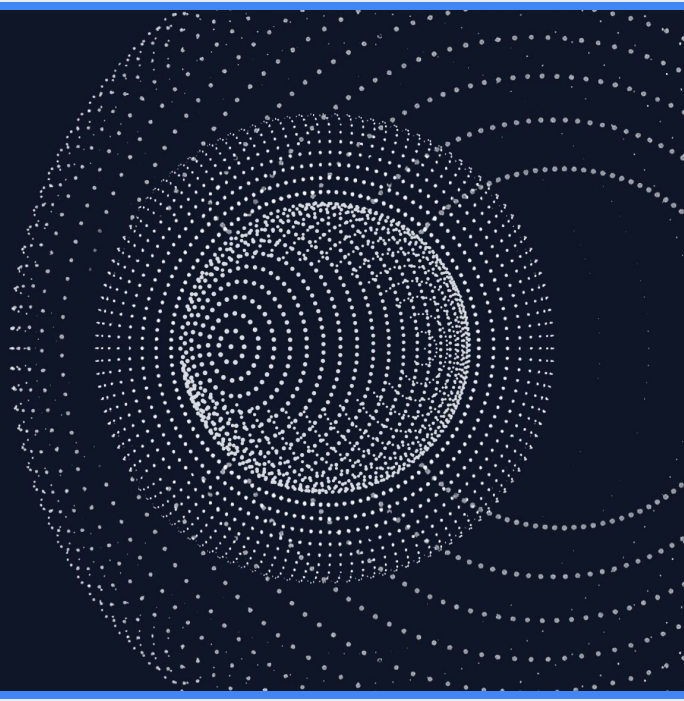
Fairgate

Building Secure and Watchtower-efficient Bitcoin Payment Channels with BitVMX



Sergio Demian Lerner | CTO, Fairgate Labs

OCT 2025

Disclaimer

I am not an expert on the Lightning Network protocol. My perspective comes from the BitVM/BitVMX domain, where I approach the subject with fresh eyes and an open mind.

I don’t use the same vocabulary.

Agenda

1. Why payment channels
2. Basic Payment Channel types
3. One Time Signatures in Bitcoin
4. OTS-based Payment Channel
5. BitVMX CPU & GC
6. BitVMX-Based Payment Channels
7. Summary

Why payment channels?

Bitcoin's base layer (on-chain) has inherent throughput, latency, and cost limitations:

- Blocks are finite in size and time (≈10 min average), so transactions compete for space.
- Each on-chain transaction incurs a fee and confirmation delay.
- Many real-world use cases (micropayments, high-frequency payments) cannot tolerate high fees or multi-minute wait times.

Why payment channels?

Payment channels are a way to move transactions off-chain between parties, while still anchoring to the blockchain for security and settlement.

Types of Payment Channels

Design	Key Idea / Mechanism	What problems it solves	Drawbacks / tradeoffs
Nakamoto	Transaction replacement before confirmation (Unidirectional)	Early idea by Satoshi	Very fragile; vuln. to transaction malleability
Spillman-style	**Unidirectional**, with expiration	**Enforced onchain**. Simple.	Unidirectional; Close before expiry; malleability
CLTV-style (CHECKLOCKTIMEVERIFY)	Use CLTV. Unidirectional, with expiration	**Eliminates malleability**. Refund enforced by scripts	Still unidirectional; expiration is required;

Types of Payment Channels

Design	Key Idea / Mechanism	What problems it solves	Drawbacks / tradeoffs
Duplex / Decker-Wattenhofer	Combine two opposite-direction uni. channels + “invalidation tree”	(Almost) **Bidirectional**.	Finite number of invalidations. Complexity of invalidation tree
Poon-Dryja / “Lightning-style”	Lock funds in a 2-of-2 multisig.	**Bidirectional,** indefinite updates, HTLCs are core of **LN**	**Complex**. Requires careful penalty logic
Eltoo / Decker -Russell-Osuntokun	Simplify dispute logic: new state “cancels” prior via a seq. number	Straightforward handling of updates. **Efficient Watchtowers**	Requires sighash flags `ANYPREVOUT` or `NOINPUT`.
Outpost / Khabbazian et al.	Moves the heavy blob off the tower (partially on-chain)	**Efficient Watchtowers** without soft-fork. Enabled by OP_RETURN > 80 bytes	Higher onchain cost and protocol complexity. Unilateral exit consumes ~1200 WU.

New Types of Payment Channels

Design	Key Idea / Mechanism	What problems it solves	Drawbacks / tradeoffs
OTS-Based	Sign state updates sequence numbers with OTS	**Simple. Efficient Watchtowers**.	Unilateral exit consumes ~750 WU. Disputes consume 750 WU.
BitVMX-Based	Uses BitVMX for dispute resolution	**Efficient Watchtowers, Higher Privacy, Full Programmability.**	Unilateral exit consumes ~750 WU. Disputes consume ~8000 WU

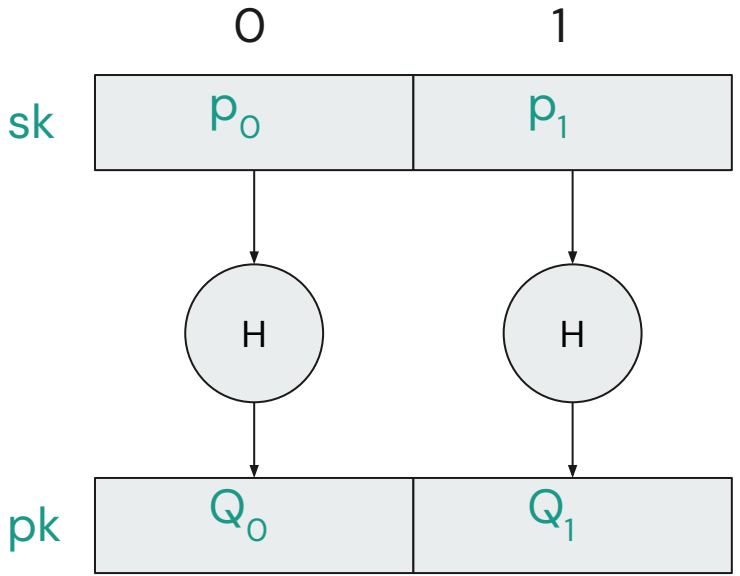
One-Time Signatures

The one-time signature scheme is defined by three algorithms:

- **OTKeyGen**(λ) $\rightarrow$ (sk, pk). The secret key is a random value sk, and pk is the associated public key
- **OTSign**(sk, m) $\rightarrow \sigma$. Where sk is the secret key, m is the message.
- **OTVerify**(pk, m, σ) $\rightarrow$ Accept or reject a signature.

1-Bit Lamport Signatures

- **OTKeyGen**(λ) → (sk, pk). sk = (p_0,p_1), pk= (Q_0,Q_1) such that H(p_1) = Q_1 and H(p_0) = Q_0
- **OTSign**(sk, m) → σ = { if m=0 then p_0 else p_1 }
- **OTVerify**(pk, m, σ) → { if m=1 then accept if H(σ)=Q_1 else accept if H(σ)=Q_0 }



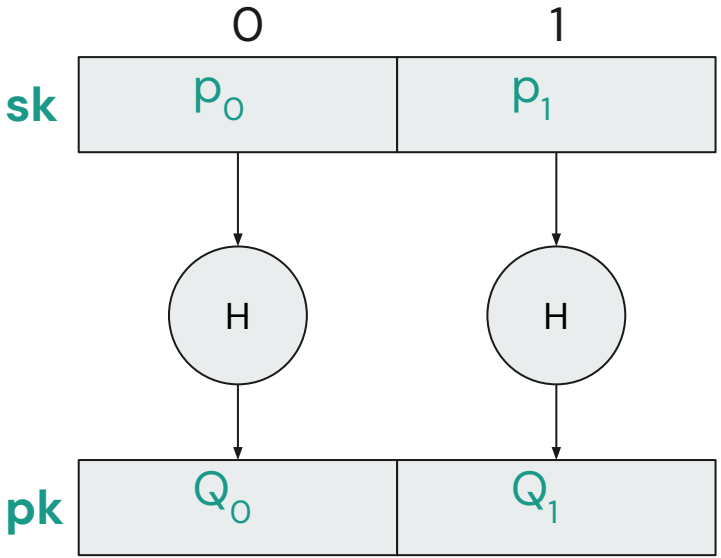
1-Bit Lamport Signatures

```
// Input sig (21b), m (1b)
OP_IF // 1
      OP_HASH160
      OP_PUSH <Q1> // 21b
      OP_EQUALVERIFY
OP_ELSE // 0
      OP_HASH160
      OP_PUSH <Q0> // 21b
      OP_EQUALVERIFY
OP_ENDIF
```

72 WU/bit
(incl. input)

or 18
vbytes/bit

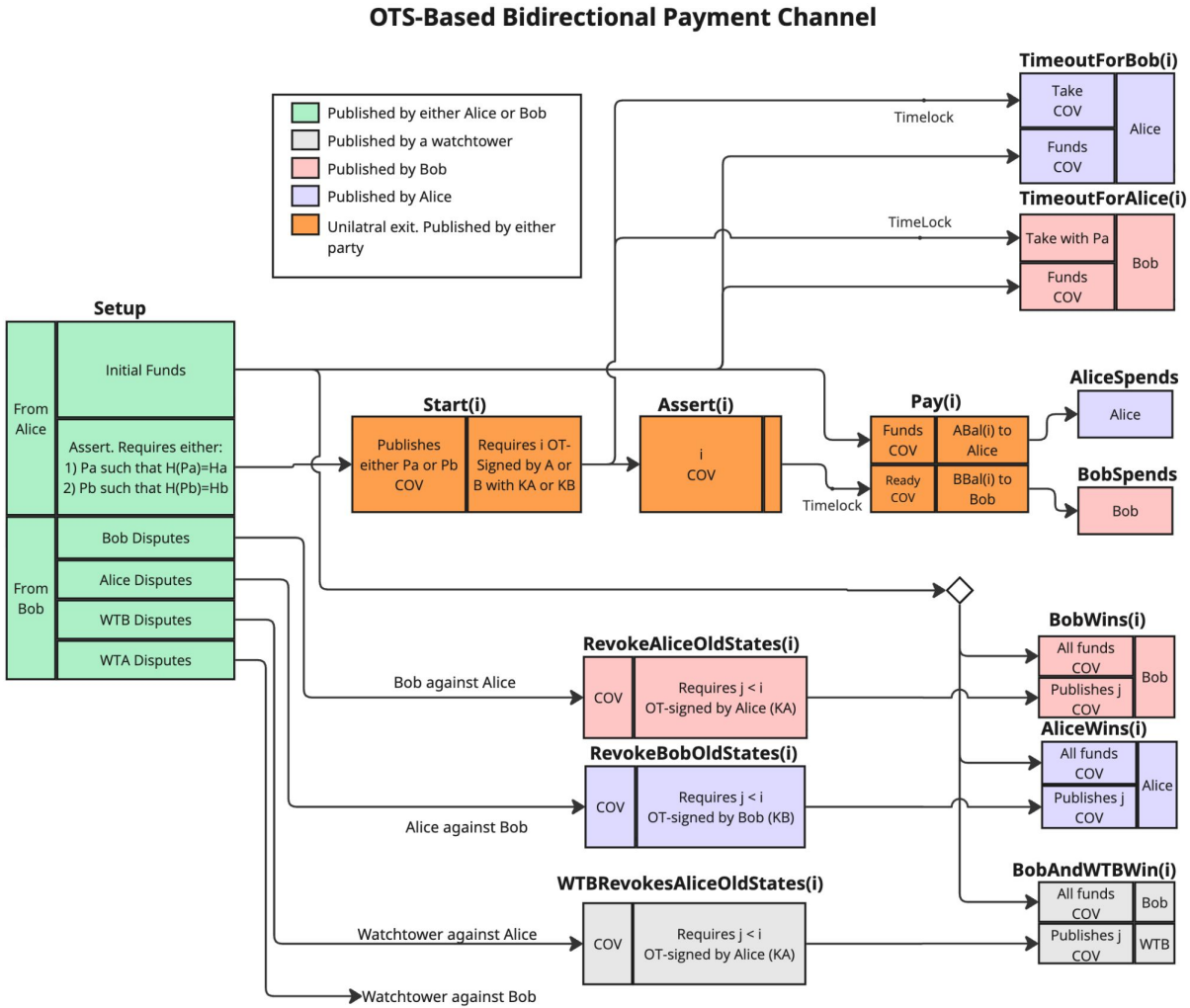
32-bit -> 576
vbytes



Optimization:
f,t length can be 128-bits

OTS-based Payment Channels

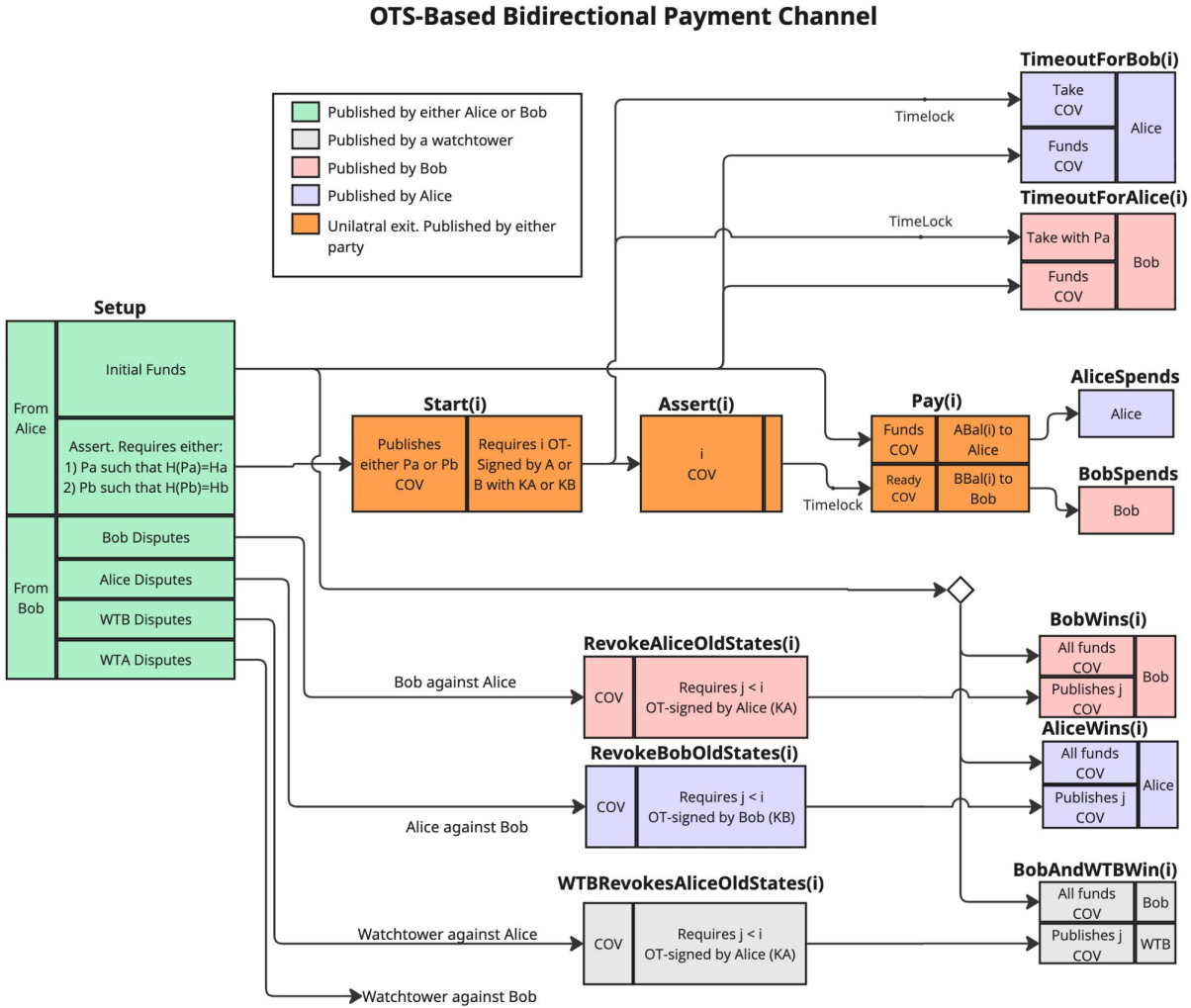
OTS-Based Payment Channel



Fairgate

OTS-Based Payment Channel

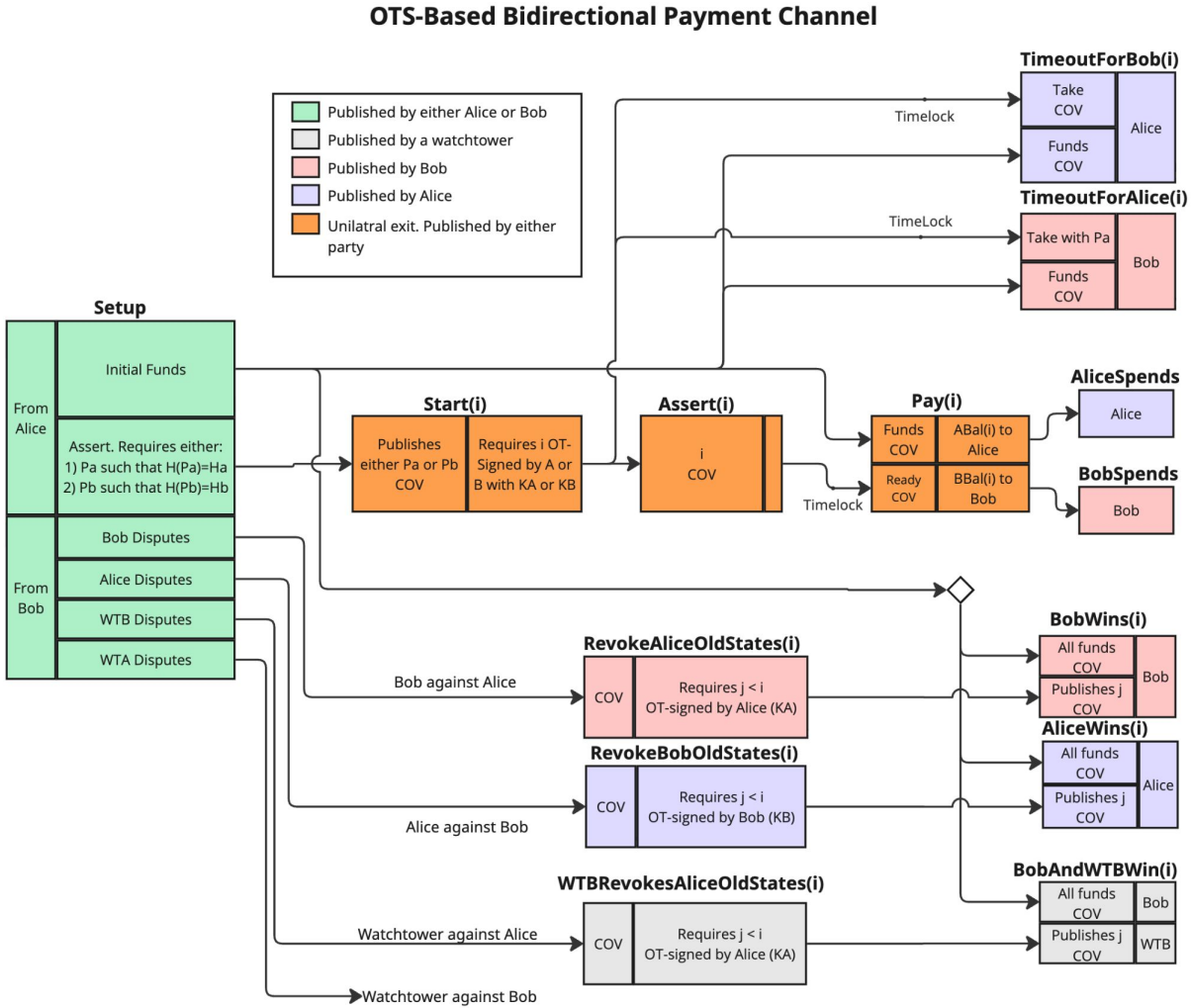
1. `Start(i)`, `Pay(i)`, `Assert(i)` commit to each new state i forcing the signature of i to split funds.
2. `RevokeXOldState(i)` punishes old states j (j< i) to be signed.
3. Unilateral exit path is the same for both parties: Pre-images Pa/Pb are used to decide which timeouts apply



Fairgate

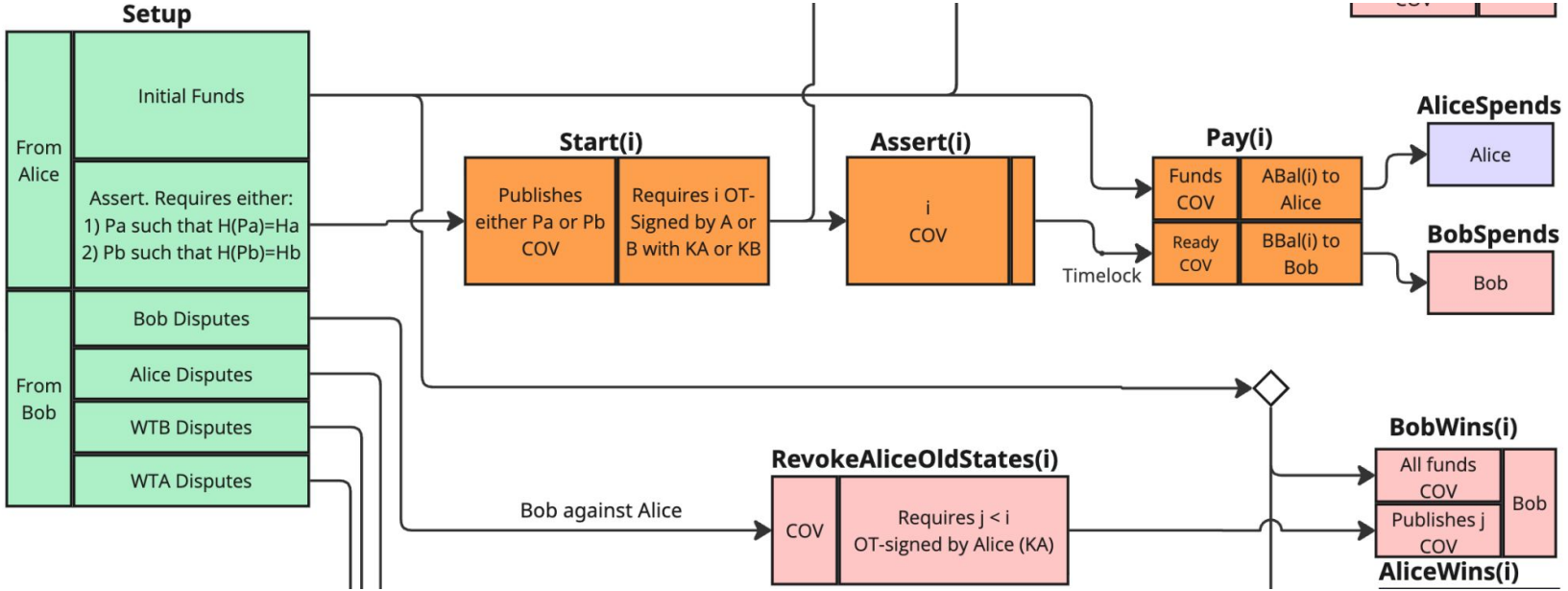
Setup

1. `Pa` and `Pb` such that (`Ha = H(Pa)` and `Hb = H(Pb)`).
2. OTS key-pairs `KA` and `KB`
3. `Setup` tx onchain.
4. `Pay(0)`, `Assert(0)`, and `Start(0)`.



Fairgate

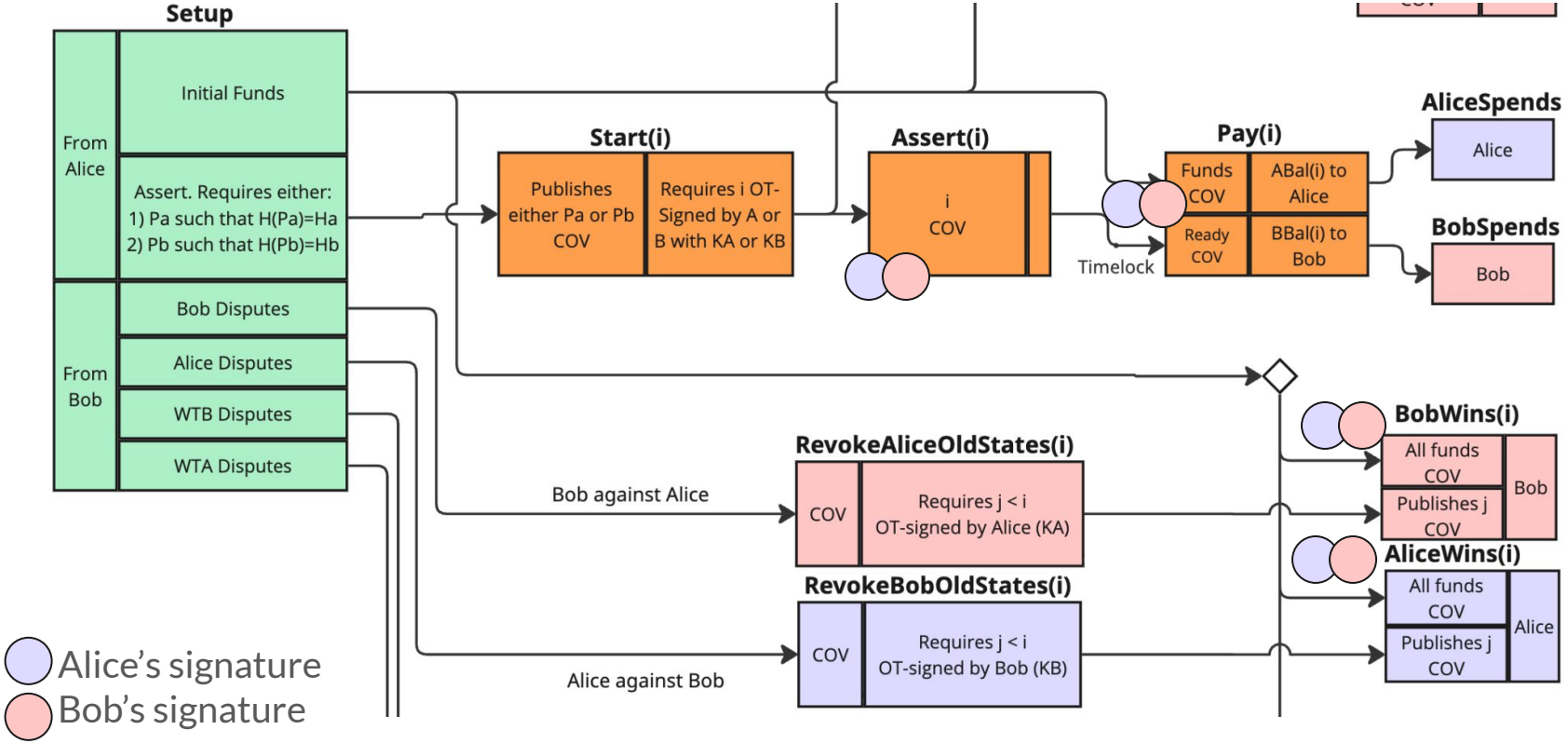
OTS-Based PC State Update (Alice pays Bob)



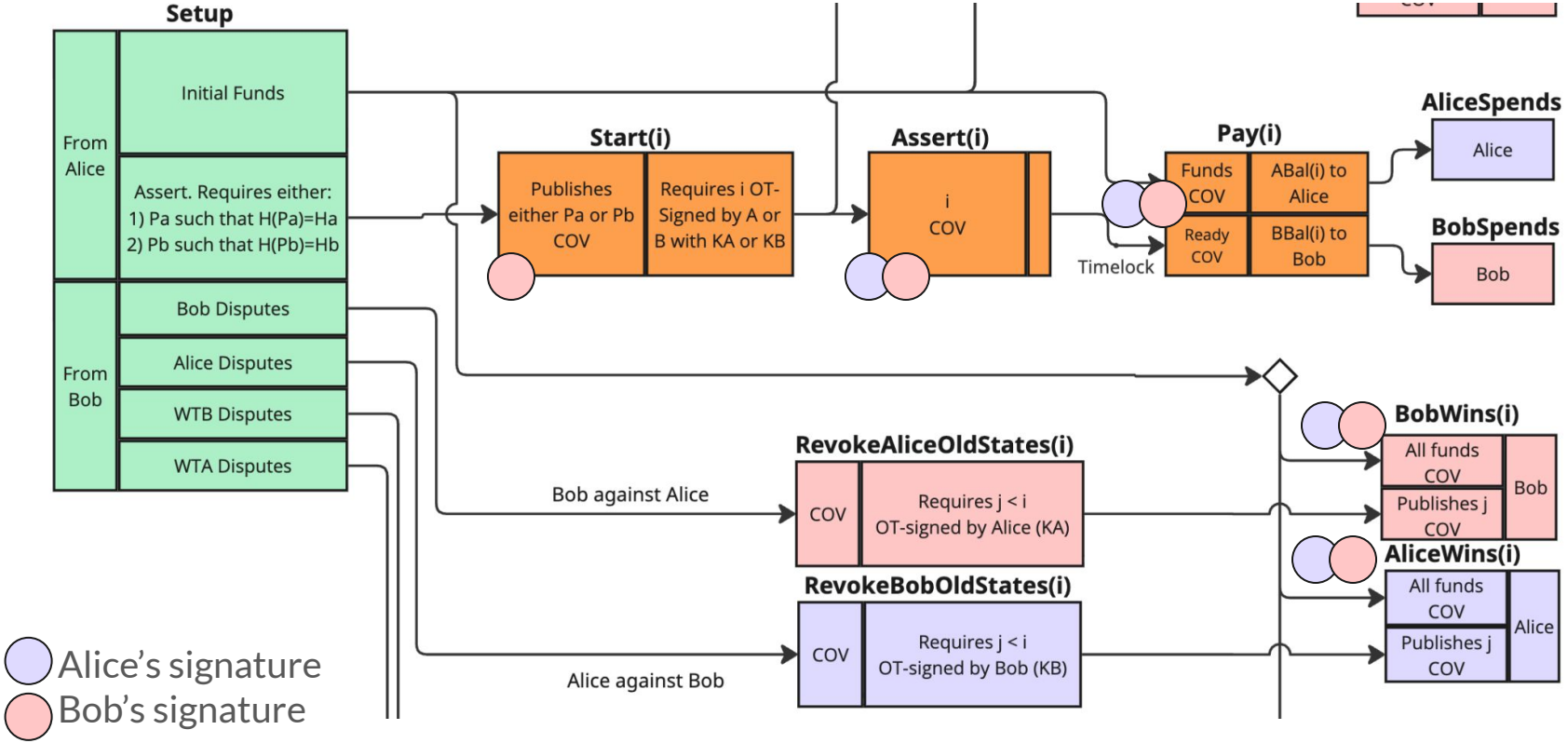
Alice's signature

Bob's signature

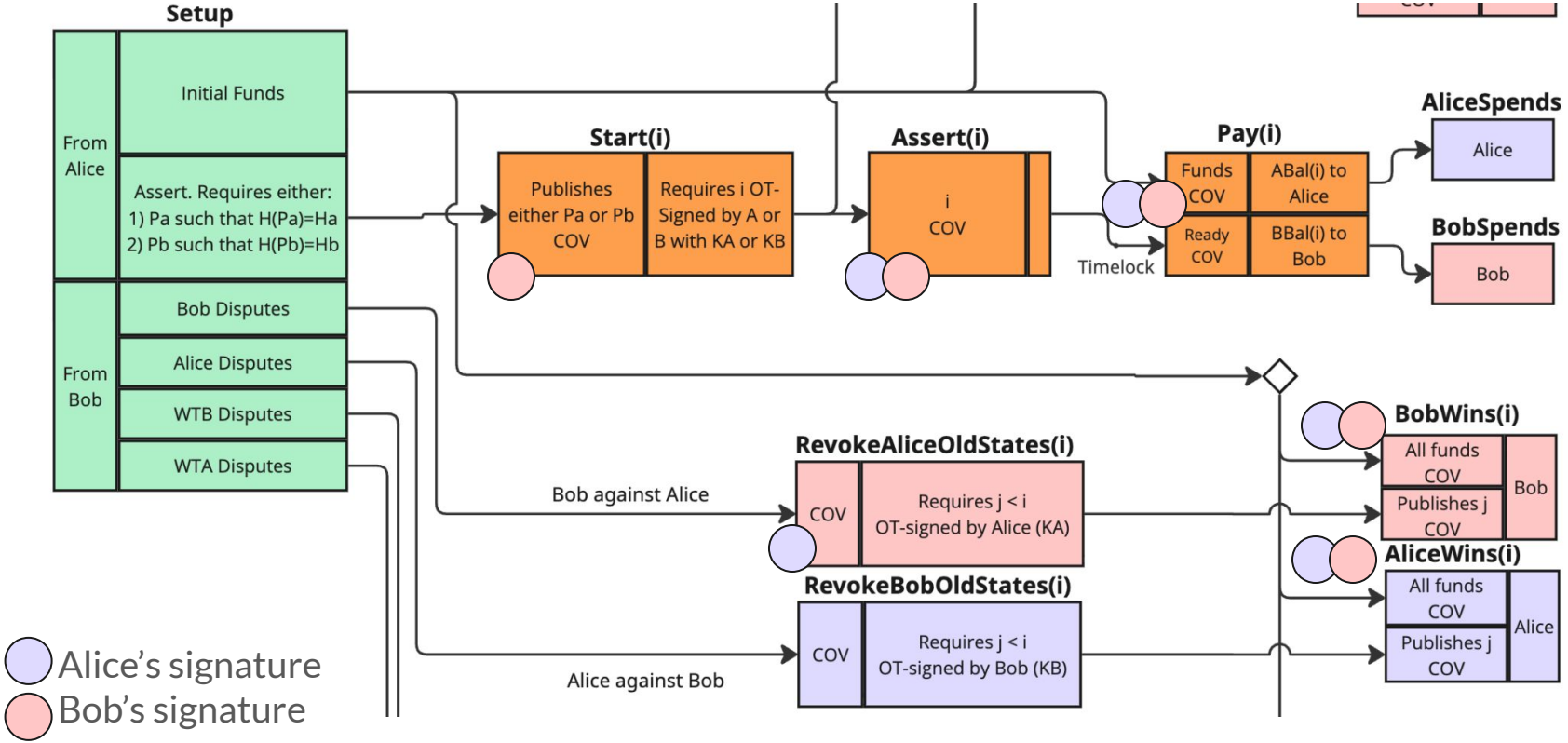
OTS-Based PC State Update (Alice pays Bob)



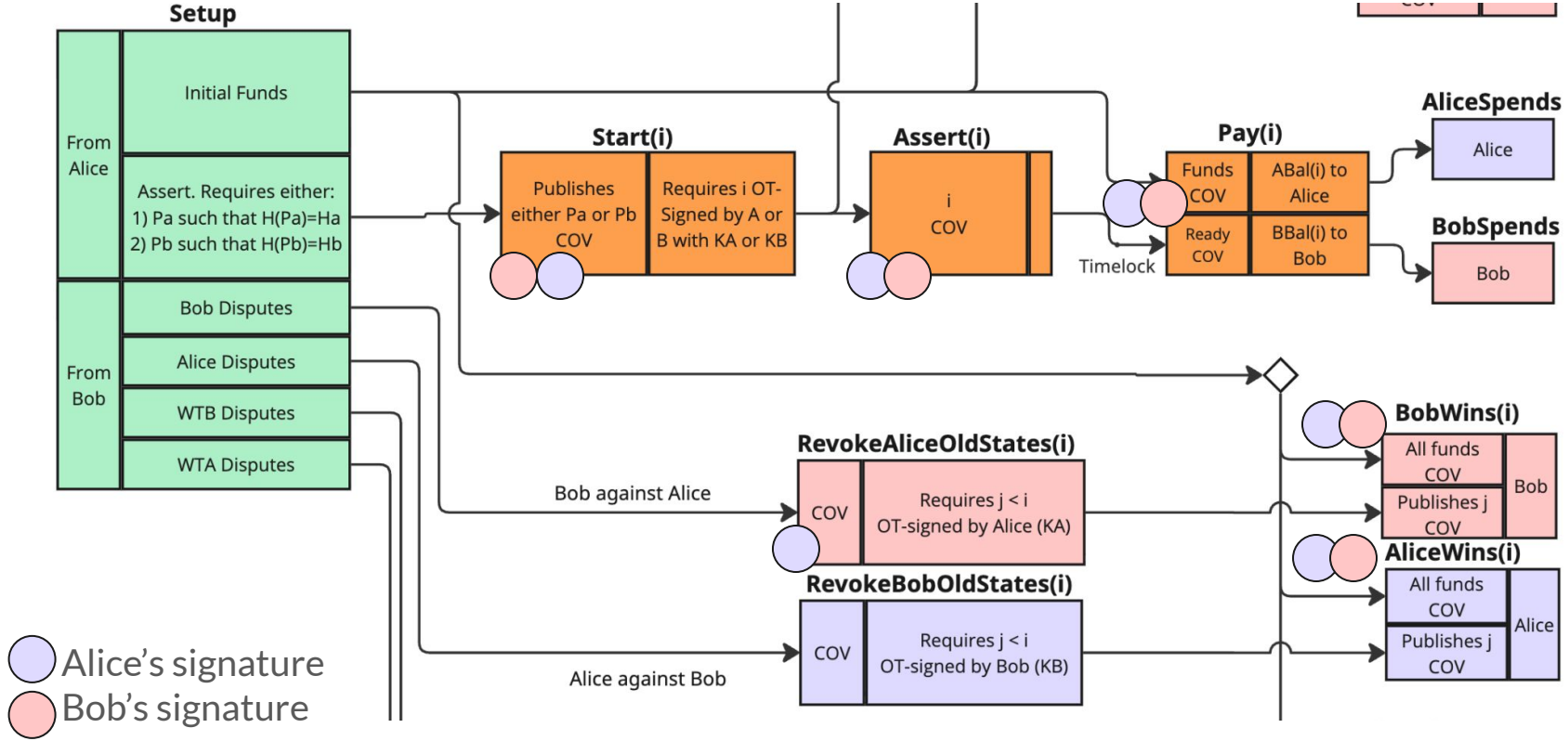
OTS-Based PC State Update (Alice pays Bob)



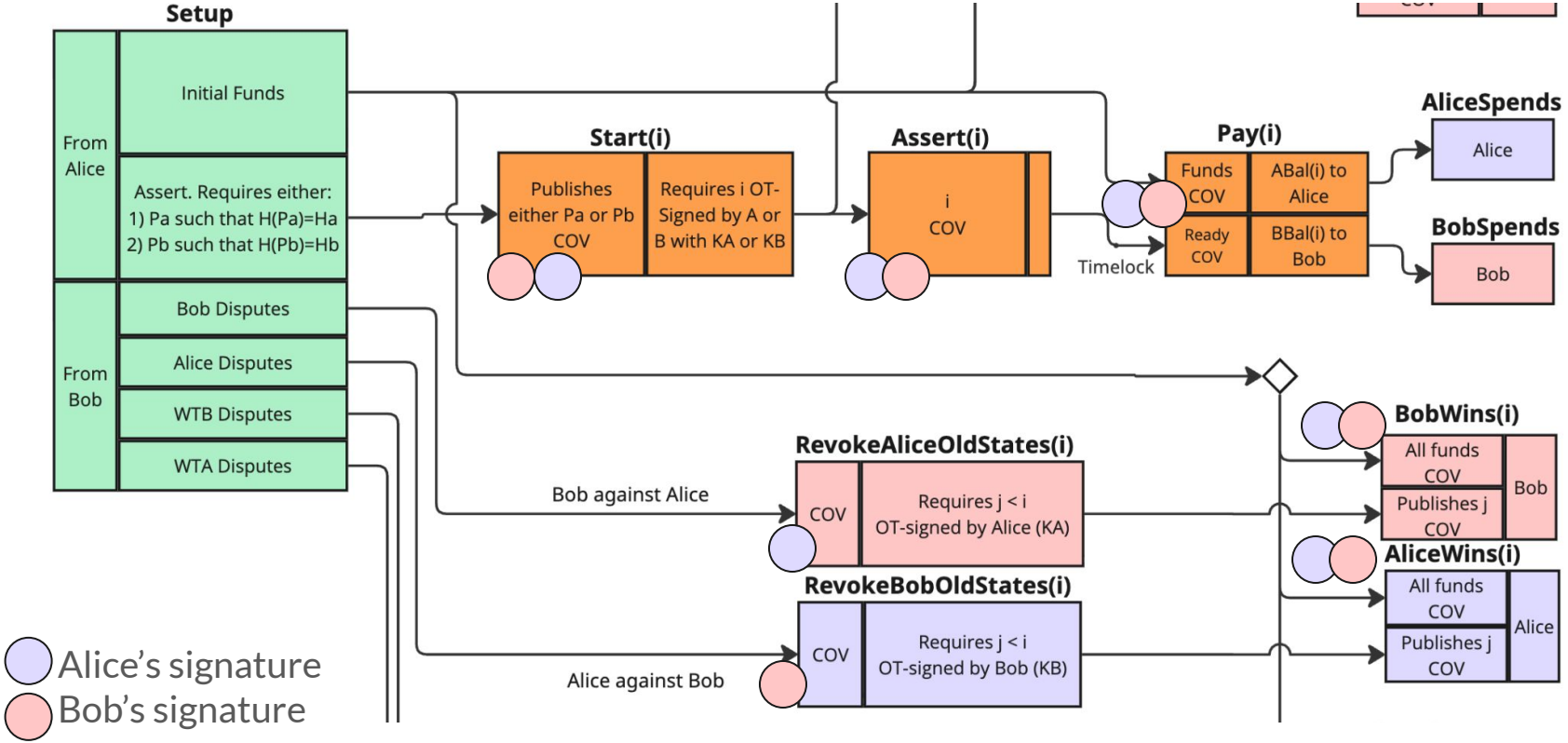
OTS-Based PC State Update (Alice pays Bob)



OTS-Based PC State Update (Alice pays Bob)



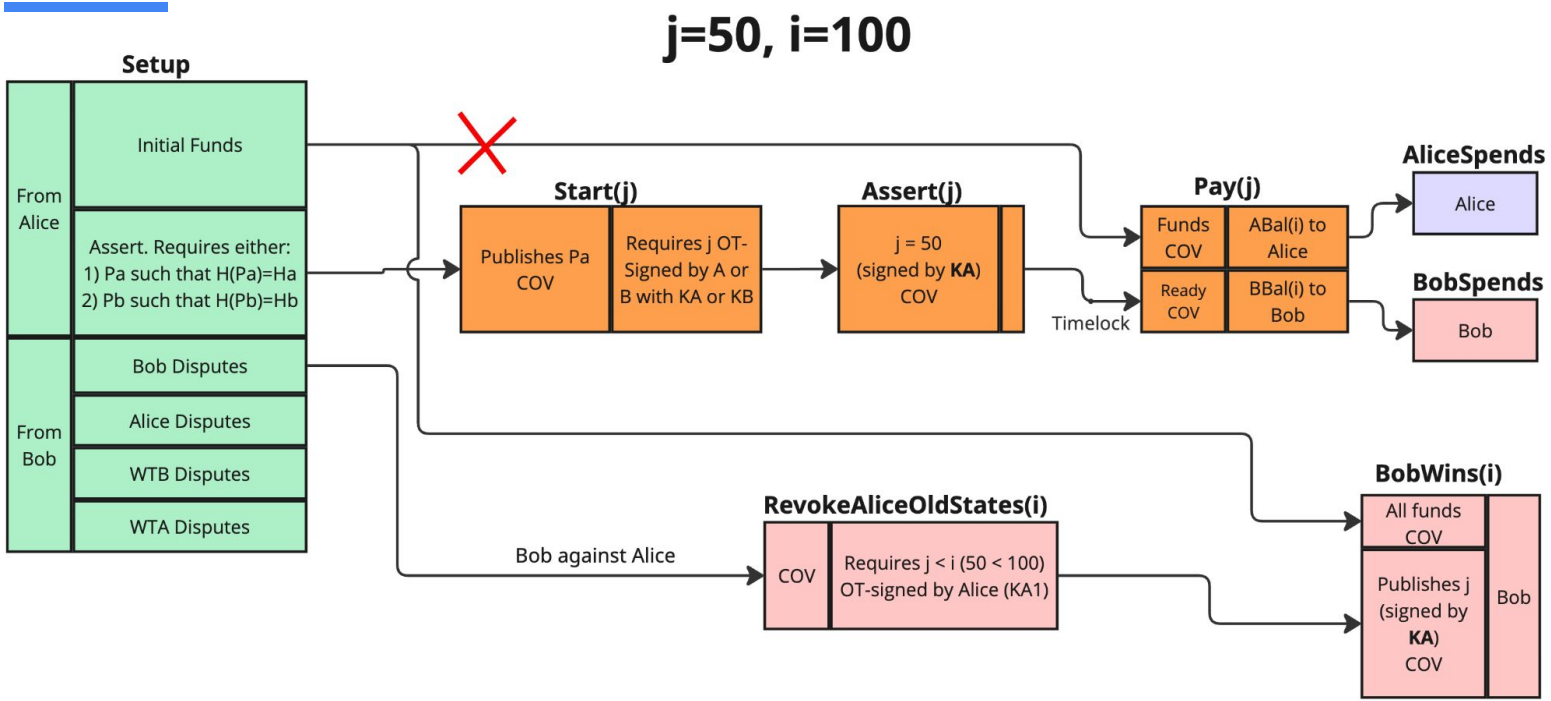
OTS-Based PC State Update (Alice pays Bob)



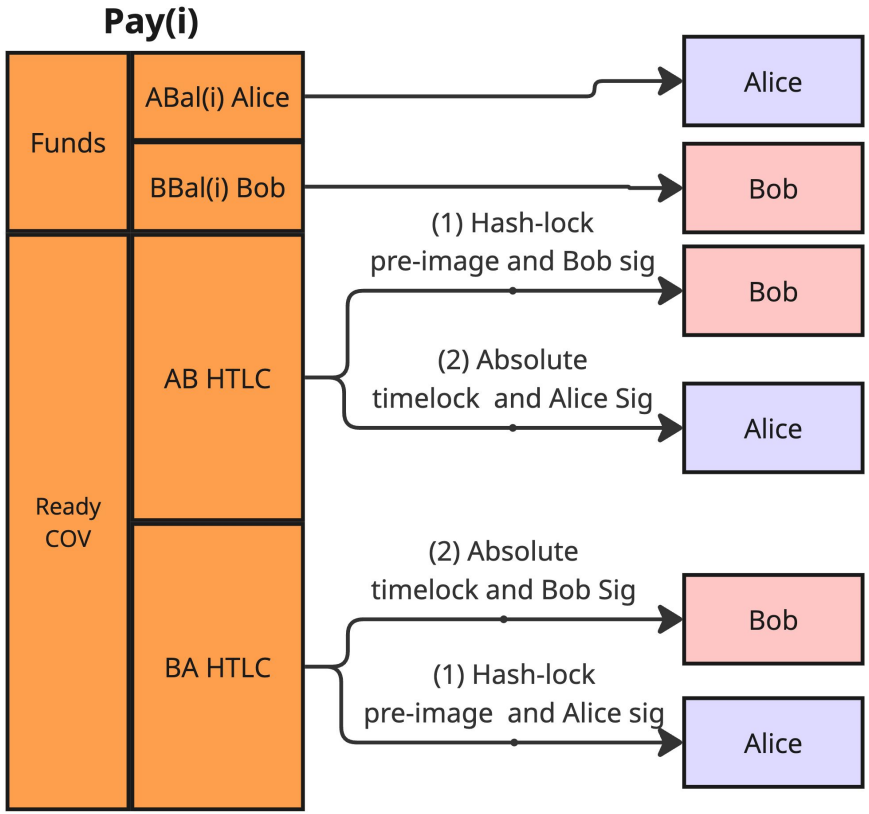
State Updates (Alice pays Bob)

1. **Alice and Bob sign** `Pay(i)` and `Assert(i)`, `AliceWins(i)`, `BobWins(i)`, `TimeoutForAliceTx(i)`, `TimeoutForBobTx(i)` exchanging signatures.
2. **Bob signs** `Start(i)` and sends it to Alice. Now Alice can continue with either state, but Bob can only use the old state. Since he received funds, Bob is motivated to proceed.
3. **Alice sign** `RevokeAliceOldStates(i)` and sends signature to Bob. Now Alice can only issue the new state, while Bob can issue only the old state.
4. **Alice co-signs** `Start(i)` and sends it to Bob. Now Alice can only issue the old state, while Bob could use either, but the new state benefits him more.
5. **Bob sign** `RevokeBobOldStates(i)` and sends signature to Alice. Now both Alice and Bob can only issue the new state.

Penalization for the Use of an Old State



HTLCs


Pay(i)
Funds
ABal(i) Alice
BBal(i) Bob
Ready COV
AB HTLC
BA HTLC
Alice
Bob
(1) Hash-lock pre-image and Bob sig
Bob
(2) Absolute timelock and Alice Sig
Alice
(2) Absolute timelock and Bob Sig
Bob
(1) Hash-lock pre-image and Alice sig
Alice

Closing the Channel

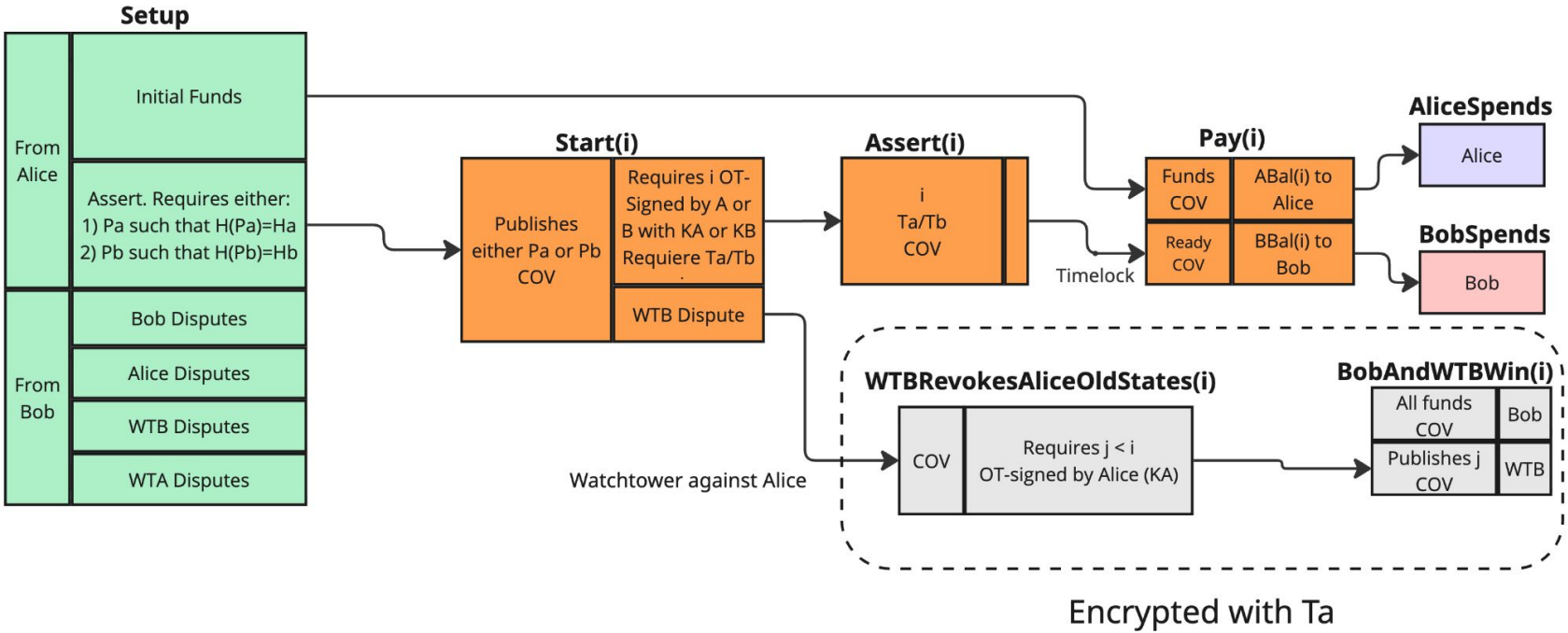
Cooperative Close: they co-sign a payment transaction that spends directly the funds from the `Setup` transaction.

Unilateral Close: (If one party becomes uncooperative)

- Publish the latest `Start(i)` and `Assert(i)`
- Wait for a potential dispute
- Publish `Pay(i)`, which is timelocked to allow dispute resolution
- Collect funds from open HTLCs, if any.

Watchtower Full Privacy

- Revoke/Win txs are encrypted with Ta (CBC) and a fresh IV each time.
- Alice can issue new fake “updates” at any rate if she doesn’t give Start ID.
- Hides: Update Rate (optionally), Channel ID and Amounts (until dispute)



Watchtower Full Privacy

- Watchtower does not learn anything (except rate of updates) until dispute arises.
- Connecting the watchtower RevokeXOldStates(i) to the Start(i) transaction allows hiding the channel an update belongs to
- Encrypting the watchtower transactions with new Ta/Tb secrets generated for each state prevents the watchtower to discover the amounts

Summary: OTS-Based Payment Channels

- New type of Payment Channel
- Constant space by Watchtowers: only needs the *latest* revocation transactions
- Storage: less than 2 Kbytes/channel to watch
- No Bitcoin soft-fork required
- Full Watchtower privacy until dispute
- Supports HTLCs

BitVMX CPU & GC

BitVMX: A VCPU for Universal Computation on Bitcoin

BitVMX is a framework designed to optimistically execute arbitrary programs on Bitcoin, leveraging the 2-party disputable computation paradigm introduced by BitVM(*).

Secure, extensible, and open-source.
BitVMX VCPU enables verifying RISC-V programs on Bitcoin.

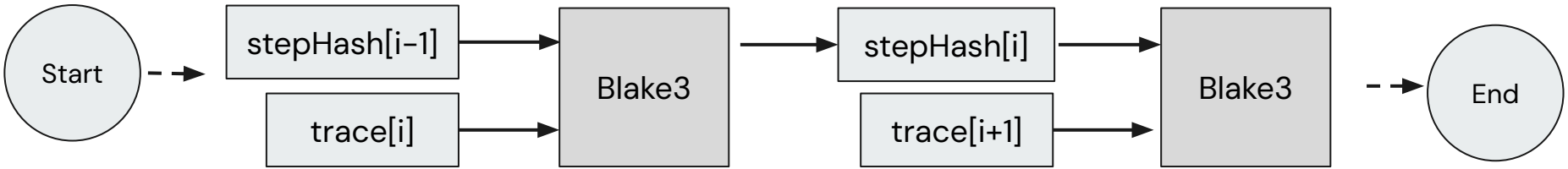
* Created by Robin Linus in 2023.

BitVMX Protocol in a Nutshell

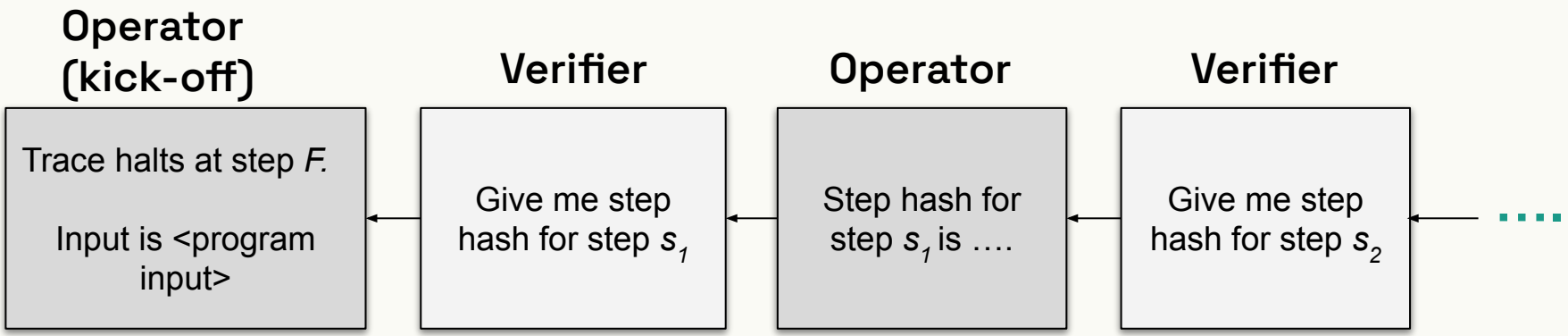
1. Prover posts the (authenticated) program input onchain
2. Verifier runs the computation locally on an VCPU, generates an execution trace.
3. The execution trace is dynamically converted into a step hash chain
4. Prover and verifiers engage in a verification game onchain, bisecting the hash chain.
5. The verifier looks for a “malfunction”: a correct step followed by an incorrect step.
6. Once it is found, the verifier requests more information about the steps
7. Challenger request proof of either: (1) hash chain step, (2) memory access, (3) opcode execution.
8. If the prover does not provide proof on time, prover loses.

BitVMX Step Hash Chain

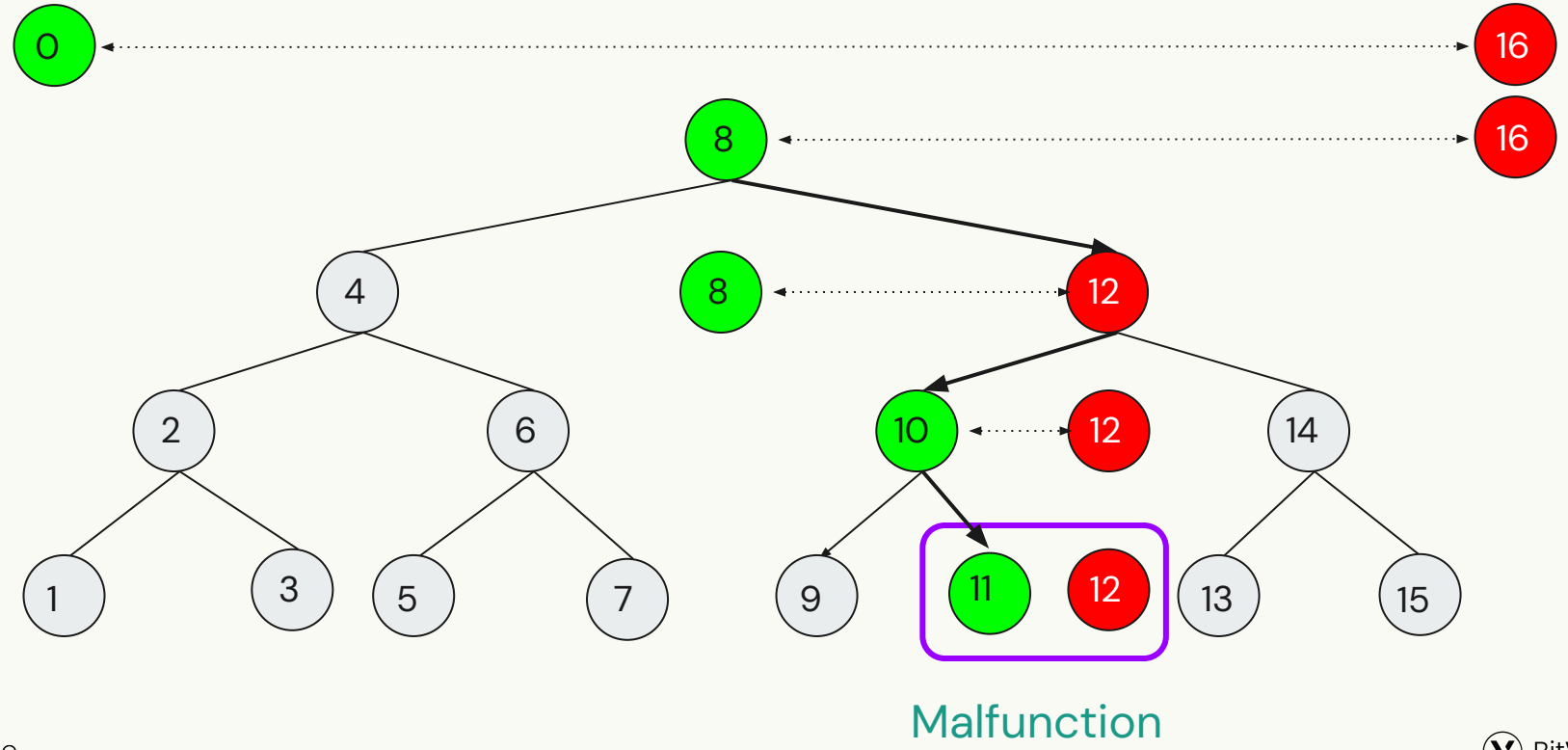
- stepHash[step] → 20 bytes
  - stepHash[Start] = 0
  - stepHash[step] = Truncate20(Blake3(stepHash[step-1] || trace[step]))



Step Hash Chain Binary Search



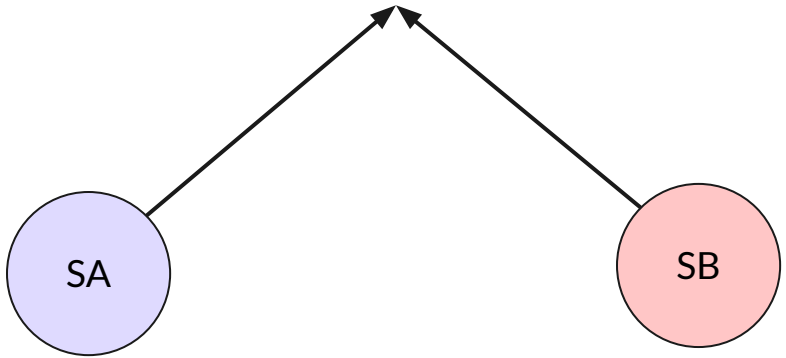
Malfunction Binary Search



BitVMX-based Payment Channels

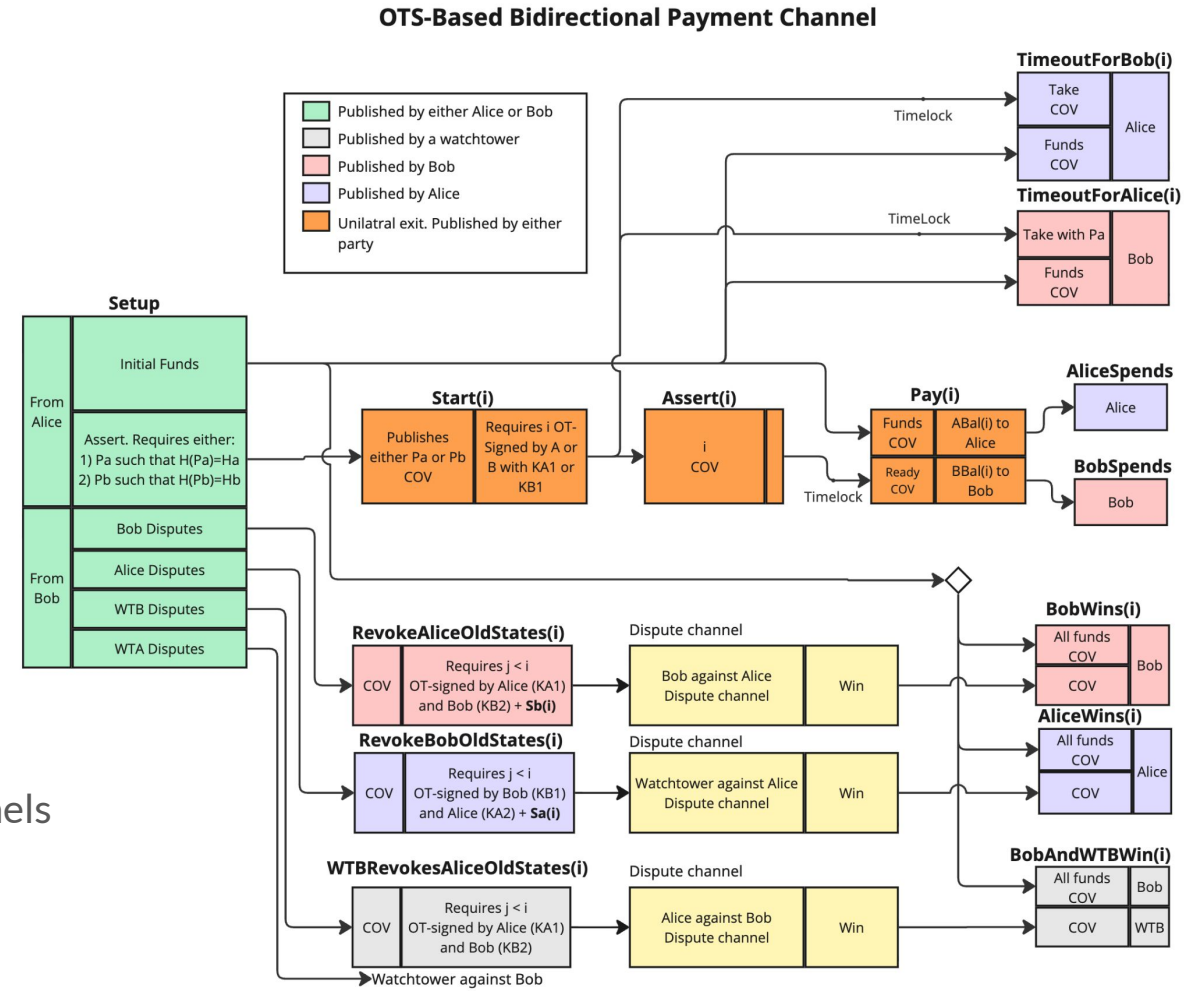
BitVMX Revocation Messages

M(i) = “We revoke all states prior i”



Schnorr signatures

BitVMX Payment Channel



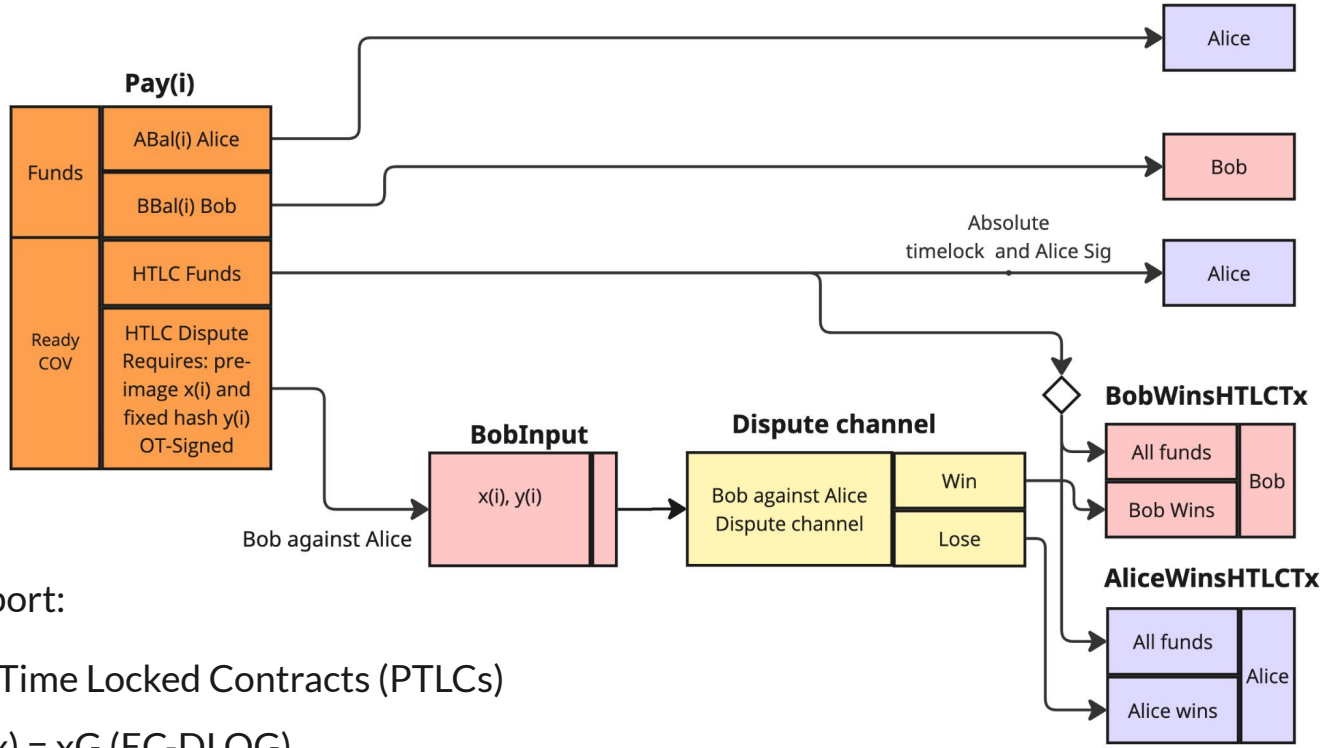
Added:

- Dispute channels
- KA2,KB2
- Sa and Sb

Fairgate

BitVMX

BitVMX HTLCs



Privacy support:

- Point Time Locked Contracts (PTLCs)
- y = H(x) = xG (EC-DLOG)
- Groth16 (ZKP)

BitVMX-Based Payment Channels

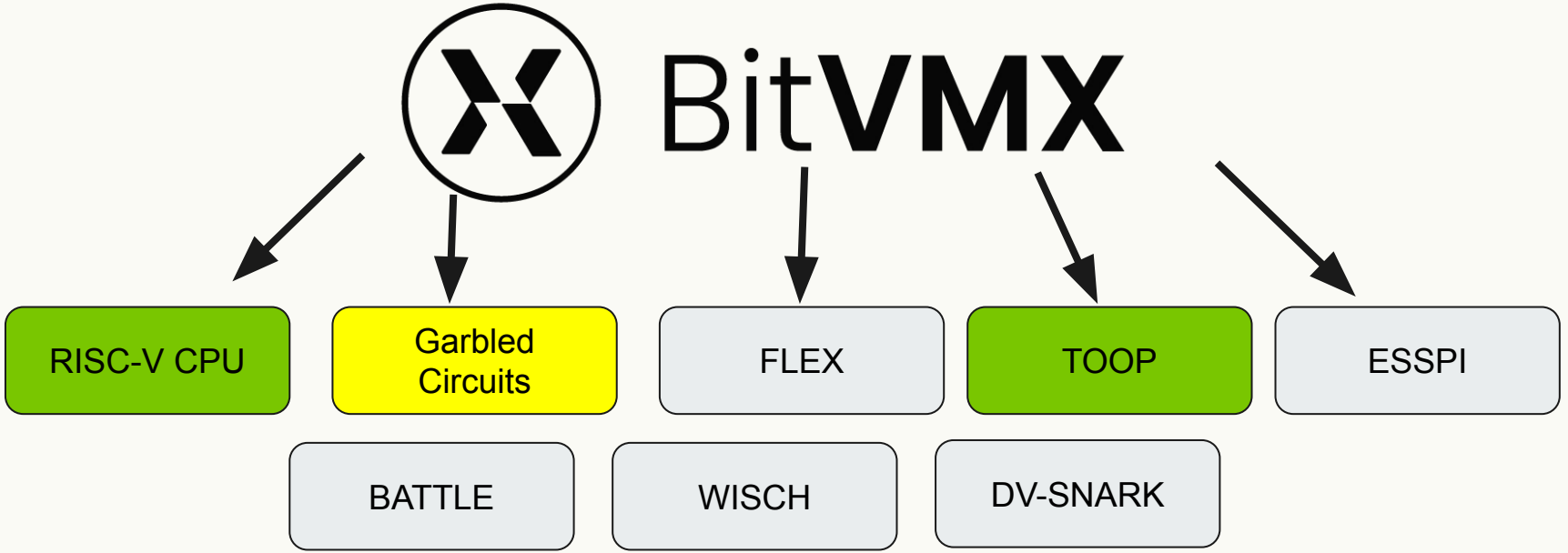
- New type of Payment Channel
- Constant space per channel for watchtowers
- No Bitcoin soft-fork required
- Supports HTLCs including hash functions that preserve route privacy

BitVMX vs BitVM2 (Schnorr Signature Verification)

Protocol	Type	Scalability	Standard Txs?	Rounds	Worst Case WU	Cost (*) USD
BitVM2 with counter-proof	Single purpose (Schnorr/SNARK)	O(√C)	No	2	~ 432 K	~108
BitVMX	Generic (RISC-V)	O(Log C)	Yes	5	~ 332 K	~83
BitVMX GC (garbled circuits)	Generic / Schnorr SNARK	O(1)	Yes	2	~67 K	~16

- 1 sats/vbyte feerate, 100K USD/BTC rate
- Dual input (proof / counter-proof)
- BitVMX 8-ary search (current protocol)
- BitVMX GC Protocol Estimation depends on future research (current estimation is 7M non-free gates)

Growing from a Protocol to a Platform



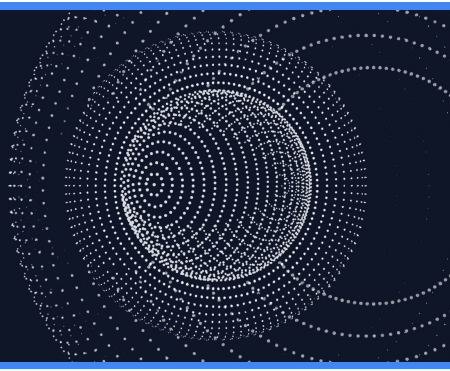
Summary

- Presented two new types of payment channel: OTS and BitVMX.
- Supports efficient and **private** watchtowers
- Supports HTLCs including Point Time Locked Contracts (PTLCs) and other privacy-preserving improvements
- Based on BitVM ideas (OTS) and BitVMX
- BitVMX implements a BitVM-style VCPU.
- BitVMX is the BitVM that works. It's cheap, robust and flexible
- Fairgate's Garbled Circuits (upcoming)
- We're just scratching the surface of GC applications on Bitcoin.

Q&A

Fairgate

www.fairgate.io



Thank You!

https://github.com/FairgateLabs

BitVMX

https://bitvmx.org

Setup

1. Each owner chooses a hidden pre-image (`Pa` by Alice and `Pb` by Bob) and sends the other the corresponding hash (`Ha = H(Pa)` and `Hb = H(Pb)`).
2. Each owner generates an OTS key pair (pair `KA1` for Alice and `KB1` for `Bob`) for signing sequence values `i`, in `Assert` using 32-bit unsigned integers to allow up to 4 billion updates. Public keys are exchanged.
3. Both parties construct a the transaction `Setup` and issues it onchain.
4. Parties create the initial transactions: `Pay(0)`, `Assert(0)`, and `Start(0)`. These define an initial exit path where each owner can withdraw their initial deposit. No revocation messages exist at this point, so no challenges are possible
5. Each owner generates another OTS key pair (`KA2` by Alice, `KB2` by Bob) for `i` values in `AliceWins`, `BobWins` transactions.
6. If watchtowers are used, they provide their OTS public keys to the owners

Open Problems under Active Research

- **Liquidity & channel balancing**
- **Routing algorithms**
- **Channel fees**
- **Payment Privacy**
- **Congestion**
- **Economic incentives & fee markets**
- **Scalability of state / channel graphs**
- **Cross-chain and interoperability**
- **Complexity in multi-party or general-purpose channels**
- **Usability / UX**
- **Watchtower efficiency**

Watchtowers

- Watchtowers receive periodic revocation messages `M(i)` for a given payment channel, identified by a **monitoring ID** (`MoId`). The `MoId` is derived from the channel's funding transaction ID and the owner's public key. If the same watchtower monitors both parties, each will have a distinct `MoId`.
- When a watchtower receives a new revocation `M(i)`, it can safely discard all prior revocations for the same `MoId`. This is a significant improvement over Lightning Network watchtowers, which must store a revocation key for every state update. Thanks to BitVMX's ability to verify complex logic off-chain, revocations become more storage-efficient and secure.

CROWN: Privacy & data exposure

- Watchtowers see “appointment blobs” (encrypted data) plus locators. These blobs are designed so the watchtower **can’t see your balances or states** except during a breach.

- But the watchtower learns **which channels you update**, the frequency of updates, and when a breach is triggered.

Sub-protocols

HTLCs Explanation

- To create an HTLC-based payment, both parties move to a new state where the payment amount is deducted from the sender's balance but not yet credited to the receiver. Instead, the amount is locked in a new output of `PayTx(i)`, controlled by a BitVMX instance. This instance is dynamically created per HTLC, but the BitVMX DAG can be mostly precomputed. The output is connected to the BitVMX program only when the HTLC is initialized.

- The sender also produces a Schnorr-signed commitment `C(i)` that includes the current sequence number `i`, the conditional payment amount, and the hash `y(i)` where `y(i) = H(x(i))`. The secret preimage `x(i)` will unlock the payment.

- To support multiple HTLCs, `C(i)` could be structured as the Merkle root of all active HTLCs.

- Assuming Alice is the sender, Bob can later reveal the preimage `x(i)` to settle the payment. Ideally, both parties cooperate to update their balances accordingly. However, if Alice refuses, Bob may initiate a dispute by revealing the signed commitment `C(j)`, the preimage `x(i)`, and showing that `H(x(i)) = y(i)`.

- If Bob is dishonest and `j < i`, Alice can respond with Bob's previously signed revocation message `Mb(i)`, invalidating Bob's claim and earning the channel funds as compensation. If she cannot provide such a revocation, Bob wins the dispute and claims the HTLC-locked funds. Crucially, the rest of the channel balance is unaffected by disputes over individual HTLCs.

Advancing to the Next State (Alice pays Bob)

1. **Alice and Bob sign** `PayTx(t)` and `AssertTx(i)`, exchanging signatures.

2. **Bob signs** `StartTx(t)` and sends it to Alice. Now Alice can continue with either state, but Bob can only use the old state. Since he received funds, Bob is motivated to proceed.

3. **Alice gives Bob** a revocation message `Ma(i)`, which is a Schnorr signature on `i`, representing: “I, Alice, revoke all states prior to `i`.” Now Alice can only use the new state.

4. **Alice co-signs** `StartTx(t)` and sends it to Bob. Alice can only use the new state; Bob could use either, but the new state benefits him more.

5. **Bob forwards** Alice’s revocation to his watchtower and issues his own revocation message `Mb(i)` to Alice. At this point, neither party can safely use the old state.

Liveness Guarantees During State Transitions

1. To prevent a party from stalling the protocol mid-update, the channel includes a timeout mechanism. If Alice issues StartTx(j) but delays or refuses to issue the corresponding AssertTx(j) in a timely manner, Bob can respond by publishing TimerForAliceTx. This transaction initiates a countdown, giving Alice a limited window to publish AssertTx(j). If Alice fails to do so before the timer expires, Bob may publish TimeoutForAliceTx, which awards him all funds in the channel as a penalty for Alice's non-cooperation.
2. The timer is triggered using the preimage Pa, which Alice reveals when she publishes StartTx(j). This ensures that only Alice can start this process, and only Bob can enforce the timeout based on her actions.

Watchtower Efficiency

- **Penalty-based LN channels (Poon–Dryja)** need “bulky” watchtowers: each time a channel state updates, the client sends the watchtower a fresh hint + encrypted-blob for that *specific* old state. Towers must keep **one blob per prior state until the channel closes** (storage grows linearly with the number of updates).

- **Eltoo-style channels** (requires ANYPREVOUT/NOINPUT) make towers **O(1)**: a tower only needs data for the *latest* state, because newer states cleanly supersede older ones.

- **Outpost / Khabbazian et al.** Tower decrypts justice transaction from unilateral exit transactions. Requires OP_RETURN payload > 80 bytes.

Proofs and Counter-Proofs