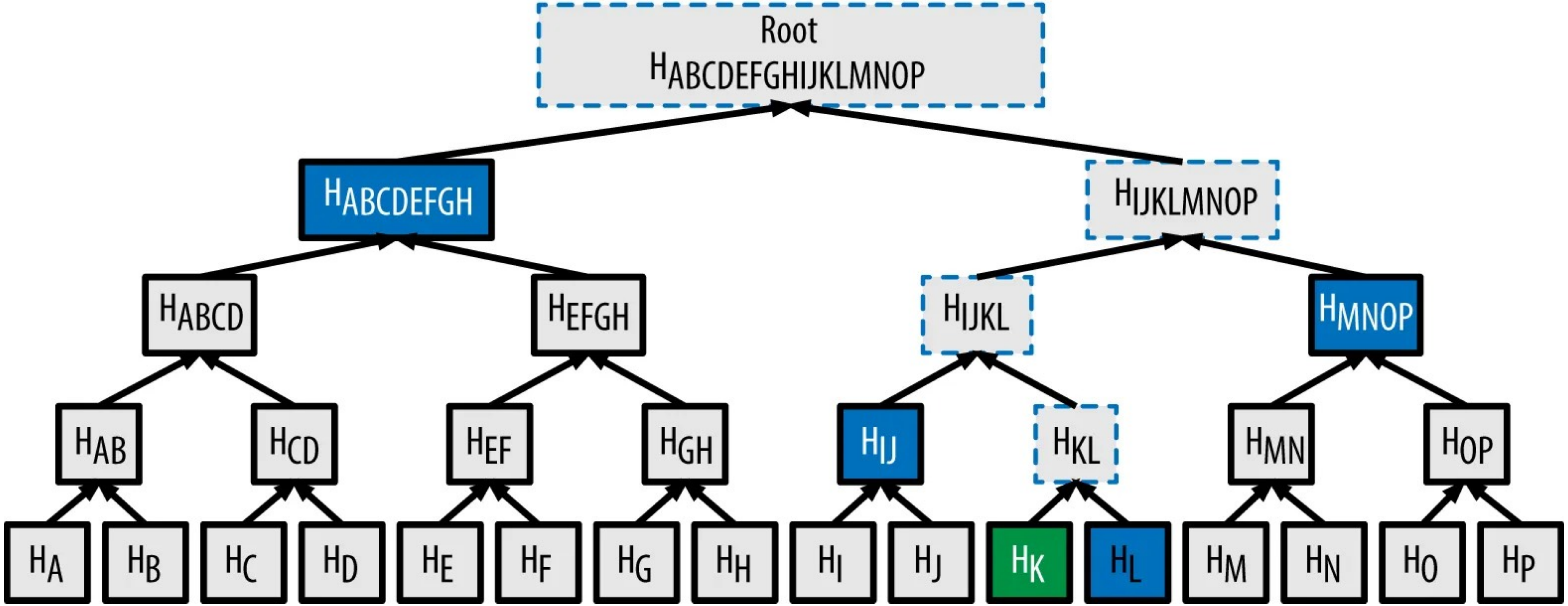
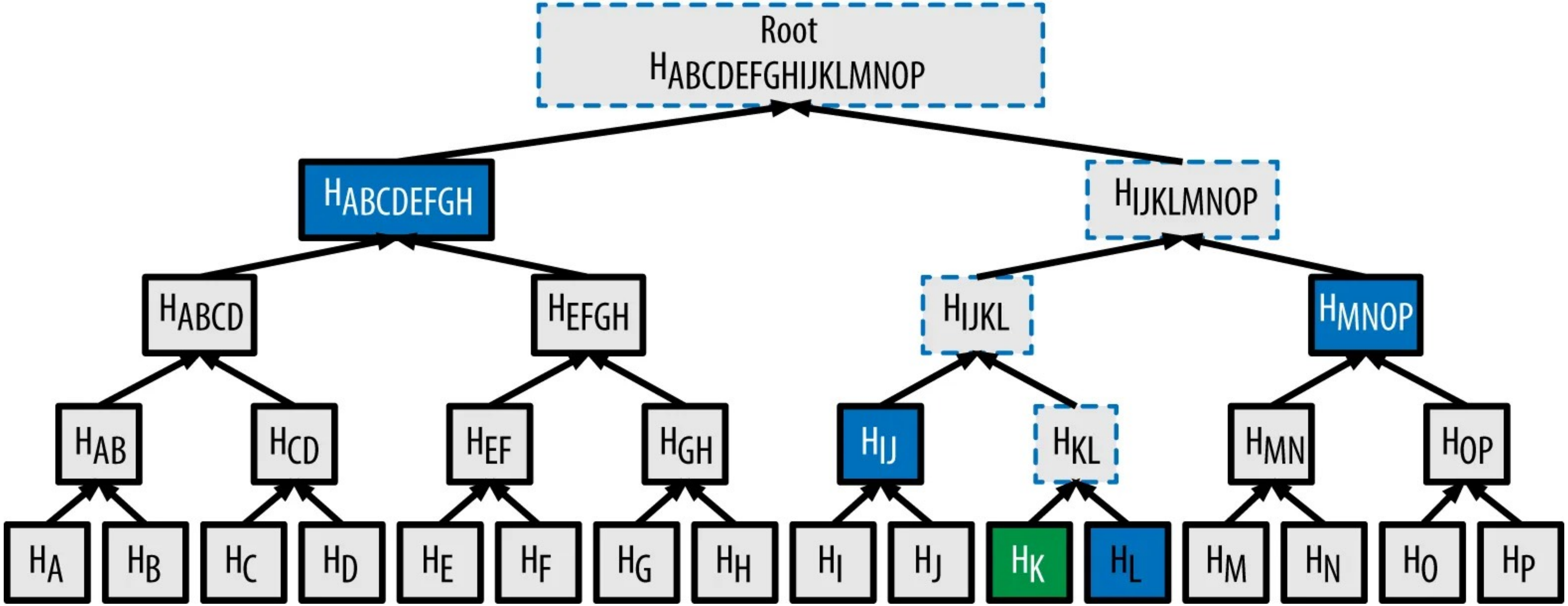
Bridging on Taiko

An intro to the built-in signal service, bridge, and vaults



Merkle Tree and Merkle Proof

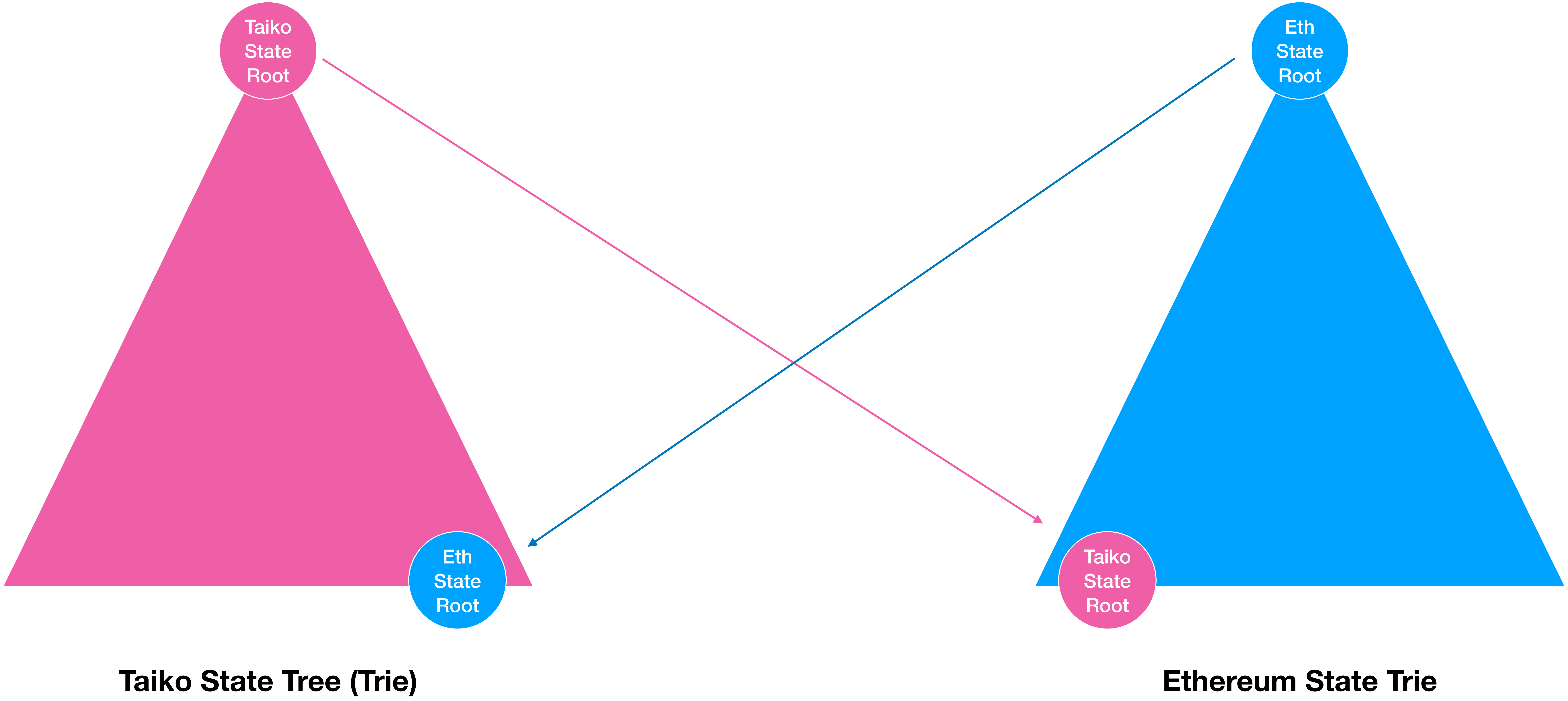


Merkle Tree and Merkle Proof

If we know the root, we can prove the inclusion of any leaf

One cross-chain solution: synchronizing Merkle roots

Root = Block Hash



Ethereum Root → Taiko (v1)

- When: a block is proposed. The latest Ethereum block hash `blockhash(block.number - 1)` is attached to the L2 block's metadata
- On L2, when the block is processed, in the block's first tx (anchor), this value is written to L2 storage.
- Later ZKP will prove the right value is the one used on L2 by the anchor tx. Using an incorrect value will invalidate this block and all future blocks.
- L1 → L2 bridging can be immediate.

```
function anchor(
    bytes32 l1Hash,
    bytes32 l1SignalRoot,
    uint64 l1Height,
    uint32 parentGasUsed
)
    external
{
    if (msg.sender != GOLDEN_TOUCH_ADDRESS) revert L2_INVALID_SENDER();

    uint256 parentHeight = block.number - 1;
    bytes32 parentHash = blockhash(parentHeight);

    (bytes32 prevPIH, bytes32 currPIH) = _calcPublicInputHash(parentHeight);

    if (publicInputHash != prevPIH) {
        revert L2_PUBLIC_INPUT_HASH_MISMATCH(publicInputHash, prevPIH);
    }

    // replace the oldest block hash with the parent's blockhash
    publicInputHash = currPIH;
    _l2Hashes[parentHeight] = parentHash;

    latestSyncedL1Height = l1Height;
    _l1VerifiedBlocks[l1Height] = VerifiedBlock(l1Hash, l1SignalRoot);

    emit CrossChainSynced(l1Height, l1Hash, l1SignalRoot);
```

[Link]

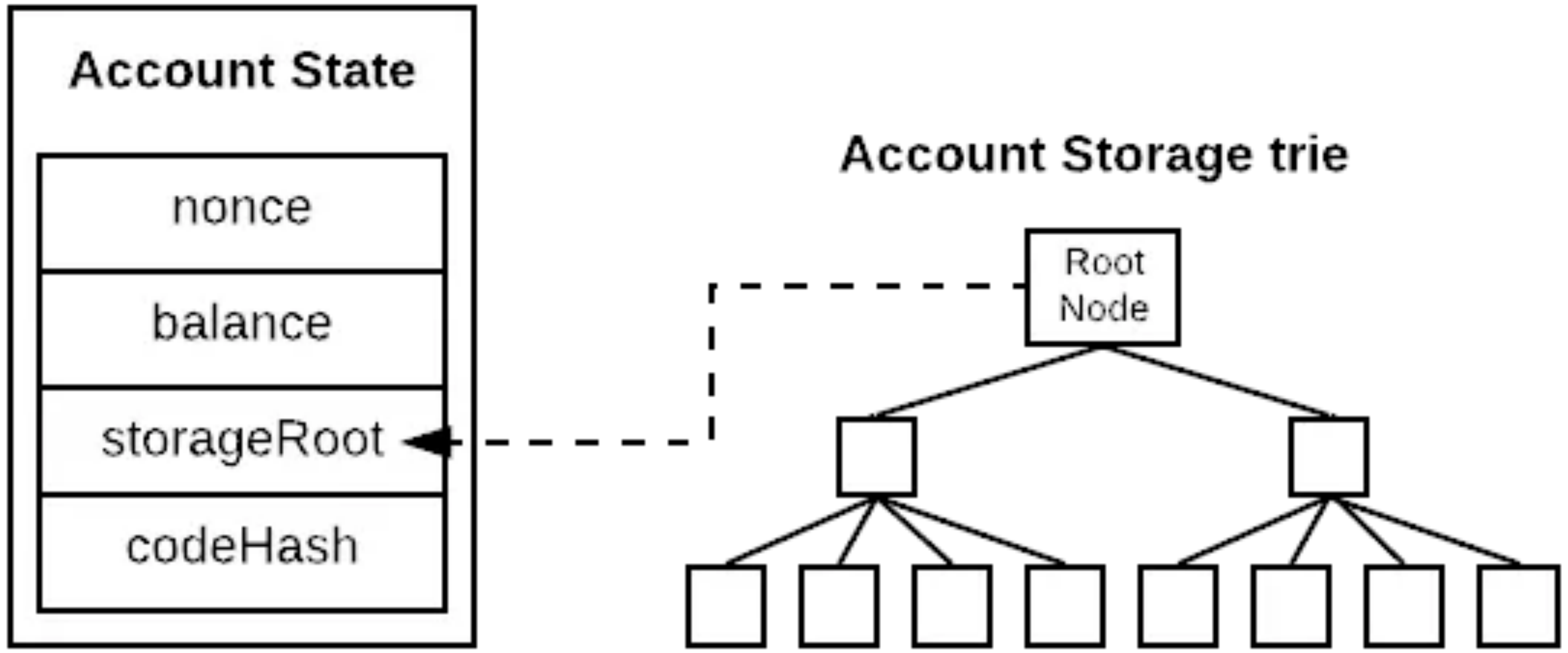
Taiko Root → Ethereum (v1)

- When: a block is verified.
- On L1, when a block is verified, the declared L2 block hash (in a fork choice) is trusted (verification happens on L1)
- L2 → L1 bridging needs to wait for ZKPs.

```
TaikoData.ForkChoice memory fc = blk.forkChoices[fcId];
if (fc.prover == address(0)) break;

uint256 proofRegularCooldown = fc.prover == address(1)
    ? config.proofOracleCooldown
    : config.proofRegularCooldown;

if (block.timestamp <= fc.provenAt + proofRegularCooldown) break;

blockHash = fc.blockHash;
gasUsed = fc.gasUsed;
signalRoot = fc.signalRoot;

_verifyBlock({
    state: state,
    config: config,
    resolver: resolver,
    blk: blk,
    fcId: fcId,
    fc: fc
});
```

[Link]

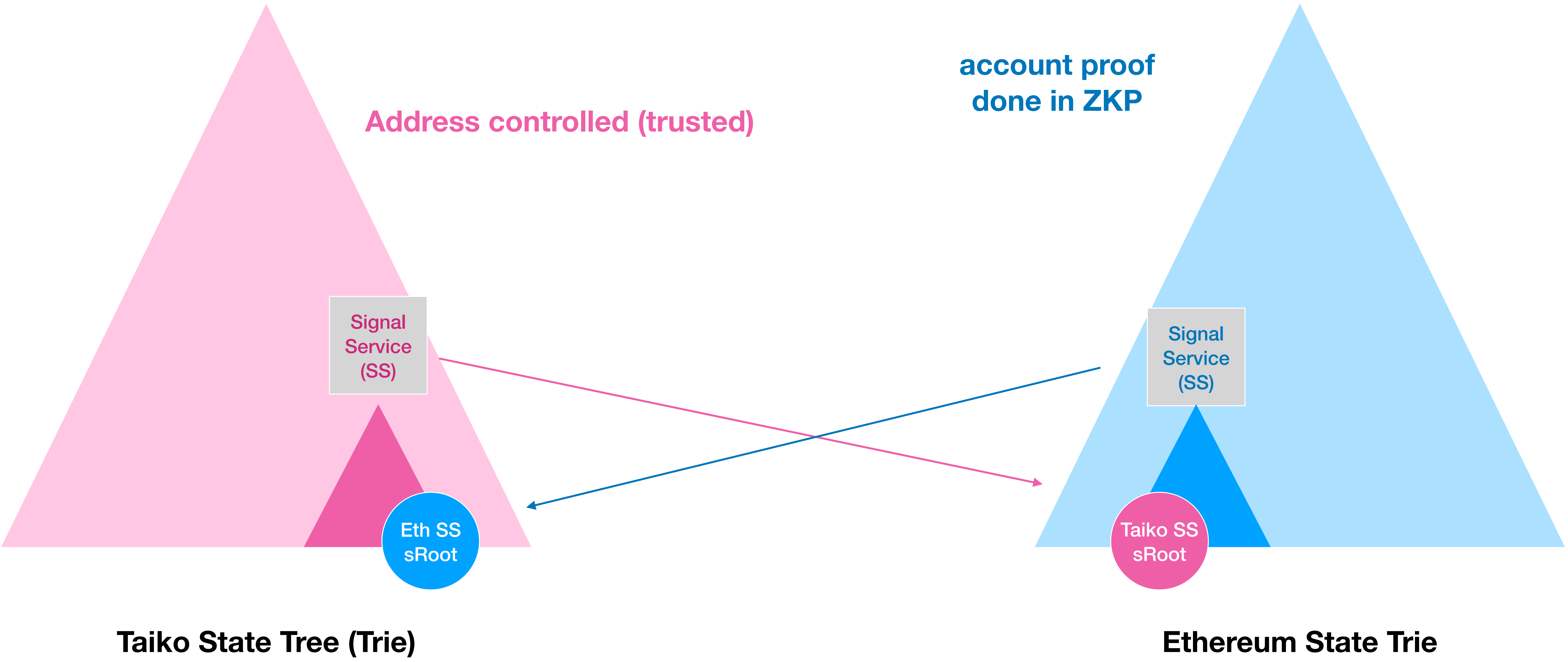
In Ethereum, two level of tree structure



A full Merkle proof has two parts:
the account proof and the account storage proof

Signal Service

- A smaller scope, not the whole tree
  A smaller merkle proof, lower cost



SignalRoot on Ethereum → Taiko (v2)

- On L2, when the block is processed, in the block's first tx (anchor), signalRoot on Ethereum is fetched and provided as a param, then written to L2 storage.
- ZKP will prove the signalRoot is correct with an account proof.
- Optimization: account proof is once per block, with v1, each cross-chain proof will have an account proof.
- Downside: apply only to storage slots in the signal service.
- Developers can still build bridges/cross-chain messaging solutions using block hash.

```
function anchor(
    bytes32 l1Hash,
    bytes32 l1SignalRoot,
    uint64 l1Height,
    uint32 parentGasUsed
)
    external
{
    if (msg.sender != GOLDEN_TOUCH_ADDRESS) revert L2_INVALID_SENDER();

    uint256 parentHeight = block.number - 1;
    bytes32 parentHash = blockhash(parentHeight);

    (bytes32 prevPIH, bytes32 currPIH) = _calcPublicInputHash(parentHeight);

    if (publicInputHash != prevPIH) {
        revert L2_PUBLIC_INPUT_HASH_MISMATCH(publicInputHash, prevPIH);
    }

    // replace the oldest block hash with the parent's blockhash
    publicInputHash = currPIH;
    _l2Hashes[parentHeight] = parentHash;

    latestSyncedL1Height = l1Height;
    _l1VerifiedBlocks[l1Height] = VerifiedBlock(l1Hash, l1SignalRoot);

    emit CrossChainSynced(l1Height, l1Hash, l1SignalRoot);
```

Link

SignalRoot on Taiko→ Ethereum (v1)

- When: a block is verified.
- On L1, when a block is verified, the declared L2 signalRoot (also in a fork choice) is trusted (verification happens on L1)
- (Same) L2’s block hash cannot be trusted without block being verified. This means L2 → L1 bridging needs to wait for ZKPs.

```
TaikoData.ForkChoice memory fc = blk.forkChoices[fcId];
if (fc.prover == address(0)) break;

uint256 proofRegularCooldown = fc.prover == address(1)
    ? config.proofOracleCooldown
    : config.proofRegularCooldown;

if (block.timestamp <= fc.provenAt + proofRegularCooldown) break;

blockHash = fc.blockHash;
gasUsed = fc.gasUsed;
signalRoot = fc.signalRoot;

_verifyBlock({
    state: state,
    config: config,
    resolver: resolver,
    blk: blk,
    fcId: fcId,
    fc: fc
});
```

Link

Sending Signals

```
function sendSignal(bytes32 signal) public returns (bytes32 storageSlot) {
    if (signal == 0) {
        revert B_ZERO_SIGNAL();
    }

    storageSlot = getSignalSlot(msg.sender, signal);
    assembly {
        sstore(storageSlot, 1)
    }
}
```

- Any address can send any non-zero bytes32 as a signal.
- slot = hash(msg.sender, signal)
- Slot value is 1.
- The same signal can be sent more than once, no side effect.
- Signal sent cannot be revoked (no delete)
- Same signal sent by different senders ends up in different slots.

Checking Signals

```
function isSignalReceived(
    uint256 srcChainId,
    address app,
    bytes32 signal,
    bytes calldata proof
)
    public
    view
    returns (bool)
{
    if (srcChainId == block.chainid) revert B_WRONG_CHAIN_ID();
    if (app == address(0)) revert B_NULL_APP_ADDR();
    if (signal == 0) revert B_ZERO_SIGNAL();

    SignalProof memory sp = abi.decode(proof, (SignalProof));

    // Resolve the TaikoL1 or TaikoL2 contract if on Ethereum or Taiko.
    bytes32 syncedSignalRoot = ICrossChainSync(resolve("taiko", false))
        .getCrossChainSignalRoot(sp.height);

    return LibSecureMerkleTrie.verifyInclusionProof(
        bytes.concat(getSignalSlot(app, signal)),
        hex"01",
        sp.proof,
        syncedSignalRoot
    );
}
```

- Given a source chain id, a sender (app), the signal, and an storage proof, return true if the signal has been sent from the source chain's signal service.
- On the dest chain, the source chain's signal service must be registered (trusted)
- True can only be returned if the signal service root from the source chain has been synchronized to this (dest) chain.

Cross-chain any message

struct AnyMessage {
...
...
}

hash(anyMessage)

```
function sendSignal(bytes32 signal) public returns (bytes32 storageSlot) {
    if (signal == 0) {
        revert B_ZERO_SIGNAL();
    }

    storageSlot = getSignalSlot(msg.sender, signal);
    assembly {
        sstore(storageSlot, 1)
    }
}
```

signalRoot sync

hash(anyMessage)

```
function isSignalReceived(
    uint256 srcChainId,
    address app,
    bytes32 signal,
    bytes calldata proof
)
    public
    view
    returns (bool)
{
    if (srcChainId == block.chainid) revert B_WRONG_CHAIN_ID();
```

"srcChainId" and "app" are provided by subscribing to source chain events.

Source chain

Dest Chain

The Bridge

PREVIOUSLY

```
struct Message {
    // Message ID.
    uint256 id;
    // Message sender address (auto filled).
    address sender;
    // Source chain ID (auto filled).
    uint256 srcChainId;
    // Destination chain ID where the `to` address lives (auto filled).
    uint256 destChainId;
    // Owner address of the bridged asset.
    address owner;
    // Destination owner address.
    address to;
    // Alternate address to send any refund. If blank, defaults to owner.
    address refundAddress;
    // Deposited Ether minus the processingFee.
    uint256 depositValue;
    // callValue to invoke on the destination chain, for ERC20 transfers.
    uint256 callValue;
    // Processing fee for the relayer. Zero if owner will process themself.
    uint256 processingFee;
    // gasLimit to invoke on the destination chain, for ERC20 transfers.
    uint256 gasLimit;
    // callData to invoke on the destination chain, for ERC20 transfers.
    bytes data;
    // Optional memo.
    string memo;
}
```

```
function sendMessage(Message calldata message)
    external
    payable
    nonReentrant
    returns (bytes32 msgHash)
{
    return LibBridgeSend.sendMessage({
        state: _state,
        resolver: AddressResolver(this),
        message: message
    });
}
```

- Get Ether from sender
- Make sure message ID is unique
- Hash the message into a (unique) signal, then send it

Source chain

```
function processMessage(
    Message calldata message,
    bytes calldata proof
)
    external
    nonReentrant
{
    return LibBridgeProcess.processMessage({
        state: _state,
        resolver: AddressResolver(this),
        message: message,
        proof: proof
    });
}
```

- Hash the message into signal and verify it has been sent using merkle proof and not processed yet
- Transfer (depositValue) Ether to users
- Call the `to` address using `data` and `callValue`
- Mark the message is processed.

Dest Chain

The Bridge

NOW

```
struct Message {
    // Message ID.
    uint256 id;
    // Message sender address (auto f
    address sender;
    // Source chain ID (auto filled).
    uint256 srcChainId;
    // Destination chain ID where the
    uint256 destChainId;
    // Owner address of the bridged a
    address owner;
    // Destination owner address.
    address to;
    // Alternate address to send any
    address refundAddress;
    // Deposited Ether minus the proc
    uint256 depositValue;
    // callValue to invoke on the des
    uint256 callValue;
    // Processing fee for the relayer
    uint256 processingFee;
    // gasLimit to invoke on the dest
    uint256 gasLimit;
    // callData to invoke on the dest
    bytes data;
    // Optional memo.
    string memo;
}
```

```
struct Message {
    // Message ID.
    uint256 id;
    // Message sender address
    address from;
    // Source chain ID (auto
    uint256 srcChainId;
    // Destination chain ID w
    uint256 destChainId;
    // User address of the br
    address user;
    // Destination address.
    address to;
    // Alternate address to s
    address refundTo;
    // value to invoke on the
    uint256 value;
    // Processing fee for the
    uint256 fee;
    // gasLimit to invoke on
    uint256 gasLimit;
    // callData to invoke on
    bytes data;
    // Optional memo.
    string memo;
}
```

```
function sendMessage(Message calldata message)
    external
    payable
    nonReentrant
    returns (bytes32 msgHash)
{
    return LibBridgeSend.sendMessage({
        state: _state,
        resolver: AddressResolver(this),
        message: message
    });
}
```

- Get Ether from sender
- Make sure message ID is unique
- Hash the message into a (unique) signal, then send it

Source chain

```
function processMessage(
    Message calldata message,
    bytes calldata proof
)
    external
    nonReentrant
{
    return LibBridgeProcess.processMessage({
        state: _state,
        resolver: AddressResolver(this),
        message: message,
        proof: proof
    });
}
```

- Hash the message into signal and verify it has been sent using merkle proof and not processed yet
- Transfer (depositValue) Ether to users
- Call the `to` address using `data` and `callValue`
- Mark the message is processed.

Dest Chain

The Bridge Context

```
function processMessage(
    Message calldata message,
    bytes calldata proof
)
    external
    nonReentrant
{
    return LibBridgeProcess.processMessage({
        state: _state,
        resolver: AddressResolver(this),
        message: message,
        proof: proof
    });
}
```

```
state.ctx = IBridge.Context({
    msgHash: msgHash,
    sender: message.sender,
    srcChainId: message.srcChainId
});
```

```
/**
 * Get the current context
 * @return Returns the current context.
 */
function context() public view returns (Context memory) {
    return _state.ctx;
}
```

- **When calling the `to` function, the bridge provides context info through a context() function so the `to` contract can perform permission checks.**

Overview


Signal Service 1
Mutually Trusted
Signal Service 2
Trust
Trust
Bridge 1
Mutually Trusted
Bridge 2
Chain 1
Chain 2

Vaults


Signal Service 1
Mutually Trusted
Signal Service 2
Trust
Trust
Bridge 1
Mutually Trusted
Bridge 2
Trust
Trust
NFT Vault 1
Mutually Trusted
NFT Vault 2
Chain 1
Chain 2

Vaults

PREVIOUSLY

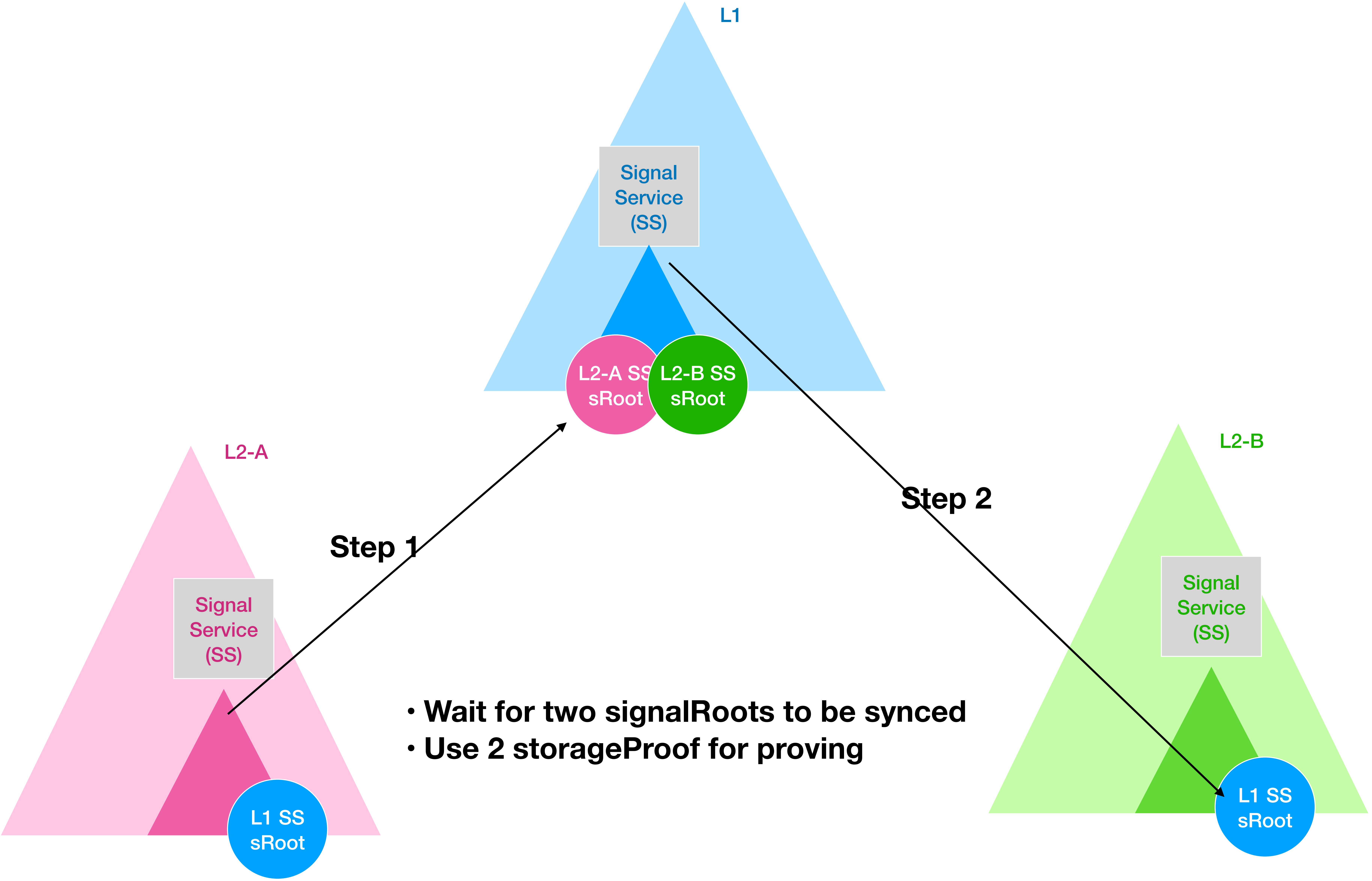
Vaults

NOW

Signal Service 1
Exchange signal Roots
Signal Service 2
Trust
Trust
Bridge 1
Bridge 2
processMessage
recallMessage() payable
Get NFT from user, construct and send the message (receiveNFT and params)
Call context() to verify message.sender is “NFT vault 1”
Call the vault receiveNFT to send the NFT to the user
receiveNFT can only be called from “Bridge 2”
Call sendNFT
NFT Vault 1
NFT Vault 2
receiveNFT is now “payable”
Chain 1
Chain 2

Vaults are application level contracts
tot part of Taiko protocol,
Developers can build and deploy other
apps to interact with the bridge.

L2 ↔ L2



- Wait for two signalRoots to be synced
- Use 2 storageProof for proving