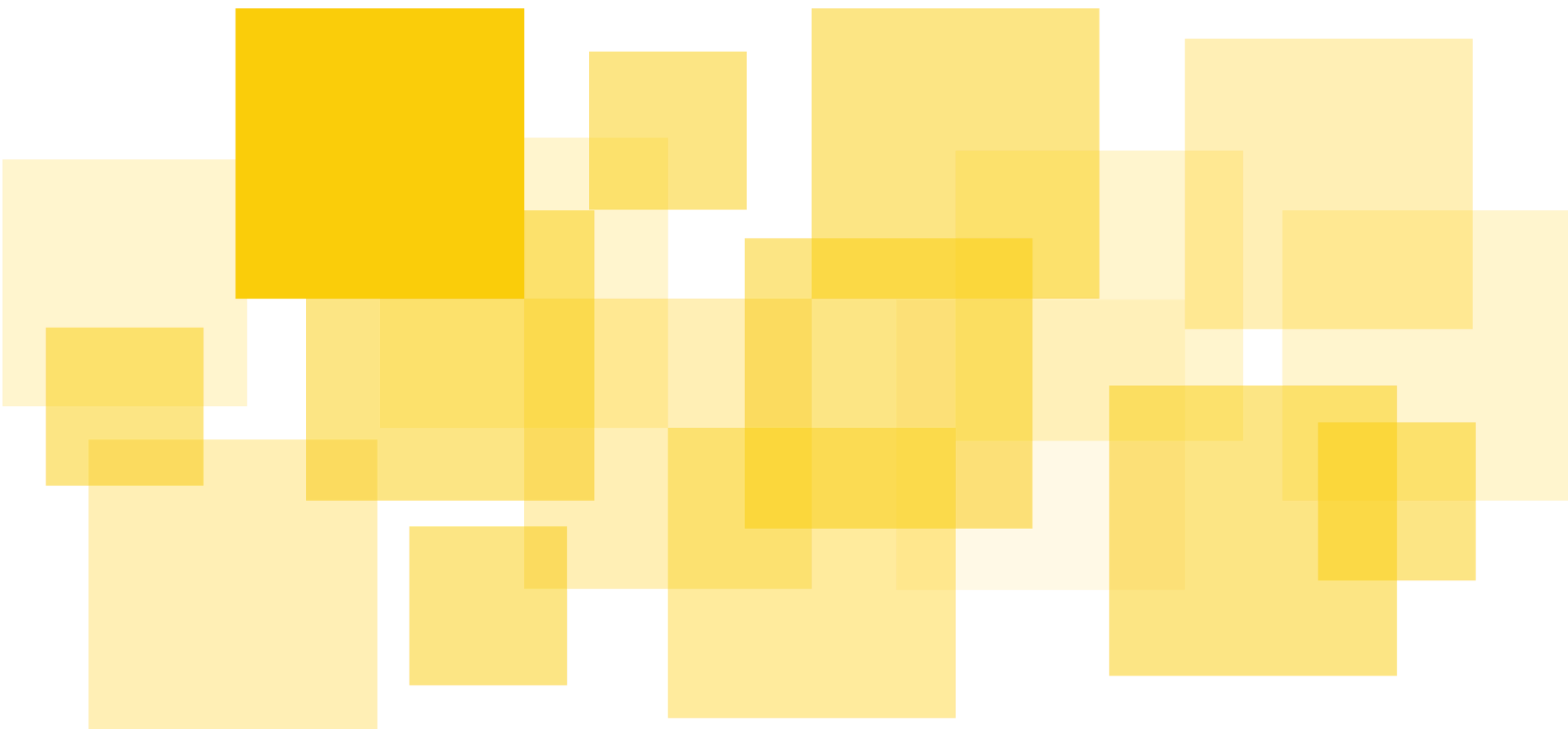


Design Review

EigenLayer

Delivered: Dec 5, 2022



Prepared for LayrLabs by Runtime Verification, Inc.



[Summary](#)

[Disclaimer](#)

[Protocol Description](#)

[Actors](#)

[Operations](#)

[Withdrawals](#)

[Assumptions](#)

[Findings](#)

[A01: Governance may be able to slash unfrozen operator](#)

[Scenario](#)

[Status](#)

[Informative findings](#)

[B01: Partial slashing is not directly supported](#)

[Status](#)

[B02: Stakers can only delegate to a single operator at a time](#)

[Status](#)

[B03: Arbitrary IDelegationTerms contracts difficult to verify](#)

[Status](#)

[B04: Combine recordLastStakeUpdate\(\) and revokeSlashingAbility\(\)](#)

[Status](#)

[B05: Nonce passed to queueWithdrawal\(\) is unnecessary](#)

[Status](#)

[B06: Storage variable delegationStatus is unnecessary](#)

[Status](#)

[B07: Unused constants may be removed](#)

[Status](#)

[B09: Separate Callbacks for Distinct Functionality](#)

[Status](#)

[B10: Using Custom Errors](#)

[Status](#)

[Appendix 1: Notation](#)

[Appendix 2: Abstract Model](#)

[EigenLayer](#)

[deposit\(\)](#)

[registerAsOperator\(\)](#)

[delegateTo\(\)](#)

[undelegate\(\)](#)

[optIntoSlashing\(\)](#)

[queueWithdrawal\(\)](#)

[completeQueuedWithdrawal\(\)](#)

[recordStakeUpdate\(\)](#)

[revokeSlashingAbility\(\)](#)

[freezeOperator\(\)](#)

[slashShares\(\)](#)

[slashQueuedWithdrawal\(\)](#)

[resetFrozenStatus\(\)](#)

[Middleware](#)

[register\(\)](#)

[performOperatorAction\(\)](#)

[prepareWithdrawal\(\)](#)



[prepareExit\(\)](#)

[slash\(\)](#)

[Appendix 3: Invariants](#)

[Main Invariant](#)

[Invariant 1](#)

[Invariant 2](#)

[Invariant 3](#)

[Invariant 4](#)

[Invariant 5](#)

[Invariant 6](#)

[Invariant 7](#)

[Invariant 8](#)

[Invariant 9](#)

[Invariant 10](#)

Summary

[Runtime Verification, Inc.](#) has reviewed the design of multiple contracts that comprise the core of the EigenLayer protocol. The review was conducted from 2022-11-21 to 2022-12-02.

LayrLabs engaged Runtime Verification in checking the security of their EigenLayer protocol, which allows operators to stake assets that are already in use in other protocols. By reusing (or *restaking*) these existing assets, operators can stake to new services without needing additional capital. Examples of assets that can be restaked include ETH that has been deposited to a validator on Ethereum, or tokens deposited to an Aave lending pool.

The issues which have been identified can be found in section [Findings](#). A number of additional suggestions have also been made, and can be found in section [Informative findings](#). We also created an abstract model of the EigenLayer protocol (see [Appendix 2: Abstract Model](#)) and used it to formulate important invariants (see [Appendix 3: Invariants](#)).

Scope

The design review was based on the following contracts:


- EigenLayrDelegationStorage
- EigenLayrDelegation
- InvestmentManagerStorage
- InvestmentManager
- Slasher
- Pausable
- PauserRegistry

These contracts are part of a private repository that was frozen for review at commit 3a6d64e746a36a5d42156c84cfbc66c2d67ed322.

Since this was a design review and not a code review, we mostly focused on the business logic implemented by the above contracts and not on the code itself. This means we did *not* thoroughly examine the code for potential implementation errors.

Methodology

Although the manual design 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. Second, we discussed the most catastrophic outcomes with the team, and reasoned backwards



from their places in the design to ensure that they are not reachable in any unintended way. Finally, we regularly participated in meetings with the LayrLabs 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. 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.

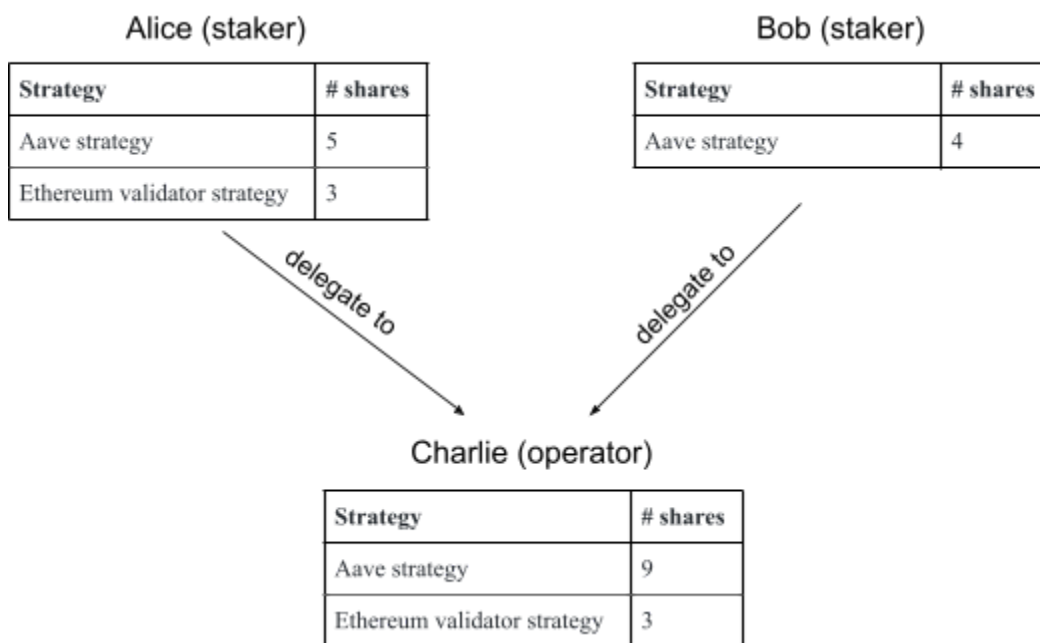
This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

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.

Protocol Description

EigenLayer allows anyone to stake assets that are already in use in other protocols, enabling operators to provide validation services for new middlewares without needing additional capital. Here, *middleware* refers to any service that requires staking to secure some part of its protocol, such as bridges or rollups. The process of reusing existing assets as stake is called *restaking*.

In EigenLayer, restaked assets are represented as *shares* of an *investment strategy* (*strategy* for short), which is a contract implementing the `IInvestmentStrategy` interface. For example, by using EigenLayer's `AaveInvestmentStrategy`, users can deposit tokens to an Aave lending pool, in return for which they receive shares that represent the deposited tokens. These shares can then be delegated to an operator as stake:



Here we have two stakers, Alice and Bob, who own shares from multiple strategies. Alice owns 5 shares from the Aave strategy, and an additional 3 shares from the Ethereum validator strategy, which can be used to restake ETH that has been deposited to a validator on Ethereum. Both Alice and Bob delegate their shares to Charlie, who acts as an operator. The total stake available to Charlie is the sum of all shares that have been delegated to him. Using this stake, Charlie can then provide validation services to any middleware that makes use of EigenLayer.

To provide his services to a particular middleware, Charlie needs to allow that middleware to slash his shares. This is a manual process because middlewares can define arbitrary slashing conditions, so before opting in to any particular middleware, operators should make sure that they are comfortable with these conditions. Once Charlie has opted into some middleware, he

can start to perform validation tasks, like for example casting a vote or proposing some data to be included on-chain (the concrete task depends on the middleware). If Charlie performs these tasks correctly, he earns staking rewards from the middleware that he then distributes among himself and all the delegators (this is handled mostly outside of EigenLayer). On the other hand, if Charlie performs a task incorrectly, the middleware may freeze his stake, and EigenLayer's governance may then slash his and all the delegator's shares.

If Alice or Bob no longer want to stake their shares, they can queue a withdrawal request. However, the withdrawal is not processed immediately because middlewares may still need to slash the shares. Thus, withdrawal requests can only be completed once all affected middlewares have given their permission.

After this brief overview, we now take a closer look at the actors and operations involved.

Actors

The following actors can be distinguished:

- **Stakers** hold shares that can be delegated to operators as stake.
- **Operators** perform validation tasks for middlewares. They use the shares that have been delegated to them as stake, and in return they give each delegator a cut of the rewards. Operators can also be stakers.
- **Middlewares** are services that require staking to secure some part of their protocol.
- **Governance** stands between middlewares and operators and is responsible for actually slashing an operator after a middleware has frozen its stake. Furthermore, governance can unfreeze an operator if it deems that freezing it was unjustified.

Operations

The following operations can be performed by **stakers** and/or **operators**:

- **Deposit** (`InvestmentManager.depositIntoStrategy()`).
Allows anyone to become a staker by depositing tokens into a strategy of their choice, in return for which they receive a corresponding amount of shares.
- **Delegate** (`EigenLayrDelegation.{delegateTo(), delegateToBySignature()})`.
Allows stakers to delegate their shares to an operator. A staker can only delegate to one operator at a time, and they can only delegate *all* their shares at once. While a staker has delegated their shares, all further deposits are immediately delegated as well.

- **Register as operator** (`EigenLayerDelegation.registerAsOperator()`).
Allows anyone to register as an operator. The only information that must be provided is the address that should receive the staking rewards. This can be either a contract implementing the `IDelegationTerms` interface that automatically gives the operator and each delegator a cut of the rewards, or it can be a EOA, in which case the rewards are distributed in a trusted manner off-chain.
- **Opt into slashing** (`Slasher.optIntoSlashing()`).
Allows operators to opt into a particular middleware in order to perform validation tasks and to receive rewards. By calling this function, the operator explicitly allows the middleware to freeze their staked shares (including the shares of all delegators).
- **Withdrawal**
(`InvestmentManager.{queueWithdrawal(),completeQueuedWithdrawal()}`).
Allows stakers to withdraw their (potentially delegated) shares to a receiver of their choice. A withdrawal request is initiated with `queueWithdrawal()`, after which the staker has to wait until all middlewares that have accepted the shares as stake have given their okay. Then, the designated receiver can complete the withdrawal by calling `completeQueuedWithdrawal()`, choosing whether they want to receive the withdrawn shares directly or as underlying tokens of the strategy. See section [Withdrawals](#) for more information.
- **Undelegate** (`InvestmentManager.undelegate()`).
A staker can only undelegate from an operator if they do not own any shares anymore. This means that a staker needs to withdraw all their shares before being able to undelegate.

The following operations can be performed by **middlewares**:

- **Record a stake update** (`Slasher.{recordFirstStakeUpdate(),recordStakeUpdate(),recordLastStakeUpdate()}`).
By calling one of these functions, a middleware provides two pieces of information to EigenLayer: (1) The most recent block number at which the middleware retrieved the operator's stake, and (2) until which time the stake must remain slashable. See section [Withdrawals](#) for more information.
- **Freeze operator** (`Slasher.freezeOperator()`).
If a middleware detects a misbehaving operator, it can freeze the operator's staked shares (including the shares of all delegators). These shares then remain locked inside the EigenLayer protocol until governance decides to either slash or unfreeze the shares, depending on whether governance deemed the freezing by the middleware to be justified.

- **Revoke slashing ability** (`Slasher.revokeSlashingAbility()`).
If an operator no longer wants to perform validation tasks for a middleware, the operator needs to notify the middleware about this decision (outside of `EigenLayer`). Afterwards, the middleware should call `revokeSlashingAbility()` and provide a timestamp after which the operator can no longer be slashed by the middleware.

The following operations can be performed by **governance**:

- **Slash shares** (`InvestmentManager.{slashShares(), slashQueuedWithdrawal()}`).
Allows governance to slash the shares of operators that haven't been frozen by a middleware. The receiver of the slashed shares can be freely chosen.
- **Unfreeze operators** (`Slasher.resetFrozenStatus()`).
If governance did not deem the freezing of an operator justified, they can unfreeze them.

Finally, the following administrative operations are supported:

- **Pause operations.** Many of the above operations can be temporarily disabled.
- **Upgrade contracts.** All the contracts we examined are upgradeable.

At launch, the intention is to use a multisig to control access to these operations. However, for a future release the plan is to allow governance to control unpausing and upgrades. On the other hand, pausing will likely remain in control of a multisig that is majority-controlled by LayrLabs in order to quickly react to emergency situations.

We have modeled the above operations in [Appendix 2: Abstract Model](#).

Withdrawals

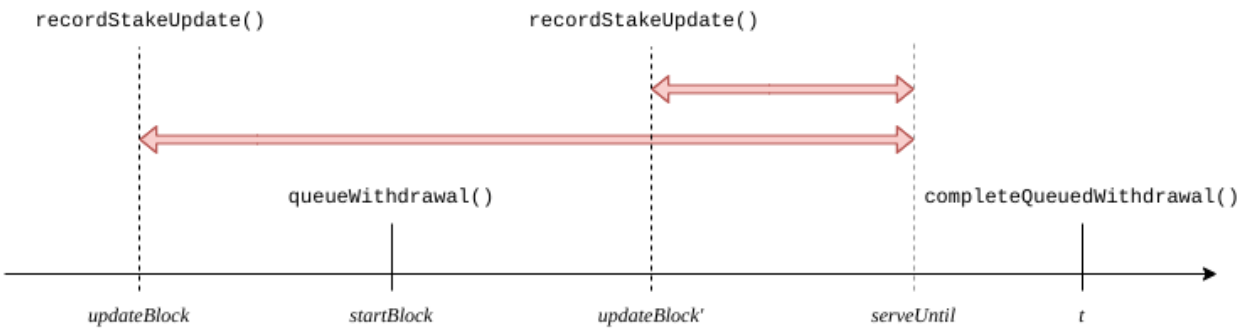
Withdrawals represent one of the more complex parts of `EigenLayer`, because it must be ensured that shares can only be withdrawn after they can no longer be slashed by any middleware. As an example, we consider the case where Alice wants to withdraw the shares that she has delegated to Charlie. For simplicity, we assume that Charlie has only opted into a single middleware. To complete the withdrawal process, the following steps need to be performed:

1. Alice needs to call `queueWithdrawal(...)` to announce that she wants to withdraw (some of) her shares. Let *startBlock* denote the block in which she makes this call.
2. Afterwards, the middleware needs to call `recordStakeUpdate(operator, updateBlock, serveUntil, ...)`. Here, *updateBlock* denotes the most recent block at which the middleware has fetched the current staking amount of the operator from `EigenLayer`, and *serveUntil* denotes the time until which that stake must remain slashable. The information about the pair (*updateBlock*, *serveUntil*) is stored in `EigenLayer`.

3. Now assume Alice tries to complete the withdrawal at time t by calling `completeQueuedWithdrawal(...)`. This is only successful if EigenLayer stores a pair $(updateBlock, serveUntil)$ such that
 - a) $startBlock < updateBlock$ and
 - b) $t > serveUntil$.

Condition a) ensures that the middleware has seen the queued withdrawal, and b) ensures that the middleware does not need to be able to slash the stake anymore. Thus, if both conditions are satisfied, Alice can safely withdraw her shares.

This withdrawal process can be illustrated as follows:



Here, we assume that the middleware has already called `recordStakeUpdate()` at some point in the past, so the pair $(updateBlock, serveUntil)$ has been stored in EigenLayer. Intuitively, it means that the middleware expects that it can slash the operator's full staking amount it retrieved at `updateBlock` until `serveUntil` (whether an operator needs to be slashed may only be known after a while, so it is important for middlewares to communicate how much time they need). Next, `queueWithdrawal()` is called in block `startBlock`. However, completing the withdrawal at this point would not be safe, as the middleware still expects to be able to slash the full staking amount. When `recordStakeUpdate()` is called again, EigenLayer updates its state to include the pair $(updateBlock', serveUntil)$. Finally, when `completeQueuedWithdrawal()` is called, both the conditions $startBlock < updateBlock'$ and $t > serveUntil$ are satisfied, and the withdrawal can be completed safely.

So far, we have described a simplified version of the withdrawal process. The actual implementation in EigenLayer makes the following changes:

Generalization to multiple middlewares. The simplified version described above only considers the case where the operator has opted into a *single* middleware. However, the existing approach can be easily adapted to support an arbitrary amount of middlewares. Conceptually, this is done by storing a pair $(updateBlock_i, serveUntil_i)$ for each middleware i , and then computing $updateBlock = \min_i updateBlock_i$ and

$$serveUntil = \max_i serveUntil_i$$

to check conditions a) and b) above.

Robustness in the presence of frequent updates. When computing *updateBlock* and *serveUntil* as the minimum/maximum like above, then it could be difficult for users who want to withdraw to actually satisfy condition b): $t > serveUntil$. This is because if middlewares update their $serveUntil_i$ frequently, then $serveUntil$ will always denote some time in the future, which means $t > serveUntil$ can never be satisfied and calling `completeQueuedWithdrawal()` always fails. The solution is to store a pair $(updateBlock_{t,i}, serveUntil_{t,i})$ for each time t at which middleware i calls `recordStakeUpdate()`. We further define

$$updateBlock_t = \min_i \max_{t' < t} updateBlock_{t',i}$$

$$serveUntil_t = \max_i \max_{t' < t} serveUntil_{t',i}$$

Then, we can adapt conditions a) and b) as follows: A call to `completeQueuedWithdrawal()` at time t for a withdrawal that was queued at block *startBlock* will succeed if there exists a t' such that $t' < t$ and the following conditions hold:

a') $startBlock < updateBlock_{t'}$, and

b') $t > serveUntil_{t'}$,

Assumptions


EigenLayer integrates many different actors whose behavior is largely out of its control. The safety of the protocol therefore depends on the assumption that these actors behave in certain ways.

Assumptions about middlewares:

- Stakers and operators need to trust that middlewares freeze operators only when the operator committed a slashable offense.

Alternatively, they need to trust that governance unfreezes their shares in the case a middleware has frozen their shares erroneously or maliciously

- Stakers and operators need to trust that middlewares pay out their staking rewards
- Stakers and operators need to trust that middlewares report updates to EigenLayer so that withdrawals can be completed



Depending on how a particular middleware is implemented, stakers and operators may be able to verify whether these assumptions are actually satisfied.

Assumptions about operators:

- Depending on the consensus mechanism that is used by a particular middleware, it may need to make assumptions about how many of the operators are honest
- Depending on how an operator chooses to distribute the staking rewards, stakers may need to trust that they actually receive their cut of the rewards
- Stakers need to trust that the operator they delegated to only opts into honest middlewares (see *Assumptions about middlewares*)

Assumptions about government:

- Middlewares need to trust that governance does not simply unfreeze frozen operators

Assumptions about investment strategies:

- Middlewares need to trust that the shares from the strategies that they accept as stake remain sufficiently valuable (depends on risk taken by the strategies)

In general, users of EigenLayer are expected to perform due diligence before when using strategies:

- Stakers must make sure to only deposit into strategies that they trust
- Middlewares must make sure to only accept shares from trusted strategies as stake

The client plans to indicate in the front-end how trustworthy each strategy is.

Findings

AO1: Governance may be able to slash unfrozen operator

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

Generally, operators are expected to stake to themselves. However, if an operator has not staked anything, they can undelegate from themselves and then delegate to a different operator. In such a situation where operator A has delegated to operator B, slashing operator A's stake also affects operator B, even if B has not been frozen and has done nothing wrong. This is clearly undesirable.

Note that only governance can slash someone's stake. Thus, if governance behaves honestly and detects the above situation before slashing someone, then this is not a problem in practice. However, the design of the EigenLayer protocol implies that governance should only be able to slash frozen operators, so it is likely that no one will look out for the situation described above, which can lead to the wrong operator being slashed.


Scenario

In EigenLayer, the same account can both be a staker and an operator. In the following scenario, we use `AliceTheStaker` and `AliceTheOperator` to differentiate between these two possible roles of Alice, but keep in mind that both refer to the same address.

1. `AliceTheStaker` has not staked anything yet and calls `registerAsOperator()`, which registers `AliceTheOperator`. Afterwards, `AliceTheStake` is delegated to `AliceTheOperator`
2. `AliceTheStaker` undelegates from `AliceTheOperator` and then delegates to `BobTheOperator`
3. Some middleware freezes `AliceTheOperator`
4. Since `AliceTheOperator` and `AliceTheStaker` share the same address, this means that `AliceTheStaker` is also frozen
5. Since `AliceTheStaker` is frozen, she can be slashed. However, because she has delegated to `BobTheOperator`, the shares will be slashed from `BobTheOperator` instead of `AliceTheOperator`

Status

Acknowledged. The client intends to address this issue by not allowing operators to undelegate from themselves. (This means that in the above scenario, step 2 is prevented.)



Note that since operators can never completely deregister from EigenLayer, this means that once someone registers as an operator, they will *never* be able to delegate to someone else with that account. They can only do so from a different account.

Informative findings

Bo1: Partial slashing is not directly supported

Slashing is a two-step process: First, a middleware freezes the stake of an operator, and then governance actually slashes the staked shares. At the moment, there is no way for the middleware to communicate how much of the stake should be slashed; the intention is that governance will slash *all* the shares staked to the operator.

However, middlewares may want to differentiate between different kinds of slashable offenses, and apply different penalties in each case. For example, on the Ethereum Beacon Chain, a validator's final penalty depends on how many other validators committed slashable offenses around the same time. Since EigenLayer aims to provide a general staking infrastructure, it may make sense to be flexible enough to support a wide range of use cases.

A possible implementation would allow middlewares to specify the slashing amount when freezing an operator. The shares of anyone who has delegated to the affected operator would then be slashed in proportion to their staking amount.

Status

The client is fully aware of this and has already been looking into possible approaches to support partial slashings. The reason it is currently not implemented is to reduce code complexity and lower gas cost, though they consider implementing partial slashings in a future version.



Bo2: Stakers can only delegate to a single operator at a time

When a staker delegates their shares to an operator, *all* their shares are delegated. There is no way to delegate only a partial amount of shares. This means that stakers who wish to stake to multiple operators need to use separate Ethereum accounts to do so, which may be inconvenient.

An alternative would be to allow each staker to have multiple *staking positions*. The staker could then deposit shares to their staking positions and delegate each position to a separate operator. This would make it easier for stakers with a lot of capital to support many operators.

Status

The client is aware of this restriction and initially supported delegation to multiple operators. This feature was then removed to simplify the protocol, though they may reintroduce it in a future release.



Bo3: Arbitrary IDelegationTerms contracts difficult to verify

Operators can specify an arbitrary contract implementing the `IDelegationTerms` interface to distribute staking rewards among the delegators. On the one hand, this provides a lot of flexibility, but it also makes it difficult for delegators to ensure that a concrete implementation of `IDelegationTerms` is trustworthy and actually gives each delegator a fair cut of the rewards.

Status

The client is aware of this and provides a default implementation for `IDelegationTerms` in the form of `MerkleDelegationTerms`. Operators that use this default contract can be marked as trustworthy in the front-end, making it safer for stakers to choose an operator.

Bo4: Combine `recordLastStakeUpdate()` and `revokeSlashingAbility()`

The functions `Slasher.recordLastStakeUpdate()` and `Slasher.revokeSlashingAbility()` both need to be called by middlewares when deregistering an operator. In order to simplify the protocol and to reduce gas costs, these functions can be combined into a single function.

Additionally, `recordLastStakeUpdate()` has a parameter called `serveUntil` that denotes the timestamp until which the operator's stake must remain slashable, and `revokeSlashingAbility()` has a parameter called `bondedUntil` that denotes the timestamp until which the operator's stake can be slashed. Note that in general we would expect that `serveUntil == bondedUntil`, because otherwise a middleware may be unable to slash misbehaving operators. By combining these functions, it can be ensured that this condition always holds.

Status

Addressed.

Bo5: Nonce passed to `queueWithdrawal()` is unnecessary

The function `InvestmentManager.queueWithdrawal()` has a parameter of type `WithdrawerAndNonce`, which contains a nonce. This nonce is then checked against the value of `numWithdrawalsQueued[msg.sender]`:

```
require(
    withdrawerAndNonce.nonce == numWithdrawalsQueued[msg.sender],
    "InvestmentManager.queueWithdrawal: provided nonce incorrect"
);
```

Afterwards, `withdrawerAndNonce` is stored inside an instance of `QueuedWithdrawal`, whose hash is stored inside the protocol:

```
QueuedWithdrawal memory queuedWithdrawal = QueuedWithdrawal({
    strategies: strategies,
    tokens: tokens,
    shares: shares,
    depositor: msg.sender,
    withdrawerAndNonce: withdrawerAndNonce,
    withdrawalStartBlock: uint32(block.number),
    delegatedAddress: delegatedAddress
});

// calculate the withdrawal root
bytes32 withdrawalRoot = calculateWithdrawalRoot(queuedWithdrawal);

// mark withdrawal as pending
withdrawalRootPending[withdrawalRoot] = true;
```

Using a nonce in this way has no effect. Instead of passing in the nonce via a function argument, it may as well directly be read from `numWithdrawalsQueued[msg.sender]`.

Status

Addressed.

Bo6: Storage variable delegationStatus is unnecessary

The storage variable `EigenLayrDelegationStorage.delegationStatus` does not provide any information that is not already provided by `EigenLayrDelegationStorage.delegatedTo`:

- `delegationStatus[addr] == UNDELEGATED` if and only if `delegatedTo[addr] == address(0)`
- `delegationStatus[addr] == DELEGATED` if and only if `delegatedTo[addr] != address(0)`

Unless the intention is to add more states to `DelegationStatus` it may make sense to remove `delegationStatus`.

Status

Addressed.

Bo7: Unused constants may be removed

The following constants are unused and may therefore be removed.

- `InvestmentManagerStorage.WITHDRAWAL_WAITING_PERIOD`
- `InvestmentManagerStorage.REASONABLE_STAKES_UPDATE_PERIOD`
- `InvestmentManager.QUEUED_WITHDRAWAL_INITIALIZED_VALUE`

Status

Addressed.



Bo9: Separate Callbacks for Distinct Functionality

The same hook is used for both `_delegate` and `increaseDelegatedShares` functions. It would be more flexible to provide two separate hooks for distinct functionalities.

Status

The client is aware of this and the situation is going to be addressed in future iterations.



B10: Using Custom Errors

Currently the `require` and `revert` statements use custom strings to provide information about the error in the contract. Custom errors may be designed and integrated to current design for a more efficient gas usage and well-defined error handling.

Status

The client is aware of this and the situation is going to be addressed in future iterations.

Appendix 1: Notation

The abstract models we define in [Appendix 2: Abstract Model](#) use a pseudo code based on Solidity. Here we present the notation and conventions used in this pseudo code.

Arrays

- `[e1, e2, ..., e3]` denotes an *array literal*: It constructs a new array containing the given elements.
- We use `++` to denote array concatenation. That is, if `arr1` and `arr2` are two Solidity arrays of the same type, then `arr1 ++ arr2` denotes the array that first contains all the elements of `arr1` followed by all the elements of `arr2`
- Whether an array contains a certain element can be tested with \in . If `arr` denotes a Solidity array, then $v \in arr$ is true if and only if there exists some $i < arr.length$ such that $v == arr[i]$.
- For an array `arr`, we use `sum(arr)` to sum up all the elements in `arr`. (Requires that `+` is defined for the element type of `arr`.)

Sets

- We introduce a type for sets: If `TYPE` denotes a Solidity type, then `set(TYPE)` denotes an unordered set of elements of type `TYPE`.
- `{e1, e2, ..., en}` denotes a *set literal*: It constructs a new set that contains the given elements.
- `{ e \in S | P(e) }` is used to construct a set containing all elements `e` of `S` that satisfy condition `P(e)`.
- We define the usual operations on sets: \cup (union), \cap (intersection), \setminus (set difference), \in (membership).
- If `s` denotes a set, we use `s.insert(v)` to insert `v` into the `s`. If `s` already contains `v`, i.e., if $v \in s$, then this operation has no effect. Furthermore, we use `s.remove(v)` to remove `v` from `s`. If $v \notin s$, then this operation has no effect.
- For a set `s`, we use `sum(s)` to sum up all the elements in `s`. (Requires that `+` is defined for the element type of `s`.)

Mappings

- `{key1 => val1, key2 => val2, ..., keyn => valn}` denotes a *mapping literal*: It constructs a new Solidity mapping that maps the given keys to the corresponding values.
- If `m` denotes a Solidity mapping, then we use `m.keys()` to denote the set of all keys that have explicitly been mapped to a value. Furthermore, we use `m.values()` to denote the set of all values that have explicitly been set of a key in `m`.
- We extend the definition of operators like `+`, `-`, `<`, `==` to mappings in an element-wise fashion. That is, if `m1` and `m2` denote two Solidity mappings of the same type, then `m1 + m2` denotes a new mapping `m` such that for each $k \in m1.keys() \cup m2.keys()$ the condition $m[k] == m1[k] + m2[k]$ holds. (If `k` is not a key in `m1` or `m2`, then like in Solidity, the default value is used.) Furthermore, a comparisons like `m1 < m2` is true if and only if for any $k \in m1.keys() \cup m2.keys()$ the conditions $m1[k] < m2[k]$ holds.

Numbers

- We use the type `number` to denote (mathematical) real numbers. This means calculations using values of this type are not affected by rounding or overflow/underflow.

Expressions

- `MIN` and `MAX`: For example, the expression `MIN(i) { arr[i] | i < arr.length }` can be used to retrieve the minimum value of an array. Here, we range over variable `i`, and the result of the expression is the minimum value of `arr[i]` for which the condition `i < arr.length` is satisfied. If the condition is never satisfied, the result is undefined.

Statements

- *Iteration over sets*: For a set `s` of type `set(TYPE)`, we use `for(TYPE e in s) { ... }` to iterate over the elements of `s` (order is undefined).
- *Iteration over mappings*: For a mapping `m` of type `mapping(T1 => T2)`, we use `for((T1 key, T2 val) in m) { ... }` to iterate over the elements of `s` (order is undefined). Note that only keys in `m.keys()` are considered.

As a simplification, if a local variable or function parameter that has a reference type (e.g., structs, arrays) is declared without a `storage` or `memory` specifier, we treat it like a value type that is copied on assignment.

Finally, we also allow functions to be defined inside structs.

Appendix 2: Abstract Model

In this section we define an abstract model of the EigenLayer protocol and of a hypothetical middleware. Our models make the following simplifications:

- Calculations are performed using real arithmetic; no rounding or underflow/overflow is considered
- Beacon Chain staking is not supported
- Globally permissioned contracts are not supported
- Protocol cannot be paused
- No upper limit on the number of strategies that a staker deposit into
- No protection against re-entrancy
- We do not model the part of the middleware that pays out operator rewards

EigenLayer

The EigenLayer model combines the functionality of the InvestmentManager, EigenLayrDelegation and Slasher contracts. It makes use of the following types and structs:

```
type StakingAmount = mapping(IInvestmentStrategy => number);

struct StakerInfo {
    // The number of shares that have been staked
    StakingAmount shares;

    // The operator to which the staker has delegated their shares.
    // address(0) is used if the staker has not yet delegated.
    address delegatedTo;
}

// A slashing window specifies that the operator's stake at block `updateBlock`
// must remain slashable until `stakeNeededUntil`.
// Roughly corresponds to the MiddlewareTimes struct in the implementation, but
// only holds information for a single middleware.
struct SlashingWindow {
    BlockNumber updateBlock;
    Timestamp stakeNeededUntil; // Corresponds to serveUntil
}

struct OperatorInfo {
    // Contract that receives the validator rewards earned by the operator.
    // These rewards are then usually distributed among the delegators
}
```

```

IDelegationTerms terms;

// The total amount of shares that have been delegated to the operator
StakingAmount stakedShares;

// Middlewares can freeze an operator if the operator commits a slashable
// offense
bool frozen;

// Time before which the middlewares are allowed to freeze (aka slash) the
// operator
mapping(IMiddleware => Timestamp) stakeSlashableBefore;

// Keep track of all slashing windows reported via recordStakeUpdate()
mapping(IMiddleware => SlashingWindow[]) slashingWindows;

// Compute a single SlashingWindow based on the SlashingWindows of all the
// middlewares that the operator has opted into.
function slashingWindowAt(uint idx) {
    return SlashingWindow({
        // Compute stalest updateBlock among all middlewares
        updateBlock: MIN(m) {
            updateBlockFor(m, idx) | m ∈ slashingWindows.keys()
        },

        // Compute latest stakeNeededUntil among all middlewares
        stakeNeededUntil: MAX(m) {
            stakeNeededUntilFor(m, idx) | m ∈ slashingWindows.keys()
        };
    });
}

// Compute the most recent updateBlock for a middleware, up to a specific
// index into the SlashingWindow array
function updateBlockFor(IMiddleware m, uint idx) {
    uint upperBound = min(slashingWindows[m].length, idx);
    return MAX(i) {slashingWindows[m][i].updateBlock | i <= upperBound};
}

// Compute the latest stakeNeededUntilFor for a middleware, up to a specific
// index into the SlashingWindow array
function stakeNeededUntilFor(IMiddleware m, uint idx) {
    uint upperBound = min(slashingWindows[m].length, idx);
    return MAX(i) {slashingWindows[m][i].stakeNeededUntil | i <= upperBound};
}
}

```

```

struct QueuedWithdrawal {
    // Staker from which the shares should be withdrawn
    address staker;

    // User who will receive the withdrawn shares
    address receiver;

    // Which shares to to withdraw
    StakingAmount sharesToWithdraw;

    // The operator that `staker` delegated to at the time the withdrawal was
    // queued. (By the time the withdrawal is completed, `staker` may already
    // have undelegated and/or delegated to a different operator)
    address operator;

    // A per-staker unique nonce
    uint nonce;

    // The block.number in which the withdrawal was queued
    BlockNumber startBlock;

    // NOTE: This is a ghost variable which is only needed for proofs.
    uint withdrawIdx;
}

```

The EigenLayer model is defined as follows:

```

contract EigenLayer {
    mapping(address => StakerInfo) stakers;
    mapping(address => OperatorInfo) operators;

    // Data needed for withdrawals
    mapping(address => uint) nonces; // staker => nonces
    set(QueuedWithdrawal) queuedWithdrawals;

    // Operator => Deposited shares
    // Mapping that stores for each operator the deposits that have been made to
    // it.
    // NOTE: This is a ghost variable which is only needed for proofs.
    mapping(address => StakingAmount[]) deposits;

    // NOTE: This is a ghost variable which is only needed for proofs.
    mapping(address => uint) withdrawalCount; // operator => number of withdrawals
}

```

```

// We assume that at the end of each external function, this variable is
// automatically set to the block.number in which the function was executed.
// NOTE: This is a ghost variable which is only needed for proofs.
BlockNumber lastBlockNumber;

// Functions are defined below...
}

```

As noted in the comments, ghost variables are used to keep track of additional information that is needed to formulate and prove certain invariants.

We define the following view functions for EigenLayer:

```

function canWithdraw(
    address operator,
    QueuedWithdrawal withdrawal,
    uint slashingWindowIdx
) {
    if(withdrawal ∉ queuedWithdrawals)
        return false;

    SlashingWindow s = operators[operator].slashingWindowAt(slashingWindowIdx);

    return withdrawal.startBlock < s.updateBlock
        && block.timestamp > s.stakeNeededUntil;
}

function canSlash(address middleware, address operator) {
    if(operator ∈ operators.keys()) {
        if(middleware ∈ operators[operator].stakeSlashableBefore.keys())
            return block.timestamp <
                operators[operator].stakeSlashableBefore[middleware];
    }

    return false;
}

function isFrozen(address staker) {
    if(staker ∈ operators.keys())
        return operators[staker].frozen;

    if(staker ∈ stakers.keys() && stakers[staker].delegatedTo != address(0)) {
        address operator = stakers[staker].delegatedTo;
        return operators[operator].frozen;
    }
}

```

```

        return false;
    }

    function sharesInWithdrawalQueue(address operator, uint withdrawIdx) {
        mapping(IInvestmentStrategy => number) shares;
        for(QueuedWithdrawal withdrawal in queuedWithdrawalsFor(operator)) {
            if(withdrawal.withdrawIdx >= withdrawIdx)
                shares += withdrawal.sharesToWithdraw; // Element-wise addition
        }

        return shares;
    }

    function totalDepositsAfter(address operator, uint depositIdx) {
        // Sum all deposits starting from `depositIdx`
        return sum(deposits[operator][depositIdx:]); // Element-wise addition
    }

    function queuedWithdrawalsFor(address operator) {
        return {w ∈ queuedWithdrawals | w.operator == operator};
    }

    function frozenOperators() {
        return {op ∈ operators.keys() | operators[op].frozen};
    }

```

In the following subsections, we take a look at all the state-modifying functions defined for the EigenLayer model.

deposit()

Preconditions:

- staker is msg.sender
- !isFrozen(staker)
- tokenAmount > 0

```

function deposit(
    address staker,
    IInvestmentStrategy strategy,
    number tokenAmount
) {
    // Stakers can deposit as often as they want. The first time they do it, we
    // need to create an entry for them

```

```

    if(staker ∉ stakers.keys()) {
        stakers[staker] = StakerInfo();
    }

    // Transfer `tokenAmount` of underlying tokens from the staker to the
    // strategy
    number shareAmount = strategy.transferTokensFrom(staker, tokenAmount);
    stakers[staker].shares[strategy] = shareAmount;

    // If the staker has already delegated to an operator, we delegate the new
    // shares as well
    if(stakers[staker].delegatedTo != address(0)) {
        address op = stakers[staker].delegatedTo;
        operators[op].stakedShares[strategy] += shareAmount;

        // Call into the operator's IDelegationTerms contract to update weights
        // of individual delegators
        operators[op].terms.onDelegationReceived(staker);

        // Update ghost variable
        deposits[op].push({strategy => shareAmount});
    }
}

```

registerAsOperator()

Preconditions:

- operator is msg.sender
- !isFrozen(operator)
- operator ∉ operators.keys()
- operator ∈ stakers.keys() ==> stakers[operators].delegatedTo = address(0)
- terms != address(0)

```

function registerAsOperator(address operator, IDelegationTerms terms) {
    if(operator ∉ stakers.keys()) {
        stakers[operator] = StakerInfo();
    }

    // Operators always delegate to themselves
    stakers[operator].delegatedTo = operator;

    operators[operator] = OperatorInfo({
        terms: terms,
        stakedShares: stakers[operator].shares
    });
}

```



```

});

operators[operator].terms.onDelegationReceived(operator);
deposits[op].push(stakers[operator].shares);
}

```

delegateTo()

Preconditions:

- staker is msg.sender
- !isFrozen(operator)
- staker ∈ stakers.keys() ==> stakers[staker].delegatedTo = address(0)
- operator ∈ operators.keys()

```

function delegateTo(address staker, address operator) {
    // The implementation allows stakers to delegate even before they have
    // called `deposit()`
    if(staker ∉ stakers.keys()) {
        stakers[staker] = StakerInfo();
    }

    // Add the shares of the staker to the operator
    operators[operator].stakedShares[strategy] += stakers[staker].shares;
    stakers[staker].delegatedTo = operator;

    operators[operator].terms.onDelegationReceived(staker);
    deposits[op].push(stakers[staker].shares);
}

```

undelegate()

Preconditions:

- staker is msg.sender
- staker ∈ stakers.keys()
- staker ∉ operators.keys() (see [Ao1](#))
- stakers[staker].shares == {} (empty mapping)

```

function undelegate(address staker) {
    stakers[staker].delegatedTo = address(0);
}

```

optIntoSlashing()

Preconditions:

- operator is msg.sender
- operator \in operators.keys()

```
function optIntoSlashing(address operator, IMiddleware middleware) {
    operators[operator].stakeSlashableBefore[middleware] = MAX_TIMESTAMP;
}
```

queueWithdrawal()

Preconditions:

- staker is msg.sender
- staker \in stakers.keys()
- !isFrozen(staker)
- for all strategies s: sharesToWithdraw[s] \leq stakers[staker].shares[s]

```
function queueWithdrawal(
    address staker,
    address receiver,
    mapping(IInvestmentStrategy => number) sharesToWithdraw,
    bool undelegateIfPossible
) {
    address operator = stakers[staker].delegatedTo;

    // Withdraw the shares from the operator that the staker has delegated to
    if(operator != address(0)) {
        operators[operator].stakedShares -= sharesToWithdraw;

        // Notify operator's DelegationTerms contract so that the reward
        // distribution among the delegators can be adjusted
        operators[operator].terms.onDelegationWithdrawn(staker);
    }

    // Now withdraw the shares from the staker itself
    stakers[staker].shares -= sharesToWithdraw;

    if(undelegateIfPossible && sum(stakers[staker].shares.values()) == 0) {
        undelegate(staker);
    }

    // Create a record for the withdrawal
    QueuedWithdrawal withdrawal = QueuedWithdrawal({
```

```

        staker: staker,
        receiver: receiver,
        sharesToWithdraw: sharesToWithdraw,
        operator: operator,
        nonce: nonces[staker]++,
        startBlock: block.number,
        withdrawIdx: withdrawalCount[operator]++
    });
    queuedWithdrawals.insert(withdrawal);

    return withdrawal;
}

```

completeQueuedWithdrawal()

Preconditions:

- receiver is msg.sender
- withdrawal \in queuedWithdrawals
- canWithdraw(withdrawal.operator, withdrawal, slashingWindowIdx)

```

function completeQueuedWithdrawal(
    address receiver,
    QueuedWithdrawal withdrawal,
    uint slashingWindowIdx,
    bool receiveAsTokens
) {
    queuedWithdrawals.remove(withdrawal);

    if(receiveAsTokens) {
        for((IInvestmentStrategy strategy, number shareAmount) in
            withdrawal.sharesToWithdraw
        ) {
            // Transfer `shareAmount` worth of underlying tokens to the receiver
            strategy.withdraw(receiver, shareAmount);
        }
    } else {
        if(receiver  $\notin$  stakers.keys()) {
            stakers[receiver] = StakerInfo();
        }

        stakers[receiver].shares += withdrawal.sharesToWithdraw;

        // If the receiver is delegated to an operator, forward any received
        // shares
    }
}

```

```

        if(stakers[receiver].delegatedTo != address(0)) {
            address operator = stakers[staker].delegatedTo;
            operators[operator].stakedShares += sharesToWithdraw;
            operators[operator].terms.onDelegationReceived(staker);
            deposits[op].push(sharesToWithdraw);
        }
    }
}

```

recordStakeUpdate()

Preconditions:

- middleware is msg.sender
- canSlash(middleware, operator)
- updateBlock <= block.number

```

function recordStakeUpdate(
    IMiddleware middleware,
    address operator,
    BlockNumber updateBlock,
    Timestamp stakeNeededUntil
) {
    operators[operator].slashingWindows[middleware].push(
        SlashingWindow(updateBlock, stakeNeededUntil)
    );
}

```

revokeSlashingAbility()

Preconditions:

- middleware is msg.sender

```

function revokeSlashingAbility(
    IMiddleware middleware,
    address operator,
    Timestamp bondedUntil
) {
    if(operators[operator].stakeSlashableBefore[middleware] == MAX_TIMESTAMP) {
        operators[operator].stakeSlashableBefore[middleware] = bondedUntil;

        // Makes all QueuedWithdrawals eligible for completion at time
        // `bondedUntil`
    }
}

```

```

        operators[operator].slashingWindows[middleware].push(
            SlashingWindow(block.number, bondedUntil)
        );
    }
}

```

freezeOperator()

Preconditions:

- middleware is msg.sender
- canSlash(middleware, operator)

```

function freezeOperator(IMiddleware middleware, address operator) {
    operators[operator].frozen = true;
}

```

slashShares()

Preconditions:

- Contract owner (governance) is msg.sender
- staker \in stakers.keys()
- isFrozen(staker)
- for all strategies s: sharesToSlash[s] \leq stakers[staker].shares[s]

```

function slashShares(
    address staker,
    address receiver,
    mapping(IInvestmentStrategy => number) sharesToSlash
) {
    // Slash the shares from the staker
    stakers[staker].shares -= sharesToSlash;

    // Transfer the slashed tokens to the receiver
    for((IInvestmentStrategy strategy, number shareAmount) in sharesToSlash) {
        strategy.withdraw(receiver, shareAmount);
    }

    // Update the stake of the operator accordingly
    address operator = stakers[staker].delegatedTo;
    operators[operator].stakedShares -= sharesToSlash;
    operators[operator].terms.onDelegationWithdrawn(staker);
}

```

slashQueuedWithdrawal()

Preconditions:

- Contract owner (governance) is `msg.sender`
- `withdrawal ∈ queuedWithdrawals`
- `isFrozen(withdrawal.operator)`

```
function slashQueuedWithdrawal(
    address receiver,
    QueuedWithdrawal withdrawal
) {
    queuedWithdrawals.remove(withdrawal);

    // Transfer the slashed tokens to the receiver
    for((IInvestmentStrategy strategy, number shareAmount) in
        withdrawal.sharesToWithdraw
    ) {
        strategy.withdraw(receiver, shareAmount);
    }
}
```

resetFrozenStatus()

Preconditions:

- Contract owner (governance) is `msg.sender`

```
function resetFrozenStatus(address operator) {
    if(operator ∈ operators.key()) {
        operators[operator].frozen = false;
    }
}
```

Middleware

Here we create a model for a very simple, hypothetical middleware. It makes use of the following structs:

```

// Information about a task that can be completed by an operator
struct TaskInfo {
    // Block in which the task was performed by the operator
    BlockNumber block;

    // The number of staked shares at the time the operator performed the task
    mapping(IInvestmentStrategy => number) stakedShares;

    // Time until which `stakedShares` must be available for slashing (i.e.,
    // withdrawals must only be possible after this time)
    Timestamp stakeNeededUntil;

    // Ghost variables
    uint withdrawIdx;
    uint depositIdx;
}

// Information that the middleware stores about each operator
struct Operator {
    // All the tasks performed by the operator
    TaskInfo[] tasks;

    // True if the middleware has caught the operator committing a slashable
    // offense
    bool slashed;

    // True when the operator no longer wants to perform any tasks
    bool exited;
}

```

The Middleware model is defined as follows:

```

contract Middleware {
    // Time needed by the middleware to decide whether an operator needs to be
    // slashed after they performed some task
    Duration constant TIME_FOR_SLASHING = ...;

    EigenLayer eigenLayer;
    mapping(address => Operator) operators;

    // We assume that at the end of each external function, these variables are
    // automatically updated to block.number and block.timestamp, respectively.
    // NOTE: These are ghost variables that are only needed for proofs.
    BlockNumber lastBlockNumber;
}

```

```

    Timestamp lastTimestamp;

    // Functions are defined below...
}

```

We define the following view functions for Middleware:

```

// Returns a pair (updateBlock, stakeNeededUntil), where
// - updateBlock denotes the block of the currently ongoing task, and
// - stakeNeededUntil denotes the timestamp until which the operator's stake
// at updateBlock must remain slashable.
//
// If there are tasks, then we simply return (block.number, 0), which makes
// all queued withdrawals immediately completeable.
//
// Requirements:
// - operator ∈ operators.keys()
function getSlashingWindow(address operator) {
    if(operators[operator].tasks.length == 0)
        return (block.number, 0);

    // Get the most recent task. This is only safe because in this model,
    // stakeNeededUntil of some task is larger than or equal to the
    // stakeNeededUntil of all previous tasks.
    TaskInfo lastTask = operators[operator].tasks[-1];

    return (lastTask.block, lastTask.stakeNeededUntil);
}

// Returns true if the stake provided by an operator is sufficiently high to
// participate
function isStakeSufficient(mapping(IInvestmentStrategy => number) stakedShares) {
    return ...;
}

function slashableOperators() {
    return {op ∈ operators.keys() | !operators[op].slashed};
}

function ongoingTasks(address operator) {
    return {task ∈ operators[operator].tasks |
        block.timestamp ≤ task.stakeNeededUntil};
}

```


In the following subsections, we take a look at all the state-modifying functions of the Middleware model.

register()

After an operator has called `EigenLayer.optIntoSlashing()` they need to call `Middleware.register()` to finalize the registration process.

Preconditions:

- operator is `msg.sender`
- if operator \in `operators.keys()` then
 - `operators[operator].exited`
 - `!operators[operator].slashed`

```
function register(address operator) {
    // Create new operator and fetch its current stake from EigenLayer
    operators[operator] = Operator();

    // This also ensures that the middleware can slash the operator
    eigenLayer.recordStakeUpdate(
        this,
        operator,
        block.number,
        0 // stakeNeededUntil
    );
}
```

performOperatorAction()

Called by an operator to complete a task, e.g., proposing some data or casting a vote.

Preconditions:

- operator is `msg.sender`
- operator \in `operators.keys()`
- `!operators[operator].slashed`
- `!operators[operator].exited`

```
function performOperatorAction(
    address operator,
    bytes payload /* whatever data is needed */
) {
    // Has the operator been frozen by another middleware?
    if(eigenLayer.operators[operator].frozen) {
```

```

        operators[operator].slashed = true;
        return;
    }

    // Get the current staking amount from EigenLayer
    mapping(IInvestmentStrategy => number) stakedShares =
        eigenLayer.operators[operator].stakedShares

    // Check if the operator's stake is sufficient for participation
    if(isStakeSufficient(stakedShares)) {
        // Since the operator is completing a task, we need to reserve some
        // time during which we are able to slash them
        operators[operator].tasks.push(TaskInfo({
            block: block.number,
            withdrawIdx: eigenLayer.withdrawalCount[operator],
            depositIdx: eigenLayer.deposits[operator].length,
            stakedShares: stakedShares,
            stakeNeededUntil: block.timestamp + TIME_FOR_SLASHING,
        }));

        // Now comes middleware-specific logic that depends on the kind of
        // task that operators need to complete
        ...
    }
}

```

prepareWithdrawal()

May need to be called by an operator or a delegator when they want to withdraw (some of) their staked shares from EigenLayer.

Preconditions:

- operator ∈ operators.keys()
- !operators[operator].exited

```

function prepareWithdrawal(address operator) {
    (BlockNumber updateBlock, Timestamp stakeNeededUntil) =
        getSlashingWindow(operator);

    eigenLayer.recordStakeUpdate(this, operator, updateBlock, stakeNeededUntil);
}

```

prepareExit()

Needs to be called by an operator when they no longer want to complete validation tasks.

Preconditions:

- operator is msg.sender
- operator \in operators.keys()
- !operators[operator].exited

```
function prepareExit(address operator) {
    operators[operator].exited = true;

    (_, Timestamp stakeNeededUntil) = getSlashingWindow(operator);
    eigenLayer.revokeSlashingAbility(this, operator, stakeNeededUntil);
}
```

slash()

Called by the middleware after it has been established that an operator has committed a slashable offense.

Preconditions:

- msg.sender is an account that has the permission to slash operators
- operator \in operators.keys()
- !operators[operator].slashed
- block.timestamp \leq operators[operator].stakeNeededUntil

```
function slash(address operator) {
    operators[operator].slashed = true;
    eigenLayer.freezeOperator(this, operator);
}
```

Appendix 3: Invariants

Each of the following invariants defines a property over the `EigenLayer` and/or `Middleware` model that is supposed to hold for all reachable states. We use `EL` and `M` to refer to instances of the `EigenLayer` and `Middleware` models, respectively.

When doing proofs, we use unprimed variables to refer to some state in the pre-state (before a function is executed) and primed variable names to refer to some state in the post-state (after a function has been executed). For example, `EL` refers to the `EigenLayer` model in the pre-state, and `EL'` refers to it in the post-state. Similarly, if a function `f` is defined in terms of `EL` and/or `M`, then we use `f()` to refer to its value in the pre-state, and `f'()` to refer to its value in the post-state.

Our goal in this section is to prove the [Main Invariant](#), which states that if an operator completes a task for a middleware, then the middleware can slash the operator's full staking amount from the time the task was completed. All other invariants in this section only serve to prove this main invariant.

NOTE: `EL` and `M` represent the *state* of the respective model, not their addresses! We use `EL.addr` and `M.addr` to refer to the address of the respective model.

NOTE: Effects of external calls to potentially dangerous contracts are not taken into account. Thus, potential re-entrancy attacks need to be analyzed separately.

Main Invariant

Main Invariant: When an operator completes a task for a middleware, then the operator's stake at that time remains slashable as long as necessary for that task (as determined by the task's `stakeNeededUntil` property).

```
∀ op ∈ M.slashableOperators() \ EL.frozenOperators():  
  ∀ task ∈ M.ongoingTasks(op):  
    task.stakedShares == EL.operators[op].stakedShares  
      + EL.sharesInWithdrawalQueue(op, task.withdrawIdx)
```

In words: Let `op` be an operator that has been registered at middleware `M` and that has not been frozen by the middleware itself or by any other middleware. Further, let `task` be some task that has been completed by the operator and for which the operator can still be slashed. Then the operator's stake at the time the task was completed (stored in `task.stakedShares`) is equal to the sum of the current operator's stake (`EL.operators[op].stakedShares`) and all the shares that are currently in the withdrawal queue that have been added after the task was completed (`EL.sharesInWithdrawalQueue(op,`

`task.withdrawIdx`). These are exactly the shares that can be slashed using `slashShares()` and `slashQueuedWithdrawal()`.

If this invariant holds, then middlewares can be sure that stakers avoid being slashed by quickly withdrawing all their stake.

NOTE: This invariant does not hold for `EL.resetFrozenStatus()`! This means that middlewares need to trust that governance uses its power to unfreeze operators responsibly.

This invariant is a direct corollary of Invariant 1.

Invariant 1

Invariant 1:

```
∀ op ∈ M.slashableOperators() \ EL.frozenOperators():  
  ∀ task ∈ M.ongoingTasks(op):  
    task.stakedShares == stakeAtop(task.withdrawIdx, task.depositIdx)
```

where

```
stakeAtop(withdrawIdx, depositIdx) =  
  EL.operators[op].stakedShares  
  + EL.sharesInWithdrawalQueue(op, withdrawIdx)  
  - EL.totalDepositsAfter(op, depositIdx)
```

NOTE: This invariant does not hold for `EL.resetFrozenStatus()`!

To prove that our model satisfies Invariant 1, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since no operators exist in the initial state, the invariant holds trivially.

Invariant holds after each function call: Let $op \in M'.slashableOperators() \setminus EL'.frozenOperators()$ and $task \in M'.ongoingTasks(op)$ be arbitrary. We need to show that if Invariant 1 holds in the pre-state, then it also holds for op and $task$ in the post-state.

- `EL.deposit(staker, strategy, tokenAmount)`

Since $M' == M$ and $EL'.frozenOperators() == EL.frozenOperators()$ we know that $op \in M.slashableOperators() \setminus EL.frozenOperators()$ and $task \in M.ongoingTasks(op)$. Hence, because we assume that Invariant 1 holds in the pre-state, we can also assume $task.stakedShares == stakeAt_{op}(task.withdrawIdx, task.depositIdx)$.

We know that `task.stakedShares` remains unchanged between the pre- and post-state. Thus, if we can show that `stakeAtop` also remains unchanged, we are done. In other words, we need to show that

★ `stakeAtop(task.withdrawIdx, task.depositIdx) == stakeAt'op(task.withdrawIdx, task.depositIdx)`

First, consider the case where `op == EL.stakers[staker].delegatedTo`. This implies the following:

- a) `EL'.operators[op].stakedShares == EL.operators[op].stakedShares + {strategy => shareAmount}`
- b) `EL'.totalDepositsAfter(op, task.depositIdx) == EL.totalDepositsAfter(op, task.depositIdx) + {strategy => shareAmount}`

This follows from `EL'.deposits[op] == EL.deposits[op] ++ [{strategy => shareAmount}]` and Invariant 3.1.

If we assume that Invariant 1 holds in the pre-state, then a) and b) imply ★.

Now consider the case where `op != EL.stakers[staker].delegatedTo`. Then the invariant is preserved because nothing relevant changes.

- `EL.registerAsOperator(operator, terms)`

First, consider the case where `op == operator`. Then by the precondition we get `op ∉ EL.operators.keys()`. This together with Invariant 6.2 implies `op ∉ M.operators.keys()`. Because `M' == M`, this also means `op ∉ M'.operators.keys()`. However, this contradicts our assumption that `op ∈ M'.slashableOperators() \ EL'.frozenOperators()`. Thus, we are done.

Now consider the case where `op != operator`. Then the invariant is preserved because nothing relevant changes.

- `EL.delegateTo(staker, operator)`

Same reasoning as for `EL.deposit()`.

- `EL.undelegate(staker)`

The invariant is preserved because nothing relevant changes.

- `EL.optIntoSlashing(operator, middleware)`

The invariant is preserved because nothing relevant changes.

- `EL.queueWithdrawal(staker, receiver, sharesToWithdraw, undelegateIfPossible)`

Since `M' == M` and `EL'.frozenOperators() == EL.frozenOperators()` we know that `op ∈ M.slashableOperators() \ EL.frozenOperators()` and `task ∈ M.ongoingTasks(op)`.

Hence, because we assume that Invariant 1 holds in the pre-state, we can also assume that `task.stakedShares == stakeAtop(task.withdrawIdx, task.depositIdx)` holds.

We know that `task.stakedShares` remains unchanged between the pre- and post-state. Thus, if we can show that `stakeAtop` also remains unchanged, we are done. In other words, we need to show that

$$\star \text{ stakeAt}_{op}(\text{task.withdrawIdx}, \text{task.depositIdx}) == \text{stakeAt}'_{op}(\text{task.withdrawIdx}, \text{task.depositIdx})$$

First, consider the case where `op == EL.stakers[staker].delegatedTo`. This implies the following facts:

- a) `EL'.operators[op].stakedShares == EL.operators[op].stakedShares - sharesToWithdraw`
- b) `EL'.sharesInWithdrawalQueue(op, task.withdrawIdx) == EL.sharesInWithdrawalQueue(op, task.withdrawIdx) + sharesToWithdraw`

The reasoning is as follows: First, let `withdrawal` denote the newly created `QueuedWithdrawal`, and observe that `withdrawal.sharesToWithdraw == sharesToWithdraw` and `withdrawal.withdrawIdx == EL.withdrawalCount[op]`. Then the claimed fact follows from `EL'.queuedWithdrawals == EL.queuedWithdrawals ∪ {withdrawal}` and Invariant 3.2.

If we assume that Invariant 1 holds in the pre-state, then a) and b) imply \star .

Now consider the case where `op != EL.stakers[staker].delegatedTo`. Then the invariant is preserved because nothing relevant changes.

- `EL.completeQueuedWithdrawal(receiver, withdrawal, slashingWindowIdx, receiveAsTokens)`

Since `M' == M` and `EL'.frozenOperators() == EL.frozenOperators()` we know that `op ∈ M.slashableOperators() \ EL.frozenOperators()` and `task ∈ M.ongoingTasks(op)`. Hence, because we assume that Invariant 1 holds in the pre-state, we can also assume that `task.stakedShares == stakeAtop(task.withdrawIdx, task.depositIdx)` holds.

We know that `task.stakedShares` remains unchanged between the pre- and post-state. Thus, if we can show that `stakeAtop` also remains unchanged, we are done. In other words, we need to show that

$$\star \text{ stakeAt}_{op}(\text{task.withdrawIdx}, \text{task.depositIdx}) == \text{stakeAt}'_{op}(\text{task.withdrawIdx}, \text{task.depositIdx})$$

First, consider the case where `op == withdrawal.operator`. Note that `EL.completeQueuedWithdrawal()` can essentially be broken into two parts:

- 1) Removing the withdrawal: `queuedWithdrawals.remove(withdrawal)`

- 2) Transferring the withdrawn tokens or shares, depending on the value of `receiveAsTokens`

Let's use EL'' to denote the state of the `EigenLayer` model after part 1), and EL' to denote the final state after part 2). If we can show that the invariant is preserved when going from EL to EL'' and when going from EL'' to EL' , then we have also shown that the invariant is preserved from EL to EL' , at which point we are done.

Note that the step from EL'' to EL' is, for the purpose of this invariant, equivalent to the case for `EL.deposit()`. Since we have already shown that `EL.deposit()` satisfies \star , the same applies to EL'' to EL' .

Thus, it remains to show that the step from EL to EL'' preserves the invariant. To this end, we make the following case distinction:

- Case `task.block <= withdrawal.startBlock`:

The precondition `canWithdraw(withdrawal.operator, withdrawal, slashingWindowIdx)` implies

a) `withdrawal.startBlock < s.updateBlock` and

b) `block.timestamp > s.stakeNeededUntil`,

where $s = EL.operators[operator].slashingWindowAt(slashingWindowIdx)$.

Because of the way `s.updateBlock` is computed, and with the help of Invariant 10, we know that

`s.updateBlock <= EL.operators[op].updateBlockFor(M.addr,slashingWindowIdx)`.

Further, observe that the assumption of this case (i.e., `task.block <= withdrawal.startBlock`) together with a) implies `task.block < s.updateBlock`.

Thus, we get

`task.block < EL.operators[op].updateBlockFor(M.addr,slashingWindowIdx)`.

This means we can apply Invariant 2 and get

c) `task.stakeNeededUntil ≤ EL.operators[op].stakeNeededUntilFor(M.addr,slashingWindowIdx)`

Because the way `s.stakeNeededUntil` is computed implies

`s.stakeNeededUntil ≥ EL.operators[op].stakeNeededUntilFor(M.addr,slashingWindowIdx)`
(with the help of Invariant 10), we also get

d) `task.stakeNeededUntil ≤ s.stakeNeededUntil`.

Additionally, observe that because `task ∈ M.ongoingTasks(op)` we know that

e) `block.timestamp ≤ task.stakeNeededUntil`.

From d) and e) we get

f) `block.timestamp ≤ s.stakeNeededUntil`.

Now b) and f) contradict each other, hence we are done.

- Case `task.block > withdrawal.startBlock`:

First, observe that `completeQueuedWithdrawal()` is implemented such that `EL''.operators[op].stakedShares` and `EL''.totalDepositsAfter(op, task.depositIdx)` remain unchanged compared to the pre-state. This means that the validity of ★ can only be affected by potential modifications to `EL''.queuedWithdrawals`. Note that the following condition is satisfied:

a) `EL''.queuedWithdrawals == EL.queuedWithdrawals \ {withdrawal}`

Our goal is to show that even though `EL''.queuedWithdrawals` is computed in this way, the following condition holds:

† `EL''.sharesInWithdrawalQueue(op, task.withdrawIdx) == EL.sharesInWithdrawalQueue(op, task.withdrawIdx)`

If we can show †, then this also implies ★.

To this end, observe that the assumption of this case (i.e., `task.block > withdrawal.startBlock`) together with Invariant 4 implies

b) `task.withdrawIdx > withdrawal.withdrawIdx`.

If we look at the implementation of `sharesInWithdrawalQueue()`, it is clear that only withdrawals with a `withdrawIdx` larger than or equal to `task.withdrawIdx` are considered. Since `withdrawal.withdrawIdx` is strictly smaller than `task.withdrawIdx`, it is not considered when computing `EL.sharesInWithdrawalQueue(op, task.withdrawIdx)`, which means that removing `withdrawal` as shown in a) has no effect. Thus, † holds and therefore also ★.

Now consider the case where `op != withdrawal.operator`. Then the invariant is preserved because nothing relevant changes.

- `EL.recordStakeUpdate(middleware, operator, updateBlock, stakeNeededUntil)`

The invariant is preserved because nothing relevant changes.

- `EL.revokeSlashingAbility(middleware, operator, bondedUntil)`

The invariant is preserved because nothing relevant changes.

- `EL.freezeOperator(middleware, operator)`

The invariant is preserved because it only cares about unfrozen operators.

- `EL.slashShares(staker, recipient, sharesToSlash)`

First, consider the case where `op == EL.stakers[staker].delegatedTo`. The precondition `EL.isFrozen(staker)` implies that one of the following conditions must be true:

- a) `EL.operators[staker].frozen`
- b) `EL.operators[EL.stakers[staker].delegatedTo].frozen`

We make a case distinction on whether a) or b) is true.

- Case a) is true.

Thus, we have `staker ∈ EL.operators.keys()`. This together with Invariant 5 gives us `EL.stakers[staker].delegatedTo == staker`, hence `staker == op` and `EL.operators[op].frozen`.

Because `EL'.frozenOperators() == EL.frozenOperators()`, this implies `EL'.operators[op].frozen`. However, this contradicts our assumption that `op ∈ M'.slashableOperators() \ EL'.frozenOperators()`. Thus, this case is unreachable and we are done.

- Case b) is true.

Because `op == EL.stakers[staker].delegatedTo` we get `EL.operators[op].frozen`. Because `EL'.frozenOperators() == EL.frozenOperators()`, this implies `EL'.operators[op].frozen`. However, this contradicts our assumption that `op ∈ M'.slashableOperators() \ EL'.frozenOperators()`. Thus, this case is unreachable and we are done.

Now consider the case where `op != stakers[staker].delegatedTo`. Then the invariant is preserved because nothing relevant changes.

- `EL.slashQueuedWithdrawal(receiver, withdrawal)`

First, consider the case where `op == withdrawal.operator`. Then the argument is the same as for `EL.slashShares()`.

Now consider the case where `op != withdrawal.operator`. Then the invariant is preserved because nothing relevant changes.

- `EL.resetFrozenStatus(operator)`

INVARIANT DOES NOT HOLD FOR THIS CASE!

- `M.register(operator)`

First, consider the case where $op == operator$. Then $M'.operators[op].tasks.length == 0$, which contradicts our assumption that $task \in M'.ongoingTasks(op)$. Thus, we are done.

Now consider the case where $op \neq operator$. Then the invariant is preserved because nothing relevant changes.

- $M.performOperatorAction(operator, payload)$

First, consider the case where $op == operator$. Then, we make the following case distinction:

- Case $EL.operators[operator].frozen$: This would imply $M'.operators[op].slashed$, which contradicts our assumption that $op \in M'.slashedOperators() \setminus EL'.frozenOperators()$.

- Case $!EL.operators[operator].frozen$:

- Case $M.isStakeSufficient(EL.operators[op].stakedShares)$:

Let $task'$ denote the newly inserted task. If $task \neq task'$, then nothing relevant changes and we are done.

Now consider the case where $task == task'$. Then we need to show $task'.stakedShares == stakeAt'_{op}(task'.withdrawIdx, task'.depositIdx)$. To this end, observe the following:

- $task'.stakedShares == EL.operators[operator].stakedShares$
- $EL.sharesInWithdrawalQueue(op, task'.withdrawIdx) == \{\}$
(where $\{\}$ denotes the empty mapping)

This follows from $task'.withdrawIdx == EL.withdrawalCount[op]$ and Invariant 7.

- $EL.totalDepositsAfter(op, task'.depositIdx) == \{\}$

This follows from $task'.depositIdx == EL.deposits[op].length$.

Then, the claim follows from a), b), c), and the fact that $EL' == EL$.

- Case $!M.isStakeSufficient(stakedShares)$: Then nothing changes and we are done.

Now consider the case where $op \neq operator$. Then the invariant is preserved because nothing relevant changes.

- $M.prepareWithdrawal(operator)$

The invariant is preserved because nothing relevant changes.

- `M.prepareExit(operator)`

The invariant is preserved because nothing relevant changes.

- `M.slash(operator)`

The only thing that changes is that an operator is removed from the set of slashable operators, which does not affect the preservation of the invariant.

Invariant 2

Invariant 2:

```


$$\begin{aligned}
&\forall op \in M.slashableOperators(): \\
&\quad \forall task \in M.operators[op].tasks: \\
&\quad \quad \forall i \leq \text{type}(uint).max: \\
&\quad \quad \quad EL.operators[op].updateBlockFor(M.addr, i) > task.block \Rightarrow \\
&\quad \quad \quad task.stakeNeededUntil \leq EL.operators[op].stakeNeededUntilFor(M.addr, i)
\end{aligned}$$


```

To prove that the model satisfies Invariant 2, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since no operators exist in the initial state, the invariant holds trivially.

Invariant holds after each function call: Let $op \in M'.slashableOperators()$, $task \in M'.operators[op].tasks$ and $i < \text{type}(uint).max$ be arbitrary. We need to show that the invariant holds for op and $task$ in the post-state, under the assumption that the invariant holds in the pre-state. In particular, we need to show

- ★ $EL'.operators[op].updateBlockFor(M'.addr, i) > task.block \Rightarrow task.stakeNeededUntil \leq EL'.operators[op].stakeNeededUntilFor(M'.addr, i)$

Let's start with the functions from the EigenLayer model:

- `EL.registerAsOperator(operator, terms)`

First, consider the case where $op == operator$. Then by the precondition we get $op \notin EL.operators.keys()$. This together with Invariant 6.2 implies $op \notin M.operators.keys()$. Because $M' == M$, this also means $op \notin M'.operators.keys()$. However, this contradicts our assumption that $op \in M'.slashableOperators()$. Thus, we are done.

Now consider the case where $op \neq operator$. Then the invariant is preserved because nothing relevant changes.

The only other functions in the EigenLayer model that modify any relevant state are `EL.recordStakeUpdate()` and `EL.revokeSlashingAbility()`. However, these functions can only be

called by Middleware. For this reason, we analyze their behavior directly at the place where they are called. Now, let's take a look at the functions of the Middleware model.

- `M.register(operator)`

First, consider the case where `op == operator`. Then `M'.operators[op].tasks.length == 0`, which contradicts our assumption that `task ∈ M'.operators[op].tasks`. Hence, we are done.

Now consider the case where `op != operator`. Then the invariant is preserved because nothing relevant changes.

- `M.performOperatorAction(operator, payload)`

First, consider the case where `op == operator`. Then we further distinguish the following cases:

- Case `EL.operators[operator].frozen`:

Then the only thing that changes is that the operator is removed from the set of slashable operators, which does not affect the preservation of the invariant.

- Case `!EL.operators[operator].frozen` and `isStakeSufficient(EL.operators[op].stakedShares)`:

Let `task'` denote the newly created task. If `task != task'`, then nothing relevant changes and we are done.

Let us now assume `task == task'`. Observe the following fact:

$$\text{a) } EL'.operators[op].updateBlockFor(M'.addr, i) \leq task'.block$$

To see this, first note that `task'.block == block.number` and `EL' == EL`. Further, Invariant 10 implies `M.addr ∈ EL.operators[op].slashingWindows.keys()`, which together with Invariant 8.1 implies `EL'.operators[op].updateBlockFor(M'.addr, i) ≤ EL.lastBlockNumber`. Finally, note that because block numbers are monotonically increasing, we know that `EL.lastBlockNumber ≤ block.number`. All of this together implies a).

Then, ★ follows directly from a).

- All other cases: Then nothing changes.

Now consider the case where `op != operator`. Then nothing relevant changes and we are done.

- `M.prepareWithdrawal(operator)`

First, consider the case where `op == operator`. We make a further case distinction:

- Case `M.operators[operator].tasks.length > 0`:

Define the following variables:

- `lastTask = M.operators[operator].tasks[-1]`
- `slashWin = SlashingWindow(
 lastTask.updateBlock,
 lastTask.stakeNeededUntil
)`

Then by the definition of `EL.recordStakeUpdate()` we know that

- `EL'.operators[op].slashingWindows[M.addr] ==
EL.operators[op].slashingWindows[M.addr] ++ [slashWin]`

We now show ★. To this end, observe the following facts:

- a) If `i < EL.operators[op].slashingWindows[M.addr].length`, then
`task.stakeNeededUntil <=`
`EL'.operators[op].stakeNeededUntilFor(M'.addr,i)`

This fact follows from the assumption that the invariant holds in the pre-state, that `M'.operators[op].tasks == M.operators[op].tasks`, and the fact that for `i < EL.operators[op].slashingWindows[M.addr].length`, we know that `EL'.operators[op].slashingWindows[M.addr][i] == EL.operators[op].slashingWindows[M.addr][i]`.

- b) If `i == EL'.operators[op].slashingWindows[M.addr].length - 1`, then
`task.stakeNeededUntil <=`
`EL'.operators[op].stakeNeededUntilFor(M'.addr,i)`

To see this, first note that Invariant 9 implies `task.stakeNeededUntil <= lastTask.stakeNeededUntil`.

Next, observe that `lastTask.stakeNeededUntil <= EL'.operators[op].stakeNeededUntilFor(M'.addr,i)` (this follows from `slashWin.stakeNeededUntil == lastTask.stakeNeededUntil` and the fact that `slashWin` has been added to `EL'.operators[op].slashingWindows[M.addr]`).

Together, this implies the claim.

- c) If `i > EL'.operators[op].slashingWindows[M.addr].length - 1`, then
`task.stakeNeededUntil <=`
`EL'.operators[op].stakeNeededUntilFor(M'.addr,i)`

Because of the way that `stakeNeededUntilFor()` is implemented, the reasoning for this case is the same as for b).

Together, a), b) and c) imply ★.

- Case $M.\text{operators}[\text{operator}].\text{tasks}.\text{length} == 0$:

This contradicts our assumption that $\text{task} \in M'.\text{operators}[\text{op}].\text{tasks}$. Hence, we are done.

Now consider the case where $\text{op} \neq \text{operator}$. Then nothing relevant changes and we are done.

- $M.\text{prepareExit}(\text{operator})$

First, consider the case where $\text{op} == \text{operator}$. Note that because $M.\text{prepareExit}()$ may call $EL.\text{revokeSlashingAbility}()$, it's possible that $EL.\text{operators}[\text{op}].\text{slashingWindows}[M.\text{addr}]$ is modified. If this happens, then the invariant is preserved for the same reason as laid out in the case for $M.\text{prepareWithdrawal}()$.

Now consider the case where $\text{op} \neq \text{operator}$. Then nothing relevant changes and we are done.

- $M.\text{slash}(\text{operator})$

The only thing that changes is that an operator is removed from the set of slashable operators, which does not affect the preservation of the invariant.

Invariant 3

Invariant 3.1:

```

 $\forall \text{ op} \in M.\text{slashableOperators}():$ 
   $\forall \text{ task} \in M.\text{ongoingTasks}(\text{op}):$ 
     $\text{task}.\text{depositIdx} \leq EL.\text{deposits}[\text{op}].\text{length}$ 

```

Invariant 3.2:

```

 $\forall \text{ op} \in M.\text{slashableOperators}():$ 
   $\forall \text{ task} \in M.\text{ongoingTasks}(\text{op}):$ 
     $\text{task}.\text{withdrawIdx} \leq EL.\text{withdrawalCount}[\text{op}]$ 

```

To prove that the model satisfies Invariant 3.1 and 3.2, we need to show that they hold both in the initial state and after each function call.

Invariants hold in the initial state: Since no operators exist in the initial state, the invariants hold trivially.

Invariants hold after each function call: Let us first consider the functions from the `EigenLayer` model:

- *Invariant 3.1:* No `EigenLayer` function modifies any middlewares, i.e., for each function we get $M' = M$. Further, note that for any operator op , deposits are only ever added to $EL.\text{deposits}[\text{op}]$,

never removed. Hence, `EL.deposits[op].length` increases monotonically. For these reasons, all `EigenLayer` functions satisfy Invariant 3.1.

- *Invariant 3.2:* Again, no `EigenLayer` function modifies any middlewares, i.e., for each function we get $M' = M$. Further, note that `EL.withdrawalCount[op]` increases monotonically. For these reasons, all `EigenLayer` functions satisfy Invariant 3.2.

Next, let us consider the functions from the `Middleware` model:

- `M.register(operator)`

Both invariants are preserved because no new tasks are added and neither `EL.deposits` nor `EL.withdrawalCount` is modified.

- `M.performOperatorAction(operator, payload)`

Both invariants are preserved because the newly created task `task` satisfies `task.depositIdx == EL.deposits[operator].length` and `task.withdrawIdx == EL.withdrawalCount[operator]`.

- `M.prepareWithdrawal(operator)`

Both invariants are preserved because no new tasks are created and neither `EL.deposits` nor `EL.withdrawalCount` is modified.

- `M.prepareExit(operator)`

Both invariants are preserved because no new tasks are created and neither `EL.deposits` nor `EL.withdrawalCount` is modified.

- `M.slash(operator)`

Both invariants are preserved because no new tasks are created and neither `EL.deposits` nor `EL.withdrawalCount` is modified.

Invariant 4

Invariant 4:

```
∀ op ∈ M.slashableOperators():  
  ∀ task ∈ M.ongoingTasks(op):  
    ∀ w ∈ EL.queuedWithdrawalsFor(op):  
      task.block > w.startBlock ⇒ task.withdrawIdx > w.withdrawIdx
```

To prove that the model satisfies Invariant 4, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since no operators exist in the initial state, the invariant holds trivially.

Invariant holds after each function call: Let $op \in M'.slashableOperators()$, $task \in M'.ongoingTasks(op)$ and $w \in EL'.queuedWithdrawalsFor(op)$. We need to show that if the invariant holds in the pre-state, then the following condition is satisfied:

$$\star \quad task.block > w.startBlock \Rightarrow task.withdrawIdx > w.withdrawIdx$$

When considering the functions of the EigenLayer model, only the following three functions modify any relevant state (i.e., they modify `EL.queuedWithdrawals`).

- `EL.queueWithdrawal(staker, receiver, sharesToWithdraw, undelegateIfPossible)`

Let `newWithdrawal` denote the `QueuedWithdrawal` that is inserted into `queuedWithdrawals`. We make the following case distinction:

- Case `w == newWithdrawal`:

Then `w.startBlock == block.number`. Further, Invariant 8.2 implies `M'.lastBlockNumber >= task.block`. This, together with the fact that `block.number >= M'.lastBlockNumber` (block numbers increase monotonically), implies `w.startBlock >= task.block`. Thus, the left-hand side of the implication in \star is false, which means we are done.

- Case `w != newWithdrawal`:

Then \star follows directly from the assumption that Invariant 4 holds in the pre-state.

- `EL.completeQueuedWithdrawal(receiver, withdrawal, slashingWindowIdx, receiveAsTokens)`

The only relevant change is that a queued withdrawal is removed from `EL.queuedWithdrawals`, which does not affect the preservation of the invariant.

- `EL.slashQueuedWithdrawal(receiver, withdrawal)`

The only relevant change is that a queued withdrawal is removed from `EL.queuedWithdrawals`, which does not affect the invariant.

Next, let's look at the functions of the Middleware model:

- `M.register(operator)`

No new tasks or withdrawals are added, which means \star follows directly from the assumption that Invariant 4 holds in the pre-state.

- `M.performOperatorAction(operator, payload)`

Let `newTask` denote the `TaskInfo` that is inserted into `M.operators[operator].tasks`. We make the following case distinction:

- Case `op == operator` and `task == newTask`:

Then `task.withdrawIdx == EL.withdrawalCount[operator]`. Additionally, Invariant 7 implies `w.withdrawIdx < EL.withdrawalCount[operator]`. Together, this gives us `task.withdrawIdx > w.withdrawIdx`, which proves ★.

- Otherwise:

Then ★ follows directly from the assumption that Invariant 4 holds in the pre-state.

- `M.prepareWithdrawal(operator)`

No new tasks or withdrawals are added, which means ★ follows directly from the assumption that Invariant 4 holds in the pre-state.

- `M.prepareExit(operator)`

No new tasks or withdrawals are added, which means ★ follows directly from the assumption that Invariant 4 holds in the pre-state.

- `M.slash(operator)`

No new tasks or withdrawals are added, which means ★ follows directly from the assumption that Invariant 4 holds in the pre-state.

Invariant 5

Invariant 5:

$$\forall op \in EL.operators.keys(): EL.stakers[op].delegatedTo == op$$

To prove that the model satisfies Invariant 5, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since no operators exist in the initial state, the invariant holds trivially.

Invariant holds after each function call: Let `op ∈ EL'.operatoros.keys()`. We need to show that if the invariant holds in the pre-state, then the following condition is satisfied:

★ `EL.stakers[op].delegatedTo == op`

We now look at all the functions of the EigenLayer model:

- `EL.deposit(staker, strategy, tokenAmount)`

First consider the case where `op == staker`. Then, `op ∈ EL'.operatoros.keys()` together with `EL'.operatoros == EL.operators` implies `op ∈ EL.operatoros.keys()`. This together with Invariant 6.1 implies `op ∈ EL.stakers.keys()`. Hence, `deposit()` does not create a new staker. From this we can deduce that `EL'.stakers[op].delegatedTo ==`

`EL.stakers[op].delegatedTo`, which means that ★ follows directly from the assumption that Invariant 5 holds in the pre-state.

If `op != staker`, then the invariant is preserved because nothing relevant changes.

- `EL.registerAsOperator(operator, terms)`

If `op == operator`, then `EL'.stakers[op].delegatedTo == op`, which means that ★ is satisfied.

If `op != operator`, then `EL'.stakers[op].delegatedTo == EL.stakers[op].delegatedTo`, which means that ★ follows directly from the assumption that Invariant 5 holds in the pre-state.

- `EL.delegateTo(staker, operator)`

First consider the case where `op == staker`. Since `EL'.operators.keys() == EL.operators.keys()` and `op ∈ EL'.operators.keys()`, we get `op ∈ EL.operators.keys()`. This together with Invariant 6.1 implies `op ∈ EL.stakers.keys()`. In turn, this together with the preconditions implies `EL.stakers[op].delegatedTo = address(0)`. However, this contradicts our assumption that Invariant 5 holds in the pre-state. Hence, we are done.

If `op != staker`, then `EL'.stakers[op].delegatedTo == EL.stakers[op].delegatedTo`, which means that ★ follows directly from the assumption that Invariant 5 holds in the pre-state.

- `EL.undelegate(staker)`

This only affects stakers who are not operators.

- `EL.queueWithdrawal(staker, receiver, sharesToWithdraw, undelegateIfPossible)`

If `undelegate()` is called, then see the case for `EL.undelegate()` on why the invariant holds.

Otherwise, nothing relevant changes.

All other functions do not modify `EL.stakers[op].delegatedTo`, hence the invariant is trivially preserved.

Invariant 6

Invariant 6.1:

$$EL.operators.keys() \subseteq EL.stakers.keys()$$

To prove that the model satisfies Invariant 6.1, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since no operators or stakers exist in the initial state, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 6.1 holds in the pre-state, we show that it also holds in the post-state for all the functions of the EigenLayer model:

- `EL.deposit(staker, strategy, tokenAmount)`

Then `EL'.operators.keys() == EL.operators.keys()` and `EL'.stakers.keys() \supseteq EL.stakers.keys()` imply that Invariant 6.1 is preserved.

- `EL.registerAsOperator(operator, terms)`

If the new operator is not already a staker, then a new staker is added as well. Hence, the invariant is preserved.

- `EL.delegateTo(staker, operator)`

Then `EL'.operators.keys() == EL.operators.keys()` and `EL'.stakers.keys() \supseteq EL.stakers.keys()` imply that Invariant 6.1 also holds in the post-state.

For all other functions, the invariant is preserved because both the set of stakers and the set of operators remain unchanged.

Invariant 6.2:

$$M.operators.keys() \subseteq EL.operators.keys()$$

To prove that the model satisfies Invariant 6.2, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any operators, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 6.2 holds in the pre-state, we show that it also holds in the post-state for all the functions of the EigenLayer model and the Middleware model.

First, let us consider the functions of the EigenLayer model. In this case, `M.operators.keys()` always remains unchanged and `EL.operators.keys()` only ever grows. Hence, the invariant is easily preserved.

Next, let us look at the Middleware functions:

- `M.register(operator)`

Let `op \in M'.operators.keys()`. We need to show that `op \in EL'.operators.keys()`.

If `op \notin M.operators.keys()`, then `op == operator`. Note that `M.register()` requires the successful execution of `EL.recordStakeUpdate()`, which in turn requires `canSlash(M.addr, op)`. This ensures that `p \in EL.operators.keys()`. Because of `EL'.operators.keys() == EL.operators.keys()` this also gives us `p \in EL'.operators.keys()`.

If $op \in M.\text{operators.keys}()$, then $p \in EL.\text{operators.keys}()$ follows from the assumption that Invariant 6.2 holds in the pre-state, and $p \in EL'.\text{operators.keys}()$ follows from $EL'.\text{operators.keys}() == EL.\text{operators.keys}()$.

- `M.performOperatorAction(operator, payload)`

The invariant is preserved because $M'.\text{operators.keys}() == M.\text{operators.keys}()$ and $EL'.\text{operators.keys}() == EL.\text{operators.keys}()$.

- `M.prepareWithdrawal(operator)`

The invariant is preserved because $M'.\text{operators.keys}() == M.\text{operators.keys}()$ and $EL'.\text{operators.keys}() == EL.\text{operators.keys}()$.

- `M.prepareExit(operator)`

The invariant is preserved because $M'.\text{operators.keys}() == M.\text{operators.keys}()$ and $EL'.\text{operators.keys}() == EL.\text{operators.keys}()$.

- `M.slash(operator)`

The invariant is preserved because $M'.\text{operators.keys}() == M.\text{operators.keys}()$ and $EL'.\text{operators.keys}() == EL.\text{operators.keys}()$.

Invariant 7

Invariant 7:

$\forall w \in EL.\text{queuedWithdrawals}:$
 $w.\text{withdrawIdx} < EL.\text{withdrawalCount}[w.\text{operator}]$

To prove that the model satisfies Invariant 7, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any queued withdrawals, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 7 holds in the pre-state, we show that it also holds in the post-state for all the functions of the EigenLayer model:

- `EL.queueWithdrawal(staker, receiver, sharesToWithdraw, undelegateIfPossible)`

Let $w \in EL'.\text{queuedWithdrawals}$ be arbitrary. We need to show $w.\text{withdrawIdx} < EL'.\text{withdrawalCount}[w.\text{operator}]$.

Next, let `newWithdrawal` denote the new `QueuedWithdrawal` that is added to `EL.queuedWithdrawals`.

If $w == \text{newWithdrawal}$, then $w.\text{withdrawIdx} == \text{EL}.\text{withdrawalCount}[w.\text{operator}]$ and $\text{EL}'.\text{withdrawalCount}[w.\text{operator}] == \text{EL}.\text{withdrawalCount}[w.\text{operator}] + 1$. Thus, this implies $w.\text{withdrawIdx} < \text{EL}'.\text{withdrawalCount}[w.\text{operator}]$ and the invariant is preserved.

If $w \neq \text{newWithdrawal}$, then the invariant is preserved because nothing relevant changes.

Note that no other function creates new withdrawals or modifies $\text{EL}.\text{withdrawalCount}$, hence the invariant is trivially preserved by them.

Invariant 8

Invariant 8.1:

$$\begin{aligned} \forall \text{ op} \in \text{EL}.\text{operators}.\text{keys}(): \\ \quad \forall \text{ m} \in \text{EL}.\text{operators}[\text{op}].\text{slashingWindows}.\text{keys}(): \\ \quad \quad \forall \text{ s} \in \text{EL}.\text{operators}[\text{op}].\text{slashingWindows}[\text{m}]: \\ \quad \quad \quad \text{s}.\text{updateBlock} \leq \text{EL}.\text{lastBlockNumber} \end{aligned}$$

To prove that the model satisfies Invariant 8.1, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any operators, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 8.1 holds in the pre-state, we show that it also holds in the post-state of any function defined by the **EigenLayer** model:

- `EL.registerAsOperator(operator, terms)`

The invariant is preserved because the newly created operator does not contain any `SlashingWindow`.

- `EL.recordStakeUpdate(middleware, operator, updateBlock, stakeNeededUntil)`

Let $\text{op} \in \text{EL}'.\text{operators}.\text{keys}()$, $\text{m} \in \text{EL}'.\text{operators}[\text{op}].\text{slashingWindows}.\text{keys}()$ and $\text{s} \in \text{EL}'.\text{operators}[\text{op}].\text{slashingWindows}[\text{m}]$. We need to show $\text{s}.\text{updateBlock} \leq \text{EL}'.\text{lastBlockNumber}$.

First, consider the case where $\text{op} == \text{operator}$. Let `slashWin` denote the newly added `SlashingWindow`. We make the following case distinction:

- Case $\text{s} == \text{slashWin}$:

Then $\text{s}.\text{updateBlock} == \text{updateBlock}$. Note that the preconditions imply $\text{s}.\text{updateBlock} \leq \text{blockNumber}$. Further, note that $\text{EL}'.\text{lastBlockNumber} == \text{block.number}$. Thus, we get $\text{s}.\text{updateBlock} \leq \text{EL}'.\text{lastBlockNumber}$, which shows that the invariant holds.

- Case $s \neq \text{slashWin}$:

Since block numbers increase monotonically, we know that $EL'.\text{lastBlockNumber} \geq EL.\text{lastBlockNumber}$. Thus, since we know that the invariant holds for s in the pre-state, this means the invariant will also hold for s in the post-state.

Next, if $op \neq \text{operator}$, then the invariant is preserved because nothing relevant changes.

For all other functions, the invariant is preserved because they do not update $EL.\text{operators}[op].\text{slashingWindows}$, and because $EL.\text{lastBlockNumber}$ increases monotonically.

Invariant 8.2:

```

 $\forall op \in M.\text{operators}.\text{keys}():$ 
   $\forall task \in M.\text{operators}[op].\text{tasks}:$ 
     $task.\text{block} \leq M.\text{lastBlockNumber} \ \&\&$ 
     $task.\text{stakeNeededUntil} \leq M.\text{lastTimestamp} + M.\text{TIME\_FOR\_SLASHING}$ 

```

To prove that the model satisfies Invariant 8.2, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any operators, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 8.2 holds in the pre-state, we show that it also holds in the post-state of any function defined by the Middleware model:

- $M.\text{register}(\text{operator})$

The invariant is preserved because the newly created operator has no tasks yet.

- $M.\text{performOperatorAction}(\text{operator}, \text{payload})$

Let $op \in M'.\text{operators}.\text{keys}()$ and $task \in M'.\text{operators}[op].\text{tasks}$. We need to show $task.\text{block} \leq M'.\text{lastBlockNumber}$ and $task.\text{stakeNeededUntil} \leq M'.\text{lastTimestamp} + M'.\text{TIME_FOR_SLASHING}$.

First, consider the case where $op == \text{operator}$. Let newTask denote the newly created `TaskInfo`. We make the following case distinction:

- Case $task == \text{newTask}$:

Then $task.\text{block} == \text{block}.\text{number}$ and $task.\text{stakeNeededUntil} == \text{block}.\text{timestamp} + M.\text{TIME_FOR_SLASHING}$.

First, note that $M'.\text{lastBlockNumber} == \text{block}.\text{number}$. Thus, we get $task.\text{block} \leq M'.\text{lastBlockNumber}$.

Further, note that $M'.lastTimestamp == block.timestamp$ and $M'.TIME_FOR_SLASHING == M.TIME_FOR_SLASHING$. Thus, we get $task.stakeNeededUntil \leq M'.lastTimestamp + M'.TIME_FOR_SLASHING$.

- Case $task \neq newTask$:

Since block numbers and timestamps increase monotonically, we know that $M'.lastBlockNumber \geq M.lastBlockNumber$ and $M'.lastTimestamp \geq M.lastTimestamp$. Thus, since we know that the invariant holds for $task$ in the pre-state, this means the invariant will also hold for $task$ in the post-state.

Next, if $op \neq operator$, then the invariant is preserved because nothing relevant changes.

For all other functions, the invariant is preserved because they do not update $M.operators[op].tasks$, and because $M.lastBlockNumber$ and $M.lastTimestamp$ increase monotonically.

Invariant 9

Invariant 9:

```

 $\forall op \in M.operators.keys():$ 
   $\forall i, i' < M.operators[op].task.length:$ 
     $i \leq i' \Rightarrow$ 
       $M.operators[op].task[i].block \leq M.operators[op].task[i'].block \ \&\&$ 
       $M.operators[op].task[i].stakeNeededUntil \leq$ 
       $M.operators[op].task[i'].stakeNeededUntil$ 

```

To prove that the model satisfies Invariant 9, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any operators, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 9 holds in the pre-state, we show that it also holds in the post-state of any function defined by the Middleware model:

- $M.register(operator)$

The invariant is preserved because the newly created operator has no tasks yet.

- $M.performOperatorAction(operator, payload)$

Let $op \in M'.operators.keys()$ and $i, i' < M'.operators[op].task.length$ be arbitrary. Further, let $t = M'.operators[op].task[i]$ and $t' = M'.operators[op].task[i']$. We need to show

★ $t.block \leq t'.block \ \&\& \ t.stakeNeededUntil \leq t'.stakeNeededUntil$

First, consider the case where $op == operator$. Let $newTask$ denote the newly created $TaskInfo$. We make the following case distinction:

- Case $t' == newTask$:

Then $t'.block == block.number$ and $t'.stakeNeededUntil == block.timestamp + M.TIME_FOR_SLASHING$.

If $t == t'$, then we immediately get ★.

If $t != t'$, then Invariant 8.2 implies $t.block \leq M.lastBlockNumber$ and $t.stakeNeededUntil \leq M.lastTimestamp + M.TIME_FOR_SLASHING$. Since $M.lastBlockNumber \leq block.number$ and $M.lastTimestamp \leq block.timestamp$, this also implies ★.

- Case $t' != newTask$:

Then both t and t' must already have existed in the pre-state. Hence, ★ follows from the assumption that Invariant 9 holds in the pre-state.

Next, if $op != operator$, then the invariant is preserved because nothing relevant changes.

For all other functions, the invariant is preserved because they do not update $M.operators[op].tasks$.

Invariant 10

Invariant 10:

```

 $\forall op \in M.slashableOperators():$ 
   $M.addr \in EL.operators[op].slashingWindows.keys()$ 

```

To prove that the model satisfies Invariant 10, we need to show that it holds both in the initial state and after each function call.

Invariant holds in the initial state: Since in the initial state there do not exist any operators, the invariant holds trivially.

Invariant holds after each function call: Assuming that Invariant 10 holds in the pre-state, we show that it also holds in the post-state of any function defined by either the EigenLayer or the Middleware model. To this end, let $op \in M'.slashableOperators()$ be arbitrary. We need to show

★ $M.addr \in EL'.operators[op].slashingWindows.keys()$

(Note that the middleware address is immutable, i.e., $M'.addr == M.addr$.)

- $EL.deposit(staker, strategy, tokenAmount)$

The invariant is preserved because nothing relevant changes.

- `EL.registerAsOperator(operator, terms)`

First, consider the case where $op == operator$. Then the preconditions imply $op \in EL.operators.keys()$. This together with Invariant 6.2 gives us $op \in M.operators.keys()$. However, this contradicts our assumption that $op \in M.slashableOperators()$.

Next, if $op \neq operator$, then the invariant is preserved because nothing relevant changes.

- `EL.delegateTo(staker, operator)`

The invariant is preserved because nothing relevant changes.

- `EL.undelegate(staker)`

The invariant is preserved because nothing relevant changes.

- `EL.optIntoSlashing(operator, middleware)`

The invariant is preserved because nothing relevant changes.

- `EL.queueWithdrawal(staker, receiver, sharesToWithdraw, undelegateIfPossible)`

The invariant is preserved because nothing relevant changes.

- `EL.completeQueuedWithdrawal(receiver, withdrawal, slashingWindowIdx, receiveAsTokens)`

The invariant is preserved because nothing relevant changes.

- `EL.recordStakeUpdate(middleware, operator, updateBlock, stakeNeededUntil)`

First, consider the case where $op == operator$. Then note that

- $EL'.operators[op].slashingWindows.keys() \supseteq EL.operators[op].slashingWindows.keys()$

This modification ensures that the invariant is preserved.

Next, if $op \neq operator$, then the invariant is preserved because nothing relevant changes.

- `EL.revokeSlashingAbility(middleware, operator, bondedUntil)`

First, consider the case where $op == operator$. Then note that

- $EL'.operators[op].slashingWindows.keys() \supseteq EL.operators[op].slashingWindows.keys()$

This modification ensures that the invariant is preserved.

Next, if $op \neq operator$, then the invariant is preserved because nothing relevant changes.

- `EL.freezeOperator(middleware, operator)`

The invariant is preserved because nothing relevant changes.

- `EL.slashShares(staker, recipient, sharesToSlash)`

The invariant is preserved because nothing relevant changes.

- `EL.slashQueuedWithdrawal(receiver, withdrawal)`

The invariant is preserved because nothing relevant changes.

- `EL.resetFrozenStatus(operator)`

The invariant is preserved because nothing relevant changes.

- `M.register(operator)`

First, consider the case where `op == operator`. Next, note that `M.register()` requires the successful execution of `EL.recordStakeUpdate()`, which ensures that `M.addr ∈ EL'.operators[op].slashingWindows.keys()`. Thus, this directly gives us ★.

Next, if `op != operator`, then the invariant is preserved because nothing relevant changes.

- `M.performOperatorAction(operator, payload)`

The only potentially relevant change is that `operator` becomes inactive due to being slashed. However, this cannot cause the invariant to fail.

- `M.prepareWithdrawal(operator)`

The only state change is due to calling `EL.recordStakeUpdate()`, which we have already shown to preserve the invariant.

- `M.prepareExit(operator)`

The only state change is due to calling `EL.revokeSlashingAbility()`, which we have already shown to preserve the invariant.

- `M.slash(operator)`

First, consider the case where `op == operator`. Then, `M'.operators[op].slashed == true`, which contradicts our assumption that `op ∈ M'.slashableOperators()`.

Next, if `op != operator`, then the invariant is preserved because nothing relevant changes.