LSD NETWORK
Formal Properties

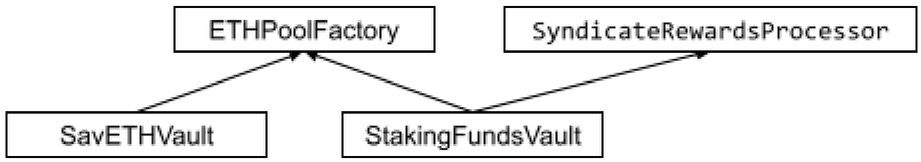
Blockswap network | Protocol: LSD Network | 2023 - Jul

Provided by Blockswap LABS

Prepared as joint work with Runtime Verification

LSD Network Contract Properties

ETH Vaults



`ETHPoolFactory` is an abstract base contract that holds common state and functionality. The two concrete implementations are `SavETHVault` and `StakingFundsVault`.

Recall that when staking with the LSD, the 32 ETH needed to activate a validator are divided as follows:

- 4 ETH are provided by the node runner. In the Stakehouse, these are represented as 4 collateralized SLOT tokens.
- 4 ETH are provided by LPs. In the Stakehouse, these are represented as 4 free-floating SLOT tokens.
- 24 ETH are provided by LPs. In the Stakehouse, these are represented as 24 dETH

Here, we only consider the 4 + 24 ETH that are provided by LPs. Collecting these funds is the responsibility of `SavETHVault` and `StakingFundsVault`, respectively. Whenever a user deposits ETH into one of these contracts, they receive an equivalent amount of LP tokens (`LPToken`) in return, where each validator has its own `LPToken`. Once the corresponding validator has been activated, the LP tokens will appreciate in value as the validator earns rewards (in the case of `SavETHVault`) or collects fees and MEV (in the case of `StakingFundsVault`).

ETHPoolFactory

Contract state:

- `numberOfLPTokensIssued : uint256`
- `maxStakingAmountPerValidator : uint256`
- `lpTokenForKnot : mapping(bytes => LPToken)`
- `knotAssociatedWithLPToken : mapping(LPToken => bytes)`

Note that `maxStakingAmountPerValidator` is set during initialization and not expected to change afterwards.

Constants:

- `MIN_STAKING_AMOUNT : uint256`

Invariants

<table>
<tr><td>Invariant</td><td>Inv-EPF-1</td></tr>
<tr><td colspan="2"><code>∀b:bytes, a:address.
    lpTokenForKnot[b] == a <==> knotAssociatedWithLPToken[a] == b</code></td></tr>
<tr><td>Description</td><td>The mappings <code>lpTokenForKnot</code> and <code>knotAssociatedWithLPToken</code> are inverses of each other.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-EPF-2</td></tr>
<tr><td colspan="2"><code>numberOfLPTokensIssued == |registeredTokens|

where registeredTokens =
    {b:bytes. lpTokenForKnot[b] | lpTokenForKnot[b] != address(0)}</code></td></tr>
<tr><td>Description</td><td><code>numberOfLPTokensIssued</code> corresponds to the number of registered <code>LPTokens</code> in <code>lpTokenForKnot</code>.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-EPF-3</td></tr>
<tr><td colspan="2"><code>∀b1:bytes, b2:bytes.
    b1 != b2 && lpTokenForKnot[b1] != ZERO && lpTokenForKnot[b2] != ZERO
    ==> lpTokenForKnot[b1] != lpTokenForKnot[b2]</code></td></tr>
<tr><td>Description</td><td>Each validator is assigned a unique LP token contract.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-EPF-4</td></tr>
<tr><td colspan="2"><code>∀a:address.
    knotAssociatedWithLPToken[a].length == 0 ||
    knotAssociatedWithLPToken[a].length == 48</code></td></tr>
<tr><td>Description</td><td>BLS public keys must be 48 bytes long.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-EPF-5</td></tr>
<tr><td colspan="2"><code>∀b:bytes.
    lpTokenForKnot[b].totalSupply() <= maxStakingAmountPerValidator</code></td></tr>
</table>

Description	The total amount of LP tokens associated with a KNOT must not exceed the maximum staking amount.

Invariant	**Inv-EPF-6**
∀ *b*:bytes. `lpTokenForKnot[`*b*`] != ZERO ==>` `liquidStakingManager.isBLSPublicKeyPartOfLSDNetwork(`*b*`)`	
Description	If a KNOT *b* is registered in the vault (i.e., `lpTokenForKnot[`*b*`]` is non-zero), then it must also be part of the LSD network. In other words, staking is only possible to KNOTs created by `LiquidStakingManager`.
Notes	This invariant requires cooperation from any deriving contract, which must reject any staking attempt to a KNOT that is unknown to the LSD.

SavETHVaultContract state:

- `indexOwnedByTheVault : uint256`
- `dETHForKnot : mapping(bytes => KnotDETHDetails)`

Note that `indexOwnedByTheVault` is set during initialization and not expected to change afterwards.

Constants:

- `KNOT_BATCH_AMOUNT : uint256 = 24 ether`

In the following invariant and function specifications, we use `this` to refer to a particular `SavETHVault` instance and `this.balance` to its ETH balance.

Reward Distribution

A fully staked validator in the `SavETHVault` is associated with 24 LP tokens. If a user has staked s LP tokens, they are entitled to $\frac{s}{24}$ of the validator's protocol rewards. To actually claim their rewards, users need to burn their LP tokens using `SavETHVault.burnLPToken()` (or one of the related functions), which also moves the validator into the open index, if this was not done already. This pays out $\frac{s}{24}$ of the rewards that the validator has earned until the point it was moved to the open index, plus the user's initial stake s. The rewards and the initial stake are paid out in either ETH or dETH, depending on whether the validator has already been activated or not. (If the validator has not been activated yet, then it has not earned any rewards, and the user simply gets back the ETH that they have staked.)

Invariants

<table>
<tr><td>Invariant</td><td>Inv-SEV-1 [may be violated by external actions]</td></tr>
<tr><td colspan="2"><code>this.balance == Σ t∈tokens. t.totalSupply()</code>

<code>where tokens = {b:bytes. lpTokenForKnot[b] |</code>
<code>lpTokenForKnot[b] != address(0) &&</code>
<code>lifecycleStatus(b) == INITIALS_REGISTERED}</code></td></tr>
<tr><td>Description</td><td>The total amount of ETH held by the <code>SavETHVault</code> contract is equal to the total LP token supply across all KNOTs that are in state <code>INITIALS_REGISTERED</code>.</td></tr>
<tr><td>Notes</td><td>Ensures that when a user burns their LP tokens using <code>SavETHVault.burnLPToken()</code>, the contract always holds enough ETH that can be transferred to the user.

Invariant may be violated by an external action: Namely, when someone transfers ETH to the contract directly. However, this does not break any functionality because it only ever increases <code>this.balance</code>, which is harmless.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-SEV-2 [may be violated by external actions]</td></tr>
<tr><td colspan="2"><code>savETHToken.balanceOf(this) == Σ b∈blsPKs. savETHFromKnot(b)</code>

<code>where blsPKs = {b:bytes. b | lpTokenForKnot[b] != address(0)}</code>

<code>and savETHFromKnot(b:bytes) = dETHForKnot[b].savETHBalance *</code>
<code>(lpTokenForKnot[b].totalSupply() / 24 ether)</code></td></tr>
<tr><td>Description</td><td>The total amount of savETH held by the <code>SavETHVault</code> contract equals the amount of savETH that the contract received by moving KNOTs to the open index.

Note that <code>savETHFromKnot(b)</code> denotes the amount of savETH that the contract received when moving b to the open index, minus any savETH that was converted to dETH via <code>SavETHVault.burnLPToken()</code>.</td></tr>
<tr><td>Notes</td><td>Ensures that when a user burns their LP tokens using <code>SavETHVault.burnLPToken()</code>, the contract holds enough savETH that can be transferred to the user.

Regarding <code>savETHFromKnot(b)</code>: <code>dETHForKnot[b].savETHBalance</code> refers to the amount of savETH that the <code>SavETHVault</code> contract received when moving</td></tr>
</table>

	KNOT *b* to the open index. (If *b* is not in the open index, then `dETHForKnot[b].savETHBalance` is zero.) Further, for a fully staked KNOT, we have `lpTokenForKnot[b].totalSupply() == 24 ether`. Whenever some amount of LP tokens is burned, then a certain amount of savETH is transferred[1] from the contract to the user such that the invariant is preserved. *Invariant may be violated by external actions:* Someone can directly transfer savETH to the `SavETHVault` contract. In this case, the contract holds savETH that is not associated with any of its KNOTs. However, this is harmless. [1] This is only true for KNOTs in state `TOKENS_MINTED and EXITED`. However, for KNOTs in different states, the contract doesn't hold any savETH, and correspondingly, `dETHForKnot[b].savETHBalance` should be zero.

Invariant	**Inv-SEV-3** *[may be violated by external actions]*

```
∀b:bytes. lpTokenForKnot[b] != ZERO ==>
   dETHForKnot[b].withdrawn <==> savETHManager.isKnotPartOfOpenIndex(b)
```

Description	If a KNOT *b* is registered (i.e., `lpTokenForKnot[b]` is non-zero), then `dETHForKnot[b].withdrawn` is true if and only if *b* is in the open index.
Notes	*Invariant may be violated by external actions:* KNOTs in the open index may be moved into private indices by anyone who provides the required savETH. If someone other than `SavETHVault` moves a KNOT registered in the LSD into an index, then the above invariant is broken (see Issue#9). See Inv-SEV-4 for a weaker version of this invariant that holds unconditionally.

Invariant	**Inv-SEV-4**

```
∀b:bytes. lpTokenForKnot[b] != ZERO ==>
   !dETHForKnot[b].withdrawn ==> savETHMan.associatedIndexIdForKnot(b) ==
                                  indexOwnedByTheVault
```

Description	If a KNOT *b* is registered (i.e., `lpTokenForKnot[b]` is non-zero) and not withdrawn then it is in the `SavETHVault` index.

Invariant	**Inv-SEV-5**

```
∀b:bytes.
```

`dETHForKnot[b].withdrawn <==> dETHForKnot[b].savETHBalance != 0`	
Description	If a KNOT is withdrawn (i.e., it has been moved to the open index), then the `SavETHVault` contract must have received a non-zero amount of savETH.
Note	This invariant implies that the field `KnotDETHDetails.withdrawn` is redundant and can be removed. See Issue#8.

Invariant	**Inv-SEV-6** *[may be violated by external actions]*

`∀b:bytes. lpTokenForKnot[b] != ZERO ==>`
`!savETHMan.isKnotPartOfOpenIndex(b) ==>`
`savETHMan.associatedIndexIdForKnot(b) == indexOwnedByTheVault`	

Description	If a KNOT *b* is registered (i.e., `lpTokenForKnot[b]` refers to an instance of `LPToken`) and *b* is not part of the open index, then it must belong to the index owned by the `SavETHVault` contract.
Notes	*May be violated by external actions:* Anyone can grab KNOTs in the open index and move them to their own index, which would violate this invariant. So while `SavETHVault` does uphold this invariant, it cannot be relied on.

SyndicateRewardsProcessor

Contract state:

- `accumulatedETHPerLPShare : uint256`
- `totalClaimed : uint256`
- `totalETHSeen : uint256`
- `accumulatedETHPerLPAtTimeOfMintingDerivatives : mapping(bytes => uint256)`
- `claimed : mapping(address => mapping(address => uint256))`

Invariants

See the invariants in StakingFundsVault.

StakingFundsVault

Receives 50% of fees & MEV rewards from the `Syndicate` contract (the other 50% go to the node runners, but this is not handled here). These rewards are distributed among all LP token holders.

The ETH held by the `StakingFundsVault` contract comes from two sources:

1. Liquidity providers
2. Transaction fees & MEV rewards

The total amount of ETH coming from liquidity providers is tracked via the variable `totalETHFromLPs`. All remaining ETH held by the contract is assumed to come from transaction fees and MEV rewards. Though anyone may deposit ETH into the StakingFundsVault.

Contract state:

- `totalShares : uint256`. Total amount of LP tokens in circulation across all registered KNOTs who are in state `TOKENS_MINTED`.
- `totalETHFromLPs : uint256`. Total amount of ETH from LPs that has not been staked in the Ethereum Deposit Contract and is held by `StakingFundsVault`.

Important operations:

- depositETHForStaking
- burnLPForETH
- claimRewards
- unstakeSyndicateSETHByBurningLP
- claimFundsFromSyndicateForDistribution

Reward Distribution

If a user stakes $s \leq 4$ ETH, they are entitled to $\frac{s}{4n}$ of the total amount of transaction fees and MEV rewards earned by all the n validators in the `StakingFundsVault`. Note that a stake of s ETH results in s LP tokens, and each active validator is associated with exactly 4 LP tokens. In other words, if $S = 4n$ denotes the total amount of LP tokens across all validators, then the user earns $\frac{s}{S}$ of the rewards.

Thus, it does not matter which validator a user stakes to, since the rewards are distributed among all the stakers.

The amount of rewards that are currently available for user u`:address` and validator v`:bytes` is computed as follows:

$$availableRewards(u, v) = unprocessedRewards(u, v) - processedRewards(u, v)$$

with the following definitions:

- $unprocessedRewards(u, v) = totalRewardsPerShare \cdot lpTokenBalance(u, v)$
  with $totalRewardsPerShare =$ `accumulatedETHPerLPShare / 1e24`

- If `claimed[u][lpTokenForKnot[v]] > 0`:
  $processedRewards(u, v) =$ `claimed[u][lpTokenForKnot[v]]`

  Otherwise:
  $processedRewards(u, v) = totalRewardsPerShareOnActivation(v) \cdot lpTokenBalance(u, v)$

with $totalRewardsPerShareOnActivation(v)$ =
`accumulatedETHPerLPAtTimeOfMintingDerivatives[v] / 1e24`

The rewards given by $availableRewards(u, v)$ can be retrieved at any time by calling `StakingFundsVault.claimRewards()`. (This is different from the `SavETHVault`, which only allows users to retrieve their rewards when they burn their LP tokens. See SavETHVault Reward Distribution.)

Invariants

<table>
<tr><td>Invariant</td><td>Inv-SFV-1</td></tr>
<tr><td colspan="2"><code>totalShares == Σ t∈tokens. t.totalSupply()</code>

<code>where tokens = {b:bytes. lpTokenForKnot[b] |</code>
<code>lpTokenForKnot[b] != address(0) &&</code>
<code>lifecycleStatus(b) == TOKENS_MINTED}</code></td></tr>
<tr><td>Description</td><td><code>totalShares</code> is equal to the total amount of LP tokens from active validators that are in circulations.</td></tr>
<tr><td>Notes</td><td>This invariant requires cooperation from <code>LiquidStakingManager</code>, which must call <code>StakingFundsVault.updateDerivativesMinted()</code> once a KNOT transitions to <code>TOKENS_MINTED</code>.</td></tr>
</table>

<table>
<tr><td>Invariant</td><td>Inv-SFV-2</td></tr>
<tr><td colspan="2"><code>totalETHFromLPs == Σ t∈tokens. t.totalSupply()</code>

<code>where tokens = {b:bytes. lpTokenForKnot[b] |</code>
<code>lpTokenForKnot[b] != address(0) &&</code>
<code>lifecycleStatus(b) == INITIALS_REGISTERED}</code></td></tr>
<tr><td>Description</td><td><code>totalETHFromLPs</code> keeps track of the amount of ETH that has been deposited by liquidity providers and that has not been sent to the Deposit Contract for staking. Note that for each deposited ETH, LPs receive 1 LP token in return.</td></tr>
<tr><td>Notes</td><td>This invariant requires cooperation from <code>LiquidStakingManager</code>, which must call <code>StakingFundsVault.withdrawETH()</code> to withdraw exactly 4 ETH once a KNOT transitions to <code>TOKENS_MINTED</code>. Also note that <code>withdrawETH()</code> must only be called for KNOTs with 4 LP tokens staked.</td></tr>
</table>

Invariant	**Inv-SFV-3**

`this.balance >= totalETHFromLPs + totalAvailableRewards`
 	
`where totalAvailableRewards = Σ u:address, b∈blsPKs.` *availableRewards*(*u*, *b*)
 	
`and blsPKs = {b:bytes. b │ lpTokenForKnot[b] != address(0)}`	

Description	The ETH held by the `StakingFundsVault` contract consists of the funds provided by liquidity providers (see Inv-SFV-2) and the total amount of unclaimed (i.e., available) rewards. The contract balance may be larger because the rewards received are only accounted for after `StakingFundsVault.updateAccumulatedETHPerLP()` has been called. On the flip side, this invariant *does* hold with strict equality immediately after this function is executed.

Invariant	**Inv-SFV-4**

`∀b:bytes. lpTokenForKnot[b] != ZERO ==>`
`lifecycleStatus(b) >= TOKENS_MINTED ==>`
`accumulatedETHPerLPAtTimeOfMintingDerivatives[b] > 0`	

Description	If a KNOT *b* is registered in `StakingFundsVault` (i.e., `lpTokenForKnot[b]` is non-zero) and its tokens have been minted, then the amount of rewards at the time when the tokens were minted must have been recorded (which must be larger than zero). This ensures that anyone who has staked to that KNOT does not earn rewards from before the KNOT was activated.
Notes	This invariant requires cooperation from `LiquidStakingManager`, which must call `StakingFundsVault.updateDerivativesMinted()` whenever a KNOT transitions to `TOKENS_MINTED`.

Invariant	**Inv-SFV-5**

`∀b:bytes.`
`accumulatedETHPerLPAtTimeOfMintingDerivatives[b] <=`
`accumulatedETHPerLPShare`	

Description	Historical snapshots of accumulated ETH at the time of minting derivatives for a specific BLS public key cannot be more than the current live accumulated ETH.

Invariant	**Inv-SFV-6**

∀ *u*:address, *b*:bytes.
  claimed[*u*][lpTokenForKnot[*b*]] <=
  accumulatedETHPerLPShare * lpTokenForKnot[*b*].balanceOf(u) / 1e24	

Description	Users cannot have claimed more rewards than are available.

Syndicate

Model

Notation:

- $S^{\times}$: Given a set $S \subseteq \mathbb{N}$, we use $S^{\times} \subseteq S \times S$ to denote the set of consecutive pairs of elements of S:
  $S^{\times} = \{(s_1, s_2) \in S \times S \mid s_1 \neq s_2 \wedge \neg(\exists s \in S.\ s_1 < s < s_2)\}$
- Let $a_0, a_1, ..., a_n \in \mathbb{R}$ be a finite sequence and $i, i' \in \mathbb{N}$. We define $a_{[i,i']} = \Sigma_{i<k\leq i'} a_k$
- We use $\mathbf{0}$ to denote the singleton set $\{0\}$

State:

- t: Counter that keeps track of the number of (public and external) function calls that have been made to the `Syndicate` contract. Whenever a function has finished execution, the counter increases (but not for nested calls). We assume deployment happens $t = 0$.
- ΔR_i: Amount of ETH received by the `Syndicate` contract in call i, where $0 \leq i \leq t$
- n_i: Number of active KNOTs at the end of call i. Corresponds to `numberOfActiveKnots`.
- $U \subseteq \{1, ..., t\}$: Set of calls to `Syndicate.updateAccruedETHPerShares()`. Initially, we assume $U = \emptyset$.
- $\alpha(k)$: If KNOT k has been activated via `Syndicate.activateProposers()` in call number c, then $\alpha(k) = \{c\}$. Otherwise, if k has not been activated yet, then $\alpha(k) = \emptyset$.
- $\delta(k)$: If KNOT k has been deactivated via `Syndicate._deRegisterKnot()` in call number c, then $\delta(k) = \{c\}$. Otherwise, if k has not been activated yet, then $\delta(k) = \emptyset$.

- **Specific to rewards for collateralized SLOT holders:**
  - ΔC_i: Amount of ETH received in call i that goes to collateralized SLOT owners. It is defined by $\Delta C_i = \Sigma_k \frac{\Delta R_k}{2}\ [n_i \neq 0 \wedge k \leq i \wedge (\forall l \in \{k, ..., i-1\}.\ n_l = 0)]$.

Most of the time, ΔC_i is just $\frac{\Delta R_i}{2}$ (i.e., half of the rewards go to collateralized SLOT owners). However, if the number of active KNOTs has been zero, then ΔC_i also includes the accumulated rewards received during that time. (Note that if $n_i = 0$, then $\Delta C_i = 0$ as well.)

- $\sigma u(k) \subseteq \{1,..., t\}$: Set of public function calls that trigger execution of the private function `Syndicate._updateCollateralizedSlotOwnersLiabilitySnapshot(k)`. Concretely, these public functions are:
    - `claimAsCollateralizedSLOTOwner(..., blsPubKeys)`, where KNOT k is mentioned in `blsPubKeys`
    - `updateCollateralizedSlotOwnersAccruedETH(k)`
    - `batchUpdateCollateralizedSlotOwnersAccruedETH(blsPubKeys)`, where KNOT k is mentioned in `blsPubKeys`
    - `unstake(..., blsPubKeys, ...)`, where KNOT k is mentioned in `blsPubKeys`
    - `deRegisterKnots(blsPubKeys)`, where KNOT k is mentioned in `blsPubKeys`
    - `informSyndicateKnotsAreKickedFromBeaconChain(blsPubKeys)`, where KNOT k is mentioned in `blsPubKeys`

  Note that $\sigma u(k)$ is assumed to only contain calls in which the following conditions hold at the beginning of the call:
    - k is active (i.e., `activateProposers()` has been called for k and `isNoLongerPartOfSyndicate[k] == false`)
    - k is slashed by less than 4 ETH

- $\sigma c(k, u) \subseteq \{1,..., t\}$: Set of calls to `Syndicate.claimAsCollateralizedSLOTOwner()` by user u for KNOT k. Initially, $\sigma c(k, u) = \emptyset$.

- $\sigma s_i(k, u)$: Amount of collateralized SLOT that user u has staked for KNOT k at the end of call i

- **Specific to rewards for free-floating SLOT holders:**
    - ΔF_i: Amount of ETH received in call i that goes to free-floating SLOT owners (i.e., those that have staked sETH). It is defined by
      $\Delta F_i = \Sigma_k \frac{\Delta R_k}{2}\ [n_i \neq 0 \wedge f_i \neq 0 \wedge k \leq i \wedge (\forall l \in \{k,..., i - 1\}.\ n_l = 0 \vee f_l = 0)]$.

      Most of the time, ΔF_i is just $\frac{\Delta R_i}{2}$ (i.e., half of the rewards go to sETH stakers). However, if the number of active KNOTs and the total amount of staked sETH have been zero, then ΔF_i also includes the accumulated rewards received during that time. (Note that if $n_i = 0$ or $f_i = 0$, then $\Delta F_i = 0$ as well.)

- f_i: Total amount of sETH staked via the Syndicate. Corresponds to `totalFreeFloatingShares`.
- $\varphi u(k, u) \subseteq \{1, ..., t\}$: Set of public function calls that update `sETHUserClaimForKnot[k][u]`. Concretely, these public functions are:
  - `stake()`
  - `unstake()`
  - `claimAsStaker()`
- $\varphi s_i(k, u)$: Amount of sETH that user u has staked for KNOT k at the end of call i

Contract state

- **KNOT status:**
  - `isKnotRegistered : mapping(bytes => bool)`
  - `isNoLongerPartOfSyndicate : mapping(bytes => bool)`
  - `activationBlock : mapping(bytes => uint256)`
  - `proposersToActivate : bytes[]`
  - `activationPointer : uint256`
  - `numberOfActiveKnots : uint256`
    Corresponds to n_t.
  - `isPriorityStaker : mapping(address => bool)`
- **Rewards for collateralized SLOT holders:**
  - `accumulatedETHPerCollateralizedSlotPerKnot : uint256`
    The total per-KNOT amount of rewards that have ever been earned for collateralized SLOT owners. Equals $\Sigma_{(a,b)\in U^{\times}} \Delta CperKnot_{[a,b]} = \Sigma_{(a,b)\in U^{\times}} \frac{\Delta C_{[a,b]}}{n_b}$ *where* $\Delta CperKnot_i = \frac{\Delta C_i}{n_i}$ *(if one of the denominators is zero, we treat the whole fraction as zero). Note that for* $a < i \leq b$*, it is ensured that* n_i *remains unchanged.*
  - `lastSeenETHPerCollateralizedSlotPerKnot : uint256`
    Total amount of rewards that have ever been received for collateralized SLOT owners. Equals $\Sigma_{i\leq max(U)} \Delta C_i$.
  - `totalETHProcessedPerCollateralizedKnot : mapping(bytes => uint256)`
    For KNOT k*, equals* $\Sigma_{(a,b)\in(\mathbf{0} \cup \alpha(k) \cup \sigma u(k))^{\times}} \Delta CperKnot_{[a,b]}$. *Intuitively, the condition*

$(a, b) \in (\mathbf{0} \cup \alpha(k) \cup \sigma u(k))^{\times}$ means the following: If $\alpha(k) = \{a\}$ (meaning k was activated in call number a) and $\sigma u(k) = \{u_1, u_2\}$ (meaning `_updateCollateralizedSlotOwnersLiabilitySnapshot(k)` *has been called in calls u_1 and u_2), then $(a, b) \in \{(0, a), (a, u_1), (u_1, u_2)\}$.*

- <u>`accruedEarningPerCollateralizedSlotOwnerOfKnot`</u> `: mapping(bytes => mapping(address => uint256))`
  For KNOT k and user u (where u is not the Syndicate contract), equals
  $\Sigma_{(a,b)\in(\mathbf{0} \cup \sigma u(k))^{\times}} \Delta CperKnot_{[a,b]} \cdot \frac{\sigma s_b(k,u)}{\Sigma_{u'} \sigma s_b(k,u')}$
  (if one of the denominators is zero, we treat the whole fraction as zero).
  In words: $\Delta CperKnot_{[a,b]}$ denotes the amount of rewards that have been assigned to k, and $\frac{\sigma s_b(k,u)}{\Sigma_{u'} \sigma s_b(k,u')}$ denotes the percentage of these rewards that are assigned to u.

  For KNOT k and user u (where u is the Syndicate contract)

  $RewardBucket_k = \Sigma_{(a,b)\in(\mathbf{0} \cup C_k)^{\times}} (4\,ether - \sigma S_b(k)) \cdot \Delta CperKnot_{[a,b]}$ *where*

  $C_k = \{m \in \sigma u(k) \mid \sigma S_m(k) \neq 4 \wedge (\forall l \in \sigma u(k).\ l > m \Rightarrow \sigma S_l(k) \neq 4)\}$

- <u>`claimedPerCollateralizedSlotOwnerOfKnot`</u> `: mapping(bytes => mapping(address => uint256))`
  For KNOT k and user u, equals
  $\Sigma_{a,b} \Delta CperKnot_{[a,b]} \cdot \frac{\sigma s_b(k,u)}{\Sigma_{u'} \sigma s_b(k,u')} \ [(a, b) \in \sigma u(k)^{\times} \wedge b \leq max(\sigma c(k, u))]$
  (if one of the denominators is zero, we treat the whole fraction as zero).
  This is very similar to the condition for <u>`accruedEarningPerCollateralizedSlotOwnerOfKnot`</u>*, except for the additional condition $b \leq max(\sigma c(k, u))$.*

- **Rewards for free-floating SLOT holders:**
  - <u>`accumulatedETHPerFreeFloatingShare`</u> `: uint256`
    The total per-sETH amount of rewards that have ever been earned for free-floating SLOT holders. Equals $\Sigma_{(a,b)\in U^{\times}} \Delta FperToken_{[a,b]} = \Sigma_{(a,b)\in U^{\times}} \frac{\Delta F_{[a,b]}}{f_b}$, where $\Delta FperToken_i = \frac{\Delta F_i}{f_i}$ (if one of the denominators is zero, we treat the whole fraction as zero). Note that for $a < i \leq b$, it is ensured that f_i remains unchanged.

- `lastSeenETHPerFreeFloating` : uint256
  Total amount of rewards that have ever been received for sETH stakers. Equals $\Sigma_{i \le max(U)} \Delta F_i$.

- `totalFreeFloatingShares` : uint256
  Total amount of sETH that has been staked to any KNOT in the Syndicate (this is modeled by f_i*). This amount corresponds to the sum of sETH that has been staked to the individual KNOTs, which is tracked by the following variable:*
  - `sETHTotalStakeForKnot` : mapping(bytes => uint256)
    Total amount of sETH that has been staked to a specific KNOT. This amount corresponds to the sum of sETH that has been staked to the KNOT by individual liquidity providers, which is tracked by the following variable:
    - `sETHStakedBalanceForKnot` : mapping(bytes => mapping(address => uint256))

- `lastAccumulatedETHPerFreeFloatingShare` : mapping(bytes => uint256)

  Equals $\sum_{(a,b) \in (\mathbf{0} \cup \alpha(k) \cup \delta(k))^{\times}} \Delta FperToken_{[a,b]}$

- `sETHUserClaimForKnot` : mapping(bytes => mapping(address => uint256))

  Equals $\sum_{(a,b) \in (\mathbf{0} \cup \varphi(k,u))^{\times}} \Delta FperToken_{[a,b]} \cdot \varphi s_b(k,u)$

- **Miscellaneous:**
  - `totalClaimed` : uint256
    Satisfies the following condition: `this.balance + totalClaimed` $= \Sigma_{i \le t} \Delta R_i$. *In words: The current contract balance together with* `totalClaimed` *equals the sum of all ETH deposits the contract has ever received. The only case in which ETH is transferred out of the contract (decreasing its balance) is when paying SLOT holders their rewards, in which case* `totalClaimed` *is increased by the same amount.*
    Thus, `totalClaimed` *equals* $(\Sigma_{i \le t} \Delta R_i) -$ `this.balance`.

Utility definitions:

- ```
  ActiveKnots = {b:bytes. b | isKnotRegistered[b] &&
                              !isNoLongerPartOfSyndicate[b] &&
                              (∃i:uint. proposersToActivate[i] == b &&
                                        i < activationPointer)}
  ```

General properties:

- For any KNOT, `isKnotRegistered` is only ever updated to true, never to false
  ```

- Elements are only ever added to `proposersToActivate`, never removed

Invariants

Invariant	**Inv-Syn-1** *[related to KNOT status]*
a) ∀i1:uint, i2:uint. i1 < proposersToActivate.length && i2 < proposersToActivate.length && i1 != i2 ==> proposersToActivate[i1] != proposersToActivate[i2] b) ∀b:bytes. isKnotRegistered[b] <==> ∃i:uint. proposersToActivate[i] == b c) ∀b:bytes. isNoLongerPartOfSyndicate[b] ==> isKnotRegistered[b] d) numberOfActiveKnots == |ActiveKnots|	
Description	a) proposersToActivate does not contain duplicate elements b) Every registered KNOT is in the proposersToActivate array. c) If a KNOT is no longer part of the LSD then it must be registered d) numberOfActiveKnots accurately reflects the number of active KNOTs

The following invariants are related to the rewards for free-floating SLOT holders.

Invariant	**Inv-Syn-**
totalFreeFloatingShares == Σ b ∈ ActiveKnots. sETHTotalStakeForKnot[b]	
Description	The total amount of free-floating sETH staked via the Syndicate is equal to the sum of sETH that has been staked to active KNOTs.
Notes	In the sum above, b must be restricted to ActiveKnots. This is because of two cases: 1. Someone staked to b before b was activated (this increases sETHTotalStakeForKnot[b] but not totalFreeFloatingShares). 2. b has been deregistered. In this case, totalFreeFloatingShares is reduced by sETHTotalStakeForKnot[b], but sETHTotalStakeForKnot[b] remains unchanged so that users can unstake.

Invariant	**Inv-Syn-**
∀ b:bytes. sETHTotalStakeForKnot[b] == Σ u:address.	

	`sETHStakedBalanceForKnot[b][u]`
Description	The total amount of sETH that has been staked to a KNOT is equal to the sum of sETH that all the individual users have staked to the KNOT.

Invariant	**Inv-Syn-**
`sETHToken.balanceOf(this) == Σ b:bytes. sETHTotalStakeForKnot[b]`	
Description	The amount of sETH held by the `Syndicate` contract equals the sum of sETH staked to all the individual KNOTs.

LPToken

Must satisfy the properties described by the ERC-20 specification. In addition, the following properties must be satisfied:

1. User balances do not change, unless they are affected by a call to `LPToken.{transfer(),transferFrom(),mint(),burn()}`.
2. Transfers made via `LPToken.{transfer(),transferFrom()}` do not take fees. In other words, whatever is deducted from the sender balance will be added to the receiver balance.
3. Only the deployer can call `LPToken.{mint(),burn()}`.
4. Self-transfers where the sender and the receiver are the same are not allowed.
5. The amount given to `LPToken.{transfer(),transferFrom(),mint(),burn()}` must be at least 0.001 LP tokens.
6. The amount given to `LPToken.{transfer(),transferFrom(),mint(),burn()}` must be a multiple of 0.001 LP tokens.
7. Whoever is passed as `_deployer` to the `LPToken.init()` function must implement the `ILiquidStakingManagerChildContract` interface.
8. If a transfer hook is passed to `LPToken.init()`, then the corresponding hooks must be called whenever one of `LPToken.{transfer(),transferFrom(),mint(),burn()}` is executed.
9. Whenever one of `LPToken.{transfer(),transferFrom(),mint(),burn()}` is called, then `LPToken.lastInteractedTimestamp` must be updated for all addresses that are involved.

Disclaimer

This report does not constitute legal or investment advice. The Blockswap Labs Formal Audit team (referred as "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 recommednt to make independent risk and security assessments.

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.