EIL under the hood - the gory details

This post focuses on EIL's "happy" and "unhappy" flows, the mechanism design and how it mitigates various attacks and adverse situations. It also elaborates on advanced composability extensions enabled by EIL.

For high level overview, vision and rationale, see EIL: Trust minimized cross-L2 interop

Definitions

Multichain Transaction - a single operation involving transactions on multiple chains, perceived by the user as a single operation.

Multichain Account - a smart account that can validate, execute, and pay for transactions across multiple chains under one signature, forming the foundation of account-based interoperability.

Account-Based Interoperability - EIL's new model for cross-rollup interaction where the user's own account initiates and settles every transaction directly, removing reliance on trusted bridges, solvers, or relayers.

CrossChainPaymaster - an ERC-4337 paymaster contract that enables trustless crosschain gas payments and offers crosschain liquidity. Combined with a Multichain Account, it abstracts gas payments and liquidity, enabling users to send Multichain Transactions on chains where they hold no assets.

Voucher - a signed message that represents a claim for gas and/or liquidity for a specific account on a specific chain. Vouchers are redeemed via CrosschainPaymaster as part of a Multichain Transaction.

Dynamic Voucher - a voucher whose value depends on transaction output not known at the time of signing, allowing users to sign once even when amounts aren't known upfront.

Voucher Note - An optional bytes32 field attached to a voucher that encodes verified metadata or instructions. It allows protocols to build composable, trust-minimized flows (i.e. intent-like operations) on top of EIL.

Cross-Chain Liquidity Provider (XLP) - An onchain participant that signs vouchers without knowing the intent or what the funds are used for. It doesn't interact directly with users, nor transact on their behalf.

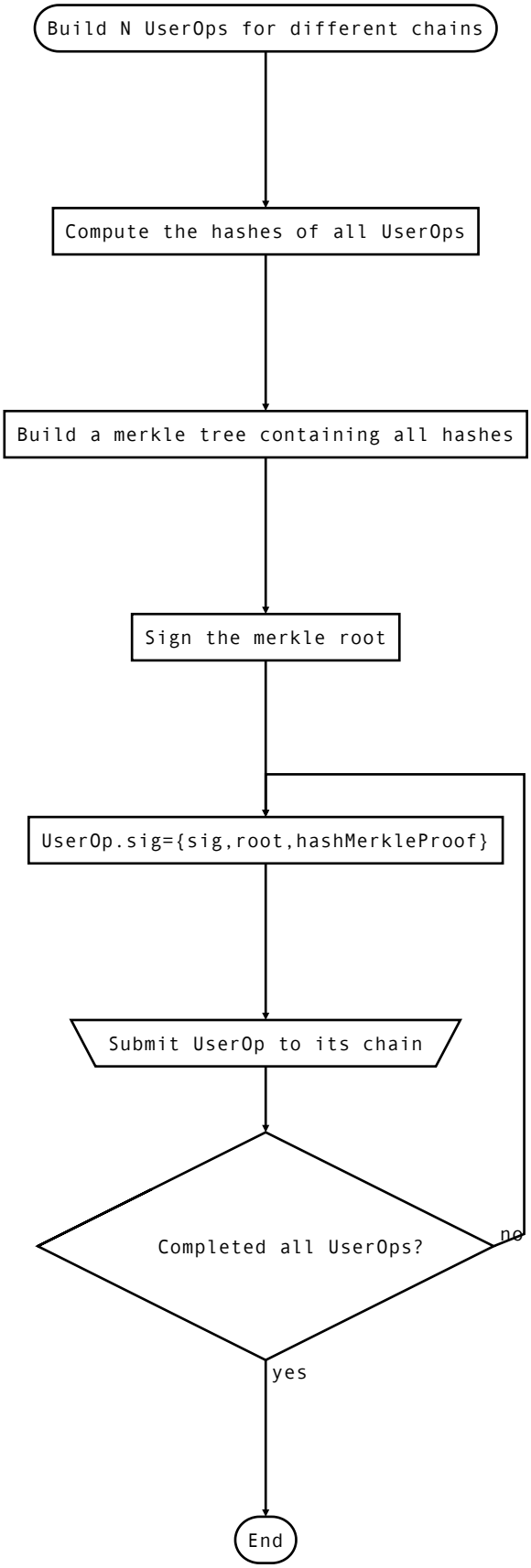
How do multichain ERC-4337 accounts work?

There are different ways to implement a multichain ERC-4337 account. One way which offers high flexibility, is to implement a validation function that validates a set of UserOps instead of a single one. The wallet can generate multiple different UserOps, then sign a single authorization that applies to all of them. The authorization can be a merkle tree containing the hashes of any number of UserOps with any number of chain IDs.

How the account works:

- In `validateUserOp`, instead of checking that `UserOp.signature` signs over `userOpHash`, check if this field contains a signed merkle root and a merkle proof that `userOpHash` is in that tree.
- Other parts of the account remain unaware of its multichain nature. Each chain sees and executes a different UserOp, authorized by the same signature.
- No need to synchronize nonces, the UserOp of each chain uses the current nonce of that account on that chain.
- On chains where the account is used for the first time, `UserOp.initCode` is provided for deployment. On others it is left blank.

How the wallet uses it:



- Build the UserOps for all the chains. For each UserOp:
  - Use the correct chain ID and the account's current nonce on that chain.
  - If the account does not exist on that chain, set UserOp.initCode to deploy it.
  - Set UserOp.callData to the actions intended to run on that chain.
  - Set the paymaster and its data.
  - Estimate and set the gas values.
  - Hash the UserOp.

- Pack all UserOp hashes in a merkle tree
- Sign its root. One signature covers the entire set of calls on all chains.
- For each UserOp
  - Set `UserOp.signature` to the merkle root and its signature
  - Append a merkle proof for the hash of this UserOp
  - Submit the UserOp to its appropriate chain

Solving token transfers without fast cross-L2 messaging

With account abstraction enabling seamless multichain calls, the remaining challenge is value transfer. We need this, both for transfers and to enable paymaster logic for crosschain gas payment.

Unlike calls, token transfers do require information flow between L2s. Currently we can't trustlessly implement fast messaging so the next best thing is Atomic Swaps.

The most widely known method is HTLC. Each party locks funds in a timelock contract on one of the chains, which can be withdrawn by the other party using a secret which is hashed in the contract. Both contracts use the same hash so as soon as the party which generated the secret performs a withdrawal, the other party sees the secret and can perform the withdrawal on the other chain. If the secret is not revealed within the preset time, each party can withdraw their original deposit. Funds can't be lost or stolen but the protocol is inefficient. It requires 1:1 relationship, multiple transactions on both chains, and potentially locks up funds for some time.

Can we do better? Yes! We have a tool at our disposal that the typical HTLC implementation doesn't: We don't have fast cheap trustless messaging but we do have slow expensive messaging via L1.

This enables optimistic design, removing the need for a secret and reducing the number of transactions to 2 on the source chain (one by each party) and 0 on the destination chain. The withdrawal doesn't require a dedicated transaction, and happens within the user's call where the funds are used. Transfers can be as fast as 1 source chain block + 1 destination chain block, 2 seconds on many current rollups.

CrossChainPaymaster

CrossChainPaymaster is an ERC-4337 paymaster for crosschain gas payments as well as a permissionless liquidity hub for ETH and ERC-20 tokens. It is designed to work with a multichain ERC-4337 account as described above.

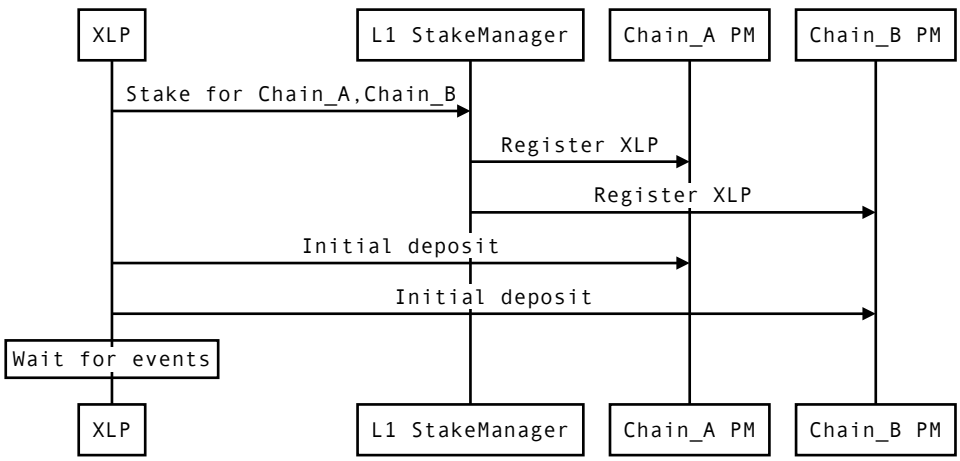
Its properties include:

- All interactions happen onchain. No offchain relationship between users and liquidity providers.
- User funds never lost or stolen.
- Liquidity Provider funds never lost or stolen.
- User controls every transaction. Liquidity provider never transacts on the user's behalf.
- Low latency - as low as a single L2 block.
- Efficient:
  - Liquidity provider transacts only on the source chain.
  - User withdrawal happens as part of the transaction that uses the funds.
- Economic incentives prevent holdups even in the presence of malicious Liquidity Providers.
- Mempool compatible.
- No dependency on future protocol changes.

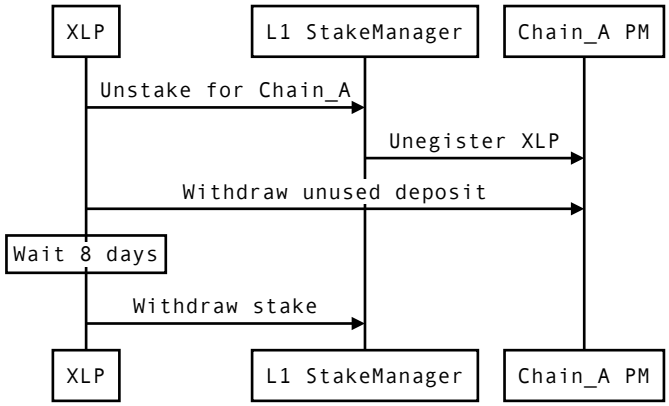
How it works

- XLPs (Crosschain Liquidity Providers) register and deposit funds in CrossChainPaymaster on multiple chains. In addition they lock a stake on L1 in L1CrossChainStakeManager. The unstake delay is 8 days - longer than the max L2 finality time. If an XLP starts the 8 days unstaking process, it immediately gets unregistered.

Registering & Staking:



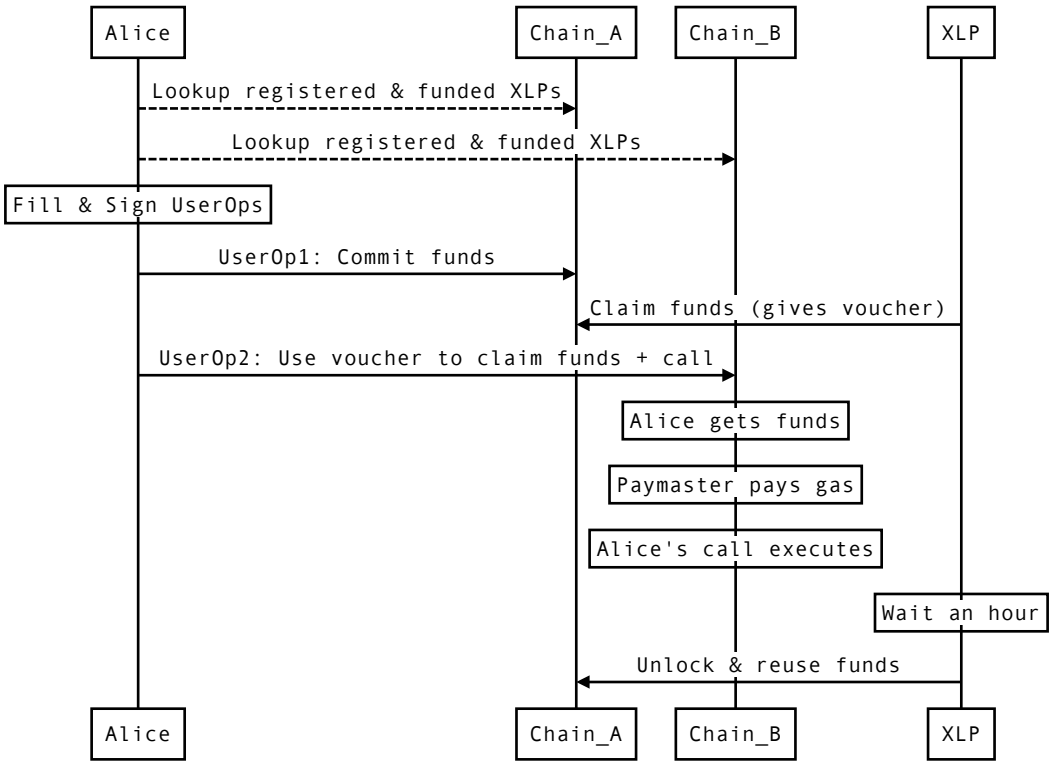
Unregistering & unstaking:



The structure and use of the stake is discussed below.

Transacting:

- Alice wants to transact from Chain_A to Chain_B. She finds registered XLPs that operate on both chains
- Alice signs a multichain UserOp. On Chain_A she locks funds in CrossChainPaymaster and requests a matching Chain_B voucher, specifying a list of XLPs she's willing to use and a fee schedule (detailed below). The request is short-lived. If a voucher is not provided promptly, Alice's funds are unlocked.
- An XLP claims her Chain_A funds by providing a signed voucher - a signed commitment for Chain_B. The same signed voucher that claims funds on Chain_A releases XLP funds on chain B to Alice - forming an atomic swap. The funds on Chain_A remain locked for an hour (to mitigate rugpull attempts - more details in the following section), after which they're credited to XLP's deposit.
- Alice appends the XLP's voucher to her Chain_B UserOp's signature and submits it to Chain_B.
- Chain_B CrossChainPaymaster verifies voucher, checks that XLP has sufficient funds deposited, pays for the gas and gives Alice the funds.
- Alice's Chain_B call gets executed and her account uses the funds during this call. Gas is paid out of XLP's Chain_B balance.



- This flow can continue and traverse any number of L2s using the same signature.
- Each iteration transfers value and performs one or more calls.
- It can also perform a completion call on Chain_A if needed.
- Calls are executed on all chains with one signature. Gas was paid on the source chain.

What could go wrong and what do we do about it? See Attacks & mitigations.

Voucher fee structure

Voucher requests offer a fee to compensate XLPs. Each request may include one or more assets, e.g. ETH for gas, ERC-20 tokens. The fee is denominated in the first asset, whether it's ETH or a token.

Requests specify multiple XLPs that may claim them. The first XLP to provide a voucher receives the fee. This creates competition between XLPs.

Fee discovery uses a reverse Dutch auction. The request specifies a fee range and a fee increase per second.

- Time T+0: UserOp is still in the mempool. Any listed XLP may provide a voucher and receive the start fee as soon as both the request and the voucher get included onchain.
- Time T+1: If no XLP provided a voucher at T+0, the request may land onchain unfulfilled. A higher fee is paid to the XLP that provides a voucher.
- The fee keeps increasing every second at the user-specified rate until a voucher is provided or the max fee is reached.
- If the request remains unclaimed, it expires and the funds are released back to the user.

Alice may start with a very low fee and let it increase until an XLP considers it sufficient. However, to minimize latency she should start close to the current market fee. If she offers the current market fee or higher, she can expect zero-latency fulfillment. Current market fees can be observed onchain as the paymaster emits an event for each voucher.

This mechanism is an optimization over how gas fees work. The start fee is equivalent to the current gas price which can be gathered from onchain data. To avoid signing a new transaction if the price increased, a range is specified. The reverse dutch auction uses the competition between XLPs to ensure that users transact at the lowest possible fee.

Mempool dynamics

An XLP that also runs a bundler and participates in the mempool can bundle the UserOp along with its own UserOp that claims the funds, thus earning the fee before other XLPs get a chance to act. Voucher requests become part of MEV.

In a competitive environment XLPs that don't do this will usually be 1 block too late and seldom earn fees.

It is therefore expected that most XLPs would join the mempool and compete on fulfilling requests in the same block where users submit them. Users benefit from 1 block crosschain swaps.

Advanced composability extensions

Beyond the basics - multichain calls and liquidity, the protocol includes features that enable more complex composability use cases.

Dynamic vouchers and return values across chains

Dynamic vouchers solve the problem of unknown outputs in multichain flows by letting users sign once even when return values aren’t known upfront.

A multichain transaction may have to send the execution result from one chain as an input on another. One such example is the multichain DEX swap described in the overview post. In this flow Alice doesn't know in advance how much RUT she's going to get on Taiko, and therefore cannot hardcode the amount that will be transferred back to Arbitrum. Her account on Taiko will perform the swap and request a voucher for sending the received RUT to Arbitrum, and her final UserOp on Arbitrum will make use of these funds. The challenge in this case is to communicate the received amount from Taiko to Arbitrum, so that the Arbitrum paymaster can enforce the voucher amount.

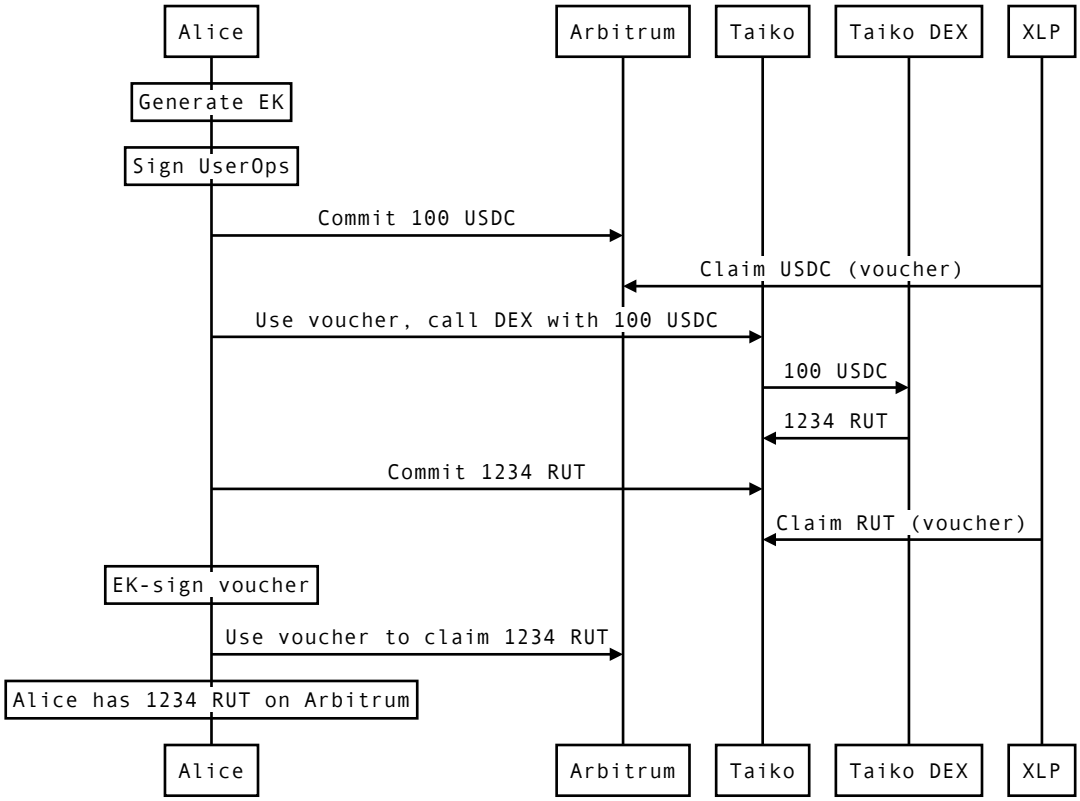
To do this without requiring Alice to sign a new UserOp when the information becomes available, we introduce an optional Ephemeral Key (EK). Alice generates EK when building the UserOps, and signs its public key as part of any UserOp that needs to receive dynamic vouchers or data. Alice's wallet submits the first UserOp, waits for an event it emits, then appends this data to UserOp.signature of the next UserOp and signs it with EK. The paymaster will validate this signature and treat the data as valid input.

The ephemeral key is not stored in the account and is only valid in the context of the current multichain transaction. It can be dropped after signing.

In the dynamic amount example above, EK will be used to sign the voucher that was received on Taiko after the swap. The subsequent Arbitrum UserOp will only be valid if it has a voucher signed by EK, which lets Alice ensure that her UserOp cannot be included with a voucher of a smaller amount.

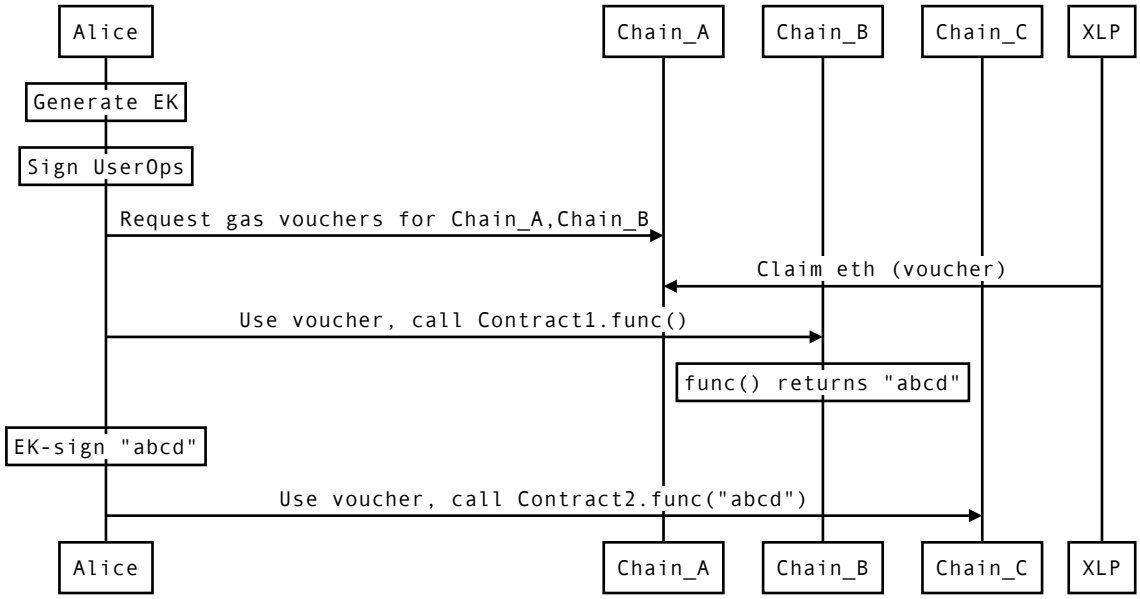
In other cases a UserOp may need arbitrary data emitted by a previous UserOp. In this case Alice's wallet signs this session data when it becomes available, and appends it to the next UserOp. Her account will be able to get this verified data from the paymaster and use it to construct a call.

Dynamic voucher flow



Alice received an amount not known before the transaction, and was able to transfer it to another chain. A frontrunner couldn't have interfered and replace the voucher for the dynamic amount because Alice's wallet signs it with the ephemeral key.

Passing arbitrary return data to another chain



Alice paid gas on chain_A for calls on Chain_B and ChainC, and used the return value of the 2nd call as input for the 3rd. She didn't have to sign a second time.

Optional note in voucher

Voucher notes enable building high level protocols using EIL.

The voucher request may include an optional `bytes32` note, which is opaque to the protocol but the XLP must include it in the voucher in order to claim the funds.

At first glance it might seem redundant since EK already enables the user to transfer data between chains. However, the trust model is different. The XLP attests for the note correctness, and won't be able to claim the funds at the source chain if it provides a voucher without the requested note. This enables use cases where the contract requesting the voucher does not fully trust the user, such as a higher level protocol implemented on top of EIL.

A protocol could be implemented on top of EIL by implementing a singleton ERC-4337 account and deploying it on multiple chains. This singleton can request vouchers with notes on one chain and process them on another when the user relays them there.

Voucher-based protocols inherit the self-service property of EIL. Instead of depending in bridge infrastructure, the protocol lets users relay their own messages.

Voucher Note trust model

The voucher note mechanism is not a general purpose messaging protocol. XLPs may collude with users and deliver vouchers at the destination, which did not really come from the source and contain false messages. However, since there was no matching request at the source, the XLP has no funds to claim there and loses the funds included in the voucher.

One way to look at it, is that a note is a crosschain message backed by the voucher amount. If an XLP signs a note in a 1000 USDC voucher, which a user delivers to the destination, then the destination contract may assume that either the message came from the source contract or the XLP "donated" 1000 USDC to the protocol by falsely attesting for it. False messages cost the sender an amount set by the protocol.

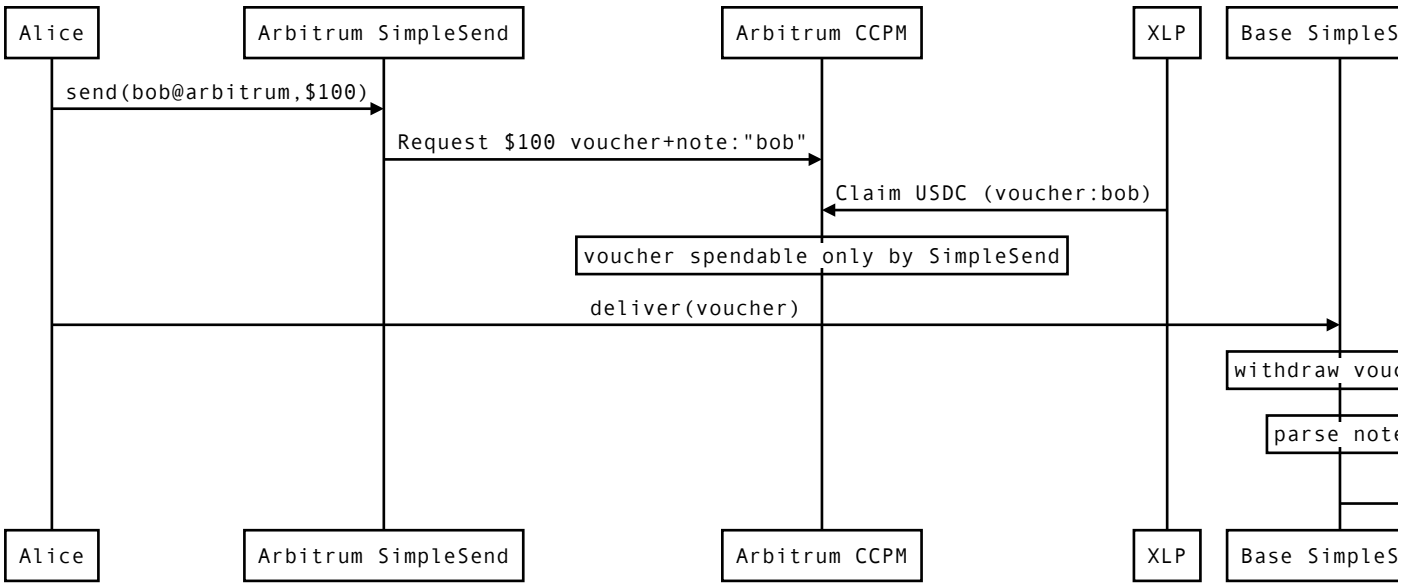
Therefore, the note can safely contain instructions for how to use the voucher funds, but not for anything that has implications beyond the value of the voucher.

Voucher Note example: Sending assets to a 3rd party - self service bridging

The usual EIL flow requires the user to use a multichain account which gets deployed at the destination, in order to withdraw the voucher and make a call there. What if Arbitrum Alice just wants to send funds directly to Base Bob? Deploying her account on Base introduces overhead and if she only plans to use Base once, she may prefer not to.

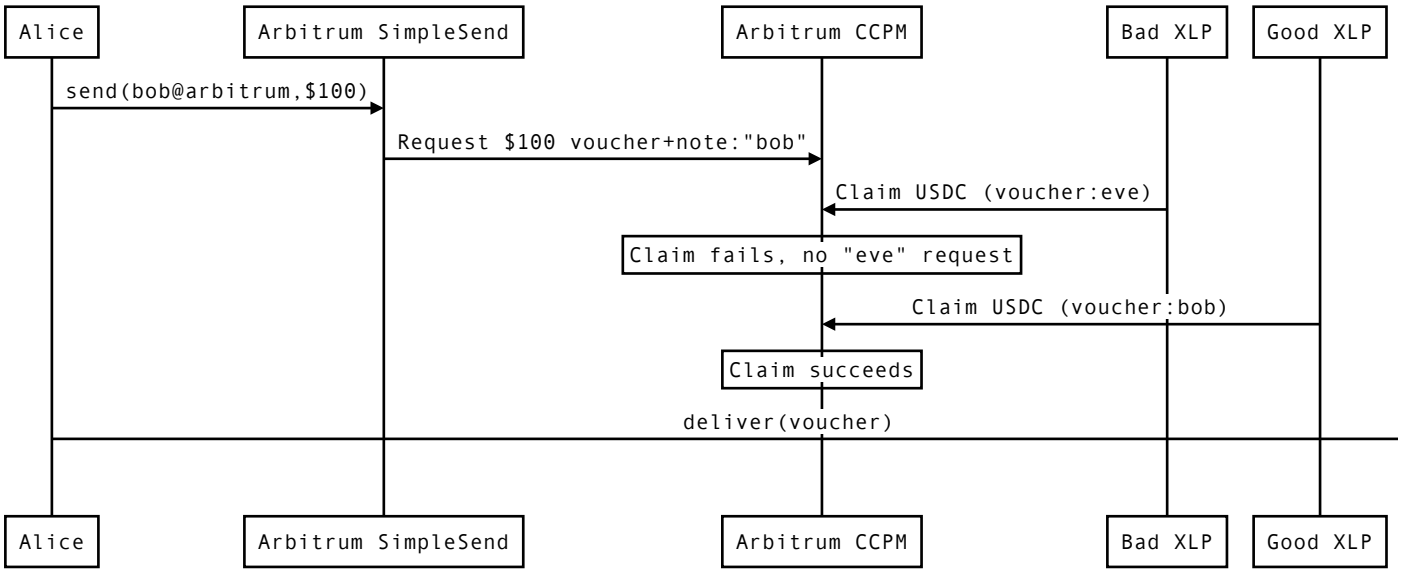
With Voucher Note we could build a simple self-service asset bridge protocol by implementing a SimpleSend contract. Alice can call it directly from her EOA without even setting a 7702 delegation. At the destination she sends a UserOp with sender=SimpleSend rather than her own address. SimpleSend accepts UserOps without a signature if they carry a voucher.

Here's how this protocol could work:

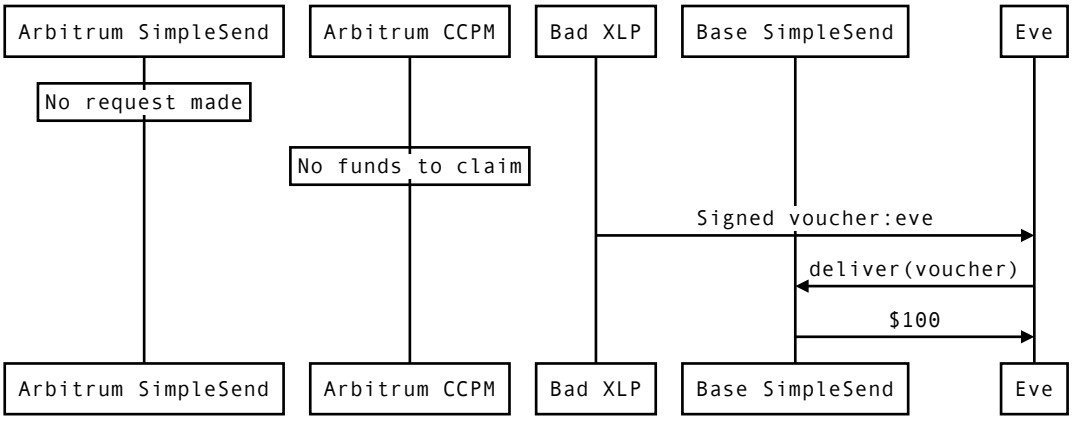


In this flow Alice bridged $100 to Bob without using bridge infrastructure.

What happens if a rogue XLP tries to change a note? Nothing:



What happens if a rogue XLP fakes a nonexistent message? XLP pays the full amount:

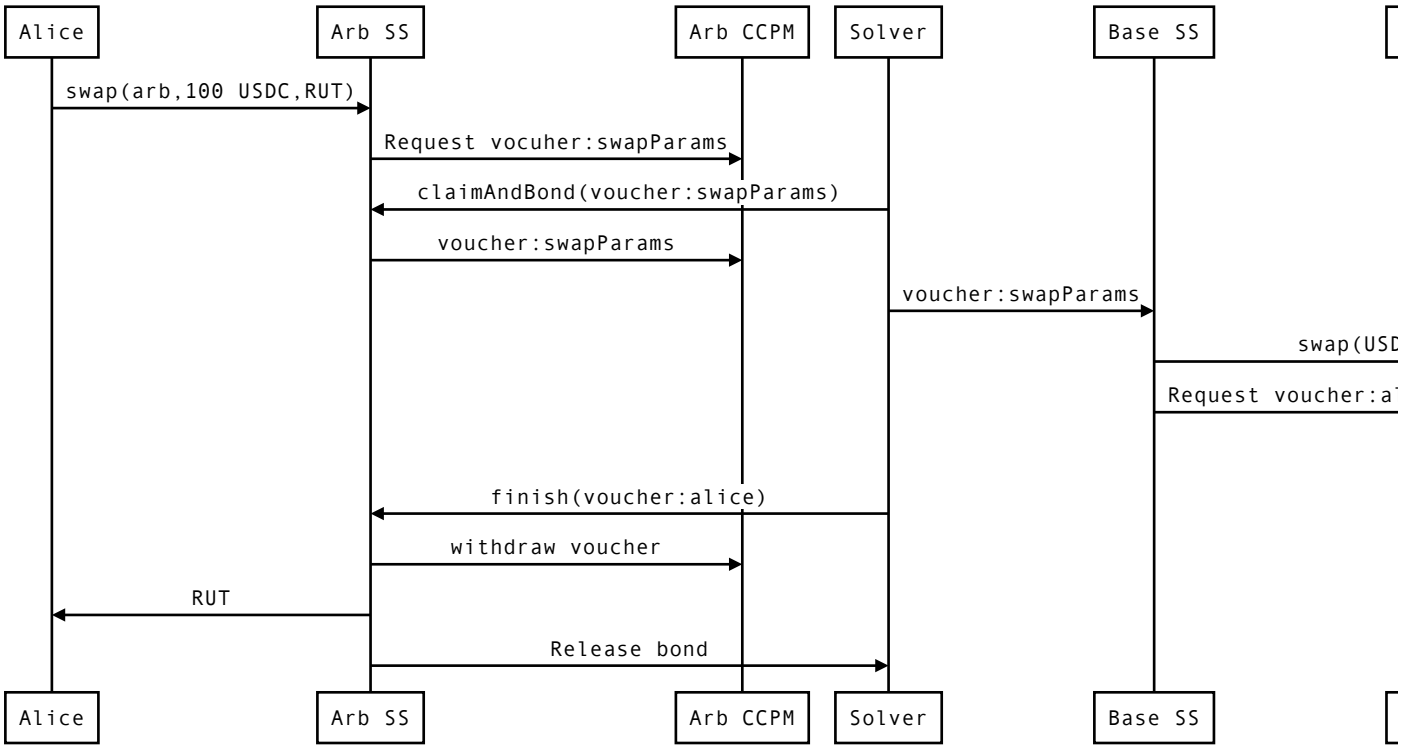


Bad XLP paid Eve $100 on Base, received nothing on Arbitrum. Could've just sent $100 directly to Eve.

Voucher Note example: Trust minimized intents

Another use case is trust-minimized intent-like protocols. Let a solver perform user calls instead of self-service, and use "note" to set the parameters to ensure that user actions are performed as requested. Let's see how a hypothetical intent protocol could be implemented.

Arbitrum Alice wants to swap USDC to RUT on Base. She uses SimpleSwapper (SS), an ERC-4337 singleton account that performs swaps on behalf of users. A solver performs both the XLP role and the remote execution. Alice doesn't need to trust the solver since it can only use her funds for the call she requested.



Alice sent her swap intent to SimpleSwapper, not interacting with a solver nor trusting it. She received the resulting RUT. The solver could not modify the swap params and could only receive its bond back after correct execution.

What happens if a rogue solver claims the swap and doesn't execute?

- Alice can reclaim her USDC after timeout. She calls SS.cancel(), SS reclaims the unused voucher funds from CCPM and gives them to Alice.
- Solver receives no funds and loses the intent bond.

What happens if a rogue solver claims the swap, executes it on Base, but doesn't finish() on Arbitrum?

- Alice or anyone else can see the voucher on Base and call finish(voucher:alice) after a short exclusivity window which the solver didn't use. The caller gets Solver's bond, which serves as an incentive for other solvers to complete the action.
- Solver loses the intent bond.

What happens if no XLP offers RUT liquidity?

- Solver is liable for the result. It'll lose its bond unless it can either deliver the RUT or refund the USDC. It shouldn't claim a swap without checking. Now it has to acquire RUT on Arbitrum and act as its own XLP to complete the operation.
- Alice doesn't lose the funds but they remain on Arbitrum, unless another solver gets RUT liquidity on Arbitrum and completes the operation for her. Other solvers are incentivized to help, as they can claim the negligent solver's bond.

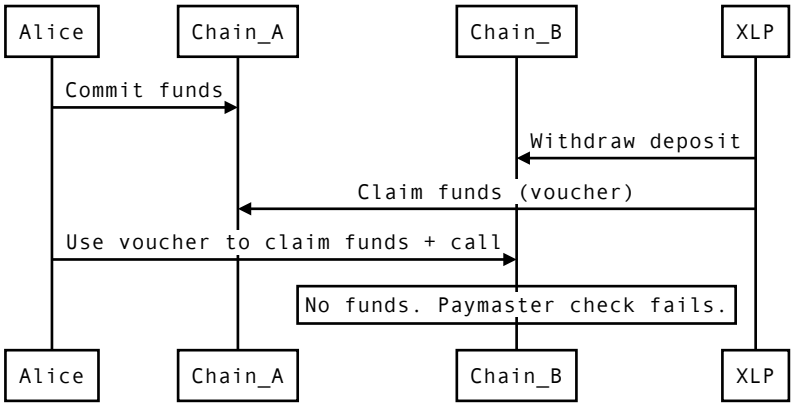
This is by no means a full Intent protocol, but demonstrates how such protocols could be built on top of EIL.

Attacks & mitigations

This section focuses on what could go wrong and how the protocol solves it. As expected, most of the complexity lies here. You may skip this section unless you enjoy the adversarial mindset.

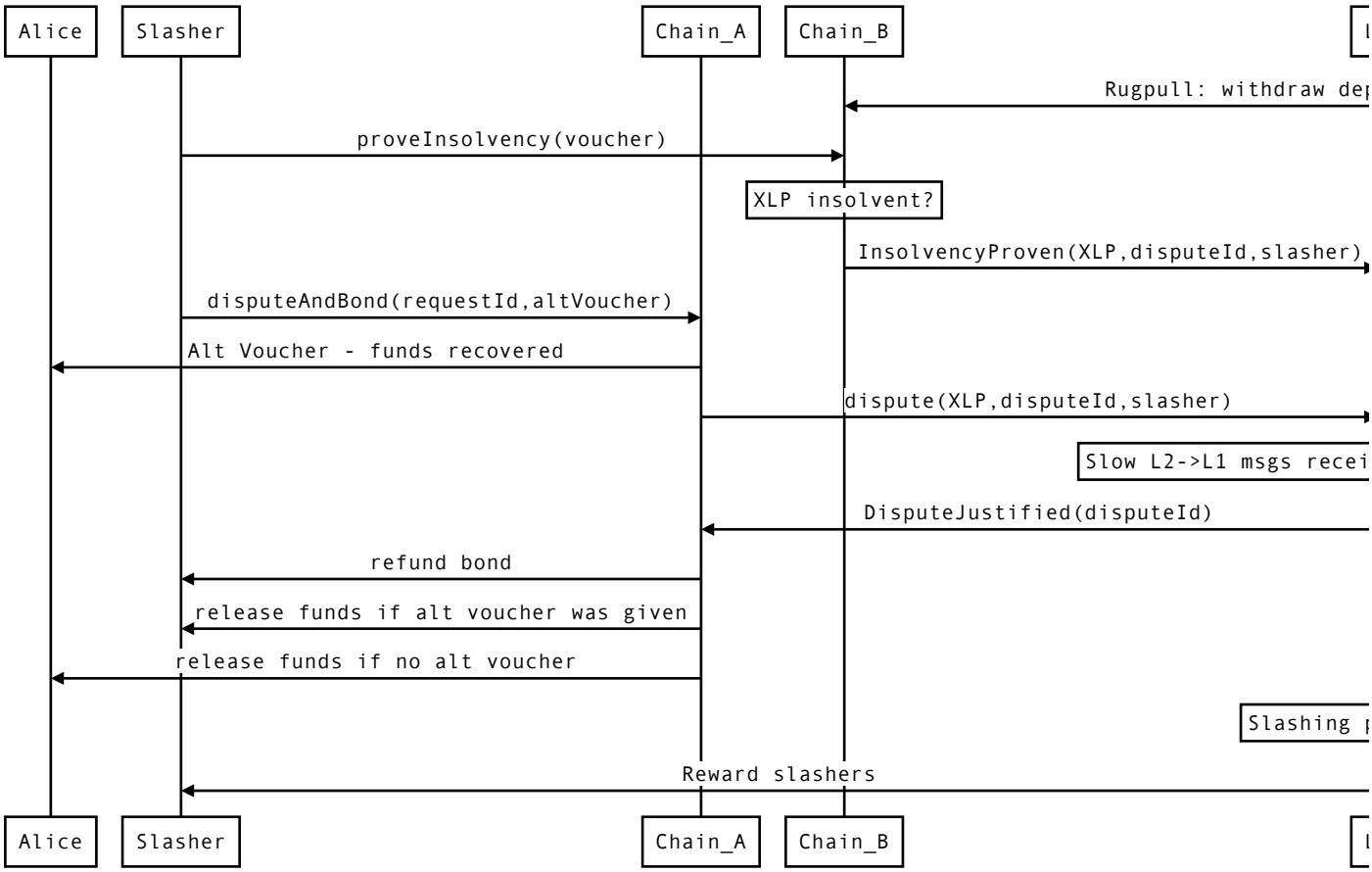
What happens if an XLP attempts to rugpull?

- XLP can remove Chain_B liquidity after Alice commits, then claim her funds on Chain_A.

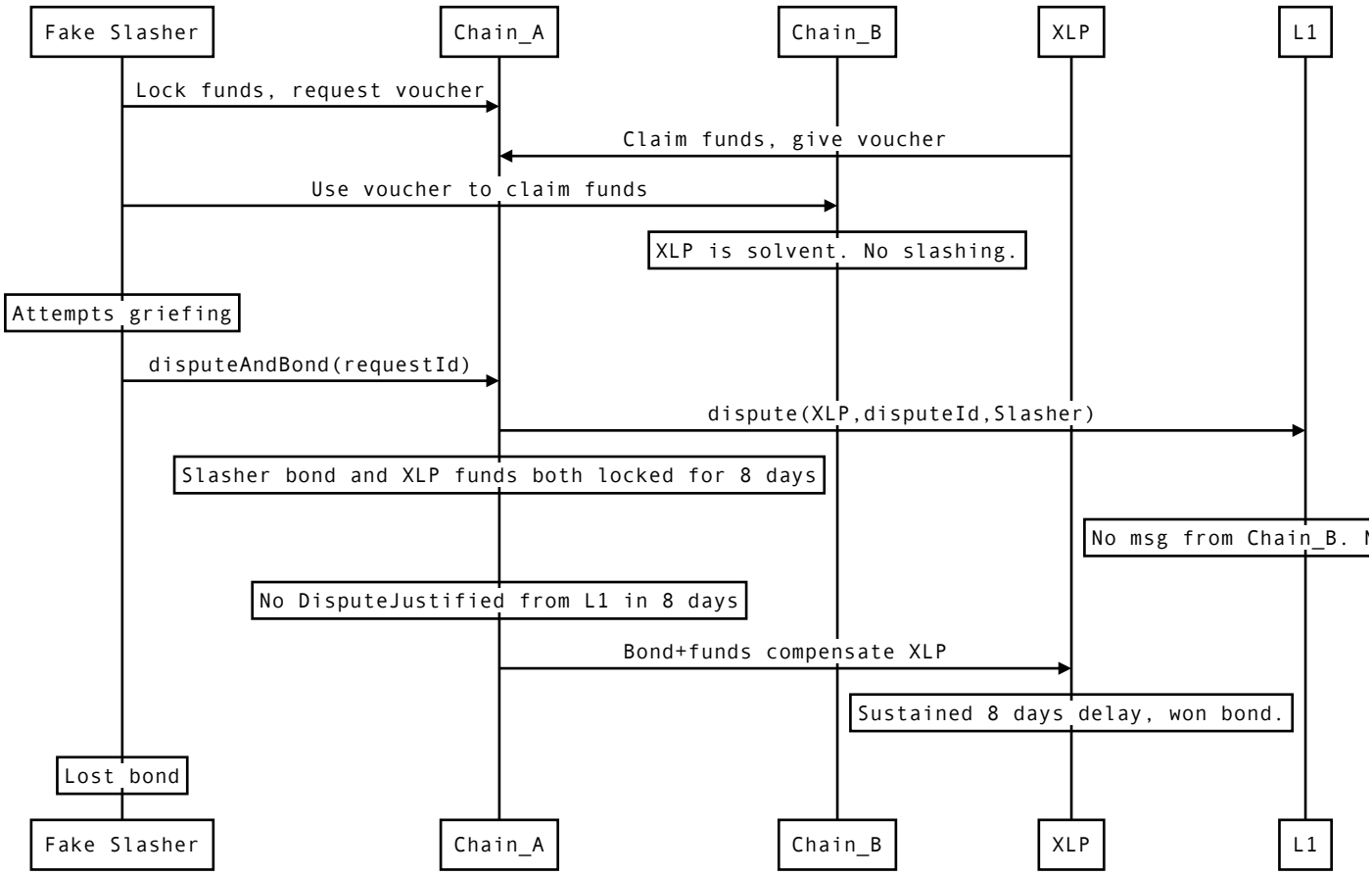


- The attack is provable on Chain_B:
  - Anyone can act as Slasher, calling Chain_B paymaster's `proveInsolvency()` with the insolvent voucher.
  - Paymaster verifies that the voucher hasn't expired and that XLP is insolvent.
  - Paymaster uses the canonical bridge to inform L1CrossChainStakeManager of the proof, burns the voucher and unregisters XLP.
  - Message takes up to 7 days. No rush, XLP's stake is locked for 8 days.
- L1CrossChainStakeManager requires two messages in order to finalize a dispute and reward the slasher: a proof from Chain_B and a dispute message from Chain_A.
- Anyone can report a dispute on Chain_A by calling `disputeAndBond()`
  - Disputer has to lock a hassle bond for 8 days. It'll be refunded upon successful dispute, or given to the misreported XLP otherwise.
  - The call must be done within an hour after the the request was made, while Alice's original funds are still locked.
  - It locks Alice's original funds for 8 days unless the dispute gets settled sooner.
- Upon dispute, any other approved XLP may provide an alt voucher for Alice's unfulfilled request on Chain_A.
  - The alt voucher can be provided either as a parameter to `disputeAndBond()` or separately by calling `replaceDisputedVoucher()`.
  - If an alt voucher was provided, Alice can use it immediately at the destination chain with her original UserOp. The incident is over for her immediately, and the dispute will get resolved between the XLPs later.
  - After Alice uses an alt voucher, the XLP that provided it calls `proveAltVoucherUsed()` on Chain_B, that informs L1CrossChainStakeManager which XLP satisfied the request.
- There may be additional insolvent pending vouchers from the same XLP, either for pre-dispute requests or post-disputes ones of users that were late to notice that the XLP got unregistered. The dispute process should include all of them:
  - The stake will be used to reward for reporting all insolvent vouchers during the slashing period.
  - A single call to `proveInsolvency()` or `disputeAndBond()` can dispute multiple vouchers. Callers are incentivized to include all known insolvent vouchers.
  - To avoid diluting the stake and diminishing the reporting incentive, L1CrossChainStakeManager will only reward the reporters that provided the longest arrays of pre-dispute and post-dispute vouchers for each pair of chains. Mechanism elaborated below.
  - All vouchers were published on Chain_A so any reporter has full knowledge and would undermine its own reward by submitting a partial report.
  - If XLP attempted to rugpull on multiple chains concurrently, its stake for each chain serves as a slashing reward for that chain.
- When L1CrossChainStakeManager receives both the Chain_A dispute and the Chain_B proof, it:
  - Finalizes the dispute.
  - Inform Chain_A paymaster via the canonical bridge that the dispute was justified.
  - Unregisters the XLP on all chains.
  - If slashing time hasn't been set for XLP, set it to `max(disputeTime,proofTime) + 8 days`.
- When Chain_A paymaster receives the message about a justified dispute, it:
  - Unregisters XLP.
  - Refunds the slasher's hassle bond.
  - Resets the lock on Alice's fund to 1 hour instead of what's remaining of the 8 days.
  - During the next hour, if it receives a message from L1 that an alt voucher was used (as reported by `proveAltVoucherUsed()`, it releases the funds to the XLP that provided it.
  - If no alt voucher was proven within an hour, the fund are released to Alice.
- When the slashing period ends - 8 days after the first dispute against the XLP is received:
  - For both Chain_A and Chain_B stakes:
    - Burn 10%.
    - Divide the remaining 90% of the stake to two parts: slashers reward, alt vouchers reward.
    - Disputers and provers may withdraw their share of the slasher reward. At most, 4 callers per chain would be rewarded:
      - Pre-dispute reporter
      - Post-dispute reporter (if exists)
      - Pre-dispute prover
      - Post-dispute prover (if exists)
    - Alt voucher providers get their share of the alt vouchers reward, proportional to the number of vouchers they provided.
    - For requests that did not receive an alt voucher, the alt voucher reward gets burned.

- After slashing concludes, further pending disputes can still finalize and release locked funds, but there is no stake to slash.



- If no `DisputeJustified` received on Chain_A within 8 days, XLP can claim both the locked funds and the Slasher's hassle bond. This disincentivizes fake slashing attempts that would delay XLP funds.



The rugpull mitigation process has the following properties:

- During normal operation no L1 messages are sent. No operational cost.
- Ensures that users never lose funds.
- Incentivizes XLPs to monitor each other. If an XLP becomes insolvent they're rewarded 60% of the stake for slashing it.
- Incentivizes XLPs to provide Alice with an alternative voucher ASAP. XLPs get 30% the stake for doing this, as well as reimbursement of the voucher.
- Disincentivizes rugpull attempts. XLP won't get away with user funds, and loses its stake.
- Disincentivizes fake slashing attempts. If a fake slasher delays XLP funds for 8 days, XLP receives them back along with the slasher's bond.

How the stake is structured and used

When registering on a chain, the XLP locks an L1 stake. This stake is split in half: one half is used to protect transfers from that chain, and the other protects transfers to that chain.

Each dispute requires two actions, which may be performed by the same disputer or by different ones:

1. Dispute at the source chain. This locks the voucher funds until the earliest of L1 settlement or 8 days.
2. Prove at the destination chain. The destination paymaster verifies the insolvency and informs L1 to settle the dispute.

For a dispute to settle on L1 and reward the disputers, both must happen. Therefore as long as the stake of at least one direction provides sufficient reward to cover both, a disputer is incentivized to perform both actions.

Since the actions may be performed by two different disputers, the stake has to reward both of them. An XLP may concurrently default on vouchers from a chain and to that chain. Splitting the chain's stake to source and destination ensures that disputes in each direction are incentivized even after a successful dispute in the other direction.

When a dispute against an XLP settles to L1:

- The source chain is informed of the dispute result, to release the funds to their rightful owners.
- If it's the first successful dispute against the XLP:
  - The XLP is unregistered from all chains and can no longer provide vouchers anywhere, but the stakes stay locked. It ensures that no further funds can be claimed by the XLP anywhere.
  - The 8 days slashing period for that XLP starts. This period is longer than the slowest L2 finality time, and ensures that the stakes will reward all disputers.

At the end of the 8 days slashing period:

- Burn 10% of each stake.
- Divide each stake to dispute rewards and alt voucher rewards.
- Source chain dispute stakes reward both pre-dispute reporters and post-dispute reporters (if exist).
- Destination chain dispute stakes reward both pre-dispute provers and post-dispute provers (if exist).
- XLPs that provided alt vouchers may claim their share of the alt voucher reward.
- The alt voucher reward for disputes that did not result in an alt voucher gets burned.

After slashing concludes, late disputes will still finalize and release locked funds at the source chain, but no reward is given. Users can use this to rescue their funds even if no XLP disputed on their behalf in a timely manner despite the incentives.

Why disputes are submitted in arrays rather than individually

An XLP may attempt to rugpull any number of current requests concurrently. It may provide vouchers to a multiple pending requests. In addition, more users might send new requests that whitelist the malicious XLP before they notice that it defaulted on its previous vouchers. The XLP's stake needs to incentivize disputers to report all of them.

Normally this wouldn't be a problem, as the reward just needs to be higher than the disputing transaction cost in order to justify reporting. However, an XLP may attempt to dilute the incentive and make it unjustifiable for disputers. The XLP rugpulls users and concurrently acts as a user, requests a large number of tiny vouchers, and provide them. Now there's a large number of disputes, and the disputing transaction cost may become higher than the reward.

To mitigate this, the disputer is expected to report **all** the vouchers by that XLP which are insolvent at the current block time. These vouchers are present in previous blocks of the source chains so there is no information asymmetry and every disputer knows about all of them. Therefore a honest disputer will always dispute all of them in a single array, and get a share of the reward for it. If multiple disputers submit arrays concurrently, only the longest one is rewarded. If same-length arrays are submitted, only the first one gets rewarded.

If the dilution attack above is performed and the XLP attempts to slash its own vouchers in many small disputes, the reward is not diluted because another disputer will include a longer array that includes **all** vouchers.

There may be two such arrays. One that includes all the insolvent vouchers at time of opening the dispute, and another for all the post-dispute vouchers that users may have requested before they learned that the XLP has defaulted. Users are given a grace period to notice that an XLP has been slashed at the destination and stop whitelisting it in their requests. A disputer reports the 2nd array at the end of this grace period.

Note that users won't lose funds even if they keep whitelisting the XLP after the grace period. The stake only rewards disputers during that period, but disputes that arrive later can still finalize. XLPs don't have an incentive to dispute after that period, but users may still dispute afterwards to rescue their funds. They'll incur the dispute transaction costs but not lose their funds.

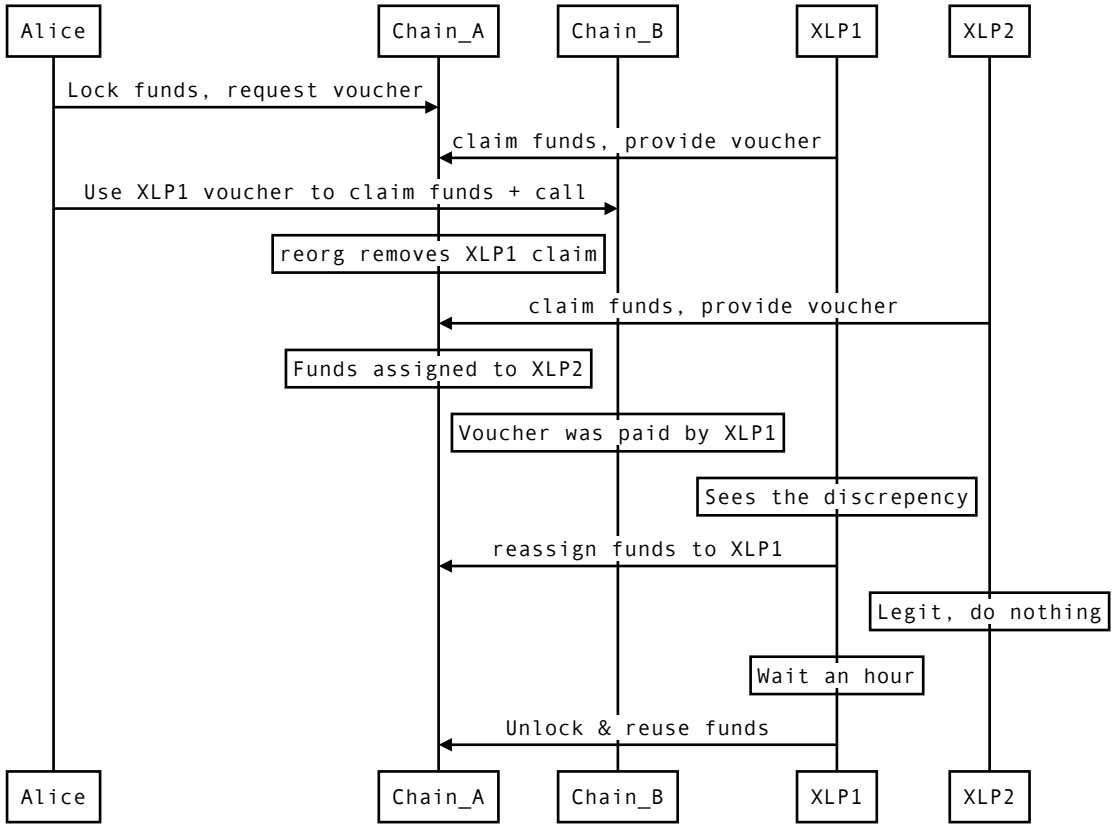
What happens if a voucher was provided on Chain_A but a different one was used on Chain_B

This can happen legitimately, though rarely, due to a Chain_A reorg where a different XLP provides the voucher. The protocol therefore supports reassigning the funds to the XLP whose voucher was actually used. This can be done without L1 involvement as long as both XLPs agree on which voucher was used. Disagreements can be resolved via L1 and slash an XLP that attempts to claim another XLP's funds.

- Voucher override requires a small bond - either 10% of the voucher's first asset, or 0.1 ETH. This provides disincentive, in addition to the stake, for attempting to delay XLP funds through false override.

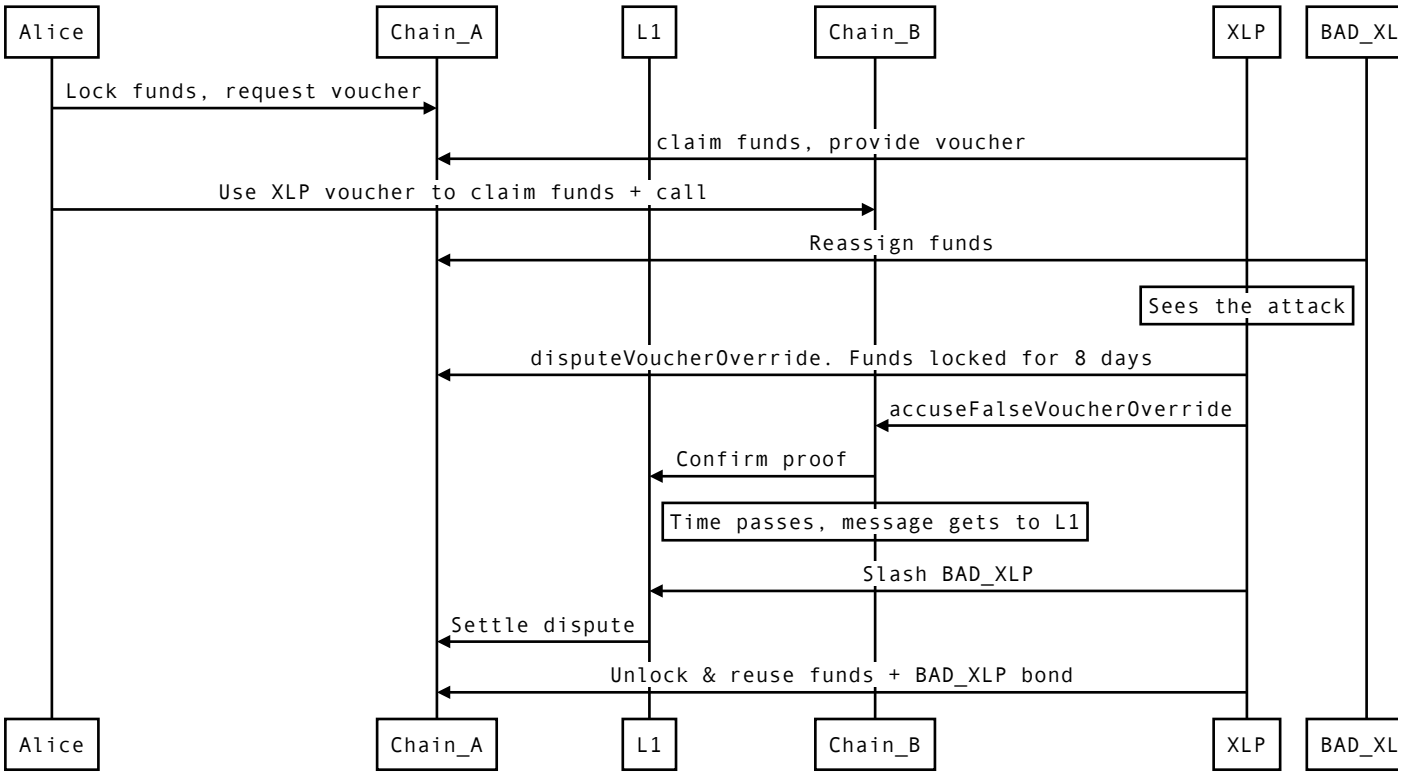
- Override can be called any number of times, only the last one applies.
- Override resets the 1 hour unlock period.
- An hour after override, the funds and the bond are released to the last caller.
- An override can be disputed. It locks the funds+bond for 8 days unless L1 settles the dispute sooner.
- In a disputed override the winner gets the funds, as well as the loser's bond and L1 stake.

Legitimate override flow



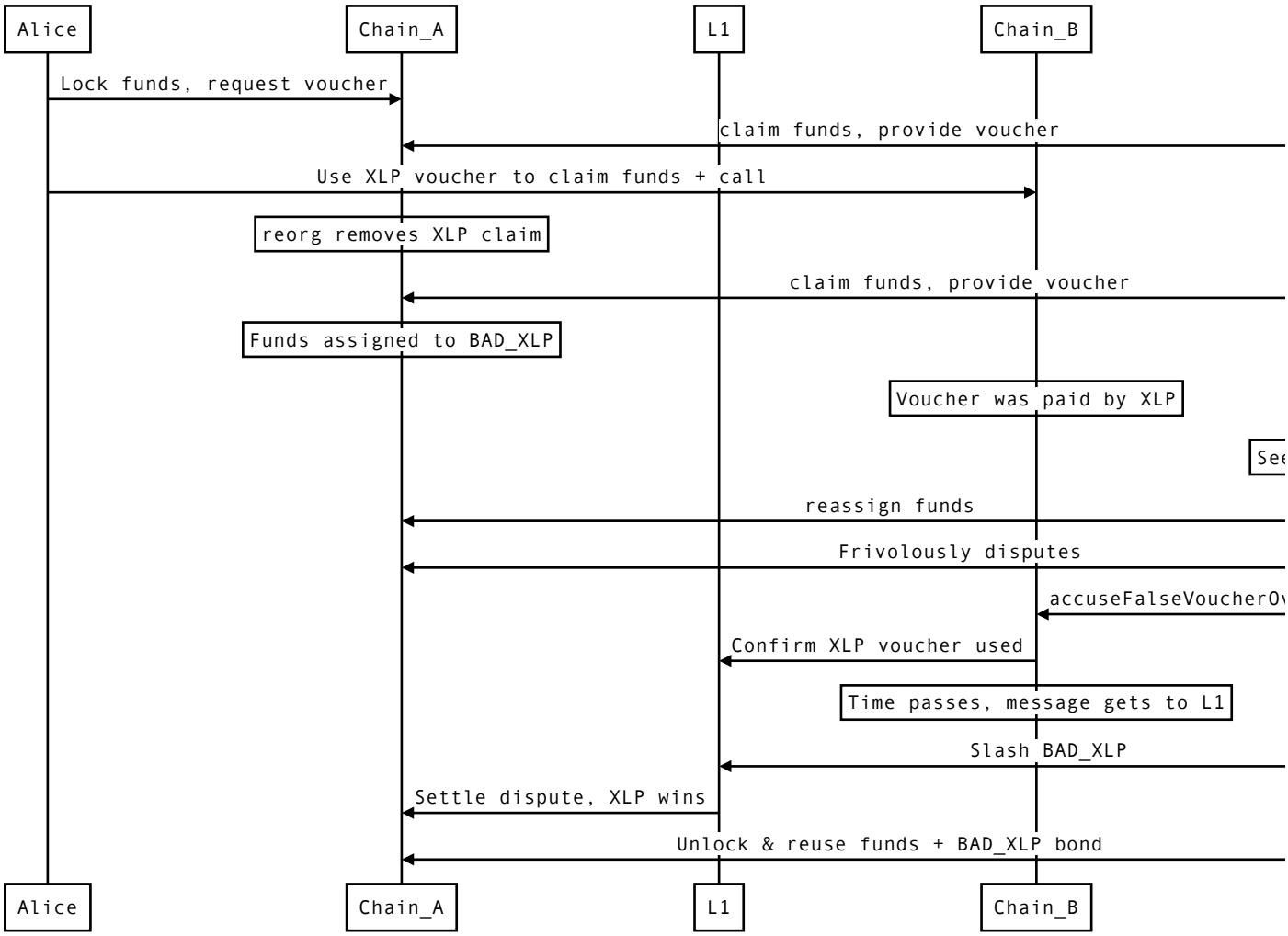
XLP2 did not complain about the override because it's legit - Alice didn't use its voucher.

Malicious override flow



BAD_XLP tried to steal XLP funds, got slashed. XLP was delayed 8 days but received funds+bond(10%)+stake.

Frivolous override dispute



BAD_XLP tried to steal XLP funds, got slashed. XLP was delayed 8 days but received funds+bond(10%)+stake.

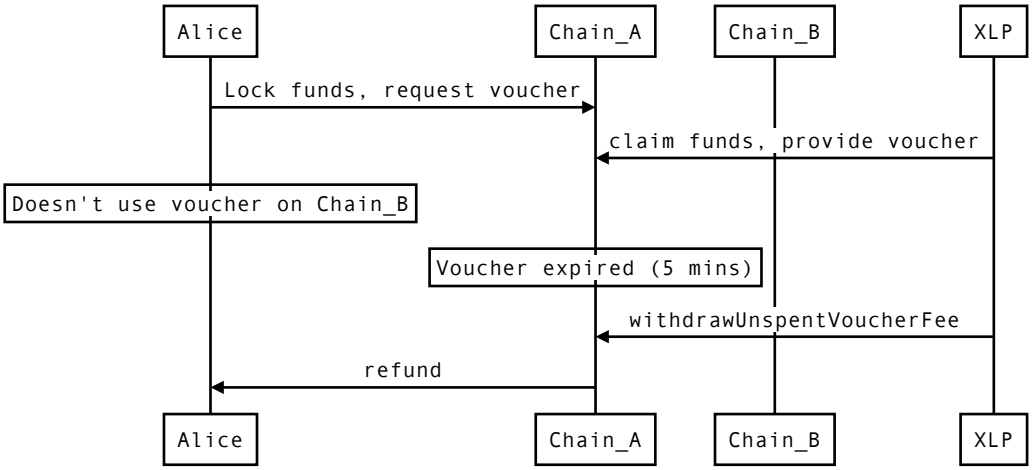
What happens if users request vouchers and don't use them, locking up XLP funds?

If Alice requests a voucher and doesn't use it, her funds are held for an hour but XLP also cannot use its funds for another voucher until Alice's expires and can no longer be used. Vouchers are meant for immediate consumption and have short expiration, but this still puts an unpaid burden on the XLP.

An unused voucher also puts Alice at risk. If Alice received a voucher and then disconnected before using it, she should be refunded and the XLP should not receive her funds. If nobody reports the voucher to Chain_A as expired, the XLP may claim her funds unjustifiably.

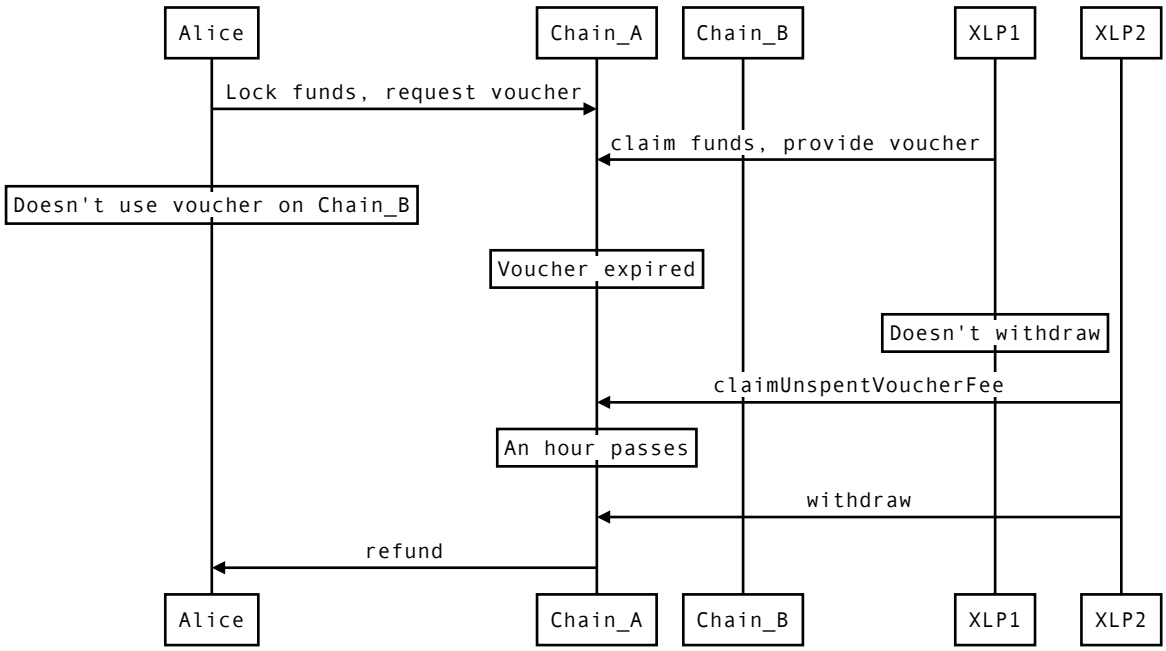
- To mitigate both of these problems, Alice's request may include a small Unspent Voucher fee.
- The XLP can withdraw it immediately as soon as the voucher expires, and Alice gets an immediate refund.
- If XLP doesn't withdraw, the fee may also be claimed by any other XLP with a 1 hour delay, by providing a proportional bond. It incentivizes other XLPs to report the voucher as unused and refund Alice if she fails to use the voucher, preventing the voucher issuer from wrongly claiming her funds.
- If an XLP wrongly reports a voucher as unspent, it may get slashed on L1 in addition to losing its bond.

Unspent voucher flow



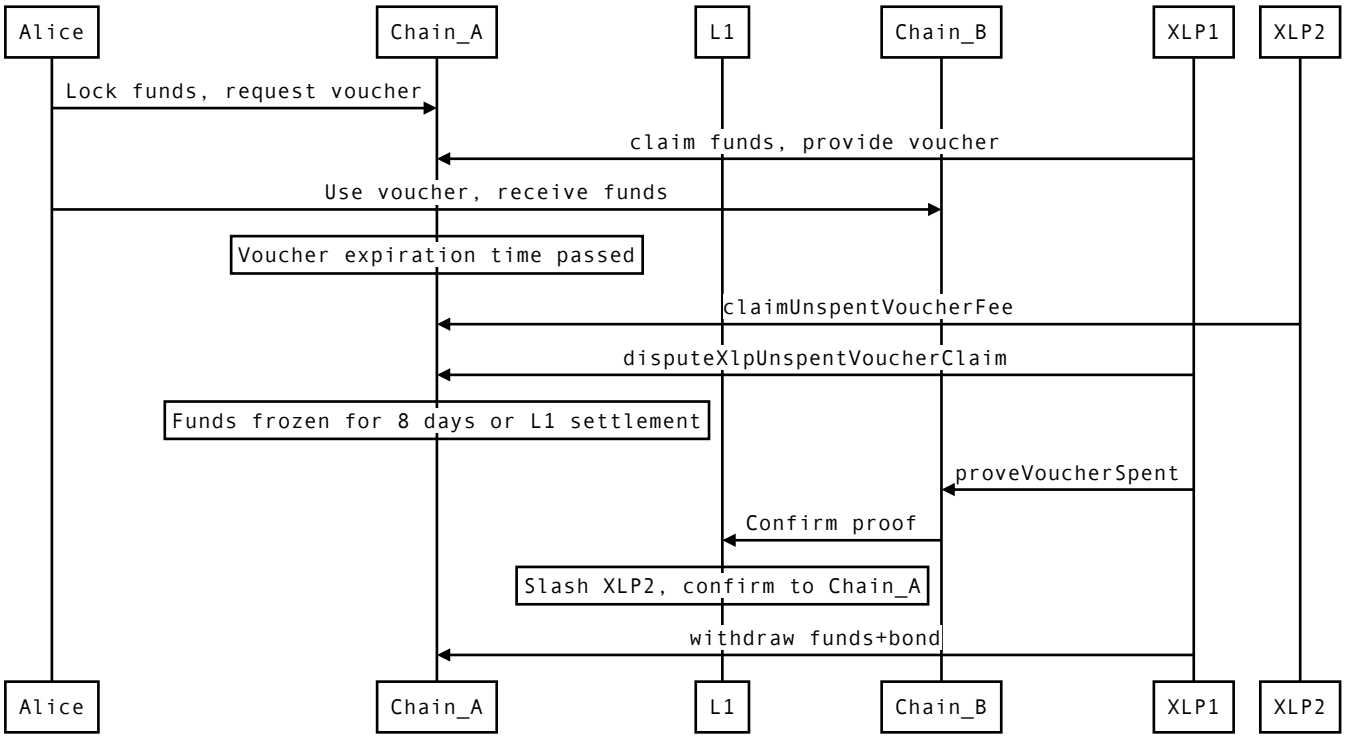
Alice didn't use her voucher. After 5 minutes XLP liquidity was released and Alice got refunded.

Unspent voucher 3rd party claim flow



Alice didn't use her voucher, XLP1 failed to report (hoping to claim her funds in 55 minutes). XLP2 claimed the fee, Alice got refunded anyway.

False unspent-voucher claim made



Alice used the voucher, XLP2 misreported it as unspent, XLP1 funds delayed. XLP2 slashed, XLP1 gets funds+bond+stake.

Note on voucher expiration time

For the protocol to work efficiently, vouchers should be short-lived. Either they're used soon after issuance, or they expire and the liquidity can be reused.

The UserOps are all part of the same multichain transaction and the wallet is expected to use the voucher at the destination's next block after it was received at the source. Vouchers should only allow a reasonable delay in case of network congestion. L2s typically have 1-2 seconds blocks and it's unlikely that a UserOp won't get included for over a minute or two.

If the destination is L1 then it makes sense to set a longer expiration time. However, even on L1 it's now rare for a transaction that pays market fee not to get included within minutes. A 5 minutes voucher seems sufficient

Trust assumptions

Censorship / DoS / griefing mitigation

- Transactions never pass through bridge contracts or XLPs. The user directly initiates all transactions. The paymaster and the XLPs enable gas payments but do not actually bridge transactions.
  - No censorship risk to users.
  - No DoS/griefing risks for XLPs.

Funds security

- Both user and XLP funds never lost/stolen.
- Both user and XLP funds unlikely to be delayed due to economic incentives.

Liveness

- Liveness not guaranteed but incentivized. Permissionless protocol invites more XLPs.
- User funds never get locked due to liveness failure:
  - If no XLPs are active, no one claims the funds and user commitment expires within seconds/minutes (the user controls the duration).
  - Once an XLP claims the funds, there is no liveness risk. The user uses the commitment on the destination chain without intermediaries.
- Liveness guarantee improves in the future with faster L2 finality.

Reorg risk

- No reorg risk for users. If an XLP claimed the funds, the user can always replay the commitment and reclaim funds on the destination chain after reorg.
- Reorg risk exists for XLPs. If source chain reorgs the user deposit may roll back after the user claims funds on the destination chain. Possible mitigation:
  - For very large amounts the XLP may choose to delay claiming and reduce the risk of user commitment getting reorged out.
  - For small amounts, ignore the risk. L2 reorgs are rare and an XLP that delays claiming will miss fees that a faster XLP will claim.
  - XLP can monitor for source chain reorgs and attempt to resubmit UserOps that got reorged out. Unless the reorg actually included a conflicting UserOp, resubmitting will recover the XLP funds.

- Another approach is to use RIP-7859 if adopted, and bind atomic swaps to the source chain's current history as well as L1's. It'll introduce a minor delay but eliminates the reorg risk. Probably an overkill except for very large swaps.

Conclusion

EIL is a wallet-centric cross-L2 interoperability protocol, designed to achieve seamless multichain UX without compromising on trust assumptions. The vision and rationale are described in a separate post: EIL: Trust minimized cross-L2 interop

This post covered the flows and mechanism design of EIL, focusing on trust assumptions, attacks and mitigations. It also described advanced composability extensions that were out of scope for the high level post.