TRAIN: A Trustless and Permissionless Intent-Based Bridging Protocol

Layerswap Labs, Inc.

March 2025

Abstract

In this paper, we introduce TRAIN (**TR**ustless **A**tomic **IN**tents), a trustless and permissionless bridging protocol designed to scale securely across an unlimited number of blockchains. TRAIN achieves this with `PreHTLC`, an advanced version of `HTLC` optimized for intent-based cross-chain bridging, featuring Local Verification and a Reward/Slash mechanism.

1 Fundamentals of Atomic Swaps

Atomic swaps rely on two core mechanisms to ensure trustless and secure asset transfers across different blockchains:

Hashlock. A `hashlock` is a cryptographic feature in a smart contract where funds are locked using the hash of a secret value. To unlock the funds, the hash preimage (the original secret value) must be revealed. This ensures that only the holder of the correct secret can unlock the funds, thereby guaranteeing secure operation.

Timelock. A `timelock` is a condition in a smart contract that specifies a fixed time frame within which a particular action must be completed. For example, in an atomic swap, if the party meant to claim the funds fails to do so within the allotted time, the contract treats the action as canceled and returns the locked assets to their original owner. This prevents assets from becoming indefinitely locked and provides a failsafe in case one party becomes unresponsive.

2 Classic Atomic Swaps with HTLC

Hashed timelock Contracts (`HTLC`s) enable peer-to-peer asset exchanges between different blockchains without direct interoperability. The general process is as follows:

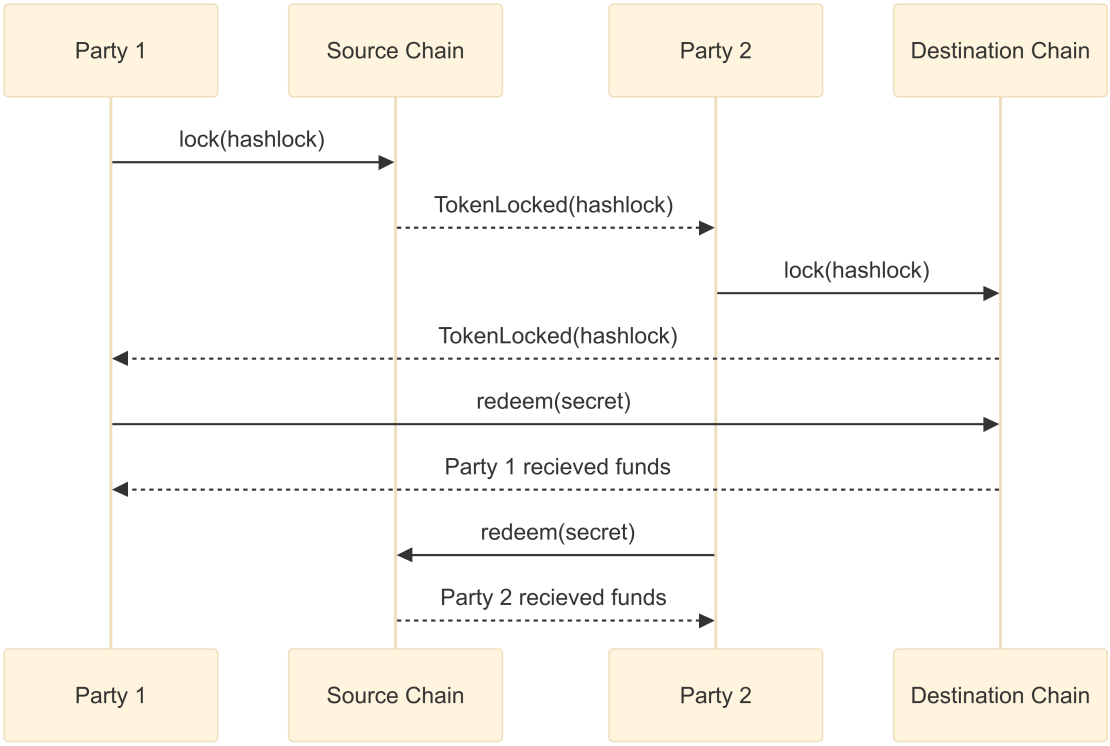

Figure 1: HTLC Sequence Flow

1. **Party 1 Lock**:
   - Party 1 generates a secret `S`.
   - Computes the `hashlock` = HASH(S).
   - Creates an `HTLC` with a `timelock` to lock funds for Party 2 on the source chain.
2. **Party 2 Lock**:
   - Party 2 detects Party 1's `HTLC`.
   - Party 2 locks funds with `timelock` for Party 1 on the destination chain using the same `hashlock`.
3. **Party 1 Unlock**:
   - Party 1 detects Party 2's `HTLC`.
   - Party 1 claims funds on the destination chain by revealing `S`.
4. **Party 2 Unlock**:
   - Party 2 detects `S` on-chain.
   - Uses `S` to claim funds on the source chain.

Despite the inherent security of `HTLC`s, they face three main practical limitations:

- **Secret Management**: Party 1 must secure and manage the secret `S` until Party 2 commits.
- **Claim Transaction Fees**: Party 1 may not have funds on the destination chain to pay transaction fees for claiming assets.
- **Liveness Issues**: If Party 2 fails to act, the swap stalls; no other entity can step in to complete the transaction.

3 Atomic Swaps with PreHTLC

To address these limitations, we propose `PreHTLC`, which extends the traditional `HTLC` model. In the `PreHTLC` model, we assume the exchange takes place between the `User` and the `Solver`. The `User` submits their intent with `PreHTLC` (without a `hashlock`), and the `Solver` is responsible for managing the secret and releasing the funds.

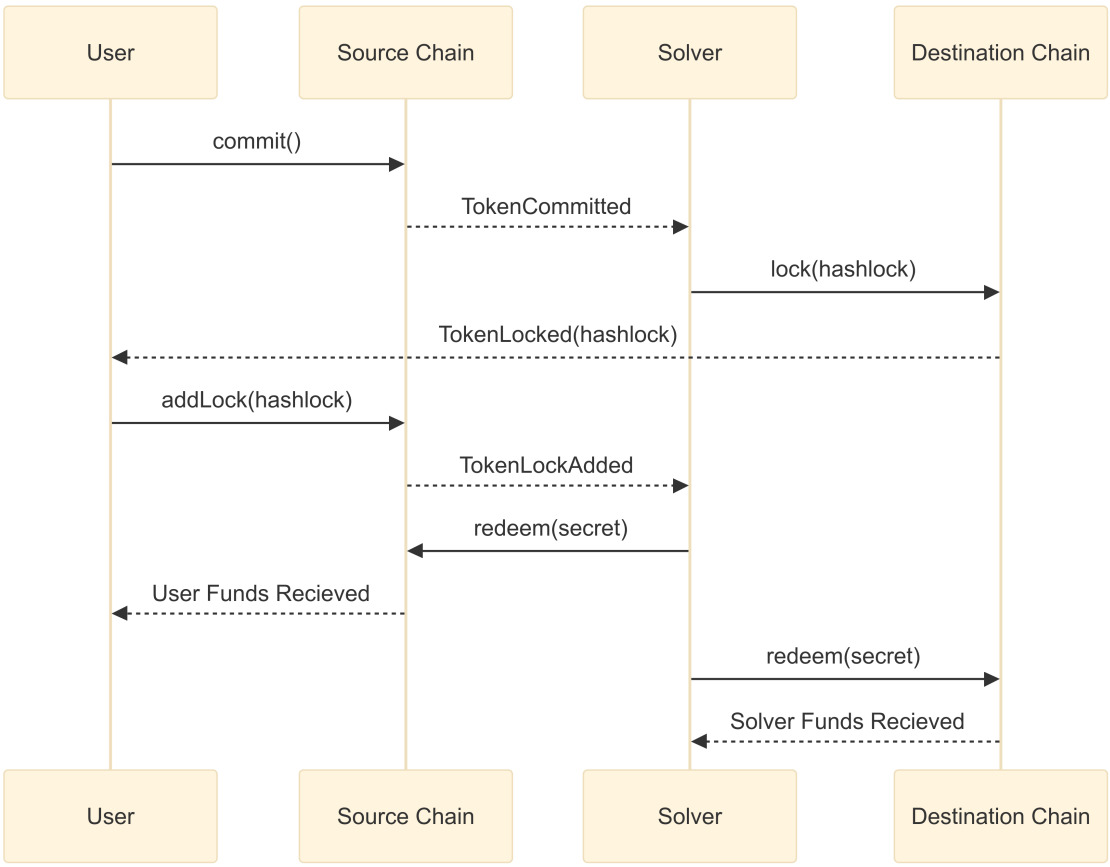

Figure 2: PreHTLC Sequence Flow

1. **User Commit**:
   - The `User` creates a `PreHTLC`, committing funds to a chosen `Solver` (or a set of `Solvers`) on the source chain.
   - This commitment has no `hashlock`, only a `timelock`.

2. **Solver Lock**:
    - The `Solver` detects this `PreHTLC`.
    - The `Solver` generates a random secret `S`.
    - Computes the `hashlock` = HASH(S).
    - Locks funds (minus the `Solver` fee) for the `User` on the destination chain.
3. **User AddLock**:
    - The `User` observes the `Solver`'s transaction on the destination chain.
    - Retrieves the `hashlock`.
    - Converts their `PreHTLC` into a standard `HTLC` (using the `hashlock`) on the source chain and confirms the final `Solver` if multiple `Solver`s were specified during the commitment.
4. **Unlocks**:
    - The `Solver` releases the `User`'s funds on the destination chain by revealing `S`.
    - The `Solver` then claims its funds on the source chain using `S`.

The Liveness Issue. `PreHTLC`s significantly reduce liveness issues by:

- Delegating secret management to the `Solver` makes the entire transaction flow fully recoverable from on-chain data for the `User`.
- Allowing `User`s to commit to multiple `Solver`s instead of a single party that may become inactive, enabling any `Solver` to step in if needed.

However, the `User` must still be online long enough to transmit the `hashlock` from the destination network back to the source network.

Delegating Secret Management. Delegating secret management to the `Solver` does not introduce additional trust assumptions. When the `Solver` reveals the secret to claim funds on the source network, the secret becomes public. As a result, any party can use it to unlock the `User`'s funds on the destination network. To incentivize the release of `User` funds, the protocol reserves a portion of the total bridging fee as a reward for performing this task.

3.1 Local Verification

During Step 3, the dApp must verify the `Solver`'s transaction on the destination chain. This verification can be done in two ways:

- **Light Client (e.g., Helios)**: If available, the dApp directly validates the transaction via the light client.
- **Multiple RPC Endpoints**: If a light client is unavailable, the dApp queries multiple RPC providers or a user-specified node. After confirming the state's integrity, it retrieves the `hashlock`.

Security Considerations

Security relies on retrieving the correct `hashlock` from the destination chain. The worst-case scenario occurs when all of the following are true simultaneously:

- No light client is available for the destination chain.
- Only a single RPC provider is accessible.
- This single RPC provider is operated by the same `Solver`.

Wallets, dApps, and users have the freedom to verify the `hashlock` however they choose. The protocol does not enforce any specific verification method. In the worst-case scenario, where a user has to manually verify the state, the trustless nature of the exchange is still preserved. If the user is not convinced, they do not have to confirm the swap—they can simply cancel it.

3.2 Reward/Slash Mechanism

In `PreHTLC`-based atomic swaps, the `Solver` is responsible for handling the `User`'s claim transaction. However, a lazy `Solver` might neglect to release the `User`'s funds even after unlocking their assets. To prevent this:

- The `Solver` locks an additional **reward amount** when committing funds to the `User`.
- If the `Solver` successfully completes the `User`'s transfer, that reward is returned to the `Solver`.
- If the `Solver` fails to act, any party (including the `User`) can step in, release the `User`'s funds, and claim the `Solver`'s reward as compensation.

This approach creates a strong economic incentive for `Solver`s to remain diligent and active.

3.3 Multi-hop Transactions

TRAIN allows the same `hashlock` to be reused across multiple networks and `Solver`s within a single bridging operation. Once `S` is revealed, all linked `HTLC`s in all networks can be unlocked.

Example The `User` wants to move assets from Chain A to Chain C, but there is no single `Solver` covering both chains. Two `Solver`s are available: `Solver(AB)` (Chain A–B) and `Solver(BC)` (Chain B–C).

1. **User Commit**: `User` locks assets in a `PreHTLC` with `Solver(AB)` on Chain A.
2. **Solver(AB) Commit**: `Solver(AB)` locks assets in a `PreHTLC` for `Solver(BC)` on Chain B.
3. **Solver(BC) Lock**: `Solver(BC)` locks funds for the `User` on Chain C (standard `HTLC`).
4. **User, Solver(AB) AddLock**:
    - The `User` detects the final lock on Chain C, retrieves the `hashlock`, and upgrades the `PreHTLC` on Chain A to a full `HTLC`.

- `Solver(AB)` upgrades the `PreHTLC` on Chain B to an `HTLC`.

5. **Unlocks**:
   - `Solver(BC)` reveals `S` on Chain C to release funds to the `User`.
   - `Solver(BC)` then claims its funds on Chain B.
   - `Solver(AB)` claims its funds on Chain A.

This procedure generalizes to any number of hops. Regardless of the hop count, the `User` needs only two source-chain transactions. In most cases, two `Solver`s will be sufficient, as popular networks (e.g., Ethereum) can act as an intermediate hub.

4 Security

TRAIN enforces strict security properties to ensure trustless, permissionless, and verifiable cross-chain transfers:

- **Homogeneous Security**: Cryptographic security remains uniform for all permissionless participants, even as new networks or `Solver`s join.
- **Battle-Tested Foundation**: TRAIN's core (atomic swaps with `HTLC`) has been successfully deployed in the Lightning Network and other production environments.
- **No Centralized Pools**: The protocol avoids large contract-held asset pools, significantly reducing the incentives for attacks.
- **Immutable Contracts**: Contracts are immutable, mitigating upgrade-related risks and preserving stability over time.
- **Simplicity**: `PreHTLC` contracts are streamlined (typically around 200 lines of code), reducing potential vulnerabilities.

5 Scalability

TRAIN's permissionless design allows any blockchain to be added without centralized coordination. The typical steps:

1. **Contract Implementation**: Implement `PreHTLC` standards on the target network.
2. **Solver Agent Implementation**: If no existing implementation is available (e.g., a generic EVM library), create a Solver Agent Chain Library for the new network to interact with `PreHTLC` contracts.
3. **Run a Solver**: Operate a `Solver` that bridges the new chain with at least one widely connected chain.
4. **dApp Implementation**: If needed, develop a Client Chain Library that manages wallet interactions and event subscriptions.

5. **dApp Deployment**: Add the new network to an existing popular bridge UI, or fork/build your own.

Once these steps are completed, users can transfer assets to and from the newly added network without gatekeepers.

6 Conclusion

By integrating an intent/`Solver`-based framework with `PreHTLC`s and local verification, the TRAIN protocol achieves:

- **Near-Immediate Settlement**: Approximately 30-second completion time for cross-chain transfers, approximated as $2 \times \text{src\_block\_time} + 2 \times \text{dst\_block\_time}$.
- **Full Control**: Users maintain total custody of their assets; neither they nor the `Solver`s rely on external actors.
- **Seamless Onboarding**: Any new blockchain network can be integrated permissionlessly, without requiring approval from any party.

This model provides a highly scalable, trustless framework for cross-chain asset transfers, eliminating the need for third-party security mechanisms.

References

1. Alt chains and atomic transfers, *Bitcoin Forum*. Available at: https://bitcointalk.org/index.php?topic=193281.0
2. Hashed Timelock Contracts, *Bitcoin Wiki*. Available at: https://bitcoinwiki.org/wiki/hashed-timelock-contracts
3. PreHTLC Atomic Swaps, *EF L2 Interop Repo*. Available at: https://github.com/ethereum/L2-interop/blob/main/docs/intents/atomic-swaps.md
4. PreHTLC Contracts Implementation, *TRAIN Repo*. Available at: https://github.com/TrainProtocol/contracts
5. Solver Implementation, *TRAIN Repo*. Available at: https://github.com/TrainProtocol/solver
6. dApp Implementation, *TRAIN Repo*. Available at: https://github.com/TrainProtocol/app