Audit Report

Blockswap Gateway

Delivered: 2022-10-17

Prepared for Blockswap by Runtime Verification, Inc.

runtime verification

Summary

Runtime Verification, Inc. has audited the smart contract source code| for Blockswap's Gateway protocol. The review was conducted from 2022-09-19 to 2022-10-14.

Blockswap engaged Runtime Verification in checking the security of their gateway protocol. The gateway allows the state of the Stakehouse protocol[1], which resides on Ethereum, to be extended to other chains and execution environments. This state extension allows users to transfer their dETH (part of the Stakehouse protocol) to different chains. At the time of the audit, gateway endpoints are only provided for Ethereum and Optimism, but more are planned in the future.

The issues which have been identified can be found in section Findings. A number of additional suggestions have also been made, and can be found in section Informative findings. We also created an abstract model of the gateway protocol and used it to formulate important invariants, which can be found in section Appendix 1: Formal Model.

Scope

The audited smart contracts are:

- `savETHDestinationGateway`
- `savETHDestinationReporter`
- `savETHGateway`
- `savETHGatewaySignatureValidator`
- `savETHOriginGateway`
- `savETHRegistryDestinationGateway`
- `StakeHouseUniverseDestinationGateway`

The audit has focused on the above smart contracts, and has assumed correctness of the libraries and external contracts they make use of. The libraries are widely used and assumed secure and functionally correct.

The review focused mainly on the `Stakehouse-V2` private code repository, which is a Hardhat project with test scripts. The code was frozen for review at commit `6e6db7bdaf98ddab65f6d735da27f5a0dcc5fcc7`.

Blockswap also provided access to the `gateway` private code repository, which contains the off-chain portion of the Gateway protocol. Although reviewing off-chain and client-side code is not in the scope of this engagement, they have been used for reference in order to understand the design of the protocol and assumptions of the on-chain code, as well as to identify potential

[1] Stakehouse is a programmable staking layer built on top of the Ethereum 2.0 Beacon Chain and has been audited separately. See Stakehouse Audit Report and Stakehouse 2nd Audit Report.

issues in the high-level design. Details of the implementation of the off-chain code are not in scope for this audit.

Assumptions

The audit is based on the following assumptions and trust model.

1. All addresses that have been assigned a role need to be trusted for as long as they hold that role. In particular, this concerns the Common Interest Manager role, which has the power to mark a particular router instance as trusted.
2. The contracts are upgradeable. Thus, whoever has the power to upgrade the contracts must be trusted, as they can significantly change the behavior of the protocol.
3. The off-chain router only signs valid transactions and Merkle roots (but see B04).
4. The Optimism team is honest and the sequencer works correctly. We trust that the Optimism team uses their ability to upgrade core contracts responsibly. Further, we trust that the sequencer only publishes valid state commitments. This is particularly important at the moment because Optimism does not have a fault proof process that would allow validators to challenge invalid state commitments. See Optimism's Security Model.

Note that assumptions 1 and 2 roughly assume honesty and competence. However, we will rely less on competence, and point out wherever possible how the contracts could better ensure that unintended mistakes cannot happen.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have used the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. Second, we carefully checked if the code is vulnerable to known security issues and attack vectors. Thirdly, we discussed the most catastrophic outcomes with the team, and reasoned backwards from their places in the code to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with the Blockswap team and offered our feedback during ongoing design discussions, and suggested development practices as well as design improvements.

This report describes the **intended** behavior and invariants of the contracts under review, and then outlines issues we have found, both in the intended behavior and in the ways the code differs from it. We also point out lesser concerns, deviations from best practice and any other weaknesses we encounter. Finally, we also give an overview of the important security properties we proved during the course of the review.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only, and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive, and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Background: Stakehouse Protocol

Since the gateway protocol builds on top of the Stakehouse protocol, we briefly introduce the main features of the Stakehouse protocol here. For a more detailed description, see the Stakehouse Audit Report.

Stakehouse is a programmable staking layer built on top of the Ethereum Beacon Chain. Unlike standard liquid staking solutions, Stakehouse does not run validators for the users or allow stakes of less than 32 ETH. Instead, it allows a user to register a validator and mint derivative tokens corresponding to the 32 ETH stake, split into 24 dETH and 8 SLOT. dETH is a risk-free asset which can be traded freely and is redeemable 1:1 for ETH, with more dETH being minted as a validator earns inflation rewards. SLOT serves as collateral to protect dETH from slashing, as well as giving rights on the management of the validator. KNOTs, as validators are known within the protocol, are divided into stakehouses, which serve as communities with an interest in helping their members run their validators effectively.

When a user first registers a KNOT in a stakehouse, the KNOT is put into an index owned by the user. An *index* refers to a collection of KNOTs, and the index owner is entitled to all the dETH rewards earned by these KNOTs. However, the dETH is held by the contract and not directly accessible to the user. To access the dETH associated with a KNOT, the index owner needs to first transfer the KNOT into a special index called the *open index*. In return, the user receives savETH corresponding to the amount of dETH that the KNOT contributes to the total amount of dETH in the open pool. The dETH associated with KNOTs in the open index is owned collectively by savETH holders. At any time, savETH holders can get their share by exchanging savETH for dETH.

Gateway: Contract Description and Invariants

The description in this section assumes that findings B04 and B05 have been addressed.

This section describes the gateway protocol at a high-level, and which invariants of its state we expect it to always respect at the end of a contract interaction. Note that we use the term *domain* to refer to different chains and execution environments like Ethereum and Optimism.

Overview

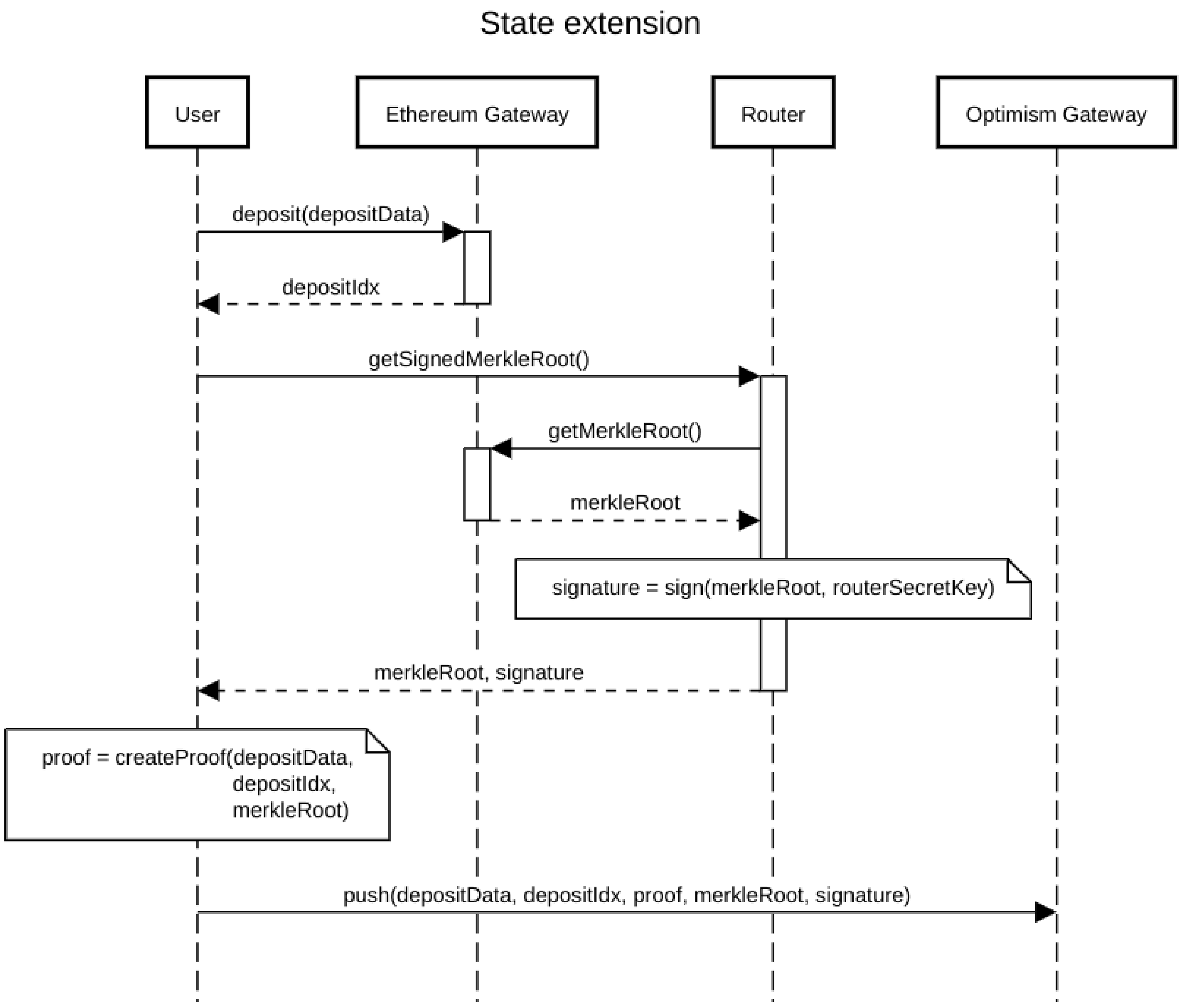
The goal of the gateway is to make the savETH (and by implication, the dETH) associated with a KNOT available on Optimism (more domains are planned). This is done in a process called *state extension*. Essentially, this involves locking the KNOT's savETH on Ethereum and creating a copy of the KNOT on Optimism, where the user can then mint the associated savETH. Going in the other direction is called *state unwinding*: The savETH on Optimism is burned and all traces of the KNOT are removed, and the original KNOT on Ethereum is unlocked, making its savETH accessible again. The state of the Stakehouse protocol on Ethereum is always considered to be the single source of truth from which its state on other domains is derived.

The gateway consists of two endpoints, one on Ethereum and one on Optimism. Both state extension and state unwinding consist of a two-step process that involves an initiating call on one domain and a finalizing call on the other domain. Furthermore, since the two endpoints cannot directly share information with each other, there exists a trusted intermediary called the *router*, which ensures that a user can only finalize an operation if it has been successfully initiated on the other side. This is achieved with the help of a Merkle tree and works as follows: When a user initiates an operation (either state extension or state unwinding) on the origin domain, then this is recorded in a Merkle tree stored in that domain's gateway. To finalize the operation on the destination domain, the user needs to pass the origin domain's Merkle root, signed by the router (but see B04), to the destination gateway, together with a proof that the Merkle tree indeed contains a record of the operation's initiation. The destination gateway then checks whether the given proof is valid for the signed Merkle root (but see B05). If the check succeeds, then it is safe to assume that the operation has been initiated on the origin domain, under the condition that the router is trusted.

The basic flow for a savETH state extension from Ethereum to Optimism is illustrated in the diagram on the next page. The user needs to perform the following steps:

1. Initiate the state extension by calling the `deposit()` function on the Ethereum gateway. The user needs to provide all the data necessary for the deposit transaction, which includes

- `blsPubKey`: the BLS public key of the KNOT whose savETH should be made available on Optimism
- `destinationIndex`: the savETH index on Optimism whose owner will have access to the migrated savETH


State extension
User
Ethereum Gateway
Router
Optimism Gateway
deposit(depositData)
depositIdx
getSignedMerkleRoot()
getMerkleRoot()
merkleRoot
signature = sign(merkleRoot, routerSecretKey)
merkleRoot, signature
proof = createProof(depositData, depositIdx, merkleRoot)
push(depositData, depositIdx, proof, merkleRoot, signature)

The `deposit()` function performs two important operations: (1) Transferring the KNOT into a special savETH index owned by the gateway and (2) inserting all the information about the deposit transaction into a Merkle tree.

The user needs to remember the index at which the deposit transaction is inserted into the Merkle tree, denoted by `depositIdx`.

2. Next, the user needs to interact with the router and request the signed Merkle root from the Ethereum gateway.

3. Afterwards, the user creates a Merkle proof to demonstrate that the deposit described by `depositData` has been inserted into the Merkle tree denoted by `merkleRoot` at index `depositIdx`.

4. Now the user can finalize the state extension by calling `push()` on the Optimism gateway. This first verifies that the signature for the Merkle root is valid (assuming B04 has been addressed), and then validates the deposit against the Merkle root using the provided proof. If these checks pass, the savETH of the KNOT identified by `blsPubKey` is made available to the owner of index `destinationIndex`.

It is important to note that when the savETH state of a KNOT is extended to Optimism, the KNOT is first transferred to a special index owned by the Ethereum gateway. This means that the KNOT's dETH cannot be accessed by any user on Ethereum. Thus, while a KNOT's dETH is available on Optimism, it is locked on Ethereum, ensuring that the total amount of spendable dETH remains the same (an important property when making tokens available across multiple domains).

State unwinding proceeds similarly to state extension, except that the roles of the Ethereum gateway and the Optimism gateway are reversed. Also, during state unwinding, all traces of the KNOT are removed from Optimism, and on Ethereum the KNOT is transferred from the gateway index to a user-owned index, making its dETH spendable again.

Besides state extension and unwinding there is one more operation provided by the gateway: updating the dETH balance of a KNOT on Optimism when new rewards are reported on Ethereum. Since users may want to extend the state of their KNOT for an extended amount of time, this allows them to forward any validator rewards reported on Ethereum to Optimism.

In summary, the gateway protocol provides the following three operations:

- *State extension:* Makes a KNOT's savETH (and by implication, its dETH) spendable on Optimism, while locking its savETH on Ethereum.
- *Balance update:* Updates the balance of a state-extended KNOT on Optimism as new validator rewards are reported on Ethereum.
- *State unwinding:* Removes all traces of a state-extended KNOT from Optimism (which includes burning all associated tokens) and unlocks the KNOT's savETH on Ethereum.

Each of these operations must be initiated at one gateway and finalized at the other by calling the corresponding function on `savETHOriginGateway` (deployed on Ethereum) and `savETHDestinationGateway` (deployed on Optimism).

	`savETHOriginGateway`	`savETHDestinationGateway`
State extension	`deposit()` →	`push()`
Balance update	`pokeLatestBalance()` →	`balanceIncrease()`
State unwinding	`push()` ←	`deposit()`

Properties

There are several important properties that should be satisfied by the gateway at all times. As a general rule, the gateway should under no circumstances be able to compromise any invariant of the Stakehouse protocol on Ethereum, even if Optimism or the router are faulty or become dishonest. This is because the state on Ethereum acts as the single source of truth for anything related to the Stakehouse protocol, and as such, the integrity of this state is of utmost importance. In order to achieve this goal, the gateway is implemented such that it only uses existing operations of the Stakehouse protocol, which ensures that no Stakehouse invariant can be broken by the gateway, even if Optimism and the router cease to be trustworthy.

In the following, we list several properties that the gateway should satisfy. These are not the only ones, but they are fundamental for the correctness of the protocol and as such deserve special attention. For each property, we also specify the trust assumptions that need to be satisfied in order for the property to hold.

P1 *On Ethereum, the gateway does not create dETH out of thin air.*
Depends on correctness of:
- Stakehouse protocol on Ethereum

P2 *On Ethereum, the gateway does not cause any dETH to be locked up forever.*
Depends on correctness of:
- Stakehouse protocol and gateway on Ethereum
- Stakehouse protocol and gateway on Optimism
- Router

Note that the Blockswap team plans a feature that allows the community to unlock dETH held by the gateway in case the Optimism L2 does not work as intended anymore. This feature would reduce the amount of trust that needs to be placed on Optimism.

P3 *On Optimism, the gateway does not create dETH out of thin air.*
Depends on correctness of:
- Stakehouse protocol and gateway on Ethereum
- Stakehouse protocol and gateway on Optimism
- Router

P4 *On Optimism, the gateway does not cause any dETH to be locked up forever.*
Depends on correctness of:

- Stakehouse protocol and gateway on Ethereum
- Stakehouse protocol and gateway on Optimism
- Router

P1 and P2 regulate the amount of spendable dETH on Ethereum. An important invariant of the Stakehouse protocol is that – in the absence of black swan events – all dETH is backed by staked ETH on the Beacon Chain. If the gateway was able to create dETH out of thin air, this invariant would be broken. As P1 states, this property only depends on the correctness of the Stakehouse protocol and is independent of the correctness of the gateway and of the trustworthiness of Optimism and the router. Another important property, which is stated by P2, is that any dETH held by the gateway can be unlocked again. If this property was broken, then some staked ETH would become inaccessible. Ensuring that this does not happen requires cooperation with Optimism and the router and hence depends on their correctness.

P3 and P4 state the same properties as P1 and P2, but this time for Optimism. In contrast to dETH on Ethereum, if Optimism or the router cease to be trustworthy, then it is possible that some dETH on Optimism is created out of thin air or locked forever. Thus, using dETH on a domain different from Ethereum means you have to extend your trust assumptions, and if these additional assumptions are broken, then there may exist some dETH on Optimism that is not backed by staked ETH on the Beacon Chain. This is to be expected but we mention this here to make it more explicit.

One fundamental invariant that should be satisfied by any bridge or gateway is that it must not affect the total amount of spendable assets. In fact, the above properties imply the following invariant:

Invariant: The total amount of spendable dETH across Ethereum and Optimism is not affected by the gateway, given that the trust assumptions of P1–P4 are satisfied.

Since this is such a fundamental invariant, we formalize it in section Appendix 1: Formal Model and provide a proof in section Appendix 2: Proofs.

Findings

A01: Gateway may be compromised by single leaked key

[Severity: High | Difficulty: High | Category: Security]

The router is only considered a temporary solution and will be replaced by a network of endorsers before release. This would address this finding.

If an attacker gets hold of the router's signing key, they are be able to do the following:

1. On Optimism, mint an arbitrary amount of dETH that is not backed by staked ETH on the Beacon Chain.
2. On Ethereum, take ownership of any KNOT (and its dETH) whose state is currently extended.

Note that in the first case, the newly minted dETH cannot be transferred to Ethereum using the gateway protocol. This is because the gateway on Ethereum only allows state unwinding for a KNOT's savETH if its state is currently extended, and only if the dETH balance on Optimism is at most as large as the dETH balance on Ethereum. This means that no matter what happens on Optimism, the dETH on Ethereum is always backed by staked ETH on the Beacon Chain.

Scenario

Scenario 1: Mint an arbitrary amount of dETH on Optimism.

1. Create and sign a `DepositMetadata` together with a fabricated Merkle root using the leaked signing key of the router. Set `DepositMetadata.depositAmount == 1000 ether` (or any other amount of dETH you want to be able to mint on Optimism) and `DepositMetadata.destinationIndexId` to a savETH index on Optimism that you own
2. Pass the signed `DepositMetadata` together with the fabricated Merkle root to `savETHDestinationGateway.push()`
3. A KNOT worth 1000 dETH is now in your index on Optimism. The dETH can be minted by moving the KNOT into the open index and exchanging the received savETH for dETH

Alternatively, instead of using `savETHDestinationGateway.push()`, you can also use `savETHDestinationGateway.balanceIncrease()` to increase the balance of a KNOT that you already own on Optimism.

Scenario 2: Take ownership of any state-extended KNOT on Ethereum and take its dETH.

1. Create and sign a `DepositMetadata` together with a fabricated Merkle root using the leaked signing key of the router. Set `DepositMetadata.blsPublicKey` such that it refers to an arbitrary KNOT that is currently state-extended (it does not matter who owns it), and set `DepositMetadata.destinationIndexId` to an savETH index you own on Ethereum
2. Pass the signed `DepositMetadata` together with the fabricated Merkle root to `savETHOriginGateway.push()`
3. The KNOT identified by `DepositMetadata.blsPublicKey` is now in an index you own. The dETH can be retrieved by moving the KNOT into the open index and exchanging the received savETH for dETH

Recommendation

It is recommended to require signatures of multiple parties before a gateway contract allows a user to push a deposit, to reduce the risk that a single leaked signing key compromises the whole protocol.

Status

The client is aware of this and considers the current router implementation only a temporary solution. Before releasing the gateway protocol, they plan to replace the router with a decentralized network of endorsers. The gateway contracts would then only allow a user to push a deposit if it has been signed by a sufficient number of endorsers.

A02: Router retrieves data from single Ethereum/Optimism node

[Severity: High | Difficulty: High | Category: Security]

When a user wants to perform a state extension or unwinding, the router is responsible for retrieving and signing the Merkle root from either the Ethereum or the Optimism gateway. If the Merkle root retrieved by the router is not correct, this could mean that the user is unable to finalize their transaction at the destination domain. It could also mean that the user is able to finalize a transaction that they are not supposed to, which may lead to the same dETH being spendable on both Ethereum and Optimism.

Thus, it is crucial for the router to fetch the correct Merkle root. Its ability to do so is directly affected by the trustworthiness and correctness of the Ethereum and Optimism nodes involved. In such a situation, depending on only a single node is generally seen as insufficient.

Recommendation

To mitigate the risk of a single malfunctioning or hacked node affecting the security of the gateway, the router should consult multiple Ethereum and Optimism nodes and only proceed if the data returned by each of them is the same.

Status

Addressed. (Note that code changes have not been audited.)

Informative findings

B01: Off-chain router vs Optimism's messaging infrastructure

[Severity: - | Difficulty: - | Category: Security]

State extension and unwinding require the Blockswap Gateway to pass messages between Ethereum and Optimism. There are multiple approaches to this, each with their own tradeoffs, see Communication Strategies for an overview.

One approach is to use the messaging infrastructure provided by Optimism itself (see Sending data between L1 and L2). However, the Blockswap Gateway uses a different approach that is based on its own off-chain router implementation. The (dis)advantages of both approaches are summarized in the following table.

	Advantages	Disadvantages
Optimism's messaging infrastructure	- **No additional trust assumptions** - Relatively easy to use	- Messages from L2 to L1 only valid after 7 days - Specific to Optimism
Blockswap's off-chain router	- Messages from L2 to L1 are finalized much faster - Same mechanism can be reused for other L2s with relatively minor changes	- **Router needs to be trusted** - Additional implementation overhead - Operational overhead - Fully centralized[2]

From a security perspective, the main difference is that relying on a custom router implementation means we need to additionally trust the router to be honest and correct. On the other hand, using Optimism's messaging infrastructure only requires trusting Optimism itself, which anyone who moves assets to Optimism needs to do anyway.

We do not strictly advise against using the router, though we do want to point out that doing so places a great amount of additional trust in the correctness and honesty of the router (also see A01 and A02). In general, approaches that require less trust are preferable, though they have their own disadvantages (for example, having to wait for the 7 day challenge period when sending messages from L2 to L1).

[2] This will likely change once the gateway switches to a network of endorsers (see A01).

B02: Consequences of ignoring Optimism's 7 day challenge period

[Severity: - | Difficulty: - | Category: Security]

Cross-domain transactions from Optimism to Ethereum usually take 7 days to complete due to the 7 day challenge period. However, when relying on trusted Optimism nodes like in the gateway protocol, this delay can be greatly reduced. Here, we take a look at the security implications of this approach.

Like any cross-domain operation, state unwinding is a multi-step process. It consists of:

1. Initiating the process on Optimism by calling `deposit()`, removing all traces of the KNOT (making the dETH associated with the KNOT unspendable on Optimism)
2. Retrieving a signed Merkle root from the router, which the router in turn fetches from some trusted Optimism node
3. Finalizing the unwinding by calling `push()` on Ethereum using the signed Merle root, and unlocking the dETH associated with the KNOT

This whole process can be completed before the 7 day challenge period for the Optimism transaction created in step 1 is over. Thus, if that transaction is successfully challenged and as a consequence of this re-executed, it is possible that the call to `push()` may fail the second time, which in turn means that the dETH associated with the KNOT remains spendable on Optimism. Since we assume that step 3 has already been completed, the same dETH is now spendable on both Ethereum *and* Optimism.

Note that most of the time, Optimism nodes are not affected by successful challenges because they execute the transactions published to the CTC (Canonical Transaction Chain) on their own and do not rely on the state from the SCC (State Commitment Chain) that is computed by the sequencer. However, because the state published to the SCC will become the official one once the 7 day challenge period is over, it must be ensured that the Optimism node used by the router reports the same state. There seem to be at least two cases where this might not be the case:

1. The Optimism node executes a transaction correctly, but the sequencer publishes a different state to the SCC and no verifier challenges this for 7 days.
2. The Optimism node executes a transaction incorrectly, either due to a malfunction or because of malicious activity.

The first case should not actually occur in practice because Optimism nodes are also verifiers that will challenge invalid state commitments. The second case cannot be fully prevented in general, but it can be mitigated to some degree by consulting multiple Optimism nodes and comparing the reported data (see A02).

Background Optimism is an optimistic rollup L2, which means that transactions processed by the Optimism sequencer are optimistically assumed to be valid. The sequencer publishes incoming transactions to the Canonical Transaction Chain (CTC), which is a contract on Ethereum. Additionally, for each transaction *T* in the CTC, the sequencer publishes a *state commitment* for *T* to another contract called the State Commitment Chain (SCC). A state commitment for a transaction is the Merkle root of the state that results from executing the transaction.

The sequencer is a centralized component and from a technical point of view, there is nothing stopping it from publishing an invalid state commitment to the SCC. To keep the sequencer honest, there is a mechanism in place to detect and report such invalid states. In particular, so-called *verifiers* continuously grab transactions from the CTC, execute them locally, and compare the resulting state with the one published to the SCC. If there is a mismatch, the verifier *challenges* the corresponding transaction by providing a fault proof. If the fault proof is correct, the sequencer is penalized for publishing an invalid state. Furthermore, the state commitments for the corresponding transaction and those following it are removed from the SCC and recomputed starting from a valid state.

After a state commitment has been published to the SCC, verifiers have a 7 day period, called the *challenge period*, to challenge that state commitment. After the challenge period is over, the transaction state is considered valid and becomes part of the official Optimism state.

B03: savETHGateway.forkDistance not meaningful on Optimism

[Severity: - | Difficulty: - | Category: Security]

To protect against reorgs, the router will not sign a deposit until a certain number of blocks have passed. The exact number of blocks to wait is retrieved from `sourceGateway.forkDistance`, where `sourceGateway` is the gateway to which the deposit has been made by calling `deposit()`.

If `sourceGateway` refers to the Ethereum gateway, then this is a robust approach, because `sourceGateway.forkDistance` is interpreted as a certain number n of Ethereum blocks. If n is chosen to be large enough, then it is ensured that the router only signs finalized deposits that can be considered immune against reorgs for most practical purposes.

On the other hand, if `sourceGateway` refers to the Optimism gateway, then `sourceGateway.forkDistance` is interpreted as a certain number of Optimism blocks. However, on Optimism, the number of L2 blocks that have been added after a transaction is not related to how safe that transaction is against reorgs. First, note that at the Optimism-level, there are actually no reorgs because once a transaction is added to the Canonical Transaction Chain (CTC), it cannot be changed or reordered anymore. However, since the CTC lives on Ethereum, it is affected by reorgs on Ethereum. Thus, if Ethereum is affected by a reorg, then a transaction previously added to the CTC may no longer be part of it, which also affects the state of Optimism. This means that a transaction on Optimism can only be considered final once the block in which it was added to the CTC on Ethereum is considered final. Or in other words: Finality on Optimism boils down to finality on Ethereum. For this reason, `sourceGateway.forkDistance` on Optimism cannot be used to protect against reorgs, especially because the block production rate on Optimism is non-constant and can thus not be related to the block production rate on Ethereum.

The reason this does not present a vulnerability is that Optimism mainnet is configured such that Ethereum reorgs that are at most 50 blocks deep have no effect on the state of Optimism.

Recommendation

The router should only sign deposits that are finalized on Ethereum, no matter whether the deposit was made on Ethereum or on Optimism. The `forkDistance` field is meant as a general approach to achieve this, but since Ethereum and Optimism each require special handling anyway, removing it may make the code clearer.

Status

Acknowledged. The client is investigating possible solutions.

B04: Router doesn't sign all DepositMetadata fields

[Severity: - | Difficulty: - | Category: Functional Correctness]

The router does not sign all the fields in the deposit metadata, namely the fields `originIndexId`, `accumulatorCount` and `gatewayRoot` are not part of the signature.

In the current context, where we do trust the router, this is not critical because the router will not sign a deposit that has not been made in the origin - we are under the assumption that the router only signs valid data. However, when going to a decentralized router, it's supposed that the endorsers agree on the Merkle root to attest that a deposit has been made.

```
// Hash the deposit metadata
bytes32 packetHash = sha256(
    abi.encode(
        _depositMetadata.stakeHouse,
        sha256(_depositMetadata.blsPublicKey),
        _depositMetadata.destinationIndexId,
        _depositMetadata.depositAmount,
        _depositMetadata.depositIndex,
        _depositMetadata.migratedIndexId,
        _depositMetadata.originGateway,
        _depositMetadata.originChainId,
        _depositMetadata.destinationGateway,
        _depositMetadata.destinationChainId
    )
);
```

Recommendation

Include the `gatewayRoot` in the signature of the router.

Status

Acknowledged. The goal is to only sign the Merkle root, since this is sufficient to ensure the correctness of the other fields. This also paves the way for replacing the router with a network of endorsers.

B05: Checks involving `is_valid_leaf` have no effect

[Severity: - | Difficulty: - | Category: Functional Correctness]

As a consequence of B04, the function `is_valid_leaf` becomes useless. It is possible, for instance, to create a `DepositMetadata` with a different `originIndexId`, providing a fake `gatewayRoot` and `_originAccumulatorProof` that would pass this check. This is not a security concern because the `originIndexId` is not being used for anything in the destination domain and because there are other checks ensuring that the same deposit cannot be pushed more than once. However, the purpose of having the mechanism of Merkle trees to extend and unwind the state becomes useless if the Merkle root cannot be trusted.

Also notice that the way the `originLeaf` is being built the additional checks on `_depositMetadata.originGateway`, `_depositMetadata.originChainId`, `_depositMetadata.originGateway` and `_depositMetadata.destinationChainId` are indeed necessary, otherwise it would be possible to construct Merkle roots and proofs for invalid data.

```
bytes32 originLeaf = sha256(
           abi.encode(
               _depositMetadata.stakeHouse,
               sha256(_depositMetadata.blsPublicKey),
               _depositMetadata.destinationIndexId,
               _depositMetadata.depositAmount,
               _depositMetadata.originIndexId,
               _depositMetadata.migratedIndexId,
               destinationGateway,
               forkId,
               address(this),
               block.chainid           )
       );
bool isValidLeafInOriginDomain = is_valid_leaf(
           originLeaf,
           _depositMetadata.depositIndex,
           _depositMetadata.accumulatorCount,
           _depositMetadata.gatewayRoot,
           _originAccumulatorProof
           );
```

Recommendation

Include the `gatewayRoot` in the signature of the router.

Status

Acknowledged. The client intends to fix the code and sign the Merkle root.

B06: `_depositMetadata.originGateway` is not being checked

[Severity: - | Difficulty: - | Category: Input Validation]

In the `push` function there is no check that `_depositMetadata.originGateway == destinationGateway`. This together with B04 and B05 makes it possible to construct a Merkle root and a proof for a deposit with the field `originGateway` referring to an arbitrary origin gateway. This means that `_depositMetadata.originGateway` cannot be trusted inside the `push()` function, which at the moment is not a security concern since it is not being read, but it can cause confusion in the future.

Recommendation

Add a sanity check:

```
if(_depositMetadata.originGateway != destinationGateway) revert InvalidOrigin;
```

Status

Addressed. (Note that code changes have not been audited.)

B07: Argument type of `_destinationIndexId`

[Severity: - | Difficulty: - | Category: Best practice]

The type of `_destinationIndexId` in the `deposit` function (line 140 of savETHGateway.sol) is `uint256`. Later we check that the argument is not greater than `type(uint64).max`.

```
function deposit(
       address _stakeHouse,
       bytes calldata _blsPublicKey,
       uint256 _destinationIndexId,
       bytes32 _deposit_data_root
   ) override external {
...
   if (_destinationIndexId == 0 || _destinationIndexId > type(uint64).max) revert
InvalidIndexId();
...
```

Recommendation

Make the argument `_destinationIndexId` of type `uint64`.

Status

Addressed. (Note that code changes have not been audited.)

B08: DepositMetadata.migratedIndexId is unnecessary

[Severity: - | Difficulty: - | Category: Optimization]

The struct field `DepositMetadata.migratedIndexId` is unnecessary and can be removed without affecting the functionality of the gateway.

Both `savETHGateway.push()` and `savETHDestinationGateway.balanceIncrease()` require that `DepositMetadata.migratedIndexId` equals `savETHGateway.forkIndex`, which refers to the savETH index owned by the gateway on the other domain. However, it is not clear what the goal of this requirement is. Since `forkIndex` is already known, there is no need to have this information also in `DepositMetadata.migratedIndexId`. The user would simply always have to set `migratedIndexId` to the same value as `forkIndex`, which does not seem useful.

Recommendation

Remove the field `DepositMetadata.migratedIndexId`.

Status

The client would like to keep `DepositMetadata.migratedIndexId`.

B09: savETHGatewaySignatureValidator.isNonceUsed is unnecessary

[Severity: - | Difficulty: - | Category: Optimization]

`savETHGatewaySignatureValidator` contains the following mapping:

```
mapping(uint256 => bool) public isNonceUsed;
```

This mapping is used to prevent users from pushing the same deposit multiple times: Whenever `savETHGateway.push()` or `savETHDestinationGateway.balanceIncrease()` is called for some deposit with index `depositIdx`, then it is first checked that `isNonceUsed[depositIdx]` is false. Afterwards, `isNonceUsed[depositIdx]` is set to true.

Further, `savETHGateway`, which derives from `savETHGatewaySignatureValidator`, contains the following mapping:

```
mapping(bytes32 => bool) public isTransactionPushed;
```

This mapping serves a similar purpose as `isNonceUsed`: Whenever `savETHGateway.push()` or `savETHDestinationGateway.balanceIncrease()` is called for some deposit with hash `depositHash`, then it is first checked that `isTransactionPushed[depositHash]` is false. Afterwards, `isTransactionPushed[depositHash]` is set to true.

Thus, the only difference between `isNonceUsed` and `isTransactionPushed` is that the former is based on the deposit index while the latter is based on the deposit hash. Since deposits have the same index if and only if they have the same hash (which follows from the requirement that the deposit has been inserted into the Merkle tree), both approaches are equivalent, which means one of them is redundant.

Note that `isTransactionPushed` is used in one other context: Namely by `savETHOriginGateway.pokeLatestBalance()` to ensure that the same balance cannot be poked multiple times. This does not affect the above reasoning, but means that `isTransactionPushed` strictly supersedes `isNonceUsed`.

Recommendation

Remove `savETHGatewaySignatureValidator.isNonceUsed`.

Status

Acknowledged. The client may remove `savETHGatewaySignatureValidator.isNonceUsed`.

B10: Logic for computing deposit hash is duplicated

[Severity: - | Difficulty: - | Category: Best practice]

In various places in the code, a hash is computed over all the data points that define a deposit. For example, the following code is used in `savETHGateway.deposit()`:

```
bytes32 node = sha256(
    abi.encode(
        _stakeHouse,             // Associated house for knot
        sha256(_blsPublicKey), // BLS pub key of the knot
        _destinationIndexId,    // destination chain savETH index ID
        depositAmount,          // balance queried from registry
        associatedIndexId,      // from this index
        migratedKnotsIndexId,   // to this frozen index
        address(this),          // from this gateway
        originId,               // on this origin EVM chain ID
        destinationGateway,     // to this destination gateway address
        forkId                  // on this fork id
    )
);
```

Here, `node` will store the sha256 hash of a deposit.

Hashes for deposits need to be computed in many places throughout the code, and each occurrence contains a snippet like the above. However, this makes the code harder to reason about for the following reasons:

- The way deposit hashes are computed is not uniform. This requires extra work to ensure that only compatible hashes are compared with each other.
- It is easy to forget to include certain data points in the hash, leading for example to Finding B06

Recommendation

Create a dedicated function that computes the hash for a deposit and use it throughout the codebase.

Status

Acknowledged. The client intends to reduce code duplication.

B11: Measuring signature deadline in blocks may not be suitable for Optimism

Signatures created by the router contain a deadline (`EIP712Signature.deadline`), which denotes the latest block number at which the signature is still valid. This is a valid approach on Ethereum, which has a predictable block production rate, but may not work as well on Optimism, where blocks are created on demand at a non-constant rate. In particular, each L2 transaction is put into its own L2 block, which means the rate at which L2 blocks are produced is equivalent to the transactions per second (TPS) that are processed by Optimism.

Since TPS may vary considerably, it may not be easy to choose a good deadline in advance without knowing the current TPS. An alternative could be to use `block.timestamp`, which should give more predictable results.

Recommendation

Instead of using a block number as the signature deadline, consider a data point that progresses at a more predictable rate on Optimism, like `block.timestamp`.

Status

Acknowledged. The client intends to investigate possible solutions.

B12: Inserting deposit into Merkle tree on push() is unnecessary and may cause loss of funds

[Severity: High | Difficulty: High | Category: Security]

Deposits are inserted into the gateway's Merkle tree not only when calling `deposit()` but also when calling `push()`. The latter is not necessary and may in fact cause users to lose access to their funds.

This follows from the deposit workflow, which is roughly as follows (see Overview):

1. Initiate a deposit by calling `gatewayA.deposit()` on the gateway on domain A. This inserts the deposit into the gateway's Merkle tree
2. Sign the deposit and the Merkle root from domain A using the router
3. Finalize the deposit by calling `gatewayB.push()`, passing in the signed deposit and Merkle root as proof that the deposit has indeed been initiated on domain A

Thus, the reason why the deposit is inserted into the Merkle tree when calling `deposit()` is because this is used to prove to the other domain that the deposit has indeed been initiated. On the other hand, there is never a need to prove to the other domain that a deposit has been finalized using `push()`. Hence, there is no need to insert pushed deposits into the Merkle tree. Not doing so would make the code easier to understand and would reduce gas costs.

Furthermore, there is the possibility that a user may lose access to some of their funds. This is because there is a maximum number of deposits that can be inserted in the Merkle tree. Thus, if the user successfully initiates a state unwinding for a KNOT on Optimism, but the Merkle tree of the gateway on Ethereum is already full, then the user is never able to finalize the state unwinding. This means the savETH associated with the KNOT will forever be held by the Ethereum gateway and there is no way for the user to retrieve it. However, note that this scenario is unlikely to occur in practice due to the very large gas costs involved in filling up the Merkle tree.

Recommendation

Remove the call to `_addTransactionHash()` (which inserts a deposit into the Merkle tree) from both `savETHRegistry.push()` and `savETHDestinationGateway.balanceIncrease()`.

Status

Acknowledged. The client will investigate further. TODO

B13: Filling up a Merkle tree makes the gateway non-functional

[Severity: High | Difficulty: High | Category: Security]

Once the Merkle tree of one of the gateway endpoints is full (i.e., contains $2^{32} - 1$ deposits), the gateway becomes non-functional. This means if either the Merkle tree of the Ethereum gateway or of the Optimism gateway is full, then state extensions, balance updates, and state unwindings become impossible. (Regarding state unwinding also see B12).

How likely is it that one of the Merkle trees actually reaches its full capacity of $2^{32} - 1$ deposits? If we only consider state extension and unwinding, and we assume that one of these operations is performed each second (which is highly unlikely, since there is cross-chain communication involved which adds network latency and block finalization latency), then this would still take over 100 years.

However, the balance update operation can be executed much more frequently, because `pokeLatestBalance()` can be called repeatedly even when the KNOT's dETH balance has not increased between the calls. Further, no cross-chain communication is necessary between the calls. Thus, the only limiting factor here is gas cost. Currently, these gas costs would make such a griefing attack close to impossible, and the attacker would not directly gain anything. However, it is not known how gas costs may develop in the future.

Recommendation

Requiring that `pokeLatestBalance()` can only be called if the KNOT's dETH has increased a certain amount would decouple the cost of the attack from the gas price, which would be more robust.

Status

Acknowledged. The client will investigate further.

Appendix 1: Formal Model

In this section we develop a formal model of the gateway protocol which allows us to formally state (and prove) important invariants about it.

Notation

We use $\mathbb{B} = \{true, \; false\}$ to denote the set of boolean values, $\mathbb{N}$ to denote the set of natural numbers (including zero), and $\mathbb{R}$ to denote the set of real numbers. Further, let $\mathbb{R}^+ = \{x \in \mathbb{R} \mid x \geq 0\}$.

If S is a set, we use $S_\perp$ to denote the set $S \cup \{\perp\}$, assuming $\perp \notin S$. Further, we use $|S|$ to denote the cardinality of S, and 2^S denotes the powerset of S

If A and B are sets, we use $A \rightharpoonup B$ to denote the set of partial functions from A to B.

We make use of the following notation for sums[3]: If $P(k)$ is any property on k, then we write $\sum_k f(k) \; [P(k)]$ instead of $\sum_{P(k)} f(k)$. For example, $\sum_{2 \leq k < 10} f(k)$ becomes $\sum_k f(k) \; [2 \leq k < 10]$. This makes handling complex conditions simpler and more readable.

State

The model makes use of various underspecified sets. On the Ethereum side, we use $Address$ to denote the set of Ethereum addresses, and $ChainID$ to denote the set of Ethereum chain IDs. On the Stakehouse side, we use elements of the sets $StakehouseID$ and $BLSPubKey$ to uniquely identify stakehouses and KNOTs, respectively. Further, we use $savETHIndexID$ to refer to the set of indices that a KNOT's savETH can be isolated to. We assume there is an element $OpenIndex \in savETHIndexID$ that denotes the open index (savETH in the open index is not in consideration for state extension).

Let $Knot$ denote the set of all KNOTs. For $k \in Knot$ we assume the following fields:

- $k.pk \in BLSPubKey$: The BLS public key that uniquely identifies the KNOT
- $k.stakehouse \in StakehouseID$: The ID of the stakehouse the KNOT belongs to
- $k.index \in savETHIndexID$: The savETH index that the KNOT is associated with
- $k.balance \in \mathbb{R}^+$: The mintable dETH balance associated with the KNOT in the savETH registry

[3] See Two notes on notation by Donald E. Knuth.

- $k.slashed \in \mathbb{B}$: Whether the validator associated with the KNOT has been slashed on the Beacon Chain
- $k.exited \in \mathbb{B}$: Whether the validator associated with the KNOT has exited the Beacon Chain

We use elements of the set $GatewayID = Address \times ChainID$ to uniquely identify a specific gateway. A gateway is identified by its contract address and the chain ID it has been deployed to.

Let $Transaction$ denote the set of transactions. (In the implementation, transactions represent leaves in the Merkle tree of the gateway contracts.) We assume that each transaction $t \in Transaction$ has the following fields:

- $t.txnIdx \in \mathbb{N}$: The transaction index, which together with $t.fromGateway$ (see below) uniquely identifies the transaction
- $t.pk \in BLSPubKey$: The BLS public key of the KNOT that is the subject of this transaction
- $t.stakehouse \in StakehouseID$: Stakehouse to which the KNOT belongs
- $t.amount \in \mathbb{R}^{+}$: mintable dETH balance associated with the KNOT at the time the transaction is made
- $t.fromIndex \in IndexID_{\perp}$: savETH index that has the mintable savETH at the time the transaction is made, or $\perp$ if the transaction is used to with the $ethPokeLatestBalance$ and $optBalanceIncrease$ operations
- $t.toIndex \in IndexID_{\perp}$: savETH index on the other domain into which the KNOT is transferred, or $\perp$ if the transaction is used to with the $ethPokeLatestBalance$ and $optBalanceIncrease$ operations
- $t.fromGateway \in GatewayID$: The ID of the gateway that has sent the transaction
- $t.toGateway \in GatewayID$: The ID of the gateway that is supposed to receive the transaction

Transactions contain all the information needed to perform a state extension, a balance update, or a state unwinding. A transaction t is classified as follows:

- If $t.fromGateway$ refers to the Ethereum gateway and $t.toGateway$ refers to the Optimsm gateway, and $t.fromIndex \neq \perp$ and $t.toIndex \neq \perp$, then t represents a state extension and we call it a extension transaction
- If $t.fromGateway$ refers to the Ethereum gateway and $t.toGateway$ refers to the Optimsm gateway, and $t.fromIndex = t.toIndex = \perp$, then t represents a balance update and we call it a update transaction
- If $t.fromGateway$ refers to the Optimism gateway and $t.toGateway$ refers to the Ethereum gateway, then t represents a state unwinding and we call it a unwinding transaction

Definition: To distinguish extension transactions and update transactions, we use the following predicates:
$isExtention(t) \equiv t.toIndex \neq \bot \wedge t.fromIndex \neq \bot$
$isUpdate(t) \equiv t.toIndex = \bot \wedge t.fromIndex = \bot$

Let $State$ denote the set of gateway states for a single domain. This means each $s \in State$ captures the complete state of either the `savETHOriginGateway` or the `savETHDestinationGateway` contract. To this end, we assume the following accessors for s:

- Accessors related to the Stakehouse registry:
  - $knots_s \in 2^{Knot}$: Set of all KNOTs across all stakehouses.
  - $indexOwner_s \in (savETHIndexID \rightharpoonup Address)$: Maps savETH indexes to the address of their owner
- Accessors related to the gateway:
  - $gatewayAddress_s \in Address$: The address to which the gateway has been deployed
  - $chainID_s \in ChainID$: The chain ID to which the gateway has been deployed
  - $gatewayIndex_s \in savETHIndexID$: The savETH index that is owned by the gateway and that contains those KNOT's whose savETH state is currently extended
  - $txns_s \in 2^{Transaction}$: Contains the transactions created by the deposit() and pokeLatestBalance() functions. Corresponds to the leafs of the Merkle tree computed by the savETHGateway
  - $pushed_s \in 2^{Transaction}$: Corresponds to savETHGateway.isTransactionPushed
  - $used_s \in 2^{Transaction}$: Corresponds to savETHGatewaySignatureValidator.isNonceUsed

We use the notation $knots_s[pk]$ to denote the unique KNOT k such that $k \in knots_s$ and $k.pk = pk$.

We define the following derived state accessors for s:

- $pks_s = \{k.pk \mid k \in knots_s\}$
- $gatewayID_s = (gatewayAddress_s, chainID_s)$
- $txnsOut_s = \{t \mid t \in txns_s \wedge t.fromGateway = gatewayID_s\}$ (all outgoing transactions)
- $pushedIn_s = \{t \mid t \in pushed_s \wedge t.toGateway = gatewayID_s\}$ (all incoming transactions that have been pushed)

Our model of the Blockswap gateway protocol consists of two states $e, o \in State$ that represent the respective state on Ethereum and Optimism, respectively.

We now define the initial state. If $g \in \{e,\ o\}$, then initially, the following conditions must be satisfied:

- $gatewayIndex_g \neq OpenIndex$
- $txns_g = \emptyset$
- $pushed_g = \emptyset$
- $used_g = \emptyset$
- $gatewayID_e \neq gatewayID_o$

The requirements for $knots_g$ and $indexOwner_g$ are specified in the following section Assumptions.

Assumptions

The model described here only covers the gateway protocol, not the Stakehouse protocol. Nevertheless, in order to prove the main invariant (see Main Invariant), we need to make some assumptions about how the Stakehouse protocol works. In particular, we need to make assumptions about $knots_g$ and $indexOwner_g$. In the following, it is assumed that g ranges over both the Ethereum state e and the Optimism state o.

Assumption 1: For all $k, k' \in knots_g$, if $k.pk = k'.pk$ then $k = k'$ (a KNOT is uniquely identified by its BLS public key).

Assumption 2: All KNOT balances are monotonically increasing.

Assumption 2 implies that if $t_1, t_2 \in txnsOut_e$ and $t_1.txnIdx \leq t_2.txnIdx$, then $t_1.amount \leq t_2.amount$, and vice versa.

Operations

We model the following operations:

- $ethDeposit(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHOriginGateway.deposit()`.

- $ethPush(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHOriginGateway.push()`.

- $ethPokeLatestBalance(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHOriginGateway.pokeLatestBalance()`.

- $optDeposit(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHDestinationGateway.deposit()`.

- $optPush(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHDestinationGateway.push()`.

- $optBalanceIncrease(sender,\ t)$, where $sender \in Address$, $t \in Transaction$.
  Corresponds to `savETHDestinationGateway.balanceIncrease()`.

These operations are supposed to be used in pairs: one to initiate the transaction on the origin domain and one to finalize the transaction on the destination domain. The following pairs are supported:

- $ethDeposit \rightarrow optPush$: Called state extension. Locks the savETH associated with a KNOT on Ethereum and makes its dETH available on Optimism.

- $optDeposit \rightarrow ethPush$: Called state unwinding. Removes a KNOT's savETH from Optimism by rage quitting the state and makes its dETH available on Ethereum again.

- $ethPokeLatestBalance \rightarrow optBalanceIncrease$: Makes the dETH rewards earned by a KNOT available on Optimism. Requires that state extension has already been performed for the KNOT.

When defining the behavior of each operation, we use e and o to denote the respective domain states before the operation has been executed, and e' and o' to denote the states after the execution. If we don't explicitly define the value of a property in the post state, it is assumed to not have changed.

For each operation, we specify the preconditions that must be satisfied before the operation can be executed, and the postconditions that the operation will satisfy. Furthermore, some operations have a *cross-domain precondition*. For example, before you can push an extension transaction on Optimism, the transaction must have been created on Ethereum. In the actual implementation of the gateway protocol, these kinds of cross-domain requirements are checked with the help of the off-chain router (which will be replaced with a network of endorsers in a future version). In our model, we instead directly state the requirements that are enforced by the router as a cross-domain precondition, which allows us to omit the router in our model.

In the actual implementation, the Merkle tree that is used to store transactions has a maximum capacity of $2^{32} - 1$ transactions. To reflect this in the model, each operation requires that the maximum number of transactions has not been reached yet. To this end, we introduce the constant $MAX\_TXNS = 2^{32} - 1$.

ethDeposit

Operation: $ethDeposit(sender,\ t)$, where $sender \in Address$, $t \in Transaction$

Preconditions:

- $|txns_e| < MAX\_TXNS$
- $t.txnIdx = |txns_e|$ (implies $t \notin txns_e$, see Invariant 5.1)
- $t.fromGateway = gatewayID_e$
- $t.toGateway = gatewayID_o$
- $t.pk \in pks_e$
- $t.fromIndex = k.index$
- $t.toIndex \neq OpenIndex \wedge t.toIndex \neq \bot$
- $t.amount = k.balance$
- $indexOwner_e(k.index) = sender$ (to protect against front running)
- $k.index \neq gatewayIndex_e$
- $k.stakehouse = t.stakehouse$
- $k.balance \geq 24$
- $k.slashed =$ false
- $k.exited =$ false

where $k = knots_e[t.pk]$.

Postconsitions:

- $knots_{e'}[t.pk].index = gatewayIndex_e$
- $txns_{e'} = txns_e \cup \{t\}$

ethPokeLatestBalance

Operation: $ethPokeLatestBalance(sender,\ t)$, where $sender \in Address$, $t \in Transaction$

Preconditions:

- $|txns_e| < MAX\_TXNS$
- $t.txnIdx = |txns_e|$ (implies $t \notin txns_e$, see Invariant 5.1)
- $t.fromGateway = gatewayID_e$
- $t.toGateway = gatewayID_o$
- $t.pk \in pks_e$

- $t.fromIndex = \bot$
- $t.toIndex = \bot$
- $t.amount = k.balance$
- $k.balance > 24$
- $k.index = gatewayIndex_e$
- $k.stakehouse = t.stakehouse$
- $k.slashed =$ false
- $k.exited =$ false

where $k = knots_e[t.pk]$.

Postconsitions:

- $txns_{e'} = txns_e \cup \{t\}$
- $pushed_{e'} = pushed_e \cup \{t[txnIdx \leftarrow n] \mid n \in \mathbb{N}\}$ (add a transaction for every transaction index so that users cannot poke the same balance twice. We use the notation $t[txnIdx \leftarrow n]$ to denote the transaction t' that is equal to t except that $t'.txnIdx = n$)

ethPush

Operation: $ethPush(sender, t)$, where $sender \in Address$, $t \in Transaction$

Cross-domain preconditions:

- $t \in txns_o$

Preconditions:

- $|txns_e| < MAX\_TXNS$
- $t \notin pushed_e$
- $t \notin used_e$ (seems redundant because of invariant $used_e \subseteq pushed_e$)
- $t.fromGateway = gatewayID_o$ (currently implied by $t \in txns_o$ because there is only one other gateway)
- $t.toGateway = gatewayID_e$ (currently implied by $t \in txns_e$ because there is only one other gateway)
- $indexOwner_e(t.toIndex) = sender$
- $t.pk \in pks_e$
- $t.amount \leq knots_e[t.pk].balance$
- $knots_e[t.pk].index = gatewayIndex_e$

Postconditions:

- $txns_{e'} = txns_e \cup \{t\}$

- $pushed_{e'} = pushed_e \cup \{t\}$
- $used_{e'} = used_e \cup \{t\}$
- $knots_{e'}[t.pk].index = t.toIndex$

optDeposit

Operation: $optDeposit(sender,\ t)$, where $sender \in Address$, $t \in Transaction$

Preconditions:

- $|txns_o| < MAX\_TXNS$
- $t.txnIdx = |txns_o|$ (implies $t \notin txns_o$, see Invariant 5.2)
- $t.fromGateway = gatewayID_o$
- $t.toGateway = gatewayID_e$
- $t.pk \in pks_o$
- $t.fromIndex = knots_o[t.pk].index$
- $indexOwner_o(t.fromIndex) = sender$ (to protect against front running)
- $t.toIndex \neq OpenIndex \wedge t.toIndex \neq \perp$
- $t.amount = knots_o[t.pk].balance$
- $knots_o[t.pk].stakehouse = t.stakehouse$

Postconsitions:

- $knots_{o'} = knots_o \setminus \{knots_o[t.pk]\}$
- $txns_{o'} = txns_o \cup \{t\}$

optBalanceIncrease

Operation: $optBalanceIncrease(sender,\ t)$, where $sender \in Address$, $t \in Transaction$

Cross-domain preconditions:

- $t \in txns_e$

Preconditions:

- $|txns_o| < MAX\_TXNS$
- $t \notin pushed_o$
- $t \notin used_o$ (seems redundant because of invariant $used_o \subseteq pushed_o$)
- $t.fromGateway = gatewayID_e$ (currently implied by $t \in txns_e$ because there is only one other gateway)

- $t.toGateway = gatewayID_o$ (currently implied by $t \in txns_e$ because there is only one other gateway)
- $t.pk \in pks_o$
- $isUpdate(t)$ (make sure transaction was created by the $ethPokeLatestBalance$ operation)

Postconditions:

- $txns_{o'} = txns_o \cup \{t\}$
- $pushed_{o'} = pushed_o \cup \{t\}$
- $used_{o'} = used_o \cup \{t\}$
- $knots_{o'}[t.pk].balance = max(t.amount, knots_o[t.pk].balance)$

optPush

Operation: $optPush(sender, t)$, where $sender \in Address$, $t \in Transaction$

Cross-domain preconditions:

- $t \in txns_e$

Preconditions:

- $|txns_o| < MAX\_TXNS$
- $t \notin pushed_o$
- $t \notin used_o$ (seems redundant because of invariant $used_o \subseteq pushed_o$)
- $t.fromGateway = gatewayID_e$ (currently implied by $t \in txns_e$ because there is only one other gateway)
- $t.toGateway = gatewayID_o$ (currently implied by $t \in txns_e$ because there is only one other gateway)
- $isExtension(t)$
- $indexOwner_o(t.toIndex) = sender$
- $t.pk \notin pks_o$

Postconditions:

- $txns_{o'} = txns_o \cup \{t\}$
- $pushed_{o'} = pushed_o \cup \{t\}$
- $used_{o'} = used_o \cup \{t\}$
- $knots_{o'}[t.pk] = k$, where
    - $k.stakehouse = t.stakehouse$
    - $k.index = t.toIndex$
    - $k.balance = t.amount$

    - $k.slashed = \text{false}$
    - $k.exited = \text{false}$
- $stakehouses_{o'} = stakehouses_o \cup \{t.stakehouse\}$

Main Invariant

The main invariant that we want to state is that the gateway operations do not affect the total supply of dETH across Ethereum and Optimism. To this end, we need a couple of definitions.

Definition: For $s \in State$, we define $spendableDETH_s$ such that

$$spendableDETH_s = \sum_{pk} spendableDETHFor(pk)\ [pk \in BLSPubKey]$$

where

- $spenableDETHFor_s(pk) = k.balance$ if there exists a $k \in knots_s$ with $k.pk = pk$ and $k.index \neq gatewayIndex_s$.

$spenableDETHFor_s(pk) = 0$ otherwise.

Intuitively, dETH is spendable if the KNOT it is associated with is not part of the gateway index.

Definition: The amount of dETH for which state extension has been initiated on Ethereum (by calling deposit() or pokeLatestBalance()) but that has not yet been finalized on Optimism (i.e., neither push() nor increaseBalance() has been called) is given by

$$notPushedToOptimism = \sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_e]$$

where

- $unpushedDETHFor(pk) = highestTxn(pk) - highestPushed(pk)$

  For a KNOT pk, the amount of unpushed dETH is the difference between the highest transaction made on Ethereum (given by $highestTxn(pk)$) and the highest transaction actually pushed to Optimism (given by $highestPushed(pk)$)

- $highestTxn(pk) = 0$ if $T = \emptyset$
  $highestTxn(pk) = max\{t.amount \mid t \in T\}$ otherwise

  with $T = currentTxnsFor(pk)$

- $highestPushed(pk) = 0 \quad$ if $T \cap pushedIn_o = \emptyset$

  $highestPushed(pk) = max\{t.amount \mid t \in T \cap pushedIn_o\}$ otherwise

  with $T = currentTxnsFor(pk)$

- $currentTxnsFor(pk) = \{t \in Transaction \mid isCurrentTxn(t) \wedge t.pk = pk\}$

- $isCurrentTxn(t)$ is true if and only if $t \in txnsOut_e$ and there exists a t' such that $isLatestExtension(t')$ and $t.txnIdx \geq t'.txnIdx$

- $isLatestExtension(t)$ is true if and only if $t \in txnsOut_e$, $isExtension(t)$, and $knots_e[t.pk].index = gatewayIndex_e$, and there exists no $t' \in txnsOut_e$ such that $isExtension(t')$, $t'.pk = t.pk$, and $t'.txnIdx > t.txnIdx$

Definition: The amount of dETH for which state unwinding has been initiated on Optimism (by calling deposit()) but that has not yet been finalized on Ethereum (i.e., push() has not been called yet) is given by

$$notPushedToEthereum = \sum_{pk} unwindingAmount(pk)\ [pk \in BLSPubKey]$$

where

- $unwindingAmount(pk) = t.amount$ if there exists a unique $t \in txnsOut_o \setminus pushedIn_e$ with $t.pk = pk$.

  $unwindingAmount(pk) = 0$ otherwise.

Definition: The total amount of unpushed dETH is simply the sum of the unpushed dETH to Ethereum and to Optimism:

$$unpushedDETH = notPushedToOptimism + notPushedToEthereum$$

Definition: Amount of dETH rewards that have been earned by KNOTs that are part of the Ethereum gateway index, and that have not yet been sent to Optimism.

$$totalGatewayRewards = \sum_{pk \in BLSPubKey} gatewayRewardsFor(pk)$$

where

- $gatewayRewardsFor(pk) = k.balance - highestTxn(pk)$
  if $pk \in pks_e$, $k = knots_e[pk]$ and

  $gatewayRewardsFor(pk) = 0$ otherwise

Main Invariant: There exists a $C \in \mathbb{R}^{+}$ such that the following condition is an invariant:

$$spendableDETH_e + spendableDETH_o + unpushedDETH + totalGatewayRewards = C$$

The above invariant accounts for every dETH token in existence, ensuring that no dETH is lost or created out of thin air. All dETH is either

- *spendable on Ethereum*, which means it is associated with a KNOT that is either in the open index or in a user-owned index on Ethereum,
- *spendable on Optimism*, which means it is associated with a KNOT that is either in the open index or in a user-owned index on Optimism,
- *unpushed*, which means it has been deposited on one gateway but not yet pushed to the other, or
- *locked as rewards in the Ethereum gateway index*, which means that a user has reported Beacon Chain rewards for a state-extended KNOT but has not transferred these rewards to Optimism yet.

A proof for the main invariant is provided in the following section Appendix 2: Proofs.

Appendix 2: Proofs

The goal of this section is to provide a rough proof of the Main Invariant. To this end, we need a couple of helper invariants.

Invariant 1

Invariant 1: (Every pushed transaction has previously been deposited on the other chain)

$$\text{(Inv.1) } pushedIn_o \subseteq txnsOut_e \text{ and (Inv.2) } pushedIn_e \subseteq txnsOut_o$$

We show that the above invariant holds for each operation. We use Inv.1 and Inv.2 to refer to the subformulas of the invariant in the pre-state, and Inv.1' and Inv.2' to refer to them in the post-state.

- $ethDeposit(sender,\ t)$:

  Regarding Inv.1: The postconditions imply $pushedIn_{o'} = pushedIn_o$ and $txnsOut_e \subseteq txnsOut_{e'}$. This together with Inv.1 implies Inv.1'.

  Regarding Inv.2: Nothing relevant changes, so Inv.2' directly follows from Inv.2.

- $ethPokeLatestBalance(sender,\ t)$:

  Regarding Inv.1: The postconditions imply $pushedIn_{o'} = pushedIn_o$ and $txnsOut_e \subseteq txnsOut_{e'}$. This together with Inv.1 implies Inv.1'.

  Regarding Inv.2: First, the postcondition implies $txnsOut_{o'} = txnsOut_{o'}$.
  Further, The postcondition implies $pushed_{e'} = pushed_e \cup \{t[txnIdx \leftarrow n] \mid n \in \mathbb{N}\}$. Note that for all the transactions t' that are newly added to $pushed_{e'}$ we have $t.toGateway = gatewayID_o \neq gatewayID_e$, which implies $pushedIn_{e'} = pushedIn_e$. Thus, we have both $txnsOut_{o'} = txnsOut_o$ and $pushedIn_{e'} = pushedIn_e$ which together with Inv.2 implies Inv.2'.

- $ethPush(sender,\ t)$:

  Regarding Inv.1: The postconditions imply $txnsOut_e \subseteq txnsOut_{e'}$ and $pushedIn_o = pushedIn_{o'}$ which together with Inv.1 gives us Inv.1'.

Regarding Inv.2: We rewrite Inv.2 until we get Inv.2'.

$pushedIn_e \subseteq txnsOut_o$

$\Rightarrow^{(a)} pushedIn_e \cup \{t\} \subseteq txnsOut_o$

$\Rightarrow^{(b)} pushedIn_{e'} \subseteq txnsOut_o$

$\Rightarrow^{(c)} pushedIn_{e'} \subseteq txnsOut_{o'}$

(a) The cross-domain precondition gives us $t \in txns_o$ which together with $t.fromGateway = gatewayID_o$ implies $t \in txnsOut_o$

(b) The postcondition states that $pushed_{e'} = pushed_e \cup \{t\}$. This together with $t.toGateway = gatewayID_e$ implies $pushedIn_{e'} = pushedIn_e \cup \{t\}$.

(c) Follows from $txnsOut_{o'} = txnsOut_o$

- $optDeposit(sender, t)$:

  Regarding Inv.1: Nothing relevant changes, so Inv.1' directly follows from Inv.1.

  Regarding Inv.2: The postconditions imply $pushedIn_{e'} = pushedIn_e$ and $txnsOut_{o'} \supseteq txnsOut_o$. This together with Inv.2 implies Inv.2'.

- $optPush(sender, t)$:

  Regarding Inv.1: We rewrite Inv.1 until we get Inv.1'.

$pushedIn_o \subseteq txnsOut_e$

$\Rightarrow^{(a)} pushedIn_o \cup \{t\} \subseteq txnsOut_e$

$\Rightarrow^{(b)} pushedIn_{o'} \subseteq txnsOut_e$

$\Rightarrow^{(c)} pushedIn_{o'} \subseteq txnsOut_{e'}$

(a) The cross-domain precondition gives us $t \in txns_e$ which together with $t.fromGateway = gatewayID_e$ implies $t \in txnsOut_e$

(b) The postcondition states that $pushed_{o'} = pushed_o \cup \{t\}$. This together with $t.toGateway = gatewayID_o$ implies $pushedIn_{o'} = pushedIn_o \cup \{t\}$.

(c) Follows from $txnsOut_{e'} = txnsOut_e$

Regarding Inv.2: The postconditions imply $txnsOut_o \subseteq txnsOut_{o'}$ and $pushedIn_e = pushedIn_{e'}$ which together with Inv.2 gives us Inv.2'.

- $optBalanceIncrease(sender,\ t)$:

  Same reasoning as for $optPush$.

Invariant 2

Invariant 2: For each $pk \in BLSPubKey$, the KNOT is in exactly one of the following states: unwound, during a state extension, extended, or during a state unwinding. For each state we define a predicate as follows:

- $unwound(pk) \equiv knots_e[pk].index \neq gatewayIndex_e$
- $duringExtension(pk) \equiv \exists t.\ t \in txnsOut_e \setminus pushedIn_o \wedge t.pk = pk \wedge isExtension(t)$
- $extended(pk) \equiv pk \in pks_o$
- $duringUnwinding(pk) \equiv \exists t.\ t \in txnsOut_o \setminus pushedIn_e \wedge t.pk = pk$

Let *Inv* denote the above invariant in the pre-state and *Inv'* in the post-state. Similarly, we use $unwound(pk)$, $duringExtension(pk)$, $extended(pk)$ and $duringUnwinding(pk)$ to denote the respective state predicate in the pre-state, and use primed variants to denote them in the post-state.

We show that the above invariant holds for each operation. To this end, let $pk \in BLSPubKey$ be arbitrary but fixed. We can assume that exactly one of the state predicates holds in the pre-state. It remains to show that exactly one of them holds in the post-state.

- $ethDeposit(sender,\ t)$
  - Assume $pk = t.pk$. Then, by the precondition of $ethDeposit$, we know $knots_e[pk].index) \neq gatewayIndex_e$. Thus, $unwound(pk)$ is true, and because we assume *Inv* holds, it follows that all other state predicates are false in the pre-state.

    Next, we show that exactly one state predicate is true in the post-state, which concludes the proof.

    - $unwound'(pk)$ is false. This follows from $knots_{e'}[t.pk].index = gatewayIndex_e$
    - $duringExtension'(pk)$ is true. First, note that $t \notin txns_e$ implies $t \notin pushedIn_o$ because of Invariant 1, and because $pushedIn_{o'} = pushedIn_o$ this also implies $t \notin pushedIn_{o'}$. Next, observe that according to the postcondition we have $txns_{e'} = txns_e \cup \{t\}$, which implies $t \in txnsOut_{e'}$.

Thus, we get $t \in txnsOut_{e'} \setminus pushedIn_{o'}$, which together with $t.pk = pk$ implies that $duringExtension'(pk)$ is true.

- $extended'(pk)$ is false. Follows from $extended(pk)$ being false and the fact that $pks_{o'} = pks_{o}$
- $duringUnwinding'(pk)$ is false. Follows from $duringUnwinding(pk)$ being false and the fact that $txnsOut_{o'} = txnsOut_{o}$ and $pushedIn_{e'} = pushedIn_{e}$

- Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.

- $ethPokeLatestBalance(sender, t)$
  - Assume $pk = t.pk$. Then because of the precondition $knots_{e}[pk].index = gatewayIndex_{e}$ we know $unwound(pk)$ is false. This means one of the remaining state predicates must be true. We make a case distinction on which of them is true.
    - Case $duringExtension(pk)$ is true: Then $duringExtension'(pk)$ is also true. The other state predicates remain false since nothing relevant to them changes. Thus, *Inv'* holds and we are done.
    - Case $extended(pk)$ is true: Then $extended(pk)$ is also true because $knots_{o'} = knots_{o}$. The other state predicates remain false since nothing relevant to them changes. (Note that adding t to $txns_{e}$ has not effect on $duringExtension(pk)$ since $t.toIndex = \bot$). Thus, *Inv'* holds and we are done.
    - Case $duringUnwinding(pk)$ is true: Then $duringUnwinding(pk)$ is also true because $txnsOut_{o'} = txnsOut_{o}$ and $pushedIn_{e'} = pushedIn_{e}$. The other state predicates remain false since nothing relevant to them changes. Thus, *Inv'* holds and we are done.
  - Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.
- $ethPush(sender, t)$
  - Assume $pk = t.pk$. Then the preconditions imply $duringUnwinding(pk)$, and because we assume *Inv* holds, it follows that all other state predicates are false in the pre-state.

    Next, we show that exactly one state predicate is true in the post-state, which concludes the proof.

- $unwound'(pk)$ is true. This follows from $knots_{e'}[t.pk].index = t.toIndex$ and the fact that $t.toIndex \neq gatewayIndex_e$. Regarding the latter, observe that the precondition gives us $indexOwner_e(t.toIndex) = sender$. Because the sender cannot be the gateway (i.e., $sender \neq gatewayAddress_e$), we know $indexOwner_e(t.toIndex) \neq gatewayAddress_e$. This implies $t.toIndex \neq gatewayIndex_e$ because $indexOwner_e(gatewayIndex_e) = gatewayAddress_e$.

- $duringExtension'(pk)$ is false. Follows from $duringExtension(pk)$ being false and the fact that $txnsOut_{e'} = txnsOut_e$ and $pushedIn_{o'} = pushedIn_o$

- $extended'(pk)$ is false. Follows from $extended(pk)$ being false and the fact that $pks_{o'} = pks_o$

- $duringUnwinding'(pk)$ is false. Follows from $t \in pushedIn_{e'}$ which implies $t \notin txnsOut_{o'} \setminus pushedIn_{e'}$.

- Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.

- $optDeposit(sender,\ t)$
  - Assume $pk = t.pk$. Then $t.pk \in pks_o$ implies $extended(pk)$, and because we assume *Inv* holds, it follows that all other state predicates are false in the pre-state.

    Next, we show that exactly one state predicate is true in the post-state, which concludes the proof.

    - $unwound'(pk)$ is false. This follows from $unwound(pk)$ being false and $knots_{e'}[pk].index = knots_e[pk].index$

    - $duringExtension'(pk)$ is false. Follows from $duringExtension(pk)$ being false and the fact that $txnsOut_{e'} = txnsOut_e$ and $pushedIn_{o'} = pushedIn_o$

    - $extended'(pk)$ is false. Follows from postcondition $knots_{o'} = knots_o \setminus \{knots_o[t.pk]\}$

    - $duringUnwinding'(pk)$ is true. First, note that $t \notin txns_o$ implies $t \notin pushedIn_e$ because of Invariant 1, and because $pushedIn_{e'} = pushedIn_e$ this also implies $t \notin pushedIn_{e'}$. Next, observe that according to the postcondition we have $txns_{o'} = txns_o \cup \{t\}$, which implies $t \in txnsOut_{o'}$.

Thus, we get $t \in txnsOut_{o'} \setminus pushedIn_{e'}$, which together with $t.pk = pk$ implies that $duringUnwinding'(pk)$ is true.

- Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.

- $optPush(sender,\ t)$
  - Assume $pk = t.pk$. Then the preconditions imply $duringExtension(pk)$, and because we assume *Inv* holds, it follows that all other state predicates are false in the pre-state.

    Next, we show that exactly one state predicate is true in the post-state, which concludes the proof.

    - $unwound'(pk)$ is false. This follows from $unwound(pk)$ being false and $knots_{e'}[pk].index = knots_{e}[pk].index$
    - $duringExtension'(pk)$ is false. Follows from $t \in pushedIn_{o'}$ which implies $t \notin txnsOut_{e'} \setminus pushedIn_{o'}$.
    - $extended'(pk)$ is true. Follows from postcondition $knots_{o'}[t.pk] = k$
    - $duringUnwinding'(pk)$ is false. Follows from $duringUnwinding(pk)$ being false and the fact that $txnsOut_{o'} = txnsOut_{o}$ and $pushedIn_{e'} = pushedIn_{e}$
  - Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.

- $optBalanceIncrease(sender,\ t)$
  - Assume $pk = t.pk$. Then the preconditions imply $extended(pk)$, and because we assume *Inv* holds, it follows that all other state predicates are false in the pre-state.

    Next, we show that exactly one state predicate is true in the post-state, which concludes the proof.

    - $unwound'(pk)$ is false. This follows from $unwound(pk)$ being false and $knots_{e'}[pk].index = knots_{e}[pk].index$
    - $duringExtension'(pk)$ is false. Follows from $duringExtension(pk)$ being false and $pushedIn_{o} \subseteq pushedIn_{o'}$
    - $extended'(pk)$ is true. Follows from $extended(pk)$ and $pks_{o'} = pks_{o}$

- $durningUnwinding'(pk)$ is false. Follows from $duringUnwinding(pk)$ being false and the fact that $txnsOut_{o'} = txnsOut_o$

- Now assume $pk \neq t.pk$. Then the state of the KNOT identified by pk does not change, and *Inv'* follows directly from *Inv*.

Invariant 3

This invariant states that for each KNOT, there can be at most one unpushed transaction on its way from Optimism to Ethereum (i.e., only one state unwinding can be initiated at any given time).

Invariant 3: If $t_1, t_2 \in txnsOut_o \setminus pushedIn_e$ such that $t_1.pk = t_2.pk$, then $t_1 = t_2$

We show that the above invariant holds for each operation. We use *Inv* to refer to the above invariant in the pre-state, and *Inv'* to refer to it in the post-state.

- $ethDeposit(sender,\ t)$: Nothing relevant changes.
- $ethPokeLatestBalance(sender,\ t)$: Nothing relevant changes.
- $ethPush(sender,\ t)$:

  The postconditions imply $txnsOut_{o'} = txnsOut_o$ and $pushedIn_{e'} \supseteq pushedIn_e$. This gives us $txnsOut_{o'} \setminus pushedIn_{e'} \subseteq txnsOut_o \setminus pushedIn_e$ which together with *Inv* implies *Inv'*.

- $optDeposit(sender,\ t)$:

  The precondition $t.pk \in pks_o$ together with invariant 2 implies that there exists no unpushed transaction $t' \in txnsOut_o \setminus pushedIn_e$ with $t'.pk = t.pk$. Thus, adding t to $txnsOut_o$ means that there is exactly one unpushed transaction for $t.pk$ in the post-state. This together with *Inv* implies *Inv'*.

- $optBalanceIncrease(sender,\ t)$: Nothing relevant changes.
- $optPush(sender,\ t)$: Nothing relevant changes.

Invariant 4

Invariant 4: For all $pk \in BLSPubKey$:

$$duringUnwinding(pk) \Rightarrow highestPushed(pk) = unwindingAmount(pk)$$

We show that Invariant 4 holds for each operation. We use *Inv* to refer to Invariant 4 in the pre-state, and *Inv'* to refer to it in the post-state.

In the following, let $pk \in BLSPubKey$ be arbitrary such that $duringUnwinding'(pk)$. We need to show $highestPushed'(pk) = unwindingAmount'(pk)$ for each operation in order to prove *Inv'*.

- $ethDeposit(sender, t)$:

  First, assume $pk = t.pk$. Then the postconditions imply $duringExtension'(pk)$, which together with our assumption $duringUnwinding'(pk)$ and Invariant 2 leads to a contradiction. Thus, there is nothing to show.

  Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

- $ethPokeLatestBalance(sender, t)$:

  First, assume $pk = t.pk$. Then, with the help of Invariant 1, we can show that $t \notin pushedIn_{o'}$ which in turn implies $highestPushed'(pk) = highestPushed(pk)$. Further, nothing relevant to $unwindingAmount$ changes, hence $unwindingAmount'(pk) = unwindingAmount(pk)$. Since neither $highestPushed'(pk)$ nor $unwindingAmount'(pk)$ changes compared to the pre-state, *Inv'* follows directly from *Inv*.

  Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

- $ethPush(sender, t)$:

  First, assume $pk = t.pk$. Then the postconditions imply $unwound'(pk)$, which together with our assumption $duringUnwinding'(pk)$ and Invariant 2 leads to a contradiction. Thus, there is nothing to show.

  Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

- $optDeposit(sender, t)$:

  First, assume $pk = t.pk$.

  - Then the postconditions imply $highestPushed'(pk) = highestPushed(pk)$. This, together with Invariant 4.1, implies $highestPushed'(pk) = knots_o[pk].balance$.
  - Further, the postconditions together with Invariant 1 imply $t \in txnsOut_{o'} \setminus pushedIn_{e'}$. By Invariant 3 we know that there exists only one such transaction for pk, which gives us $unwindingAmount'(pk) = knots_o[pk].balance$.

The above implies *Inv'*.

Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

- $optBalanceIncrease(sender, t)$:

  First, assume $pk = t.pk$. Then the postconditions imply $extended'(pk)$, which together with our assumption $duringUnwinding'(pk)$ and Invariant 2 leads to a contradiction. Thus, there is nothing to show.

  Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

- $optPush(sender, t)$:

  First, assume $pk = t.pk$. Then the postconditions imply $extended'(pk)$, which together with our assumption $duringUnwinding'(pk)$ and Invariant 2 leads to a contradiction. Thus, there is nothing to show.

  Now assume $pk \neq t.pk$. Then, $highestPushed'(pk) = highestPushed(pk)$ and $unwindingAmount'(pk) = unwindingAmount(pk)$. Thus, *Inv'* follows from *Inv*.

Invariant 4.1: For all $k \in knots_o$: $highestPushed(k.pk) = k.balance$

We show that Invariant 4.1 holds for each operation. We use *Inv* to refer to Invariant 4.1 in the pre-state, and *Inv'* to refer to it in the post-state.

In the following, let $k' \in knots_{o'}$ denote an arbitrary KNOT in the post-state. We need to show $highestPushed'(k'.pk) = k'.balance$ for each operation in order to prove *Inv'*.

- $ethDeposit(sender, t)$:

  First, assume $t.pk = k'.pk$. This immediately leads to a contradiction: The postconditions imply $duringExtension(k'.pk)$, which together with Invariant 2 means that $k'.pk \notin pks_{o'}$. This contradicts the fact that $k' \in knots_{o'}$.

  Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

- $ethPokeLatestBalance(sender, t)$:

First, assume $t.pk = k'.pk$. Then, with the help of Invariant 1, we can show that $t \notin pushedIn_{o'}$. This in turn implies $highestPushed'(k'.pk) = highestPushed(k'.pk)$, which together $k'.balance = knots_o[k'.pk].balance$ and *Inv* implies *Inv'*.

Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

- $ethPush(sender,\ t)$:

  First, assume $t.pk = k'.pk$. This immediately leads to a contradiction: The postconditions imply $unwound(k'.pk)$, which together with Invariant 2 means that $k'.pk \notin knotIDs_{o'}$. This contradicts the fact that $k' \in knots_{o'}$.

  Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

- $optDeposit(sender,\ t)$:

  First, assume $t.pk = k'.pk$. This immediately leads to a contradiction: The postconditions imply $k \notin knots_{o'}$. This contradicts the fact that $k' \in knots_{o'}$.

  Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

- $optBalanceIncrease(sender,\ t)$:

  First, assume $t.pk = k'.pk$. We make the following case distinction:

  - Case $t.amount \leq knots_o[t.pk].balance$:
    - Then the postconditions imply $k'.balance = knots_o[t.pk].balance$
    - *Inv* gives us $highestPushed(t.pk) = knots_o[t.pk].balance$. This together with the assumption of this case implies $t.amount \leq highestPushed(t.pk)$. Since $t.amount$ is not larger than the previous highest pushed amount, this implies $highestPushed'(t.pk) = highestPushed(t.pk)$
    - The previous two points imply that neither $k'.balance$ nor $highestPushed'(t.pk)$ changes compared to the pre-state. Thus, *Inv'* follows directly from *Inv*.

  - Case $t.amount > knots_o[t.pk].balance$
    - Then the postconditions imply $k'.balance = t.amount$
    - *Inv* gives us $highestPushed(t.pk) = knots_o[t.pk].balance$. This together with the assumption of this case implies $t.amount > highestPushed(t.pk)$.

With Assumption 2, this implies that $t.txnIdx$ must be larger than or equal to the transaction index of the currently highest pushed transaction. All of this taken together implies $highestPushed'(t.pk) = t.amount$

- The previous two points imply that both $k'.balance$ and $highestPushed'(t.pk)$ are equal to $t.amount$, which shows that *Inv'* holds

Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

- $optPush(sender, t)$:

First, assume $t.pk = k'.pk$. Then the postconditions imply $k'.balance = t.amount$. Further, together with Invariant 5, they imply $highestPushed'(t.pk) = t.amount$. The previous two points imply that both $k'.balance$ and $highestPushed'(t.pk)$ are equal to $t.amount$, which shows that *Inv'* holds.

Now assume $t.pk \neq k'.pk$. Then, $highestPushed'(k'.pk) = highestPushed(k'.pk)$ and $k'.balance = knots_o[k'.pk].balance$. Thus, *Inv'* follows from *Inv*.

Invariant 5

The following invariant states that if there is an unpushed extension transaction for a KNOT, then that transaction can only be followed by balance update transactions, and none of them has been pushed to Optimism yet.

Invariant 5: If $t \in txnsOut_e \setminus pushedIn_o$ and $isExtension(t)$, then for all $\hat{t} \in txnsOut_e$ with $\hat{t}.txnIdx > t.txnIdx$ and $\hat{t}.pk = t.pk$ we get $\hat{t} \notin pushedIn_o$ and $isUpdate(\hat{t})$

We show that Invariant 5 holds for each operation. We use *Inv* to refer to Invariant 5 in the pre-state, and *Inv'* to refer to it in the post-state.

In the following, let $t' \in txnsOut_{e'} \setminus pushedIn_{o'}$ denote an arbitrary unpushed transaction in the post-state such that $isExtension(t')$. We assume there exists a $\hat{t} \in txnsOut_{e'}$ such that $\hat{t}.txnIdx > t'.txnIdx$ and $\hat{t}.pk = t'.pk$. In order to show *Inv'*, we need to prove $\hat{t} \notin pushedIn_{o'}$ and $isUpdate(\hat{t})$.

- $ethDeposit(sender, t)$:

First, we consider the case $t' = t$. This leads to a contradiction, because $t.txnIdx$ is the largest transaction index and hence $\hat{t}.txnIdx > t'.txnIdx$ cannot hold (follows from $t.txnIdx = |txns_e|$ and Invariant 5.1).

Next, we consider the case $t' \neq t$. This leads to a contradiction as well: Because of $unwound(t.pk)$, Invariant 2 implies that there does not exist any unpushed extension transaction for $t.pk$ in the pre-state. Hence, in the post-state, there exists exactly one such transaction, namely t. This means $t' = t$, which contradicts the assumption for this case.

- $ethPokeLatestBalance(sender, t)$:

  Because $t \notin pushedIn_{o'}$ (can be shown with the help of Invariant 1) and $isUpdate(t)$, it is clear that t cannot violate the invariant. Thus, *Inv'* follows from *Inv*.

- $ethPush(sender, t)$: Nothing relevant changes.
- $optDeposit(sender, t)$: Nothing relevant changes.
- $optBalanceIncrease(sender, t)$: Invariant 2 applied to the post-state implies that t' cannot exist as defined. Thus, we are done.
- $optPush(sender, t)$: Invariant 2 applied to the post-state implies that t' cannot exist as defined. Thus, we are done.

Invariant 5.1: If $t \in txnsOut_e$ then $t.txnIdx < |txns_e|$

We show that Invariant 5.1 holds for each operation. We use *Inv* to refer to Invariant 5.1 in the pre-state, and *Inv'* to refer to it in the post-state.

- $ethDeposit(sender, t)$:

  Let $t' \in txnsOut_{e'}$. We need to show $t'.txnIdx < |txns_{e'}|$. If $t' = t$, then this follows from $t.txnIdx = |txns_e|$ and $|txns_{e'}| = |txns_e| + 1$. If $t' \neq t$, then this follows from $|txns_{e'}| = |txns_e| + 1$ and *Inv*.

- $ethPokeLatestBalance(sender, t)$:

  Let $t' \in txnsOut_{e'}$. We need to show $t'.txnIdx < |txns_{e'}|$. If $t' = t$, then this follows from $t.txnIdx = |txns_e|$ and $|txns_{e'}| = |txns_e| + 1$. If $t' \neq t$, then this follows from $|txns_{e'}| = |txns_e| + 1$ and *Inv*.

- $ethPush(sender, t)$: Since $txnOut_{e'} = txnOut_e$ and $|txns_{e'}| = |txns_e| + 1$, *Inv'* follows directly from *Inv*.
- $optDeposit(sender, t)$: Since $txnOut_{e'} = txnOut_e$ and $|txns_{e'}| = |txns_e|$, *Inv'* follows directly from *Inv*.
- $optBalanceIncrease(sender, t)$: Since $txnOut_{e'} = txnOut_e$ and $|txns_{e'}| = |txns_e|$, *Inv'* follows directly from *Inv*.
- $optPush(sender, t)$: Since $txnOut_{e'} = txnOut_e$ and $|txns_{e'}| = |txns_e|$, *Inv'* follows directly from *Inv*.

Invariant 5.2: If $t \in txnsOut_o$ then $t.txnIdx < |txns_o|$

The proof is analogous to the one for Invariant 5.1.

Invariant 6

Invariant 6: For all $k \in knots_o$ we have $k.index \neq gatewayIndex_o$

- $ethDeposit(sender, t)$: *Inv'* follows directly from *Inv*.
- $ethPokeLatestBalance(sender, t)$: *Inv'* follows directly from *Inv*.
- $ethPush(sender, t)$: *Inv'* follows directly from *Inv*.
- $optDeposit(sender, t)$: *Inv'* follows directly from *Inv*.
- $optBalanceIncrease(sender, t)$: *Inv'* follows directly from *Inv*.
- $optPush(sender, t)$:

  The postconditions imply $knots_{o'}[t.pk].index = t.toIndex$, with the requirement that $indexOwner_o(t.toIndex) = sender$. Since $indexOwner_o(gatewayIndex_o) = gatewayAddress_o$ and $gatewayAddress_o$ cannot be $sender$, this implies $knots_{o'}[t.pk].index \neq gatewayIndex_{o'}$

Main Invariant

Main Invariant: There exists a $C \in \mathbb{R}^+$ such that the following condition is an invariant:

$$spendableDETH_e + spendableDETH_o + unpushedDETH + totalGatewayRewards = C$$

We show that the above invariant holds for each operation. We use *Inv* to refer to the above invariant in the pre-state, and *Inv'* to refer to it in the post-state.

- $ethDeposit(sender,\ t)$:

  From the postcondition we get:
  - $knots_{e'}[t.pk].index = gatewayIndex_e$
  - $txns_{e'} = txns_e \cup \{t\}$

  We show the following facts:

  1. $spendableDETH_{e'} = spendableDETH_e - t.amount$

     Let $k = knots_e[t.pk]$ and $k' = knots_{e'}[t.pk]$.
     First, note that $k.index \neq gatewayIndex_e$ which follows from $indexOwner_e(k.index) = sender$ and the fact that $sender$ cannot be the owner of the gateway index. From $k.index \neq gatewayIndex_e$ and $k'.index = gatewayIndex_e$ (which follows from the postcondition) we can conclude that $k.balance$ is included in $spendableDETH_e$ but not in $spendableDETH_{e'}$, which concludes the proof.

  2. $spendableDETH_{o'} = spendableDETH_o$

     Follows from $knots_{o'} = knots_o$

  3. $unpushedDETH' = unpushedDETH + t.amount$

     The claim follows from the following two facts:

     a. $notPushedToOptimism' = notPushedToOptimism + t.amount$

        We rewrite the left-hand side until we get to the right-hand side:

        $notPushedToOptimism'$

        $= \sum_{pk} unpushedDETHFor'(pk)\ [pk \in pks_{e'}]$

$=^{(1)} \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \ \in \ pks_e \setminus \{t.pk\}] \right)$

$+ unpushedDETHFor'(t.pk)$

$=^{(2)} \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \ \in \ pks_e] \right)$

$+ unpushedDETHFor'(t.pk)$

$= notPushedToOptimism + unpushedDETHFor'(t.pk)$

$=^{(3)} notPushedToOptimism + t.amount$

Explanation:

(1) The value of $unpushedDETHFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
(2) This follows from $unpushedDETHFor(t.pk) = 0$. To see this, observe that the preconditions imply $knots_e[t.pk].index \neq gatewayIndex$. This in turn implies $currentTxnsFor(t.pk) = \emptyset$, which finally implies $unpushedDETHFor(t.pk) = 0$.
(3) The pre- and postconditions imply $currentTxnsFor(t.pk) = \{t\}$. This together with the fact that $t \notin pushedIn_{o'}$ (follows from Invariant 1 and $pushedIn_{o'} = pushedIn_o$) gives us $unpushedDETHFor'(t.pk) = t.amount$.

b. $notPushedToEthereum' = notPushedToEthereum$

Follows from $txnsOut_{o'} = txnsOut_o$ and $pushedIn_{e'} = pushedIn_e$

4. $totalGatewayRewards' = totalGatewayRewards$

Observe:

$totalGatewayRewards'$

$= \sum_{pk} gatewayRewardsFor'(pk)\ [pk \in \ BLSPubKey]$

$=^{(1)} \left(\sum_{pk} gatewayRewardsFor(pk)\ [pk \in \ BLSPubKey \setminus \{t.pk\}] \right)$

$+ gatewayRewardsFor'(t.pk)$

$= \left(\sum_{pk} gatewayRewardsFor(pk)\ [pk \in \ BLSPubKey] \right)$

$- gatewayRewardsFor(t.pk) + gatewayRewardsFor'(t.pk)$

$= totalGatewayRewards - gatewayRewardsFor(t.pk) + gatewayRewardsFor'(t.pk)$

$=^{(2)} totalGatewayRewards$

Explanation:

(1) The value of $gatewayRewardsFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

(2) First, note that $gatewayRewardsFor(t.pk) = 0$. This follows from $knots_e[t.pk].index \neq gatewayIndex_e$. Next, note that $gatewayRewardsFor'(t.pk) = 0$ as well. This follows from $knots_e[t.pk].balance = highestTxn(t.pk)$

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.

- $ethPokeLatestBalance(sender, t)$:

From the postcondition we get:

- $txns_{e'} = txns_e \cup \{t\}$
- $pushed_{e'} = pushed_e \cup \{t[txnIdx \leftarrow n] \mid n \in \mathbb{N}\}$

Let $\Delta = highestTxn'(t.pk) - highestTxn(t.pk)$. We show the following facts:

1. $spendableDETH_{e'} = spendableDETH_e$

   Follows from $knots_{e'} = knots_e$

2. $spendableDETH_{o'} = spendableDETH_o$

   Follows from $knots_{o'} = knots_o$

3. $unpushedDETH' = unpushedDETH + \Delta$

   The claim follows from the following two facts:

   a. $notPushedToOptimism' = notPushedToOptimism + \Delta$

      Observe:

      $notPushedToOptimism'$

      $= \sum_{pk} unpushedDETHFor'(pk)\ [pk \in pks_{e'}]$

$=^{(1)} (\sum_{pk} unpushedDETHFor(pk)\ [pk\ \in\ pks_e \setminus \{t.pk\}])$

$+ unpushedDETHFor'(t.pk)$

$= (\sum_{pk} unpushedDETHFor(pk)\ [pk\ \in\ pks_e])$

$-\ unpushedDETHFor(t.pk)\ +\ unpushedDETHFor'(t.pk)$

$=\ notPushedToOptimism\ +$
$unpushedDETHFor'(t.pk)\ -\ unpushedDETHFor(t.pk)$

$=^{(2)} notPushedToOptimism\ +\ highestTxn'(t.pk)\ -\ highestTxn(t.pk)$

$=\ notPushedToOptimism\ +\ \Delta$

Explanation:

(1) The value of $unpushedDETHFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
(2) Observe the following:
$unpushedDETHFor'(t.pk)\ -\ unpushedDETHFor(t.pk)$
$= highestTxn'(t.pk)\ -\ highestPushed'(t.pk)\ -$
$(highestTxn(t.pk)\ -\ highestPushed(t.pk)$
$=\ highestTxn'(t.pk)\ -\ highestTxn(t.pk)$

This follows from the fact that
$highestPushed'(t.pk)\ =\ highestPushed(t.pk)$.

b. $notPushedToEthereum'\ =\ notPushedToEthereum$

Follows from $txnsOut_{o'}\ =\ txnsOut_o$ and $pushedIn_{e'}\ =\ pushedIn_e$

4. $totalGatewayRewards'\ =\ totalGatewayRewards\ -\ \Delta$

Let $k\ =\ knots_e[t.pk]$ and $k'\ =\ knots_{e'}[t.pk]$. Note that $k'.balance\ =\ k.balance$. Observe:

$totalGatewayRewards'$

$= \sum_{pk} gatewayRewardsFor'(pk)\ [pk\ \in\ BLSPubKey]$

$=^{(1)} (\sum_{pk} gatewayRewardsFor(pk)\ [pk\ \in\ BLSPubKey \setminus \{t.pk\}])$

$+\ gatewayRewardsFor'(t.pk)$

$= (\sum_{pk} gatewayRewardsFor(pk) \, [pk \in BLSPubKey])$

$- gatewayRewardsFor(t.pk) + gatewayRewardsFor'(t.pk)$

$= totalGatewayRewards - (gatewayRewardsFor(t.pk) - gatewayRewardsFor'(t.pk))$

$= totalGatewayRewards -$
$(k.balance - highestTxn(t.pk) - (k'.balance - highestTxn'(t.pk))$

$= totalGatewayRewards - (highestTxn'(t.pk) - highestTxn(t.pk))$

$= totalGatewayRewards - \Delta$

Explanation:

(1) The value of $gatewayRewardsFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.

- $ethPush(sender, \, t)$:

  From the postcondition we get:
  - $txns_{e'} = txns_{e} \cup \{t\}$
  - $pushed_{e'} = pushed_{e} \cup \{t\}$
  - $knots_{e'}[t.pk].index = t.toIndex$

  Let $k = knots_{e}[t.pk]$. We show the following facts:

  1. $spendableDETH_{e'} = spendableDETH_{e} + k.balance$

     Observe:

     $spendableDETH_{e'}$

     $= \sum_{pk} spendableDETHFor_{e'}(pk) \, [pk \in BLSPubKey]$

     $=^{(1)} (\sum_{pk} spendableDETHFor_{e}(pk) \, [pk \in BLSPubKey \setminus \{t.pk\}])$
     $+ spendableDETHFor_{e'}(t.pk)$

     $= (\sum_{pk} spendableDETHFor_{e}(pk) \, [pk \in BLSPubKey])$
     $- spendableDETHFor_{e}(t.pk) + spendableDETHFor_{e'}(t.pk)$

$= spendableDETH_e - spendableDETHFor_e(t.pk) + spendableDETHFor_{e'}(t.pk)$

$=^{(2)} spendableDETH_e + knots_{e'}[t.pk].balance$

$=^{(3)} spendableDETH_e + k.balance$

Explanation:

(1) The value of $spendableDETHFor_e$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

(2) $spendableDETHFor_e(t.pk) = 0$ follows from $knots_e[t.pk].index = gatewayIndex_e$. Further, $spendableDETHFor_{e'}(t.pk) = knots_{e'}[t.pk].balance$ follows from $knots_{e'}[t.pk].index \neq gatewayIndex_{e'}$

(3) Follows from $knots_{e'}[t.pk].balance = k.balance$

2. $spendableDETH_{o'} = spendableDETH_o$

Follows from $knots_{o'} = knots_o$

3. $unpushedDETH' = unpushedDETH - highestTxn(t.pk)$

First, we show the following two facts:

a. $notPushedToOptimism' = notPushedToOptimism - unpushedDETHFor(t.pk)$

Observe:

$notPushedToOptimism'$

$= \sum_{pk} unpushedDETHFor'(pk)\ [pk \in pks_{e'}]$

$=^{(1)} \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_e \setminus \{t.pk\}] \right)$

$+ unpushedDETHFor'(t.pk)$

$= \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_e] \right)$

$- unpushedDETHFor(t.pk) + unpushedDETHFor'(t.pk)$

$= notPushedToOptimism + unpushedDETHFor'(t.pk) - unpushedDETHFor(t.pk)$

$=^{(2)} notPushedToOptimism - unpushedDETHFor(t.pk)$

Explanation:

(1) The value of $unpushedDETHFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

(2) $unpushedDETHFor'(t.pk) = 0$ follows from $knots_e[t.pk].index \neq gatewayIndex_e$

b. $notPushedToEthereum' = notPushedToEthereum - unwindingAmount(k.pk)$

Observe:

$notPushedToEthereum'$

$= \sum_{pk} unwindingAmount'(pk)\ [pk \in BLSPubKey]$

$=^{(1)} \left(\sum_{pk} unwindingAmount(pk)\ [pk \in BLSPubKey \setminus \{t.pk\}] \right)$

$+ unwindingAmount'(t.pk)$

$= \left(\sum_{pk} unwindingAmount(pk)\ [pk \in BLSPubKey] \right)$

$- unwindingAmount(t.pk) + unwindingAmount'(t.pk)$

$= notPushedToEthereum - unwindingAmount(t.pk) + unwindingAmount'(t.pk)$

$=^{(2)} notPushedToEthereum - unwindingAmount(t.pk)$

Explanation:

(1) The value of $unwindingAmount$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

(2) We need to show that $unwindingAmount'(t.pk) = 0$:

   (a) First, note that $t \in txnsOut_o \setminus pushedIn_e$

   (b) Then, Invariant 3 implies that there does not exist any other transaction $t' \in txnOut_o \setminus pushedIn_e$ with $t'.pk = t.pk$

   (c) Furthermore, note that $t \in pushedIn_{e'}$ implies $t \notin txnOut_{e'} \setminus pushedIn_{e'}$

   (d) Together, (b) and (c) imply $unwindingAmount'(t.pk) = 0$

Next, observe that

$unpushedDETHFor(k.pk) + unwindingAmount(k.pk)$
$= highestTxn(k.pk) - highestPushed(k.pk) + unwindingAmount(k.pk)$
$= highestTxn(k.pk) - unwindingAmount(k.pk$
$+ unwindingAmount(k.pk)$ (By Invariant 4)
$= highestTxn(k.pk)$

Remember that
$unpushedDETH' = notPushedToOptimism' + notPushedToEthereum'$.
This together with a., b. and
$unpushedDETHFor(k.pk) + unwindingAmount(k.pk) = highestTxn(k.pk)$
proves the claim.

4. $totalGatewayRewards' = totalGatewayRewards - (k.balance - highestTxn(k.pk))$

Observe:

$totalGatewayRewards'$

$= \sum_{pk} gatewayRewardsFor'(pk)\ [pk \in BLSPubKey]$

$=^{(1)} (\sum_{pk} gatewayRewardsFor(pk)\ [pk \in BLSPubKey \setminus \{t.pk\}])$

$+ gatewayRewardsFor'(t.pk)$

$= (\sum_{pk} gatewayRewardsFor(pk)\ [pk \in BLSPubKey])$

$- gatewayRewardsFor(t.pk) + gatewayRewardsFor'(t.pk)$

$= totalGatewayRewards - gatewayRewardsFor(t.pk) + gatewayRewardsFor'(t.pk)$

$=^{(2)} totalGatewayRewards - gatewayRewardsFor(t.pk)$

$= totalGatewayRewards - (k.balance - highestTxn(t.pk))$

Explanation:

(1) The value of $gatewayRewardsFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
(2) Note that $gatewayRewardsFor'(t.pk) = 0$ because $k.index \neq gatewayIndex_e$

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.

- $optDeposit(sender,\ t)$:

From the postcondition we get:

- $knots_{o'} = knots_{o} \setminus \{knots_{o}[t.pk]\}$
- $txns_{o'} = txns_{o} \cup \{t\}$

We show the following facts:

1. $spendableDETH_{e'} = spendableDETH_{e}$

   Follows from $knots_{e'} = knots_{e}$

2. $spendableDETH_{o'} = spendableDETH_{o} - t.amount$

   Observe:

   $spendableDETH_{o'}$

   $= \sum_{pk} spendableDETHFor_{o'}(pk)\ [pk \in BLSPubKey]$

   $=^{(1)} \left(\sum_{pk} spendableDETHFor_{o}(pk)\ [pk \in BLSPubKey \setminus \{t.pk\}] \right) + spendableDETHFor_{o'}(t.pk)$

   $= \left(\sum_{pk} spendableDETHFor_{o}(pk)\ [pk \in BLSPubKey] \right) - spendableDETHFor_{o}(t.pk) + spendableDETHFor_{o'}(t.pk)$

   $= spendableDETH_{o} - spendableDETHFor_{o}(t.pk) + spendableDETHFor_{o'}(t.pk)$

   $=^{(2)} spendableDETH_{o} - knots_{o}[t.pk].balance$

   $=^{(3)} spendableDETH_{o} - t.amount$

   Explanation:

   (1) The value of $spendableDETHFor_{o}$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
   (2) Follows from $spendableDETHFor_{o}(t.pk) = knots_{o}[t.pk].balance$ and $spendableDETHFor_{o'}(t.pk) = 0$
   (3) Follows from the precondition $t.amount = knots_{o}[t.pk].balance$

3. $unpushedDETH' = unpushedDETH + t.amount$

   The claim follows from the following two facts:

a. $notPushedToOptimism' = notPushedToOptimism$

This follows from the fact that nothing relevant changes.

b. $notPushedToEthereum' = notPushedToEthereum + t.amount$

Observe:

$notPushedToEthereum'$

$= \sum_{pk} unwindingAmount'(pk)\ [pk \in BLSPubKey]$

$=^{(1)} (\sum_{pk} unwindingAmount(pk)\ [pk \in BLSPubKey \setminus \{t.pk\}])$

$+ unwindingAmount'(t.pk)$

$= (\sum_{pk} unwindingAmount(pk)\ [pk \in BLSPubKey])$

$- unwindingAmount(t.pk) + unwindingAmount'(t.pk)$

$= notPushedToEthereum - unwindingAmount(t.pk) + unwindingAmount'(t.pk)$

$=^{(2)} notPushedToEthereum + unwindingAmount'(t.pk)$

$=^{(3)} notPushedToEthereum + t.amount$

Explanation:

(1) The value of $unwindingAmount$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
(2) The preconditions together with Invariant 2 imply that there does not exist any $t' \in txnsOut_o \setminus pushedIn_e$ with $t'.pk = t.pk$. This in turn implies $unwindingAmount(t.pk) = 0$
(3) The postconditions and Invariant 1 imply $t \in txnsOut_{o'} \setminus pushedIn_{e'}$. This in turn implies $unwindingAmount'(t.pk) = t.amount$

4. $totalGatewayRewards' = totalGatewayRewards$

Follows from the fact that no relevant state changes ($knots_{e'} = knots_e$ and $txns_{e'} = txns_e$).

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.

- $optPush(sender, t)$:

From the postcondition we get:

- $txns_{o'} = txns_o \cup \{t\}$
- $pushed_{o'} = pushed_o \cup \{t\}$
- $knots_{o'}[t.pk] = k$

We show the following facts:

1. $spendableDETH_{e'} = spendableDETH_e$

   Follows from $knots_{e'} = knots_e$.

2. $spendableDETH_{o'} = spendableDETH_o + t.amount$

   Follows from KNOT k being added to $knots_{o'}$ and $k.index \neq gatewayIndex_o$. The latter follows from $k.index = t.toIndex$ and $indexOwner_o(t.toIndex) = sender$. Since the owner of $gatewayIndex_o$ is $gatewayAddress_o$ which must be different from $sender$, this implies $k.index \neq gatewayIndex_o$

3. $unpushedDETH' = unpushedDETH - t.amount$

   The claim follows from the following two facts:

   a. $notPushedToOptimism' = notPushedToOptimism - t.amount$

      Observe:

      $notPushedToOptimism'$

      $= \sum_{pk} unpushedDETHFor'(pk)\ [pk \in pks_{e'}]$

      $=^{(1)} \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_e \setminus \{t.pk\}] \right)$

      $+ unpushedDETHFor'(t.pk)$

      $= \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_e] \right)$

      $- unpushedDETHFor(t.pk) + unpushedDETHFor'(t.pk)$

      $= notPushedToOptimism -$
      $(unpushedDETHFor(t.pk) - unpushedDETHFor'(t.pk))$

      $=^{(2)} notPushedToOptimism - t.amount$

Exlanation:

(1) The value of $unpushedDETHFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state

(2) This follows from:

$$unpushedDETHFor(t.pk) - unpushedDETHFor'(t.pk)$$
$$= highestTxn(t.pk) - highestPushed(t.pk) - (highestTxn'(t.pk) - highestPushed'(t.pk))$$

$$=^{(a)} highestPushed'(t.pk) - highestPushed(t.pk)$$

$$=^{(b)} highestPushed'(t.pk)$$

$$=^{(c)} t.amount$$

Regarding (a): This step follows from $highestTxn'(t.pk) = highestTxn(t.pk)$.

Regarding (b): Because of $t \in txnOut_e \setminus pushedIn_o$ and Invariant 5 we know that there does not exist a transaction $t' \in txnsOut_e$ with $t'.pk = t.pk$ and $t.txnIdx < t'.txnIdx$. This in turn implies $highestPushed(t.pk) = 0$.

Regarding (c): Invariant 5 implies that t is the latest extension transaction and that it is not follows by any pushed transactions. This implies $currentTxnsFor(t.pk) = \{t\}$, which in turn implies $highestPushed'(t.pk) = t.amount$.

b. $notPushedToEthereum' = notPushedToEthereum$

Follows from $txnsOut_{o'} = txnsOut_o$ and $pushedIn_{e'} = pushedIn_e$

4. $totalGatewayRewards' = totalGatewayRewards$

Follows from the fact that no relevant state changes ($knots_{e'} = knots_e$ and $txns_{e'} = txns_e$).

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.

- $optBalanceIncrease(sender,\ t)$:

From the postconditions we get:

- $txns_{o'} = txns_o \cup \{t\}$
- $pushed_{o'} = pushed_o \cup \{t\}$

- $knots_{o'}[t.pk].balance = max(t.amount,\ knots_{o}[t.pk].balance)$

Let $\Delta = highestPushed'(t.pk) - highestPushed(t.pk)$. We show the following facts:

1. $spendableDETH_{e'} = spendableDETH_{e}$

   Follows from $knots_{e'} = knots_{e}$.

2. $spendableDETH_{o'} = spendableDETH_{o} + \Delta$

   Observe:

   $spendableDETH_{o'}$

   $= \sum_{pk} spendableDETHFor_{o'}(pk)\ [pk \in BLSPubKey]$

   $=^{(1)} (\sum_{pk} spendableDETHFor_{o}(pk)\ [pk \in BLSPubKey \setminus \{t.pk\}])$
   $\quad + spendableDETHFor_{o'}(t.pk)$

   $= (\sum_{pk} spendableDETHFor_{o}(pk)\ [pk \in BLSPubKey])$
   $\quad - spendableDETHFor_{o}(t.pk) + spendableDETHFor_{o'}(t.pk)$

   $= spendableDETH_{o} + (spendableDETHFor_{o'}(t.pk) - spendableDETHFor_{o}(t.pk))$

   $=^{(2)} spendableDETH_{o} + (knots_{o'}[t.pk].balance - knots_{o}[t.pk].balance)$

   $=^{(3)} spendableDETH_{o} + (highestPushed'(t.pk) - highestPushed(t.pk))$

   $= spendableDETH_{o} + \Delta$

   Explanation:

   (1) The value of $spendableDETHFor_{o}$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
   (2) Invariant 6 implies $knots_{o}[t.pk] \neq gatewayIndex_{o}$ and $knots_{o'}[t.pk] \neq gatewayIndex_{o}$. This in turn implies $spendableDETHFor_{o}(t.pk) = knots_{o}[t.pk].balance$ and $spendableDETHFor_{o'}(t.pk) = knots_{o'}[t.pk].balance$
   (3) Follows from Invariant 4.1

3. $unpushedDETH' = unpushedDETH - \Delta$

The claim follows from the following two facts:

a. $notPushedToOptimism' = notPushedToOptimism - \Delta$

Observe:

$notPushedToOptimism'$

$= \sum_{pk} unpushedDETHFor'(pk)\ [pk \in pks_{e'}]$

$=^{(1)} \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_{e} \setminus \{t.pk\}] \right)$

$+ unpushedDETHFor'(t.pk)$

$= \left(\sum_{pk} unpushedDETHFor(pk)\ [pk \in pks_{e}] \right)$

$- unpushedDETHFor(t.pk) + unpushedDETHFor'(t.pk)$

$= notPushedToOptimism -$
$(unpushedDETHFor(t.pk) - unpushedDETHFor'(t.pk))$

$=^{(2)} notPushedToOptimism - (highestPushed'(t.pk) - highestPushed(t.pk))$

$= notPushedToOptimism - \Delta$

Explanation:

(1) The value of $unpushedDETHFor$ for all KNOTs other than $t.pk$ does not change between the pre- and post-state
(2) To see this, observe the following:

$unpushedDETHFor(t.pk) - unpushedDETHFor'(t.pk)$

$= highestTxn(t.pk) - highestPushed(t.pk) -$
$(highestTxn'(t.pk) - highestPushed'(t.pk))$

$= highestPushed'(t.pk) - highestPushed(t.pk)$

The last step follows from $highestTxn'(t.pk) = highestTxn(t.pk)$

b. $notPushedToEthereum' = notPushedToEthereum$

Follows from $txnsOut_{o'} = txnsOut_{o}$ and $pushedIn_{e'} = pushedIn_{e}$

4. $totalGatewayRewards' = totalGatewayRewards$

Follows from the fact that no relevant state changes ($knots_{e'} = knots_{e}$ and $txns_{e'} = txns_{e}$).

Facts 1 to 4 show that the total amount of dETH remains the same. This together with *Inv* implies *Inv'*.