

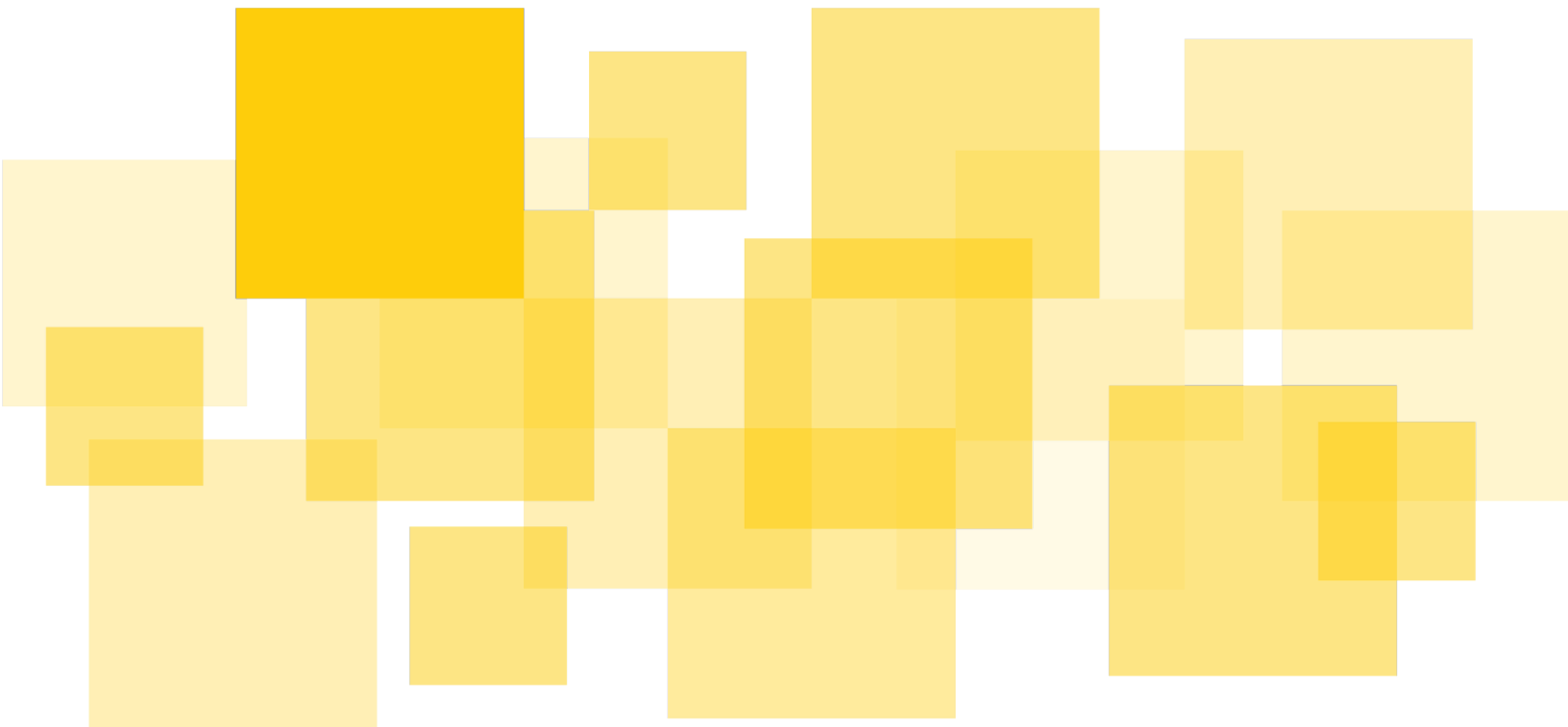
# Audit Report

---

## Blockswap Stakehouse (2nd audit)

**Delivered: 2022-06-06**

**Last updated: 2022-11-25**



Prepared for Blockswap by Runtime Verification, Inc.





[Summary](#)

[Disclaimer](#)

[Description of changes](#)

[Balance reporting](#)

[Topping up slashed SLOT](#)

[Applying correct penalty when KNOT is slashed](#)

[Encrypting a KNOT's signing key](#)

[Improved rounding errors in SlotSettlementRegistry](#)

[Potential reentrancy problem in StakeHouseRegistry](#)

[Approvals in savETHRegistry](#)

[Rage-Quitting: Special Exit Fee](#)

[SafeBox](#)

[Findings](#)

[A01: Deposit router fetches data from Subgraph node non-atomically](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

[A02: Deposit router fetches data from Stakehouse non-atomically](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

[A03: Deposit router fetches data from Beacon Chain non-atomically](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

A04: Beacon Chain node may be ahead of Subgraph node

Scenario

Recommendation

Status

A05: Paying the special exit fee can be mostly avoided

Scenario

Recommendation

Status

A06: AccountManager does not follow checks-effect-interactions pattern

Recommendation

Status

A07: SlotSettlementRegistry: exchangeRate() and sETHForSLOTBalance() behave inconsistently

Recommendation

Status

A08: Rounding errors in sETH contract can be reduced

Recommendation

Status

A09: Threshold value computed in the SafeBox contract may differ from whitepaper

Recommendation

Status

A10: SafeBox does not always check whether guardian has relinquished his duties

Recommendation

Status

Informative findings

[B01: Contract variables can be made constant](#)

[Recommendation](#)

[Status](#)

[B02: SafeBox contract is missing a function to reveal shares of malicious guardians during DKG protocol](#)

[Details](#)

[Status](#)

[B03: After paying the special exit fee, KNOTs are still negatively affected](#)

[Details](#)

[Scenario](#)

[Recommendation](#)

[Status](#)

[Appendix 1: Balance reporting mechanism](#)

[Requirements](#)

[Formalization](#)

[Correctness](#)

[Model](#)

[Proof sketch: IndexerWorld](#)

[Proof sketch: ActualWorld](#)

[Appendix 2: SafeBox](#)

[DKG protocol](#)

[Threshold decryption protocol](#)

# Summary

---

[Runtime Verification, Inc.](#) has audited the smart contract source code for Blockswap's Stakehouse protocol. The review was conducted from 2022-05-16 to 2022-06-03.

Blockswap engaged Runtime Verification in checking the security of their Stakehouse protocol. Stakehouse is a programmable staking layer built on top of the Ethereum 2.0 Beacon Chain. Unlike standard liquid staking solutions, Stakehouse does not run validators for the users or allows 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.

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](#). Details about the balance reporting mechanism and the on-chain part of the CIP protocol can be found in [Appendix 1](#) and [Appendix 2](#), respectively.

This is the second time Runtime Verification has audited the Stakehouse protocol.<sup>1</sup> The main goal is to ensure that the findings from the previous audit have been addressed and that the changes to the Stakehouse protocol do not have any unforeseen consequences.

## Scope

The following smart contracts were in scope of the audit:

- SlotSettlementRegistry
- savETHRegistry
- savETH
- CollateralisedSlotManager
- sETH
- Banking
- dETH
- savETHManager
- Streamer
- AccountManager
- BalanceReporter

---

<sup>1</sup> The first audit can be found here:

[https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Blockswap\\_Stakehouse.pdf](https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Blockswap_Stakehouse.pdf)

- TransactionRouter
- SignatureValidator
- StakeHouseUUPSModule
- UpgradeableBeacon
- BaseModuleGuards
- ModuleGuards
- SafeBox
- DKGRegistry
- StakeHouseRegistry
- StakeHouseUniverse
- StakeHouseAccessControls

Since this is a follow-up audit, we have not re-examined each of these contracts in detail. Instead, we have focused on the most significant changes as described in [Description of changes](#), and only analyzed the relevant code as needed. We have assumed correctness of the libraries and external contracts that are referenced. 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 `1b26fef97a9c4abaf1e557c5df520c0d86fcc8ba`. We also selectively evaluated some minor changes at commits `dad02c11ad9be82dad9d2d182f946717843d66a0` and `fa77f3d25e9d81a88c3837cefbb2f3b12fe7900d`, but did not perform reviews.

Blockswap also provided access to the `common-interest-protocol` and `deposit-router` private code repositories, which contain off-chain portions of the 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 issues in the high-level design. Details of the implementation of the off-chain code, as well as the security of the cryptography underlying the Common Interest Protocol, are not in scope for this audit.

## Assumptions

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

1. All users that have been assigned a role need to be trusted for as long as they hold that role. The intent is for all roles to be eventually revoked once there is confidence that the protocol is operating as intended and no more upgrades are needed.
2. The assumptions regarding the Common Interest Protocol (see previous audit report) are satisfied.
3. KNOTs who are penalized by more than 8 ETH are rage-quitted in a timely manner, ensuring that dETH is always fully backed by ETH.

4. dETH has enough liquidity and fluctuations of its value in the open market are not too large.

For more details on these assumptions see the previous audit report.

Note that assumption 1 roughly assumes 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.



# Description of changes

---

In the previous audit<sup>2</sup> we described the Stakehouse protocol in detail. In this section, we go over the major changes that have been made since then. Since we also want to mention which issues have been addressed, we often need to refer to findings from the previous audit. To avoid confusion, we prefix them with “P-”. For example, P-AO1 refers to the first finding of the previous audit, while AO1 refers to the first finding of this audit.

## Balance reporting

---

Here we describe the changes that were made to address findings P-A14, P-A17 and P-A18. For a description of the final balance reporting mechanism see [Appendix 1](#). Of particular interest in that section is the list of requirements that must be enforced in order for the balance reporting mechanism to be correct. It can be shown that any issue related to balance reporting that we found in the previous audit violates either the first or the second requirement. For convenience, we repeat these requirements here. For more details, see [Requirements](#).

**Req-1.** When the deposit router computes the adjusted active balance for a validator V and fetches `active_balance` from a Beacon Chain node and `total_unknown_top_ups` from an indexer node<sup>3</sup>, then both nodes must have processed the same deposits for V.

**Req-2.** When a report is submitted to the Stakehouse, then any information in the report must be at least as recent as any information the Stakehouse already has. Concretely, this means that any deposit known to the Stakehouse at the time the report is submitted must also have been known to the Beacon Chain node (and by Req-1 also by the indexer node) at the time the report was validated by the deposit router.

We now briefly go over the findings from the previous audit and describe how they have been addressed.

- P-A14 illustrates a problem where the deposit router would calculate the adjusted active balance incorrectly. This would happen whenever the indexer node had processed a deposit that was not yet processed by the Beacon Chain, which violates Req-1.

---

<sup>2</sup>

[https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Blockswap\\_Stakehouse.pdf](https://github.com/runtimeverification/publications/blob/main/reports/smart-contracts/Blockswap_Stakehouse.pdf)

<sup>3</sup> In the previous report, we used the term “Subgraph” instead of “indexer node”. Subgraphs are part of [The Graph](#) project and provide indexing services. We now use the more general term “indexer node” to emphasize that the Stakehouse protocol does not depend on one particular indexing service, and that while the client currently does use The Graph, this may change in the future.

To address this, the deposit router now performs an additional check, Chk-1, which is described in section [Requirements](#).

- P-A17 describes the following scenario: Assume there is a KNOT that has been slashed by 1 SLOT, i.e., its balance on the Beacon Chain and in the Stakehouse is 31 ETH. Next, Alice creates and signs (but does not submit) a report for this KNOT. This report contains the current validator balance, which is 31 ETH. Independently, assume someone tops up the 1 slashed SLOT for the KNOT, restoring it to full health and increasing its balance on the Beacon Chain and in the Stakehouse to 32 ETH. Now, if Alice submits the previously created report, then the balance of the KNOT in the Stakehouse is reduced again to 31 ETH, resulting in 1 collateralized SLOT being slashed. This is not fair, because now someone needs to top up this 1 slashed SLOT again, even though there was no new penalty on the Beacon Chain.

The above scenario violates Req-2, because the Stakehouse knows about a deposit (namely the one from topping up the 1 slashed SLOT) that was not known to the Beacon Chain at the time the report was created and signed by the deposit router. To address this issue, the Stakehouse now invalidates the internal nonce that it stores for each KNOT when topping up slashed SLOT. This ensures that reports created before the slashed SLOT is topped up cannot be submitted afterwards, because their nonce is now invalid.

- P-A18 describes a rather complex scenario, but the fundamental issue is that Req-2 was violated in certain cases. In particular, when an attacker created and signed a report immediately after someone had topped up slashed SLOT, it was possible that the report did not yet include the deposit that was made when topping up the slashed SLOT. On the other hand, the Stakehouse *did* know about this deposit, so submitting the report would result in overwriting the more recent information the Stakehouse already had with the outdated information in the report.

To address this issue, Chk-2c was introduced (see [Requirements](#)): For each KNOT, the Stakehouse now stores the deposit index of the latest deposit that it knows about. When validating a report, the deposit router then fetches this deposit index from the Stakehouse and requires that it is smaller than or equal to the deposit index fetched from the indexer node.

## Topping up slashed SLOT

---

P-A16 describes a problem in which the amount of slashed SLOT of a KNOT would not accurately reflect the amount of penalties that were incurred on the Beacon Chain. More specifically, the amount of slashed SLOT was less than the actual amount of penalties on the Beacon Chain, with no way to report the remaining penalty. For example, a KONT could have a balance of 30 ETH on the Beacon Chain, but in the Stakehouse it was only slashed by 1 SLOT. To

accurately represent the Beacon Chain balance, the KNOT should be slashed by 2 SLOT in the Stakehouse, but it was not possible to submit a report to that effect. Due to this, it was possible that a KNOT that had been penalized by more than 4 ETH was not kicked from the Stakehouse, even though this should have happened.

In particular, the issue was related to how the last seen balance of a KNOT was updated: For each KNOT, the Stakehouse keeps track of the latest reported adjusted active balance, also known as the *last seen balance*. When a penalty is reported to the Stakehouse, the amount of SLOT that should be slashed is calculated by taking the last seen balance and subtracting the reported balance (which is smaller than the last seen balance because of the penalty). For example, if the last seen balance is 32 ETH and the reported balance is 31 ETH, then the slashing amount is calculated by  $32 \text{ ETH} - 31 \text{ ETH} = 1 \text{ ETH}$ , which means the KNOT is slashed by 1 SLOT.

As it turns out, the problem was that when slashed SLOT was topped up, the last seen balance was not increased accordingly. This has now been addressed. (Incidentally, fixing this issue introduced more issues, namely P-A17 and P-A18. See [Balance reporting](#) for how these issues have been addressed.)

For every issue it is instructive to look for the concrete property that it violates. Knowing this property gives a principled approach to finding similar issues, both in the presence and in the future. In the case of P-A16, the following (slightly simplified) invariant was broken after topping up slashed SLOT:

$$k.\text{last\_seen\_balance} + k.\text{slashed\_SLOT} = k.\text{ideal\_balance}$$

This invariant states that for each KNOT  $k$ , the sum of its last seen balance and its amount of slashed SLOT should equal its ideal balance. (Note that for simplicity, we ignore a KNOT's top up queue and special exit fee here. In order for the above invariant to actually hold, we would need to factor in these two elements, but for illustration purposes we chose to leave them out.) The ideal balance of a KNOT is computed the same way as the last seen balance, except that it is unaffected by penalties. In this sense, it represents the best (or "ideal") balance the KNOT could possibly have. If no penalty has ever been reported for a KNOT, then the ideal balance equals the last seen balance. Note that the ideal balance is purely conceptual and is not actually stored in the Stakehouse protocol.

Now we can see why the above invariant should hold: The last seen balance represents the actual balance of the KNOT, while the amount of slashed SLOT represents the amount of penalties the KNOT has incurred. Added together, they should give us the ideal balance. However, in the original implementation, this invariant was broken when slashed SLOT was topped up. In particular, topping up slashed SLOT would decrease the amount of slashed SLOT, but would leave the last seen balance unchanged. If we assume the above invariant initially holds, then topping up slashed SLOT would only modify the amount of slashed SLOT of the KNOT, but leave the last seen balance and ideal balance unchanged, thus breaking the invariant. In

contrast, the new implementation increases the last seen balance by the same amount that the amount of slashed SLOT is decreased, which upholds the invariant.

## Applying correct penalty when KNOT is slashed

---

When a validator is kicked from the Beacon Chain after performing a slashable offense and this is reported to the Stakehouse, then the previous version of the Stakehouse protocol would assume a minimum penalty of 1 ETH. In P-A24, we described that this assumption does not hold in general because the slashing penalty is not fixed but depends on the effective balance of a validator. In particular, if the effective balance is below 32 ETH, then the slashing penalty is less than 1 ETH. Thus, in this case the Stakehouse would assume a larger penalty than was actually incurred on the Beacon Chain. In the current version, this has been addressed by not assuming any minimum penalty.

## Encrypting a KNOT's signing key

---

One task of the deposit router is to encrypt a KNOT's signing key with the shared public key of the Common Interest Protocol (CIP) when the KNOT is registered. Since the CIP was not fully implemented at the time of the first audit, the deposit router did not actually encrypt the KNOT's signing key but instead hashed it, making it impossible to decrypt the signing key later. See P-A20.

This has been addressed, and the deposit router now properly encrypts a KNOT's signing key with the CIP public key.

## Improved rounding errors in SlotSettlementRegistry

---

In the previous audit, we provided a detailed analysis of potential rounding errors in the `savETHRegistry` and also suggested and reviewed improvements that reduce the worst-case error bounds. The `SlotSettlementRegistry` performs similar calculations as the `savETHRegistry`, and similar improvements relating to rounding errors were made, but we did not have time to properly review these changes.

During this round of the audit, we did review these changes to `SlotSettlementRegistry` and found that all fixed-point calculations introduce minimal rounding errors of less than 1 wei, which is optimal. This is achieved by ensuring that division operations are always the last operation in a calculation.

We also examined how the functions of the `SlotSettlementRegistry` that involve fixed-point calculations are used by other contracts, because even if the rounding errors of the individual

functions are minimal, this does not necessarily imply the same for their composition. In this process, we found possible improvements for the sETH contract. See [Ao8](#).

## Potential reentrancy problem in StakeHouseRegistry

---

The creator of a Stakehouse has the ability to register a gatekeeper contract that is called whenever a new KNOT is about to join the Stakehouse. This allows the Stakehouse creator to block certain KNOTs from joining the Stakehouse.

Since the gatekeeper contract can contain arbitrary user-provided code, special care must be taken to ensure that reentrant calls to the Stakehouse cannot be exploited by an attacker. In the previous audit we found that at the time the gatekeeper contract is called, the Stakehouse is in an inconsistent state. Even though we did not find a concrete attack scenario, in finding P-A19 we recommended changing the code such that the Stakehouse cannot be observed in an inconsistent state.

Since the first audit, the code has been refactored to follow our recommendation. However, at the moment the code does not strictly follow the generally recommended checks-effect-interactions pattern. See [Ao6](#).

## Approvals in savETHRegistry

---

The Blockswap shared a problem with us related to how approvals are implemented in the savETHRegistry contract.

For context: All KNOTs in the Stakehouse protocol are either in the open index or in a private, user-owned index. The rewards of KNOTs in the open index are shared among all savETH holders, while the rewards of KNOTs in a private index go entirely to the index owner. An index owner can move the KNOTs in their index to other private indices (potentially owned by someone else) or to the open index. Additionally, an index owner can approve a third party to move the KNOTs in their index to other indices (including the open index). Finally, an index owner can also transfer ownership of their index to another user.

The problem the Blockswap team shared with us is as follows: Assume Alice owns an index and has approved Bob to move the KNOTs in her index to other indices. Now assume Alice transfers the ownership of her index to Charlie. In the original implementation, the approval made by Alice would still be valid, which means that Bob would still be able to move the KNOTs out of the index that is now owned by Charlie. For Charlie, this is probably very unexpected behavior, because he never approved Bob to move the KNOTs out of his index.

The above example suggests that approvals should only be valid if the current index owner is the same user who created the approval. This is also the intended behavior of the current implementation, and after analyzing the code we agree that this is indeed the case.

## Rage-Quitting: Special Exit Fee

---

The Blockswap team shared a new concept `specialExitFee(blsPubKey)` with us related to the handling of a special exit penalty when a black-swan event takes place.

When a KNOT joins a Stakehouse, 24 ETH out of the initial 32 ETH deposit are minted as dETH, and the other 8 ETH are minted as SLOT tokens. SLOT is the risk-bearing asset of the Stakehouse protocol. The 8 SLOT minted are split into 4 collateralized SLOT and 4 free-floating SLOT. The collateralized SLOT is available for slashing if any balance reduction events happen on the beacon chain consensus layer. Barring black-swan events, it should take very long for a KNOT to lose all of the 4 SLOT.

The main purpose of the `specialExitFee(blsPubKey)` is to track the amount of loss that goes beyond the 4 SLOT collateral allocated for the KNOT from slashing. When a KNOT is penalized in the Beacon Chain, the balance of the KNOT decreases and the same amount of collateralized SLOT is slashed. In order to rage-quit the protocol, any slashed SLOT must be topped up to restore the KNOT to full health and the amount of special exit fee must be paid. Also, a balance report is required to ensure that the KNOT's balance in the protocol is consistent with the Beacon Chain.

In addition to the special exit fee, the protocol also includes a waiting period for actual rage-quit a protocol. When the slashing event is first reported via `BalanceReporter`, the epoch number will be recorded. In order to rage-quit the protocol, the balance report has to state at least 4100 epochs since the slashing report.

During this round of audit, we reviewed the changes made to the `BalanceReporter` and `SlotSettlementRegistry`, and reexamined the invariants of `SlotSettlementRegistry` provided in the previous report.

In the case of P-B12, a KNOT that has been kicked might have a lower balance than is reflected in the Stakehouse. This is because the amount of collateral that can be slashed from a KNOT is capped at 4 SLOT. When a KNOT loses more than 4 of its collateralized SLOT of ETH in the Beacon Chain and gets kicked from the Stakehouse, the additional leakage of the Beacon Chain balance is not reflected by slashing SLOT. Introducing the special exit fee can cover the additional leakage addresses P-B12 when rage-quit a kicked KNOT.

The invariant below captures the relation between the balance of a KNOT in the Beacon Chain and the Stakehouse:



**Invariant:** If both the Stakehouse and the Beacon Chain are in sync, i.e., if they know about the same rewards, penalties, and deposits, then the following condition holds:  
 $dETH + SLOT \leq \text{active balance} + \text{queue} + \text{special exit fee}.$

- *dETH* is the total amount of dETH minted for the KNOT (24 dETH for the initial deposit + minted rewards).
- *SLOT* is the amount of SLOT remaining of the KNOT.
- *active balance* is the active balance of the validator on the Beacon Chain.
- *queue* is the amount stored in the KNOT's top-up queue.
- *special exit fee* is the amount of loss that goes beyond 4 SLOT collateral allocated for the KNOT from slashing.

The assumption that the Beacon Chain and the Stakehouse are in sync is required because the condition contains variables from both of these systems, hence they must be consistent with each other to make sense. Note that in the previous audit we proved a somewhat similar but different invariant. See [Bo3](#) for more details.

## SafeBox

---

The `SafeBox` contract (called `SafeBoxManager` at the time of the previous audit) implements the on-chain part of the Common Interest Protocol (CIP). The CIP allows anyone who owns more than 2 collateralized SLOT of a KNOT to request the KNOT's signing key. This mechanism is put in place so that a poorly performing KNOT can be taken over by the other members of a Stakehouse, who can then either run the KNOT themselves or exit the KNOT from the Beacon Chain in order to rage quit. See the previous audit for a more detailed description.

The main purpose of the `SafeBox` contract is to serve as a communication channel for the off-chain clients that are run by the guardians of the CIP. Functionality-wise, the contract has remained mostly the same compared to the previous audit, though the code has been refactored since then. Furthermore, the functions related to the committee handover protocol have been removed, though the plan is to support committee handover at some point after launch.

One problem identified in the previous audit was that you could not request the signing key of a KNOT that had been kicked from its Stakehouse. (Note that even a kicked KNOT is still part of the Stakehouse.) A KNOT is for example kicked from its Stakehouse if it has lost close to 4 collateralized SLOT. This means that if a KNOT has been abandoned by its original owner and is continually being penalized until its 4 collateralized SLOT have been burned and the KNOT is being kicked from its Stakehouse, then the other members of the Stakehouse have no way of taking over the KNOT in order to rage-quit and removing it from the Stakehouse. This issue is described in P-A23 and has since been fixed. Thus, users can now request the signing key of a KNOT even if the KNOT has been kicked.

See [Appendix 2](#) for the properties that the `SafeBox` contract guarantees.

# Findings

---

## AO1: Deposit router fetches data from Subgraph node non-atomically

---

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

When validating a Beacon Chain report for some validator  $V$ , the deposit router fetches, among other things, the following two data points from a Subgraph node:

- The deposit index of the latest deposit made to  $V$  via the Deposit Contract
- The total amount of unknown top ups for  $V$

These two data points are retrieved non-atomically using two separate network requests. Thus, it is possible that the state of the Subgraph node changes in between these requests. This can lead to a situation where the total amount of unknown top ups retrieved in the second request includes a deposit whose deposit index is larger than the one retrieved in the first request. In this case, a report may be created that causes  $V$  to receive a slashing in the Stakehouse even when  $V$  has incurred no penalty on the Beacon Chain.

### Scenario

1. Assume a freshly registered validator with a balance of 32 ETH
2. Alice tops up the validator by 1 ETH
3. Assume neither the Beacon Chain nodes nor the Subgraph node used by the deposit router have processed this deposit yet
4. Bob instructs the deposit router to sign a report. Assume the following happens:
  - The deposit router fetches the validator balance and the deposit count from the Beacon Chain node before Alice's deposit has been processed. Thus, the balance that the deposit router receives is 32 ETH
  - Also before Alice's deposit has been processed, the deposit router fetches the deposit index from the Subgraph node
  - To ensure that all deposits indexed by Subgraph have also been processed by the Beacon Chain, the deposit router checks that the deposit index from Subgraph is smaller than the deposit count from the Beacon Chain. Assume this check passes
  - Now, after Alice's deposit has been processed by the Subgraph node, the deposit router fetches the total amount of unknown top ups from Subgraph. This is now 1 ETH
  - To compute the adjusted active balance, the deposit router subtracts the total amount of unknown top ups (1 ETH) from the validator balance (32 ETH). Thus,



the adjusted balance is  $32 \text{ ETH} - 1 \text{ ETH} = 31 \text{ ETH}$ . However, note that the correct adjusted active balance would have been 32 ETH

5. Bob submits the report. Since the KNOT's balance in the Stakehouse is 32 ETH while the reported balance is 31 ETH, the KNOT is slashed by 1 SLOT

The problem is that in step 5), the KNOT is slashed by 1 SLOT even though there was no penalty on the Beacon Chain.

The main precondition that needs to be satisfied for this scenario is that the Subgraph node has indexed a deposit made to the Deposit Contract before that deposit has been processed by the Beacon Chain. This is practically always the case when you make a deposit, because the Subgraph node will usually index the deposit within seconds, while the Beacon Chain takes hours or even days. For an attacker, the main difficulty is to sign his report using the deposit router at exactly the right time to ensure that the Subgraph indexes the deposit between fetching the deposit count and the total amount of unknown top ups. The chances of this happening can be increased by making a lot of requests to the deposit router.

It is important to note that the scenario described above does not need an active attacker. Simply using the deposit router at an unfortunate time can result in the same undesired outcome. Even though this is unlikely to occur for a specific request to the deposit router, over many requests it seems probable to occur at some point.

## Recommendation

Fix the code such that the state of the Subgraph node cannot change between fetching the validator's latest deposit index and fetching the total amount of unknown top ups, or that such changes are reliably detected.

## Status

Acknowledged. The client intends to address this issue by fetching both the deposit index and the amount of unknown top ups from the Subgraph using a single, atomic query.

## Ao2: Deposit router fetches data from Stakehouse non-atomically

---

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

When validating a Beacon Chain report for some validator  $V$ , the deposit router fetches, among other things, the following two data points from the Stakehouse:

- The latest deposit index for  $V$  that the Stakehouse knows of
- An internal nonce for  $V$  that is used to prevent the same report from being submitted multiple times


These two data points are retrieved non-atomically using two separate network requests. Thus, it is possible that the state of the Stakehouse changes in between these requests. This can lead to a situation where the same penalty can be reported again after the slashed SLOT has been topped up.

### Scenario

1. Assume a freshly registered KNOT with a balance of 32 ETH
2. The KNOT is penalized by 1 ETH. This is reported to the Stakehouse, which updates the KNOT's balance to 31 ETH
3. Now Alice comes along and wants to create another report using the deposit router.

Assume the following happens:

- The deposit router fetches the deposit index from the Stakehouse and from the Subgraph. Then, the deposit router checks whether the deposit index from the Stakehouse is smaller than or equal to the deposit index from the Subgraph. The reason for this check is to ensure that if some deposit has been made via the Stakehouse, then in order to continue, we must wait until the Subgraph has indexed that deposit. Assume this check passes successfully
- Next, the deposit router calculates the adjusted active balance by fetching the validator's active balance from the Beacon Chain and the total amount of unknown top ups from the Subgraph. Since the validator balance is 31 ETH and there have not been any top ups yet, the adjusted active balance is  $31 \text{ ETH} - 0 \text{ ETH} = 31 \text{ ETH}$
- Bob comes along and tops up the 1 slashed SLOT of the KNOT. Thus, the KNOT's balance in the Stakehouse increases back to 32 ETH. Also, the internal nonce for the KNOT is invalidated
- The deposit router fetches the new nonce from the Stakehouse and signs the report

- 
4. Alice submits the report. Since the report uses the correct nonce, it is accepted by the Stakehouse. Because the KNOT's balance in the Stakehouse is 32 ETH while the reported balance is 31 ETH, the KNOT is slashed by 1 ETH, resulting in 1 slashed SLOT.

The problem is that in step 4), the KNOT is slashed again, even though Bob had already topped up the slashed SLOT.

## Recommendation

Fix the code such that the state of the Stakehouse cannot change between fetching the KNOT's latest deposit index and fetching the internal nonce, or that such changes are reliably detected.

## Status

Acknowledged. The client intends to address this issue.

## A03: Deposit router fetches data from Beacon Chain non-atomically

---

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

When validating a Beacon Chain report for some validator  $V$ , the deposit router fetches, among other things, the following two data points from the Beacon Chain:

- The balance of  $V$
- The deposit count, i.e., the total number of deposits to the Deposit Contract that have been processed by the Beacon Chain

These two data points are retrieved non-atomically using two separate network requests. Thus, it is possible that the state of the Beacon Chain changes in between these requests. This can make it possible to report a slashing for  $V$  to the Stakehouse even if  $V$  has not actually incurred any penalty on the Beacon Chain.

### Scenario

1. Assume a freshly registered KNOT with a balance of 32 ETH
2. Alice tops up the KNOT by 1 ETH
3. Assume the Subgraph has already processed the deposit, but the Beacon Chain has not
4. Bob instructs the deposit router to sign a report. Assume the following happens:
  - The deposit router fetches the validator balance from the Beacon Chain before Alice's deposit has been processed, i.e., the balance the deposit router receives is still at 32 ETH
  - Next, the deposit router fetches both
    - the deposit count from the Beacon Chain and
    - the deposit index from Subgraph

*after* Alice's deposit has been processed. In particular, this means the Subgraph knows about the 1 ETH top up made by Alice

- To ensure that all deposits indexed by Subgraph have also been processed by the Beacon Chain, the deposit router checks that the deposit index from the Subgraph is smaller than the deposit count from the Beacon Chain. Assume this check passes
- To compute the adjusted active balance, the deposit router subtracts the total amount of top ups (including Alice's 1 ETH deposit) from the validator balance (which does not yet include the 1 ETH deposit). Thus, the adjusted balance is 32 ETH - 1 ETH = 31 ETH. However, note that the correct adjusted active balance would have been 32 ETH

5. Bob submits the report. Since the KNOT's balance in the Stakehouse is 32 ETH while the reported balance is 31 ETH, the KNOT is slashed by 1 ETH

The problem is that in step 5), the KNOT is slashed by 1 ETH even though the validator has not incurred any penalty on the Beacon Chain.

Note that this scenario is very unlikely to occur in practice, because the Beacon Chain would need to process Alice's deposit in between the queries that fetch the validator balance and the deposit count. Further, this must not only happen once, but three times, because the deposit router fetches data from three different Beacon Chain nodes.

## Recommendation

Fix the code such that the state of the Beacon Chain node cannot change between fetching the validator balance and fetching the deposit count, or that such changes are reliably detected.

## Status

Acknowledged. The client intends to address this issue.

## AO4: Beacon Chain node may be ahead of Subgraph node

---

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

The deposit router currently only requires `Subgraph.deposit_index < BeaconChain.deposit_count`. While this ensures that any deposit indexed by the Subgraph node has also been processed by the Beacon Chain node, it does not ensure that any deposit processed by the Beacon Chain node has also been indexed by the Subgraph node. Thus, it is possible that a validator balance on the Beacon Chain includes a deposit that the Subgraph is not aware of. This in turn can lead to the adjusted active balance computed by the deposit router being too large. When reported to the Stakehouse, this would be treated as rewards and dETH would be minted even though no rewards have been earned on the Beacon Chain.

Note that the deposit router fetches data from the following systems:

- From a Beacon Chain node, the deposit router fetches the number of processed deposits (among other things)
- From a Subgraph node, it fetches the amount of unknown top ups
- From a Eth1 node, it fetches various data points from both the Stakehouse protocol and the Deposit Contract

The deposit router requires the number of blocks processed by the Subgraph node and the Eth1 node differ by at most 8 blocks.

For the issue described here to be possible, the Beacon Chain node must have processed a deposit faster than the Subgraph node. Since the Beacon Chain only considers deposits that are at least 2048 blocks old, this means that the Subgraph node lags behind the Eth1 chain by more than 2048 blocks. Additionally, because the Subgraph node and the Eth1 node used by the deposit router are required to be within 8 blocks of each other, this means that the Eth1 node must also lag behind the Eth1 chain by more than 2048 blocks.

Since this finding depends on two independent nodes to lag behind by more than 2048 blocks, it is unlikely to occur in practice.

### Scenario

The deposit router fetches data points from various sources:

- `DepositContract.deposit_count` denotes the total number of deposits that have been made to the Deposit Contract. It is fetched from the Eth1 node.
- `BeaconChain.deposit_count` denotes the total number of deposits to the Deposit Contract that have been processed by the Beacon Chain. It is fetched from the Beacon Chain node.

- `Subgraph.deposit_index` denotes the deposit index of the most recent deposit indexed by the Subgraph node. It is fetched from the Subgraph node.

The scenario is as follows:

1. Assume `DepositContract.deposit_count = 100` and `BeaconChain.deposit_count = 100`, i.e., the Beacon Chain has processed all deposits. Further assume `Subgraph.deposit_index = 50`. Thus, since `Subgraph.deposit_index < BeaconChain.deposit_count`, we know that all deposits indexed by the Subgraph have also been processed by the Beacon Chain.
2. Further assume there is a freshly registered validator with a balance of 32 ETH.
3. Alice makes a deposit of 1 ETH to this validator directly via the Deposit Contract. This means the deposit index of this deposit is 100, and the deposit contract is updated such that `DepositContract.deposit_count = 101`
4. The Beacon Chain processes Alice's deposit and is updated such that the validator balance becomes 33 ETH and `BeaconChain.deposit_count = 101`
5. Assume the Subgraph node and the Eth1 node are temporarily out of order. They still process queries, but they haven't processed any new blocks from Eth1 Mainnet for more than 2048 blocks. In particular, this means the Subgraph node has not indexed Alice's deposit yet. Thus, we still have `Subgraph.deposit_index = 50`
6. Alice wants to sign a report using the deposit router. Assume the following happens:
  - The deposit router checks whether the Subgraph node and the Eth1 node are within 8 blocks of each other. We assume that both systems stopped processing new data from Mainnet at around the same time, so this check passes
  - Next, the deposit router performs the following check to ensure all deposits indexed by the Subgraph node have also been processed by the Beacon Chain node:  

$$\text{Subgraph.deposit\_index} < \text{BeaconChain.deposit\_count}$$
 Since this is equivalent to  $50 < 101$  the check passes successfully.
  - To compute the adjusted active balance, the deposit router fetches the validator balance from the Beacon Chain, which is 33 ETH, and the amount of unknown top ups from the Subgraph. Since the Subgraph node has not yet indexed Alice's deposit, we have 0 ETH of unknown top ups. Thus, the adjusted active balance is computed as  $33 \text{ ETH} - 0 \text{ ETH} = 33 \text{ ETH}$
7. Alice submits the signed report to the Stakehouse. Since the validator was freshly registered, the last seen balance is still 32 ETH. Because the reported balance is 33 ETH, the Stakehouse mints 1 dETH in rewards.

The problem is that dETH should only be minted for rewards earned on the Beacon Chain, but in step 7) 1 dETH is minted even though the validator has not actually earned any rewards.



## Recommendation

Fix the code such that the deposit router computes the adjusted active balance correctly even if the Subgraph node lags behind the Eth1 chain by more than 2048 blocks.

## Status

Acknowledged. The client intends to address this issue.



## AO5: Paying the special exit fee can be mostly avoided

[ Severity: Medium | Difficulty: Low | Category: Security ]

If a KNOT is penalized by more than 4 ETH, then in addition to all the KNOT's collateralized SLOT being slashed, there is also a special exit fee that is applied. Before rage-quitting such a KNOT, the slashed SLOT needs to be topped up and the special exit fee needs to be paid. However, in the current version of the Stakehouse protocol, it is possible to almost completely avoid paying the special exit fee.

The problem occurs when topping up slashed SLOT using `BalanceReporter::topUpSlashedSlot()`. When calling this function, the following piece of code is executed:

```
if (msg.value > _amountOfSLOTPurchased) {  
    // If the ETH attached is more than amount needed for SLOT, then at least make sure that  
    // - The TX is at least 1 ETH so that funds are sent to the deposit contract  
    // - There is also additional excess attached to cover the special exit fee if there is  
    //   one  
    require(msg.value >= (1 ether + specialExitFee[_blsPublicKey]), "Invalid ETH");  
    excessETH = msg.value - _amountOfSLOTPurchased;  
    specialExitFee[_blsPublicKey] = 0; // msg.value has enough excess to pay for the special  
                                     // exit fee  
    sendFundsToDepositContract = true;  
}
```

The amount of slashed SLOT a user wants to purchase is stored in `_amountOfSLOTPurchased`. However, `topUpSlashedSlot()` allows a user to send more ETH than that, which is the case when `msg.value > _amountOfSLOTPurchased`. This excess ETH is then used to pay the special exit fee. However, as can be seen above, the special exit fee is always set to zero, even if the excess ETH is insufficient to fully cover the fee.

Note that a KNOT being penalized by more than 4 ETH is expected to be a very low-probability event. Thus, it is unlikely that the special exit fee is applied in the first place, which in turn means that the scenario described here is generally not likely to occur in practice.

### Scenario

1. A validator with an initial balance of 32 ETH is penalized by 5 ETH. When this is reported to the Stakehouse, the corresponding KNOT is slashed by 4 SLOT and the special exit fee is set to 1 ETH

2. A user wants to top up the slashed SLOT and calls `BalanceReporter::topUpSlashedSlot()`. He specifies that he wants to top up 4 slashed SLOT and transfers 4.1 ETH
3. This results in the KNOT being fully collateralized again (which is expected) and the special exit fee being set to zero (which is not expected)
4. Since the KNOT has now been brought back to full health it can be rage-quitted

Thus, the KNOT can be rage-quitted after paying only 4.1 ETH (and lower amounts are possible, as long as it is larger than 4 ETH), even though the intention is that 5 ETH need to be paid.

## Recommendation

Fix the code such that instead of always setting the special fee to zero, it is instead reduced by the amount of excess ETH that is provided.

## Status

Addressed in commit `fa77f3d25e9d81a88c3837cefbb2f3b12fe7900d`.

## A06: AccountManager does not follow checks-effect-interactions pattern

---

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

Before letting a KNOT join a Stakehouse, the AccountManager contract calls the function `isMemberPermitted()` on a user-provided contract to check whether the KNOT is allowed to join the Stakehouse.

```
function joinStakehouse(
    address _applicant,
    bytes calldata _blsPublicKey,
    address _stakehouse,
    uint256 _brandTokenId,
    uint256 _savETHIndexId
) external override onlyModule {
    _performStakeHousePreFlightChecks(_applicant, _blsPublicKey);

    // This executes isMemberPermitted(), which is provided by an untrusted user
    require(IMembershipRegistry(_stakehouse).isMemberPermitted(_blsPublicKey), "Blocked");

    _appendStakehouse(_blsPublicKey, _stakehouse, _applicant, _brandTokenId,
        _savETHIndexId);
    _setLifecycleStatus(_blsPublicKey, LifecycleStatus.TOKENS_MINTED);
    _storeReportDataForNewKnot(_blsPublicKey);

    emit CheckpointB2(_blsPublicKey);
}
```

As can be seen in the above (slightly abbreviated) code snippet, after the call to `isMemberPermitted()`, further updates are performed. This violates the checks-effect-interactions pattern, which recommends making calls to user-provided functions only at the end of a transaction, after all updates have been performed. A quick analysis did not reveal any potential attack scenario in this particular case, but we still recommend to follow the checks-effects-interactions pattern to be safe.

### Recommendation

Fix the code such that the call to `isMemberPermitted()` is called at the end of the `joinStakehouse()` function.

## Status

Addressed in commit [dad02c11ad9be82dad9d2d182f946717843d66a0](#).

## A07: SlotSettlementRegistry: exchangeRate() and sETHForSLOTBalance() behave inconsistently

---

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

For a given amount of X SLOT, the function `SlotSettlementRegistry::sETHForSLOTBalance(X)` returns value of X in sETH. Furthermore, the function `SlotSettlementRegistry::exchangeRate()` returns the current SLOT-to-sETH exchange rate. Intuitively, one would expect that `sETHForSLOTBalance()` is implemented using `exchangeRate()`. However, in order to reduce rounding errors, this is not the case. Nevertheless, one would still expect that  $\text{sETHForSLOTBalance}(X) \approx X * \text{exchangeRate}()$  holds at least approximately, but this is not always the case. In particular, when no SLOT has been minted (i.e., the Stakehouse is empty), then for any X we get

- `sETHForSLOTBalance(X) == 0`
- `X * exchangeRate() == X * 3`

When no SLOT has been minted, then `sETHForSLOTBalance()` simply returns zero while `exchangeRate()` returns the base exchange rate of 3.

Since `exchangeRate()` is not used by any contract in the Stakehouse protocol, this is not directly a problem. However, this inconsistent behavior might be confusing for users who use these functions directly.

### Recommendation

Fix the code such that `sETHForSLOTBalance()` and `exchangeRate()` behave consistently even when no SLOT has been minted.

### Status

Addressed in commit `dad02c11ad9be82dad9d2d182f946717843d66a0`.

## Ao8: Rounding errors in sETH contract can be reduced

---

[ Severity: Low | Difficulty: - | Category: Arithmetic Precision ]

The functions `slot()`, `activeBalanceOf()` and `slotForActiveBalance()` of the sETH contract all perform fixed-point calculations with nested division operations. This means that rounding errors may be amplified and visibly affect the final result. Since these functions are intended for users interested in additional information and are not actually used by any contract in the Stakehouse protocol, these rounding errors are not directly a problem. However, if the computed value deviates too much from the correct value, then this might be confusing to the users of the Stakehouse.

### Recommendation

Fix the code for `slot()`, `activeBalanceOf()` and `slotForActiveBalance()` such that the rounding error has a small and fixed upper bound.

### Status

Addressed in commit `dad02c11ad9be82dad9d2d182f946717843d66a0`.

## A09: Threshold value computed in the SafeBox contract may differ from whitepaper

---

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

The Blockswap team provided a whitepaper detailing the workings of the Common Interest Protocol (CIP). According to this document, for a CIP committee size of  $n$ , there is a threshold value  $t$  that is supposed to be  $t = \text{ceil}(n/2) - 1$  (using real arithmetic). However, in the SafeBox contract, the threshold is computed by `threshold = _participantCount / 2 - 1`. Since this is using integer arithmetic, it is equivalent to  $t = \text{floor}(n/2) - 1$  in real arithmetic, which differs from the threshold value expected by the whitepaper. In particular, if  $n$  is odd, then the threshold in the SafeBox contract is one smaller compared to the whitepaper.

### Recommendation

To match the whitepaper, the SafeBox contract should compute the threshold value as follows:  
`threshold = (_participantCount + 1) / 2 - 1`.

### Status

Addressed in commit `dad02c11ad9be82dad9d2d182f946717843d66a0`.

## A10: SafeBox does not always check whether guardian has relinquished his duties

---

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

The SafeBox contract acts as an on-chain communication channel for the off-chain CIP clients that are run by the CIP guardians.<sup>4</sup> As such, it provides functions that can be called by the CIP clients to send data to the other clients. These functions usually perform some checks and then emit an event that the other CIP clients can consume.

One of these checks ensures that a CIP client is only allowed to send data if the corresponding guardian has not relinquished their duties yet. However, this check is missing from `submitNoComplaint()` and `submitDecryption()`.

### Recommendation

Add the missing check to `submitNoComplaint()` and `submitDecryption()`.

### Status

Addressed in commit `dad02c11ad9be82dad9d2d182f946717843d66a0`.

---

<sup>4</sup> See the previous report for a detailed description of the CIP protocol.



# Informative findings

---

## Bo1: Contract variables can be made constant

---

[ Severity: - | Difficulty: - | Category: Gas optimization ]

The contract `SignatureValidator` defines the following variables (among others):

```
bytes32 private _REPORT_TYPEHASH;  
bytes32 private _CRYPTO_PACKET_TYPEHASH;  
bytes32 private _RAGE_QUIT_PACKET_TYPEHASH;
```

In the initialization they are initialized as follows:

```
_REPORT_TYPEHASH = keccak256(  
    "Report(bytes blsPublicKey,bytes32 reportHash,uint256 deadline,uint256 nonce)"  
);  
_CRYPTO_PACKET_TYPEHASH = keccak256(  
    "KeyEncryptionPacket(bytes blsPublicKey,bytes32 encryptionHash,uint256  
deadline,uint256 nonce)"  
);  
_RAGE_QUIT_PACKET_TYPEHASH = keccak256(  
    "RageQuitAuth(bytes32 rageQuitParamsHash,uint256 deadline,uint256 nonce)"  
);
```

Whenever one of these variables is accessed, one has to pay gas for the storage access. However, in this case it is actually possible to prevent these storage accesses by making these variables constant, which is more gas efficient.

## Recommendation

Declare the above mentioned variables as constant and initialize them inline.

## Status

Acknowledged. The client intends to follow the recommendation.

## Bo2: SafeBox contract is missing a function to reveal shares of malicious guardians during DKG protocol

---

The off-chain CIP clients (each of which representing a CIP guardian) use the `SafeBox` contract as a communication channel for the execution of various protocols. While we do not audit off-chain code, we do want to make sure that the `SafeBox` contract provides and correctly implements all the functionality needed by the CIP clients.

To this end, we examined if the `SafeBox` contract allows the CIP clients to send all the messages that are needed to implement the protocols described in the whitepaper provided by Blockswap. During this examination, we found that the `SafeBox` manager is missing a function that would let CIP clients reveal the shares of other, malicious clients in round 5 of the distributed key generation (DKG) protocol. However, even without this function, the DKG protocol can be implemented correctly, with the caveat that the provided security property is slightly weaker.

Cryptographic protocols are not in scope of this audit. Thus, we did not analyze this issue in more depth. However, the Blockswap team assured us that even though the provided security property may be slightly weaker, it can still be considered secure. Furthermore, because the CIP will initially be operated by Blockswap itself, the guardians can be considered honest.

### Details

We now try to give an intuitive explanation for why the missing functions lead to a slightly less secure protocol. For more details, see (Gennaro et al. 1999)<sup>5</sup>, which is also referenced by the whitepaper.

The final result of the DKG protocol is a public-private key pair  $PK, SK$  such that knowledge of  $SK$  is distributed among the guardians in such a way that no guardian ever has full knowledge of  $SK$ . Since the DKG protocol needs to cope with malicious guardians, there are mechanisms in place to detect invalid data during the generation of  $PK$  and  $SK$ . If such invalid data is sent by some guardian  $G$ , then the honest guardians will ignore this invalid data and disqualify  $G$ , which means  $G$  is not allowed to further participate in the protocol.

This approach of discarding data from malicious guardians is intuitive, but under certain circumstances it can also be used by an attacker to his advantage. The reason is that if the information from malicious guardians is discarded, then this influences the final result of the DKG protocol. This is obvious: If we exclude certain information from the calculations, then we get a different result than if we included it. Thus, if an attacker controls multiple guardians, he

---

<sup>5</sup> Gennaro, R., Jarecki, S., Krawczyk, H., Rabin, T. (1999). Secure Distributed Key Generation for Discrete-Log Based Cryptosystems. In: Stern, J. (eds) Advances in Cryptology — EUROCRYPT '99. EUROCRYPT 1999.

can in a limited way influence the result of the DKG protocol by choosing which of his guardians should get disqualified due to the submission of invalid data.

In the DKG protocol, there are two rounds in which invalid data can be detected: round 2 and 4. In the whitepaper, if a malicious guardian  $G$  is detected in round 2, it is disqualified, and any information sent by  $G$  is discarded. However – and this is the important part – if a malicious guardian  $G$  is detected in round 4, then the information sent by  $G$  is *not* discarded. The reason is as follows: In round 2, no important information has been shared yet. So an attacker who wants to influence the final result of the DKG protocol by deliberately letting some of his guardians get disqualified has simply not enough information to do this productively. Thus, simply discarding the information from malicious guardians in round 2 is safe. On the other hand, in round 4, relevant information has been shared among the guardians. At this point, the attacker *can* meaningfully influence the final result by deciding which of his guardians he wants to get disqualified. This is why if invalid data is detected in round 4, the honest guardians do not discard the data sent by the offending guardian  $G$ , but instead publish any information they received from  $G$ . This allows the honest guardians to continue the protocol as if the offending guardian had sent valid data. Thus, after round 2, the attacker cannot actually influence the final result any longer.

The above describes the DKG protocol as specified in the whitepaper. In contrast, in the current implementation, if a malicious guardian  $G$  is detected in round 4, then  $G$  is disqualified and all data sent by  $G$  is discarded. This means an attacker can use the information he learned in round 2 to decide which of his guardians should get disqualified in round 4. In this way, the attacker can deliberately influence the final result of the DKG protocol. Note that "influencing" here means the attacker can influence the probability distribution of  $SK$ , but this does not mean that the attacker gains knowledge of  $SK$ .

## Status

Since even without the missing function the DKG protocol can still be considered secure, and because initially the guardians are chosen by Blockswap and can thus be considered to be honest, the client decided to leave the code as is for the initial launch. However, the client intends to implement the missing function before the CIP is handed over to the community.

## Bo3: After paying the special exit fee, KNOTs are still negatively affected

---

The special exit fee and slashed SLOT are similar in that they both need to be paid for before rage-quitting. However, there is a difference in how they interact with earning rewards in the Stakehouse.

First, consider a KNOT that is penalized by less than 4 ETH. When this is reported to the Stakehouse, it results in the same amount of SLOT being slashed but does not lead to any special exit fee. To bring this KNOT back to full health, all the slashed SLOT needs to be topped up. Once this is done, the KNOT behaves like it had never been penalized in the first place: Any rewards that are earned on the Beacon Chain will mint the corresponding amount of dETH when reported to the Stakehouse.

On the other hand, now consider a KNOT that is penalized by more than 4 ETH. This results in all 4 collateralized SLOT being slashed. In addition, any penalty in excess of the 4 ETH is added to the special exit fee. To bring this KNOT back to health, both the slashed SLOT and the special exit fee need to be paid. However, once this is done, the KNOT does *not* behave like it had never been penalized. In particular, not all of the rewards on the Beacon Chain will mint the corresponding amount of dETH when reported to the Stakehouse.

From these two examples it follows that there is a fundamental difference between being penalized by up to 4 ETH and being penalized by more than that. In the first case, you can fully undo the effect of the penalty by topping up the slashed SLOT. However, in the second case, even after topping up the slashed SLOT and paying the special exit fee, the KNOT is still negatively affected.

Note that the lasting negative effect of the special exit fee is unlikely to occur in practice. The reason is that the negative effect is only related to earning rewards. Thus, it only affects KNOTs that are actually still able to earn rewards after being penalized by more than 4 ETH. This is only possible if more than 4 ETH are lost due to inactivity, which should take multiple years. (If they are lost due to a slashing offense on the Beacon Chain, then the KNOT cannot earn any rewards anymore.) During this time of inactivity, it is expected that the other members of the Stakehouse deal with the problematic KNOT in some way, for example by rotating ownership of the KNOT using the Common Interest Protocol.

### Details

The reason for this different behavior is that while topping up slashed SLOT increases the adjusted active balance, paying the special exit fee does not. In other words, paying the special exit fee is treated as an unknown top up. To illustrate the consequences of this, consider the

following invariant that we proved in the previous report in section C5 (slightly edited but equivalent):

**Invariant (previous audit):** Assuming the KNOT has not been kicked, the following condition holds:

$dETH + SLOT \leq \text{adjusted active balance} + \text{queue}.$

- $dETH$  is the total amount of dETH minted for the KNOT (24 dETH for the initial deposit + minted rewards).
- $SLOT$  is the amount of SLOT remaining for the KNOT.
- $\text{adjusted active balance}$  is the last active balance of the Beacon Chain registered in the Stakehouse (adjusted to ignore unknown top-ups).
- $\text{queue}$  is the amount stored in the KNOT's top-up queue.

This invariant only held for KNOTs that had not been kicked, i.e., that had been penalized by less than 4 ETH. The reason for this restriction was that at the time of the previous audit, the special exit fee did not exist yet, so any penalty above 4 ETH was not accounted for in the Stakehouse.

Since the new version of the Stakehouse protocol *does* account for penalties above 4 ETH by using the special exit fee, one might think that we could get rid of the assumption that the KNOT has not been kicked:

**Invariant (does not hold!):** For *all* KNOTs, the following condition holds:

$dETH + SLOT \leq \text{adjusted active balance} + \text{queue} + \text{special exit fee}.$

However, this invariant does not hold: Consider a freshly registered KNOT that is penalized by 5 ETH which is reported to the Stakehouse. Plugging the corresponding values into the above invariant gives us:

$24 \text{ dETH} + 4 \text{ SLOT} \leq 27 \text{ ETH} + 0 \text{ ETH} + 1 \text{ ETH}.$

So far so good. Now assume we top up the slashed SLOT and pay the special exit fee:

$24 \text{ dETH} + 8 \text{ SLOT} \leq 31 \text{ ETH} + 0 \text{ ETH} + 0 \text{ ETH}.$  (Does not hold!)

This is where the invariant breaks: Even though we paid 5 ETH in total (4 ETH to top up the slashed SLOT and 1 ETH for the special exit fee), the adjusted active balance on the right-hand side only increased to 31 ETH. This is because paying the special exit fee is treated like an unknown top up, which means it has no effect on the adjusted active balance.

Thus, the invariant from the previous report cannot easily be modified to hold for all KNOTs, even if it takes advantage of the special exit fee. However, under certain additional assumptions a somewhat similar invariant *can* be made to hold, see [Rage-Quitting: Special Exit Fee](#).

## Scenario

First, consider the following scenario only involving slashed SLOT and no special exit fee:

1. Assume a fresh KNOT with a balance of 32 ETH
2. The KNOT is penalized by 4 ETH without being kicked from the Beacon Chain. Its balance drops to 28 ETH and a report is submitted to the Stakehouse, resulting in 4 slashed SLOT
3. The 4 slashed SLOT are topped up. The adjusted active balance of the KNOT is brought back to 32 ETH
4. The KNOT earns 1 ETH in rewards on the Beacon Chain. Its adjusted active balance goes up to 33 ETH
5. When this is reported to the Stakehouse, 1 dETH is minted


Here, we have a KNOT that loses 4 ETH in step 2 but is brought back to full health in step 3. When the KNOT earns 1 ETH in rewards in step 4, this results in 1 dETH being minted in the Stakehouse. This is to say that even if a KNOT has been penalized, once it is brought back to health, it behaves like nothing has happened.

In contrast, consider the following example that involves the special exit fee:

1. Assume a fresh KNOT with a balance of 32 ETH
2. KNOT is penalized by 5 ETH without being kicked from the Beacon Chain. Its balance drops to 27 ETH and a report is submitted to the Stakehouse, resulting in 4 slashed SLOT and a special exit fee of 1 ETH
3. The 4 slashed SLOT are topped up and the special exit fee of 1 ETH is paid. The adjusted active balance of the KNOT increases to 31 ETH. Note that paying the special exit fee is treated like an unknown top up, so it does not affect the adjusted active balance.
4. The KNOT earns 1 ETH in rewards on the Beacon Chain. Its adjusted active balance goes up to 32 ETH
5. When this is reported to the Stakehouse, no dETH is minted, because the adjusted active balance does not exceed the highest seen balance.

The only difference compared to the previous scenario is that the KNOT is penalized by 5 ETH instead of 4 ETH. Since any penalty above 4 ETH is added to the special exit fee, this means that in order to bring the KNOT back to health, we need to (1) top up the 4 slashed SLOT and (2) pay 1 ETH for the special exit fee. When this is done in step 3, note that the adjusted active balance of the KNOT is only 31 ETH. So even though the KNOT has been brought back to health, its adjusted active balance is below the original balance of 32 ETH. The consequence of this is that when the KNOT earns 1 ETH in rewards it will not mint any dETH when reported to the Stakehouse.

To summarize: In the first scenario, a 4 ETH penalty was matched with a 4 ETH payment, after which the KNOT was behaving like the penalty did not happen. In the second scenario, a 5 ETH



penalty was matched with a 5 ETH payment, but after that the KNOT was still negatively affected.

## Recommendation

Document prominently, so that users know the different consequences for penalties of up to 4 ETH and for penalties of more than that.

## Status

Acknowledged.

# Appendix 1: Balance reporting mechanism

---

A Stakehouse contains a number of KNOTs, each of them representing a validator on the Beacon Chain. For each KNOT, the Stakehouse keeps track of the validator balance, whether the validator has been slashed, etc. However, since the Stakehouse protocol lives on the Eth1 chain, it cannot directly access information about validators from the Beacon Chain. Thus, in order to keep the information in the Stakehouse up-to-date, changes on the Beacon Chain must be explicitly reported to the Stakehouse. In general, anyone can submit a Beacon Chain report to a Stakehouse, though there are certain mechanisms in place to ensure that the submitted information is valid, i.e., does not deviate from the information on the Beacon Chain.

At a high level, a user wanting to submit a Beacon Chain report to a Stakehouse must perform the following steps (see the previous report for more details).

1. The user needs to create a Beacon Chain report containing the current information for a specific validator. How they do this is entirely up to them, but for simplicity Blockswap provides a utility app to generate such reports.
2. To ensure that only valid information enters a Stakehouse, all reports need to be signed by a trusted entity. To this end, the user needs to send their report to a *deposit router*, which is an off-chain server application that validates the information in a report against the current state of the Beacon Chain. If a deposit router deems a report to be valid, it returns a signed version of the report back to the user.
3. Once the user has a signed report, they can submit it to a Stakehouse using the `BalanceReporter` contract. Before accepting the report, the `BalanceReporter` checks that the entity that signed the report is trusted. The set of trusted entities can be managed on-chain by users who have the right permissions. If all checks pass, the state of the Stakehouse is updated with the information provided in the report.

One complication not mentioned yet is that the validator balance reported to the Stakehouse does not exactly match the active balance of the validator on the Beacon Chain. This is because the Stakehouse is *only* interested in balance increases due to earned rewards. This means that balance increases due to deposits (called *unknown top ups*) need to be filtered out, which gives us the *adjusted active balance* of a validator. In fact, any balance stored in the Stakehouse protocol is such an adjusted balance with any unknown top ups removed. However, there is one more twist: Not *all* deposits are filtered out. If a user tops up slashed SLOT from a validator that has been penalized, then the resulting deposit *does* affect the adjusted active balance.

In summary, when a report is submitted to a Stakehouse, the report does not simply contain the current balance of the corresponding validator. Instead, it contains the adjusted active balance, which is computed as follows:



$$\text{adjusted\_active\_balance} = \text{active\_balance} - \text{total\_unknown\_top\_ups}$$

where

$$\text{total\_unknown\_top\_ups} = \text{total\_top\_ups} - \text{SLOT\_top\_ups}.$$

Here, `active_balance` denotes the active balance of the validator on the Beacon Chain, from which we subtract all the unknown top ups made to that specific validator. The total amount of unknown top ups is computed by taking the total amount of top ups (i.e., the sum of *all* deposits to the Eth2 Deposit Contract for that particular validator) and subtracting all top ups that were made as a result of topping up slashed SLOT via the Stakehouse.

Since reports contain the adjusted active balance, it means the deposit router needs to be able to compute the adjusted active balance as well, in order to ensure the data in the report is correct. To this end, the deposit router needs to know both the active balance of the validator and the total amount of unknown top ups. The active balance can be easily fetched from the Beacon Chain, but retrieving the total amount of unknown top ups, `total_unknown_top_ups`, is a bit more complicated. To do this, we need to be able to compute both `total_top_ups` and `SLOT_top_ups`.

- `total_top_ups` is the sum of all deposits made to the Deposit Contract for one particular validator, but the Deposit Contract itself does not store this information directly. Fortunately, every deposit to the Deposit Contract emits a `DepositEvent` event containing the receiving validator and the deposit amount. These events can then be processed by an indexer<sup>6</sup> in order to compute `total_top_ups`.
- `SLOT_top_ups` is increased whenever slashed SLOT is topped up via the Stakehouse (more specifically, when calling `BalanceReporter::topUpSlashedSlot()`). Whenever this is done, the Stakehouse emits a `FundsSentToDepositContract` event containing the receiving validator and the amount of slashed SLOT that has been topped up. These events can be indexed as well in order to compute `SLOT_top_ups`.

To sup up, in order to calculate the adjusted active balance, the deposit router needs to fetch the following information:

- `active_balance` is fetched from a Beacon Chain node
- `total_unknown_top_ups` is fetched from an indexer node, which computes it based on the `DepositEvent` events emitted by the Deposit Contract and the `FundsSentToDepositContract` events emitted by the Stakehouse

When combining data from different systems, it is critical that all systems are in sync, or otherwise the result is nonsense. For example, when a deposit is made to the Deposit Contract, then an indexer node will usually process the emitted `DepositEvent` within seconds, while the

---

<sup>6</sup> Currently, this is done using a Subgraph (part of [The Graph](#) project), though this may change in the future.

time it takes the Beacon Chain to process that same deposit is measured in hours or even days. Thus, it is possible (even likely) that the total top ups as reported by the indexer node include deposits that have not reached the Beacon Chain yet. If this is the case, then the adjusted active balance computed by the deposit router will be too small, since it subtracts an unknown top up from the active validator balance even though the corresponding deposit has not reached the Beacon Chain yet. (This led to finding P-A14 in the previous report.)

## Requirements

---

When submitting a validator report to the Stakehouse, the report contains the adjusted active balance of the validator computed by the deposit router. Since the implementation of the deposit router is rather complex due to the fact that multiple asynchronous systems are involved, we want to precisely define the requirements that the deposit router needs to fulfill in order to be correct.

Even though we have not formally defined what it means for the reported adjusted active balance to be *correct* (see [Correctness](#) for a formal definition), we can still intuitively state the requirements that need to be fulfilled:

- Req-1.** When the deposit router computes the adjusted active balance for a validator *V* and fetches `active_balance` from a Beacon Chain node and `total_unknown_top_ups` from an indexer node, then both nodes must have processed the same deposits for *V*.
- Req-2.** When a report is submitted to the Stakehouse, then any information in the report must be at least as recent as any information the Stakehouse already has. Concretely, this means that any deposit known to the Stakehouse at the time the report is submitted must also have been known to the Beacon Chain node (and by Req-1 also by the indexer node) at the time the report was validated by the deposit router.

Intuitively, Req-1 ensures that the adjusted active balance is computed correctly, while Req-2 ensures that when submitting a report to the Stakehouse, then the state of the Stakehouse is only ever updated to a newer state and never to an older state (the state of the Stakehouse should move forward in time, never backwards).

In addition to the high-level general requirements above, there are also some more technical requirements:

- Req-3.** When the indexer has processed a `DepositEvent` that was emitted because someone topped up slashed `SLOT` via the Stakehouse, then the indexer must also have processed the corresponding `FundsSentToDepositContract` event that contains the amount of `SLOT` topped up.

**Req-4.** When the deposit router queries data from an external system (like the Beacon Chain, the indexer, etc), then it must be ensured that the state of the system does not change in between these queries. In other words, data must be queried atomically.

If Req-3 is violated, then the deposit router would treat a deposit as an unknown top up even though it was used to top up slashed SLOT. Further, if Req-4 is violated, then the data that the deposit router has retrieved is inconsistent, which can have all sorts of negative consequences.

To explain how these requirements are implemented, we need to introduce the concept of *deposit indices*: Each deposit to the Deposit Contract is associated with an index, the *deposit index*, which is simply a monotonically increasing counter starting at zero. The first deposit has an index of zero, the second index has an index of one, etc. Since the Beacon Chain keeps track of the total number of deposits it has processed, this gives an easy way to check whether a specific deposit has been processed. Concretely, this is the case if the deposit index of the deposit is smaller than the number of deposits processed by the Beacon Chain.

With this knowledge, we can now describe the checks that enforce the above requirements.

**Chk-1.** Req-1 is partly enforced by requiring that the deposit index of the most recent deposit processed by the indexer is smaller than the deposit count from the Beacon Chain. This ensures that every deposit known to the indexer is also known to the Beacon Chain.

Note that Req-1 is currently only partly enforced, because under some circumstances it is possible for a deposit to be known to the Beacon Chain but not known to the indexer. See [Ao4](#).

**Chk-2.** Req-2 is enforced by a combination of mechanisms:

- a. For each KNOT, the Stakehouse stores a nonce, which is fetched by the deposit router when signing a report. When submitting the report, the Stakehouse checks whether the nonce from the report matches the locally stored nonce. If this is the case, the report is accepted and the nonce invalidated. This ensures that reports can only be submitted once.
- b. Each report signed by the deposit router contains the Beacon Chain epoch number at the time the report was created. Additionally, for each KNOT, the Stakehouse stores the epoch number of the most recently submitted report. When a report is submitted, the Stakehouse requires that the epoch number from the report is larger than the locally stored epoch number. This ensures that the reported information is more recent than the previously reported information.
- c. When a user tops up slashed SLOT via the Stakehouse and the sent ETH is forwarded to the Deposit Contract, the Stakehouse stores the deposit index of this deposit. When validating a report, the deposit router fetches this deposit index from the Stakehouse and compares it with the deposit index from the indexer

node. Only if the deposit index from the Stakehouse is smaller than or equal to the deposit index from the indexer node will the deposit router sign the report. This ensures that if the Stakehouse knows about a deposit, then so does the indexer node.

**Chk-3.** Req-3 is ensured by the fact that (1) if a `DepositEvent` is emitted because a user topped up slashed `SLOT`, then the corresponding `FundsSentToDepositContract` event is emitted in the same transaction, and (2) the indexer node processes events emitted in the same transaction atomically.

**Chk-4.** Req-4 is currently not enforced by any checks. See [A01](#), [A02](#) and [A03](#).

It may not be obvious that these checks actually enforce all the requirements, or that the requirements ensure correctness, or what correctness even means. For these reasons, we have created a formalization of the balance reporting mechanism to help answer these questions.

## Formalization

---

In this section we discuss our formalization of the balance reporting process. The [Model](#) is written in pseudo-code inspired by both Solidity and more functional languages.

There are four independent systems to consider when talking about balance reporting:

- Stakehouse
- Deposit Contract
- Indexer
- Beacon Chain

Note that for the purposes of the formalization, we assume that the Indexer and the Stakehouse only contain a single `KNOT`. This is not a limitation, as the formalization could be easily extended to support multiple `KNOTs`, but assuming only a single `KNOT` makes the formalization more concise.

The state of each system can be completely defined by the sequence of events they have processed. Here, “event” does not only refer to events emitted from contracts (like the `DepositEvent` emitted by the Deposit Contract) but also to things like a validator receiving a reward or being slashed and kicked from the Beacon Chain. Basically, anything that changes the state of a system is considered an event.

For the purposes of balance reporting, the following events are of interest:

```
data Event = DepositEvent(uint deposit_index, uint amount, ...)
           | ValidatorEvent(...)
           | NextFinalizedEpoch
```

```
| NextNonce
| FundsSentToDepositContract(uint amount, ...)
```

A `DepositEvent` is emitted whenever a deposit is made to the Deposit Contract. A `ValidatorEvent` describes any event that affects a validator, except for deposits. This includes rewards, penalties, being kicked from the Beacon Chain, etc. The `NextFinalizedEpoch` event is used when the Beacon Chain has finalized an epoch. The `NextNonce` is used whenever the internal nonce in the Stakehouse is increased. Finally, `FundsSentToDepositContract` is emitted when topping up slashed SLOTS via the Stakehouse, and `amount` denotes the amount of slashed SLOTS that has been topped up.

The state of each of the above mentioned systems can be completely described by the sequence of events they have processed:

```
type State = List<Event>
```

The world state is then defined as follows:

```
struct World {
    State B; // Beacon Chain state
    State I; // Indexer state
    State D; // Deposit Contract state
    State S; // Stakehouse state

    // Actions
    function deposit(uint amount) { ... }
    function top_up_slashed_SLOT(uint amount) { ... }
    function beacon_validator_event(...) { ... }
    function beacon_next_epoch() { ... }

    // Synchronization
    function beacon_process_deposit() { ... }
    function indexer_process_deposit() { ... }
}
```

The world defines the exact semantics of the operations we are interested in. The functions `beacon_process_deposit()` and `beacon_next_epoch()` define how the Beacon Chain processes deposits made to the Deposit Contract, and what it means to move to the next epoch, respectively. Additionally, the function `beacon_validator_event(...)` is used to model various changes to the state of a validator, like rewards, penalties, etc. The function `indexer_process_deposit()` defines how the indexer processes the events emitted by the Deposit Contract and the Stakehouse. Finally, `deposit()` models deposits made to the Deposit Contract, and `top_up_slashed_SLOT()` models the behavior of `BalanceReporter::topUpSlashedSlot()`.

The two notable functions that are missing are those for creating and submitting reports. These are implemented in structs that derive from World. There are multiple such structs:

- `IdealWorld`: Models an ideal world in which there is no need for the Subgraph and the Beacon Chain state can simply be synchronized to the Stakehouse. Having such an ideal world makes it easier to see what we actually want to achieve with the additional code required in the actual world.
- `ActualWorld`: Models the behavior of the actual Deposit Router and Stakehouse.
- `IndexerWorld`: This model sits between the ideal and actual world. It does use an indexer node, but makes certain simplifications.

Each of these three worlds defines a `create_report()` and `submit_report()` function that performs the necessary checks to ensure correctness.

## Correctness

We want to ensure that the adjusted active balance computed by the deposit router and later submitted to the Stakehouse is correct, but what exactly does this mean? We define the adjusted active balance for a KNOT in a Stakehouse  $S$  as follows:

```
function adjusted_active_balance(State S) -> uint {  
    return 32 + total_SLOT_top_ups(S) + total_rewards(S) - total_penalties(S);  
}
```

Can this serve as *the* correct definition of the adjusted active balance? To convince ourselves that answering this question with yes is a good idea, we make the following observations: First, the Stakehouse knows about all SLOT top ups, because topping up slashed SLOT can only be done via the Stakehouse. Furthermore, since submitting a report to a Stakehouse simply means informing the Stakehouse about all Beacon Chain events, the Stakehouse also knows about a validator's rewards, penalties, and deposits. Finally, the above definition is only concerned with a single system, so we do not need to worry about different systems being "in sync". All of this implies that the Stakehouse has all the information available to compute the adjusted active balance, and all this information is in sync. Thus, it seems that the above function can indeed serve as the canonically correct definition of the adjusted active balance.

Having precisely defined what the adjusted active balance is, we can continue and define when balance reporting is correct.

Let  $r$  denote a report and  $S$  the state of the Stakehouse immediately before  $r$  is submitted. We use  $r.B$  and  $r.I$  to refer to the state of the Beacon Chain and of the indexer at the time the report was created, respectively. Finally, let  $S'$  denote the state of the Stakehouse immediately after  $r$  has been submitted. In other words,  $S'$  contains all the events in  $S$  as well as in  $r.B$ .

We say that balance reporting is *correct* if the following property is satisfied:

$$\text{active\_balance}(r.B) - \text{unknown\_top\_ups}(r.I) = \text{adjusted\_active\_balance}(S')$$

The intuition is as follows: When the deposit router calculates the adjusted active balance, it fetches the active balance from the Beacon Chain and then subtracts the total amount of unknown top ups retrieved from the indexer. Thus, the result of

$$\text{active\_balance}(r.B) - \text{unknown\_top\_ups}(r.I)$$

denotes the adjusted active balance as computed by the deposit router. We then simply require that the result matches our definition of the adjusted active balance in the Stakehouse state  $S'$  immediately after the report has been submitted.

## Model

The model makes use of the following definitions:

```
data Event = DepositEvent(uint deposit_index, uint amount, ...)
           | ValidatorEvent(...)
           | NextFinalizedEpoch
           | NextNonce
           | FundsSentToDepositContract(uint amount, ...)

type State = List<Event>

function deposit_count(State S) -> uint {
    return number of DepositEvent(...) in S;
}

function finalized_epoch(State S) -> uint {
    return number of NextFinalizedEpoch in S;
}

function total_rewards(State S) -> uint {
    return sum over all validator rewards;
}

function total_penalties(State S) -> uint {
    return sum over all validator penalties;
}

function active_balance(State S) -> uint {
    return active balance based on DepositEvents and ValidatorEvents in S;
}
```

```

function internal_nonce(State S) -> uint {
    return number of NextNonce events in S;
}

function deposit_index(State S) -> uint {
    return deposit_index of last DepositEvent in S, or zero if there is no such
    event;
}

function total_top_ups(State S) -> uint {
    return sum of ev.amount of all ev in S such that ev = DepositEvent(...);
}

function SLOT_top_ups(State S) -> uint {
    return sum of ev.amount of all ev in S such that ev =
    FundsSentToDepositContract(...)
}

function unknown_top_ups(State S) -> uint {
    return total_top_ups(S) - SLOT_top_ups(S) - 32;
}

function adjusted_active_balance(State S) -> uint {
    return active_balance(S) - total_top_ups(S) + SLOT_top_ups(S) + 32;
}

```

Using these definitions, we can now define the world state. Note that we make the following simplifying assumptions:

1. All functions are executed atomically
2. Reorgs never happen. This means events are only ever added to a state, never removed

Regarding the first assumption: The functions in this model formalize either functions from smart contract or from the deposit contract. In the former case, atomicity is a given. In the latter case, atomicity can be ensured with a careful implementation.

Regarding the second assumption: Since the deposit router waits for a deposit to be processed by the Beacon Chain, it generally only considers deposits that are at least 2048 blocks old, because that is the minimum amount of time before the Beacon Chain even considers the inclusion of a deposit. After this amount of time, reorgs can be disregarded.

```

struct World {
    State B; // Beacon Chain state
    State I; // Indexer state
}

```



```

State D; // Deposit Contract state
State S; // Stakehouse state


// Actions

function deposit(uint amount) {
    Event deposit = DepositEvent(deposit_count(D), amount);
    D.add(deposit);
}

function top_up_slashed_SLOT(uint amount) {
    Event deposit = DepositEvent(deposit_count(D), amount);
    Event SLOT_amount = FundsSentToDepositContract(amount);

    D.add(deposit);
    S.add(deposit);
    S.add(SLOT_amount);
    S.add(NextNonce);
}

function beacon_validator_event(...) {
    B.add(ValidatorEvent(...));
}

function beacon_next_epoch() {
    B.add(NextFinalizedEpoch);
}


// Synchronization

function beacon_process_deposit() {
    if number of DepositEvent events in D is larger than in B {
        Event deposit = first DepositEvent in D that is not in B;
        B.add(deposit);
    }
}

function indexer_process_deposit() {
    if number of DepositEvent events in D is larger than in I {
        Event deposit = first DepositEvent in D that is not in I;
        I.add(deposit);

        if number of FundsSentToDepositContract events in S is larger than in I {
            Event SLOT_amount = first FundsSentToDepositContract in S that is not in I;
            I.add(SLOT_amount);
        }
    }
}

```

```

    }
  }
}
}

```

Our definition for `World` misses the functions `create_report()` and `submit_report()`. In fact, we provide multiple definitions of these functions at different levels of abstraction. We start at the most abstract level in an ideal world. In an ideal world we do not need the indexer. Instead, we simply synchronize the whole Beacon Chain state to the Stakehouse when submitting a report:

```

struct IdealReport {
  State B;
}

struct IdealWorld extends World {

  function create_report() -> IdealReport {
    return IdealReport{B: B};
  }

  function submit_report(IdealReport r) {
    Add all those events to S that are in r.B but not in S (preserving order);
  }
}

```

Next, we define the indexer world, which is slightly less abstract than the ideal world:

```

struct IndexerReport {
  property adjusted_balance = {
    return active_balance(B) - unknown_top_ups(I);
  }

  // In reality, only `adjusted_balance` is part of the report, but to
  // prove correctness, we need some additional information. In particular, we
  // need to know the state of the Beacon Chain and of the Indexer at the
  // time the report was created.
  State B;
  State I;
}

struct IndexerWorld extends World {

  function create_report() -> IndexerReport {
    // Req-1: Check that Beacon Chain and Indexer are in sync, i.e., that they have
    //          both processed the same deposit events

```

```

    require(all DepositEvent events in I are also in B, and vice versa);

    return IndexerReport{B: B, I: I};
}

// We assume that `r` has been created using the `create_report()` function
function submit_report(IndexerReport r) {
    // Req-2: Check that the report is not out of date
    require(all DepositEvent and ValidatorEvent events in S are also in r.B);

    Add all those events to S that are in r.B but not in S (preserving order);

    // Assert correctness
    assert(
        active_balance(r.B) - unknown_top_ups(r.I) = adjusted_active_balance(S)
    );
}
}

```

Finally, we define the actual world that matches the actual implementation of the Stakehouse protocol:

```

struct ActualReport {
    property nonce = {
        return internal_nonce(S);
    }

    property adjusted_balance = {
        return active_balance(B) - unknown_top_ups(I);
    }

    // In reality, only `nonce` and `adjusted_balance` are part of the report, but to
    // prove correctness, we need some additional information. In particular, we
    // need to know the state of the Beacon Chain, the Indexer and the
    // Stakehouse at the time the report was created.
    State B;
    State I;
    State S;
}

struct ActualWorld extends World {

    function create_report() -> ActualReport {
        require(deposit_index(S) <= deposit_index(I) < deposit_count(B));

        return ActualReport{

```

```

    B: B,
    I: I,
    S: S,
  };
}

// We assume that `r` has been created using the `create_report()` function
function submit_report(ActualReport r) {
  // Check that the report is not out of date
  require(finalized_epoch(S) < finalized_epoch(r.B));
  require(r.nonce = internal_nonce(S));

  Add all those events to S that are in r.B but not in S (preserving order);
  S.add(NextNonce);

  // Assert correctness
  assert(
    active_balance(r.B) - unknown_top_ups(r.I) = adjusted_active_balance(S)
  );
}
}

```

One thing to note is that in this model, the state of the Stakehouse  $S'$  at the time a report is submitted is always at least as recent as the state of the Stakehouse  $S$  at the time the report was created. In other words,  $S'$  contains all the events from  $S$ . This is in line with our assumption that there are no block reorgs. To see why, assume that the Stakehouse state when the report is submitted is older than the Stakehouse state when the report is created. This would mean that when the report is submitted to the older Stakehouse state, that state would then fundamentally differ from the newer Stakehouse state. To reconcile both states, at least one reorg would be necessary.

## Proof sketch: IndexerWorld

We want to show that the correctness assertion in `IndexerWorld::submit_report()` holds. Thus, we need to show

$$\text{active\_balance}(r.B) - \text{unknown\_top\_ups}(r.I) = \text{active\_balance}(S') - \text{total\_top\_ups}(S') + \text{SLOT\_top\_ups}(S') + 32$$

where  $S'$  denotes the Stakehouse state at the end of the function.

To this end, it suffices to show the following:

1.  $\text{active\_balance}(r.B) = \text{active\_balance}(S')$

- First, note that the active balance depends only on DepositEvent and ValidatorEvent events
- By Req-2 we know that any such event in  $S$  is also in  $r.B$
- This implies that  $S'$  and  $r.B$  contain the same DepositEvent and ValidatorEvent events
- Thus,  $\text{active\_balance}(r.B) = \text{active\_balance}(S')$
- 2.  $\text{unknown\_top\_ups}(r.I) = \text{total\_top\_ups}(S') - \text{SLOT\_top\_ups}(S') - 32$ 
  - This is equivalent to
 
$$\text{total\_top\_ups}(r.I) - \text{SLOT\_top\_ups}(r.I) - 32 = \text{total\_top\_ups}(S') - \text{SLOT\_top\_ups}(S') - 32$$
  - First, we show  $\text{total\_top\_ups}(r.I) = \text{total\_top\_ups}(S')$ 
    - Note that the total top ups only depend on DepositEvent events
    - By Req-2 we know that any such event in  $S$  is also in  $r.B$
    - This implies that  $S'$  and  $r.B$  contain the same DepositEvent events
    - Req-1 implies that  $r.B$  and  $r.I$  contain the same DepositEvent events
    - The previous two points imply that  $S'$  and  $r.I$  contain the same DepositEvent events
    - This in turn implies  $\text{total\_top\_ups}(r.I) = \text{total\_top\_ups}(S')$
  - Next, we show  $\text{SLOT\_top\_ups}(r.I) = \text{SLOT\_top\_ups}(S')$ 
    - It suffices to show that  $r.I$  and  $S'$  contain the same FundsSentToDepositContract events
    - From the previous step, we already know that  $r.I$  and  $S'$  contain the same DepositEvent events
    - The definition of World implies that FundsSentToDepositContract events are added to the Stakehouse only in function `World::top_up_slashed_SLOT()`
      - This in turn implies that FundsSentToDepositContract events are only added together with a corresponding DepositEvent event
    - The definition of World implies that FundsSentToDepositContract events are added to the Indexer only in function `World::indexer_process_deposit()`
      - This in turn implies that FundsSentToDepositContract events are only added together with a corresponding DepositEvent event
    - Since  $r.I$  and  $S'$  contain the same DepositEvent events, and FundsSentToDepositContract events are only added together with a DepositEvent event, we can conclude that  $r.I$  and  $S'$  contain the same FundsSentToDepositContract events

## Proof sketch: ActualWorld

We want to show that the correctness assertion in `ActualWorld::submit_report()` holds, i.e., that after the function has executed, the following condition is satisfied:

$$\text{active\_balance}(r.B) - \text{unknown\_top\_ups}(r.I) = \text{adjusted\_active\_balance}(S)$$

We do this by showing that the requirements in `ActualWorld` imply the requirements in `IndexerWorld`. Because we have already shown that the above condition holds for `IndexerWorld`, this then implies the same for `ActualWorld`.

1. Show that the requirements in `ActualWorld` imply Req-1 from `IndexerWorld`
  - We need to show that all `DepositEvent` events in `r.I` are also in `r.B`, and vice versa
  - First, we show that all `DepositEvent` events in `r.I` are also in `r.B`
    - From `ActualWorld::create_report()` we know that `deposit_index(r.I) < deposit_count(r.B)`
    - `deposit_index(r.I)` denotes the deposit index of the last `DepositEvent` in `r.I`. Since each `DepositEvent` in `r.I` has a larger deposit index than the previous one, this also means `deposit_index(r.I)` denotes the `DepositEvent` with the largest deposit index in `r.I`
    - `r.B` contains all `DepositEvent` events with a deposit index less than `deposit_count(r.B)`
      - This follows from the fact that the sequence of `DepositEvent` events in `r.B` is an initial segment of the sequence of `DepositEvent` events in the Deposit Contract `D`. And in `D`, each `DepositEvent` has a deposit index that is larger by one compared to the deposit index of the previous `DepositEvent`, where the first `DepositEvent` event has a deposit index of zero
    - The last two points imply the claim
  - Next, we need to show that all `DepositEvent` events in `r.B` are also in `r.I`
    - This does not currently hold, see [Ao4](#)
2. Show that the requirements in `ActualWorld` imply Req-2 from `IndexerWorld`
  - We need to show that when `submit_report()` is called, the following condition holds:  
all `DepositEvent` and `ValidatorEvent` events in `S` are also in `r.B`
  - In other words, for any `ev` in `S`, we need to show that `ev` is also in `r.B`
  - Case `ev = DepositEvent`:
    - By the definition of `ActualWorld::create_report()` we know `deposit_index(r.S) < deposit_count(r.B)`
    - Because the deposit index can never decrease, we know `deposit_index(r.S) <= deposit_index(S)`

- Case  $\text{deposit\_index}(r.S) = \text{deposit\_index}(S)$ 
  - Then  $\text{deposit\_index}(S) < \text{deposit\_count}(r.B)$
  - We also know that  $\text{ev.deposit\_index} \leq \text{deposit\_index}(S)$
  - Thus,  $\text{ev.deposit\_index} < \text{deposit\_count}(r.B)$
  - Then the claim follows from the fact that  $r.B$  contains all deposits with a deposit index smaller than  $\text{deposit\_count}(r.B)$
- Case  $\text{deposit\_index}(r.S) < \text{deposit\_index}(S)$ 
  - This is only possible if between  $\text{create\_report}()$  and  $\text{submit\_report}()$ , one of the following functions has been called:
    - $\text{top\_up\_slashed\_SLOT}()$
    - $\text{submit\_report}()$
  - In either case,  $\text{internal\_nonce}(r.S) \neq \text{internal\_nonce}(S)$
  - Thus, the call to  $\text{submit\_report}()$  aborts and we are done
- Case  $\text{ev} \neq \text{DepositEvent}$ :
  - Then  $\text{ev}$  can only have ended up in  $S$  because of a previous call to  $\text{submit\_report}()$
  - Let  $r^*$  denote this earlier submitted report such that  $\text{ev}$  is in  $r^*.B$
  - Case  $r^*.B \sqsubseteq r.B$ :
    - Then  $\text{ev}$  is also known to  $r.B$  and we are done
  - Case  $r^*.B \not\sqsubseteq r.B$ :
    - Then  $\text{finalized\_epoch}(r^*.B) \geq \text{finalized\_epoch}(r.B)$
    - We also know  $\text{finalized\_epoch}(S) \geq \text{finalized\_epoch}(r^*.B)$
    - This implies  $\text{finalized\_epoch}(S) \geq \text{finalized\_epoch}(r.B)$
    - Thus, the call to  $\text{submit\_report}()$  aborts and we are done

## Appendix 2: SafeBox

---

Since most of the Common Interest Protocol (CIP) is implemented off-chain and not in scope of the audit, we can not fully analyze its correctness. What we can do instead is extract and document the properties that the on-chain code (i.e., the SafeBox contract) enforces. These properties can then be depended on by the off-chain code. In other words, the CIP as a whole is correct if the off-chain code (1) is correct and (2) only depends on properties of the SafeBox contract that are documented here.

As noted before, the main role of the SafeBox contract is to serve as a communication channel for the off-chain clients that are run by *guardians*, who make up the members of the CIP committee. In particular, the off-chain clients use the SafeBox contract as a communication channel for two protocols: The distributed key generation (DKG) protocol and the threshold decryption protocol.

The DKG protocol is executed once when the Stakehouse is initially deployed. It is jointly executed by an initial set of *guardians* who make up the initial CIP committee. The goal is to generate the CIP key pair consisting of the public key  $PK$  and secret key  $SK$ . Note that  $SK$  is never actually constructed anywhere in full. Instead, knowledge of  $SK$  is distributed among the members of the CIP committee such that each member  $i$  has a share  $sk_i$  of  $SK$ .  $PK$  is used by the deposit router to encrypt a KNOT's signing key.

The threshold decryption protocol is initiated when a user requests a KNOT's signing key. If the request is valid (i.e., the user holds more than 2 collateralized SLOT for the KNOT), then the members of the CIP committee can use their shared knowledge of  $SK$  to decrypt the KNOT's signing key and send it to the user.

In the next two sections we describe the properties that the SafeBox guarantees during the execution of the DKG and threshold decryption protocol.

### DKG protocol

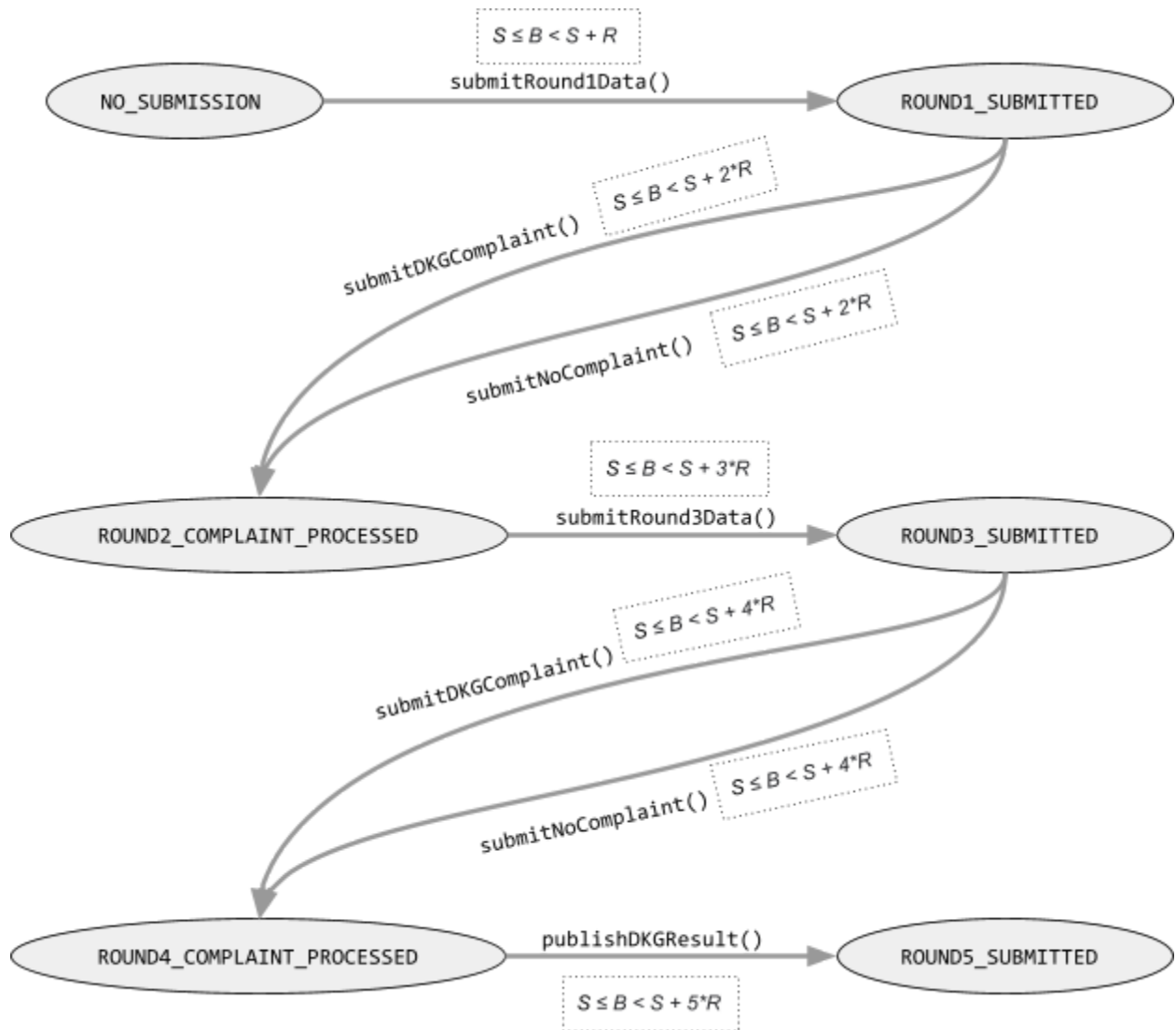
---

The DKG protocol consists of multiple rounds. In each round, the participating guardians exchange information with each other. The first round starts at a predefined block number  $S$ . In each round, a guardian must send its data within  $R$  blocks. If a guardian takes longer than that he can no longer participate in the protocol.

The figure on the next page illustrates the transition system that the SafeBox enforces for each guardian. Each guardian starts in state `NO_SUBMISSION`. Once the current Eth1 block number  $B$  satisfies  $S \leq B < S + R$ , guardians can call `SafeBox::submitRound1Data()` to submit their data



for the first round. Having done that, their state transitions to ROUND1\_SUBMITTED. The remaining transitions are similar.



Besides ensuring that the guardians correctly follow the above transition system, the SafeBox contract also imposes the following requirements:

- Only the guardians who were specified during the deployment of the SafeBox contract are allowed to participate in the DKG protocol. Guardians who have joined later are not allowed.
- Guardians who have relinquished their duties cannot further participate in the protocol (also see [A10](#)).
- The data sent by the guardians must pass various length checks.

## Threshold decryption protocol

The flow of the threshold decryption protocol is as follows:

1. A user (the recipient) requests a KNOT's signing key using `SafeBox::applyForDecryption()`. The following information must be provided:
  - The BLS public key and the Stakehouse of the KNOT
  - The public key  $pk_r$  of the recipient
2. A member of the CIP committee  $i$  who sees the request calls `SafeBox::submitDecryption()` in order to provide the following information:
  - $E_i$ , which denotes an encrypted partial decryption solution corresponding to the KNOT's signing key and is computed as follows:
    - Let  $C = (c_1, u)$  denote the KNOT's encrypted signing key. Here,  $u$  represents the actual ciphertext, encrypted using a symmetric encryption key  $k$ . Further,  $c_1$  is computed by encrypting  $k$  with the CIP public key  $PK$
    - Obtain the partial decryption solution  $D_i$  from  $c_1$  and  $sk_i$  (where  $sk_i$  is the guardian's share of  $SK$ )
    - Finally,  $E_i$  is obtained by encrypting  $D_i$  with the recipient public key  $pk_r$
  - A non-interactive zero-knowledge (NIZK) proof that can be used to verify the correctness of  $D_i$
3. Let  $n$  denote the number of members of the CIP committee and  $t = \text{ceil}(\frac{n}{2}) - 1$ . Then, for any committee member  $i$  who completed step 2, the recipient performs the following steps:
  - Decrypt  $E_i$  with his secret key  $sk_r$  in order to obtain  $D_i$
  - Verify the correctness of  $D_i$  using the provided NIZK proof

If at least  $t + 1$  committee members have completed step 2, then the recipient can combine the received  $D_i$  in order to obtain the decryption key  $D$ .

4. Finally, the recipient uses  $D$  to decrypt the encrypted signing key of the KNOT (which can be retrieved from the Stakehouse protocol)

Let  $V$  denote the number of blocks for which a decryption request is valid. We have verified that the `SafeBox` contract enforces the following properties:

- When a user decryption request for a KNOT's signing key by calling `applyForDecryption()`, then
  - the KNOT is active and has not rage-quit,
  - the KNOT has no slashed SLOT,

- the user owns more than 2 collateralized SLOT for the KNOT,
  - The user has not submitted any decryption request for the KNOT within the last  $V$  blocks.
- When a guardian submits information for a decryption request by calling `submitDecryption()`, then
  - the guardian must be registered with the `SafeBox` contract,
  - the decryption request must be less than  $V$  blocks old,
  - the guardian must not already have submitted any information for this specific decryption request,
  - the guardian must not have relinquished his duties (also see [A10](#)).