Fairgate

ESSPI

ECDSA/Schnorr Signed Program Input for BitVMX

Sergio Demian Lerner | CTO, Fairgate Labs

BitVMX: A CPU for Universal Computation on Bitcoin

BitVMX is a cutting-edge framework designed to optimistically execute arbitrary programs on Bitcoin, leveraging the N-party disputable computation paradigm introduced by BitVM(*).

With its foundation in secure, extensible, and open-source principles, **BitVMX paves the way for running any CPU on Bitcoin**.

***** Created by Robin Linus in 2023.

Fairgate

Problem

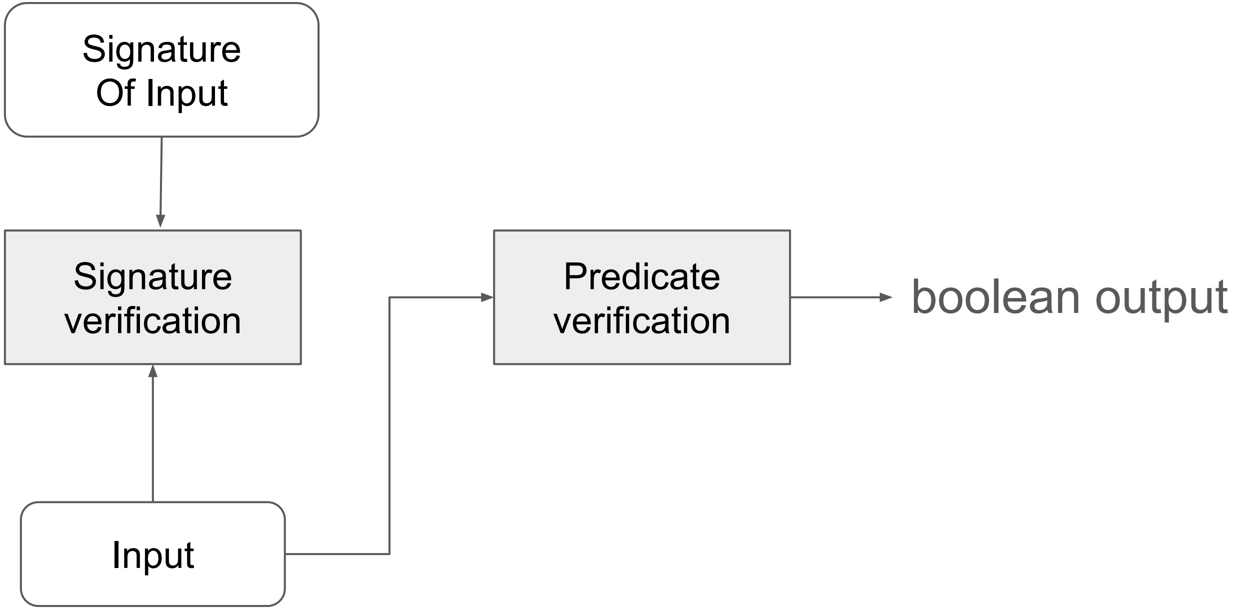
BitVM and BitVMX inputs must be signed with the Winternitz scheme. The existing implementation expands each signed byte to 200 vbytes.

This makes BitVM protocols very expensive when verifying other computation integrity proofs such as STARKs or Nova.

Solution

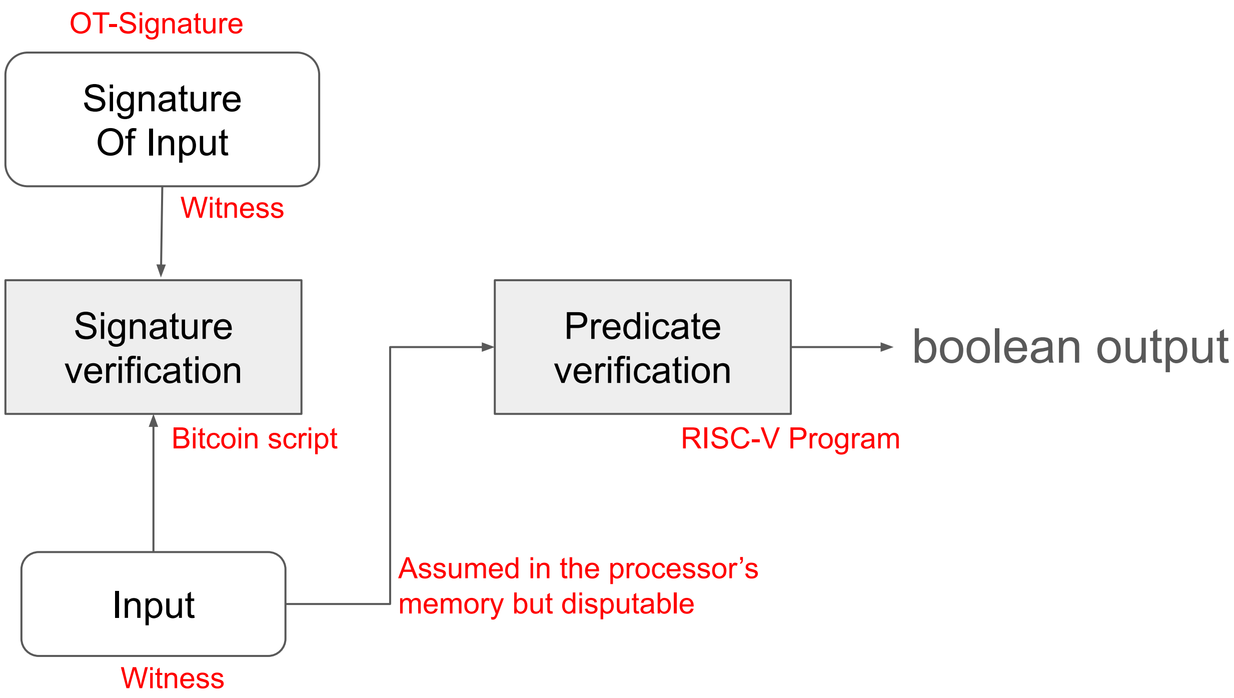
Use standard Bitcoin transaction to store the data, as transactions are already signed by Schnorr or ECDSA signatures.

Abstract Bitcoin Predicate Verification Machine



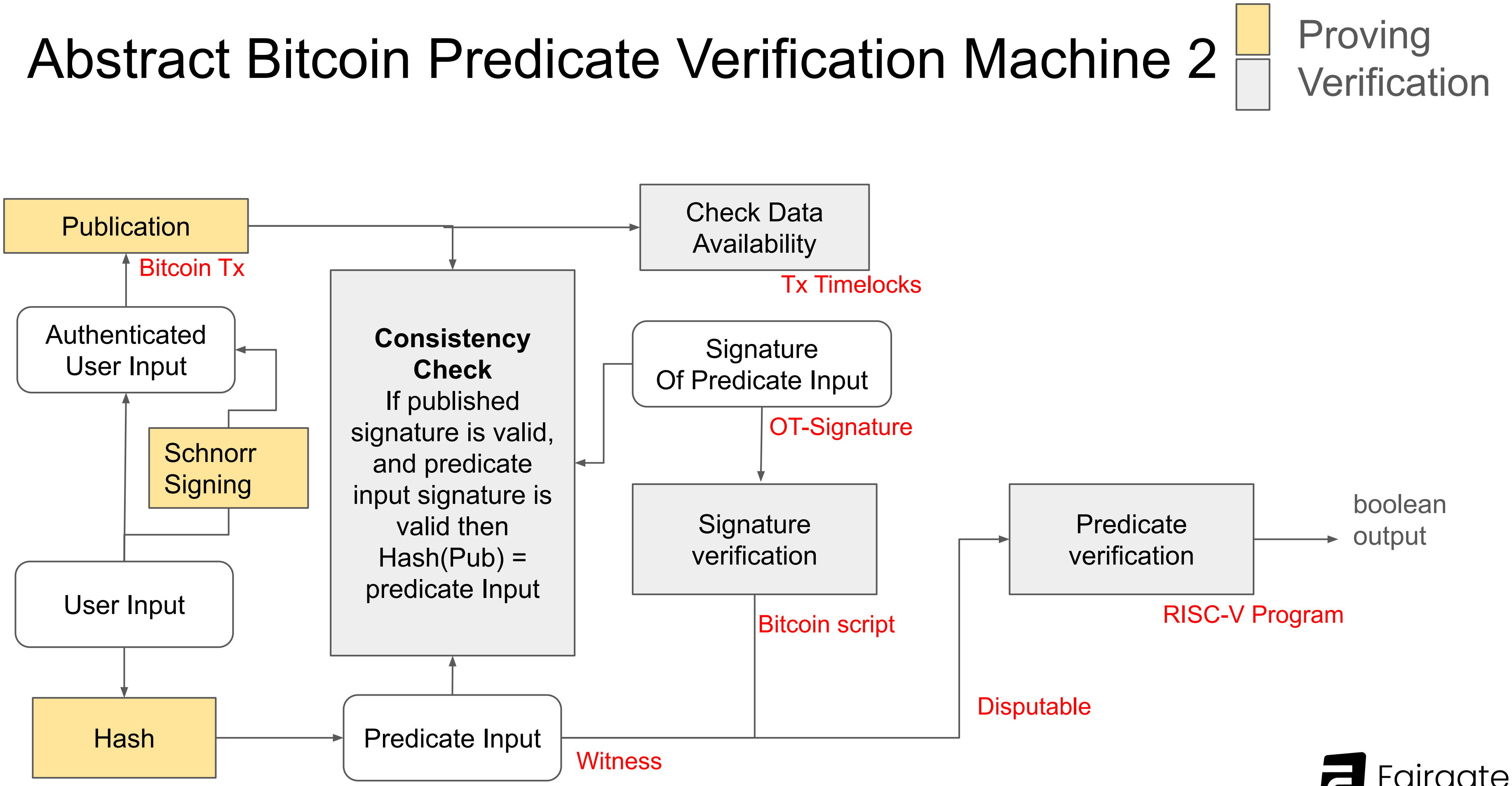
Fairgate

Abstract Bitcoin Predicate Verification Machine



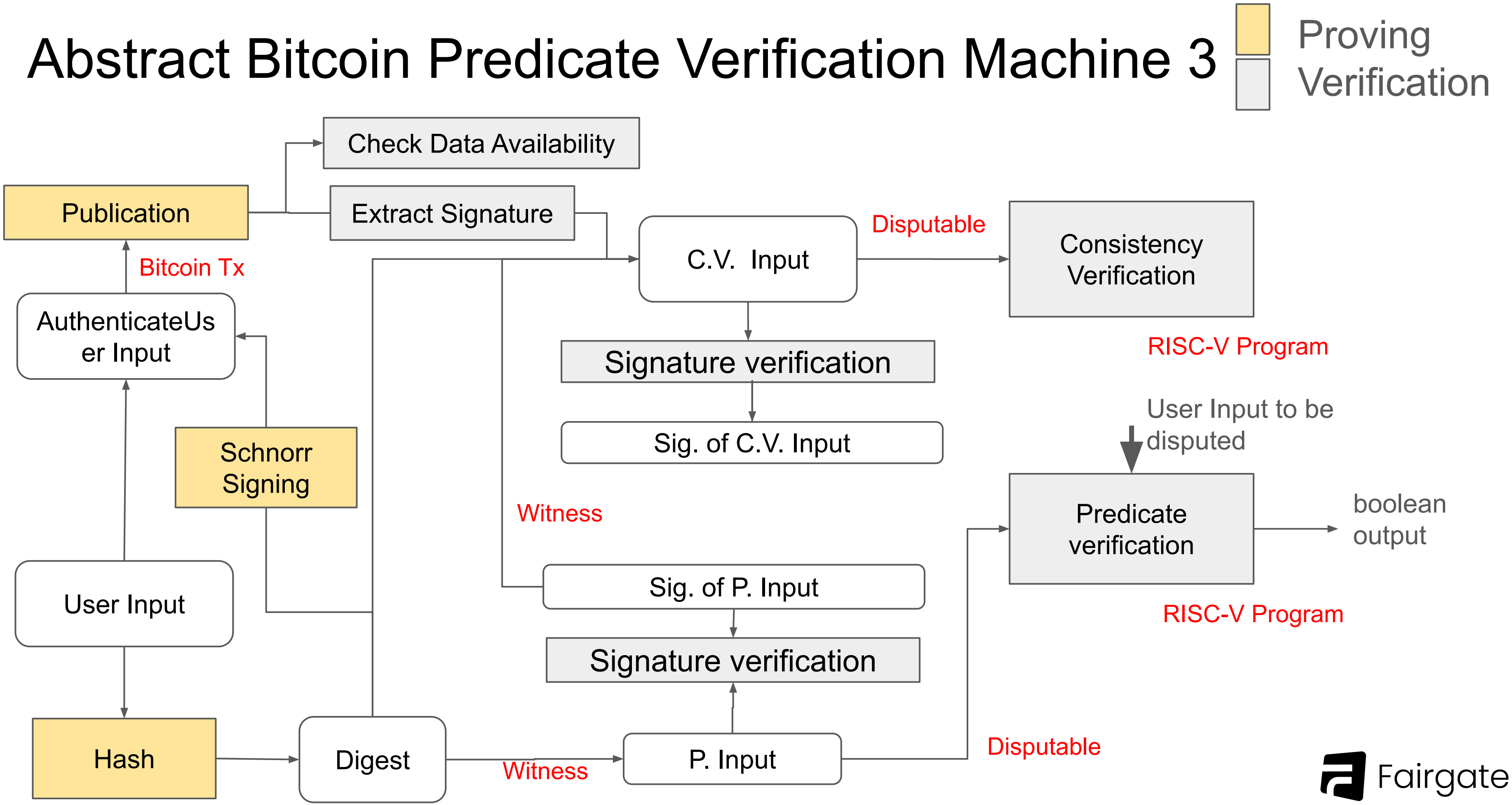
Fairgate

Abstract Bitcoin Predicate Verification Machine 2



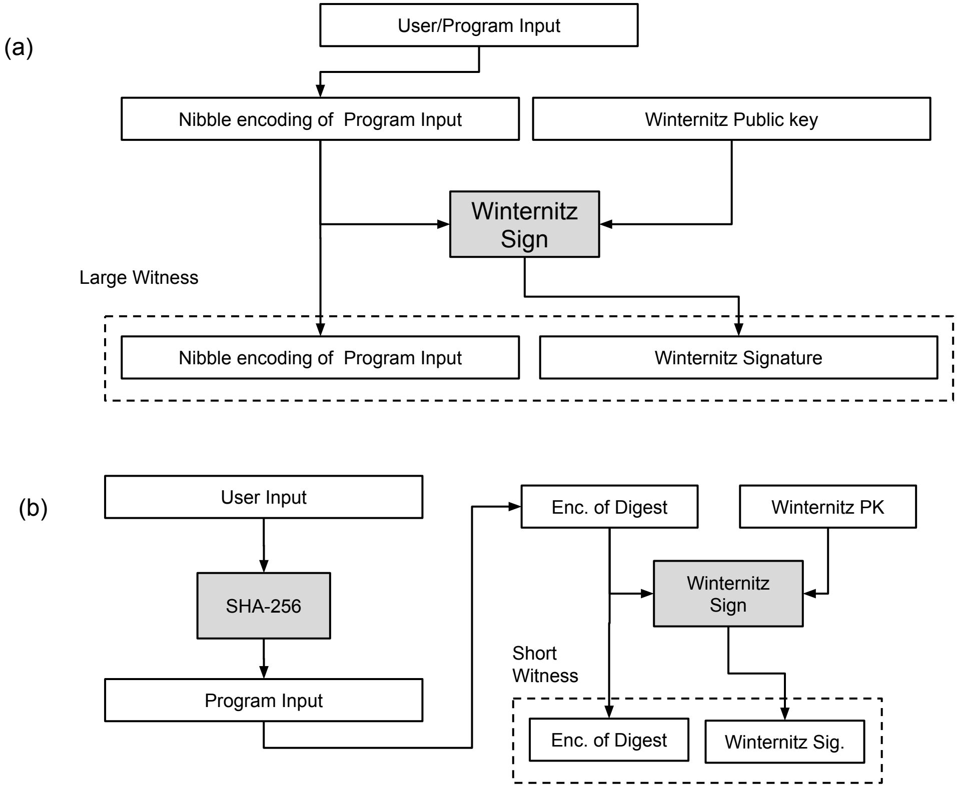
Fairgate

Abstract Bitcoin Predicate Verification Machine 3



Fairgate

OT-Signing the Input vs a hash of the Input



Fairgate

The Basic DAG

While conceptually simple, is very tricky to get all the details right!

Terminology Used for Transactions

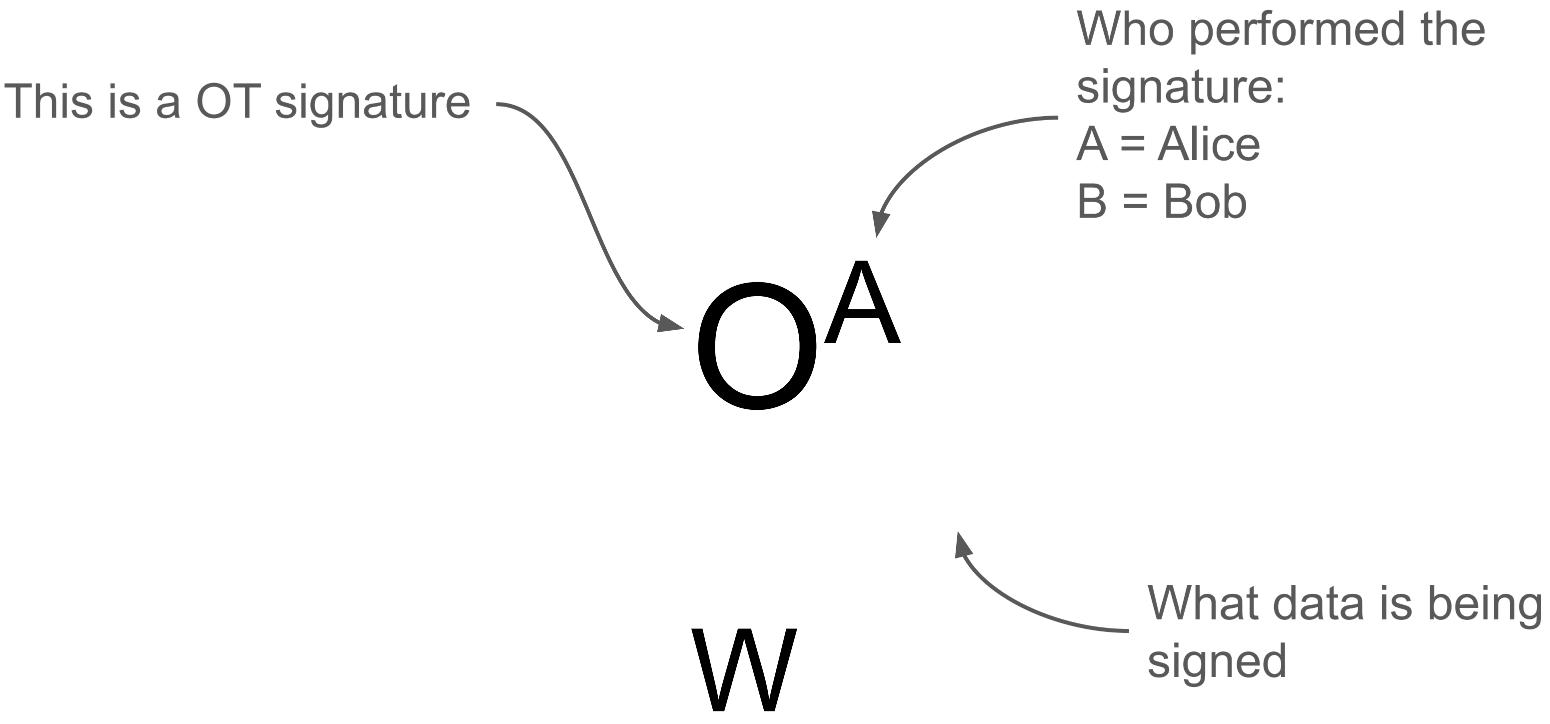
$$P^{A}_{C}$$

Type of transaction:
P = Penalization
K = Kick-off
D = Data
C = Commitment
R = Reveal

Who published the transaction:
A = Alice
B = Bob

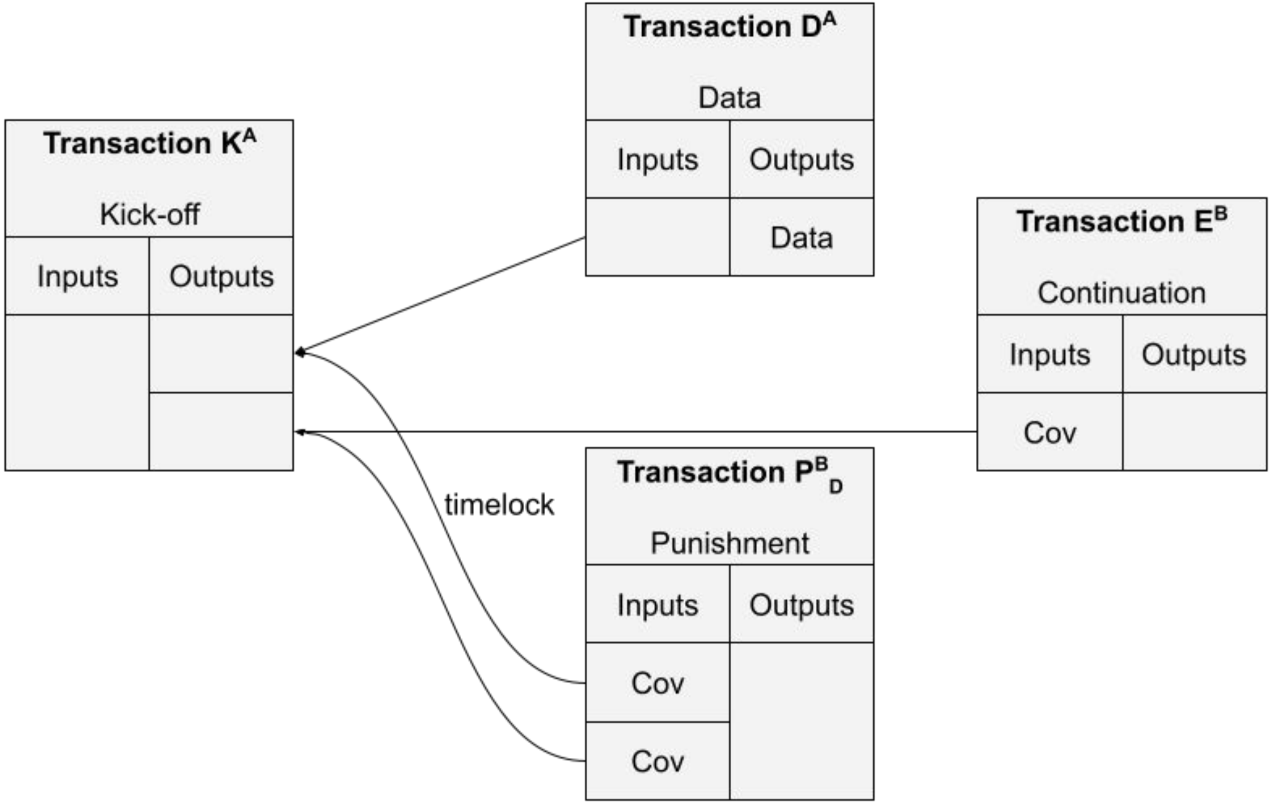
What other transaction this is responding to:
D = Data
C = Commitment
R = Reveal

Terminology Used for One-Time Signatures



Fairgate

Simple Scheme to Force Publication of Data in Bitcoin



Fairgate

Storing Signed Data in a Bitcoin Transaction

- **OP_RETURN**. Data stored in an output containing an OP_RETURN opcode in its scriptPub. Data in output
- **Enveloping**. Data pushed into the stack in a ScriptPub and surrounded by a skipping conditional (OP_PUSH 0 / OP_IF / OP_ENDIF).
- **Annex**. Data in Segwit annex.
- **P2WSH Address**. Data stored in multiple standard outputs as (un-owned) addresses. Data in output
- **ScriptPub with P2PK**. Data can be stored directly in P2PK outputs as 64-byte public keys. Data in output
- **ScriptPub** with bare multisigs. Data encoded in up to 3 public keys. Data in output

Storing Signed Data in a Bitcoin Transaction

PR #32406: uncap datacarrier by default

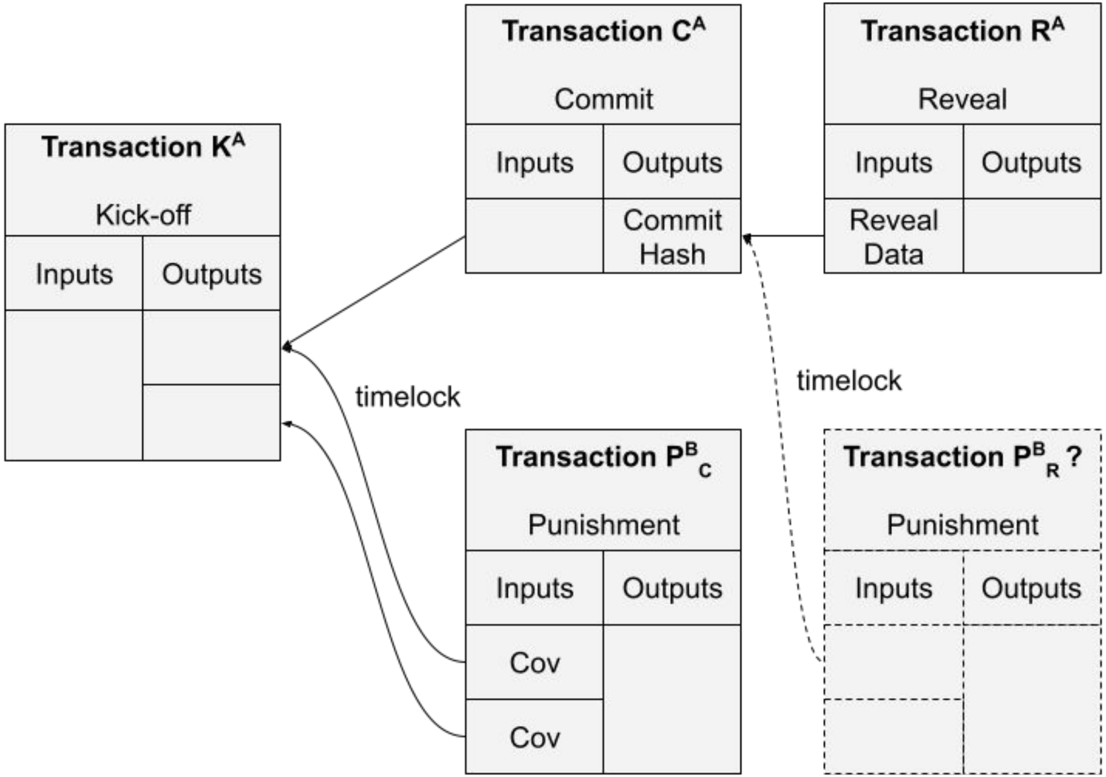
- **OP_RETURN**. Data stored in an output containing an OP_RETURN opcode in its scriptPub.
- **Enveloping**. Data pushed into the stack in a ScriptPub and surrounded by a skipping conditional (OP_PUSH 0 / OP_IF / OP_ENDIF).

Standard and 4x lower cost

- **Annex**. Data in Segwit annex.
- **P2WSH Address**. Data stored in multiple standard outputs as (un-owned) addresses.
- **ScriptPub with P2PK**. Data can be stored directly in P2PK outputs as 64-byte public keys.
- **ScriptPub** with bare multisigs. Data encoded in up to 3 public keys.

Fairgate

Enveloping with Timeouts - An impossible feat?



Fairgate

Proving Data Availability to BitVMX

- Inclusion-Proof DA
- Timelock-based DA

Inclusion-Proof DA

- A fist BitVMX instance proves input data availability by verifying an **SPV proof.**
  - SNARK signed by Winternitz OTS consumes approximately 60K vbytes.

- The second BitVMX instance receives as input a hash of the data proved to be available in the first instance.
  - 32 bytes with the Winternitz OTS consumes 6.4K vbytes.

Timelock-based DA

- P2SH-based (high cost or non-standard, lower complexity)
- Enveloping-based (standard and low cost, higher complexity)

Problems Still Unsolved (up to this point)

- How can the RISC-V can load the user input into memory, compare with the input hash and how can the challenger dispute it

- How to obtain a hash of the User Input, if Bitcoin doesn’t sign the OP_RETURN data, but the whole transaction.

Fairgate

Building the real Transaction DAG

* Created by Robin Linus in 2023.

Fairgate

Definitions

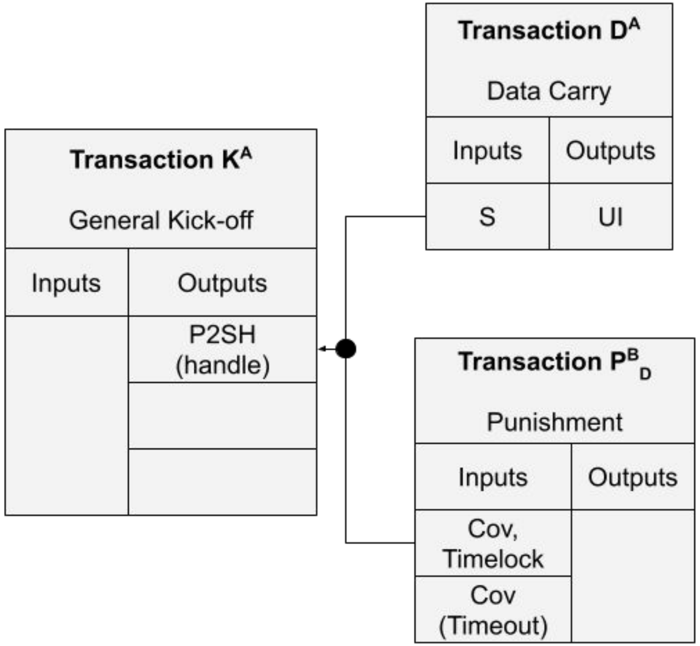
- The **User Input** (UI) as the input the user program will need to consume to decide the outcome of the BitVMX protocol (accept or reject the spending).

- The **Program Input** (PI) will be a message that can be accessed by the BitVMX CPU and contains the UI, but may also contain additional padding, header or footer that should be skipped by the user program.

Definitions

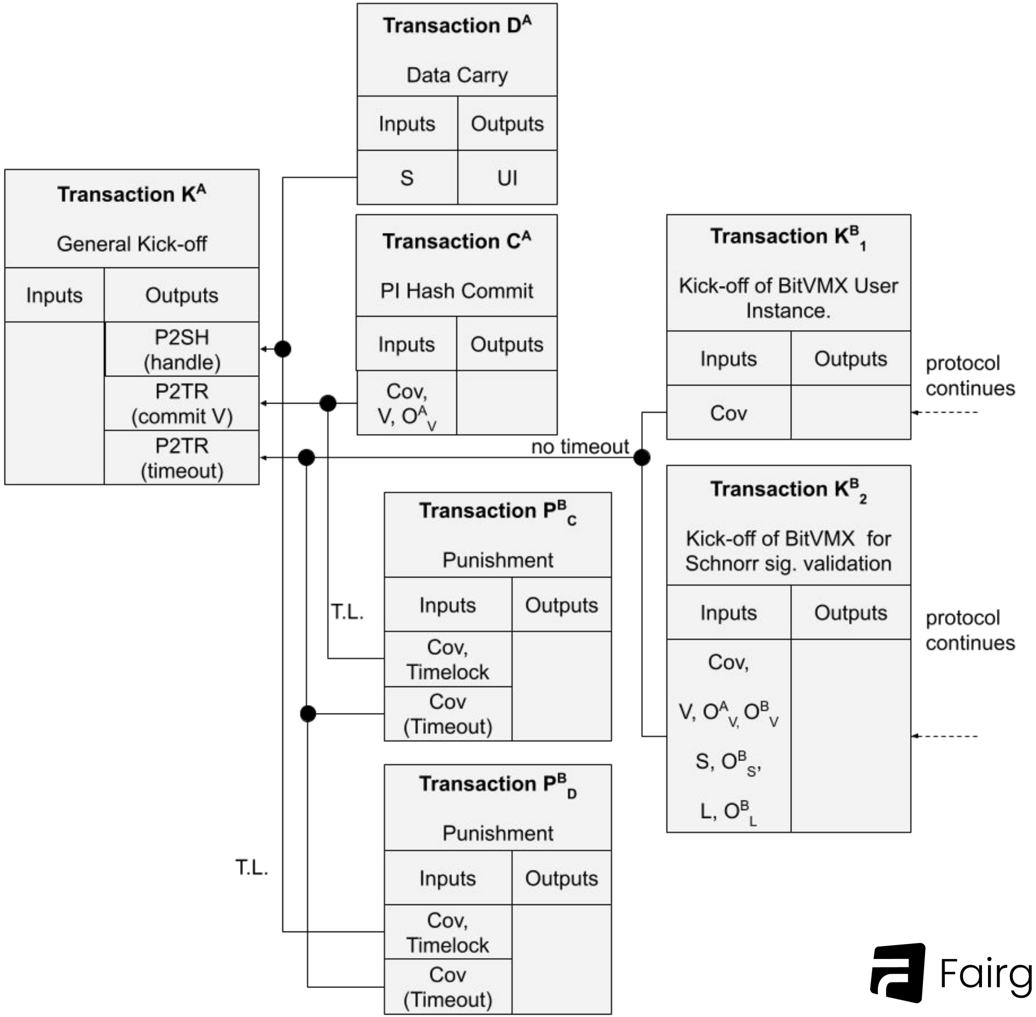
A first kick-off transaction K^A contains a predefined P2SH output called "handle" that contains two spending paths (using `OP_IF/OP_ELSE/OP_ENDIF`).

- The first is consumed by a transaction D^A which provides the User Input (UI Data).
- The second output of K is consumed by a penalization transaction P^B_D, has a relative timelock and requires an emulated covenant



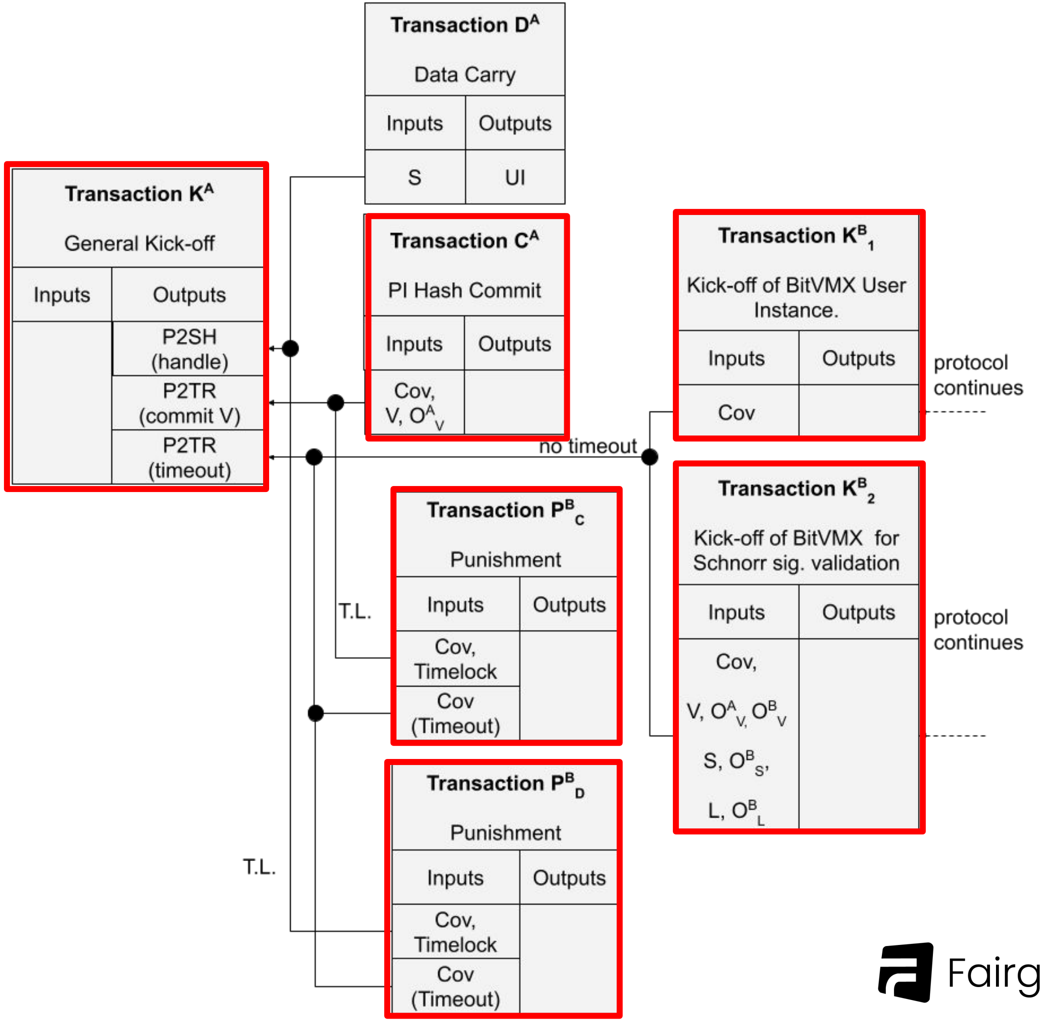
Fairgate

Full DAG



Fairgate

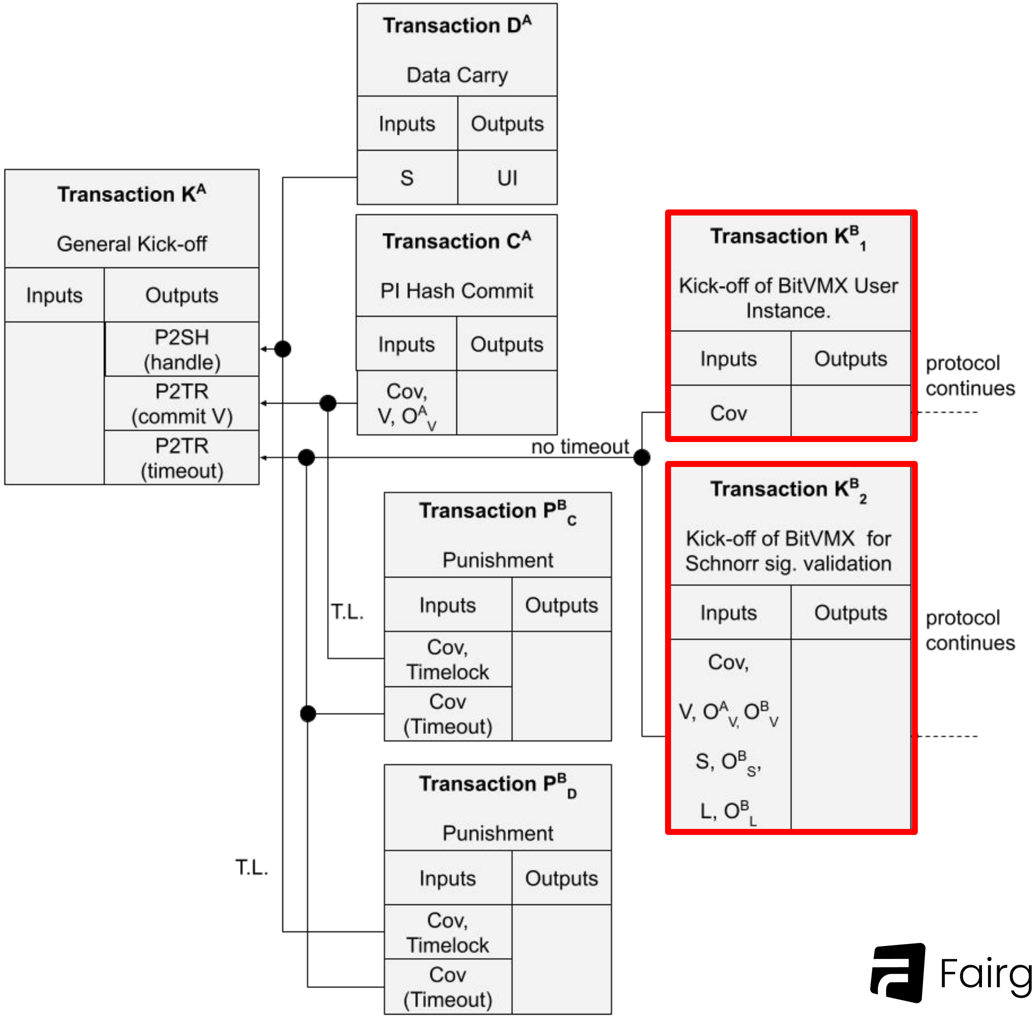
The transactions K^A, P^B_C, P^B_D, K^B_1 and K^B_2 are pre-signed by both participants **emulating covenants**.



Fairgate

Two instances of BitVMX (primary and secondary).

Secondary instance is used by Bob to prove Alice frauds in building her transactions. This is the **Consistency Check.**

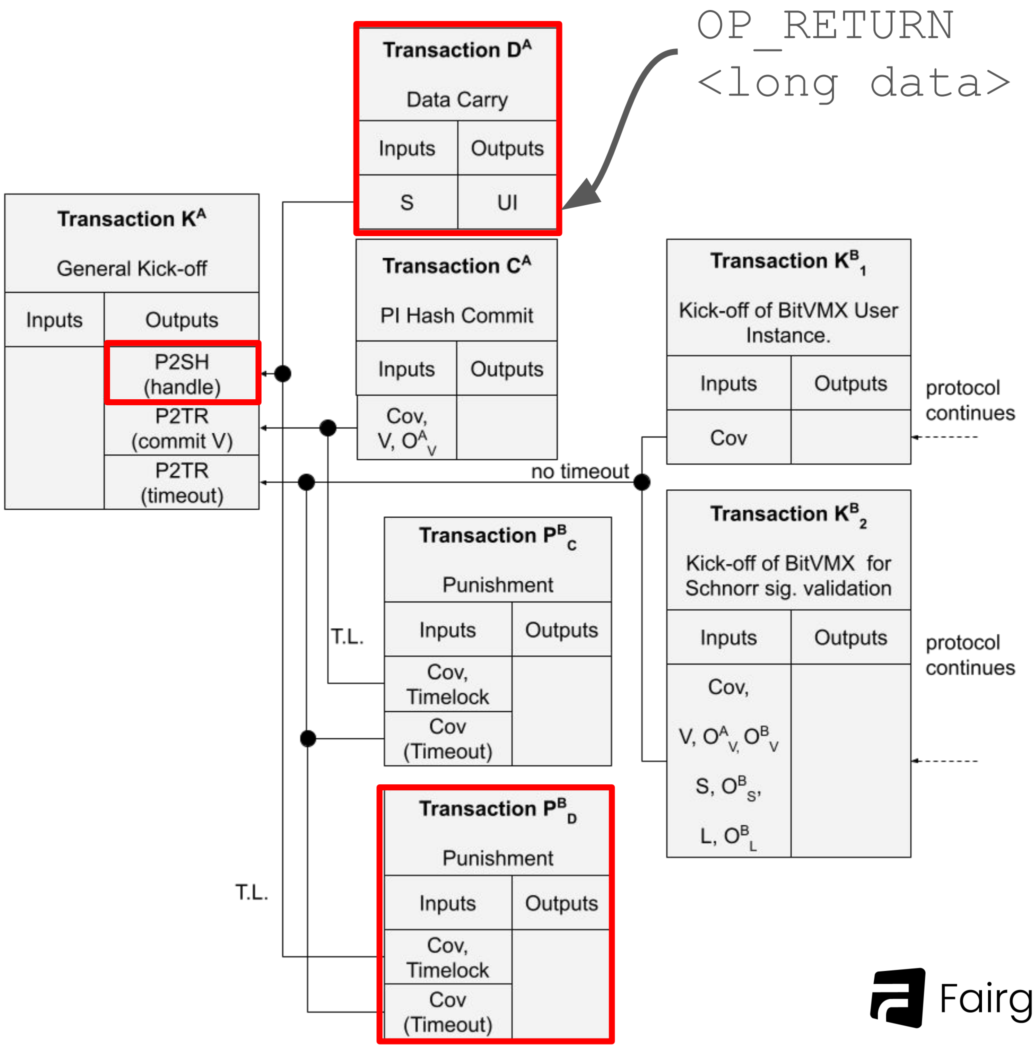


Fairgate

A first kick-off transaction K^A contains a predefined P2SH output called “handle” that contains two spending paths (using `OP_IF/OP_ELSE/OP_ENDIF`).

The first path: transaction D^A which provides the User Input (UI Data).

The second path: a penalization transaction P^B_D, with a relative timelock.

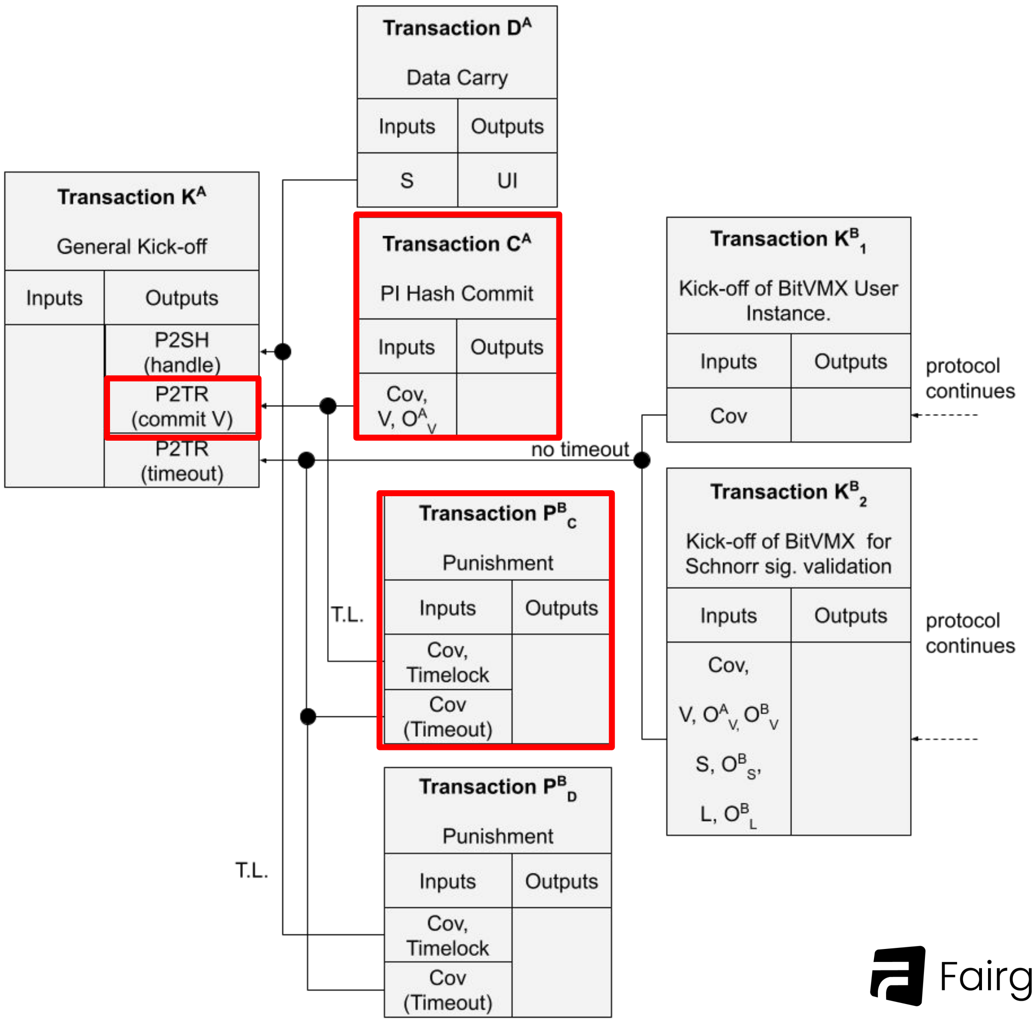


Fairgate

The commit V output is used by Alice to commit to the value V (a hash of the PI).
It can be spent by one of two transactions

The first path: used by a transaction C^A to publish V and O^A_V.

The second path: used by a penalization transaction P^B_C, with a relative timelock.

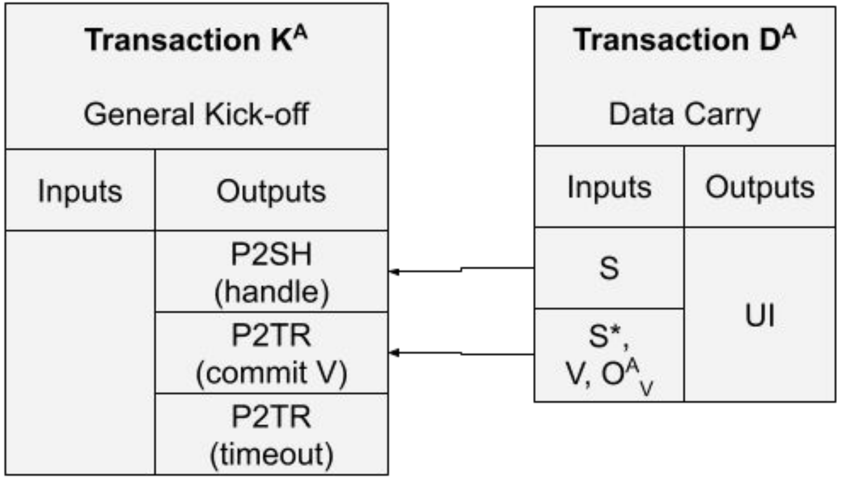


Fairgate

Variant D^A could spend both the handle and commit V outputs.

Downsides:

- PI becomes longer (but worst case is always 1 MB)



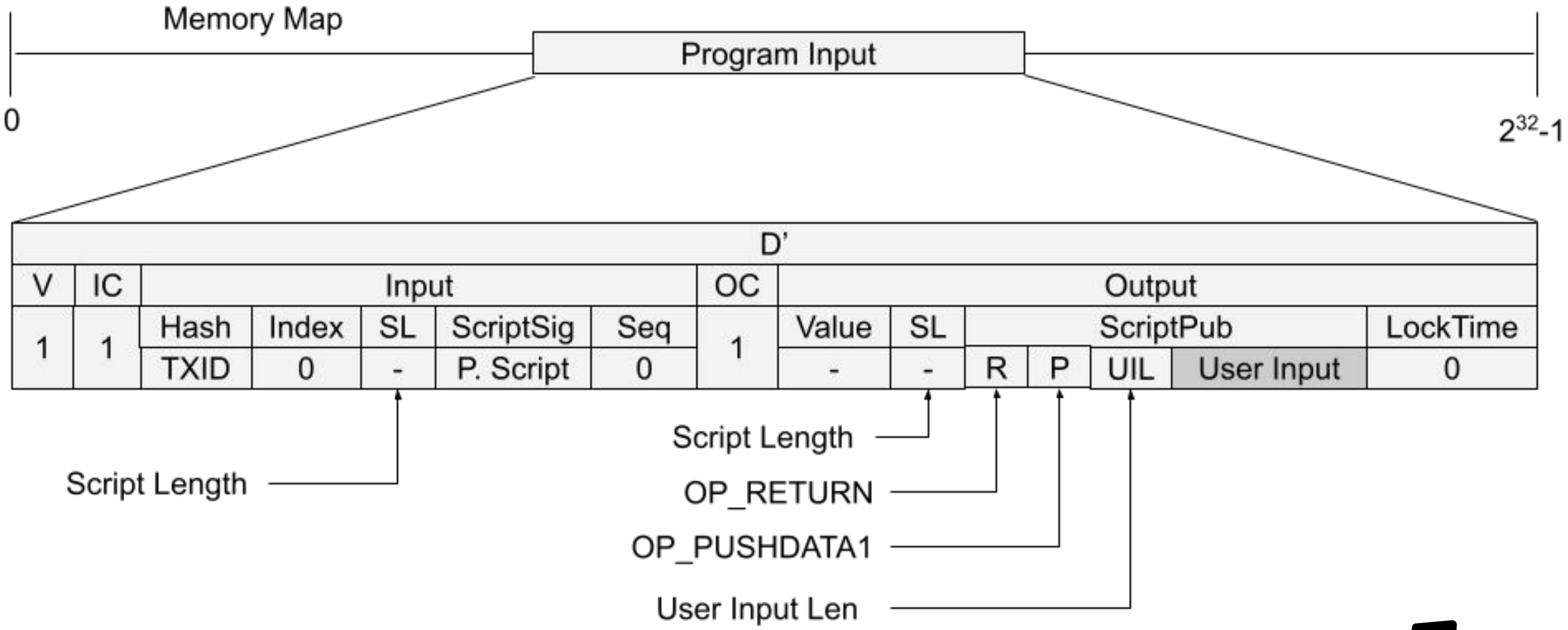
Fairgate

Definitions

- The **User Input** (UI) as the input the user program will need to consume to decide the outcome of the BitVMX protocol (accept or reject the spending).

- The **Program Input** (PI) will be a message that can be accessed by the BitVMX CPU and contains the UI, but may also contain additional padding, header or footer that should be skipped by the user program.

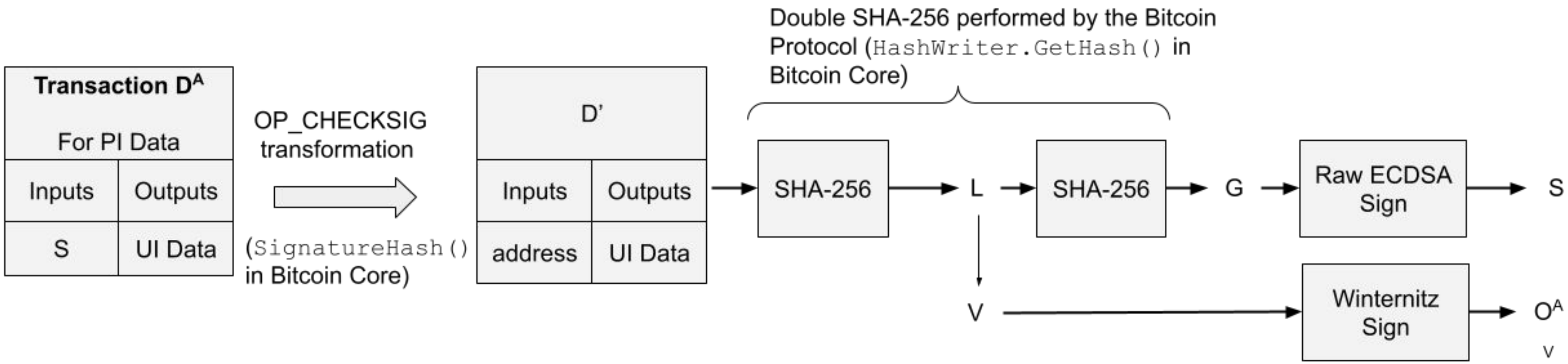
32-Bit BitVMX CPU - Memory-Mapped Program Input



- Program must parse D' and skip inputs.

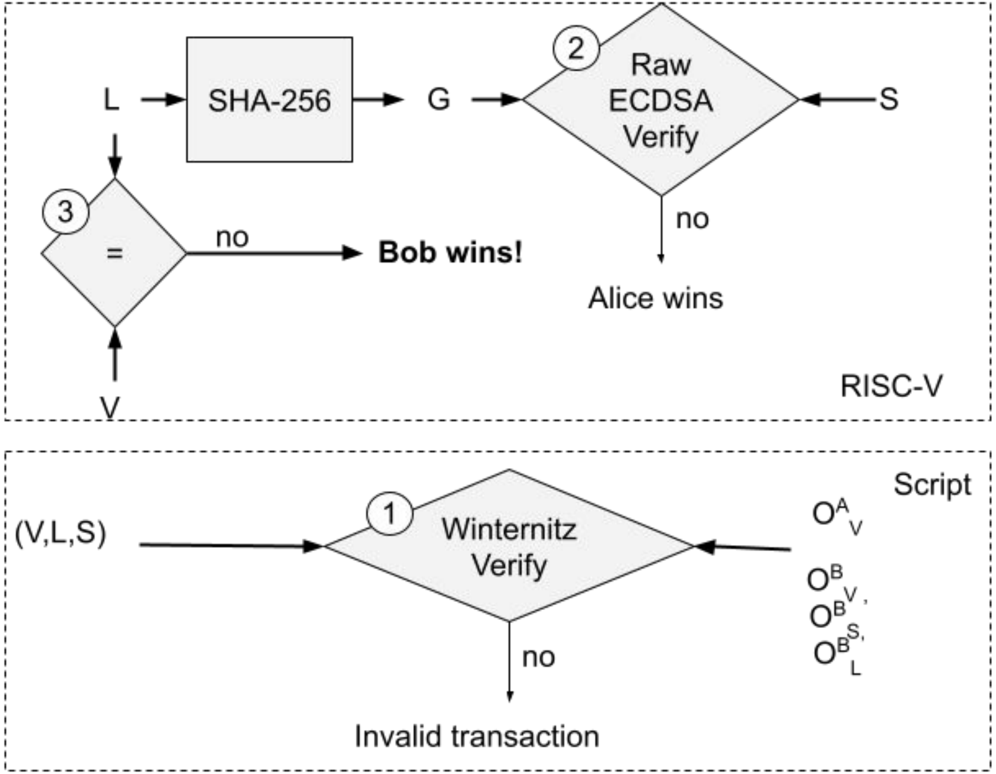
Fairgate

How Alice computes V



Goal: Check S against V without showing the UI to the secondary BitVMX instance

The Consistency Check (Secondary BitVMX instance)



Fairgate

How Program Inputs are structured ?

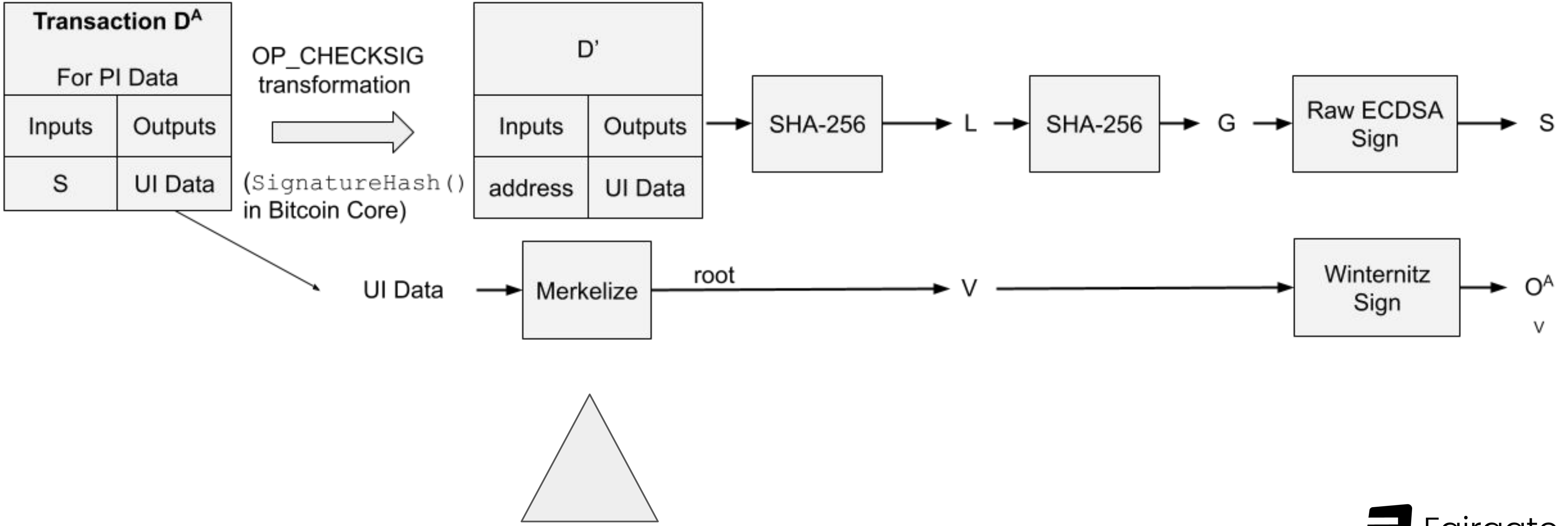
Address type of the handle	UI stored in..	Program Input Type
P2TR	Script	Tapleaf tagged message
P2TR	Tx output	a CTxOut structure (referenced by *sha_single_output*)
P2WSH	WitnessScript	a ScriptPub
P2WSH	Tx output	a CTxOut structure (referenced by *hashOutputs*)
P2SH	ScriptSig	Impossible because the scriptSig is not signed
P2SH	Tx output	a modified transaction

Fairgate

Is there other way to avoid showing the User Input to the secondary BitVMX instance ?

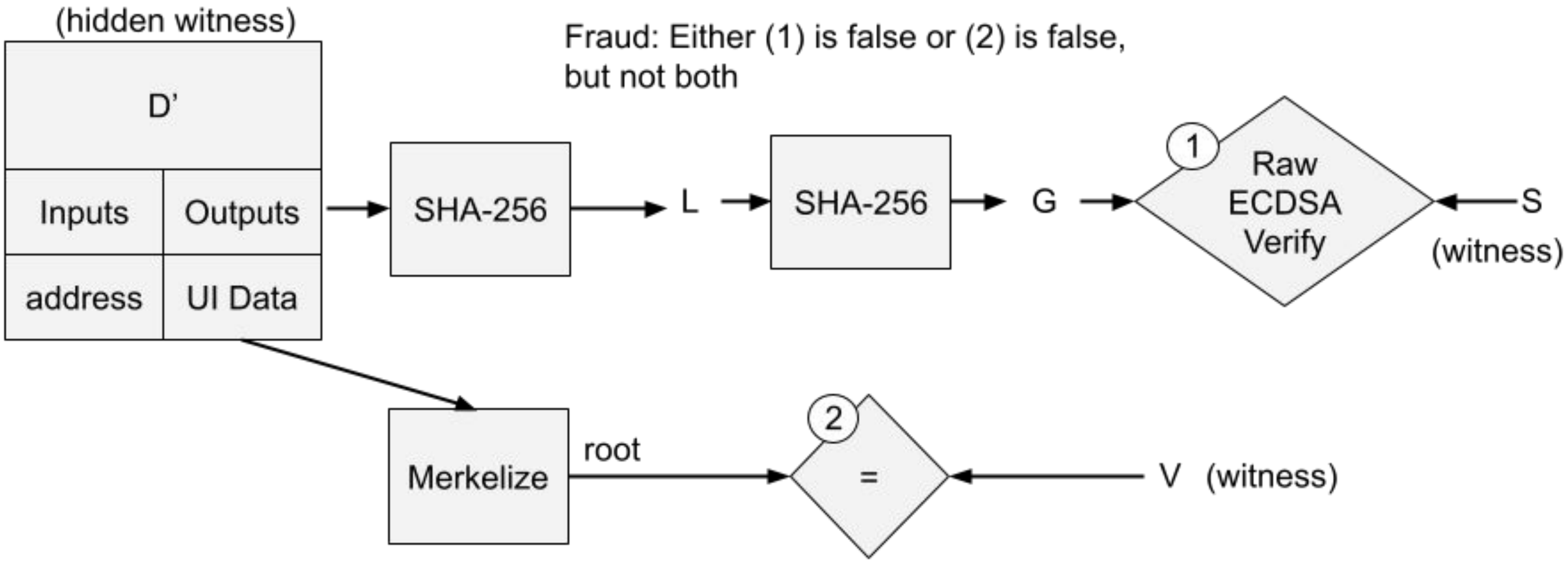
- Yes, make that 2nd instance validate a SNARK that proves D is hashed to a RAM Merkle Root V, making D a hidden witness.
- Use a BitVM1 verifier that uses Merklelized RAM for each step.
- **But**: we need to OT-sign 300 bytes and …. we add a lot of complexity.

Merkelizing RAM: Alice's work



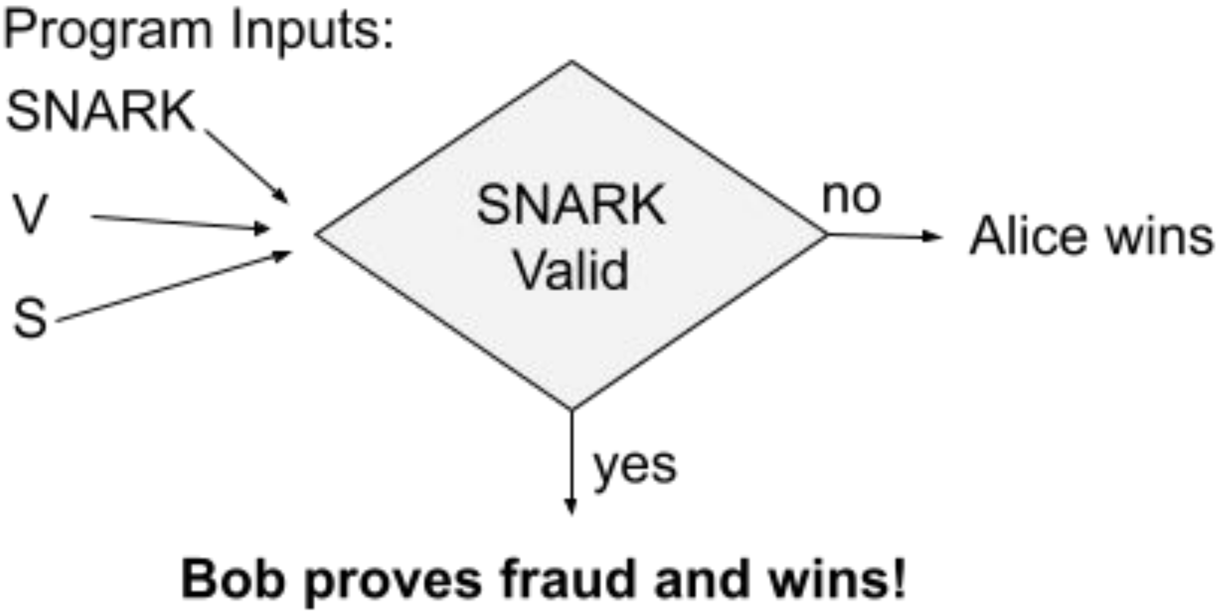
Fairgate

Merkelizing RAM: Bob's SNARK-based fraud proof



Fairgate

Merkelizing RAM: Secondary BitVMX Instance check



Can we use the (unsigned) Input
Data from the RISC-V CPU ?

The ICM CPU Mode

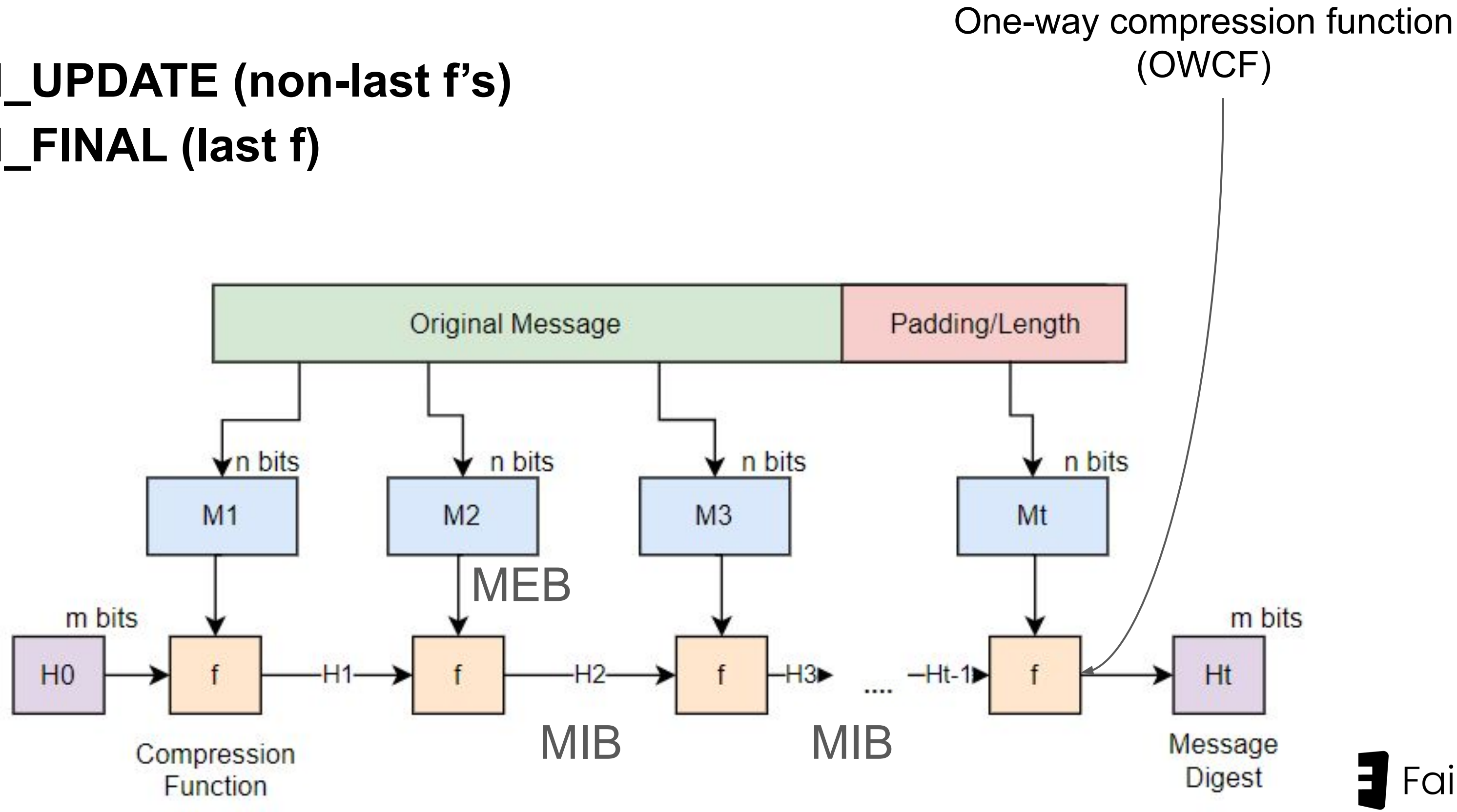
The ICM CPU Mode: New Memory Areas

- **UPI** (Unsigned Program Input): Holds the unsigned program input and this data is RAM memory-mapped
- **SPI** (Signed Program Input). This is a normal OT signed program input (by Alice). When using UPI, this area will be used to store a single hash digest. It's also RAM memory-mapped.
- **MEB** (State Buffer). This is a small 64-byte buffer that is used to store the message input to the SHA-256 function.
- **MIB** (Midstate Buffer). This 32-byte buffer stores the midstate or final state of the SHA-256 function.

Fairgate

The ICM CPU Mode: New Opcodes

- **HASH_UPDATE (non-last f's)**
- **HASH_FINAL (last f)**



Fairgate

The ICM CPU Mode: New RISC-V Opcode

- `lssw imm(rs1)`
- Load Store-Store Word
- Loads word from memory and stores it in two places simultaneously:
  - Same location where it was read (read1.address !!!)
  - In the MEB at address: (read1.address - UPI_base_Offset) % 64 + MEB_base_offset

Fairgate

The ICM CPU Mode: LSSW Trace

- `read1.address`$_i$ `== imm(rs1)`
- `write.address`$_i$ `== read1.address`$_i$
- `read1.value`$_i$ `== word stored in UPI at offset imm(rs1)`
- `write.value`$_i$ `== read1.value`$_i$

When challenging a read/write operation, the `lssw` instruction will be also valid as if the following write trace had been produced:

- `write.address`$_i$ `== ( read1.address`$_i$ `- UPI_base_Offset ) % 64 + MEB_base_offset`

Fairgate

Fun Facts

- Writing a word to two locations related by a known delta is as challengeable as writing it to a single location.
- An instruction `BLKSTORE imm(rs1) <- rs2` that fills 1 Megabyte of RAM with a certain word value can be challenged as easy as a single word write.
- Can a program be made more efficient using BLKSTORE ?

Fairgate

The ICM CPU Mode: Program Sections

- Section A:
  - Run in ICM mode.
  - Program reads the UPI, move the bytes to the MEB, perform hashing operations using HASH_UPDATE, finalize with HASH_FINAL and leave the final result in the MIB.
  - Uses LSSW to move words from UPI to MEB.
- Section B:
  - Normal program operation. Program must parse D’ to extract UI.

Fairgate

The ICM CPU Mode: Section A Code

```
int bmax = maxPI / 64; // Assumes maxPI % 64 == 0, and maxPI is OT-signed by Alice

for (int b=0;i<bmax;b++) {

     for(int j = 0;j<64;j++) {

          MEB[i % 64],UPI[i] = UPI[i]  // use LSSW

          i++;

     }

     HASH_UPDATE  < — CHALLENGABLE

}

padding();

HASH_FINAL    < — CHALLENGABLE
```

Fairgate

Additional Research Done

- Can we use an interactive version of Schnorr-signed messages where Alice and Bob exchange signed messages before the BitVMX protocol starts? YES, but there is no evident use case.
- Can we use Schnorr signatures to publish and sign the midstates within BitVMX ? Yes, but the benefit is not significant
- How does this protocol extend to multiple parties ? Yes!

Fairgate

Summary

- We have presented a new method to sign BitVMX program inputs with ECDSA or Schnorr signatures, instead of using an OTS scheme.
- We achieved a 1:1 data expansion factor (vs 1:200 for Winternitz)
- Now we can verify uncompressed SPV proofs, STARKs, NOVA, bulletproofs.
- To protect from malformed or fraudulent data publications we use a secondary BitVMX.
- We use the Winternitz signature of the sequential hash inside the BitVMX CPU.
- We add a SHA-256 hasher to the BitVMX CPU to hash the program input
- Our most advanced scheme based on enveloping uses standard Bitcoin transactions and has minimal overhead

BitVMX summary

The execution trace is defined as:

$\texttt{trace}_i = \texttt{write.address}_i \mid\mid \texttt{write.value}_i \mid\mid \texttt{writePC.pc}_i$

The full trace is defined as:

$\texttt{full trace}_i = \texttt{read1.address}_i \mid\mid \texttt{read1.value}_i \mid\mid \texttt{read1.lastStep}_i \mid\mid ... \mid\mid \texttt{writePC.pc}_i$

The step hash is defined is:

$\texttt{stepHash}_i = \texttt{h}(\texttt{stepHash}_{i-1} \mid\mid \texttt{trace}_i)$

Fairgate

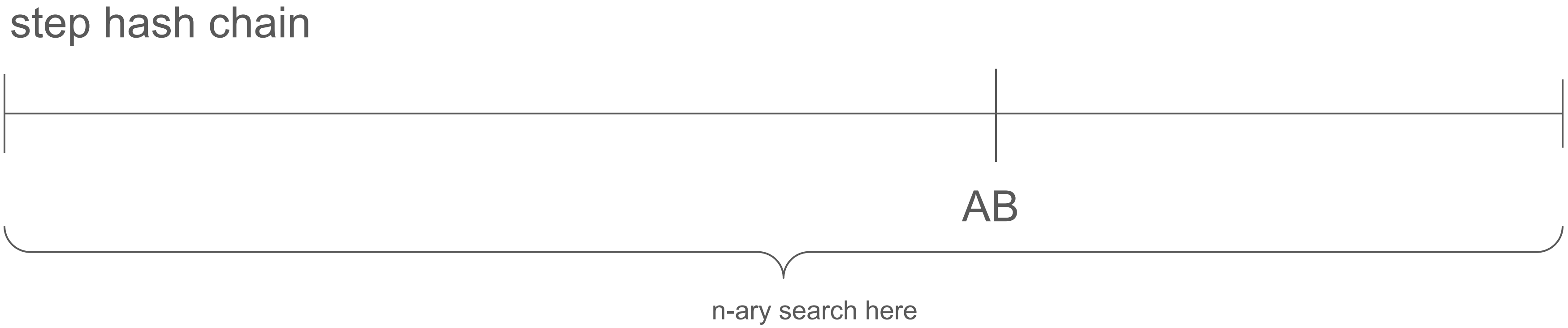
The ICM CPU Mode: New Trace

$\mathtt{trace}_i = \mathtt{write.address}_i \mathbin{||} \mathtt{write.value}_i \mathbin{||} \mathtt{writePC.pc}_i \mathbin{||} \mathbf{MIB}_i \mathbin{||} \mathbf{opcode}_i$

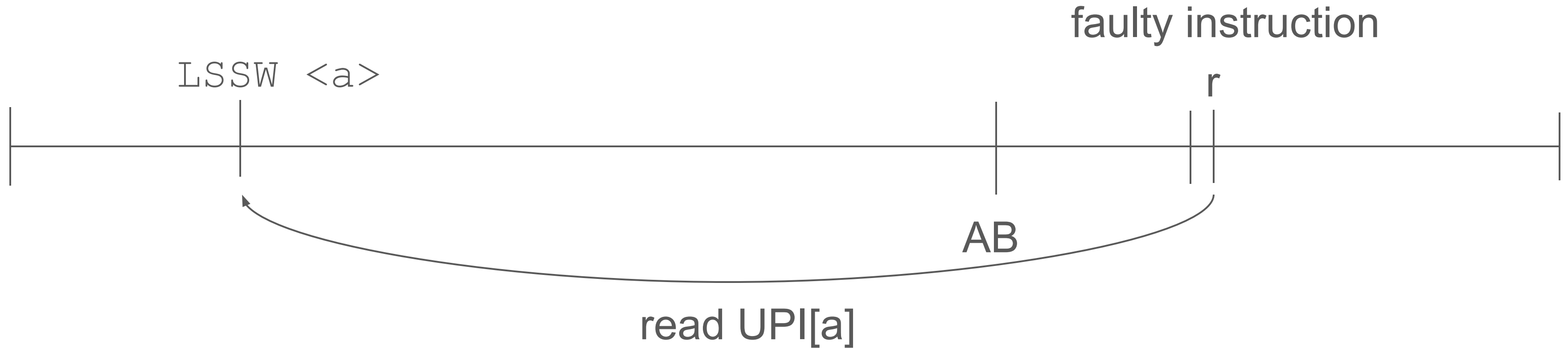
- Trace size: 49 bytes

Fairgate

The ICM CPU Mode: New Search Process



The ICM CPU Mode: New Search Process



Fairgate

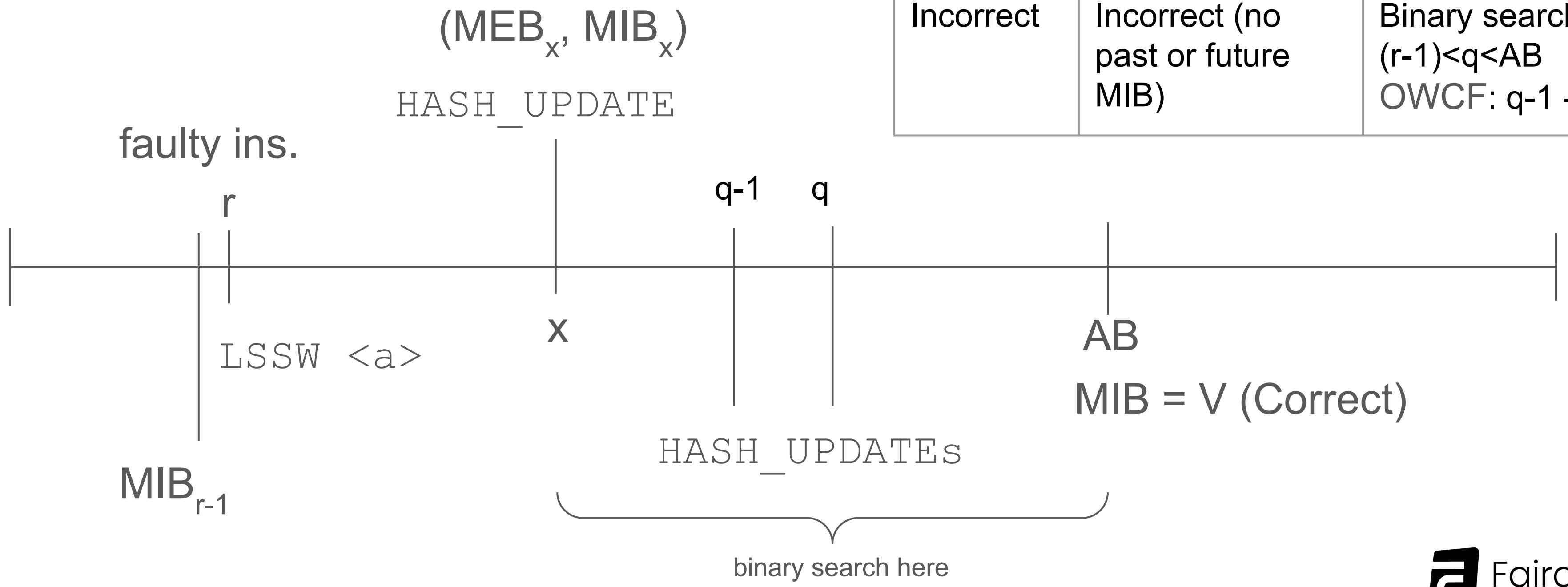
Search Process

Fairgate

Search Process

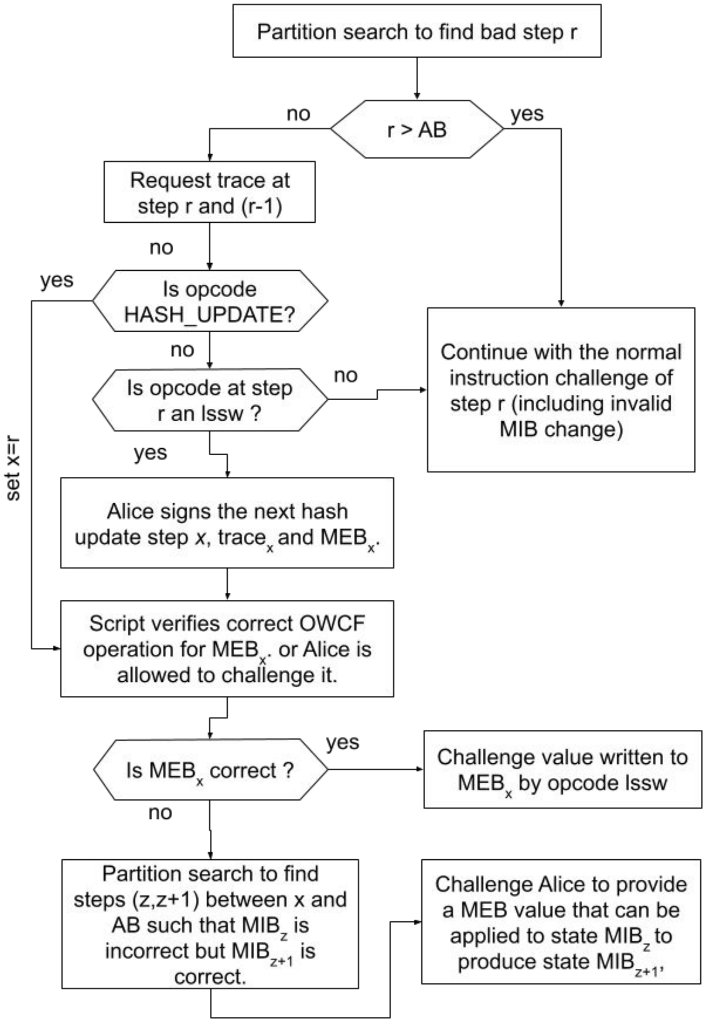
(1) If OWCF(MIB_{r-1}, MEB_x,MIB_x) is incorrect challenge!
(2) Use the table on the right:

MEB_x	MIB_x	Challenge..
Correct	Correct	LSSR write
Correct	Incorrect	not possible
Incorrect	Correct	not possible
Incorrect	Incorrect (no past or future MIB)	Binary search (r-1)<q<AB OWCF: q-1 -> q



Fairgate

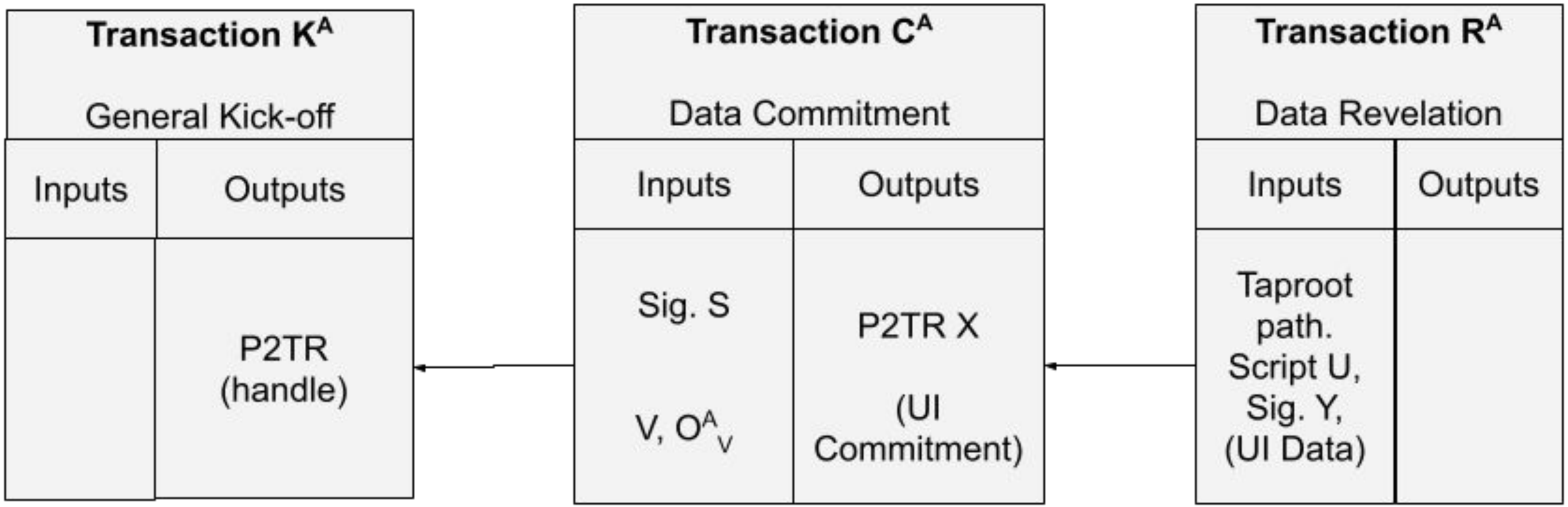
The ICM CPU Mode: New Search Process



Fairgate

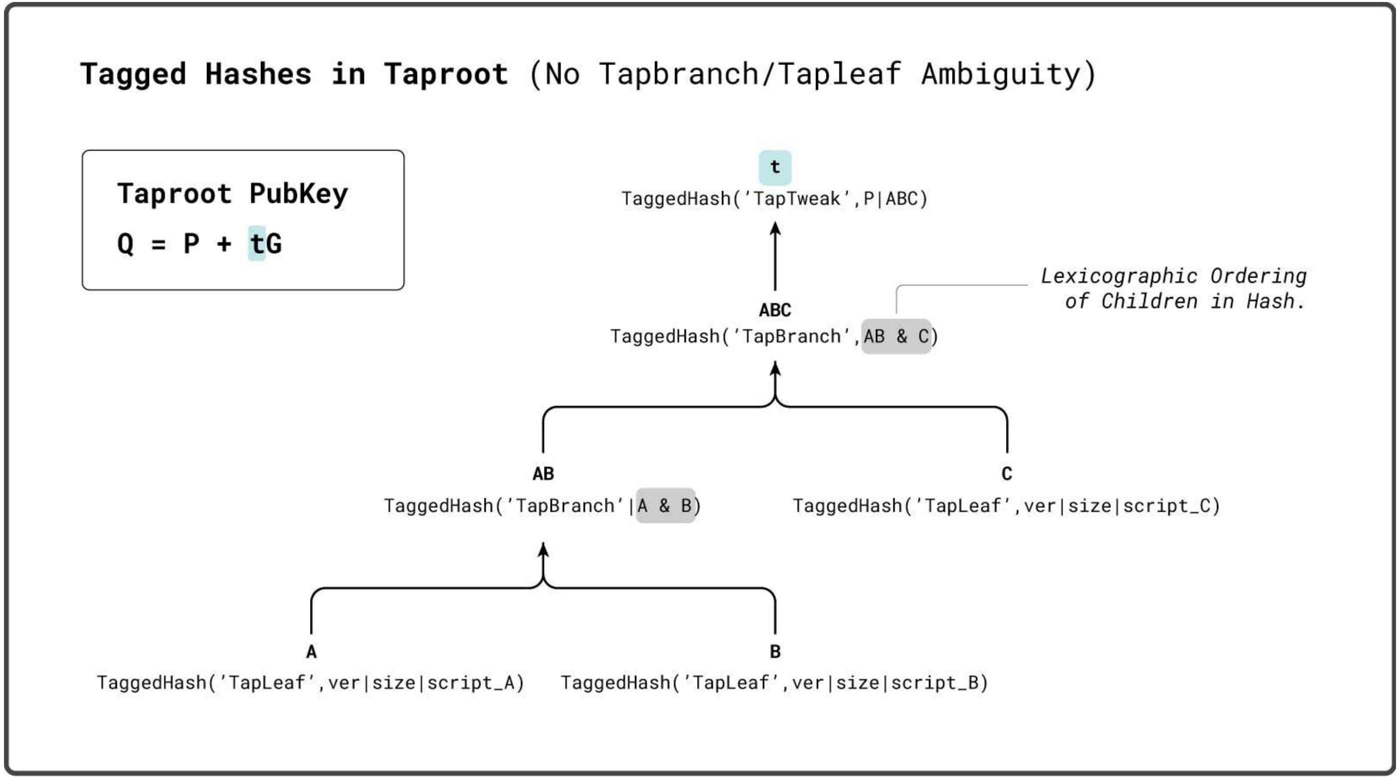
Using Enveloping for the Timelock-based DA Scheme

Using Enveloping for the Timelock-based DA Scheme

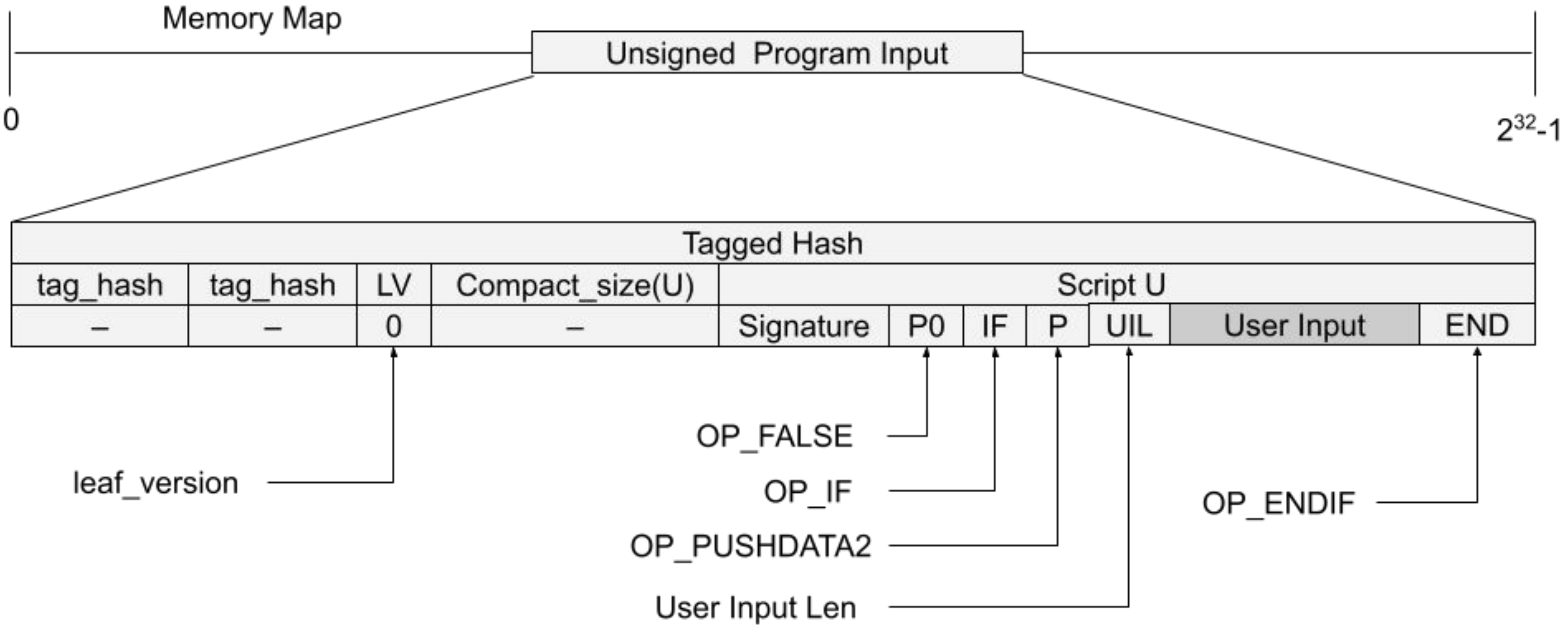


Fairgate

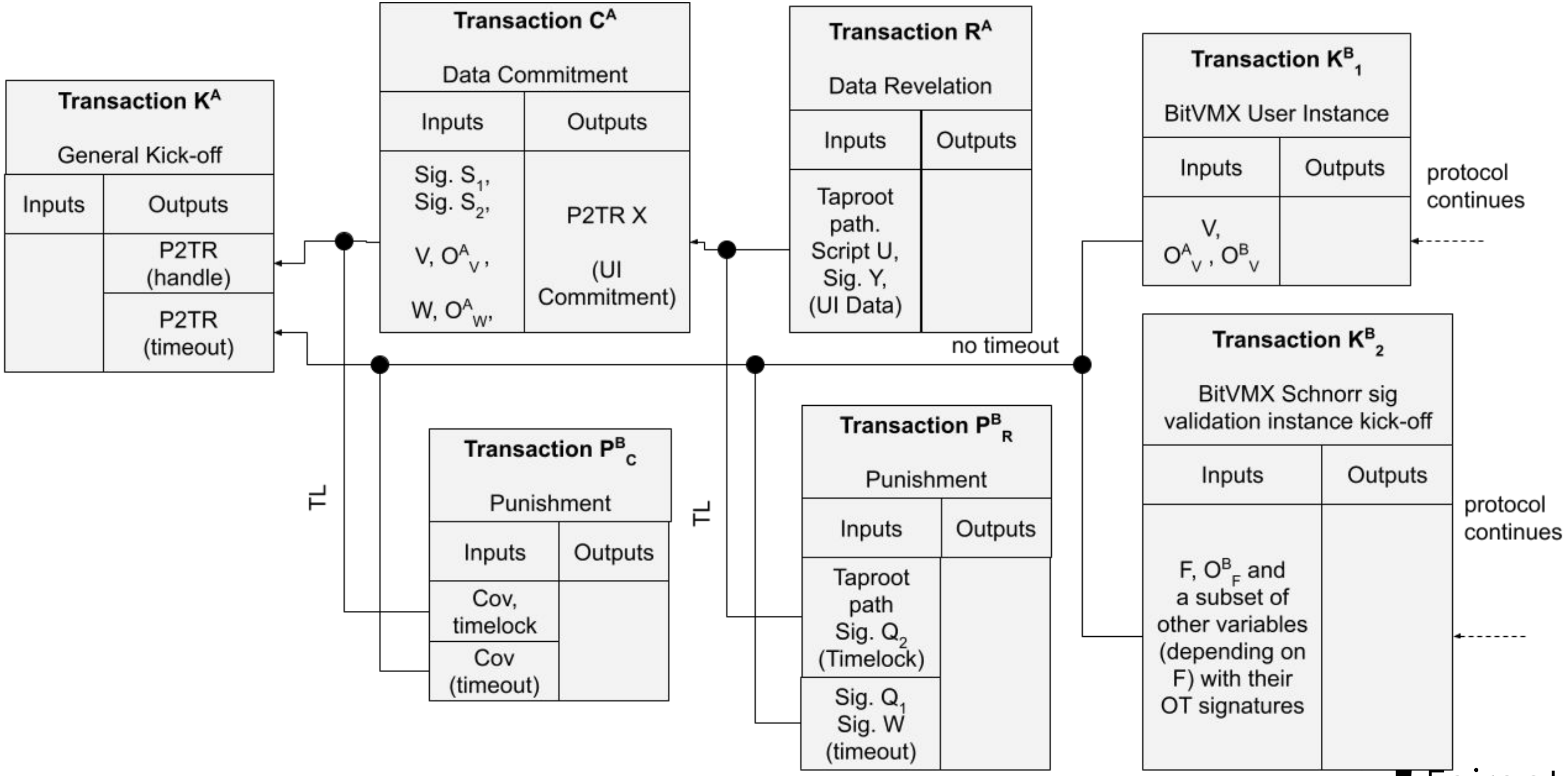
Program Input in Enveloping: TapLeaf



Program Input in Enveloping

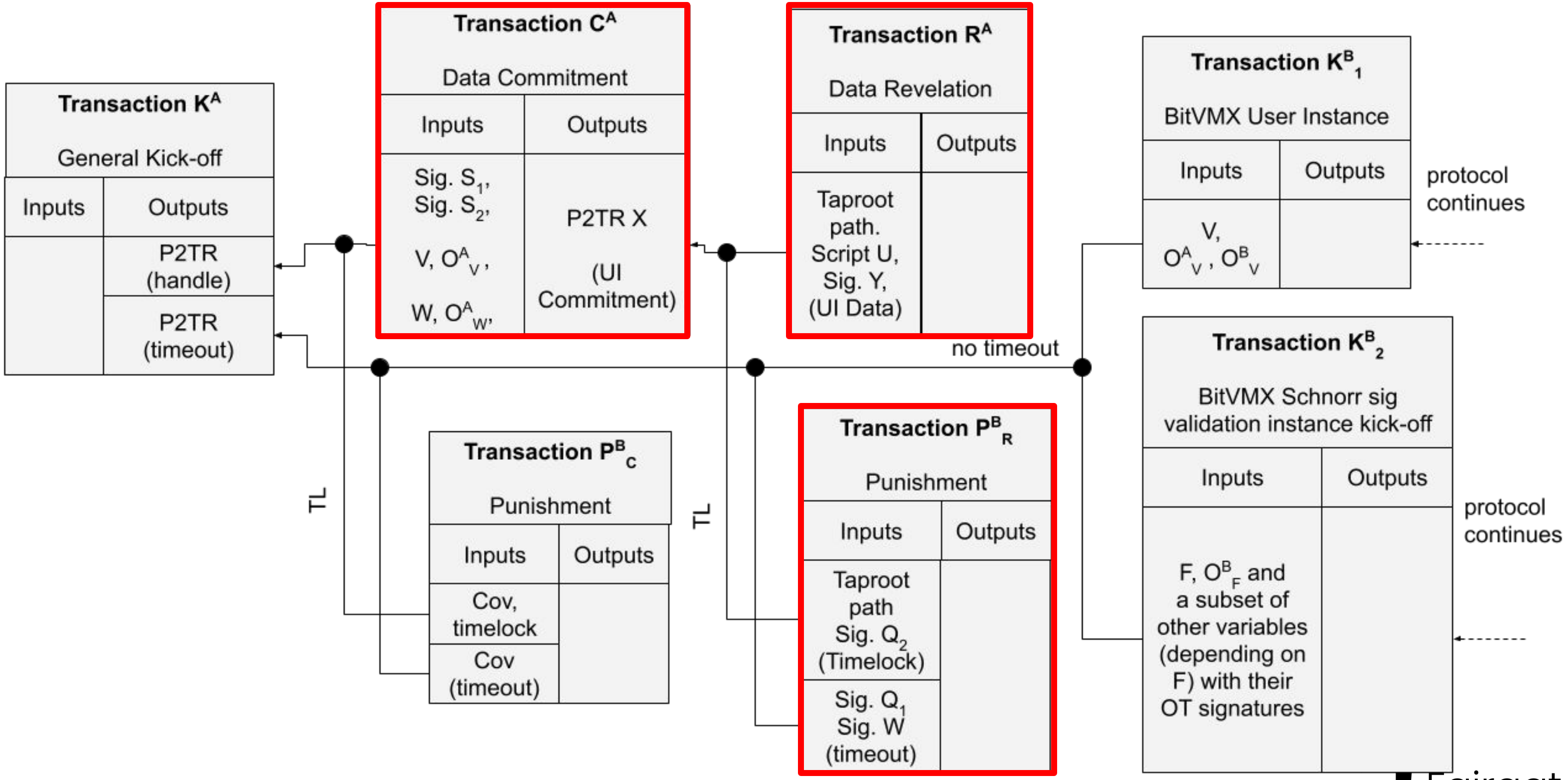


The DA-DAG



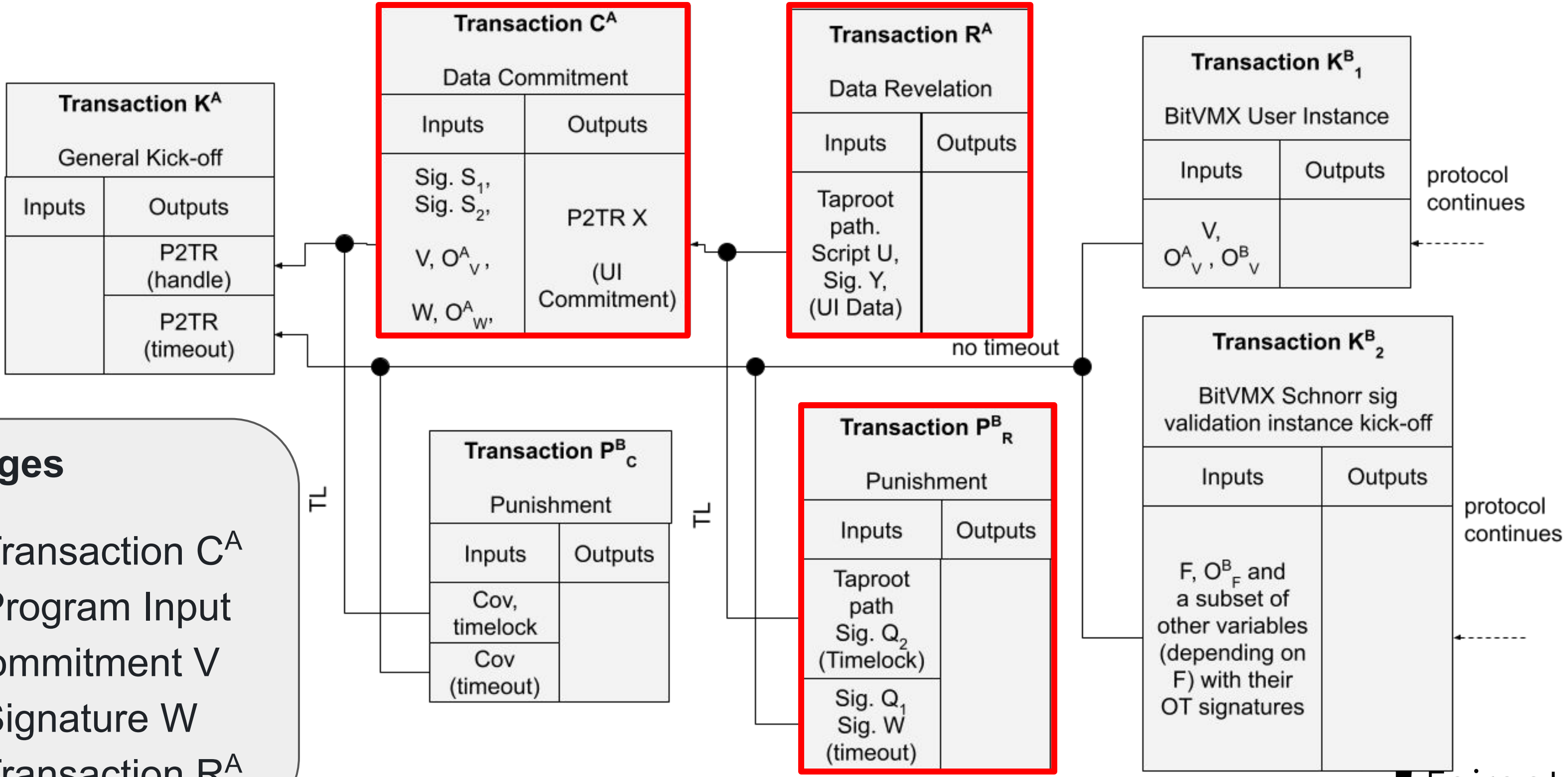
Fairgate

Transactions without covenants



Fairgate

Transactions without covenants

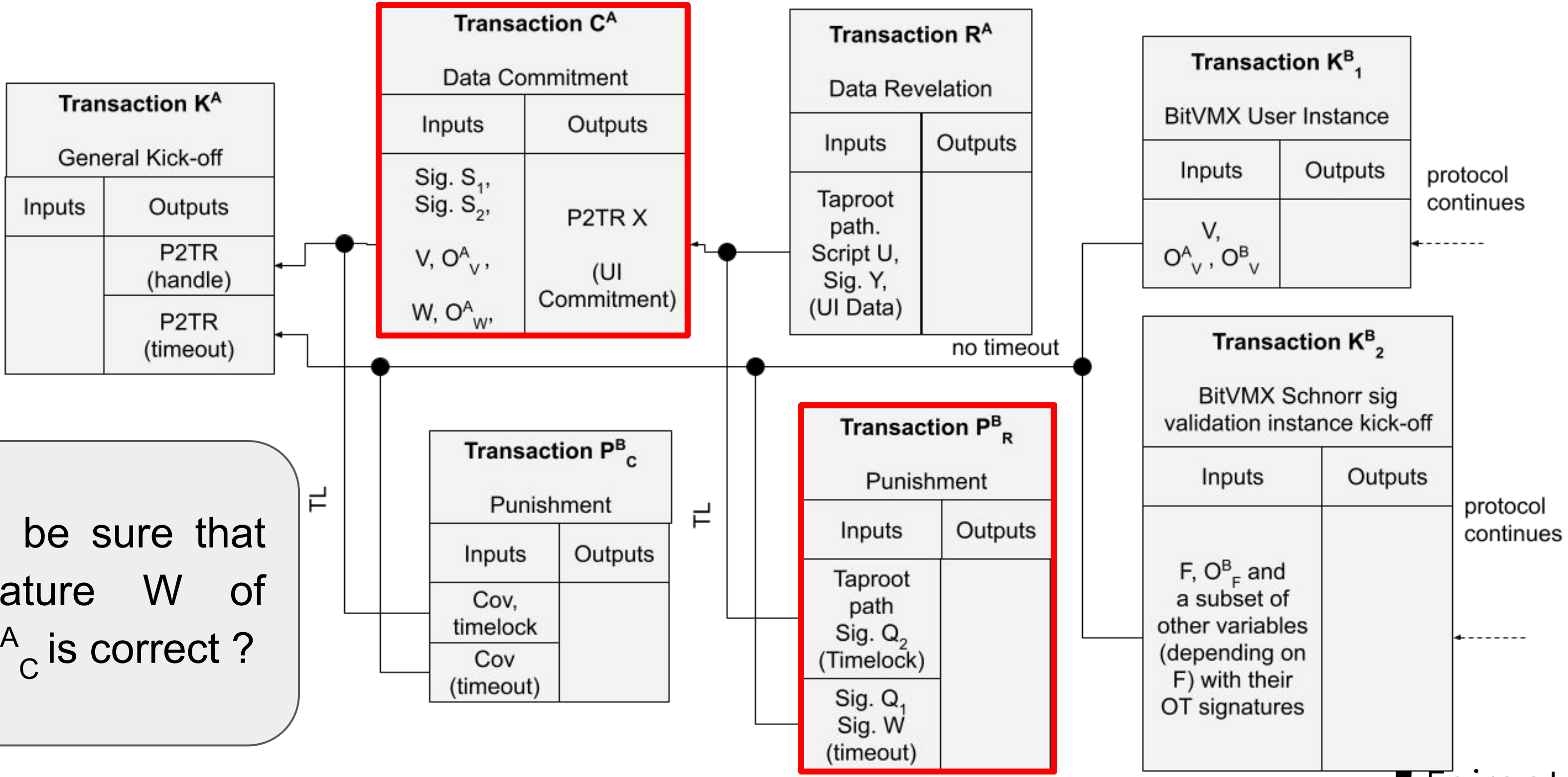


Challenges

1. Invalid Transaction C^A
2. Invalid Program Input Hash Commitment V
3. Invalid Signature W
4. Invalid Transaction R^A

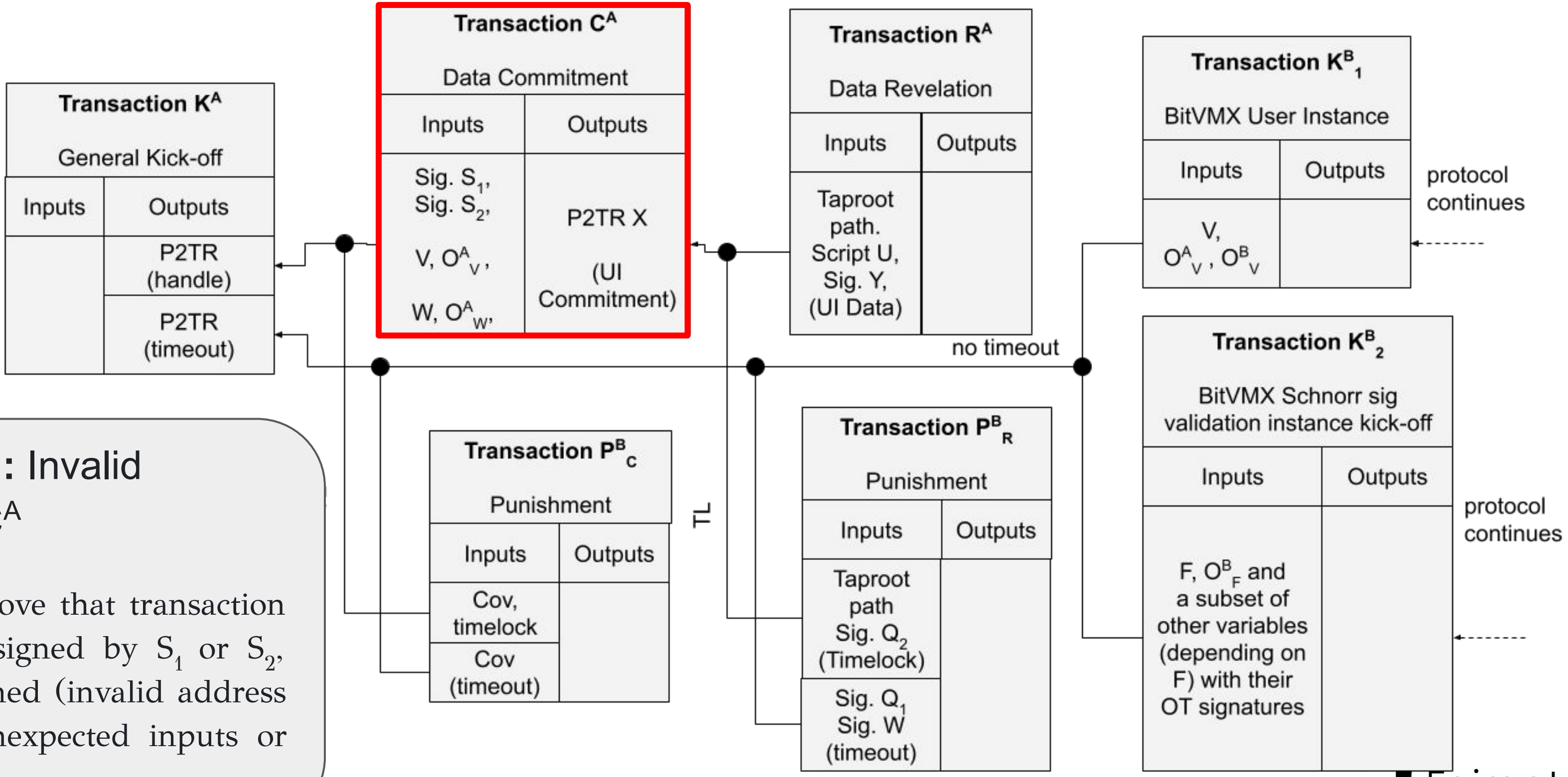
Fairgate

Signatures of Future Transactions



How can we be sure that Alice's signature W of transaction P^A_C is correct ?

Challenge 1: Invalid Transaction C^A



Challenges 1: Invalid Transaction C^A

Bob wants to prove that transaction C^A is correctly signed by S_1 or S_2, but it is malformed (invalid address X, additional unexpected inputs or outputs).

Fairgate

Field	Size	Description
hash_type	1	A byte indicating the which inputs/outputs are being signed
nVersion	4	The transaction version field.
nLockTime	4	The transaction locktime field.
sha_prevouts	32	The SHA-256 hash of the txid+vout outpoints for all the inputs included in the transaction. `~SIGHASH_ANYONECANPAY`
sha_amounts	32	The SHA-256 hash of all the output amount fields for all the inputs included in the transaction. `~SIGHASH_ANYONECANPAY`
sha_scriptpubkeys	32	The SHA-256 hash all the output scriptpubkeys for all the inputs included in the transaction. `~SIGHASH_ANYONECANPAY`
sha_sequences	32	The SHA-256 hash of all the sequence fields for all the inputs included in the transaction.
sha_outputs	32	The SHA-256 hash of all the outputs in the transaction. `~(SIGHASH_NONE` or `SIGHASH_SINGLE)`
spend_type	1	A single byte that encodes the *extension flag* and *annex present* values.
outpoint (input)	36	The txid+vout outpoint of the input being signed for. `SIGHASH_ANYONECANPAY`
amount (input)	8	The amount field of the input being signed for. `SIGHASH_ANYONECANPAY`
scriptPubKey (input)	*variable*	The scriptpubkey of the input being signed for. `SIGHASH_ANYONECANPAY`
nSequence (input)	4	The sequence field of the input being signed for. `SIGHASH_ANYONECANPAY`
input_index	4	The vin of the input being signed for. `~SIGHASH_ANYONECANPAY`

Field	Size	Description
Sha_annex	32	The SHA-256 of the optional annex included at the end of the witness field.
Sha_single_output	32	The SHA-256 of the output opposite the input currently being signed for. `SIGHASH_SINGLE`
tapleaf_hash	32	The leaf hash for the chosen script you're using from the script tree. Script path spend extension (tapscript)
key_version	1	The type of public key used in the leaf script. Script path spend extension (tapscript)
codesep_pos	4	The opcode position of the last OP_CODESEPARATOR in the leaf script. Script path spend extension (tapscript)

What Taproot signs

Fairgate

Detecting Additional Inputs (method 3)

Field	Size	Description
hash_type	1	A byte indicating the which inputs/outputs are being signed
nVersion	4	The transaction version field.
nLockTime	4	The transaction locktime field.
sha_prevouts	32	The SHA-256 hash of the txid+vout outpoints for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_amounts	32	The SHA-256 hash of all the output amount fields for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_scriptpubkeys	32	The SHA-256 hash all the output scriptpubkeys for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_sequences	32	The SHA-256 hash of all the sequence fields for all the inputs included in the transaction.
sha_outputs	32	The SHA-256 hash of all the outputs in the transaction. ~(SIGHASH_NONE or SIGHASH_SINGLE)
spend_type	1	A single byte that encodes the *extension flag* and *annex present* values.
outpoint (input)	36	The txid+vout outpoint of the input being signed for. SIGHASH_ANYONECANPAY
amount (input)	8	The amount field of the input being signed for. SIGHASH_ANYONECANPAY
scriptPubKey (input)	*variable*	The scriptpubkey of the input being signed for. SIGHASH_ANYONECANPAY
nSequence (input)	4	The sequence field of the input being signed for. SIGHASH_ANYONECANPAY
input_index	4	The vin of the input being signed for. ~SIGHASH_ANYONECANPAY

Field	Size	Description
Sha_annex	32	The SHA-256 of the optional annex included at the end of the witness field.
Sha_single_output	32	The SHA-256 of the output opposite the input currently being signed for. SIGHASH_SINGLE
tapleaf_hash	32	The leaf hash for the chosen script you're using from the script tree. Script path spend extension (tapscript)
key_version	1	The type of public key used in the leaf script. Script path spend extension (tapscript)
codesep_pos	4	The opcode position of the last OP_CODESEPARATOR in the leaf script. Script path spend extension (tapscript)

Fairgate

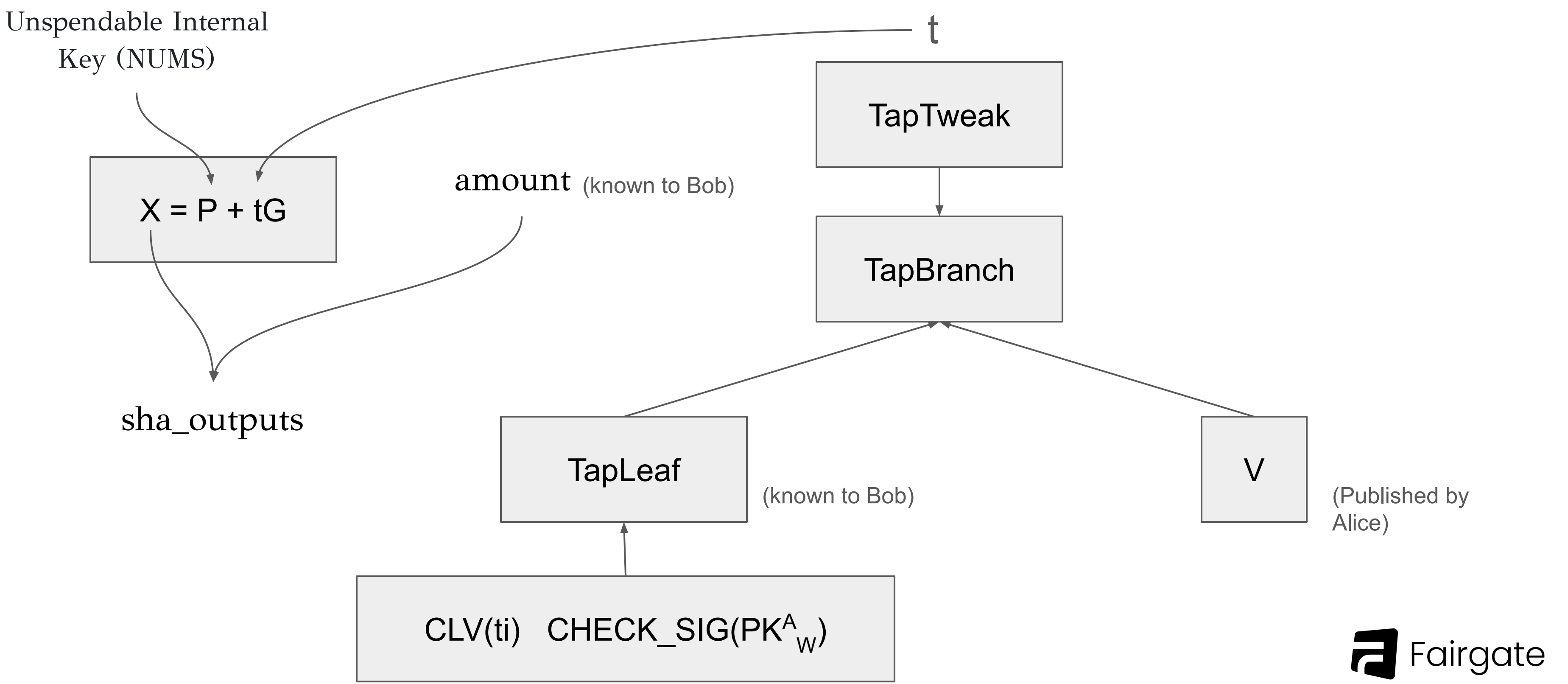
Field	Size	Description
hash_type	1	A byte indicating the which inputs/outputs are being signed
nVersion	4	The transaction version field.
nLockTime	4	The transaction locktime field.
sha_prevouts	32	The SHA-256 hash of the txid+vout outpoints for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_amounts	32	The SHA-256 hash of all the output amount fields for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_scriptpubkeys	32	The SHA-256 hash all the output scriptpubkeys for all the inputs included in the transaction. ~SIGHASH_ANYONECANPAY
sha_sequences	32	The SHA-256 hash of all the sequence fields for all the inputs included in the transaction.
sha_outputs	32	The SHA-256 hash of all the outputs in the transaction. ~(SIGHASH_NONE or SIGHASH_SINGLE)
spend_type	1	A single byte that encodes the *extension flag* and *annex present* values.
outpoint (input)	36	The txid+vout outpoint of the input being signed for. SIGHASH_ANYONECANPAY
amount (input)	8	The amount field of the input being signed for. SIGHASH_ANYONECANPAY
scriptPubKey (input)	*vari able*	The scriptpubkey of the input being signed for. SIGHASH_ANYONECANPAY
nSequence (input)	4	The sequence field of the input being signed for. SIGHASH_ANYONECANPAY
input_index	4	The vin of the input being signed for. ~SIGHASH_ANYONECANPAY

Field	Size	Description
Sha_annex	32	The SHA-256 of the optional annex included at the end of the witness field.
Sha_single_output	32	The SHA-256 of the output opposite the input currently being signed for. SIGHASH_SINGLE
tapleaf_hash	32	The leaf hash for the chosen script you're using from the script tree. Script path spend extension (tapscript)
key_version	1	The type of public key used in the leaf script. Script path spend extension (tapscript)
codesep_pos	4	The opcode position of the last OP_CODESEPARATOR in the leaf script. Script path spend extension (tapscript)

Detecting Additional Outputs

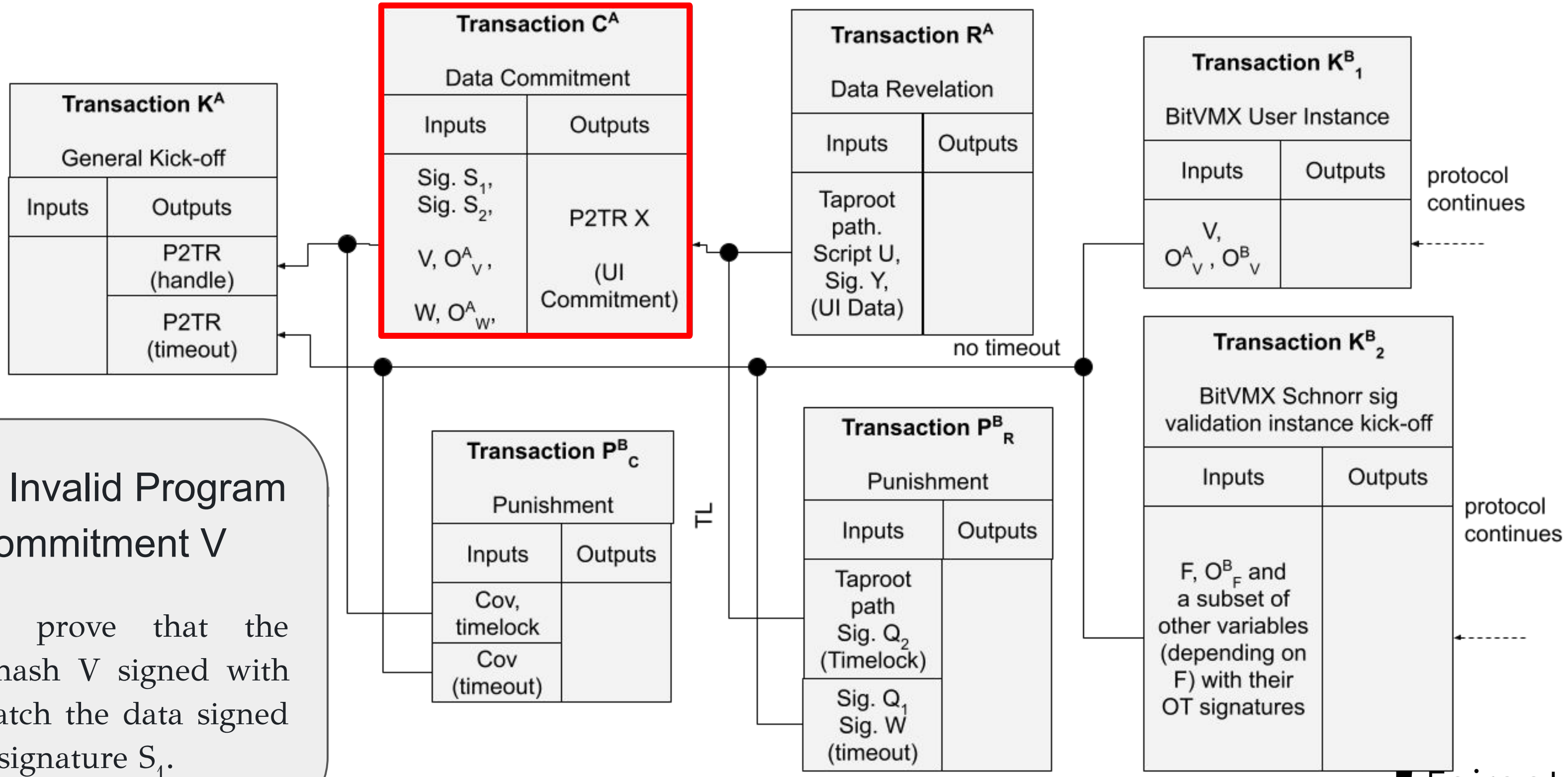
Fairgate

Detecting an invalid output X



Fairgate

Challenge 2: Invalid Program Input Hash V

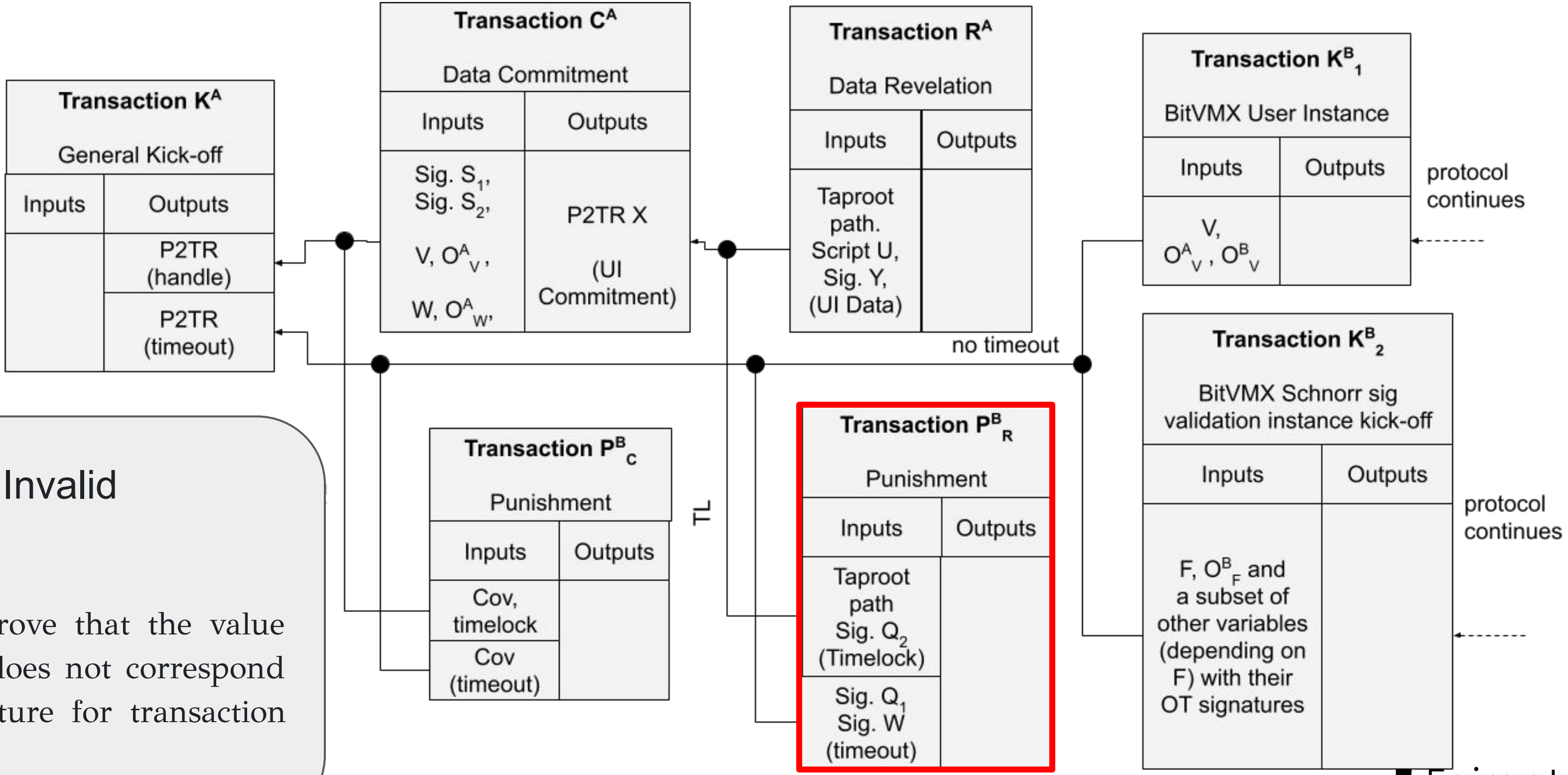


Challenge 2: Invalid Program Input Hash Commitment V

Bob wants to prove that the program input hash V signed with O^A_V does not match the data signed with Schnorr in signature S_1.

Fairgate

Challenge 3: Invalid Signature W

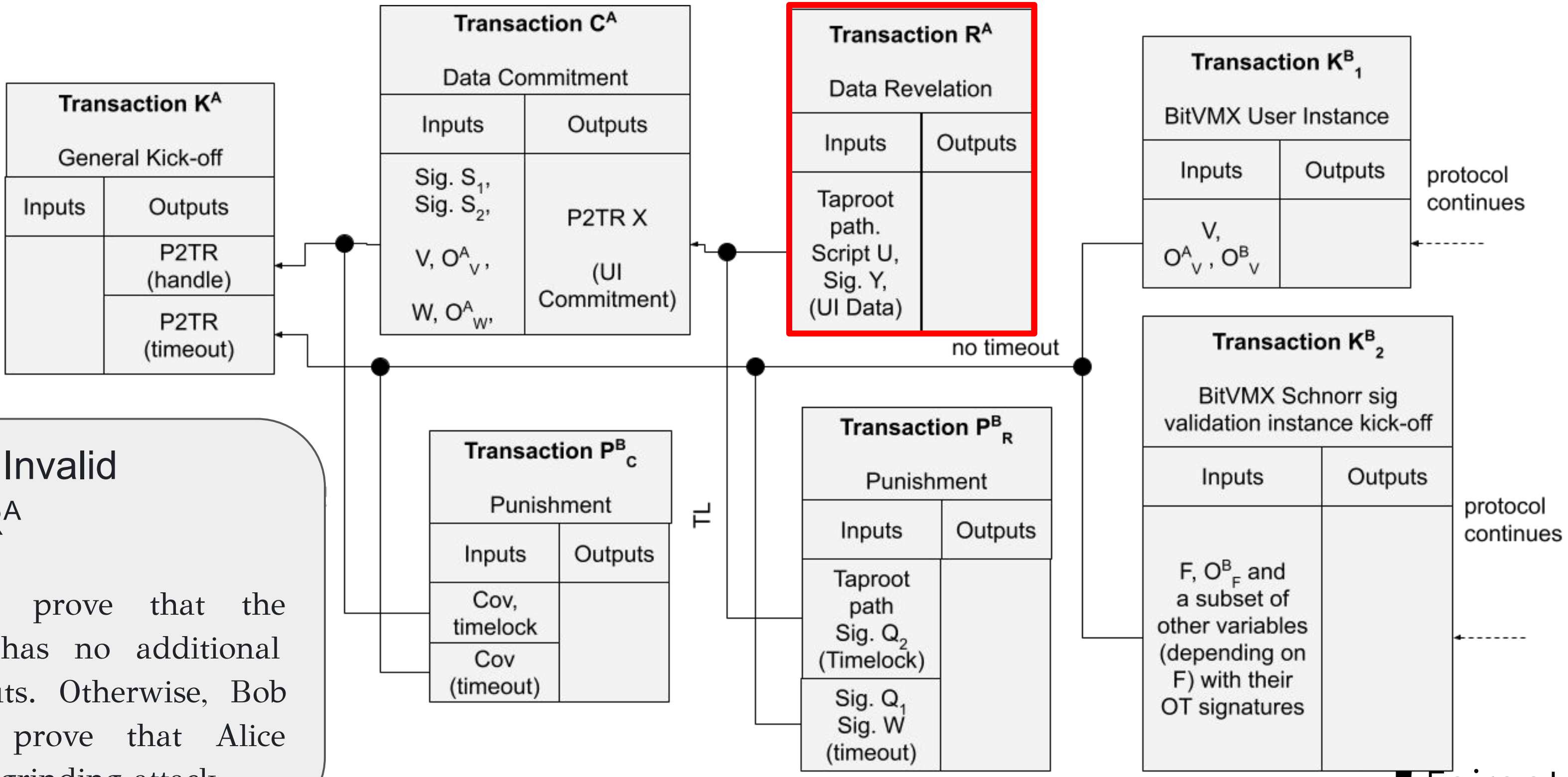


Challenge 3: Invalid Signature W

Bob wants to prove that the value W given in C^A does not correspond to a valid signature for transaction P^B_R.

Fairgate

Challenge 4: Invalid Transaction R^A



Challenge 4: Invalid Transaction R^A

Bob wants to prove that the transaction R^A has no additional inputs or outputs. Otherwise, Bob may want to prove that Alice performed a I/O grinding attack.

Future Research

- Can we use an interactive version of Schnorr-signed messages where Alice and Bob exchange signed messages before the BitVMX protocol starts?
- Can we use Schnorr signatures to publish and sign the midstates within BitVMX ?
- How does this protocol extend to multiple parties ?

Summary

- We have presented a new method to sign BitVMX program inputs with ECDSA or Schnorr signatures, instead of using an OTS scheme.
- We achieved a 1:1 data expansion factor (vs 1:200 for Winternitz)
- Now we can verify uncompressed SPV proofs, STARKs, NOVA, bulletprofs.
- To protect from malformed or fraudulent data publications we use a secondary BitVMX.
- We use the Winternitz signature of the sequential hash inside the BitVMX CPU.
- We add a SHA-256 hasher to the BitVMX CPU to hash the program input
- Our most advanced scheme based on enveloping uses standard Bitcoin transactions and has minimal overhead

ESSPI: ECDSA/Schnorr Signed Program Input for BitVMX

Sergio Demian Lerner, Martin Jonas, and Ariel Futoransky

"*To deeply understand most things, it takes more than one hour of a remote meeting*" - Old proverb

Fairgate