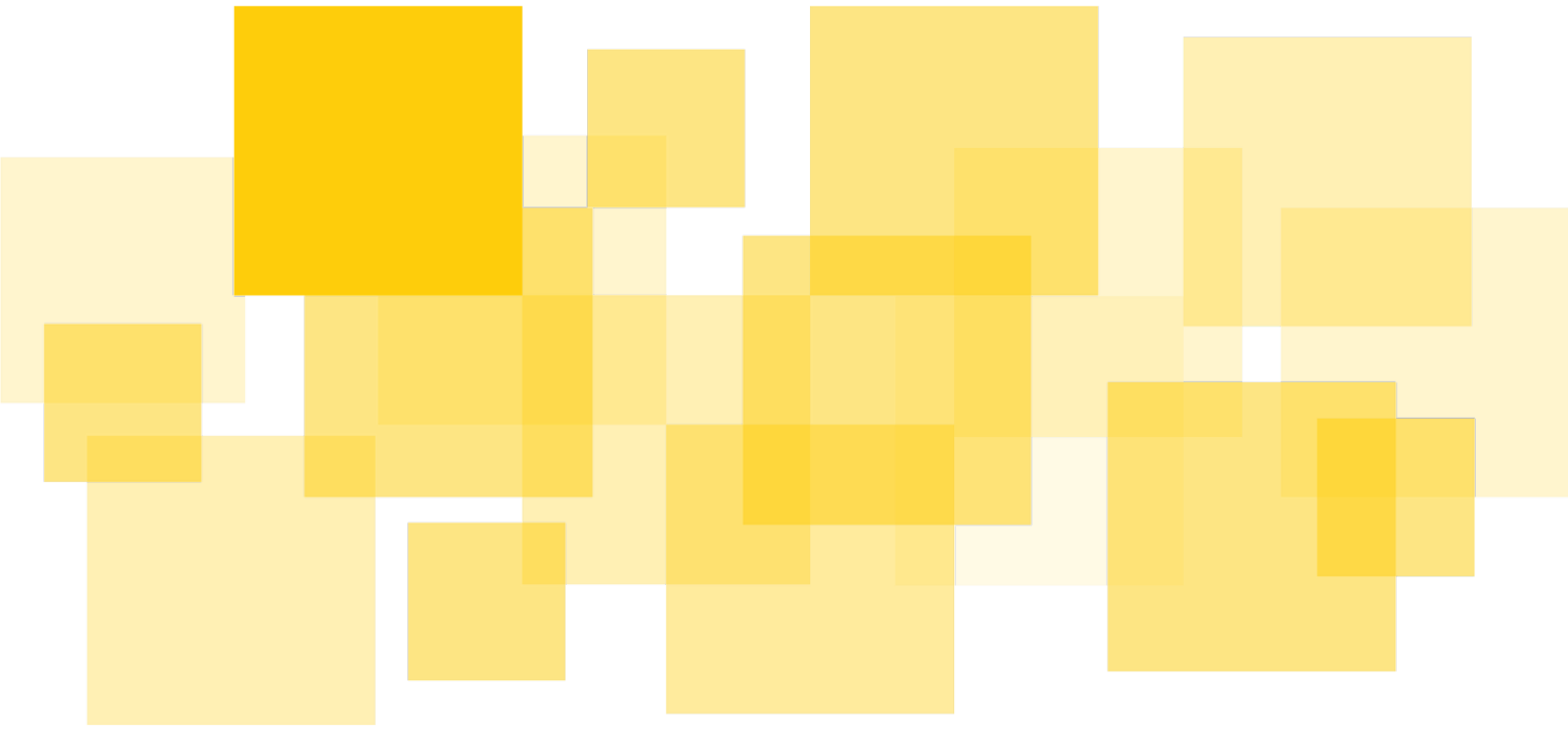
Security Audit Report

Swell

Delivered: May 2, 2022



Prepared for Swell Network by Runtime Verification, Inc.

Contents

Summary

Runtime Verification, Inc. has audited the smart contract source code for the Swell contract. The review was conducted from 2022-04-11 to 2022-04-29.

Swell Network engaged Runtime Verification in checking the security of their Swell contract, which is a permissionless liquid staking protocol. Anyone who provides at least 16 ETH can become a node operator, and anyone who provides at least 1 ETH can become a staker. Validator rewards are distributed pro rata among the node operator and the stakers after deducting a fee. Stakers additionally receive an amount of swETH that is equal to the amount of ETH they staked. swETH is a new token that is fully backed by staked ETH and can be used in other DeFi protocols while the staked ETH is locked. The goal is to have a 1:1 exchange rate between swETH and ETH on external markets. Further, once The Merge has taken place and withdrawals from the Beacon Chain become possible, the Swell contract will provide a way to redeem swETH for ETH at a guaranteed 1:1 exchange rate.

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 Informative findings. We have also verified a number of security properties, which can be found under Checked properties.

The code generally follows best practices and is easy to follow. It is important to point out that since not all functionality was implemented at the time of the audit, we were not able to evaluate the complete protocol. As such, we first tried to understand the intended behavior of the complete contract and then evaluated whether the already implemented functions are consistent with the overall design. However, this is *not* equivalent to auditing a finished contract, because verifying the functional correct of the existing code without knowing the exact assumptions made by the missing parts is not possible.

Scope

The audited smart contracts are:

- `contracts/swNFTUpgrade.sol` (ERC721 token and main contract)
- `contracts/swETH.sol` (ERC20 token)
- `contracts/swDAO.sol` (ERC20 token)
- `contracts/helpers.sol`
- `contracts/libraries/HexStrings.sol`
- `contracts/interfaces/ISWETH.sol`
- `contracts/interfaces/IStrategy.sol`
- `contracts/interfaces/ISWNFT.sol`

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

The review encompassed one private code repository, which contains a Hardhat project with test scripts. The code was frozen for review at commit `2a3a07969cba13c04a16f41aa3d1b17aa6acfe1b`.

The review is limited in scope to consider only contract code. Off-chain and client-side portions of the codebase are not in the scope of this engagement.

Assumptions

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

- The owner (deployer) is absolutely trusted. (The owner can call privileged functions and also upgrade the contract.)
- The total amount of penalties that a non-whitelisted validators incurs does not exceed 16 ETH. (The DAO would be responsible for handling such cases, but because this is not implemented yet, we assume this never happens.)
- Whitelisted validators are never penalized. (The DAO would be responsible for handling such cases, but because this is not implemented yet, we assume this never happens.)

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

Methodology

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

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

Disclaimer

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

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

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

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

Background: Beacon Chain and Liquid Staking

Ethereum is currently in the process of introducing a proof-of-stake consensus mechanism, which uses validators instead of miners to reach consensus about which new blocks to include in the blockchain. To run a validator, you need to lock up 32 ETH in a process called *staking*. If a validator performs its duties according to the consensus protocol, it earns rewards. On the other hand, if a validator deviates from the protocol, it is penalized. Depending on the severity of the offense, a validator might even be banned from further participation, which is called *slashing*. Rewards earned by the validator increase the initially provided stake, while penalties decrease it. Only once a validator no longer participates in the protocol (either by exiting voluntarily or by being slashed) can the staked ETH be withdrawn. This mechanism incentivises users who run validators, also called *node operators*, to follow the protocol, as otherwise they may lose (part of) their stake.

Each validator has a signing key that is used to sign messages, and a corresponding public key, which is used to uniquely identify the validator. Another important piece of information associated with a validator is who will receive the deposited stake and rewards once the validator has exited. The receiver can either be denoted by an account on the Beacon Chain or by an Eth1 address.

Proof-of-stake is currently only available on the *Beacon Chain*, a relatively new chain that is still separate from Ethereum Mainnet. The intention is to connect both chains soon in an upgrade called The Merge. Despite this separation, staking the required 32 ETH to a validator is possible via the Eth2 Deposit Contract that is deployed on Mainnet. However, moving funds in the opposite direction is not yet possible. This means that withdrawing a validator's stake and rewards to an Eth1 address only becomes possible after The Merge.

While running a validator provides a relatively safe way to generate income, the disadvantage is – even after The Merge – that both the initial stake of 32 ETH and any rewards only become accessible once the validator has exited. This is where liquid staking protocols come in: Users who stake through such a protocol immediately receive some derivative token in return that can be freely used in other DeFi protocols while the staked ETH is locked up. The value of this derivative token derives from the fact that it is backed 1:1 by staked ETH, and can be redeemed once the validator has exited and the staked ETH becomes available.

Contract description and invariants

This section describes the Swell contract at a high-level, and which invariants we expect to hold at the end of every contract interaction. Note that for a better understanding, we describe the intended behavior of the *complete* protocol, including the parts that have not been implemented yet. However, we point out any parts that were still missing in the version of the code that this audit is based on.

Overview

The Swell protocol is a permissionless liquid staking protocol. Anyone who owns at least 16 ETH can become a node operator in Swell, and anyone who owns at least 1 ETH can become a staker. When a node operator or staker deposits ETH to some validator via the Swell contract, they receive a swNFT in return, which denotes their staking position. A staking position includes information such as the validator that was staked to and the amount that was staked. Additionally, whenever a swNFT is minted for a staker (but not for a node operator), an amount of swETH that is equivalent to the staking amount is minted. swETH is a derivative token that is backed by staked ETH and can be used instead of it while the staked ETH is locked up. The reason that node operators do not get any swETH is that their deposit is used as a buffer against penalties. Since node operators are required to deposit at least 16 ETH, this means that validators can be penalized by up to 16 ETH without endangering the desired 1:1 redeemability of swETH for ETH.

While a validator is running, rewards are distributed pro rata among the node operator and the stakers based on the amount they staked. Since node operators always stake 16 of the 32 ETH needed to run a validator, they receive half the rewards. Similarly, a staker who has deposited 8 ETH would receive 25% of the rewards. In contrast, penalties are entirely the responsibility of the node operator and will be deducted from his 16 ETH deposit.

As mentioned, when a staker stakes to a validator, two types of tokens are minted: a swNFT denoting the staking position, and the staking amount in swETH. However, this swETH is not directly transferred to the staker but instead held by the Swell contract. The minted swETH is connected to the swNFT, and the owner of the swNFT can do mostly two things with it: (1) Withdraw the swETH to his own address, or (2) instruct the Swell contract to put (part of) the swETH into a vault (also called strategy) for investing in other DeFi protocols.

For node operators, using the Swell contract typically consists of the following steps:

1. Generate a new validator key pair
2. Deposit 16 ETH to the validator using `SWNFTUpgrade::stake()`. This mints a new swNFT but no swETH
3. Wait until stakers provide another 16 ETH via `SWNFTUpgrade::stake()`, at which point the Beacon Chain will activate the validator
4. Keep the validator running, if possible without being penalized
5. After some time, voluntarily exit the validator on the Beacon Chain
6. Receive the initial 16 ETH deposit (with any penalties deducted) plus half of the earned rewards by calling `SWNFTUpgrade::unstake()`. *Note: This functionality is not implemented yet.*

The above describes a typical scenario for operating a *non-whitelisted* validator, which is permissionless. Unsurprisingly, there are also *whitelisted* validators which do not require the node operator to provide a 16 ETH deposit. This means that whitelisted validators do not have any buffer that can absorb penalties. For this reason. whitelisting a validator requires the manual approval of the Swell Network. This approval is only given if the node operator signs a paper contract that legally requires them to compensate any incurred validator penalties.

For stakers, using the Swell contract typically consists of the following steps:

1. Stake some X amount of ETH to an existing validator using `SWNFTUpgrade::stake()`. This mints a new swNFT and X swETH
2. Either withdraw the swETH using `SWNFTUpgrade::withdraw()` and use it in other DeFi protocols as a liquid substitute for the locked ETH, or put the swETH into a vault using `SWNFTUpgrade::enterStrategy()`
3. After some time, call `SWNFTUpgrade::unstake()` to receive a proportionate amount of the earned rewards, minus a fee. *Note: This functionality is not implemented yet.*

Altogether, the following operations were implemented at the time of the audit:

- Stake: Deposit ETH to a validator, minting a swNFT representing the staking position. The first deposit to a non-whitelisted validator is assumed to come from the node operator and must consist of at least 16 ETH that is used as collateral against penalties. Note that no swETH is minted for node operators. Subsequent deposits by stakers can be as low 1 ETH and mint an equivalent amount of swETH.
- Withdraw: Transfer the swETH associated with a swNFT to the staker. Can only be done by the owner of the swNFT.
- Deposit: Transfer swETH from a staker to a swNFT. Can only be done by the owner of the swNFT.
- Enter strategy: Transfer (some of) the swETH associated with a swNFT to a strategy. Can only be done by the owner of the swNFT.
- Exit strategy: Transfer swETH from a strategy back to a swNFT. Can only be done by the owner of the swNFT.

Note that while transferring swETH to/from a strategy is already fully implemented, the strategy contract itself is not, and hence was not part of the audit.

Additionally, the following operations are planned but not yet implemented, and hence have *not* been audited. Here we describe their intended behavior to the best of our understanding.

- Unstake: Unstaking a swNFT consist of the following steps:
    1. Calculating the amount of rewards that have been earned since the last stake or unstake operation, and minting a corresponding amount of swETH to the swNFT. Rewards are calculated by subtracting the current validator balance from the balance at the time of the last stake or unstake operation.
    2. Transfer all swETH associated with the swNFT to the owner of the swNFT.

3. If the validator has exited the Beacon Chain, burn the swNFT. Otherwise, transfer ownership of the swNFT to the Swell contract. This makes the swNFT available for purchase using the swap operation described below.

- Swap: swNFTs that are unstaked while the corresponding validator is still running are transferred from the original owner to the Swell contract. These swNFTs can then be bought by anyone who provides an amount of swETH equivalent to the staking amount of the swNFT.
- Redeem: Allows redeeming swETH for ETH at a guaranteed 1:1 exchange rate. The user provides some swETH to the Swell contract, which then burns it and transfers an equivalent amount of ETH back to the user. Note that this service can only be provided if the Swell contract holds enough ETH to fulfill the requested redemption. This may not always be the case, because the ETH held by the Swell contract comes from withdrawing the final validator stakes from the Beacon Chain, which only becomes possible after The Merge. And even after The Merge it may be possible that not many node operators unstake, in which case the Swell contract may only have a small supply of ETH available for redemption.

  In any case, being able to redeem swETH for ETH at a guaranteed 1:1 exchange rate is an important instrument to help keep the exchange rate in external markets close to 1:1 as well.

We have created two abstract models (see and Appendix 1 and Appendix 2) that describe the above functionality on a high-level, leaving out any low-level details that stand in the way of understanding the core business logic. These models also include the operations that are not actually implemented yet in the real contract.

Invariants

The invariants below describe desired properties that should hold at all times. We usually formulate invariants against the abstract model we create, and in a separate step try to show that our abstract model matches the actual implementation. By using abstract models that capture the whole contract (and not only the parts that are currently implemented) we can formulate much more interesting invariants than if we would only concentrate on the actually implemented parts. This also allowed us to evaluate whether the parts that are implemented are actually compatible with the planned additions. For example, Finding A05 describes an issue that would have made it impossible (or at least very difficult) to correctly implement unstaking later on.

When formulating these invariants, we made the following assumptions:

- The assumptions described in section Assumptions hold.
- Findings A1, A2 and A4 have been addressed.
- The behavior of our model for as-of-yet unimplemented functionality matches the intended behavior of the final implementation.
- All mathematical operations are carried out using real (as opposed to integral) numbers, meaning overflow and rounding errors are not taken into account.

Invariant 1 All swETH is backed by ETH that is either staked on the Beacon Chain or held by the Swell contract itself.

The above invariant touches many aspects of the Swell contract, the Beacon Chain, and the interactions between the two. It may be interesting to show some of the simpler invariants that are

implied by Invariant 1:

Invariant 2 The staking positions of node operators for non-whitelisted validators are at least 16 ETH.

Invariant 3 The total amount of swETH that has been minted for a validator that has not exited the Beacon Chain yet is less than or equal to the validator balance.

Invariant 4 If a staker stakes to a validator, then the withdrawal credentials of that validator allow the Swell contract to withdraw the validator stake once the validator has exited.

Invariant 5 If everyone unstakes, and all staked ETH is withdrawn from the validators, then the total supply of swETH equals the amount of ETH held by the Swell contract.

Some other invariants:

Invariant 6 The sum of all swETH associated swNFTs is equal to the amount of swETH held by the Swell contract.

Invariant 7 Non-whitelisted validators have exactly one staking position by a node operator.

Invariant 8 Whitelisted validators do not have any staking position by a node operator.

Invariant 9 swNFTs by node operators cannot be owned by the Swell contract. (This is because node operators should only be allowed to unstake once the validator has exited, and at that time unstaking also burns the swNFT.)

We did not have time to verify all of these invariants. See Checked properties for a list of properties we verified.

Findings

A01: Attacker can provide invalid withdrawal credentials

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

An attacker can create a validator with his own withdrawal credentials and then stake to the validator via `SWNFTUpgrade::stake()`. This allows the attacker to both exchange the minted swETH for ETH *and* to withdraw the staked ETH from the Beacon Chain. This leads to a situation where not all swETH is backed by ETH.

Note that this affects only non-whitelisted validators. Node operators of whitelisted validators are legally bound to cover any losses caused by their validators, and we assume that they will actually do so. Together with the fact that the first deposit to the Deposit Contract requires a valid signature that can only be created by the node operator (who has the validator's signing key), this means that if the withdrawal credentials of a whitelisted validator are incorrect, then the node operator must have done so himself, and he is responsible for covering any losses created by this.

Scenario

For example, consider an attacker who performs the following steps:

1. The attacker deposits a small amount of ETH to a fresh validator via the Deposit Contract. Assume this is the first deposit that has ever been made to that validator. Further assume that the attacker specifies his own address as withdrawal credentials.
2. The attacker then calls `swNFTUpgrade::stake()` for this validator, providing the 16 ETH that are required for node operators. Note that the withdrawal credentials specified in the call to the Deposit Contract will be ignored, since this is already the second deposit.
3. He then calls `swNFTUpgrade::stake()` again, and again deposits 16 ETH. This time, 16 swETH will be minted.
4. Next, the attacker withdraws the newly minted 16 swETH and exchanges them for 16 ETH
5. After The Merge, the attacker exits his validator and withdraws the staked 32 ETH (plus potential rewards).
6. In the end, the attacker walks away with 48 ETH, which are 16 ETH more than he started with. Further, 16 swETH are not backed by staked ETH anymore.

Recommendation

The code should be changed such that stakers can only stake to validators with correct withdrawal credentials, i.e., the withdrawal credentials must allow the Swell contract to withdraw the validator's stake once it is unlocked.

Status

Acknowledged. The solution proposed by the client is to distinguish between verified and unverified validators: Verified validators are those whose withdrawal credentials have been checked to be correct, and unverified validators are those for which this is not known yet. For unverified validators, only the initial deposit by the node operator is allowed. All further deposits are rejected, since they

would mint swETH. Once the node operator made the initial deposit, a trusted third party checks the Beacon Chain state to ensure that the validator has the correct withdrawal credentials. If this is the case, the validator can be considered verified, and stakers can be allowed to stake to it.

A02: Minted swETH may not be backed by ETH (non-whitelisted validators)

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

The function `SWNFTUpgrade::_stake()` assumes that if a deposit to the Eth2 Deposit Contract is successful (i.e., does not revert), then the deposit will also be accepted by the Beacon Chain. However, this is not the case if (1) this is the first deposit made to a specific validator and (2) the provided signature is invalid. Thus, it is possible that a staking position created by `_stake()` does not correspond to any stake on the Beacon Chain. This is problematic because in this case, the swETH minted for the staking position is not backed by any staked ETH on the Beacon Chain, endangering the 1:1 redeemability of swETH for ETH.

This finding only considers non-whitelisted validators. See A3 for how whitelisetd validators are affected.

Scenario

Assume that `pubKey` denotes the public key of a fresh validator to which no deposit has been made yet.

1. The node operator wants to deposit 16 ETH to `pubKey` by calling `_stake()`. However, assume he passes an invalid signature to `_stake()`. Then, when `_stake()` forwards the 16 ETH to the Deposit Contract along with the invalid signature, the deposit succeeds because the Deposit Contract does not verify the signature. Since the call to the Deposit Contract seems successful, execution of `_stake()` terminates successfully. However, the 16 ETH deposit will *not* be accepted by the Beacon Chain.
2. Next, assume a regular staker comes along and wants to stake 16 ETH to `pubKey` by calling `_stake()`. The staker provides an invalid signature, since by looking at the state of the Swell contract it seems that the node operator has already made the initial deposit, which means signatures will not actually be checked anymore by the Beacon Chain.
3. As a result, `_stake()` will mint 16 swETH. But since there has not actually been any successful deposit to `pubKey`, the Beacon Chain will see the invalid signature and also reject this deposit. Thus, the 16 new minted swETH are not backed by staked ETH on the Beacon Chain.

Note that in this scenario, the node operator loses 16 ETH, so this is unlikely to happen on purpose.

Recommendation

The fundamental problem is the same as the one described in A1, namely that swETH may be minted before it is ensured that the deposited ETH can later be withdrawn by the Swell contract. The code should be changed such that this cannot happen.

Status

Acknowledged. The client intents to implement the solution described in A1, which also addresses this finding.

A03: Invalid signature may lead to swETH not being backed by ETH (whitelisted validators)

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

The fundamental issue here is the same as in A2. However, in this finding we consider whitelisted validators. For whitelisted validators, performing the attack is easier, as attackers do not lose any tokens.

Scenario

1. The owner of the Swell contract whitelists a validator to which no deposits have been made yet.
2. The attacker stakes 32 ETH to that validator, but provides an invalid signature. The Deposit Contract accepts this 32 ETH deposit anyway, and the Swell contract mints 32 swETH.
3. When the 32 ETH deposit by the attacker is processed by the Beacon Chain, the invalid signature is detected and the deposit discarded.
4. Now 32 swETH have been minted that are not backed by any ETH on the Beacon Chain.

Note that the attacker can immediately exchange his 32 swETH for ETH on the open market. Thus, the attacker does not have any loss. The only thing that slows down the attack is that there must be whitelisted validators that no one has staked to yet.

Recommendation

For whitelisted validators, the solution proposed in A1 does not prevent the attack. The reason is that node operators of whitelisted validators do not need to make any deposit. Thus, the first deposit to a whitelisted validator will be made by a staker, which immediately mints swETH. This means a different solution has to be provided to whitelisted validators.

Status

Acknowledged. The client intents to require node operators for whitelisted validators to deposit 1 ETH to their validators. This way, the same solution as for A2 can be used.

A04: The first staking position should not mint any swETH

[Severity: High | Difficulty: N/A | Category: Security]

In order to protect against penalties, node operators of non-whitelisted validators are required to stake at least 16 ETH for which no swETH will be minted. Thus, when a validator is fully staked and gets activated, 16 swETH are backed by 32 ETH, allowing the validator to be penalized by 16 ETH without endangering swETH-to-ETH redeemability.

The first call to `_stake()` for a specific validator is assumed to be made by the node operator, meaning the staking amount should be at least 16 ETH, and no swETH should be minted. However, the current version of `_stake()` *does* mint swETH in this case.

Scenario

- A node operator calls `stake()` for the first time and deposits the required 16 ETH. Currently, this also mints 16 swETH.
- A staker stakes 16 ETH to the same validator, which mints another 16 swETH. At this points, 32 swETH have been minted, which are backed by 32 ETH on the Beacon Chain.
- The validator gets penalized such that its balance drops to 31 ETH.

In the last step, we are in a situation where not all swETH is backed by ETH, even though our Assumptions allow non-whitelisted validators to be penalized by up 16 ETH.

Recommendation

To fix the code, swETH should only be minted if the variable `operator` is true. Further, if `operator` is true, then the `baseTokenBalance` of the staking position should be set to zero.

Status

Fixed in commit `17cc8bc90e287c8bf5b8a0e57f8b4da811e62d69`.

A05: Requirement that node operators unstake last is not enforceable

[Severity: High | Difficulty: N/A | Category: Functional correctness]

Node operators must provide an initial deposit of 16 ETH that is used as a buffer for penalties. The intended behavior of the to-be-implemented unstaking function is that the node operator receives whatever is left of the 16 ETH after paying out the stakers, plus his share of the rewards. However, to know how much of the node operator's 16 ETH is actually left, we need to know the amount of rewards that are supposed to go to the stakers. This cannot easily be computed, as each staker may have started at a different time. Thus, the simplest way is to wait until all stakers have unstaked and claimed their rewards, and only then allow the node operator to also unstake and claim whatever is left.

However, currently it does not seem possible to enforce this check, since the Swell contract does not keep track of the number of stakers that have not yet unstaked.

Recommendation

Change the code to make it possible to check how many stakers have not yet unstaked.

Status

Acknowledged. The solution proposed by the client is to introduce a counter per validator that increases upon staking and decreases upon unstaking. Then, a node operator is only allowed to unstake if this counter is one.

A06: Logic contract is left uninitialized

[Severity: Unknown | Difficulty: N/A | Category: Security]

`SWNFTUpgrade` is an upgradeable contract. As such, `SWNFTUpgrade` functions as a logic contract for a proxy like `ERC1967Proxy`. As required for logic contracts, `SWNFTUpgrade` does not use a constructor for initialization but an ordinary function. To prevent double initialization, `SWNFTUpgrade` inherits from OpenZeppelin's `Initializable` contract.

Currently, deploying `SWNFTUpgrade` does not automatically initialize it. This may allow an attacker to call the initialize function himself and take over the logic contract. This is usually harmless, as the logic contract is only used via a proxy and never directly, but in some cases the attacker may be able to influence the behavior of the proxy. Thus, it is generally recommended to never leave the logic contract uninitialized. Even though we did not find a concrete attack scenario, we still recommend following this recommendation as a defensive mechanism.

Recommendation

We recommend to add the following constructor to `SWNFTUpgrade`:

```
constructor() initializer {}
```

Status

Acknowledged. The client intents to follow the recommendation.

A07: _stake() does not follow checks-effects-interactions pattern

[Severity: Unknown | Difficulty: N/A | Category: Security]

The last two lines of the function `SWNFTUpgrade::_stake()` look like this:

```
_safeMint(msg.sender, newItemId);
ISWETH(swETHAddress).mint(amount);
```

If `msg.sender` denotes the address of a contract, then `_safeMint()` may execute arbitrary user-controlled code. In this case, this would allow an attacker to observe the Swell contract in an inconsistent state. This does not seem to be a security issue, nevertheless it is recommended to be on the safe side and follow the checks-effects-interactions pattern.

Recommendation

Swap the last two lines of `_stake()` such that user-controlled code is executed last:

```
ISWETH(swETHAddress).mint(amount);
_safeMint(msg.sender, newItemId);
```

Status

Acknowledged. The client intents to follow the recommendation.

A08: enterStrategy() may enter the wrong strategy

[Severity: Low | Difficulty: N/A | Category: Usability]

To specify which strategy to enter when calling `SWNFTUgrade::enterStrategy()`, the user provides an index into the array `SWNFTUgrade::strategies`. In combination with `SWNFTUgrade::removeStrategy()`, which reorders the strategies array, this may lead to a situation where the user enters a strategy he did not intend to enter. Note that users can easily recover from entering the wrong strategy by simply leaving and then entering the correct strategy. However, this requires the user to notice the mistake in the first place.

Scenario

Consider the case where strategies contains the strategies S_0, S_1, and S_2 (in this order). Now imagine the following scenario:

1. A user wants to enter strategy S_0. He looks up the index of S_0, which is 0
2. The owner of the Swell contract removes S_0 from `strategies`. Because of the way `removeStrategy()` works, this means that `strategies` now contains the elements S_2 and S_1 (in this order)
3. The user now calls `enterStrategy()` with strategy index 0. This index now refers to S_2, hence this is the strategy that is entered. However, the strategy the user actually wanted to enter was S_0

Recommendation

We recommand to directly pass the address of the desired strategy to `enterStrategy()`. This may require to add another mapping to the contract like `mapping(address => bool) strategyExists` to check whether the strategy passed to `enterStrategy()` actually exists.

Status

Acknowledged. The client intents to change the type of `strategies` from `address[]` to `EnumerableSet.AddressSet`, which is provided by OpenZeppelin. This type provides functions to add and remove addresses and to check whether an address has been added.

Informative findings

B01: Depositing and withdrawing swETH assumes sender is not contract itself

The function `SWNFTUgrade::deposit()` allows the owner of a swNFT to deposit swETH to the swNFT. Roughly, the code looks as follows:

```
positions[tokenId].baseTokenBalance += amount;
success = ISWETH(swETHAddress).transferFrom(msg.sender, address(this), amount);
```

Here, `tokenId` represents the swNFT to which `msg.sender` wants to deposit swETH, and `baseTokenBalance` stores the current amount of swETH associated with the swNFT. If `msg.sender` wants to deposit `amount` swETH, we simply increase the `baseTokenBalance` and then transfer a corresponding amount of swETH from `msg.sender` to the contract.

A problem occurs if `msg.sender` is the contract itself, i.e., if `msg.sender == address(this)`. In this case `baseTokenBalance` is increased, but the amount of swETH held by the contract remains the same. This may lead to a situation where it is not possible to withdraw swETH from a swNFT even when `baseTokenBalance` is large enough, because there may not be any swETH left. A similar problem can occur with `withdraw()`.

Both `deposit()` and `withdraw()` check that `msg.sender` is the owner of the swNFT. Thus, the problem described here can only occur for swNFTs owned by the Swell contract. Currently, there is no reason for the Swell contract to own any swNFT, but when unstaking is implemented in a feature version of the code, then swNFTs may be transferred from stakers to the Swell contract upon unstaking.

Recommendation

Note that it seems that the problem cannot currently occur, as the Swell contract does not call either `deposit()` or `withdraw()`. However, as a defensive mechanism and for documentation purposes, it is recommended to add the following require clause to both functions:

```
require(msg.sender != address(this), "Contract cannot deposit/withdraw");
```

Alternatively, adding a comment above these functions may also suffice.

Status

Acknowledged.

B02: Unnecessary contract variables

The contract variables `SWNFTUpgrade::GWEI` is not currently used and can just be removed.

The contract variable `SWNFTUpgrade::ETHER` is initialized to 1e18, which is the equivalent to the expression `1 ether`. Thus, any use of `SWNFTUpgrade::ETHER` can simply be replaced by `1 ether`, which is also more gas efficient because it does not need to read from storage. Alternatively, `SWNFTUpgrade::ETHER` can be declared to be `constant`.

Recommendation

Remove `SWNFTUpgrade::GWEI` and `SWNFTUpgrade::ETHER`.

Status

Acknowledged.

B03: Contract variable swETHSymbol can be declared constant

The contract variable `SWNFTUpgrade::swETHSymbol` is set to the string literal `"swETH"` in the initialization function and never modified again. Thus, it is recommended to declare `swETHSymbol` as `constant`, which enforces the intention that the variable is never modified and also saves some gas.

Recommendation

Declare `swETHSymbol` as `constant`.

Status

Acknowledged.

B04: tvl() has very high gas requirements

The function `SWNFTUpgrade::tvl()` iterates over all validators to compute the total amount that has bee deposited via the contract. As the number of validators grows, this requires more and more gas. For example, for 3000 validators, `tvl()` needs approximately 4,000,000 gas. This is still below the block gas limit of currently 30,000,000, but calling `tvl()` from another contract is likely to be prohibitively expensive. Also, the block gas limit is variable and may change in the future.

Recommendation

We recommend to store the total locked value in a storage variable like `uint256 _tvl` that is updated appropriately. Then, `tvl()` only needs to return the value of this variable, which has a constant gas cost.

Status

Acknowledged. The client decided to leave the implementation as is for now and is considering an off-chain solution to compute the tvl.

B05: Redundant require clauses

The functions `deposit()`, `withdraw()`, `enterStrategy()` and `exitStrategy()` of contract `SWNFTUpgrade` contain the following requirement:

```
require(_exists(tokenId), "Query for nonexistent token");
```

However, this requirement is redundant, because all these functions also require the following:

```
require(ownerOf(tokenId) == msg.sender, "Only owner can ...");
```

This second requirement can never be fulfilled for non-existent swNFTs, because `ownerOf()` itself already checks that the swNFT exists.

Recommendation

Remove `require(_exists(tokenId), "Query for nonexistent token")` from the aforementioned functions, which saves a little gas.

Status

Fixed in commit `4061d874a3352d1cccab7a56b8e8a1ef665d7ce6`.

B06: Redundant require clauses (round 2)

The functions `enterStrategy()` and `exitStrategy()` of contract `SWNFTUpgrade` contain the following requirement:

```
require(strategies[strategy] != address(0), "strategy does not exist");
```

This check is unnecessary because strategies only contains non-zero addresses (this is enforced by the `addStrategy()` function).

Recommendation

Remove the check.

Status

Fixed in commit `cdf02ca2e72e76b1931f0f47cb703a21bd24a64b`.

B07: enterStrategy() should check if swNFT contains enough swETH

The function `enterStrategy(tokenId, strategy, amount)` of contract `SWNFTUpgrade` transfers swETH from the staking position denoted by `tokenId` to the strategy denoted by `strategy`. However, a user can transfer at most the amount of swETH that is contained in the position denoted by `tokenId`:

```
positions[tokenId].baseTokenBalance -= amount;
```

Here, `baseTokenBalance` is of type `uint`, which means that if `positions[tokenId].baseTokenBalance < amount`, then the function will revert. There is nothing wrong from a security perspective, but for the user it is not immediately clear what caused the error.

Recommendation

To give a more helpful error message, it might make sense to add the following requirement:

```
require(positions[tokenId].baseTokenBalance >= amount, "Not enough swETH available");
```

Status

Acknowledged. The client decided to leave the code as is.

B08: Provide better error message if not enough ETH is send when staking

The internal function `SWNFTUpgrade::_stake()` contains the following check:

```
require(amount <= msg.value, "cannot stake more than sent");
```

However, this check does not catch all cases in which an insufficient amount of ETH is sent, because `_stake()` is called in a loop:

```
uint totalAmount = msg.value;
for(uint i = 0; i < stakes.length; i++){
    ids[i] = _stake(stakes[i].pubKey, stakes[i].signature, stakes[i].depositDataRoot,
                    stakes[i].amount);
    totalAmount -= stakes[i].amount;
}
```

The aforementioned `require` clause only catches the case where a single staking amount is larger than `msg.value`, but does not catch there case where the sum of all staking amounts is larger than `msg.value` while each individual staking amount is smaller. In this case, the expression `totalAmount -= stakes[i].amount;` will underflow and the transaction reverts. There is nothing wrong from a security perspective, but for the user it is not immediately clear what caused the error.

Recommendation

To give a better error message in these situations, consider adding the following check at the beginning of the loop:

```
require(totalAmount >= stakes[i].amount, "cannot stake more than sent");
```

Further, the following check can be removed from `SWNFTUpgrade::_stake()`:

```
require(amount <= msg.value, "cannot stake more than sent");
```

Status

Acknowledged.

B09: Unnecessary requires clause in staking function

Function `SWNFTUpgrade::stake()` contains the following checks:

```
require(msg.value >= 1 ether, "Must send at least 1 ETH");
require(msg.value % ETHER == 0, "stake value not multiple of Ether");
```

These checks are insufficient, because the actual staking amounts are passed via the `stakes` argument, so checking `msg.value` does not actually detect invalid staking amount. These checks are also unnecessary, as they are correctly performed in function `SWNFTUpgrade::_stake()`.

Recommendation

Remove the checks.

Status

Acknowledged.

B10: Contract variable storing fee is internal

The contract variable `SWNFTUpgrade::fee` has visibility `internal`, making it not easily accessible from outside the contract. Since this variable stores information that is useful to users, it may make sense to make it `public`.

Recommendation

Declare `SWNFTUpgrade::fee` as `public`.

Status

Acknowledged.

B11: Contract variable storing fee is never modified

The contract variable `SWNFTUpgrade::fee` is set in the initialization function to the fixed value `1e17` and then never changed.

Recommendation

If the fee is supposed to be `constant`, then it is recommended to declare it as such. Otherwise, a setter function should be added.

Status

The client confirmed that the fee should be able to change, and that an appropriate setter function will be added.

B12: Unnecessary import statement

The file `contracts/swNFTUpgrade.sol` contains the following import statement:

```
import "base64-sol/base64.sol";
```

However, this import is never used.

Recommendation

Remove the import.

Status

Acknowledged.

Checked properties

There is more to an audit than finding bugs. We also want to report what errors we have ruled out, and what properties we have verified.

Automatically checked invariants

We have used the Solidity SMTChecker to verify the following two invariants in a simplified version of the Swell contract:

Invariant The sum of all swETH associated swNFTs is equal to the amount of swETH held by the Swell contract.

Invariant The staking positions of node operators for non-whitelisted validators are at least 16 ETH.

See Appendix 3 for the Solidity code.

All swETH is backed by staked ETH

In this section we provide a detailed proof sketch to show that the mathematical model described in Appendix 1 maintains the invariant that all swETH is backed by ETH. For this invariant to hold, we have to make the following assumption:

Assumption 1. Let $s_0 \in \mathsf{InitState}$ and $s \in \mathsf{reachable}(s_0)$ be arbitrary. We assume that for all $pk \in \mathsf{NonWithdrawnPubKeys}_s$ the following condition holds:

$$v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_s(pk) \leq v.\mathsf{balance},$$

where $v := s.\mathsf{B.validators}(pk)$.

Intuitively, this assumption states that the total penalty that a validator incurs must not exceed the provided collateral. If pk refers to a whitelisted validator, then $\mathsf{collateral}_s(pk) = 16$ ETH. The ideal balance of a validator is computed the same way as its actual balance, except that it is not affected by penalties. Thus, for a validator v, the difference $v.\mathsf{ideal\text{-}balance} - v.\mathsf{balance}$ tells you the total amount of penalties that v has incurred. To make it more clear, one can rearrange the above assumption as follows:

$$v.\mathsf{ideal\text{-}balance} - v.\mathsf{balance} \leq \mathsf{collateral}_s(pk).$$

This matches the stated intuition that the total penalty that a validator incurs must not exceed the provided collateral.

The remainder of this section is structured as follows: We first introduce some more Definitions, then prove utility lemmas for Non-withdrawn validators and Withdrawn validators, and finally prove the Main invariant.

Definitions

When formulating our invariants, we often need to know the amount of rewards a staker *would* get if they claimed their rewards now. This is computed by the following function for some staking position $p \in \mathsf{Position}$ in state $s \in \mathsf{State}$:

$$\mathsf{unminted}_s(p) := \mathsf{max}(0, \mathsf{reward\text{-}balance}_s(p.\mathsf{pubkey}) - p.\mathsf{validator\text{-}balance}) \cdot \frac{p.\mathsf{staking\text{-}amount}}{32}$$

This definitions simply mirrors the calculation made by the transition relation for the $\mathsf{claim}(\cdot)$ event.

We extend this definition to validators identified by their public key $pk \in \mathsf{BLSPublicKey}$:

$$\begin{aligned}\mathsf{unminted}_s(pk) := &\sum p \in P\colon \mathsf{unminted}_s(p) \\ &+ \mathsf{max}(0, \mathsf{reward\text{-}balance}_s(pk) - 32) \cdot \frac{\mathit{remaining\text{-}stake}}{32}\end{aligned}$$

where $\mathit{remaining\text{-}stake} := 32 - \mathsf{validator\text{-}deposits}_s(pk)$ and $P := \{p \in \mathsf{staker\text{-}positions}_s(pk) \mid \mathsf{is\text{-}staked}_s(p) \wedge p.\mathsf{nft} \notin s.\mathsf{C.backed\text{-}by\text{-}contract}\}$.

Intuitively, $\mathsf{unminted}_s(pk)$ computes the amount of ETH that needs to be provided by the validator identified by pk in order to back all the swETH that would be minted if all current and future stakers claimed their rewards.

The first part of the sum accounts for all the staking positions $p \in \mathsf{staker\text{-}positions}_s(pk)$ such that (1) $\mathsf{is\text{-}staked}_s(p)$ and (2) $p.\mathsf{nft} \notin s.\mathsf{C.backed\text{-}by\text{-}contract}$. The first condition ensures that we are only

considering staking positions that can actually claim any rewards (if $\neg\mathsf{is\text{-}staked}_s(p)$, then $p.\mathsf{nft}$ has already been burned and no rewards can actually be claimed anymore for p). The second condition filters out any p whose minted swETH is backed by ETH that is held by the Swell contract instead of by the validator.

The second part of the sum, $\mathsf{max}(0, \mathsf{reward\text{-}balance}_s(pk) - 32) \cdot \frac{\mathit{remaining\text{-}stake}}{32}$, computes the amount of rewards of staking positions that do not exist yet but may in the future.

The final definition we need to introduce is the following:

$$\mathsf{unassigned\text{-}rewards}_s(pk) := \frac{\mathsf{max}(0, v.\mathsf{final\text{-}balance} - 32)}{32} \cdot \mathit{remaining\text{-}stake},$$

where $v := s.\mathsf{B.validators}(pk)$ and

$$\mathit{remaining\text{-}stake} := 32 - \sum p \in \mathsf{positions}_s(pk), p.\mathsf{nft} \in s.\mathsf{C.backed\text{-}by\text{-}contract}\colon p.\mathsf{staking\text{-}amount}.$$

Unassigned rewards are needed for the corner case where a staker creates a staking position p for a validator after the validator balance has already been withdrawn to the Swell contract. In this case, the swETH rewards that are minted for p are backed by ETH that is held by the contract and not by the validator.

Non-withdrawn validators

The following predicate is true if the swETH minted for some validator identified by pk is fully backed by staked ETH in state s.

$$\mathsf{is\text{-}swETH\text{-}backed}_s(pk) := \mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_s(pk)$$

where $v = s.\mathsf{B.validators}(pk)$.

Invariant 1. $s \in \mathsf{INV\text{-}1}$ iff the following condition holds.

$$\forall pk \in \mathsf{NonWithdrawnPubKeys}_s\colon \mathsf{is\text{-}swETH\text{-}backed}_s(pk)$$

Lemma 1. If $s_0 \in \mathsf{InitState}$, then $\mathsf{reachable}(s_0) \subseteq \mathsf{INV\text{-}1}$.

Proof sketch. Let $s_0 \in \mathsf{InitState}$ and $s' \in \mathsf{State}$ be arbitrary such that $s' \in \mathsf{reachable}(s_0)$. We need to show that for arbitrary $pk^* \in \mathsf{NonWithdrawnPubKeys}_{s'}$ the condition $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$ holds. The proof is by induction on s'.

Base case Then $s' = s_0$. Since $s_0 \in \mathsf{InitState}$ we also get $s' \in \mathsf{InitState}$.

$s' \in \mathsf{InitState}$ implies $\mathsf{dom}(s'.\mathsf{B.validators}) = \emptyset$, hence $\mathsf{NonWithdrawnPubKeys}_{s'} = \emptyset$. Thus, the claim holds trivially.

Inductive case Then there exist s, ev such that $s \in \mathsf{reachable}(s_0)$ and $s \xrightarrow{ev} s'$.

The induction hypothesis implies $s \in \mathsf{INV\text{-}1}$. We make a case distinction on ev.

Case: $ev = \mathsf{reward}(pk, x)$ for some pk, x

Then by the definition of the transition relation we get

$s' = s[\mathsf{B.validators}(pk)[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]]$

We make a case distinction on whether $pk = pk^*$.

Case: $pk = pk^*$

This implies

(a) $\mathsf{rewards\text{-}minted}_s(pk) = \mathsf{rewards\text{-}minted}_{s'}(pk)$

(b) $\mathsf{stake\text{-}minted}_{s'}(pk) = \mathsf{stake\text{-}minted}_{s'}(pk)$

(d) $\mathsf{total\text{-}minted}_s(pk) = \mathsf{total\text{-}minted}_{s'}(pk)$

Follows from (a) and (b).

(e) $\mathsf{unminted}_{s'}(pk) = \mathsf{unminted}_s(pk) + y$ for some $0 \leq y \leq x$

(f) $\mathsf{collateral}_{s'}(pk) = \mathsf{collateral}_s(pk)$

Define the following variables.

$v := s.\mathsf{B.validators}(pk)$

$v' := s'.\mathsf{B.validators}(pk)$

Then

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \iff & \mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) + y \leq v.\mathsf{ideal\text{-}balance} + x - \mathsf{collateral}_s(pk) \\ \iff & \mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_s(pk) + x - y \end{aligned}$$

The last inequality follows from $\mathsf{is\text{-}swETH\text{-}backed}_s(pk)$ and the fact that $x - y \geq 0$. Thus, we get $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk)$, which directly implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $pk \neq pk^*$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{penalty}(pk, x)$

Case: $pk = pk^*$

First, define the following variables.

$v := s.\mathsf{B.validators}(pk)$

$v' := v[\mathsf{balance} \mathrel{-}= x]$

Then by the definition of the transition relation this implies

1. $s' = s[\mathsf{B.validators}(pk) := v']$.
2. $x \leq v.\mathsf{balance}$,
3. $\mathsf{collateral}_{s'}(pk) = \mathsf{collateral}_s(pk)$
4. $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_s(pk)$
5. $\mathsf{unminted}_{s'}(pk) \leq \mathsf{unminted}_s(pk)$

6. $v'.\mathsf{ideal\text{-}balance} = v.\mathsf{ideal\text{-}balance}$

Then

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \iff & \mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s'}(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s}(pk) \end{aligned}$$

The last inequality follows from $\mathsf{is\text{-}swETH\text{-}backed}_s(pk)$ and the fact that $\mathsf{unminted}_{s'}(pk) \leq \mathsf{unminted}_s(pk)$. Thus, we get $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk)$, which directly implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $pk \neq pk^*$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{deposit}(pk, x)$

Case: $pk = pk^*$

Case: $pk \in \mathsf{dom}(s.\mathsf{B.validators})$

Then the reasoning is the same as for the case where $ev = \mathsf{reward}(pk, x)$.

Case: $pk \notin \mathsf{dom}(s.\mathsf{B.validators})$

Then it can be shown that there is no staking position $p \in \mathsf{rng}(s.\mathsf{C.positions})$ such that $p.\mathsf{pubkey} = pk$. This implies

- $\mathsf{total\text{-}minted}_{s'}(pk) = 0$
- $\mathsf{unminted}_{s'}(pk) = \mathsf{max}(0, v.\mathsf{ideal\text{-}balance} - 32)$, where $v := s'.\mathsf{B.validators}(pk)$
- $\mathsf{collateral}_{s'}(pk) = 0$

Then

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \iff & \mathsf{max}(0, v.\mathsf{ideal\text{-}balance} - 32) \leq v.\mathsf{ideal\text{-}balance} \end{aligned}$$

The last inequality follows from the fact that $0 \leq v.\mathsf{ideal\text{-}balance}$.

Case: $pk \neq pk^*$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{withdraw}(pk)$

Case: $pk = pk^*$

Then $pk \notin \mathsf{NonWithdrawnPubKeys}_{s'}$. However, this contradicts the assumption that $pk^* \in \mathsf{NonWithdrawnPubKeys}_{s'}$. Hence, we are done.

Case: $pk \neq pk^*$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{exit}(pk)$

Nothing relevant changes.

Case: $ev = \mathsf{next\text{-}epoch}$

Case: $s.\mathsf{B.validators}(pk^*).\mathsf{withdrawable\text{-}epoch} = s.\mathsf{B.epoch}$

Let

$v := s.\mathsf{B.validators}(pk^*)$

$v' := s'.\mathsf{B.validators}(pk^*)$

Then the definition of the transition relation implies

$v' = v[\mathsf{final\text{-}balance} := v.\mathsf{balance}]$

This in turn implies

(a) $\mathsf{reward\text{-}balance}_{s'}(pk^*) = \mathsf{reward\text{-}balance}_{s}(pk^*)$

(b) $\mathsf{unminted}_{s'}(pk^*) = \mathsf{unminted}_{s}(pk^*)$ (Follows from (a))

(c) $\mathsf{total\text{-}minted}_{s'}(pk^*) = \mathsf{total\text{-}minted}_{s}(pk^*)$

(c) $\mathsf{collateral}_{s'}(pk^*) = \mathsf{collateral}_{s}(pk^*)$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the above facts implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $s.\mathsf{B.validators}(pk^*).\mathsf{withdrawable\text{-}epoch} \neq s.\mathsf{B.epoch}$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{stake}(addr, pk, x)$

Case: $pk = pk^*$

Let

$is\text{-}op := (\mathsf{validator\text{-}deposits}_s(pk) = 0 \wedge pk \notin s.\mathsf{C.whitelist})$

$$v := \begin{cases} s.\mathsf{B.validators}(pk) & \text{if } pk \in \mathsf{dom}(s.\mathsf{B.validators}) \\ \mathsf{fresh\text{-}validator}(pk) & \text{otherwise} \end{cases}$$

$v' := v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]$

Then the transition relation implies

$pk \in \mathsf{dom}(s.\mathsf{B.validators}) \implies s.\mathsf{B.validators}(pk) = v$

$s'.\mathsf{B.validators}(pk) = v'$

$x \leq 32$

We make a case distinction on whether the validator already existed or not.

Case: $pk \in \mathsf{dom}(s.\mathsf{B.validators})$

Case: *is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_{s}(pk)$
2. $\mathsf{unminted}_{s'}(pk) \leq \mathsf{unminted}_{s}(pk)$ (Follows from the fact that there are no existing staking positions for pk)
3. $v'.\mathsf{ideal\text{-}balance} = v.\mathsf{ideal\text{-}balance} + x$
4. $\mathsf{collateral}_{s'}(pk) = \mathsf{collateral}_{s}(pk) + x$

This implies

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \implies & \mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s}(pk) \end{aligned}$$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_{s}(pk^*)$. This together with the above implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $\neg$*is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_{s}(v) + x$
2. $\mathsf{unminted}_{s'}(pk) = \mathsf{unminted}_{s}(pk)$
3. $v'.\mathsf{ideal\text{-}balance} = v.\mathsf{ideal\text{-}balance} + x$
4. $\mathsf{collateral}_{s'}(pk) = \mathsf{collateral}_{s}(pk)$

This implies

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \iff & \mathsf{total\text{-}minted}_{s}(pk) + x + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{ideal\text{-}balance} + x - \mathsf{collateral}_{s}(pk) \\ \iff & \mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s}(pk) \end{aligned}$$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_{s}(pk^*)$. This together with the above implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $pk \notin \mathsf{dom}(s.\mathsf{B.validators})$

Case: *is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = 0$
2. $\mathsf{unminted}_{s'}(pk) = 0$ (Follows from the fact that $v'.\mathsf{balance} \leq 32$)
3. $v'.\mathsf{ideal\text{-}balance} = x$
4. $\mathsf{collateral}_{s'}(pk) = x$

This gives us

$$\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \iff 0 \leq 0$$

This directly implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $\neg$*is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = x$
2. $\mathsf{unminted}_{s'}(pk) = 0$ (Follows from the fact that $v'.\mathsf{balance} \leq 32$)
3. $v'.\mathsf{ideal\text{-}balance} = x$
4. $\mathsf{collateral}_{s'}(pk) = 0$

This gives us

$$\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \iff x \leq x$$

This directly implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $pk \neq pk^*$

Since $s \in$ INV-1 we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{claim}(nft)$

Let $p := s.\mathsf{C.positions}(nft)$.

Case: $p.\mathsf{pubkey} = pk^*$

Let

a) $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$

b) $v' := s'.\mathsf{B.validators}(p.\mathsf{pubkey})$

c) $r := \mathsf{max}(0, v.\mathsf{balance} - p.\mathsf{validator\text{-}balance}) * \frac{p.\mathsf{staking\text{-}amount}}{32}$

d) $p' := s'.\mathsf{C.positions}(nft)$

Then by the definition of the transition relation this implies

e) $v' = v$

f) $p' = p[\mathsf{validator\text{-}balance} := v.\mathsf{balance}]$
$[\mathsf{rewards\text{-}minted} \mathrel{+}= r]$
$[\mathsf{swETH\text{-}balance} \mathrel{+}= r]$

Together, this implies

g) $\mathsf{total\text{-}minted}_{s'}(p.\mathsf{pubkey}) = \mathsf{total\text{-}minted}_s(p.\mathsf{pubkey}) + r$

h) $\mathsf{unminted}_{s'}(p.\mathsf{pubkey}) = \mathsf{unminted}_s(p.\mathsf{pubkey}) - r$

i) $\mathsf{collateral}_{s'}(p.\mathsf{pubkey}) = \mathsf{collateral}_s(p.\mathsf{pubkey})$

Let $pk = p.\mathsf{pubkey}$. Then

$$\begin{aligned}
&\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\
\iff &\mathsf{total\text{-}minted}_s(pk) + r + \mathsf{unminted}_s(pk) - r \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_s(pk) \\
\iff &\mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) \leq v.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_s(pk)
\end{aligned}$$

Since $s \in$ INV-1 we get $\mathsf{is\text{-}swETH\text{-}backed}_s(p.\mathsf{pubkey})$. This together with the above implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $p.\mathsf{pubkey} \neq pk^*$

Since $s \in$ INV-1 we get $\mathsf{is\text{-}swETH\text{-}backed}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{unstake}(addr, nft)$

Then by the transition relation there exists a s'' such that $s \xrightarrow{\mathsf{claim}(nft)} s''$. This implies $s'' \in$ INV-1 (see case for $\mathsf{claim}(nft)$).

s' is then obtained by applying some more modifications to s''. The only change relevant for INV-1 is that nft may be burned. Let $p := s''.\mathsf{positions}(nft)$. If nft is burned, then $\mathsf{unminted}_{s'}(p.\mathsf{pubkey}) \leq \mathsf{unminted}_{s''}(p.\mathsf{pubkey})$. This together with the fact that all other terms in $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(p.\mathsf{pubkey})$ remain unchanged compared to $\mathsf{is\text{-}swETH\text{-}backed}_{s''}(p.\mathsf{pubkey})$ implies that $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(p.\mathsf{pubkey})$ holds.

Case: $ev = \mathsf{unstake\text{-}op}(addr, nft)$

Let $p := s.\mathsf{C.positions}(nft)$.

Case: $p.\mathsf{pubkey} = pk^*$

First, we introduce the following variables:

a) $pk := p.\mathsf{pubkey}$

b) $v := s.\mathsf{B.validators}(pk)$

c) $v' := s'.\mathsf{B.validators}(pk)$

d) $x := v.\mathsf{final\text{-}balance} - \mathsf{total\text{-}minted}_s(v.\mathsf{pubkey})$

e) $p' := p[\mathsf{rewards\text{-}minted} \mathrel{+}= x]$
$[\mathsf{swETH\text{-}balance} := 0]$
$[\mathsf{validator\text{-}balance} := v.\mathsf{balance}]$

Then by the definition of the transition relation this implies

f) $p.\mathsf{by\text{-}operator} = \mathsf{true}$

g) $x > 0$

h) $p' = s'.\mathsf{C.positions}(nft)$

i) $v' = s'.\mathsf{B.validators}(p.\mathsf{pubkey})$

We further get

j) $v' = v$

k) $\mathsf{total\text{-}minted}_{s'}(v.\mathsf{pubkey}) = \mathsf{total\text{-}minted}_s(v.\mathsf{pubkey}) + x$

l) $\mathsf{unminted}_{s'}(v.\mathsf{pubkey}) = 0$

This follows from the fact that all staking positions have been unstaked.

m) $\mathsf{collateral}_{s'}(pk) = 0$

Then

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) && \leq v'.\mathsf{ideal\text{-}balance} - \mathsf{collateral}_{s'}(pk) \\ \iff & \mathsf{total\text{-}minted}_{s}(pk) + x && \leq v.\mathsf{ideal\text{-}balance} \\ \iff & v.\mathsf{final\text{-}balance} && \leq v.\mathsf{ideal\text{-}balance} \end{aligned}$$

It can be shown that the definition of the transition relation implies $v.\mathsf{final\text{-}balance} \leq v.\mathsf{ideal\text{-}balance}$, so we are done. Thus, the above implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $p.\mathsf{pubkey} \neq pk^*$

Since $s \in \mathsf{INV\text{-}1}$ we get $\mathsf{is\text{-}swETH\text{-}backed}_{s}(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{is\text{-}swETH\text{-}backed}_{s'}(pk^*)$.

Case: $ev = \mathsf{swap}(addr, nft)$

Then s' only differs from s in ways that are not relevant for the invariant we are considering here. Thus, $s \in \mathsf{INV\text{-}1}$ implies $s' \in \mathsf{INV\text{-}1}$.

Case: $ev = \mathsf{redeem}(addr, x)$

Then s' only differs from s in ways that are not relevant for the invariant we are considering here. Thus, $s \in \mathsf{INV\text{-}1}$ implies $s' \in \mathsf{INV\text{-}1}$.

□

Invariant 2. $s \in \mathsf{INV\text{-}2}$ iff for all $v \in \mathsf{NonWithdrawnPubKeys}_s$ following condition holds.
$\mathsf{total\text{-}minted}_s(v) + \mathsf{unminted}_s(v) \leq v.\mathsf{balance}$,
where $v := s.\mathsf{B.validators}(pk)$.

This invariant states that all swETH minted for non-withdrawn validators is backed by ETH on the Beacon Chain.

Lemma 2. If $s_0 \in \mathsf{InitState}$, then $\mathsf{reachable}(s_0) \subseteq \mathsf{INV\text{-}2}$.

Proof. Follows from Invariant 1 and Assumption 1. □

Withdrawn validators

$$\begin{aligned} \mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk) := \mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) \leq \\ v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_s(pk) \end{aligned}$$

where $v := s.\mathsf{B.validators}(pk)$.

Invariant 3. $s \in \mathsf{INV\text{-}3}$ iff the following condition holds.
$\forall v \in \mathsf{WithdrawnPubKeys}_s$: $\mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk)$

Lemma 3. If $s_0 \in \mathsf{InitState}$, then $\mathsf{reachable}(s_0) \subseteq \mathsf{INV\text{-}3}$.

Proof sketch. Let $s_0 \in \mathsf{InitState}$ and $s' \in \mathsf{State}$ be arbitrary such that $s' \in \mathsf{reachable}(s_0)$. We need to show that for arbitrary $pk^* \in \mathsf{WithdrawnPubKeys}_{s'}$ the condition $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$ holds. The proof is by induction on s'.

Base case Then $s' = s_0$. Since $s_0 \in \mathsf{InitState}$ we also get $s' \in \mathsf{InitState}$.

$s' \in \mathsf{InitState}$ implies $\mathsf{dom}(s'.\mathsf{B.validators}) = \emptyset$, hence $\mathsf{WithdrawnPubKeys}_{s'} = \emptyset$. Thus, the claim holds trivially.

Inductive case Then there exist s, ev such that $s \in \mathsf{reachable}(s_0)$ and $s \xrightarrow{ev} s'$.

The induction hypothesis implies $s \in \mathsf{INV\text{-}3}$. We make a case distinction on ev.

Case: $ev = \mathsf{reward}(pk, x)$

Not possible for withdrawn validators.

Case: $ev = \mathsf{penalty}(pk, x)$

Not possible for withdrawn validators.

Case: $ev = \mathsf{deposit}(pk, x)$

Case: $pk = pk^*$

Let

* $v := s.\mathsf{B.validators}(pk)$ (We know $v \in \mathsf{dom}(s.\mathsf{B.validators})$ because v has already been withdrawn.)
* $v' := s'.\mathsf{B.validators}(pk)$

Then the transition relation implies

* $v' = v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]$
* $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_{s}(pk)$
* $\mathsf{unminted}_{s'}(pk) = \mathsf{unminted}_{s}(pk)$
* $\mathsf{unassigned\text{-}rewards}_{s'}(pk) = \mathsf{unassigned\text{-}rewards}_{s}(pk)$

Then

$$\begin{aligned} &\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \\ \iff &\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + x + \mathsf{unassigned\text{-}rewards}_{s}(pk) \end{aligned}$$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $pk \neq pk^*$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $ev = \mathsf{withdraw}(pk)$

Case: $pk = pk^*$

Let $v' := s'.\mathsf{B.validators}(pk)$. Then the definition of the transition relation implies

a) $v'.\mathsf{balance} = 0$

b) $\mathsf{total\text{-}minted}_{s'}(pk) = 0$

Then

$$\begin{aligned}&\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk)\\ \iff &\mathsf{unminted}_{s}(pk) \leq \mathsf{unassigned\text{-}rewards}_{s}(pk)\end{aligned}$$

Thus, it remains to show that $\mathsf{unminted}_{s'}(pk) \leq \mathsf{unassigned\text{-}rewards}_{s'}(pk)$. It turns out they are equal, because both their *remaining-stake* are equal

Since $s \in$ INV-3 we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $pk \neq pk^*$

Since $s \in$ INV-3 we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $ev = \mathsf{exit}(pk)$

Nothing relevant changes.

Case: $ev = \mathsf{next\text{-}epoch}$

Nothing relevant changes.

Case: $ev = \mathsf{stake}(addr, pk, x)$

Case: $pk = pk^*$

Let

* $\mathit{is\text{-}op} := (\mathsf{validator\text{-}deposits}_s(pk) = 0 \wedge pk \notin s.\mathsf{C.whitelist})$
* $v := \begin{cases} s.\mathsf{B.validators}(pk) & \text{if } pk \in \mathsf{dom}(s.\mathsf{B.validators}) \\ \mathsf{fresh\text{-}validator}(pk) & \text{otherwise} \end{cases}$
* $v' := v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]$

Then the transition relation implies

* $pk \in \mathsf{dom}(s.\mathsf{B.validators}) \implies s.\mathsf{B.validators}(pk) = v$
* $s'.\mathsf{B.validators}(pk) = v'$
* $x \leq 32$

We make a case distinction on whether the validator already existed or not.

Case: $pk \in \mathsf{dom}(s.\mathsf{B.validators})$

Case: *is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_{s}(pk)$
2. $\mathsf{unminted}_{s'}(pk) \leq \mathsf{unminted}_{s}(pk)$ (Follows from the fact that there are no existing staking positions for pk)
3. $v'.\mathsf{balance} = v.\mathsf{balance} + x$

4. $\mathsf{unassigned\text{-}rewards}_{s'}(pk) = \mathsf{unassigned\text{-}rewards}_{s}(pk)$

This implies

$$\begin{aligned}
&\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \\
\implies &\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + x + \mathsf{unassigned\text{-}rewards}_{s}(pk) \\
\implies &\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s}(pk)
\end{aligned}$$

Since $s \in$ INV-3 we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk^*)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $\neg$*is-op*

Then

1. $\mathsf{total\text{-}minted}_{s'}(pk) = \mathsf{total\text{-}minted}_{s}(v) + x$
2. $\mathsf{unminted}_{s'}(pk) = \mathsf{unminted}_{s}(pk)$
3. $v'.\mathsf{balance} = v.\mathsf{balance} + x$
4. $\mathsf{unassigned\text{-}rewards}_{s'}(pk) = \mathsf{unassigned\text{-}rewards}_{s}(pk)$

This implies

$$\begin{aligned}
&\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \\
\iff &\mathsf{total\text{-}minted}_{s}(pk) + x + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + x + \mathsf{unassigned\text{-}rewards}_{s}(pk) \\
\iff &\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s}(pk)
\end{aligned}$$

Since $s \in$ INV-3 we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk^*)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $pk \notin \mathsf{dom}(s.\mathsf{B.validators})$

This case is not possible because $pk \in \mathsf{WithdrawnPubKeys}_{s'}$, and according to the transition relation, validators cannot be created and withdrawn in a single step.

Case: $pk \neq pk^*$

Since $s \in$ INV-3 we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $ev = \mathsf{claim}(nft)$

Let $p := s.\mathsf{C.positions}(nft)$.

Case: $p.\mathsf{pubkey} \neq pk^*$

Then let

a) $pk := p.\mathsf{pubkey}$.

b) $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$

c) $r := \mathsf{max}(0, v.\mathsf{final\text{-}balance} - p.\mathsf{validator\text{-}balance}) * \frac{p.\mathsf{staking\text{-}amount}}{32}$

d) $p' := p[\mathsf{validator\text{-}balance} := v.\mathsf{final\text{-}balance}][\mathsf{rewards\text{-}minted} \mathrel{+}= r]$

e) $v' := s'.\mathsf{B.validators}(p.\mathsf{pubkey})$

By the definition of the transition relation this implies

f) $v'.\mathsf{balance} = v.\mathsf{balance}$

g) $\mathsf{stake\text{-}minted}_{s'}(p.\mathsf{pubkey}) = \mathsf{stake\text{-}minted}_s(p.\mathsf{pubkey})$

h) $\mathsf{rewards\text{-}minted}_{s'}(p.\mathsf{pubkey}) = \mathsf{rewards\text{-}minted}_s(p.\mathsf{pubkey}) + r$

i) $\mathsf{unminted}_{s'}(pk) = \mathsf{unminted}_s(pk) - r$

j) $\mathsf{unassigned\text{-}rewards}_{s'}(pk) = \mathsf{unassigned\text{-}rewards}_s(pk)$

Then

$$\begin{aligned}
&\mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \\
\iff &\mathsf{total\text{-}minted}_s(pk) + r + \mathsf{unminted}_s(pk) - r \leq v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_s(pk) \\
\iff &\mathsf{total\text{-}minted}_s(pk) + \mathsf{unminted}_s(pk) \leq v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_s(pk)
\end{aligned}$$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $p.\mathsf{pubkey} \neq pk^*$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_s(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $ev = \mathsf{unstake}(addr, nft)$

Then by the transition relation there exists a s'' such that $s \xrightarrow{\mathsf{claim}(nft)} s''$. This implies $s'' \in \mathsf{INV\text{-}3}$ (see case for $\mathsf{claim}(nft)$).

s' is then obtained by applying some more modifications to s''. The only change relevant for $\mathsf{INV\text{-}3}$ is that nft may be burned. Let $p := s''.\mathsf{positions}(nft)$. If nft is burned, then $\mathsf{unminted}_{s'}(p.\mathsf{pubkey}) \leq \mathsf{unminted}_{s''}(p.\mathsf{pubkey})$. This together with the fact that all other terms in $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(p.\mathsf{pubkey})$ remain unchanged compared to $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s''}(p.\mathsf{pubkey})$ implies that $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(p.\mathsf{pubkey})$ holds.

Case: $ev = \mathsf{unstake\text{-}op}(addr, nft)$

Let $p := s'.\mathsf{positions}(nft)$.

Case: $p.\mathsf{pubkey} = pk^*$

Let

* $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$
* $v' := s'.\mathsf{B.validators}(p.\mathsf{pubkey})$

Then the definition of the transition relation implies

1. $v' = v$

2. $\mathsf{total\text{-}minted}_{s'}(p.\mathsf{pubkey}) \leq \mathsf{total\text{-}minted}_{s}(p.\mathsf{pubkey})$

   This is because $p.\mathsf{nft} \in s'.\mathsf{C.backed\text{-}by\text{-}contract}$, which means p will be ignored by $\mathsf{rewards\text{-}minted}_{s'}(p.\mathsf{pubkey})$.

3. $\mathsf{unminted}_{s'}(p.\mathsf{pubkey}) = \mathsf{unminted}_{s}(p.\mathsf{pubkey})$

4. $\mathsf{unassigned\text{-}rewards}_{s'}(p.\mathsf{pubkey}) = \mathsf{unassigned\text{-}rewards}_{s}(p.\mathsf{pubkey})$

Let $pk := p.\mathsf{pubkey}$. Then

$$\begin{aligned} & \mathsf{total\text{-}minted}_{s'}(pk) + \mathsf{unminted}_{s'}(pk) \leq v'.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \\ \implies & \mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance} + \mathsf{unassigned\text{-}rewards}_{s}(pk) \end{aligned}$$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk)$. This together with the above implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $p.\mathsf{pubkey} \neq pk^*$

Since $s \in \mathsf{INV\text{-}3}$ we get $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s}(pk^*)$. This together with the fact that all information related to pk^* remains unchanged implies $\mathsf{swETH\text{-}backed\text{-}withdrawn}_{s'}(pk^*)$.

Case: $ev = \mathsf{redeem}(addr, x)$

Then s' only differs from s in ways that are not relevant for the invariant we are considering here. Thus, $s \in \mathsf{INV\text{-}3}$ implies $s' \in \mathsf{INV\text{-}3}$.

Case: $ev = \mathsf{swap}(addr, nft)$

Then s' only differs from s in ways that are not relevant for the invariant we are considering here. Thus, $s \in \mathsf{INV\text{-}3}$ implies $s' \in \mathsf{INV\text{-}3}$.

□

Main invariant

In order to formulate the invariant that all swETH is backed by ETH, we first need to precisely define to how much ETH is available for use by the Swell contract:

$$\begin{aligned} \mathsf{available\text{-}ETH}(s) := s.\mathsf{C.balance} \\ & + \sum pk \in \mathsf{NonWithdrawnPubKeys}_s\text{: } \mathsf{total\text{-}minted}_s(pk) \\ & + \sum pk \in \mathsf{WithdrawnPubKeys}_s\text{: } (\mathsf{total\text{-}minted}_s(pk) - \mathsf{unassigned\text{-}rewards}_s(pk)) \end{aligned}$$

Intuitively, the ETH that the Swell contract has access to comes from two sources: The ETH that the contract holds itself, and the ETH that staked on the Beacon Chain. To go into more detail, let us look at the individual terms that make up the above sum:

- $s.\mathsf{C.balance}$ This is the amount of ETH that the Swell contract holds itself and thus has immediate access to.
- $\sum pk \in \mathsf{NonWithdrawnPubKeys}_s\text{: } \mathsf{total\text{-}minted}_s(pk)$ This sum only considers validators whose stake has not been withdrawn yet. In particular, it computes the total amount of swETH that has been minted for each validator (identified by their public key pk).

In Lemma 2 we have shown that for any $pk \in \mathsf{NonWithdrawnPubKeys}_s$, the total amount of swETH minted for pk is less then or equal to the corresponding validator balance.

Thus, this sum is backed by ETH which becomes available to the Swell contract once the validator stakes can be withdrawn.

- $\sum pk \in \mathsf{WithdrawnPubKeys}_s$: $(\mathsf{total\text{-}minted}_s(pk) - \mathsf{unassigned\text{-}rewards}_s(pk))$ Here, only withdrawn validators are considered. It is a bit more complicated due to the fact that the Swell contract often does not know the validator state on the Beacon Chain. This means that it is for example possible to stake to a validator even after its balance has been withdrawn.

  The important thing is that Lemma 3 shows that for any $pk \in \mathsf{WithdrawnPubKeys}_s$, the result of $\mathsf{total\text{-}minted}_s(pk) - \mathsf{unassigned\text{-}rewards}_s(pk)$ is smaller than the validator balance, which means it is fully backed by staked ETH.

The above reasoning implies that for any state s, the amount of ETH calculated by $\mathsf{available\text{-}ETH}(s)$ is or will become available to the Swell contract. Thus, if we can show that the total supply of swETH is always less then or equal to $\mathsf{available\text{-}ETH}(s)$, then we have proven that all swETH is backed by ETH. This is captured by INV-4:

Invariant 4. $s \in \mathsf{INV\text{-}4}$ iff the following condition holds.

$$\mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) \leq \mathsf{available\text{-}ETH}(s)$$

Lemma 4. If $s_0 \in \mathsf{InitState}$, then $\mathsf{reachable}(s_0) \subseteq \mathsf{INV\text{-}4}$.

Proof sketch. Let $s_0 \in \mathsf{InitState}$ and $s' \in \mathsf{State}$ be arbitrary such that $s' \in \mathsf{reachable}(s_0)$. We need to show that $s' \in \mathsf{INV\text{-}4}$. The proof is by induction on s'.

Base case Then $s' = s_0$. Since $s_0 \in \mathsf{InitState}$ we also get $s' \in \mathsf{InitState}$.

$s' \in \mathsf{InitState}$ implies $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = 0$ and $\mathsf{available\text{-}ETH}(s') = 0$. Thus, the claim holds trivially.

Inductive case Then there exist s, ev such that $s \in \mathsf{reachable}(s_0)$ and $s \xrightarrow{ev} s'$.

The induction hypothesis implies $s \in \mathsf{INV\text{-}4}$. We make a case distinction on ev.

Case: $ev = \mathsf{reward}(pk, x)$

Nothing of relevance changes.

Case: $ev = \mathsf{penalty}(pk, x)$

Nothing of relevance changes.

Case: $ev = \mathsf{deposit}(pk, x)$

Nothing of relevance changes.

Case: $ev = \mathsf{withdraw}(pk)$

First, define the following variables.

a) $v := s.\mathsf{B.validators}(pk)$

b) $nw\text{-}sum := \sum pk \in \mathsf{NonWithdrawnPubKeys}_s$: $\mathsf{total\text{-}minted}_s(pk)$

c) $nw\text{-}sum' := \sum pk \in \mathsf{NonWithdrawnPubKeys}_{s'}$: $\mathsf{total\text{-}minted}_{s'}(pk)$

d) $w\text{-}sum := \sum pk \in \mathsf{WithdrawnPubKeys}_{s}$: $(\mathsf{total\text{-}minted}_{s}(pk) - \mathsf{unassigned\text{-}rewards}_{s}(pk))$

e) $w\text{-}sum' := \sum pk \in \mathsf{WithdrawnPubKeys}_{s'}$: $(\mathsf{total\text{-}minted}_{s'}(pk) - \mathsf{unassigned\text{-}rewards}_{s'}(pk))$

By the definition of the transition we get

f) $pk \in \mathsf{WithdrawnPubKeys}_{s'}$

g) $s'.\mathsf{C.balance} = s.\mathsf{C.balance} + v.\mathsf{balance}$

h) $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances})$

We make a case distinction on whether this is the first time pk is withdrawn.

Case: $pk \in \mathsf{NonWithdrawnPubKeys}_{s}$

Then

* $nw\text{-}sum' = nw\text{-}sum - \mathsf{total\text{-}minted}_{s}(pk)$
* $w\text{-}sum' = w\text{-}sum - \mathsf{unassigned\text{-}rewards}_{s'}(pk)$
* $\mathsf{unassigned\text{-}rewards}_{s'}(pk) \leq \mathsf{unminted}_{s}(pk)$

  Proof: Since this is a withdraw operation, we know that for all staking positions $p \in \mathsf{positions}_{s'}(pk)$, the conditions $p.nft \notin s.\mathsf{C.backed\text{-}by\text{-}contract}$ and $p.nft \in s'.\mathsf{C.backed\text{-}by\text{-}contract}$ holds. This means that the variable *remaining-stake* as computed by both $\mathsf{unassigned\text{-}rewards}_{s'}(pk)$ and $\mathsf{unminted}_{s}(pk)$ is equal. This directly implies the claim.

To prove $s' \in \mathsf{INV\text{-}4}$, it suffices to show $\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \leq v.\mathsf{balance}$.

Because of Lemma 2 we know

$$\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unminted}_{s}(pk) \leq v.\mathsf{balance}.$$

Then the above directly gives us

$$\mathsf{total\text{-}minted}_{s}(pk) + \mathsf{unassigned\text{-}rewards}_{s'}(pk) \leq v.\mathsf{balance}.$$

Case: $pk \notin \mathsf{NonWithdrawnPubKeys}_{s}$

First, observe that $\mathsf{unassigned\text{-}rewards}_{s'}(pk) \leq \mathsf{unassigned\text{-}rewards}_{s}(pk)$.

Proof: The only thing that can change is that the variable *remaining-stake* as computed by $\mathsf{unassigned\text{-}rewards}_{s'}(pk)$ is smaller than the same variable computed by $\mathsf{unassigned\text{-}rewards}_{s}(pk)$. This directly implies the claim.

Then

* $nw\text{-}sum' = nw\text{-}sum$
* $w\text{-}sum' \geq w\text{-}sum$ (This is because $\mathsf{unassigned\text{-}rewards}_{s'}(pk) \leq \mathsf{unassigned\text{-}rewards}_{s}(pk)$.)

From this we get $s' \in \mathsf{INV\text{-}4}$ because the right-hand side of the inequality increases while the left-hand side remains the same.

Case: $ev = \mathsf{exit}(pk)$

Nothing relevant changes.

Case: $ev = \mathsf{next\text{-}epoch}$

Nothing relevant changes.

Case: $ev = \mathsf{stake}(addr, pk, x)$

First, define the following variables.

a) $\mathit{is\text{-}op} := (\mathsf{validator\text{-}deposits}_s(pk) = 0 \wedge pk \notin s.\mathsf{C.whitelist})$

b) $\mathit{mint\text{-}amount} := \begin{cases} 0 & \text{if } \mathit{is\text{-}op} \\ x & \text{otherwise} \end{cases}$

c) $v := \begin{cases} s.\mathsf{B.validators}(pk) & \text{if } pk \in \mathsf{dom}(s.\mathsf{B.validators}) \\ \mathsf{fresh\text{-}validator}(pk) & \text{otherwise} \end{cases}$

d) $v' := v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]$

e) $\mathit{nw\text{-}sum} := \sum pk \in \mathsf{NonWithdrawnPubKeys}_s \colon \mathsf{total\text{-}minted}_s(pk)$

f) $\mathit{nw\text{-}sum}' := \sum pk \in \mathsf{NonWithdrawnPubKeys}_{s'} \colon \mathsf{total\text{-}minted}_{s'}(pk)$

g) $\mathit{w\text{-}sum} := \sum pk \in \mathsf{WithdrawnPubKeys}_s \colon (\mathsf{total\text{-}minted}_s(pk) - \mathsf{unassigned\text{-}rewards}_s(pk))$

h) $\mathit{w\text{-}sum}' := \sum pk \in \mathsf{WithdrawnPubKeys}_{s'} \colon (\mathsf{total\text{-}minted}_{s'}(pk) - \mathsf{unassigned\text{-}rewards}_{s'}(pk))$

Then by the definition of the transition relation we get

i) $v' = s.\mathsf{B.validators}(pk)$

j) $s'.\mathsf{C.balance} = s.\mathsf{C.balance}$

k) $s'.\mathsf{C.swETH\text{-}balances} = s.\mathsf{C.swETH\text{-}balances} + \mathit{mint\text{-}amount}$

We make a case distinction on whether the validator identified by pk already exists, and if yes, whether it has already been withdrawn.

Case: $pk \in \mathsf{NonWithdrawnPubKeys}_s$

Then

* $\mathit{nw\text{-}sum}' = \mathit{nw\text{-}sum} + \mathit{mint\text{-}amount}$
* $\mathit{w\text{-}sum}' = \mathit{w\text{-}sum}$
* $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) + \mathit{mint\text{-}amount}$

From this we can conclude $s' \in \mathsf{INV\text{-}4}$ because both sides of the inequality increase by *mint-amount*.

Case: $pk \notin \mathsf{NonWithdrawnPubKeys}_s \wedge pk \notin \mathsf{WithdrawnPubKeys}_s$

Same as previous case.

Case: $pk \in \mathsf{WithdrawnPubKeys}_s$

Then

* $nw\text{-}sum' = nw\text{-}sum$
* $w\text{-}sum' = w\text{-}sum + mint\text{-}amount$
* $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) + mint\text{-}amount$

From this we can conclude $s' \in \mathsf{INV\text{-}4}$ because both sides of the inequality increase by $mint\text{-}amount$.

Case: $ev = \mathsf{claim}(nft)$

Let

(a) $p := s.\mathsf{C.positions}(nft)$

(b) $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$

(c) $r := \mathsf{max}(0, v.\mathsf{balance} - p.\mathsf{validator\text{-}balance}) * \frac{p.\mathsf{staking\text{-}amount}}{32}$

(d) $p' := p[\mathsf{validator\text{-}balance} := v.\mathsf{balance}][\mathsf{rewards\text{-}minted} \mathrel{+}= r]$

Then by the definition of the transition relation this implies

(e) $p' = s'.\mathsf{C.positions}(nft)$

(f) $s'.\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.fee\text{-}pool}) = s.\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.fee\text{-}pool}) + r \cdot \mathsf{FEE}$

(g) $s'.\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) = s.\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) + r \cdot (1 - \mathsf{FEE})$

This implies

(h) $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) + r$

(i) $\mathsf{total\text{-}minted}_{s'}(p.\mathsf{pubkey}) = \mathsf{total\text{-}minted}_{s}(p.\mathsf{pubkey}) + r$

Then from (i) we get $\mathsf{available\text{-}ETH}(s') = \mathsf{available\text{-}ETH}(s) + r$. This together (h) and $s \in \mathsf{INV\text{-}4}$ gives us $s' \in \mathsf{INV\text{-}4}$.

Case: $ev = \mathsf{unstake}(addr, nft)$

Then by the transition relation there exists a s'' such that $s \xrightarrow{\mathsf{claim}(nft)} s''$. This implies $s'' \in \mathsf{INV\text{-}4}$ (see case for $\mathsf{claim}(nft)$).

s' is then obtained by applying some more modifications to s''. The only change relevant for INV-4 is that swETH is transferred from $s''.\mathsf{C.this}$ to $addr$. However, this does not change the total supply of swETH. Thus, $s'' \in \mathsf{INV\text{-}4}$ also implies $s' \in \mathsf{INV\text{-}4}$.

Case: $ev = \mathsf{unstake\text{-}op}(addr, nft)$

Then there exist p, p', x and N such that

- $s' = s[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{+}= p.\mathsf{swETH\text{-}balance} + x]$
  $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) \mathrel{-}= p.\mathsf{swETH\text{-}balance}]$
  $[\mathsf{C.positions}(nft) := p']$
  $[\mathsf{C.swNFT\text{-}owners} := N]$

- $p = s.\mathsf{C.positions}(nft)$
- $p'.\mathsf{rewards\text{-}minted} = p.\mathsf{rewards\text{-}minted} + x$

This implies

- $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) + x$
- $\mathsf{available\text{-}ETH}(s') = \mathsf{available\text{-}ETH}(s) + x$

This together $s \in$ INV-4 gives us $s' \in$ INV-4.

Case: $ev = \mathsf{redeem}(addr, x)$

Then

- $s' = s[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{-}= x]$
  $[\mathsf{C.balance} \mathrel{-}= x]$

This implies

- $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances}) - x$
- $\mathsf{available\text{-}ETH}(s') = \mathsf{available\text{-}ETH}(s) - x$

This together with $s \in$ INV-4 implies $s' \in$ INV-4.

Case: $ev = \mathsf{swap}(addr, nft)$

Then

- $\mathsf{total\text{-}supply}(s'.\mathsf{C.swETH\text{-}balances}) = \mathsf{total\text{-}supply}(s.\mathsf{C.swETH\text{-}balances})$
- $\mathsf{available\text{-}ETH}(s') = \mathsf{available\text{-}ETH}(s)$

This together with $s \in$ INV-4 implies $s' \in$ INV-4.

□

Appendix 1: Mathematical model

In this section we develop a simplified mathematical model of the Beacon Chain and the Swell contract. This model forms the basis for the proof in section All swETH is backed by staked ETH.

The model is simplified in the following ways:

- The behavior of the Beacon Chain is over-approximated
- Deposits to the Eth2 Deposit Contract are processed immediately by the Beacon Chain
- When calculating staking rewards, the model always uses the most recent validator balance
- Validators that have been staked to via the Swell contract have withdrawal credentials that allow the Swell contract to withdraw the validator balance once the validator becomes withdrawable
- Arithmetic operations are carried out using real arithmetic. Rounding errors and overflow are not considered

We first introduce some Mathematical notation and then present the Transition system.

Mathematical notation

Equality We use $:=$ to denote *definitional equality*: The assertion that the newly introduced name on the left is equal to the term on the right. On the other hand, we use $=$ to denote *propositional equality*: The proposition that the term on the left is equal to the term on the right, which may or may not be true. Propositions denote a value from the following set:

$$\mathbb{B} := \{\mathsf{true}, \mathsf{false}\}$$

For example, the proposition $a = b$ may either hold, in which case it denotes the value true, or it may not hold, in which case it denotes the value false.

Sets We use $\mathbb{R}^+ := \{x \in \mathbb{R} \mid x \geq 0\}$ to denote the set of all non-negative real numbers. Further, we define $\mathbb{N}^\infty := \mathbb{N} \cup \{\infty\}$ such that $\forall n \in \mathbb{N}\colon n < \infty$. We assume $0 \in \mathbb{N}$.

Functions If A and B are sets, then $A \to B$ denotes the set of total functions from A to B, and $A \rightharpoonup B$ denotes the set of partial functions from A to B. The *domain* of a function f is denoted by $\mathsf{dom}(f)$. That is, $\mathsf{dom}(f) := \{x \mid \exists y\colon f(x) = y\}$. Similarly, the *range* of a function f is given by $\mathsf{rng}(f) := \{y \mid \exists x\colon f(x) = y\}$.

Records To make the usage of complex data structures concise, we introduce a record notation. A record type is a set of tuples together with some functions to access the individual elements.

$$R := (f_1 : S_1) \times \ldots \times (f_n : S_n)$$

The above definition introduces the record type R such that $R = S_1 \times \ldots \times S_n$. Additionally, it defines the functions $f_i : R \to S_i$ for all $i \in \{1, \ldots, n\}$. If $r \in R$ with $r = (s_1, \ldots, s_n)$, then $f_i(r) = s_i$. However, instead of $f_i(r)$ we will usually write $r.f_i$.

Update notation We do not formally define the semantics of our notation for updating the fields of records, but instead give a couple of examples. If r denotes a record with field f, then $r[f := x]$ denotes a new record that is equal to r for all fields except f, which has the value x. Further, if field f itself denotes a record with field g, then we write $r[f.g := y]$ to update the nested field g.

Instead of $:=$ we also allow the usage of $+=$ and $-=$. For example, if $r' = r[f \mathrel{+}= x]$, then $r'.f = r.f + x$.

Finally, we also support function updates: If field f denotes a function and $r' = r[f(a) := b]$, then $r'.f(a) = b$ and $\forall x \in \mathsf{dom}(r.f)\colon x \neq a \Rightarrow r'.f(x) = r.f(x)$.

Transition system

State

We use Address to denote the set of Eth1 addresses, BLSPublicKey to denote the set of BLS public keys used to identify validators on the Beacon Chain, and swNFT to denote the set of all swNFTs. We assume all these sets are finite but leave them otherwise underspecified.

Validators on the Beacon Chain are represented by the following record type:

$$\begin{aligned}\mathsf{BeaconChainValidator} := {} & (\mathsf{pubkey} : \mathsf{BLSPublicKey}) \times \\ & (\mathsf{balance} : \mathbb{R}^+) \times \\ & (\mathsf{ideal\text{-}balance} : \mathbb{R}^+) \times \\ & (\mathsf{final\text{-}balance} : \mathbb{R}^+) \times \\ & (\mathsf{withdrawn\text{-}at\text{-}least\text{-}once} : \mathbb{B}) \times \\ & (\mathsf{activation\text{-}epoch} : \mathbb{N}^\infty) \times \\ & (\mathsf{exit\text{-}epoch} : \mathbb{N}^\infty) \times \\ & (\mathsf{withdrawable\text{-}epoch} : \mathbb{N}^\infty)\end{aligned}$$

The field pubkey denotes the validator public key and uniquely identifies the validator. balance denotes the current active balance of the validator. It increases whenever a deposit is made or rewards are earned, and decreases whenever a penalty is incurred or the validator stake is withdrawn. ideal-balance is computed the same way as balance, except that it is not affected by penalties. Thus, at any time, ideal-balance denotes the balance that the validator would have if it never got penalized. final-balance is set to the current validator balance right before the validator becomes withdrawable. withdrawn-at-least-once is set to true the first time the validator balance is withdrawn and then stays that way. The fields activation-epoch, exit-epoch and withdrawable-epoch denote the epochs when the validator gets is activated, exits the Beacon Chain, and becomes withdrawable, respectively.

Note that the fields ideal-balance and withdrawn-at-least-once are *ghost fields* in the sense that they are only used to formulate our invariants; they are not needed by the transition relation.

A fresh validator is created using the following function:

$$\mathsf{fresh\text{-}validator}(pk) := (pk, 0, 0, 0, \mathsf{false}, \infty, \infty, \infty)$$

Note that activation-epoch, exit-epoch and withdrawable-epoch are initialized to ∞, which denotes an epoch infinitely far into the future. This means that the validator is initially neither activated, exited, nor withdrawable.

The full state of our simplified model of the Beacon Chain is defined as follows:

$$\mathsf{BeaconChainState} := (\mathsf{epoch} : \mathbb{N}) \times (\mathsf{validators} : \mathsf{BLSPublicKey} \rightharpoonup \mathsf{BeaconChainValidator})$$

It only consists of two fields: epoch stores the current epoch, and validators is a partial function mapping BLS public keys to their corresponding validator.

We now move on to define the state of the Swell contract. To start off, here is the record type for staking positions.

$$\begin{aligned}\mathsf{Position} := {} & (\mathsf{pubkey} : \mathsf{BLSPublicKey}) \times \\ & (\mathsf{nft} : \mathsf{swNFT}) \times \\ & (\mathsf{staking\text{-}amount} : \mathbb{R}^+) \times \\ & (\mathsf{swETH\text{-}balance} : \mathbb{R}^+) \times \\ & (\mathsf{validator\text{-}balance} : \mathbb{R}^+) \times \\ & (\mathsf{rewards\text{-}minted} : \mathbb{R}^+) \times \\ & (\mathsf{by\text{-}operator} : \mathbb{B})\end{aligned}$$

The field pubkey denotes the validator of the staking position, and *nft* denotes the swNFT that uniquely identifies the position. staking-amount stores the amount of ETH that was initially staked by the staker. swETH-balance keeps track of the amount of swETH currently held by the staking position. The field validator-balance stores the validator balance at the time when staking started and is used to calculate rewards. rewards-minted denotes the amount of swETH that has been minted as rewards, and by-operator is true if and only if the staking position was created by a node operator.

Next, let us look at the full state for the Swell contract:

$$\begin{aligned}\mathsf{ContractState} := {} & (\mathsf{positions} : \mathsf{swNFT} \rightharpoonup \mathsf{Position}) \times \\ & (\mathsf{balance} : \mathbb{R}^+) \times \\ & (\mathsf{swETH\text{-}balances} : \mathsf{Address} \to \mathbb{R}^+) \times \\ & (\mathsf{swNFT\text{-}owners} : \mathsf{swNFT} \rightharpoonup \mathsf{Address}) \times \\ & (\mathsf{whitelist} : \mathcal{P}(\mathsf{BLSPublicKey})) \times \\ & (\mathsf{fee\text{-}pool} : \mathsf{Address}) \times \\ & (\mathsf{backed\text{-}by\text{-}contract} : \mathcal{P}(\mathsf{swNFT}) \times \\ & (\mathsf{this} : \mathsf{Address})\end{aligned}$$

The field positions is a partial function mapping swNFTs to their staking position. balance denotes the ETH balance of the contract. swETH-balances models the swETH ERC20 token and keeps track of the swETH balance for each address. Similarly, swNFT-owners models the swNFT ERC721 token and keeps track of the owner of each existing swNFT. whitelist stores the public keys of all those validators that do not need the node operator to deposit 16 ETH. The field fee-pool denotes the address to which collected fees are transferred. backed-by-contract keeps track of all staking positions (identified by their swNFT) whose validator has been withdrawn. This means that the swETH minted for such positions is backed by ETH held by the Swell contract and not by staked ETH on the Beacon Chain. Finally, the field this stores the address of the Swell contract.

Having defined the state for both the Beacon Chain and the Swell contract, we can define the full

state of the transition system:

$$\mathsf{State} := (\mathsf{B} : \mathsf{BeaconChainState}) \times (\mathsf{C} : \mathsf{ContractState})$$

The set of initial states $\mathsf{InitState} \subseteq \mathsf{State}$ is defined such that $s \in \mathsf{InitState}$ iff the following conditions hold:

1. $s.\mathsf{B.epoch} = 0$
2. $\mathsf{dom}(s.\mathsf{B.validators}) = \emptyset$
3. $\mathsf{dom}(s.\mathsf{C.positions}) = \emptyset$
4. $s.\mathsf{C.balance} = 0$
5. $s.\mathsf{C.swETH\text{-}balances}(addr) = 0$ for all $addr \in \mathsf{Address}$
6. $\mathsf{dom}(s.\mathsf{C.swNFT\text{-}owners}) = \emptyset$
7. $\mathsf{dom}(s.\mathsf{C.whitelist}) = \emptyset$
8. $s.\mathsf{C.backed\text{-}by\text{-}contract} = \emptyset$

Utility definitions

The first batch of definitions concerns the public keys of validators:

$$\begin{aligned}
\mathsf{RegisteredPubKeys}_s &:= \{p.\mathsf{pubkey} \mid p \in \mathsf{rng}(s.\mathsf{C.positions})\} \\
\mathsf{WithdrawnPubKeys}_s &:= \{pk \in \mathsf{RegisteredPubKeys}_s \mid \exists v\text{: } v = s.\mathsf{B.validators}(pk) \wedge \\
&\qquad\qquad v.\mathsf{withdrawn\text{-}at\text{-}least\text{-}once}\} \\
\mathsf{NonWithdrawnPubKeys}_s &:= \{pk \in \mathsf{RegisteredPubKeys}_s \mid \exists v\text{: } v = s.\mathsf{B.validators}(pk) \wedge \\
&\qquad\qquad \neg v.\mathsf{withdrawn\text{-}at\text{-}least\text{-}once}\}
\end{aligned}$$

The family of sets $\mathsf{RegisteredPubKeys}_s$ contains all validator public keys for which staking positions have been created in state s. In other words: All validator public keys that have been staked to using the Swell contract. We also define sets that contain the public keys of validators that have or have not been withdrawn.

Next, we define several functions to retrieve the staking positions of some validator identified by its public key pk:

$$\begin{aligned}
\mathsf{positions}_s(pk) &:= \{p \in \mathsf{rng}(s.\mathsf{C.positions}) \mid p.\mathsf{pubkey} = pk\} \\
\mathsf{staker\text{-}positions}_s(pk) &:= \{p \in \mathsf{positions}_s(pk) \mid \neg p.\mathsf{by\text{-}operator}\} \\
\mathsf{op\text{-}positions}_s(pk) &:= \{p \in \mathsf{positions}_s(pk) \mid p.\mathsf{by\text{-}operator}\}
\end{aligned}$$

We continue with several functions that operate over a valdiator's staking positions:

$$\begin{aligned}
\mathsf{is\text{-}staked}_s(p) &:= p.\mathsf{nft} \in \mathsf{dom}(s.\mathsf{C.swNFT\text{-}owners}) \\
\mathsf{stake\text{-}minted}_s(pk) &:= \sum p \in \mathsf{staker\text{-}positions}_s(pk), p.\mathsf{nft} \notin s.\mathsf{C.backed\text{-}by\text{-}contract}\text{: } p.\mathsf{staking\text{-}amount} \\
\mathsf{rewards\text{-}minted}_s(pk) &:= \sum p \in \mathsf{positions}_s(pk), p.\mathsf{nft} \notin s.\mathsf{C.backed\text{-}by\text{-}contract}\text{: } p.\mathsf{rewards\text{-}minted} \\
\mathsf{total\text{-}minted}_s(pk) &:= \mathsf{stake\text{-}minted}_s(pk) + \mathsf{rewards\text{-}minted}_s(pk) \\
\mathsf{collateral}_s(pk) &:= \sum p \in \mathsf{op\text{-}positions}_s(pk), \mathsf{is\text{-}staked}_s(p)\text{: } p.\mathsf{staking\text{-}amount} \\
\mathsf{validator\text{-}deposits}_s(pk) &:= \sum p \in \mathsf{positions}_s(pk)\text{: } p.\mathsf{staking\text{-}amount}
\end{aligned}$$

The predicate $\mathsf{is\text{-}staked}_s(p)$ is true for staking position p if its swNFT has not been burned yet. (We keep staking positions around even after the corresponding swNFT has been burned since this makes it easier to formulate certain invariants.) The function $\mathsf{stake\text{-}minted}_s(pk)$ computes the amount of swETH that has been minted for pk (excluding rewards) and that is backed by ETH on the Beacon Chain (which is why it only considers positions for which $p.\mathsf{nft} \notin s.\mathsf{C.backed\text{-}by\text{-}contract}$). $\mathsf{rewards\text{-}minted}_s(pk)$ is similar, but only considers swETH that was minted because of rewards. Next, the function $\mathsf{collateral}_s(pk)$ computes the amount of collateral that is provided by the node operator as a buffer for penalties. For non-whitelisted validators, this is at least 16 ETH. Finally, $\mathsf{validator\text{-}deposits}_s(pk)$ computes the total amount of deposits that have been made to pk via the Swell contract. This can at most be 32 ETH. (Note that more deposits may have been made to pk outside of the Swell contract, i.e., directly via the Eth2 Deposit Contract.)

The next function is used when computing the rewards for a staking position:

$$\mathsf{reward\text{-}balance}_s(pk) := \begin{cases} v.\mathsf{balance} & \text{if } s.\mathsf{B.epoch} < v.\mathsf{withdrawable\text{-}epoch} \\ v.\mathsf{final\text{-}balance} & \text{otherwise,} \end{cases}$$

where $v := s.\mathsf{B.validators}(pk)$.

Rewards are calculated based on the current validator balance. However, when the validator balance is withdrawn, its balance drops to zero. This would mean that any staker who unstakes after the validator balance has been withdrawn would not get any rewards. For this reason, once a validator becomes withdrawable, we only consider its final balance right before it became withdrawable.

Finally, for a function $bals : \mathsf{Address} \to \mathbb{R}^+$ that stores token balances, we define the following function to compute the total token supply:

$$\mathsf{total\text{-}supply}(bals) := \sum addr \in \mathsf{Address}\colon bals(addr).$$

Transition relation

The events of the transition system can be categorized into those that affect the Beacon Chain and those that affect the Swell contract:

$$\begin{aligned} \mathsf{BeaconEvent} := \{&\mathsf{reward}(pk, x), \mathsf{penalty}(pk, x), \mathsf{deposit}(pk, x), \mathsf{exit}(pk), \mathsf{withdraw}(pk), \mathsf{next\text{-}epoch} \mid \\ &pk \in \mathsf{BLSPublicKey}, x \in \mathbb{R}^+\} \end{aligned}$$

$$\begin{aligned} \mathsf{ContractEvent} := \{&\mathsf{stake}(addr, pk, x), \mathsf{claim}(nft), \mathsf{unstake}(addr, nft), \mathsf{unstake\text{-}op}(addr, nft), \mathsf{redeem}(addr, x), \\ &\mathsf{swap}(addr, nft) \mid pk \in \mathsf{BLSPublicKey}, x \in \mathbb{R}^+, nft \in \mathsf{swNFT}, addr \in \mathsf{Address}\} \end{aligned}$$

$$\mathsf{Event} := \mathsf{BeaconEvent} \cup \mathsf{ContractEvent}$$

The event $\mathsf{reward}(pk, x)$ means that the validator identified by pk has been rewarded x ETH. $\mathsf{penalty}(pk, x)$ and $\mathsf{deposit}(pk, x)$ then have their obvious meaning. $\mathsf{exit}(pk)$ initiates the exit of pk from the Beacon Chain by setting both the validator's exit and withdrawal epoch. Once the withdrawal epoch of a validator has been reached, its balance can be withdrawn by the $\mathsf{withdraw}(pk)$ event. Finally, $\mathsf{next\text{-}epoch}$ simply moves the Beacon Chain to the next epoch.

Switching to the Swell contract, the event $\mathsf{stake}(addr, pk, x)$ denotes a staking operation by sender $addr$ to the validator identified by pk and a staking amount of x ETH. $\mathsf{claim}(nft)$ collects the rewards for the staking position identified by the swNFT nft and mints an appropriate amount of swETH.

The event $\mathsf{unstake}(addr, nft)$ allows the sender $addr$ to unstake the staking position identified by nft. However, note that this event only allows unstaking for stakers. Node operators can unstake using the $\mathsf{unstake\text{-}op}(addr, nft)$ event. $\mathsf{redeem}(addr, x)$ allows the sender $addr$ to exchange x swETH for the same amount of ETH. Finally, $\mathsf{swap}(addr, nft)$ allows the sender to purchase the staking position identified by nft.

The transition relation $\rightarrow \subseteq \mathsf{State} \times \mathsf{Event} \times \mathsf{State}$ is defined such that $s \xrightarrow{ev} s'$ holds if and only if one of the following conditions is satisfied.

- $ev = \mathsf{reward}(pk, x)$ and
  - $s.\mathsf{B.validators}(pk).\mathsf{activation\text{-}epoch} \leq s.\mathsf{B.epoch}$
  - $s.\mathsf{B.epoch} < s.\mathsf{B.validators}(pk).\mathsf{withdrawable\text{-}epoch}$
  - $s' = s[\mathsf{B.validators}(pk)[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]]$

  Since rewards are computed for the previous epoch, it seems a validator may get rewarded $\mathsf{exit\text{-}epoch} + 1$*. To be safe and because it doesn't matter for our case, we assume that rewards can be applied until* $\mathsf{withdrawable\text{-}epoch}$*.*
- $ev = \mathsf{penalty}(pk, x)$ and
  - $s.\mathsf{B.validators}(pk).\mathsf{activation\text{-}epoch} \leq s.\mathsf{B.epoch}$
  - $s.\mathsf{B.epoch} < s.\mathsf{B.validators}(pk).\mathsf{withdrawable\text{-}epoch}$
  - $x \leq s.\mathsf{balance}$
  - $s' = s[\mathsf{B.validators}(pk)[\mathsf{balance} \mathrel{-}= x]]$
- $ev = \mathsf{deposit}(pk, x)$ and $s' = s[\mathsf{B.validators}(pk) := v']$, where
  - $v' := v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]$
  - $v := \begin{cases} s.\mathsf{B.validators}(pk) & \text{if } pk \in \mathsf{dom}(s.\mathsf{B.validators}) \\ \mathsf{fresh\text{-}validator}(pk) & \text{otherwise} \end{cases}$
- $ev = \mathsf{withdraw}(pk)$ and
  - $s.\mathsf{B.validators}(pk).\mathsf{withdrawable\text{-}epoch} \leq s.\mathsf{B.epoch}$
  - $s' = s[\mathsf{B.validators}(pk)[\mathsf{balance} := 0, \mathsf{ideal\text{-}balance} := 0, \mathsf{withdrawn\text{-}at\text{-}least\text{-}once} := \mathsf{true}]]$
    $[\mathsf{C.backed\text{-}by\text{-}contract} := \mathsf{C.backed\text{-}by\text{-}contract} \cup \{p.\mathsf{nft} \mid p \in \mathsf{positions}_s(pk)\}]$
    $[\mathsf{C.balance} \mathrel{+}= x]$

  where
  $$x := \begin{cases} s.\mathsf{B.validators}(pk).\mathsf{balance} & \text{if } pk \in \mathsf{RegisteredPubKeys}_s \\ 0 & \text{otherwise} \end{cases}$$
- $ev = \mathsf{exit}(pk)$ for some pk, and
  - $s.\mathsf{B.validators}(pk).\mathsf{exit\text{-}epoch} = \infty$
  - $s' = s[\mathsf{B.validators}(pk)[\mathsf{exit\text{-}epoch} := e_1, \mathsf{withdrawable\text{-}epoch} := e_2]]$
    where $e_1, e_2 \in \mathbb{N}$ such that $s.\mathsf{B.epoch} < e_1$ and $e_1 < e_2$.

- $ev = \mathsf{next\text{-}epoch}$ and $s' = s[\mathsf{B.epoch} \mathrel{+}= 1][\mathsf{B.validators} := V]$, where

$$V(pk) := \begin{cases} v[\mathsf{final\text{-}balance} := v.\mathsf{balance}] & \text{if } v.\mathsf{withdrawable\text{-}epoch} = s.\mathsf{B.epoch} \\ v & \text{otherwise} \end{cases}$$

- $ev = \mathsf{stake}(addr, pk, x)$ and
  - $\mathsf{validator\text{-}deposits}_s(pk) + x \leq 32$
  - if *is-op* then $16 \leq x$, else $1 \leq x$
  - $s' = s[\mathsf{B.validators}(pk) := v[\mathsf{balance} \mathrel{+}= x, \mathsf{ideal\text{-}balance} \mathrel{+}= x]]$
    $[\mathsf{C.swNFT\text{-}owners}(addr) := nft]$
    $[\mathsf{C.positions}(nft) := p]$
    $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) \mathrel{+}= \mathit{mint\text{-}amount}]$

  where
  - $v := \begin{cases} s.\mathsf{B.validators}(pk) & \text{if } pk \in \mathsf{dom}(s.\mathsf{B.validators}) \\ \mathsf{fresh\text{-}validator}(pk) & \text{otherwise} \end{cases}$
  - $\mathit{is\text{-}op} := (\mathsf{validator\text{-}deposits}_s(pk) = 0 \wedge pk \notin s.\mathsf{C.whitelist})$
  - $\mathit{mint\text{-}amount} := \begin{cases} 0 & \text{if } \mathit{is\text{-}op} \\ x & \text{otherwise} \end{cases}$
  - Choose $nft \in \mathsf{swNFT}$ such that $nft \notin \mathsf{dom}(s.\mathsf{C.swNFT\text{-}owners})$
  - $p := (\mathsf{pubkey} : pk,$
    $\mathsf{nft} : nft,$
    $\mathsf{staking\text{-}amount} : x,$
    $\mathsf{swETH\text{-}balance} : \mathit{mint\text{-}amount},$
    $\mathsf{validator\text{-}balance} : 32,$
    $\mathsf{rewards\text{-}minted} : 0,$
    $\mathsf{by\text{-}operator} : \mathit{is\text{-}op})$

  Note that we assume that deposits made to the Eth2 Deposit Contract are immediately reflected on the Beacon Chain.

- $ev = \mathsf{claim}(nft)$ and
  - $\mathsf{is\text{-}staked}_s(p)$
  - $\neg p.\mathsf{by\text{-}operator}$
  - $\mathsf{validator\text{-}deposits}_s(p.\mathsf{pubkey}) = 32$
  - $s' = s[\mathsf{C.positions}(nft) := p']$
    $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) \mathrel{+}= r \cdot (1 - \mathsf{FEE})]$
    $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.fee\text{-}pool}) \mathrel{+}= r \cdot \mathsf{FEE}]$

  where

- $p := s.\mathsf{C.positions}(nft)$
- $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$
- $r := \max(0, \mathsf{reward\text{-}balance}_s(p.\mathsf{pubkey}) - p.\mathsf{validator\text{-}balance}) * \frac{p.\mathsf{staking\text{-}amount}}{32}$
- $p' := p[\mathsf{validator\text{-}balance} := \mathsf{reward\text{-}balance}_s(p.\mathsf{pubkey})]$
  $[\mathsf{rewards\text{-}minted} \mathrel{+}= r]$
  $[\mathsf{swETH\text{-}balance} \mathrel{+}= r]$

Note that we directly access the validator balance from the Beacon Chain. This is equivalent to the assumption that the BeaconChainOracle in the pseudo-code model defined in Appendix 2 always returns the current validator balance.

- $ev = \mathsf{unstake}(addr, nft)$ and there exists a s'' such that
  - $s \xrightarrow{\mathsf{claim}(nft)} s''$
  - $addr = s.\mathsf{C.swNFT\text{-}owners}(nft)$
  - $s' = s''[\mathsf{C.positions}(nft) := p']$
    $[\mathsf{C.swETH\text{-}balances}(s''.\mathsf{C.this}) \mathrel{-}= p.\mathsf{swETH\text{-}balance}]$
    $[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{+}= p.\mathsf{swETH\text{-}balance}]$
    $[\mathsf{C.swNFT\text{-}owners} := N]$

  where
  - $p := s''.\mathsf{C.positions}(nft)$
  - $p' := p[\mathsf{swETH\text{-}balance} := 0]$
  - If $v.\mathsf{exit\text{-}epoch} \leq s''.\mathsf{B.epoch}$ then define N such $\mathsf{dom}(N) = \mathsf{dom}(s''.\mathsf{C.swNFT\text{-}owners}) \setminus \{nft\}$ and $\forall nft' \in \mathsf{dom}(N)$: $N(nft') = s''.\mathsf{C.swNFT\text{-}owners}(nft')$.

    Otherwise, $N := s''.\mathsf{C.swNFT\text{-}owners}[nft := s''.\mathsf{C.this}]$.
  - $v := s''.\mathsf{B.validators}(p.\mathsf{pubkey})$
- $ev = \mathsf{unstake\text{-}op}(addr, nft)$ and
  - $addr = s.\mathsf{C.swNFT\text{-}owners}(nft)$
  - $\mathsf{validator\text{-}deposits}_s(v.\mathsf{pubkey}) = 32$
  - $p.\mathsf{by\text{-}operator}$
  - $v.\mathsf{withdrawable\text{-}epoch} \leq s.\mathsf{B.epoch}$
  - $\{p^* \in \mathsf{positions}_s(v.\mathsf{pubkey}) \mid \mathsf{is\text{-}staked}_s(p^*)\} = \{p\}$
  - $\mathsf{total\text{-}minted}_s(v.\mathsf{pubkey}) \leq v.\mathsf{final\text{-}balance}$
  - $s' = s[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{+}= p.\mathsf{swETH\text{-}balance} + x]$
    $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) \mathrel{-}= p.\mathsf{swETH\text{-}balance}]$
    $[\mathsf{C.positions}(nft) := p']$
    $[\mathsf{C.swNFT\text{-}owners} := N]$

where

- $p := s.\mathsf{C.positions}(nft)$
- $p' := p[\mathsf{rewards\text{-}minted} \mathrel{+}= x]$
  $[\mathsf{validator\text{-}balance} := v.\mathsf{final\text{-}balance}]$
  $[\mathsf{swETH\text{-}balance} := 0]$
- $v := s.\mathsf{B.validators}(p.\mathsf{pubkey})$
- $x := v.\mathsf{final\text{-}balance} - \mathsf{total\text{-}minted}_s(v.\mathsf{pubkey})$
- Define N such $\mathsf{dom}(N) = \mathsf{dom}(s.\mathsf{C.swNFT\text{-}owners}) \setminus \{nft\}$ and $\forall nft' \in \mathsf{dom}(N)$: $N(nft') = s.\mathsf{C.swNFT\text{-}owners}(nft')$

- $ev = \mathsf{swap}(addr, nft)$ and
  - $s.\mathsf{C.this} = s.\mathsf{C.swNFT\text{-}owners}(nft)$
  - $s.\mathsf{C.swETH\text{-}balances}(addr) \geq x$
  - $s' = s[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{-}= x]$
    $[\mathsf{C.swETH\text{-}balances}(s.\mathsf{C.this}) \mathrel{+}= x]$
    $[\mathsf{C.swNFT\text{-}owners}(nft) := addr]$
    $[\mathsf{C.positions}(nft).\mathsf{swETH\text{-}balance} := x]$

  where

  - $x := s.\mathsf{C.positions}(nft).\mathsf{staking\text{-}amount}$

- $ev = \mathsf{redeem}(addr, x)$ and
  - $s.\mathsf{C.swETH\text{-}balances}(addr) \geq x$
  - $s.\mathsf{C.balance} \geq x$
  - $s' = s[\mathsf{C.swETH\text{-}balances}(addr) \mathrel{-}= x]$
    $[\mathsf{C.balance} \mathrel{-}= x]$

  Note that since we do not fully model ETH balances for all addresses, we only burn the required amount of ETH held by the contract, but do not actually send it to the user.

Finally, we define the set of states reachable from some initial state. In particular, for a state $s_0 \in \mathsf{State}$, let $\mathsf{reachable}(s_0) \subseteq \mathsf{State}$ denote the smallest set such that the following conditions hold.

1. $s_0 \in \mathsf{reachable}(s_0)$
2. If $s \in \mathsf{reachable}(s_0)$ and there exist s', ev such that $s \xrightarrow{ev} s'$, then $s' \in \mathsf{reachable}(s_0)$

Appendix 2: Pseudo-code model

The following model is written in a Solidity-inspired pseudo code and describes our understanding of the core functions of the Swell contract. Note that the type `num` represents a mathematical real number.

```
contract IBeaconChainOracle {
    // If the current epoch is larger than the withdrawable_epoch of the
    // validator, then return the validator balance at withdrawable_epoch - 1.
    // This ensures that even if a user unstakes after the stake has been
    // withdrawn, he still gets his rewards.
    function getValidatorBalance(BLSPublicKey pubKey);

    function validatorHasExited(BLSPublicKey pubKey);
}


// - Does not implement enterStrategy()/exitStrategy() for now
// - Assumes a simplified Eth2 Deposit Contract
// - Uses real numbers
contract SwellNetwork {
    struct Position {
        BLSPublicKey pubKey;
        swNFTTokenID nft;
        num stakingAmount;
        num swETHBalance;
        num validatorBalance;
        // Sum of all swETH rewards that have been minted for this position
        num rewardsMinted;
        bool operator;
    }

    mapping(BLSPublicKey => bool) whitelist;
    mapping(BLSPublicKey => num) numStaked;
    mapping(swNFTTokenID => Position) positions;

    swETH swETH_Token;
    swNFT swNFT_Token;
    address feePool; // Address to which collected fees are transferred

    IBeaconChainOracle beaconOracle;
    IDepositContract depositContract;


    //============================================
    // onlyOwner functions
    //============================================
    function addWhitelistedValidator(BLSPublicKey pubKey) onlyOwner {
        whitelist[pubKey] = true;
    }


    //============================================
    // Permissionless function
```

```
//==========================================
function stake(
    BLSPublicKey pubKey,
    num amount
) {
    require(amount >= 1 ether, "Must be at least 1 ETH");
    require(amount % 1 ether == 0, "Must be a multiple of 1 ETH");
    require(
        validatorDeposits(pubKey) + amount <= 32 ether,
        "cannot stake more than 32 ETH"
    );

    bool isFirstDeposit = validatorDeposits(pubKey) == 0;
    bool isOperator = isFirstDeposit && !whitelist[pubKey];
    if(isOperator)
        require(amount >= 16 ether, "Node operators must deposit at least ETH");

    depositContract.deposit(pubKey, amount);

    // Mint swETH
    num swETHAmount = isOperator ? 0 : amount;
    swETH_Token.mint(this, swETHAmount);

    // Mint an swNFT token for the staker
    swNFTTokenID nft = swNFT_Token.mintFor(msg.sender);
    positions[nft] = Position({
        pubKey: pubKey,
        nft: nft,
        stakingAmount: amount,
        swETHBalance: swETHAmount,
        validatorBalance: 32 ether,
        rewardsMinted: 0,
        operator: isOperator,
    });

    numStaked[pubKey] += 1;

    return nft;
}

function deposit(swNFTTokenID nft, num amount) {
    require(swNFT_Token.ownerOf(nft) == msg.sender);

    positions[nft].swETHBalance += amount;
    swETH_Token.transfer(msg.sender, this, amount);
}

function withdraw(swNFTTokenID nft, num amount) {
    require(swNFT_Token.ownerOf(nft) == msg.sender);
    require(positions[nft].swETHBalance >= amount);

    positions[nft].swETHBalance -= amount;
    swETH_Token.transfer(this, msg.sender, amount);
}
```

```
// Mint swETH for rewards earned on Beacon Chain and transfer them to the swNFT
function claim(swNFTTokenID nft) {
    require(swNFT_Token.exists(nft), "swNFT does not exist (anymore)");

    Position p = positions[nft];
    require(!p.operator, "Node operators cannot claim rewards");
    require(
        validatorDeposits(pubKey) == 32,
        "Cannot claim rewards before validator is fully staked"
    );

    // Compute rewards the validator has earned since the user started staking
    num activeBalanceNow = beaconOracle.getValidatorBalance(p.pubKey);
    num validatorRewards = max(0, activeBalanceNow - p.validatorBalance);

    // Compute the user's share of the rewards
    num userRewards = validatorRewards * (p.stakingAmount / 32);
    num feeAmount = userRewards * FEE;
    num actualUserRewards = userRewards - feeAmount;

    // Mint swETH for rewards
    swETH_Token.mint(this, userRewards);
    swETH_Token.transfer(this, feePool, feeAmount);
    p.rewardsMinted += userRewards;
    p.swETHBalance += actualUserRewards;

    p.validatorBalance = activeBalanceNow;
}

function unstake(swNFTTokenID nft) {
    require(swNFT_Token.ownerOf(nft) == msg.sender, "Only owner of swNFT can unstake");
    require(!positions[nft].operator, "Staking position not by staker");

    claim(nft);

    // Transfer all swETH contained in the swNFT to user
    swETH_Token.transfer(this, msg.sender, positions[nft].swETHBalance);
    positions[nft].swETHBalance = 0;

    // If the validator has exited the Beacon Chain, we burn the swNFT.
    // Otherwise, we transfer ownership to the contract so that someone else can buy it.
    if(beaconOracle.validatorHasExited(positions[nft].pubKey)) {
        _burnNFT(nft);
        numStaked[positions[nft].pubKey] -= 1;
    }
    else
        swNFT_Token.transfer(msg.sender, this, nft);
}

function unstakeNodeOperator(swNFTToken nft) {
    require(swNFT_Token.ownerOf(nft) == msg.sender, "Only owner of swNFT can unstake");
    require(
        validatorDeposits(pubKey) == 32,
```

```
            "Only allowed to unstake if validator is fully staked"
        );

        Position p = positions[nft];

        require(p.operator, "Staking position not by node operator");
        require(
            isWithdrawable(p.pubKey),
            "Node operator may only unstake once validator stake is withdrawable"
        );

        require(numStaked[p.pubKey] == 1, "Node operator must unstake last");

        num swETH_minted_initially = validatorDeposits(p.pubKey) - p.stakingAmount;
        num swETH_minted_total = swETH_minted_initially + rewardsMintedForValidator(p.pubKey);
        num activeBalanceNow = beaconOracle.getValidatorBalance(p.pubKey);
        require(swETH_minted_total <= activeBalanceNow, "Not all minted swETH is backed by ETH");

        num swETH_unminted = activeBalanceNow - swETH_minted_total;
        swETH_Token.mint(msg.sender, swETH_unminted);
        swETH_Token.transfer(this, msg.sender, p.swETHBalance);
        p.rewardsMinted += swETH_unminted;
        p.validatorBalance = activeBalanceNow;
        p.swETHBalance = 0;

        numStaked[p.pubKey] -= 1;
        _burnNFT(nft);
    }

    function swap(swNFTTokenID nft) {
        require(swNFT_Token.ownerOf(nft) == this);

        Position p = positions[nft];

        swETH_Token.transfer(msg.sender, this, p.stakingAmount);
        p.swETHBalance = p.stakingAmount;
        swNFTTokenID.transfer(this, msg.sender, nft);
    }

    function redeem(num amount) {
        require(swETH_Token.balanceOf(msg.sender) >= amount, "Not enough swETH available");

        // Burn swETH
        swETH_Token.transfer(msg.sender, this, amount);
        swETH_Token.burn(amount);

        // Transfer ETH
        msg.sender.transfer(amount);
    }


    //==========================================
    // Utility functions
    //==========================================
```

```
    function rewardsMintedForValidator(BLSPublicKey pubKey) {
        num totalRewardsMinted = 0;
        for(Position p: positions.values()) {
            if(p.pubKey == pubKey)
                totalRewardsMinted += p.rewardsMinted;
        }

        return totalRewardsMinted;
    }

    function validatorDeposits(BLSPublicKey pubKey) {
        num amountDeposited = 0;
        for(Position p: positions.values()) {
            if(p.pubKey == pubKey)
                amountDeposited += p.stakingAmount;
        }

        return amountDeposited;
    }

    function _burnNFT(swNFTTokenID nft) {
        swNFT_Token.burn(nft);

        // A real implementation might also want to `delete positions[nft]`.
        // However, we don't do this in the model because keeping the state
        // makes it easier to formulate invariants.
    }
}
```

Appendix 3: Simplified contract used for SMTChecker

```
pragma solidity ^0.8.11;

// - Does not implement enterStrategy()/exitStrategy() for now
// - Does not consider whiteList
// - Uses real numbers

contract swNFTUpgrade {
    uint constant UINT_MAX = ~uint256(0);

    struct Position {
        uint pubKey;
        uint stakingAmount;
        uint swETHBalance;
        uint validatorBalance;
        uint rewardsMinted;
        bool operator;
    }

    mapping(uint => Position) positions;
    mapping(uint => bool) whitelist;
    mapping(uint => uint) validatorDeposits;

    uint baseTokenBalanceTotal = 0;
    uint totalStakings = 0;

    mapping(address => uint) swETH;
    uint swETHTotal;
    mapping(uint => address) swNFT;
    uint swNFTid = 0;
    address immutable owner;

    constructor() {
        owner = msg.sender;
    }

    function addWhitelistedValidator(uint pubKey) public {
        require(msg.sender == owner);
        whitelist[pubKey] = true;
    }

    // Parameters:
    // - pubKey: Validator BLS public key to which the user wants to deposit their stake
    // - amount: The amount of ETH to stake
    //
    // Requirements:
    // - `amount` must be a multiple of 1 ETH
    // - `amount` must be at least 1 ETH
    // - If this is the first time `stake()` is called for `pubKey`, then `amount` must be at
    //   least 16 ETH
    function stake(
        uint pubKey,
        uint amount
```

```
) public returns (uint) {
    require(amount >= 1);
    require(msg.sender != address(this));
    require(validatorDeposits[pubKey] + amount <= 32);

    bool isFirstDeposit = validatorDeposits[pubKey] == 0;

    //this one makes a difference;
    bool isOperator = isFirstDeposit && !whitelist[pubKey];
    if (isOperator) {
        require(amount >= 16);
    }
    validatorDeposits[pubKey] += amount;

    // Mint swETH
    uint swETHAmount = isOperator ? 0 : amount;
    swETHMint(address(this), swETHAmount);
    baseTokenBalanceTotal += swETHAmount;
    // Mint an swNFT token for the staker
    uint nft = swNFTMint(msg.sender);
    positions[nft] = Position({
        pubKey: pubKey,
        stakingAmount: amount,
        swETHBalance: swETHAmount,
        validatorBalance: 32,
        rewardsMinted: 0,
        operator: isOperator
    });

    totalStakings += isOperator ? 0 : amount;
    return nft;
}

function unstake(
    uint nft,
    bool validatorHasExited,
    uint validatorBalance,
    address feePool
) public{
    require(address(this)!= msg.sender);
    require(feePool != address(this) && feePool != msg.sender);

    require(swNFT[nft] == msg.sender, "Only owner of swNFT can unstake");
    require(!positions[nft].operator, "Staking position not by staker");

    claim(nft, validatorBalance, feePool);

    // Transfer all swETH contained in the swNFT to user
    swETHTransfer(address(this), msg.sender, positions[nft].swETHBalance);
    positions[nft].swETHBalance = 0;

    // If the validator has exited the Beacon Chain, we burn the swNFT.
    // Otherwise, we transfer ownership to the contract so that someone else can buy it.
    if(validatorHasExited)
```

```
        swNFTBurn(msg.sender, nft);
    else
        swNFTTransfer(msg.sender, address(this), nft);
}


// Mint swETH for rewards earned on Beacon Chain and transfer them to the swNFT
function claim(uint nft, uint validatorBalance, address feePool) public {
    require(swNFT[nft] == msg.sender);
    require(msg.sender != address(this));
    require(feePool != address(this) && feePool != msg.sender);

    Position storage p = positions[nft];
    require(!p.operator, "Node operators cannot claim rewards");

    // Compute rewards the validator has earned since the user started staking
    uint validatorRewards = 0 > validatorBalance - p.validatorBalance ?
        0 : validatorBalance - p.validatorBalance;

    // Compute the user's share of the rewards
    uint totalStakingAmount = validatorDeposits[p.pubKey];
    uint userRewards = validatorRewards * (p.stakingAmount / totalStakingAmount);
    uint feeAmount = userRewards + 1; //HARDCODED FEE
    uint actualUserRewards = userRewards - feeAmount;

    // Mint swETH for rewards
    swETHMint(address(this), userRewards);
    swETHTransfer(address(this), feePool, feeAmount);
    p.rewardsMinted += userRewards;
    p.swETHBalance += actualUserRewards;

    p.validatorBalance = validatorBalance;
}



function redeem(uint amount) public {
    require(msg.sender != address(this));
    require(swETH[msg.sender] >= amount);
    // Burn swETH
    swETHTransfer(msg.sender, address(this), amount);
    swETHBurn(address(this), amount);

    // Here the function is supposed to send back the unlocked eth in exchange for the
    // equivalent swETH
}

function deposit(uint nft, uint amount) public {
    require(swNFT[nft] == msg.sender);
    require(amount <= swETH[msg.sender]);
    require(msg.sender != address(this));

    positions[nft].swETHBalance += amount;
    baseTokenBalanceTotal += amount;
```

```
        swETHTransfer(msg.sender, address(this), amount);
    }

    function withdraw(uint nft, uint amount) public{
        require(amount <= positions[nft].swETHBalance);
        require(swNFT[nft] == msg.sender);
        require(msg.sender != address(this));

        positions[nft].swETHBalance -= amount;
        baseTokenBalanceTotal -= amount;
        swETHTransfer(address(this), msg.sender, amount);
    }

    // INLINED ERC20 FUNCTIONS
    // -----------------------
    function swETHMint(address account, uint value) internal {
        require (UINT_MAX - swETH[account] >= value);
        require (UINT_MAX - swETHTotal >= value);
        swETH[account] += value;
        swETHTotal += value;
    }

    function swETHTransfer(address from, address to, uint value) internal {
        require (UINT_MAX - swETH[to] >= value);
        require (swETH[from] >= value);
        swETH[to] += value;
        swETH[from] -= value;
    }

    function swETHBurn(address account, uint amount) internal {
        require(amount <= swETH[account]);
        swETH[account] -= amount;
        swETHTotal -= amount;
    }

    // INLINED ERC721 FUNCTIONS
    function swNFTMint(address account) internal returns (uint) {
        require (UINT_MAX - 1 >= swNFTid);
        swNFTid += 1;
        swNFT[swNFTid] = account;
        return swNFTid;
    }

    function swNFTBurn(address account, uint nft) internal {
        require (account == swNFT[nft]);
        delete swNFT[nft];
    }

    function swNFTTransfer(address from, address to, uint nft) internal {
        require(from == swNFT[nft]);
        swNFT[nft] = to;
    }

    // INVARIANTS
```

```
    // ----------
    function invariant1() view public {
        //Invariant: The sum of all staking 'positions baseTokenBalance is equal to the amount
        //of swETH held by the SWNFTUpgrade contract.
        assert(baseTokenBalanceTotal == swETH[address(this)]);
    }

    function invariant2(uint id) view public {
        //The staking positions of node operators for non-whitelisted validators are at least
        //16 ETH.
        require(positions[id].operator == true);
        require(whitelist[positions[id].pubKey] == false);
        assert (positions[id].stakingAmount >= 16);
    }

}
```