Formal Verification

Protocol: Stakehouse V1

Delivery: July, 2022

Join work by Certora and Blockswap Labs

CERTORA

Blockswap LABS

Overview

Blockswap Labs ("Blockswap") collaborated with Certora Inc ("Certora"), to conduct a joint formal verification of its Stakehouse mainnet protocol codebase. The work was undertaken from March - July 2022.

This document is a compilation of formal verification rules performed on the Stakehouse codebase using Certora Prover; both teams jointly reviewed adherence to the defined rules. The verification rules and test plan were mostly written by Blockswap with inputs from the Certora team in technical implementation and logical correctness assurance. It was developed as a supplement to the smart contract audits and unit tests to ensure Stakehouse's functional correctness requirements are met.

Formal verification proofs were designed to avoid false positive (vacuous) rules and to ensure that created specifications maximized the number of possible states the system can cover. The tool is very powerful with a vast array of execution paths to cover the bytecode of Stakehouse smart contracts. Although the tool is only as powerful as the written specifications, the tests were able to uncover results infeasible to traditional testing tools.

Throughout this engagement, the Certora team performed independent bug injections for testing violations and potential flaws. They generated the verification report/verification statement in accordance with Certora Prover best practices and requirements. All raised issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. Additionally, various safety checks recommended by Certora were implemented such as solidity asserts within the slashing logic to ensure the slashing operation completes as intended.

This document aims to provide objective and quality-assured performance data on Stakehouse’s functional correctness. Users, developers, and the security community can now make better-informed decisions about the Stakehouse protocol.

Stakehouse Mainnet Code Commit: 00e8999b50d786bd19942257af982aad36f9f66d

Scope of Contracts Covered:

- savETHRegistry
- SlotRegistry
- StakehouseRegistry
- AccountManager
- Safebox
- Transaction Router
- Balance Reporter

Below we go through the smart contracts that have been rigorously tested. The rules were written with focus on technical correctness and economic integrity of the system i.e. ensuring the issuance and redemption of dETH and SLOT functions behave as intended whilst also ensuring that account management and disaster recovery mechanics aren’t violated.

Engagement with Certora

Collaborating with Certora gave us an opportunity to greatly expand our knowledge of the EVM, obtain the necessary formal verification skills, and push the limits of testing decentralized finance protocols. Generally, there are a lot of unknown unknowns. Our aim was to try to get some predictability with a brand new, never seen registry paradigm. There are many unknowns, even at the compiler level (for example, Certora has found various bugs in different Solidity compiler versions over time). Our work and engagement with Certora is ongoing. We aim to continue testing the Stakehouse protocol and all future periphery smart contract work undertaken by Blockswap Labs.

Overall, whilst we did not encounter critical vulnerabilities while using the tool, the whole process has still been valuable from many perspectives:

- **Fixing small logic errors as well as optimizations;**
    - Code simplification - less code that does the same job should reduce the attack surface and contract size.
    - Introducing the idea of an entry and exit rule for deposits on savETH registry by having min and max deposit and withdrawal amounts.
- **Automated Bug injection** from Certora to find invalid or unreachable failed conditions (asserts) and then training the team on how to do this manually - Click Here for Report
- **Rule suggestions** from Certora - Click Here for Report
- **Specification reviews** - Click Here for Report
- **Weekly meetings** to ensure we are getting the most out of the tool.

We faced intricacies and challenges associated with storing **Boneh–Lynn–Shacham (BLS)** public keys from the Ethereum Consensus Layer (bytes are inherently unbounded data types). This presented additional constraints while working with storage pointer calculations in arrays and mappings. Bytes are used to store validator BLS public keys as per the Ethereum Deposit Contract. BLS12-381 public keys are alien to Solidity and the EVM, but native to the Consensus layer. They have a different form of identity from an ECDSA address which is a primary form of identification in Solidity.

Our dual factor registry tooling around the Ethereum Deposit Contract brings some other interesting challenges. For example, Consensus layer derivatives can't be obtained immediately after performing an ETH deposit. Derivatives can only be minted once the validator has been activated on the Beacon Chain which is one of the challenging cross-chain elements of the model. We have been able to use the Certora prover tool to implement checks in relation to vulnerabilities identified in our audits in order to ensure that the logic bugs have been fixed - this is invaluable. The Certora Verification Language (CVL) is the language used to write specifications for smart contracts and is given as input to the Certora prover tool. We leveraged powerful features of CVL including "ghosts", which are used to hook into the SSTORE opcode to track state changes and perform verification accordingly. Parametric invocation allows high coverage of code with minimal CVL lines of code compared to the equivalent size of required unit tests.

Another aspect of the protocol that is extremely hard to simulate is the queue mechanism for a leaked top-up for a specific validator on the Consensus layer. Ethereum deposit contract has a minimum deposit rule of 1 ETH where leaking is a much lesser amount and it may take months for the queue of SLOT top-ups to reach 1 ETH. The protocol has a queue flushing mechanism that allows users to jump the queue and dispatch the top-up to the validator, this comes with additional checks and adjustments to the adjusted active balance of a KNOT <> Consensus layer balance. We discovered that the state was not being reset when ETH was sent to the deposit contract which could lead to inventory accounting errors. Once the logic was corrected, we could use the Certora Prover to make sure that any time ETH is sent to the deposit contract, the queue is cleared and this can be seen as a property implemented for the Transaction Router smart contract.

Notation

For each tested smart contract, we provide a simple description of rules, invariants, and ghosts as well as a report generated by the Certora Verification Tool. The rules and descriptions are laid out in an easy-to-understand table within the specific smart contract section. Here is an example:

Rule Name:	Example Rule
Description:	X is not allowed to be more than 4

You can use the rule name to search in the Certora Verification report link provided with each contract section.

Disclaimer

This report does not constitute legal or investment advice. The teams behind this report from Blockswap Labs and Certora (collectively referred to as "preparers") of this report present it as an informational exercise documenting the due diligence involved in the functional correctness of listed contracts only and making 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 carry substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always reading the audits and other documents along with self-due diligence on deployed contracts.

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 review process is very real. The formal verification is carried out using a tool that performs simulated adversarial scenarios for verification that may significantly differ from real-time results. The

tool - Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and the Certora Prover does not check any cases not covered by the specification.

This report provides no warranty of any kind, explicit or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Blockswap or Certora or any of its employees be liable for any claim, damages, or other liability, whether in an action of contract, tort, or otherwise, arising from, out of, or in connection with the results reported here.

Smart Contract Verification

✅Verification of savETHRegistry.sol

The registry that controls the issuance and redemption of dETH and savETH tokens via indexes.

There are 2 verification reports attached with this smart contract. One is the core report with the majority of the rules that were verified. The other is the verification report for the ghost that was run against the contract in order to ensure that the sum of individual balances of KNOTs in indices does not exceed total dETH in all indices.

Core Verification results

Ghost Results

Rule Name:	**newKNOTSetsMintedFlagToTrue**
Description:	Supply only increases by 24 dETH once for a given KNOT

Rule Name:	**transferIndexOwnershipClearsApproval**
Description:	When index ownership is transferred to another ECDSA account, any approvals (for ownership transfer) are cleared

Rule Name:	**invariant_newKnotDoesNotMintdETHorSavETHTokens**
Description:	The balance added to the user index is a mintable balance (an optimistic UTXO)

Rule Name:	**newKNOTIsAddedToTheCorrectIndex** and **newKNOTHasCorrectDETHBalInIndex**
Description:	The KNOT is added to a chosen index and the 24 dETH mintable balance applied

Rule Name:	**newKNOTIncreasesDETHMintedInHouseByTwentyFour**
Description:	Total dETH minted within a house increases by exactly 24 for each KNOT added to the protocol

Rule Name:	**isolatingKnotFromOpenIndexReducesSavETHSupplyAndAddsDETHToIndex**
Description:	savETH is burned when isolating dETH into an index. savETH is burned at the correct exchange rate and the index balance for the knot is correct

Rule Name:	**addingKnotToOpenIndexMintsSavETH**
Description:	When the dETH from an index moves into the savETH registry but not into the open market, savETH shares are minted for the dETH brought into the open index

Rule Name:	**withdrawReducesSavETHSupplyAndIncreasesDETHSupply**
Description:	Withdrawing dETH from the registry mints new dETH ERC20 tokens and burns savETH for open index knots

Rule Name:	**depositDecreasesDETHSupplyAndIncreasesSavETHSupply**
Description:	Depositing dETH back into the registry reduces the ERC20 supply of dETH and mints savETH when leaving the assets in

	the open index (savETH has a claim to the mintable dETH balance in the open index)

Rule Name:	**transferKnotToAnotherIndexTransfersFullDETHBalanceToNewIndex**
Description:	When transferring assets between indices, no balance is left behind in old indices and approvals are cleared

Rule Name:	**addKnotToOpenIndexAndWithdrawClearsTheIndexAndIncreasesDETHSupplyOnly**
Description:	KNOT is added to open index and instead of keeping dETH in the open index and minting savETH, the dETH is completely brought into the open market so ERC20 dETH should only increase in supply and dETH in open index should not change

Rule Name:	**depositAndIsolateKnotIntoIndexDecreasesDETHTokenSupplyAndAddsTheBalanceIntoAnIndex**
Description:	Isolating a knot from the open index burns savETH and assigns correct amount of dETH to knot within an index

Rule Name:	**transferKnotToAnotherIndexClearsApproval**
Description:	When a knot, its tranche of dETH and inflation rights are moved to another index, any knot approval is cleared

Rule Name:	**transferIndexOwnershipInvalidatesKnotApproval**
Description:	Any individual knot tranches that have been approved do not carry over to new index owner when index ownership is transferred

Rule Name:	transferIndexOwnershipCorrectlyUpdatesOwner
Description:	When transferring index ownership, new owner address is correctly recorded and any existing approval is cleared

Rule Name:	openIndexRewardsCorrectlyRecorded
Description:	dETH inflation rewards are shared pro rata with open index savETH token holders

Rule Name:	dETHRewardsMintedCorrectlyRecorded
Description:	New inflation rewards increases dETH by the exact amount reported

Rule Name:	createIndexIncrementsTheIndexPointerByOne
Description:	Creating an index assigns new IDs each time to the account creating the index

Rule Name:	invariant_indexPointerIsStaticOutsideOfCreatingAnIndex
Description:	The zero address cannot own assets

Rule Name:	rageQuitKnotClearsBalances
Description:	Burns dETH in index for given KNOT, Reduces dETH in circulation, Reduces total dETH minted in house

Rule Name:	onlyAddingAKnotOrMintingInflationRewardsCanIncreaseCirculatingSupplyOfdETH
Description:	Ensuring only a new knot or new inflation rewards from the beacon chain are the only ways that dETH circulating supply can increase.

Rule Name:	**onlyAddingAKnotOrMintingInflationRewardsCanIncreaseTotalDETHMintedInHouse**
Description:	Ensuring only a new knot or new inflation rewards from the beacon chain are the only ways that the dETH minted within a Stakehouse registry can increase (Also known as total dETH minted within a house)

Rule Name:	**invariant_dETHInCirculationStaysConstant**
Description:	dETH supply stays constant unless new KNOT is added or inflation rewards added (i.e. there are no surprises around minting done from other methods)

Rule Name:	**invariant_totaldETHUnderManagementStaysSame**
Description:	Ensure that the set of state changing functions that modify total dETH under management in the open index is known

Rule Name:	**invariant_totalSavETHSupplyStaysSame**
Description:	Ensure that the set of state changing functions that modify total savETH supply is known

Rule Name:	**invariant_dETHBalanceInIndexIsAlwaysTwentyFourPlusRewards**
Description:	dETH Index balance for KNOT is == 24 dETH + dETH Rewards minted for KNOT

Rule Name:	**dETHUnderManagementAndTotalSavETHSupplyAreEitherZeroOrNonZeroAtTheSameTime**
Description:	dETHUnderManagement == 0 <=> totalSavETHSupply == 0 which means dETH under management in the open index

	is either zero at the same time savETH supply is zero or the two values are non zero.

Rule Name:	**invariant_nonCoreModuleShouldNotBeAbleToCallAnyStateChangeMethod**
Description:	Only adaptors can call state changing functions within the savETH registry

Rule Name:	**indexPointerOnlyIncreases**
Description:	Ensure that new indexes that are created are assigned an index from a pointer that is monotonically increasing

Ghosts implemented:

Ghost Verification Results

Ghost Name:	**sumOfAllDETHBalanceInIndicesDoesNotExceedTotalDETHInIndices**
Description:	The sum of all dETH balances of knots in an index does not exceed the total allocated to indices

✅Verification of SLOT registry

The registry that controls the issuance and redemption of SLOT and sETH for a given Stakehouse.

There are 3 verification reports attached with this smart contract. One is the core report with the majority of the rules that were verified. The other is the verification report for the ghost that was run against the contract in order to ensure that the sum of collateralized SLOT users own at the house level does not exceed total SLOT minted for house. Finally, some properties on the SLOT Registry were also verified via the Stakehouse Registry contract and that report is attached too.

Verification Result for SLOT Registry

Ghost Results

Stakehouse Registry + SLOT Verification Results

Rule Name:	**deployingStakeHouseTokenSetsUpMappingCorrectly**
Description:	When creating a new Stakehouse registry, an sETH token for the house is set up correctly

Adding a new KNOT:

Rule Name:	**newKnotSetsMintedFlagToTrue**
Description:	Ensuring that 8 SLOT is only minted once for a KNOT

Rule Name:	**activeSlotMintedInHouseIsBasedOnNumberOfActiveKNOTsThatHaveNotRageQuit**
Description:	Total minted in a house is always a multiple of 8 SLOT

Rule Name:	**activeCollateralisedSlotMintedInHouseIsBasedOnNumberOfActiveKNOTsThatHaveNotRageQuit**
Description:	Total collateralised SLOT in a house is always a multiple of 4 SLOT

Rule Name:	**newKNOTIncreasesCollateralisedSLOTByFourForRecipient**
Description:	Account adding the KNOT gets assigned 4 SLOT exactly for that knot in the vault, total collateralised SLOT owned at house level goes up by 4 SLOT too and they are added to the list of collateralised SLOT owners for that knot only once

Rule Name:	**rageQuitKnotClearsBalances**
Description:	Rage quitting correctly burns 8 SLOT for a given KNOT

Slashing:

Rule Name:	**slashingIsCorrectlyRecorded**
Description:	Slashing a KNOT reduces the collateralised SLOT balance for collateralised SLOT owner(s)

Rule Name:	**slashingIsCorrectlyRecorded**
Description:	Slashing a KNOT increases the total slashed at the house level by the amount slashed

Rule Name:	**slashingIsCorrectlyRecorded**
Description:	Slashing a KNOT increases the total slashed at the KNOT level by the amount slashed but up to the ceiling of 4 SLOT

Topping up slashed slot:

Rule Name:	**whenAlreadyCollateralisedOwnerForKnotTopupSLOTDoesNotAddOwnerAgain**
Description:	If already a collateralised SLOT owner for a KNOT, the address is not added again (list of owners must be a unique list)

Rule Name:	**toppingUpSlotNeverInflatesDETH**
Description:	Never mints dETH within a house

Rule Name:	**toppingUpSlashedSlotUpdatesSlashedAtHouseAndKnotLevel**
Description:	Accounting is correct following a top up (based on the amount being topped up, the correct amount of SLOT in the vault is allocated)

Slashing and topping up SLOT in the same transaction:

Rule Name:	noSlashingIsRecordedAtHouseLevelWhenSlashingAndToppingUpInOneTX
Description:	No slashing recorded at house level. Slashing and topping up in the same transaction leaves no slashed slot behind

Rule Name:	redemptionRateGoesUpWithSlashingAndDownWithTopUps
Description:	Any slashing at a knot level increases the redemption rate at the house level but decreases the redemption rate back towards the minimum of 3

sETH <> SLOT exchange rate:

Rule Name:	exchangeRateIncreasesAsDETHInHouseIncreases
Description:	increases as dETH minted within the house increases

Rule Name:	houseExchangeRateIsThreeToOneWhenZeroKnotsInHouse
Description:	When zero knots in a house after rage quit, the exchange rate is never less than 3:1

Invariants:

Rule Name:	invariant_slotBalanceInVaultIsNeverMoreThanFour
Description:	Total user collateralised SLOT balance for knot never exceeds 4 SLOT

Ghosts verification:

Ghost Verification Results

Ghost Name:	**sumAllSLOTInVaultsDoesNotExceedTotalCollateralisedInHouse**
Description:	Sum of all collateralised SLOT owned by a user in the collateralised vault for all knots is equal to the total collateralised SLOT owned by the user at the house level (individual balances for knots in the vault do not exceed the amount tracked at the house level)

✅Verification of StakeHouse Universe Factory

The smart contract that facilitates either spinning up a Stakehouse registry (simply a membership ledger of KNOTs) or joining a chosen registry (known as a Stakehouse).

Attached is the verification report results.

Verification Results

Rule Name:	**invariant_nonCoreModuleShouldNotBeAbleToCallAnyStateChangeMethod**
Description:	Only core modules can invoke state changing functions

Rule Name:	**numberOfStakeHousesIncreases**
Description:	Each deployment of a Stakehouse increases the total stakehouse pointer correctly

Rule Name:	**newHouseAssociatedWithCurrentIndex**
Description:	The correct index ID is appointed to the newly deployed house (house coordinates are formed correctly)

Rule Name:	**joiningAHouseMintsTheCorrectBatchOfTokens**
Description:	New knot correctly mints 24 dETH and 8 SLOT within the transaction (verified

	independently too at the savETH Registry and the SLOT registry level)

Rule Name:	**numberOfStakeHousesStaysStaticWhenAddingAMemberToAnExistingHouse**
Description:	Total number of houses is static when adding a member to an existing house (pointer only increases on house creation)

Rule Name:	**numberOfStakeHousesStaysStaticWhenAddingAMemberToAnExistingHouseAndCreatingABrand**
Description:	Total number of houses is static when adding a member to an existing house and creating a brand (pointer only increases on house creation)

Rule Name:	**calling_newStakeHouse_correctlySetsUpMemberHouseMapping**
Description:	New knot is correctly assigned to the chosen Stakehouse registry (including when house is created)

Rule Name:	**universeHasLessThanOrEqualStakehousesThanKnots**
Description:	Number of knots in the universe always upper bounds number of Stakehouses

Rule Name:	**numberOfKnotsUpperBoundedByNumberOfAccounts**
Description:	Number of accounts in the Account Manager always upper bounds number of knots

Rule Name:	**numberOfStakehousesUpperBoundedByNumberOfAccounts**

Description:	Number of accounts in the Account Manager always upper bounds number of houses

✅Verification of StakeHouse Registry

Smart contract deployed when a user creates their own Stakehouse and wants to spin up their own network of validators.

Verification Results

Rule Name:	**invariant_nonCoreModuleShouldNotBeAbleToCallAnyStateChangeMethod**
Description:	Only core modules can call state changing functions

Rule Name:	**exchangeRateIncreasesAsDETHInHouseIncreases**
Description:	Exchange rate increases as total dETH in house increases

Rule Name:	**newMemberIncreasesKnotPointerByOne**
Description:	New knot increases total number of knots in house by 1

Rule Name:	**newMemberIsAssignedCorrectValues**
Description:	New knots get assigned the correct metadata and are assigned the correct coordinates

Rule Name:	onlyAddMemberCanIncreaseIndexPointer
Description:	Only the add member function can increase the index pointer

✅Verification of Balance reporter

Smart contract that processes new information from the consensus layer.

Attached are 2 certora verification reports. The core results are run against the TransactionRouter which inherits the BalanceReporter abstract contract and the second runs further rules on a test harness of the BalanceReporter contract which exposes some helper functions for the tool.

Core Verification Results

Results for Balance Increase Invalidates Reporting Nonce

Rule Name:	topUpNeverIncreasesKnotActiveBalance
Description:	An unknown top up cannot inflate the adjusted active balance known to the smart contract

Rule Name:	topUpThatFlushesTheQueueIncreasesKnotActiveBalance
Description:	If there is any ETH below Ethereum Deposit Contract minimum 1 Ether deposit in the queue for a knot that was related to topping up slashed slot, then an unknown top up can flush the queue but only the portion of ETH related to topping up SLOT can increase the adjusted active balance for a knot reflecting the beacon chain

Rule Name:	toppingUpAndFlushingMechanismCanPayDownTheSpecialExitFee

Description:	The full special exit fee applied to a knot can be fully paid down through the in protocol top up mechanism leaving nothing in the queue and re-setting special exit fee to zero

Rule Name:	**partialPaymentOfSpecialExitFeePossible**
Description:	dETH holders can partially pay the special exit fee after it is applied leaving the remaining amount to be paid by SLOT holders. This directly addresses an issue found in one of the audits.

Rule Name:	**toppingUpSlashedSlotInvalidatesTheNonceForTheBLSPubKey**
Description:	When performing an unknown top up the nonce used for reporting is invalidated

Rule Name:	**reportingNonceInvalidatedWhenCallingBalanceIncrease**
Description:	The balance reporting nonce for the KNOT is incremented after reporting a balance increase for any given knot

✅Verification of SafeBox

Communication channel in the Common Interest Protocol requests.

Verification Report

Rule Name:	**onlyIncreasingDKGStatus**
Description:	Status during the distributed key generation procedure can only be increasing

Rule Name:	**incrementsHappenBy1**
Description:	Any method call can only increase the DKG LifecycleStatus status by 1

Rule Name:	**submitRound1DataIncrementsCorrectly**

Description:	Submitting round 1 data can only increment the index from 1 to 2

Rule Name:	**submitRound3DataIncrementsCorrectly**
Description:	Submitting round 3 data increments tatus from 3 to 4

Rule Name:	**submitDkgComplaintIncrementsCorrecty**
Description:	Submitting DKG complaint can only increment the status from 2 to 3 or from 4 to 5

Rule Name:	**submitNoComplaintIncrementsCorrectly**
Description:	Submitting no DKG complaint (to proceed further in the DKG process) increments the status 2 -> 3 or 3 -> 2

Rule Name:	**refuseGuardianDutiesOnlyWorksForGuardians**
Description:	Refuse guardian duties can only be executed for a guardian

Rule Name:	**joinGuardiansDoesNotWorkForGuardians**
Description:	Joining guardians for a second time is rejected

Rule Name:	**guardianIndexPointerOnlyIncreasing**
Description:	Index counting guardians can only increase

Rule Name:	**activeRequestLocksInternalNonceIncrements**
Description:	Block calling all the state-changing functions while the decryption request is active

Rule Name:	**bootstrapGuardiansAreRegistered**
Description:	All bootstrap guardians must be marked as

	registered, and this condition can't change

Rule Name:	**guardianIndexPointerMatchesRegisteredGuardianCount**
Description:	Guardian index pointer must always match the register guardian count, since that's what is being counted

Rule Name:	**relinquishedGuardiansWereOnceRegistered**
Description:	All guardians that refused duties were once registered guardians

Rule Name:	**complaintsAreOnlyPossibleByBootstrapGuardians**
Description:	Complaints are impossible to submit by non-bootstrap guardians

Rule Name:	**guardianIndexPointerIsZeroIffNoGuardiansExist**
Description:	Similar to the previous rule where we counted guardians and equated it to the index pointer

Rule Name:	**noUnauthorizedMethodChangesRegistrationStatus**
Description:	No method ever can change isGuardianRegistered mapping, since this tracks historical registrations

✅Verification of Account Manager

Core Module responsible for managing the essential data belonging to the KNOTs.

Verification Results

Rule Name:	**registerInitialsIncrementsLifecycleStatusByOne**
Description:	Making sure that the only status increment possible is 1

Rule Name:	createStakehouseIncrementsLifecycleStatusByOne
Description:	Making sure that the only status increment possible is 1

Rule Name:	joinStakehouseIncrementsLifecycleStatusByOne
Description:	Making sure that the only status increment possible is 1

Rule Name:	joinStakehouseAndCreateBrandIncrementsLifecycleStatusByOne
Description:	Making sure that the only status increment possible is 1

Rule Name:	elementIsPushed
Description:	Checking if account is pushed correctly into the accounts array in AccountManager

Rule Name:	accountArrayIndexSetCorrectly
Description:	Account array index correctness checks. Here we check if last registered account index is equal the account count

✅Transaction Router

An adaptor meant to route ether transactions and notify the AccountManager about relevant account changes.

Verification Report

Rule Name:	representativeAuthorized
Description:	Check if representative is authorized correctly

Rule Name:	noInitialRegistrationByNonRepresentative

Description:	Non-representatives can't register initials for some other actor

Rule Name:	**topUpQueueClearanceCompletesCorrectly**
Description:	Making sure that once ether is sent to the deposit contract from the topUpQueue it's completely cleared