes
- Bitcoin SV (TVL: USD $977m)
  - Native token is BSC, a separate token
  - Deviates from Bitcoin by having larger blocks and script support
- Fractal Mainnet (TVL: USD $25m)
  - Native token is FB, a separate token
  - Deviates from Bitcoin **only** in having `OP_CAT`
- Bitcoin Signet (no TVL, testnet)
  - Native token is Signet BTC, a testnet token that is rendered no value
  - Deviates from Bitcoin **only** in having `OP_CTV`, `OP_CAT`, `ANYPREVOUT`

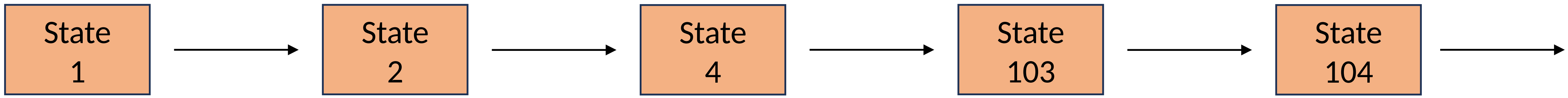
OP_CAT covenant is highly efficient

- Example: counter
- At a fee rate 2 sat/vByte, every transaction costs USD $0.55

Count-update script

Three functions:

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



BitVM benefits from OP_CAT

- BitVM requires a per-instance trusted setup ceremony
  - MultiSig that depends on the programs but does not depend on proofs or data
  - Purpose: do covenants without OP_CAT

- OP_CAT removes this setup
  - Script can directly enforce covenants.
  - No need for Lamport or Winternitz signatures.
  - Low challenging deposit

OP_CAT subsumes OP_CTV

- OP_CTV can be implemented with OP_CAT
  - OP_CAT can pull the information of the entire transaction and compute OP_CTV template hash.
- OP_CTV is weaker than OP_CAT.
- OP_CTV does not enable recursive covenants.

OP_CAT can (somewhat) do ANYPREVOUT

- ANYPREVOUT standard refers to a Schnorr signature over a hash that does not commit “previous outputs” data.

- OP_CAT can compute the message to be signed by the Schnorr but verifying this Schnorr signature cannot be done by OP_CHECKSIG.
- One must emulate Schnorr signature verification (which involves elliptic curve and finite field) in the Bitcoin script.
- Without ANYPREVOUT, signatures for OP_CHECKSIG are not directly reusable.

- Some use cases may have an easier implementation directly via OP_CAT.

L2 and Bitcoin ZK verifier

Reasons for L2

- Scalability, latency, fee
- Sequencing, account abstraction
- Computation model (account vs UTXO)
- Programming language (EVM/WASM vs Bitcoin script)

Even if some of the features could be emulated within Bitcoin script, but Bitcoin has a limited processing capacity that cannot afford the virtualization overhead.

BitVM is an option, but not ideal for a layer-2

- For liveness, needs to trust at least one of the designated operators
  - Otherwise, operators can lock all users' assets
  - No unilateral exit
- Peg-in needs to be a fixed amount during the setup
- Operator needs to front BTC during peg-out, and operator will need to wait for a period to get the money back (for security against forking attacks)

- Many other issues have solutions.

STARK vs SNARK

- Hash-based ZK, commonly known as STARK, can be made Bitcoin-friendly so a single proof verification takes about 6MB of script.
- Elliptic curve-based ZK, commonly known as SNARK, is known to require at least 2GB of script.

- The cost of both can be dramatically lowered down if we do optimistic ZK proof verification (if someone will challenge if the proof is wrong)

Our work: Circle Plonk verifier

- Circle Plonk is a proof system that replaces a few components from the Plonk protocol:
  - Use Circle M31 for field and FRI for polynomial commitment
  - Further simplify the circuit representation, reduce the number of columns
- Support R1CS. Developers can use circuits from existing DSLs, such as Arkworks-rs or Circom.

- Verifying a small Circle Plonk proof takes about 3.6MB of script (which would cost $1162 to verify one proof with a fee rate of 2sat/vByte). We can optimize it to $660 by reusing Bitcoin PoW. This number can go up for larger proofs.
- Optimistic ZK verifier would be very cheap.

Writing the Bitcoin script for the verifier

- We currently use an embedded DSL in Rust (based on the design of Arkworks-rs) to write Bitcoin script.

Field arithmetic

```
pub fn ibutterfly(
    table: &TableVar,
    v0: &QM31Var,
    v1: &QM31Var,
    itwid: &M31Var,
) -> (QM31Var, QM31Var) {
    let new_v0 = v0 + v1;

    let diff = v0 - v1;
    let new_v1 = &diff * (table, itwid);

    (new_v0, new_v1)
}
```

Hashing

```
channel_var = &channel_var + &interaction_commitment_var;
channel_var = &channel_var + &constant_commitment_var;
```

Memory

```
let alpha4: QM31Var = ldm.read("line_batch_random_coeff_4")?;

let alpha4composition_l = &alpha4 * (&table, &sum_num_composition_l);
let alpha4composition_r = &alpha4 * (&table, &sum_num_composition_r);

ldm.write(
    format!("alpha4composition_{}_l", query_idx),
    &alpha4composition_l,
)?;
```

Writing the Bitcoin script for the verifier

- Recently we attempted using it to implement Blake3.

Blake3 rotate right shift by 16

```
pub fn rotate_right_shift_16(self) -> Self {
    let limbs : [U4Var; 8] = self.limbs;
    let new_limbs : [U4Var; 8] = [
        limbs[4].clone(),
        limbs[5].clone(),
        limbs[6].clone(),
        limbs[7].clone(),
        limbs[0].clone(),
        limbs[1].clone(),
        limbs[2].clone(),
        limbs[3].clone(),
    ];
    Self { limbs: new_limbs }
}
```

Blake3 g function

```
pub fn g(
    table: &LookupTableVar,
    a_ref: &mut U32Var,
    b_ref: &mut U32Var,
    c_ref: &mut U32Var,
    d_ref: &mut U32Var,
    m_0: &U32Var,
    m_1: &U32Var,
) {
    let mut a : U32Var = a_ref.clone();
    let mut b : U32Var = b_ref.clone();
    let mut c : U32Var = c_ref.clone();
    let mut d : U32Var = d_ref.clone();

    a = &a + (table, &b, m_0);
    d = (&d ^ (table, &a)).rotate_right_shift_16();
    c = &c + (table, &d);
    b = (&b ^ (table, &c)).rotate_right_shift_12();
    a = &a + (table, &b, m_1);
    d = (&d ^ (table, &a)).rotate_right_shift_8();
    c = &c + (table, &d);
    b = (&b ^ (table, &c)).rotate_right_shift_7(table);
```

Next steps

- Reusing Bitcoin PoW
- Decorrelated transaction flow
- Fraud proof version

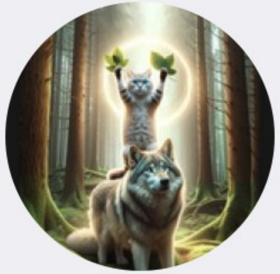
- Recursive verifier: verifying Circle Plonk in Circle Plonk
- Exploring STARK verifier in BitVM

Thank you

Bitcoin Wildlife Sanctuary

A cat, a wolf (stark), two leaves (Merkle tree), and a circle (circle curve)

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



Bitcoin Wildlife Conservatorium (Stark + CAT)

100 members, 2 online