Zellic

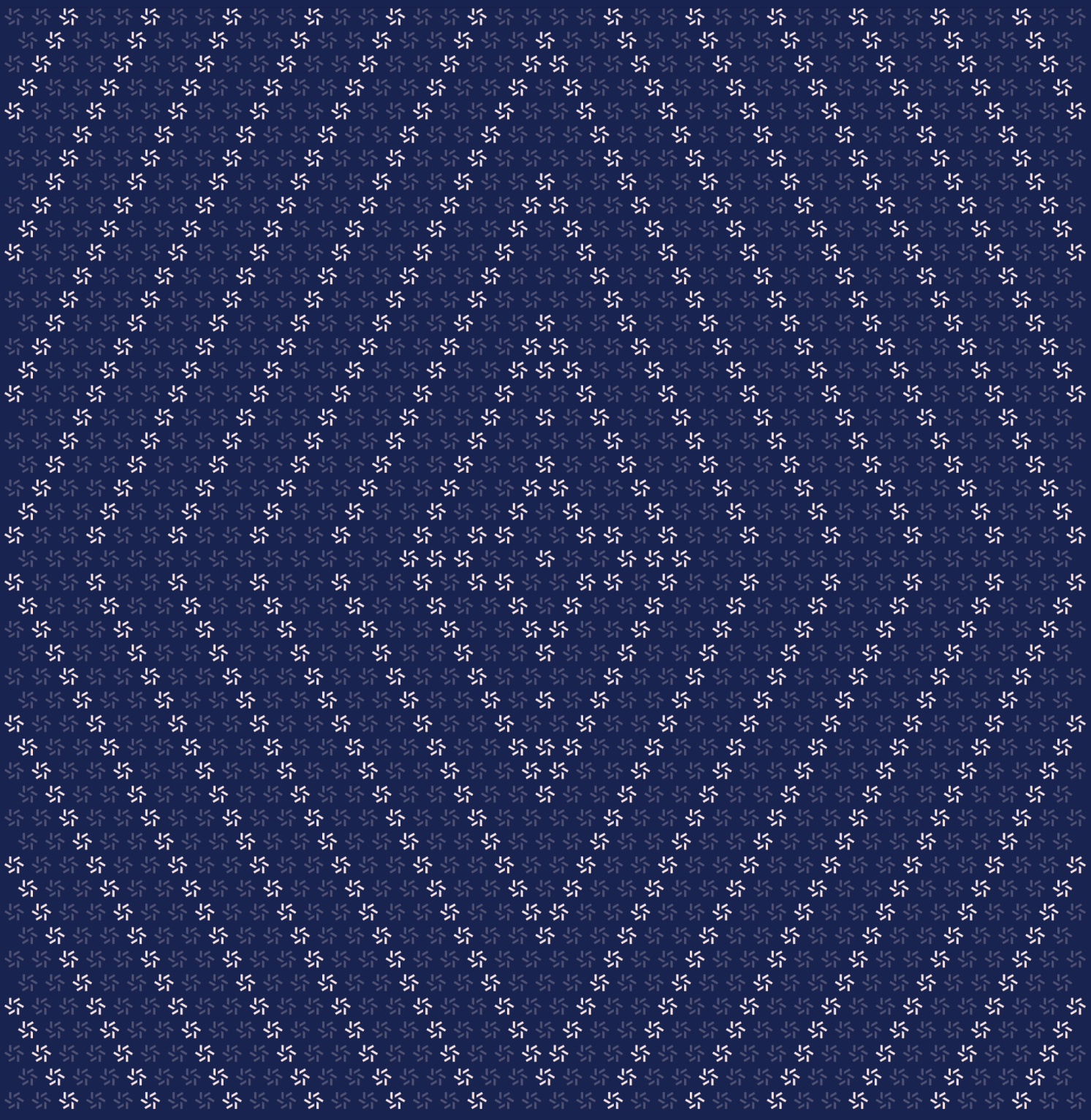
Prepared for
Eddy
Thai
Mitosis

Prepared by
Hojung Han
Jaeeu Kim
Zellic

May 22, 2025

Mitosis

Smart Contract Security Assessment



Contents

About Zellic

Zellic is a vulnerability research firm with deep expertise in blockchain security. We specialize in EVM, Move (Aptos and Sui), and Solana as well as Cairo, NEAR, and Cosmos. We review L1s and L2s, cross-chain protocols, wallets and applied cryptography, zero-knowledge circuits, web applications, and more.

Prior to Zellic, we founded the #1 CTF (competitive hacking) team ↗ worldwide in 2020, 2021, and 2023. Our engineers bring a rich set of skills and backgrounds, including cryptography, web security, mobile security, low-level exploitation, and finance. Our background in traditional information security and competitive hacking has enabled us to consistently discover hidden vulnerabilities and develop novel security research, earning us the reputation as the go-to security firm for teams whose rate of innovation outpaces the existing security landscape.

For more on Zellic's ongoing security research initiatives, check out our website zellic.io ↗ and follow @zellic_io ↗ on Twitter. If you are interested in partnering with Zellic, contact us at hello@zellic.io ↗.

1. Overview

1.1. Executive Summary

Zellic conducted a security assessment for Mitosis from March 31st to May 15th, 2025. During this engagement, Zellic reviewed Mitosis's code for security vulnerabilities, design issues, and general weaknesses in security posture.

Pull Requests #349 ↗ and #28 ↗ were reviewed as part of the secondary review period conducted from May 14 to May 15, 2025.

1.2. Goals of the Assessment

In a security assessment, goals are framed in terms of questions that we wish to answer. These questions are agreed upon through close communication between Zellic and the client. In this assessment, we sought to answer the following questions:

Chain side

- What malicious actions can a block proposer perform?
- Is the replacement for the existing staking module fully compatible with the rest of the system?
- In the forked evmengine (octane) module from Omni chain, are there any modifications compared to the original that could cause issues?
- What are the potential issues related to chain reorganization?

Contract side

- Are there cases where excessive authority is granted relative to the staked assets?
- Is there any risk of staked assets being leaked or stolen?
- Is there a possibility for users to receive excessive or unintended rewards?
- Is there a risk that staked assets could become unintentionally frozen?
- Could unexpected behavior occur due to issues with cross-chain transactions?

1.3. Non-goals and Limitations

We did not assess the following areas that were outside the scope of this engagement:

- Front-end components
- Infrastructure relating to the project
- Key custody

Due to the time-boxed nature of security assessments in general, there are limitations in the coverage an assessment can provide.

1.4. Results

During our assessment on the scoped Mitosis contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature.

Additionally, Zellic recorded its notes and observations from the assessment for the benefit of Mitosis in the Discussion section (4. ↗).

Breakdown of Finding Impacts

Impact Level	Count
Critical	1
High	2
Medium	1
Low	1
Informational	6

2. Introduction

2.1. About Mitosis

Mitosis contributed the following description of Mitosis:

> Mitosis is a L1 network designed for programmable liquidity that enhances the liquidity provision experience for both DeFi projects and liquidity providers. With Mitosis, DeFi projects can conduct targeted marketing to attract liquidity. Liquidity providers can trade tokenized LP positions on the Mitosis chain, allowing users to easily trade yield positions.

2.2. Methodology

During a security assessment, Zellic works through standard phases of security auditing, including both automated testing and manual review. These processes can vary significantly per engagement, but the majority of the time is spent on a thorough manual review of the entire scope.

Alongside a variety of tools and analyzers used on an as-needed basis, Zellic focuses primarily on the following classes of security and reliability issues:

Basic coding mistakes. Many critical vulnerabilities in the past have been caused by simple, surface-level mistakes that could have easily been caught ahead of time by code review. Depending on the engagement, we may also employ sophisticated analyzers such as model checkers, theorem provers, fuzzers, and so on as necessary. We also perform a cursory review of the code to familiarize ourselves with the contracts.

Business logic errors. Business logic is the heart of any smart contract application. We examine the specifications and designs for inconsistencies, flaws, and weaknesses that create opportunities for abuse. For example, these include problems like unrealistic tokenomics or dangerous arbitrage opportunities. To the best of our abilities, time permitting, we also review the contract logic to ensure that the code implements the expected functionality as specified in the platform's design documents.

Integration risks. Several well-known exploits have not been the result of any bug within the contract itself; rather, they are an unintended consequence of the contract's interaction with the broader DeFi ecosystem. Time permitting, we review external interactions and summarize the associated risks: for example, flash loan attacks, oracle price manipulation, MEV/sandwich attacks, and so on.

Code maturity. We look for potential improvements in the codebase in general. We look for violations of industry best practices and guidelines and code quality standards. We also provide suggestions for possible optimizations, such as gas optimization, upgradability weaknesses, centralization risks, and so on.

For each finding, Zellic assigns it an impact rating based on its severity and likelihood. There is no hard-and-fast formula for calculating a finding's impact. Instead, we assign it on a case-by-case

basis based on our judgment and experience. Both the severity and likelihood of an issue affect its impact. For instance, a highly severe issue's impact may be attenuated by a low likelihood. We assign the following impact ratings (ordered by importance): Critical, High, Medium, Low, and Informational.

Zellic organizes its reports such that the most important findings come first in the document, rather than being strictly ordered on impact alone. Thus, we may sometimes emphasize an "Informational" finding higher than a "Low" finding. The key distinction is that although certain findings may have the same impact rating, their *importance* may differ. This varies based on various soft factors, like our clients' threat models, their business needs, and so on. We aim to provide useful and actionable advice to our partners considering their long-term goals, rather than a simple list of security issues at present.

Finally, Zellic provides a list of miscellaneous observations that do not have security impact or are not directly related to the scoped contracts itself. These observations — found in the Discussion (4. ↗) section of the document — may include suggestions for improving the codebase, or general recommendations, but do not necessarily convey that we suggest a code change.

2.3. Scope

The engagement involved a review of the following targets:

Mitosis Contracts

Types	Solidity, Go
Platform	EVM-compatible
Target	protocol
Repository	https://github.com/mitosis-org/protocol ↗
Version	`f18b1965c3d5816e422edb206efe598f0fb39899`
Programs	`src/*`
Target	protocol - PR #349 secondary review
Repository	https://github.com/mitosis-org/protocol ↗
Version	`Diffs in PR #349 between f18b1965...65b4febd`
Programs	`src/*`

Target	chain
Repository	https://github.com/mitosis-org/chain ↗
Version	`070481b91d51c8da3e5520cd3c7c32a5fc2ec999`
Programs	`app/*` `x/*`

Target	chain - PR #28 secondary review
Repository	https://github.com/mitosis-org/chain ↗
Version	`Diffs in PR #28 between 070481b9...827918e4`
Programs	`keeper/*` `types/*`

2.4. Project Overview

Zellic was contracted to perform a security assessment for a total of 8.1 person-weeks. The assessment was conducted by two consultants over the course of 7 calendar weeks.

Pull Requests #349 ↗ and #28 ↗ were reviewed as part of the secondary review period conducted from May 14 to May 15, 2025.

Contact Information

The following project managers were associated with the engagement:

Jacob Goreski
Engagement Manager
jacob@zellic.io

Chad McDonald
Engagement Manager
chad@zellic.io

The following consultants were engaged to conduct the assessment:

Hojung Han
Engineer
hojung@zellic.io

Jaeeu Kim
Engineer
jaeeu@zellic.io

2.5. Project Timeline

March 31, 2025	Kick-off call
March 31, 2025	Start of primary review period
May 14th, 2025	Start of secondary review period
May 15, 2025	End of review period

3. Detailed Findings

3.1. Missing unstake-amount validation

Target	protocol/src/hub/staking/ValidatorStaking.sol		
Category	Coding Mistakes	**Severity**	Critical
Likelihood	High	**Impact**	Critical

Description

The `_requestUnstake` function in the ValidatorStaking contract does not properly validate whether the unstaking amount is less than or equal to the user's staked amount. This oversight could potentially allow users to unstake more tokens than they have actually staked.

The `_assertUnstakeAmountCondition` function only checks whether the unstaking amount is greater than or equal to the minimum unstaking amount; it fails to verify that the amount is within the user's available stake:

```
function _assertUnstakeAmountCondition(StorageV1 storage $, address valAddr,
    address staker, uint256 amount)
  internal
  view
{
  uint256 currentStaked = $.staked[valAddr][staker].latest();
  uint256 minUnstaking = $.minUnstakingAmount;

  if (amount != currentStaked) {
    require(amount >= minUnstaking,
    IValidatorStaking__InsufficientMinimumAmount(minUnstaking));
  }
}
```

This issue is particularly concerning because the state storage system uses checkpoints that operate with the `unchecked` keyword, which allows underflows when subtracting values:

```
function _storeUnstake(StorageV1 storage $, uint48 now_, address valAddr,
    address staker, uint208 amount)
  internal
  virtual
{
  _push($.staked[valAddr][staker], now_, amount, _opSub);
  _push($.stakerTotal[staker], now_, amount, _opSub);
  _push($.validatorTotal[valAddr], now_, amount, _opSub);
```

```
}
// ...
function _opSub(uint208 x, uint208 y) private pure returns (uint208) {
  unchecked {
    return x - y;
  }
}
```

When a user attempts to unstake more than they have staked, the `_opSub` function would cause an underflow due to the `unchecked` block, potentially allowing users to drain the vault with arbitrary unstaking amounts.

Impact

This issue could allow malicious users to unstake more tokens than they have staked, potentially draining the entire vault and affecting all users' funds.

Recommendations

Add a validation check in the `_assertUnstakeAmountCondition` function to ensure that the unstaking amount is less than or equal to the user's current staked amount.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 7be2b3ba ↗.

3.2. Slashing bypass through unbonding mechanism

Target	x/evmvalidator		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

In Mitosis, when a validator fails to secure sufficient voting power to be included in the validator set that participates in the consensus process, their `Bonded` field is set to `false`.

L101-L139 ↗

```
k.IterateLastValidatorPowers(sdkCtx, func(valAddr mitotypes.EthAddress, power
    int64) bool {
    if bondedVals[valAddr] {
        // This validator is still bonded in the active set
        return false
    }

    // This validator is no longer bonded in the active set. So we need to
    unbond it.
    validator, found := k.GetValidator(sdkCtx, valAddr)
    if !found {
        err = errors.Wrap(types.ErrValidatorNotFound, "validator not found for
    address %s [BUG]", valAddr)
        return true
    }

    // Remove from last validator powers since it's no longer active validator
    k.DeleteLastValidatorPower(sdkCtx, valAddr)

    // Set the validator as not bonded
    validator.Bonded = false
    k.SetValidator(sdkCtx, validator)

    // Append to validator updates
    abciUpdate, err2 := validator.ABCIValidatorUpdateForUnbonding()
    if err2 != nil {
        err = errors.Wrap(err2, "create abci validator update")
        return true
    }
```

```
    validatorUpdates = append(validatorUpdates, abciUpdate)

    // Log the removal
    k.Logger(sdkCtx).Info("Active Validator Set: Unbonded",
        "val_addr", valAddr.String(),
        "val_pubkey", fmt.Sprintf("%X", validator.Pubkey),
        "cons_addr_hex", fmt.Sprintf("%X", validator.MustConsAddr().Bytes()),
        "previous_power", power,
    )

    return false // continue iteration
})
```

And when the `Bonded` field is set to false, the `IsUnbonded` function returns true.

L28-L30 ↗

```
// IsUnbonded implements ValidatorI
func (v Validator) IsUnbonded() bool {
    return !v.Bonded
}
```

The problem lies in the fact that when this function returns true, the `handleEquivocationEvidence` function, which is provided by the Cosmos SDK by default for double-signing punishment, does not execute the actual slashing logic.

L36-L40 ↗

```
func (k Keeper) handleEquivocationEvidence(ctx context.Context, evidence
    *types.Equivocation) error {
    sdkCtx := sdk.UnwrapSDKContext(ctx)
    logger := k.Logger(ctx)
    consAddr :=
    evidence.GetConsensusAddress(k.stakingKeeper.ConsensusAddressCodec())

    validator, err := k.stakingKeeper.ValidatorByConsAddr(ctx, consAddr)
    if err != nil {
        return err
    }
    if validator == nil || validator.IsUnbonded() {
        // Defensive: Simulation doesn't take unbonding periods into account,
    and
        // CometBFT might break this assumption at some point.
        return nil
    }
    ...
```

```
}
```

Impact

If a malicious validator performs double signing at a specific block and then deliberately chooses to be pushed out of the voting-power ranking, they will not be included in the active validator set when the slashing evidence is submitted, resulting in no slashing despite the evidence being submitted.

Recommendations

Remove the part that calls the `IsUnbonded` function in the Cosmos SDK evidence module.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 0037f639 ↗.

3.3. First depositor inflation attack via missing `_decimalsOffset()` override in ERC-4626 vault

Target	protocol/src/hub/EOLVault.sol, protocol/src/hub/MatrixVault.sol		
Category	Coding Mistakes	**Severity**	High
Likelihood	Low	**Impact**	High

Description

There is an issue in the ERC-4626 implementation of the MitosisVault contract. The current implementation has the `_decimalsOffset()` function returning the default value of 0, making it susceptible to a first-depositor attack.

Examine Solady's ERC-4626 implementation:

```
/// @dev Override to return a non-zero value to make the inflation attack even
    more unfeasible.
/// Only used when {_useVirtualShares} returns true.
/// Default: 0.
///
/// - MUST NOT revert.
function _decimalsOffset() internal view virtual returns (uint8) {
    return _DEFAULT_DECIMALS_OFFSET;
}

/// @dev The default decimals offset.
uint8 internal constant _DEFAULT_DECIMALS_OFFSET = 0;
```

This `_decimalsOffset` function is used within the `convertToShares` function:

```
function convertToShares(uint256 assets)
    public view virtual returns (uint256 shares) {
    if (!_useVirtualShares()) {
        uint256 supply = totalSupply();
        return _eitherIsZero(assets, supply)
            ? _initialConvertToShares(assets)
            : FixedPointMathLib.fullMulDiv(assets, supply, totalAssets());
    }
    uint256 o = _decimalsOffset();
    if (o == uint256(0)) {
        return FixedPointMathLib.fullMulDiv(assets, totalSupply() + 1,
```

```
    _inc(totalAssets()));
    }
    return FixedPointMathLib.fullMulDiv(assets, totalSupply() + 10 ** o,
    _inc(totalAssets()));
}
```

This issue becomes particularly concerning when the virtual supply is only 1 and multiple deposits of similar size occur within a single block. In this scenario, an attacker who happens to be the block proposer could exploit the first-depositor bug to gain an unfair advantage.

To verify this, a proof of concept (POC) test was created that demonstrates the attack vector:

```
function test_first_deposit_attack_to_multiple_deposits() public {
    address victim1 = vm.addr(0x9708);
    address victim2 = vm.addr(0x9709);
    address victim3 = vm.addr(0x970a);
    address attacker = vm.addr(0x9999);

    vm.deal(attacker, 100 ether);
    vm.deal(victim1, 100 ether);
    vm.deal(victim2, 100 ether);
    vm.deal(victim3, 100 ether);

    vm.startPrank(attacker);
    weth.deposit{ value: 100 ether }();
    weth.approve(address(vault), type(uint256).max);
    vault.deposit(1, attacker);
    console.log("attacker share in vault : %d", vault.balanceOf(attacker));
    weth.transfer(address(vault), 10 ether);
    vm.stopPrank();

    vm.startPrank(victim1);
    weth.deposit{ value: 100 ether }();
    weth.approve(address(vault), type(uint256).max);
    vault.deposit(5 ether, victim1);
    console.log("victim1 share in vault : %d", vault.balanceOf(victim1));
    vm.stopPrank();

    vm.startPrank(victim2);
    weth.deposit{ value: 100 ether }();
    weth.approve(address(vault), type(uint256).max);

    vault.deposit(6 ether, victim2);
    console.log("victim2 share in vault : %d", vault.balanceOf(victim2));
    vm.stopPrank();
```

```
    vm.startPrank(victim3);
    weth.deposit{ value: 100 ether }();
    weth.approve(address(vault), type(uint256).max);

    vault.deposit(7 ether, victim3);
    console.log("victim3 share in vault : %d", vault.balanceOf(victim3));
    vm.stopPrank();

    vm.startPrank(attacker);
    vault.redeem(1, attacker, attacker);
    console.log("attacker balance after attack : %d",
    weth.balanceOf(attacker));
    vm.stopPrank();
}
```

As shown, the attacker deposits just 1 Wei, donates 10 ETH to the vault, and then when victims deposit 5 ETH, 6 ETH, and 7 ETH respectively, they receive zero shares:

```
attacker share in vault : 1
victim1 share in vault : 0
victim2 share in vault : 0
victim3 share in vault : 0
attacker balance after attack : 104000000000000000000
```

The attacker ultimately redeems their single share for 104 ETH, representing a 4 ETH profit at the expense of other depositors.

Impact

This issue is particularly dangerous in MEV-prone environments or when multiple transactions are included in the same block by a malicious proposer.

Recommendations

To mitigate this issue, it is recommended to override the `_decimalsOffset()` function to return a value 6–8, which is a common mitigation strategy in ERC-4626 implementations to prevent first-depositor attacks.

```
function _decimalsOffset() internal view virtual override returns (uint8) {
    return 6; // or an appropriate value between 6-8
}
```

This change would significantly strengthen the contract against first-depositor attacks. A higher

decimals' offset value normalizes the initial share ratio, making it more difficult for attackers to gain disproportionate benefits.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 65b4febd ↗.

3.4. Incorrect cap reset in `setCap` function

Target	protocol/src/hub/vault/MitosisVault.sol		
Category	Business Logic	**Severity**	Medium
Likelihood	Medium	**Impact**	Medium

Description

The MitosisVault contract has a flaw in the cap-management implementation that could lead to exceeding the intended maximum deposit cap for assets.

In the contract, `maxCap` represents an asset's maximum cap, while `availableCap` tracks the remaining capacity that can be deposited. However, the `_setCap` function incorrectly resets the `availableCap` to the new cap value without accounting for the already deposited tokens:

```
function _setCap(StorageV1 storage $, address asset, uint256 newCap)
    internal {
    AssetInfo storage assetInfo = $.assets[asset];
    uint256 prevCap = assetInfo.maxCap;
    assetInfo.maxCap = newCap;
    assetInfo.availableCap = newCap;
    emit CapSet(_msgSender(), asset, prevCap, newCap);
}
```

The `_deposit` function then subtracts the deposited amount from `availableCap`:

```
function _deposit(address asset, address to, uint256 amount)
    internal override(MitosisVaultMatrix, MitosisVaultEOL) {
    // ...
    _assertCapNotExceeded($, asset, amount);

    $.assets[asset].availableCap -= amount;
    IERC20(asset).safeTransferFrom(_msgSender(), address(this), amount);
}
```

And the deposit validation only checks against the `availableCap`, not the actual balance in relation to `maxCap`:

```
function _assertCapNotExceeded(StorageV1 storage $, address asset,
    uint256 amount) internal view {
```

```
    uint256 available = $.assets[asset].availableCap;
    require(available >= amount, IMitosisVault__ExceededCap(asset, amount,
    available));
}
```

This creates a scenario where calling `setCap` effectively resets the cap tracking, allowing more deposits than intended:

1. Set cap to 100 by calling `setCap`.
2. Deposit 50 tokens (vault holds 50 tokens, `availableCap` = 50).
3. Set new cap to 500 by calling `setCap` — `maxCap` = 500, and `availableCap` is reset to 500 (losing track of the 50 already deposited).
4. Deposit 500 more tokens (vault now holds 550 tokens). The vault now contains 550 tokens, exceeding the intended maximum of 500.

The `maxCap` effectively becomes meaningless since the contract fails to account for the current vault balance when resetting `availableCap`.

Impact

This issue allows deposits to exceed the intended maximum cap, which could lead to unexpected behavior in dependent systems and potentially violate economic assumptions of the protocol.

Recommendations

Update the `_setCap` function to properly calculate the `availableCap` by accounting for the tokens already deposited in the vault.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit d0c850d2 ↗.

3.5. Missing `availableCap` update in `withdraw` function

Target	protocol/src/branch/MitosisVault.sol		
Category	Coding Mistakes	**Severity**	Low
Likelihood	Low	**Impact**	Low

Description

There is an inconsistency in the implementation of the MitosisVault contract. In the current code, the `_deposit` function deducts from `availableCap`, but the `withdraw` function does not perform any operations with `availableCap`.

See the `_deposit` function in the MitosisVault contract: L217 ↗

```
function _deposit(address asset, address to, uint256 amount)
    internal override(MitosisVaultMatrix, MitosisVaultEOL) {
    StorageV1 storage $ = _getStorageV1();
    require(to != address(0), StdError.ZeroAddress('to'));
    require(amount != 0, StdError.ZeroAmount());

    _assertAssetInitialized(asset);
    _assertNotHalted($, asset, AssetAction.Deposit);
    _assertCapNotExceeded($, asset, amount);

    $.assets[asset].availableCap -= amount;
    IERC20(asset).safeTransferFrom(_msgSender(), address(this), amount);
}
```

This function deducts the amount from `availableCap` when depositing.

In contrast, see the `withdraw` function: L118-L127 ↗

```
function withdraw(address asset, address to, uint256 amount)
    external whenNotPaused {
    StorageV1 storage $ = _getStorageV1();

    _assertOnlyEntrypoint($);
    _assertAssetInitialized(asset);

    IERC20(asset).safeTransfer(to, amount);
```

```
    emit Withdrawn(asset, to, amount);
}
```

This function does not perform any operations with `availableCap` when withdrawing. This is not the intended behavior, and the `withdraw` function should also affect the `availableCap` state.

Impact

This inconsistency can lead to inaccurate management of the vault's `availableCap` state. Since `availableCap` decreases every time a user deposits funds but does not increase when they withdraw, over time, `availableCap` will continuously decrease.

Recommendations

The `withdraw` function should be modified to properly update `availableCap`.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit d0c850d2 ↗.

3.6. Missing instance index update in migration function

Target	protocol/src/hub/eol/EOLVaultFactory.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the EOLVaultFactory contract, the `migrate` function fails to properly update the instance index mapping for the destination vault type. While the function correctly removes the instance from the source vault type's tracking, it does not set the corresponding index in the destination's `instanceIndex` mapping:

```
function migrate(VaultType from, VaultType to, address instance,
    bytes calldata data) external onlyOwner {
    // ...
    $.infos[from].instances.pop();
    delete $.infos[from].instanceIndex[instance]; // Use delete instead of
    setting to 0

    $.infos[to].instances.push(instance);
    // Missing: $.infos[to].instanceIndex[instance] =
    $.infos[to].instances.length - 1;
    // ...
}
```

The function adds the instance to the destination's `instances` array but fails to update the `instanceIndex` mapping that would associate the instance address with its index in the array. This index is typically used for efficient instance lookup and to support operations like removal.

Impact

This is a minor code-quality issue that could potentially lead to inconsistencies in the vault factory's state tracking, though the impact is limited to administrative functionality.

Recommendations

Add the missing mapping update line to properly maintain the instance index:

```
$.infos[to].instanceIndex[instance] = $.infos[to].instances.length - 1;
```

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit b98239c3 ↗.

3.7. Uncapped withdrawals-processing configuration

Target	chain/src/app/app_config.go		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `MaxWithdrawalsPerBlock` value in the application configuration is set to 0:

```
// app/app_config.go
defaultEvmEngineConfig = EvmEngineConfig{
    ...
    MaxWithdrawalsPerBlock: 0, // In original Omni chain, this value is set to
    32
    ...
}
```

When `MaxWithdrawalsPerBlock` is set to 0, the system iterates over all entries in the withdrawal table without any limit:

```
// octane/evmengine/keeper/db.go
func (k *Keeper) EligibleWithdrawals(ctx context.Context, optimisticBuild
    bool) ([]*types.Withdrawal, error) {
    ...
    var limit int
    if optimisticBuild {
        limit = 0 // In optimistic build mode, we just build this as early as
    possible.
    } else {
        limit = k.cfg.MaxWithdrawalsPerBlock // When == 0, we include all
    withdrawals.
    }
    ...
    // Iterate over all eligible withdrawals until limit
    cnt := 0
    var withdrawals []*types.Withdrawal
    for iter.Valid() {
        ...
        withdrawals = append(withdrawals, withdrawal)
        cnt++
```

```
        if limit > 0 && cnt >= limit {
            break // Stop after reaching limit
        }
        ...
    }
    ...
}
```

This differs from the original Omni-chain implementation, which sets this value to 32.

Impact

This configuration does not pose a practical security risk.

The fallback logic in event procedures that would lead to withdrawal table entries is not triggered in normal operation due to existing validation mechanisms. Proper verification logic exists on the contract side (ValidatorManager.sol), ensuring that functions like `registerValidator` and `depositCollateral` function as expected.

This is purely a code-quality consideration rather than a security issue.

Recommendations

It is recommended to explicitly set the `MaxWithdrawalsPerBlock` parameter to an appropriate value (such as 32, as used in the original Omni chain) as a best practice.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 195c4e8d ↗.

3.8. Missing public-key validation

Target	protocol/src/lib/LibSecp256k1.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

There is an issue in Mitosis protocol's LibSecp256k1 library, where proper curve validation is not performed during the public-key decompression process. In the current LibSecp256k1 Solidity code, the `uncompressPubkey` function is implemented as follows:

```
function verifyCmpPubkey(bytes memory cmpPubkey) internal pure {
uncompressPubkey(cmpPubkey);
}
...
function uncompressPubkey(bytes memory cmpPubkey)
    internal pure returns (bytes memory uncmpPubkey) {
require(cmpPubkey.length == 33,
    StdError.InvalidParameter('cmpPubKey.length'));
require(cmpPubkey[0] == 0x02 || cmpPubkey[0] == 0x03,
    StdError.InvalidParameter('cmpPubKey[0]'));

uint8 prefix = uint8(cmpPubkey[0]);
uint256 x;
assembly {
x := mload(add(cmpPubkey, 0x21))
}
uint256 y = EllipticCurve.deriveY(prefix, x, AA, BB, PP);

uncmpPubkey = new bytes(65);
uncmpPubkey[0] = 0x04;
assembly {
mstore(add(uncmpPubkey, 0x21), x)
mstore(add(uncmpPubkey, 0x41), y)
}
return uncmpPubkey;
}
```

This function takes a compressed public key as input and performs decompression to return x, y coordinates, but it lacks a verification process to check if the resulting coordinates actually lie on

the Secp256k1 elliptic curve.

In elliptic-curve cryptography, public keys must be valid elliptic-curve points. Performing curve operations without validity checks can result in incorrect cryptographic results or potential security vulnerabilities. Fortunately, this part of the code is not directly called in the current product.

Impact

This issue opens up the possibility of bypassing elliptic-curve point validation, potentially weakening the security of logic that relies on public-key verification. Attacks using invalid curve points may become possible, which could pose risks to authentication systems or signature verification.

Recommendations

Additional validation logic should be added to the `uncompressPubkey` function to verify that the returned x, y coordinates satisfy the Secp256k1 elliptic-curve equation.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit f8d4e514 ↗.

3.9. Uninitialized return value in `addStage` function

Target	protocol/src/hub/reward/MerkleRewardDistributor.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

The `addStage` function in src/hub/reward/MerkleRewardDistributor.sol has an uninitialized return value:

```
function addStage(
    bytes32 merkleRoot,
    uint256 stage,
    uint256 nonce,
    address[] calldata rewards,
    uint256[] calldata amounts
) external onlyRole(MANAGER_ROLE) returns (uint256 merkleStage) {
    // Function implementation
    // ...

    // merkleStage is not assigned any value

    return merkleStage;
}
```

The function declares a named return variable `merkleStage`, but this variable is never assigned a value throughout the function. In Solidity, when a named return variable is not explicitly assigned, it takes its default value — which for `uint256` is zero. Therefore, the function will always return `0` regardless of the function's execution.

Impact

This is a code-quality issue that may cause confusion for callers expecting a meaningful return value, but it does not impact security or core functionality.

Recommendations

Either assign a meaningful value to `merkleStage` before returning (e.g., `merkleStage = stage;`) or remove the return value declaration if not needed.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 63a2b268 ↗.

3.10. Redelegation cooldown calculation issue

Target	protocol/src/hub/staking/ValidatorStaking.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Note: This issue was discovered and fixed by Mitosis independently during the audit period.

Description

There is an issue in the redelegation logic of the ValidatorStaking contract. In the `_checkRedelegationCooldown` function, data for a custom error is calculated even when the validation check passes, which indicates a potential underflow/overflow error.

See the original implementation using the `require` statement:

```
function _checkRedelegationCooldown(StorageV1 storage $, uint48 now_,
    address delegator, address valAddr)
  internal
  view
{
  uint256 lastRedelegationTime_ = $.lastRedelegationTime[delegator][valAddr];

  if (lastRedelegationTime_ > 0) {
    uint48 cooldown = $.redelegationCooldown;
    uint48 lasttime = lastRedelegationTime_.toUint48();
    require(
      now_ >= lasttime + cooldown, //
      IValidatorStaking__CooldownNotPassed(lasttime, now_, (lasttime
    + cooldown) - now_)
    );
  }
}
```

In this implementation, the third parameter for the `IValidatorStaking__CooldownNotPassed` error calculates `(lasttime + cooldown) - now_` even when `now_ >= lasttime + cooldown` is true. This calculation would cause an underflow when the cooldown has passed.

The issue was fixed by refactoring the code to use an if statement with revert:

```
function _checkRedelegationCooldown(StorageV1 storage $, uint48 now_,
    address delegator, address valAddr)
  internal
  view
{
  uint256 lastRedelegationTime_ = $.lastRedelegationTime[delegator][valAddr];

  if (lastRedelegationTime_ > 0) {
    uint48 cooldown = $.redelegationCooldown;
    uint48 lasttime = lastRedelegationTime_.toUint48();
    if (now_ < lasttime + cooldown) {
      revert IValidatorStaking__CooldownNotPassed(lasttime, now_, (lasttime
    + cooldown) - now_);
    }
  }
}
```

Impact

This is a minor code-quality issue that could potentially lead to unexpected reverts due to underflow when calculating the error-message parameters.

Recommendations

Consider refactoring the code to use an `if` statement that explicitly checks the condition and triggers a revert.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit `7ef4c62e` ↗.

3.11. Temporary high vote power when claiming in native token

Target	protocol/src/hub/staking/ValidatorStakingGovMITO.sol		
Category	Coding Mistakes	**Severity**	Informational
Likelihood	N/A	**Impact**	Informational

Description

In the ValidatorStakingGovMITO contract, there is a issue that allows users to temporarily gain excessive voting power when claiming unstaked tokens in native token.

The issue stems from the order of operations in the `_claimUnstake` function. When a user claims their unstaked tokens in native token, the function first transfers the ETH to the user before burning the corresponding voting units:

```
function _claimUnstake(StorageV1 storage $, address receiver)
    internal virtual returns (uint256) {
    // ...
    if (_baseAsset == NATIVE_TOKEN) receiver.safeTransferETH(claimed);
    else _baseAsset.safeTransfer(receiver, claimed);

    // apply to state
    _push($.totalUnstaking, now_, claimed.toUint208(), _opSub);
    emit UnstakeClaimed(receiver, claimed, reqIdFrom, reqIdTo);
    return claimed;
}
```

In the ValidatorStakingGovMITO contract, the `_claimUnstake` function is overridden to burn voting units after the base implementation is called:

```
function _claimUnstake(StorageV1 storage $, address receiver)
    internal override returns (uint256) {
    uint256 claimed = super._claimUnstake($, receiver);

    // burn the voting units
    _moveDelegateVotes(delegates(receiver), address(0), claimed);

    return claimed;
}
```

This creates an issue because when ETH is transferred to the user via `safeTransferETH`, it triggers

the user's fallback function. A malicious user could implement their fallback function to immediately restake the received ETH before their voting units are burned.

Here is an attack scenario:

1. User A stakes 200 ETH.
2. User A requests to unstake 200 ETH.
3. User A claims 200 ETH.
    (a) The `_claimUnstake` function is called.
    (b) User A receives 200 ETH via `safeTransferETH`.
    (c) User A's fallback function executes, staking the 200 ETH again.
    (d) User A now has 400 ETH voting power (200 from the original stake plus 200 from the new stake).
    (e) Only then `_moveDelegateVotes` is called to burn 200 ETH worth of voting power.
4. User A ends up with 200 ETH voting power again.

This has been verified with a POC test showing a temporary vote power of 400 ETH during the attack.

```
contract MaliciousContract {

   ValidatorStakingGovMITO public vault;
   address public val;
   constructor(address _vault, address _val) {
      vault = ValidatorStakingGovMITO(_vault);
      val = _val;
   }
   fallback() external payable {
      vault.stake{value: 200 ether}(val, address(this), 200 ether);
      console.log("temp vote", vault.getVotes(address(this)));
   }
}

...

function test_temporaryVoteUsingClaimCall() public {
    test_setDelegationManager();

    manager.setRet(abi.encodeCall(IValidatorManager.isValidator, (val)),
    false, abi.encode(true));

    address user1 = address(new MaliciousContract(address(vault), val));
```

```
    vm.deal(user1, 200 ether);

    vm.prank(user1);
    vault.stake{value: 200 ether}(val, user1, 200 ether);

    vm.prank(user1);
    vault.requestUnstake(val, user1, 200 ether);

    vm.prank(delegationManager);
    vault.sudoDelegate(user1, user1);

    vm.warp(_now() + 1);
    assertEq(vault.getVotes(user1), 200 ether);
    assertEq(vault.getPastTotalSupply(_now() - 1), 0);

    // try to claim

    vm.warp(_now() + UNSTAKING_COOLDOWN - 1);
    uint256 claimed = vault.claimUnstake(user1);
    console.log("final vote", vault.getVotes(user1));
}
```

Impact

This issue allows users to temporarily increase voting power, which could be exploited in a time-sensitive governance vote to have a larger influence than their actual stake would allow.

Recommendations

Reorder the operations in the `_claimUnstake` function to first burn the voting units and then transfer the ETH to the user:

```
function _claimUnstake(StorageV1 storage $, address receiver)
    internal override returns (uint256) {
    uint256 claimed = super._claimUnstake($, receiver);

    // First burn the voting units
    _moveDelegateVotes(delegates(receiver), address(0), claimed);

    // Then transfer ETH to the user
    if (_baseAsset == NATIVE_TOKEN) receiver.safeTransferETH(claimed);
    else _baseAsset.safeTransfer(receiver, claimed);

    return claimed;
```

```
}
```

Alternatively, implement a reentrancy guard to prevent the exploit.

Remediation

This issue has been acknowledged by Mitosis, and a fix was implemented in commit 0ffd1608 ↗.

4. Discussion

The purpose of this section is to document miscellaneous observations that we made during the assessment. These discussion notes are not necessarily security related and do not convey that we are suggesting a code change.

4.1. Validator slashing-mechanism perspective

According to the Mitosis team's explanation, this project intentionally excludes delegation from slashing targets. The reason is that by using the maximum leverage ratio (`max_leverage_ratio`), they prevent delegations with excessive voting power from allocating too much voting power to validators. They explain that if this value is properly set, it would be impossible for malicious validators to evade slashing risk by depositing only minimal collateral while acquiring the rest through delegation.

We were concerned that delegation could become a means for malicious validators to evade slashing since delegations are not penalized in slashing scenarios. However, after hearing the Mitosis team's explanation and reviewing the code, we agree with the Mitosis team's opinion.

5. System Design

This provides a description of the high-level components of the system and how they interact, including details like a function's externally controllable inputs and how an attacker could leverage each input to cause harm or which invariants or constraints of the system are critical and must always be upheld.

Not all components in the audit scope may have been modeled. The absence of a component in this section does not necessarily suggest that it is safe.

5.1. Module: evmengine

Description

The evmengine module is provided as part of Omni Network's Octane framework for the purpose of integration with the EVM execution layer.

It interacts with the evmgov and evmvalidator modules that exist within the project.

Unlike the Ethermint project, this module aims to achieve separation between the consensus layer and execution layer by utilizing the Engine API of the execution layer.

Messages

Message: `MsgExecutionPayload`

This message handler is only executed in `Finalize` mode where actual state transition occurs.

For the message received as a parameter, this message handler enforces the following elements:

- Only one of the `ExecutionPayloadDeneb` and `ExecutionPayload` fields must be used to transfer the payload to the execution layer.
- The list of withdrawals in the proposed payload must match the list of eligible withdrawals.
- The `ExecutionWitness` field should be empty as it is not used.
- The fee-recipient address must be valid and approved. (However, the code to check this has not been implemented yet.)
- Block continuity must be maintained. That is, it must have the previous block hash as the parent block hash, and the current block number must be 1 greater than the previous block number.
- Block timestamp must be after the most recently executed block and before the current consensus block time.
- The RandAO value (random field) must match the parent block hash.

Subsequently, it calls the Engine API's `engine_newPayloadV4` and `engine_forkchoiceUpdatedV3` with the following conditions and roles:

- `engine_newPayloadV4`

- It uses a lock to prevent multiple `engine_newPayloadV4` endpoint calls from being processed simultaneously.
- The passed `ExecutableData` must be convertible to a valid block.
- Blocks that have already been processed once are not processed again and return a `VALID` status.
- It checks if the block is connected to a previously rejected chain, and if so, increases the hit counter of that block. If it exceeds the threshold, it removes it from the list to give a reprocessing opportunity; otherwise, it also adds the current head to the invalid list. Finally, it sets the parent of the invalid block as the last valid hash (with proof-of-work terminal block exception handling) and returns an "INVALID" status response to block propagation of the invalid chain.
- It verifies that the parent block of the current block to be executed already exists locally and checks if the timestamp of the current block is greater than that of the parent block.
- It checks if the execution-layer client is in `FullSync` state. (If synchronizing, it returns a `SYNCING` status.)
- After all these checks, it creates a block based on the inputs received and connects it to the chain (`InsertBlockWithoutSetHead`).
- If the execution layer returns an unexpected error or `INVALID` status when calling this Engine API endpoint, it immediately terminates the message handler, but if the execution layer is still attempting to sync, it repeatedly calls that Engine API endpoint.

- `engine_forkchoiceUpdatedV3`
  - The received `Withdrawals` and `BeaconRoot` must not be empty.
  - The timestamp of the received block must be later than the cancun and prague fork points specified in the configuration.
  - The received `HeadBlockHash` must not be a zero hash, and block data matching the block hash must be stored locally or, at least, the block header must be stored in the remote block cache.
  - Block data corresponding to `FinalizedBlockHash` and `SafeBlockHash` must be stored locally and be part of the canonical chain.
  - After these verifications are complete, chain reorganization is performed if the designated head is not the head of the canonical chain.
  - It selects one of the determined statuses (`VALID`, `SYNCING`, `INVALID`) and returns a response.
  - If the execution layer returns an unexpected error, returns an `INVALID` status, or is still attempting to sync when calling this Engine API endpoint, the message handler is immediately terminated.

When the Engine API call to the execution layer is completed, it retrieves the executed events based on the block hash of the execution layer specified in the payload. If there are events defined in advance within the module among those events, they are passed to the evmgov module and the evmvalidator module respectively, and appropriate message handlers are called.

When all these processes are completed, it removes the processed withdrawals and updates the latest block information of the execution layer within the consensus layer.

ABCI++ handlers

Handler: `PostFinalize`

This is a handler that is called when the current block is finalized and the proposer of the next block is oneself.

It starts building the payload to be submitted to the execution layer at the time of block-proposal generation. It retrieves the latest information at the time the current block is finalized, sets the appropriate timestamp, and sets the `Head`, `Safe`, and `Finalize` block hashes in the execution layer, respectively.

It also configures a payload that includes the list of withdrawals to be processed, Beacon root (app hash), and fee-recipient information, and it calls the `engine_forkchoiceUpdatedV3` among the Engine API endpoints located in the execution layer.

Handler: `PrepareProposal`

This is a handler for preparing the Block to execute when a validator is selected as a block proposer during the Tendermint consensus process.

If preparation exceeds a 10-second time-out, it will propose an empty block.

There should be no transactions arbitrarily inserted by the block proposer. (The `ProcessProposal` handler that receives the block proposal also thoroughly checks this part to restrict the actions of malicious block proposers.)

It checks if the maximum transaction bytes' size is more than 90MB. (The figure of 90MB may vary depending on one of the configurations of cometBFT, `MaxBlockSizeBytes`.)

If the payload that was built previously in the `PostFinalize` handler is not based on the current block height with the latest block information at the time this handler is executed, it rebuilds the payload to be submitted to the execution layer.

If there is no issue with the built payload, it composes a `MsgExecutePayload` message based on the payload, inserts it as the only transaction in the block, and completes preparations to broadcast the block. (Unlike the original Omni Network code, `VoteExtension` is not used separately, so an explanation of that part is not included.)

Handler: `ProcessProposal`

This handler verifies that there is indeed only one transaction included in the block proposal received from the block proposer.

Also, to verify that the previous block has been committed, it checks the vote information of the validators to confirm that more than two thirds of the total voting power has been secured. And it iterates through the transactions included in the block proposal, checks if they are allowed messages, and if so, calls the appropriate message handler. In the current code, the only allowed message is `MsgExecutePayload`, and it is restricted to be executed only once per block.

5.2. Module: evmgov

Description

The evmgov module interacts with EVM-based governance entry points to convert and execute governance messages (software upgrades, parameter changes, etc.) originating from the EVM into Cosmos SDK messages.

In summary, the functions performed by the evmgov module can be outlined as follows:

- Receiving EVM events (the `MsgExecute` event from ConsensusGovernanceEntrypoint)
- Converting messages extracted from EVM events into Cosmos SDK type messages then executing the relevant message handler to perform the actual governance action

Event processing

The evmgov module's event handler processes events emitted by the ConsensusGovernanceEntrypoint contract on the EVM and performs the following key functions:

Event filtering. The `FilterParams` function only processes events from a specific contract address (`govEntrypointContractAddr`) and with a specific event ID (`MsgExecute`). It improves system efficiency by preventing unnecessary event processing.

Event delivery and handling. The `Deliver` function is the main function that receives and handles EVM log events. It uses a cached context to ensure atomicity of state changes and contains logic to either ignore or propagate errors based on their severity.

EVM event-execution processing. The `processMsgExecute` function executes governance messages passed from the EVM. Each message goes through the following steps.

- `ParseMessage`: This parses the incoming message. The incoming message must comply with the Cosmos SDK message type.
- `ExecuteMessage`: This executes the parsed message. The predefined message handler will be executed.

5.3. Module: evmvalidator

Description

The evmvalidator module receives EVM events from the execution layer and handles core validator-related logic, including validator set management, slashing evidence management, and stake withdrawal.

It aims to deactivate some code in the staking module provided by default in the Cosmos SDK and to replace those functionalities.

Messages

Message: `MsgUpdateValidatorEntrypointContractAddr`

The `MsgUpdateParams` message updates the `ValidatorEntrypointContract` address. It can only be executed through a governance proposal.

Message: `MsgUpdateParams`

The `MsgUpdateParams` message updates the evmvalidator module's parameters. It can only be executed through a governance proposal.

Event processing

The evmvalidator module's event handler processes events emitted by the ConsensusValidatorEntrypoint contract on the EVM and performs the following key functions:

Event filtering. The `FilterParams` function only processes events from a specific contract address (`govEntrypointContractAddr`) and with a specific event ID (`MsgExecute`). It improves system efficiency by preventing unnecessary event processing.

Event delivery and handling. The `Deliver` function is the main function that receives and handles EVM log events. It uses a cached context to ensure atomicity of state changes and contains logic to either ignore or propagate errors based on their severity.

EVM event-execution processing. This involves the following functions.

- `processRegisterValidator`: This registers a new validator to the consensus layer and sets initial collateral and voting power.
- `processDepositCollateral`: This increases a validator's collateral and voting power.
- `processWithdrawCollateral`: This decreases a validator's collateral and voting power. The withdrawn balance is reflected to the execution layer after a certain period.
- `processTransferCollateralOwnership`: This transfers collateral ownership from one address to another, changing the ownership structure.

- `processUnjail`: This reactivates a jailed validator to participate in the consensus process again.
- `processUpdateExtraVotingPower`: This reflects the voting power from delegations occurring in the execution layer to the validator's voting power in the consensus layer.

The staking module vs evmvalidator module

This section outlines the differences between the original staking module and the evmvalidator module.

- **Removal of the delegation system.** In Cosmos SDK, delegators delegate tokens to validators. In EVMValidator, delegators also delegate tokens to validators, but this does not use the Cosmos SDK's staking module. Instead, a smart contract on the execution layer handles delegation functionality. When a delegation is created in the contract, an event is emitted and relayed to the consensus layer, where it is added to the validator's voting power as `ExtraVotingPower`.
- **Simplification of the slashing mechanism.** In Cosmos SDK, slashing applies not only to the validator's self-stake but also to the delegated stake from delegators. In EVMValidator, slashing applies only to the validator's own bonded tokens. However, limiting slashing to only the validator's stake does not allow malicious validators to abuse delegation as a way to avoid slashing. This is enforced by the `max_leverage_ratio` parameter.

ABCI++ handlers

Handler: `EndBlocker`

The `EndBlocker` of the evmvalidator module processes matured withdrawal requests. During this process, it handles only as many requests as defined by the `WithdrawalLimit` parameter, which sets the maximum number of requests processed per block. For each withdrawal request, it calls the `InsertWithdrawal` function of the evmengine module to update the recipient's balance in the execution layer through the Engine API. Once processed, withdrawal requests are removed from the state.

After completing all withdrawal-request processing, the `EndBlocker` retrieves a validator list sorted by current power, limited to the maximum number defined by the `MaxValidators` parameter. This list excludes validators in the jail state. Each validator's power is recalculated when changes occur to their collateral balance (when calling functions like `DepositCollateral`, `RegisterValidator`, `Slash_`, `UpdateExtraVotingPower`, or `WithdrawCollateral`) or when module parameters related to voting power are updated (when calling `UpdateParams`).

The `EndBlocker` then iterates through this validator list and compares it with the previous block. If a validator was included in the consensus participation ranking in the previous block but is excluded in the current block, they are removed from the list of validators participating in the next consensus process. Conversely, if a validator was not ranked in the previous block but has newly entered the ranking in the current block, they are included in the list for participating in the

consensus process of the next block. In this case, the `AfterValidatorBonded` hook of the slashing module is called to update the `ValidatorSigningInfo` data that records history related to signing during the consensus process.

5.4. Contracts: MitosisVault, MitosisVaultEOL, and MitosisVaultMatrix

Description

MitosisVault is a unified, upgradable vault proxy combining Matrix and EOL strategies. It enforces per-asset caps, pause/halt controls, and entry point–restricted withdrawals. It is secured via owner-managed beacon proxy, pause, and reentrancy guards.

MitosisVaultEOL is the core implementation for EOL vaults, linking deposits and initialization to the branch-chain entry point. It uses ERC-7201 for action namespacing and two-step ownership with pausable safety. It records per-vault initialization and halt status to block unauthorized operations.

MitosisVaultMatrix is the core implementation for Matrix vaults, handling deposits, initialization, and liquidity coordination via entry point. It applies pausability, two-step ownership, and reentrancy protection. It maintains available liquidity and strategy executor settings per vault, emitting state-change events.

Core functions and invariants

Function: `initializeAsset(address asset)`

- Caller must be the entry point.
- Check `asset` is already initialized.
- Set `assets[asset].initialized = true` and halt deposits for that asset.
- Emit `AssetInitialized(asset)`.

Function: `deposit(address asset, address to, uint256 amount)`

- Check `to` must be nonzero and `amount > 0`.
- Asset must be initialized and not halted for deposit.
- Check that available cap for `asset` decreases by `amount`.
- Transfer `amount` of `asset` from the sender to the vault.
- Call entry point `deposit(asset, to, amount)` and emit `Deposited(asset, to, amount)`.

Function: `withdraw(address asset, address to, uint256 amount)`

- Caller must be the entry point.
- Check `asset` is initialized.
- Transfer `amount` of `asset` from the vault to `to`.
- Emit `Withdrawn(asset, to, amount)`.

Function: `setEntrypoint(address entrypoint_)`

- Caller must be the owner of the contract.
- Update entry point to `entrypoint_`.
- Emit `EntrypointSet(entrypoint_)`.

Function: `setCap(address asset, uint256 newCap)`

- Caller must be the owner of the contract.
- Check `asset` is initialized.
- Update `assets[asset].maxCap` and `assets[asset].availableCap` to `newCap`.
- Emit `CapSet(caller, asset, previousCap, newCap)`.

Function: `haltAsset(address asset, AssetAction action)`

- Caller must be the owner of the contract.
- Check `asset` is initialized.
- Mark `assets[asset].isHalted[action] = true`.
- Emit `AssetHalted(asset, action)`.

Function: `resumeAsset(address asset, AssetAction action)`

- Caller must be the owner of the contract.
- Check `asset` is initialized.
- Mark `assets[asset].isHalted[action] = false`.
- Emit `AssetResumed(asset, action)`.

EOL functions and invariants

Function: `depositWithSupplyEOL(address asset, address to, address hubEOLVault, uint256 amount)`

- Invoke `_deposit(asset, to, amount)` and execute before any state checks.
- Vault `hubEOLVault` must be initialized and accept this `asset`.
- Forward call to `IMitosisVaultEntrypoint.depositWithSupplyEOL(asset, to, hubEOLVault, amount)`.
- Emit `EOLDepositedWithSupply(asset, to, hubEOLVault, amount)`.

Function: `initializeEOL(address hubEOLVault, address asset)`

- Caller must be the entry point.
- Vault `hubEOLVault` must not already be initialized.
- Underlying `asset` must already be initialized (via `_assertAssetInitialized`).
- Mark `eols[hubEOLVault].initialized = true` and set `eols[hubEOLVault].asset = asset`.
- Emit `EOLInitialized(hubEOLVault, asset)`.

Matrix functions and invariants

Function: `depositWithSupplyMatrix(address asset, address to, address hubMatrixVault, uint256 amount)`

- Invoke `_deposit(asset, to, amount)` and execute before checks.
- Vault `hubMatrixVault` must be initialized and match `asset`.
- Call `IMitosisVaultEntrypoint.depositWithSupplyMatrix(asset, to, hubMatrixVault, amount)`.
- Emit `MatrixDepositedWithSupply(asset, to, hubMatrixVault, amount)`.

Function: `initializeMatrix(address hubMatrixVault, address asset)`

- Caller must be the entry point.
- Vault must not already be initialized; `asset` must be initialized externally.
- Set `.initialized = true` and `.asset = asset`.
- Emit `MatrixInitialized(hubMatrixVault, asset)`.

Function: `allocateMatrix(address hubMatrixVault, uint256 amount)`

- Caller must be the entry point.
- Vault must be initialized.
- Increment `availableLiquidity` by `amount`.
- Emit `MatrixAllocated(hubMatrixVault, amount)`.

Function: `deallocateMatrix(address hubMatrixVault, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized.
- Decrement `availableLiquidity` by `amount`.
- Call `IMitosisVaultEntrypoint.deallocateMatrix(hubMatrixVault, amount)`.
- Emit `MatrixDeallocated(hubMatrixVault, amount)`.

Function: `fetchMatrix(address hubMatrixVault, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized and fetch not halted.
- Decrement `availableLiquidity` by `amount`.
- Transfer `amount` of underlying `asset` to `strategyExecutor`.
- Emit `MatrixFetched(hubMatrixVault, amount)`.

Function: `returnMatrix(address hubMatrixVault, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized.
- Increment `availableLiquidity` by `amount`.
- Transfer `amount` of underlying `asset` from `strategyExecutor`.

- Emit `MatrixReturned(hubMatrixVault, amount)`.

Function: `settleMatrixYield(address hubMatrixVault, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized.
- Call `IMitosisVaultEntrypoint.settleMatrixYield(hubMatrixVault, amount)`.
- Emit `MatrixYieldSettled(hubMatrixVault, amount)`.

Function: `settleMatrixLoss(address hubMatrixVault, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized.
- Call `IMitosisVaultEntrypoint.settleMatrixLoss(hubMatrixVault, amount)`.
- Emit `MatrixLossSettled(hubMatrixVault, amount)`.

Function: `settleMatrixExtraRewards(address hubMatrixVault, address reward, uint256 amount)`

- Caller must be `strategyExecutor`.
- Vault must be initialized.
- The `reward` must be initialized and different from the vault's `asset`.
- Transfer `amount` of `reward` from caller.
- Call `IMitosisVaultEntrypoint.settleMatrixExtraRewards(hubMatrixVault, reward, amount)`.
- Emit `MatrixExtraRewardsSettled(hubMatrixVault, reward, amount)`.

Function: `haltMatrix(address hubMatrixVault, MatrixAction action)`

- Caller must be the owner of the contract.
- Vault must be initialized.
- Set `isHalted[action] = true`.
- Emit `MatrixHalted(hubMatrixVault, action)`.

Function: `resumeMatrix(address hubMatrixVault, MatrixAction action)`

- Caller must be the owner of the contract.
- Vault must be initialized.
- Set `isHalted[action] = false`.
- Emit `MatrixResumed(hubMatrixVault, action)`.

Function: `setMatrixStrategyExecutor(address hubMatrixVault, address strategyExecutor_)`

- Caller must be the owner of the contract.
- Vault must be initialized.
- If a previous executor exists, it must be drained.
- The `strategyExecutor_.hubMatrixVault()`, `.vault()`, and `.asset()` must match.

- Update `strategyExecutor`.
- Emit `MatrixStrategyExecutorSet(hubMatrixVault, strategyExecutor_)`.

Test coverage

- **Initialization**
  - `setUp` deploys proxy and initializes vault, entry point, and token/Matrix mocks.
  - Verify that `isAssetInitialized` and `isMatrixInitialized` default to `false`.
- **`initializeAsset`**
  - `test_initializeAsset` authorizes entry point to initialize asset.
  - `test_initializeAsset_Unauthorized` rejects non–entry-point and owner calls.
  - `test_initializeAsset_AssetAlreadyInitialized` rejects reinitialization.
- **`deposit`**
  - `test_deposit` covers happy-path — initialize, set cap, resume, approve and deposit, and balance changes.
  - `test_setCap_IncorrectCap` shows cap adjustment preserves previously deposited amounts.
  - `test_deposit_AssetNotInitialized` rejects deposit before initialization.
  - `test_deposit_AssetHalted` rejects when deposit is halted.
  - `test_deposit_ZeroAddress` rejects zero-address receiver.
  - `test_deposit_ZeroAmount` rejects zero amount.
- **`depositWithSupplyMatrix`**
  - `test_depositWithSupplyMatrix` covers Matrix flow after initialize and resume.
  - `test_depositWithSupplyMatrix_AssetNotInitialized` rejects if asset is not initialized.
  - `test_depositWithSupplyMatrix_AssetHalted` rejects when halted.
  - `test_depositWithSupplyMatrix_MatrixNotInitialized` rejects if Matrix vault is not initialized.
  - `test_depositWithSupplyMatrix_ZeroAddress` and `test_depositWithSupplyMatrix_ZeroAmount` reject invalid inputs.
- **`withdraw`**
  - `test_withdraw` covers happy-path via entry point and checks vault and recipient balances.
  - `test_withdraw_Unauthorized` rejects non–entry-point calls.
  - `test_withdraw_AssetNotInitialized` rejects uninitialized asset.
  - `test_withdraw_NotEnoughBalance` rejects overwithdraw.
- **`initializeMatrix`**
  - `test_initializeMatrix` authorizes entry point.
  - `test_initializeMatrix_Unauthorized` rejects unauthorized calls.
  - `test_initializeMatrix_MatrixAlreadyInitialized` rejects double init.

  - `test_initializeMatrix_AssetNotInitialized` rejects if asset is not initialized.
- **`allocateMatrix` / `deallocateMatrix`**
  - `test_allocateMatrix` covers entry-point allocation and `availableMatrix`.
  - `test_allocateMatrix_Unauthorized` and `test_allocateMatrix_MatrixNotInitialized` reject invalid calls.
  - `test_deallocateMatrix` covers strategy executor-draining behavior.
  - `test_deallocateMatrix_Unauthorized` and `test_deallocateMatrix_InsufficientMatrix` reject invalid deallocations.
- **`fetchMatrix`**
  - `test_fetchMatrix` covers executor fetch and token transfer.
  - `test_fetchMatrix_Unauthorized`, `test_fetchMatrix_AssetHalted`, and `test_fetchMatrix_InsufficientMatrix` reject invalid conditions.
- **`returnMatrix`**
  - `test_returnMatrix` covers executor return and cap restoration.
  - `test_returnMatrix_Unauthorized` rejects nonexecutor calls.
- **`settleMatrixYield` / `settleMatrixLoss` / `settleMatrixExtraRewards`**
  - `test_settleMatrixYield` and `test_settleMatrixLoss` cover entry point via executor calls.
  - `test_settleMatrixExtraRewards` covers full reward token flow.
  - Corresponding `_Unauthorized`, `_NotInitialized`, and `_InvalidRewardAddress` tests reject invalid conditions.
- **`setEntrypoint` / `setMatrixStrategyExecutor`**
  - `test_setEntrypoint_Unauthorized` rejects non-owner.
  - `test_setMatrixStrategyExecutor` verifies valid assignment.
  - `test_setMatrixStrategyExecutor_MatrixNotInitialized`, `_NotDrained`, and invalid-address variants reject improper executor updates.
- **Action halts**
  - `test_isMatrixActionHalted` toggles and reads Matrix action halt flag.
  - `test_isAssetActionHalted` toggles and reads asset action halt flag.

5.5. Contract: MitosisVaultEntrypoint

Description

MitosisVaultEntrypoint is the EVM entry point for branch-chain vaults, forwarding vault actions into cross-chain Hyperlane messages. It restricts calls to a single trusted vault contract and a specific Hyperlane domain/address. It manages router enrollment, gas quoting/dispatch, and UUPS upgrades under owner control.

Core functions and invariants]

Function: `deposit(address asset, address to, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgDeposit` correctly and call `_dispatchToMitosis`.

Function: `depositWithSupplyMatrix(address asset, address to, address hubMatrixVault, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgDepositWithSupplyMatrix` correctly and call `_dispatchToMitosis`.

Function: `depositWithSupplyEOL(address asset, address to, address hubEOLVault, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgDepositWithSupplyEOL` correctly and call `_dispatchToMitosis`.

Function: `deallocateMatrix(address hubMatrixVault, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgDeallocateMatrix` correctly and call `_dispatchToMitosis`.

Function: `settleMatrixYield(address hubMatrixVault, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgSettleMatrixYield` correctly and call `_dispatchToMitosis`.

Function: `settleMatrixLoss(address hubMatrixVault, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgSettleMatrixLoss` correctly and call `_dispatchToMitosis`.

Function: `settleMatrixExtraRewards(address hubMatrixVault, address reward, uint256 amount)`

- Caller must be the trusted `_vault`.
- Encode `MsgSettleMatrixExtraRewards` correctly and call `_dispatchToMitosis`.

Function: `_dispatchToMitosis(bytes memory enc)`

- Compute `fee` via `_GasRouter_quoteDispatch` for the configured domain.
- Call `_GasRouter_dispatch` with exact `fee`, `enc`, and mailbox hook.

Function: `_handle(uint32 origin, bytes32 sender, bytes calldata msg_)`

- Check `origin` and `sender` match the configured domain and address.
- Switch on `msg_.msgType()`, and for each `MsgType`, decode the correct `Msg...` struct and invoke the corresponding function on `_vault` with decoded parameters.
- Do not modify entry-point storage.

Test coverage

- **Initial ownership state**
  - `test_ownershipInitialState` verifies `owner()` equals deployer and `pendingOwner()` is zero.
- **Two-step ownership transfer**
  - `test_transferOwnership_twoStep` covers initiating transfer (`pendingOwner` set) and acceptance by new owner (owner updated, `pendingOwner` cleared).
- **Unauthorized transfer attempts**
  - `test_nonOwnerCannotTransferOwnership` asserts revert when a non-owner calls `transferOwnership`.
  - `test_nonPendingOwnerCannotAcceptOwnership` asserts revert when someone other than the pending owner calls `acceptOwnership`.
- **Cancellation of pending transfer**
  - `test_ownerCanCancelTransferOwnership` covers owner resetting `pendingOwner` to zero by transferring to `address(0)`.
- **Multiple successive transfers**
  - `test_ownerCanTransferOwnershipMultipleTimes` covers owner changing `pendingOwner` twice before acceptance and final ownership change.
- **Edge-case non-owner call**
  - `test_transferOwnership_with_NotOwner` attempts transfer from non-owner without explicit revert check (ensures no unintended state change).

5.6. Contract: GovernanceEntrypoint

Description

GovernanceEntrypoint is the EVM entry point on branch chains for handling governance proposals dispatched from the hub via Hyperlane. It verifies origin domain and sender, decodes governance execution messages, and schedules execution through a TimelockController. It manages Hyperlane router enrollment and UUPS upgrades under access-control safeguards.

Core functions and invariants

Function: `_handle(uint32 origin, bytes32 sender, bytes calldata msg_) internal override`

- Revert if `origin != _mitosisDomain` or `sender != _mitosisAddr`.
- Determine `msgType` via `msg_.msgType()`.
- On `MsgDispatchGovernanceExecution`, decode into `MsgDispatchGovernanceExecution` struct.
- Schedule a batched Timelock execution via `_timelock.scheduleBatch(...)` with correct targets, values, data, predecessor, salt, and minimum delay.

- Do not allow any other message types.

Function: `_convertBytes32ArrayToAddressArray(bytes32[] memory targets) internal pure`

- Convert each `bytes32` element to `address` via `toAddress()`.
- Return an array of matching length with no data corruption.

Test coverage

N/A.

5.7. Contract: MatrixStrategyExecutor

Description

MatrixStrategyExecutor executes strategy actions for Matrix vaults by coordinating with a hub vault and on-chain tally. It restricts liquidity management to a designated strategist and contract calls to a designated executor. It safeguards interactions with reentrancy guards and two-step ownership upgradability.

Core functions and invariants

Function: `deallocateLiquidity(uint256 amount)`

- Check `amount > 0`.
- Caller must equal the stored `strategist`.
- Invoke `vault.deallocateMatrix(hubMatrixVault, amount)`.

Function: `fetchLiquidity(uint256 amount)`

- Check `amount > 0`.
- Caller must equal the stored `strategist`.
- Call `vault.fetchMatrix(hubMatrixVault, amount)`.
- Increment `storedTotalBalance` by `amount`.

Function: `returnLiquidity(uint256 amount)`

- Check `amount > 0`.
- Caller must equal the stored `strategist`.
- Approve `vault` to transfer `amount` of `asset`.
- Call `vault.returnMatrix(hubMatrixVault, amount)`.
- Decrement `storedTotalBalance` by `amount`.

Function: `settle()`

- Caller must equal the stored `strategist`.

- Compute `totalBalance = asset.balanceOf(this) + tally balances`.
- If `totalBalance >= storedTotalBalance`, call `vault.settleMatrixYield(hubMatrixVault, totalBalance - storedTotalBalance)`. Otherwise, call `vault.settleMatrixLoss(hubMatrixVault, storedTotalBalance - totalBalance)`.
- Update `storedTotalBalance` to `totalBalance`.

Function: `settleExtraRewards(address reward, uint256 amount)`

- Check `amount > 0`.
- Caller must equal the stored `strategist`.
- Check `reward != address(asset)`.
- Approve `vault` to transfer `amount` of `reward`.
- Call `vault.settleMatrixExtraRewards(hubMatrixVault, reward, amount)`.

Function: `execute(address target, bytes calldata data, uint256 value) → bytes memory`

- Caller must equal the stored `executor`.
- Check `tally.protocolAddress() == target`.
- Forward `data` and `value` via low-level call and return the result.

Function: `execute(address[] calldata targets, bytes[] calldata data, uint256[] calldata values) → bytes[] memory`

- Caller must equal the stored `executor`.
- Check `targets.length == data.length == values.length`.
- Check each `targets[i]` matches `tally.protocolAddress()`.
- Return an array of call results.

Function: `setTally(address implementation)`

- Caller must be the owner of the contract.
- Check `implementation.code.length > 0`.
- Check either `tally` was unset or `_tallyTotalBalance` is zero.
- Update `tally` and emit `TallySet(implementation)`.

Function: `setStrategist(address strategist_)`

- Caller must be the owner of the contract.
- Check `strategist_ != address(0)`.
- Update `strategist` and emit `StrategistSet(strategist_)`.

Function: `setExecutor(address executor_)`

- Caller must be the owner of the contract.
- Check `executor_ != address(0)`.
- Update `executor` and emit `ExecutorSet(executor_)`.

Function: `unsetStrategist()`

- Caller must be the owner of the contract.
- Clear `strategist` to `address(0)` and emit `StrategistSet(address(0))`.

Function: `unsetExecutor()`

- Caller must be the owner of the contract.
- Clear `executor` to `address(0)` and emit `ExecutorSet(address(0))`.

Test coverage

- **Happy-path execution**
  - `test_execute` verifies that when `executor` and `tally` are properly set, calling `execute` transfers assets, calls the vault correctly, and updates `storedTotalBalance`.
- **Unauthorized executor**
  - `test_execute_InvalidAddress_executor` asserts that a call from any address other than the configured `executor` reverts with `Unauthorized`.
- **Tally not set**
  - `test_execute_InvalidAddress_TallyNotSet` asserts that calling `execute` without a configured `tally` address reverts with `IMatrixStrategyExecutor__TallyNotSet`.

5.8. Contract: GovMITO

Description

GovMITO is the contract for the ERC-20–based governance token (gMITO) with integrated voting (ERC-6372) and time-delayed withdrawals via a queue.

It supports on-chain and off-chain delegation through `delegate` and `delegateBySig`, leveraging `SudoVotes` for multisource vote aggregation. The minter address can mint by sending ETH, and users queue withdrawals, which unlock after a configurable period. It includes owner-only controls over minter, module access, whitelisted senders, and withdrawal period.

Core functions and invariants

Function: `delegate(address delegatee)`

- Override `IVotes` and `SudoVotes` to call combined `super.delegate`.
- Update vote delegation for `msg.sender` to `delegatee`.
- Emit `DelegateChanged` and `DelegateVotesChanged` via base contracts.

Function: `delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v,`

bytes32 r, bytes32 s)

- Validate signature (`expiry ≥ block.timestamp`) and `nonce` via `NoncesUpgradeable`.
- Recover `signer` and call `super.delegateBySig`.
- Update delegation and emit vote events as in `delegate`.

Function: `mint(address to) payable`

- Caller must be the minter role.
- Check `msg.value > 0`.
- Mint `msg.value` gMITO to `to`.
- Emit `Minted(to, msg.value)`.

Function: `requestWithdraw(address receiver, uint256 amount)`

- Check `receiver != address(0)` and `amount > 0`.
- Burn `amount` from `msg.sender`.
- Append a queue entry with `(clock(), amount)`.
- Emit `WithdrawRequested(msg.sender, receiver, amount, requestId)`.
- Return the generated `requestId`.

Function: `claimWithdraw(address receiver)`

- Compute `claimed` to equal the sum of all queue entries older than `clock() - withdrawalPeriod`.
- Transfer `claimed` ETH to `receiver`.
- Emit `WithdrawRequestClaimed(receiver, claimed, fromRequestId, toRequestId)`.
- Return `claimed`.

Function: `setMinter(address minter_)`

- Caller must be the owner of the contract.
- Update `minter` in storage.
- Emit `MinterSet(minter_)`.

Function: `setModule(address addr, bool isModule_)`

- Caller must be the owner of the contract.
- Check `addr != address(0)`.
- Set module flag mapping.
- Emit `ModuleSet(addr, isModule_)`.

Function: `setWhitelistedSender(address sender, bool isWhitelisted)`

- Caller must be the owner of the contract.
- Check `sender != address(0)`.
- Update `isWhitelistedSender[sender]`.
- Emit `WhitelistedSenderSet(sender, isWhitelisted)`.

Function: `setWithdrawalPeriod(uint256 withdrawalPeriod_)`

- Caller must be the owner of the contract.
- Check `withdrawalPeriod_ > 0`.
- Update `withdrawalPeriod`.
- Emit `WithdrawalPeriodSet(withdrawalPeriod_)`.

Test coverage

- **Initialization**
  - `test_init` verifies name, symbol, decimals, owner, minter, `delegationManager` default, and `withdrawalPeriod`.
- **Minting**
  - `test_mint` mints gMITO when called by the designated minter.
  - `test_mint_NotMinter` reverts for unauthorized callers.
- **Withdraw flow**
  - `test_withdraw_basic` covers `requestWithdraw`, `previewClaimWithdraw` before and after the period, and `claimWithdraw`.
  - `test_withdraw_requestTwiceAndClaimOnce` and `test_withdraw_requestTwiceAndClaimTwice` cover multiple withdraw requests and successive claims.
  - `test_withdraw_requestAfterClaimable` tests new withdraw requests after claimable period.
  - `test_withdraw_severalUsers` tests parallel withdraws and claims for two users.
  - `test_withdraw_differentReceiver` verifies claiming to a different receiver address.
  - `test_withdraw_anyoneCanClaim` confirms anyone can invoke `claimWithdraw` for a user.
  - `test_withdraw_ERC20InsufficientBalance` reverts when requesting more than the user's balance.
- **Whitelist behavior**
  - `test_whiteListedSender` covers transfers and approvals by whitelisted senders.
  - `test_whiteListedSender_NotWhitelisted` reverts transfers and approvals for nonwhitelisted senders.
- **Delegation controls**
  - `test_delegate` and `test_delegateBySig` revert for unsupported delegation methods.
  - `test_setDelegationManager` reverts for non-owner and emits `DelegationManagerSet` on success.
  - `test_sudoDelegate` reverts for unauthorized callers and emits `DelegateChanged` / `DelegateVotesChanged` when called by the delegation manager.

- **Role and module settings**
  - `test_setMinter` reverts for non-owner and emits `MinterSet` when updated by owner.
  - `test_module` reverts for non-owner then allows module transfers and approvals once whitelisted, validating module permissions.

5.9. Contract: GovMITOEmission

Description

GovMITOEmission manages gMITO emissions for validator rewards using customizable rate schedules and an epoch feeder. It allows owners and designated reward managers to configure emission parameters and recipients. It enables on-chain funding via payable emission additions and controlled reward requests per epoch.

Core functions and invariants

Function: `requestValidatorReward(uint256 epoch, address recipient, uint256 amount) external returns (uint256)`

- Caller must equal current reward recipient.
- Check `amount > 0` and `total >= spent + amount`.
- Increment `spent` by `amount`.
- Transfer `amount` of gMITO to `recipient`.
- Emit `ValidatorRewardRequested(epoch, recipient, amount)`.
- Return the requested `amount`.

Function: `addValidatorRewardEmission() external payable`

- Check `msg.value > 0`.
- Increase `total` by `msg.value`.
- Call `govMITO.mint{ value: msg.value }(address(this))`.
- Emit `ValidatorRewardEmissionAdded(_msgSender(), msg.value)`.

Function: `configureValidatorRewardEmission(uint256 rps, uint160 rateMultiplier, uint48 renewalPeriod, uint48 applyFrom) external onlyRole(VALIDATOR_REWARD_MANAGER_ROLE)`

- Check `applyFrom > block.timestamp`.
- Check `emissions` is empty or `applyFrom > last emission.timestamp`.
- Append new emission parameters to storage.
- Emit `ValidatorRewardEmissionConfigured(rps, rateMultiplier, renewalPeriod, applyFrom)`.

Function: `setValidatorRewardRecipient(address recipient) external onlyOwner`

- Check `recipient` is nonzero.
- Update stored recipient and emit `ValidatorRewardRecipientSet(previous, recipient)`.

Test coverage

- **Initialization**
  - `test_init` verifies role setup, `addValidatorRewardEmission` behavior, `validatorRewardTotal`, `validatorRewardSpent`, `validatorRewardEmissionsCount`, and emission lookup by index and time.
  - `test_init_invalidParameter` ensures revert when `startsFrom` is in the past.
- **Adding emissions**
  - `test_addValidatorRewardEmission` adds multiple emissions and confirms `validatorRewardTotal` accumulates correctly and `ValidatorRewardEmissionAdded` events fire.
- **Requesting rewards**
  - `test_requestValidatorReward` mocks epoch times, expects `ValidatorRewardRequested` event for epoch 1, and checks `validatorRewardSpent` increments.
- **Reward calculations (no config changes)**
  - `test_validatorReward_clean` asserts `validatorReward(epoch)` matches the expected rate schedule for epochs 1–5.
- **Reward calculations (with config updates)**
  - `test_validatorReward_dirty` grants manager role, applies several configuration updates, mocks epoch times, and validates that `validatorReward(epoch)` for epochs 1–9 reflects combined rates.

5.10. Contract: ReclaimQueue

Description

ReclaimQueue manages per-vault redeem queues for ERC-4626 vaults, allowing users to queue share redemptions, claim assets after a reclaim period, and synchronize on-chain asset reserves. It integrates with `IAssetManager` to coordinate asset flows and emits detailed logs for requests, claims, and sync operations. It includes owner-controlled enablement, asset-manager address, and reclaim period settings, with pause and upgrade safety.

Core functions

Function: `enableQueue(address vault)`

- Caller must be the owner of the contract.
- Set `queues[vault].isEnabled = true`.

- Compute and store `decimalsOffset` and `underlyingDecimals` for `vault`.
- Emit `QueueEnabled(vault)`.

Function: `setAssetManager(address assetManager_)`

- Caller must be the owner of the contract.
- Check `assetManager_.code.length > 0`.
- Update `assetManager` and emit `AssetManagerSet(assetManager_)`.

Function: `setReclaimPeriod(address vault, uint256 reclaimPeriod_)`

- Caller must be the owner of the contract.
- Check `queues[vault].isEnabled == true`.
- Update `queues[vault].reclaimPeriod = reclaimPeriod_`.
- Emit `ReclaimPeriodSet(vault, reclaimPeriod_)`.

Function: `request(uint256 shares, address receiver, address vault) returns (uint256)`

- Check `queues[vault].isEnabled == true`.
- Transfer `shares` of `vault` from caller to contract.
- Compute `assets = previewRedeem(shares)`.
- Append a new `Request(timestamp, assets, cumulativeShares)` to `queues[vault].items`.
- Add `reqId` to `queues[vault].indexes[receiver]`.
- Emit `Requested(receiver, vault, reqId, shares, assets)` and return `reqId`.

Function: `claim(address receiver, address vault) nonReentrant returns (uint256, uint256)`

- Check `queues[vault].isEnabled == true`.
- Check `indexes[receiver].size > 0` and `offset < size`.
- Processes up to `MAX_CLAIM_SIZE` requests older than reclaim period.
- Remove claimed entries by advancing `indexes[receiver].offset`.
- Emit `ClaimSucceeded(receiver, vault, result)` and transfer `totalAssetsClaimed` to `receiver`.
- Return `(totalSharesClaimed, totalAssetsClaimed)`.

Function: `sync(address executor, address vault, uint256 requestCount) returns (uint256, uint256)`

- Caller must be `assetManager`.
- Check `queues[vault].isEnabled == true` and `items.length > 0`.
- Aggregate up to `requestCount` pending items.
- Withdraw `min(totalAssetsOnRequest, totalAssetsOnReserve)` from `vault`.
- Update `queues[vault].offset` and append a `SyncLog`.
- Emit `Synced(executor, vault, result)` and return `(totalSharesSynced, assetsSynced)`.

Test coverage

- **Initialization**
  - `test_init` verifies `assetManager()`, `isEnabled(vault)` true, `isEnabled(vault2)` false, and correct `queueInfo` and `queueIndex` values.
- **Requesting redemptions**
  - `test_request` checks that calling `request(shares, user, vault)` transfers shares, increments `itemsLen` and `size`, and stores the correct `Request` in both `queueItem` and `queueIndexItem`.
  - `test_request_queueNotEnabled` reverts when queue is disabled for the vault.
- **Sync operations**
  - `test_sync` covers `previewSync` and `sync` for two batches — verifies computed `(totalSharesSynced, totalAssetsSynced)`, emitted `Synced` events, ERC-20 transfers, and updated `queueSyncLog`.
  - `test_sync_loss` simulates a loss (reserve < supply) and verifies sync behavior, transfers, and logs.
  - `test_sync_yield` simulates yield (reserve > supply) and verifies sync, remaining share balance, and logs.
  - `test_sync_unauthorized` reverts when called by non-`assetManager`.
  - `test_sync_queueNotEnabled` reverts when queue is disabled.
  - `test_sync_nothingToSync_init` and `test_sync_nothingToSync_afterRequest` revert when there is nothing to sync.
- **Claim operations**
  - `test_claim`, after multiple `request` and `sync` calls, warps past reclaim period then checks `previewClaim` results and full `claim(user, vault)` sequence — emitted `Claimed` and `ClaimSucceeded` events, asset transfers, and updated `queueIndex`.
  - `test_claim_queueNotEnabled` reverts when queue is disabled.
  - `test_claim_nothingToClaim_init` and `test_claim_nothingToClaim_afterRequest` revert when there are no claimable items.
- **Validation of accessors**
  - `test_queueItem_validation`, `test_queueIndexItem_validation`, and `test_queueSyncLog_validation` each verify that the corresponding getters revert with the correct error when the queue is empty or the index is out of bounds.

5.11. Contract: ConsensusValidatorEntrypoint

Description

ConsensusValidatorEntrypoint serves as the EVM-level bridge for validator operations, emitting events that the Cosmos SDK consensus layer consumes. It validates and forwards registration, staking, withdrawal, unjail, and voting-power update requests via signed EVM logs. It restricts all actions to an allowlist of callers and burns any accompanying ETH to represent collateral.

Core functions and invariants

Function: `registerValidator(address valAddr, bytes calldata pubKey, address initialCollateralOwner)`

- Caller must be in `isPermittedCaller`.
- `initialCollateralOwner` must be not zero address.
- Public keys pass `verifyPubKeyWithAddress` to verify that the given validator key is in valid format and corresponds to the expected address.
- Payable calls require `msg.value > 0` and `msg.value % 1 gwei == 0`.
- Emit `MsgRegisterValidator` event with parameters.
- All received ETH is forwarded to `address(0)`.

Function: `depositCollateral(address valAddr, address collateralOwner)`

- Caller must be in `isPermittedCaller`.
- Payable calls require `msg.value > 0` and `msg.value % 1 gwei == 0`.
- `collateralOwner` must be not zero address.
- Emit `MsgDepositCollateral` event with parameters.
- All received ETH is forwarded to `address(0)`.

Function: `function withdrawCollateral(address valAddr,address collateralOwner, address receiver, uint256 amount, uint48 maturesAt)`

- Caller must be in `isPermittedCaller`.
- collateralOwner must be not zero address.
- `amount` must be > 0 and `amount % 1 gwei == 0`.
- Emit `MsgWithdrawCollateral` event with parameters.

Function: `unjail(address valAddr)`

- Caller must be in `isPermittedCaller`.
- Emit `MsgUnjail` event with parameters.

Function: `updateExtraVotingPower(address valAddr, uint256 extraVotingPower)`

- Caller must be in `isPermittedCaller`.
- Emit `MsgUpdateExtraVotingPower` event with parameters.

Function: `transferCollateralOwnership(address valAddr, address prevOwner, address newOwner)`

- Caller must be in `isPermittedCaller`.
- `newOwner` must be not zero address.
- Emit `MsgTransferCollateralOwnership` event with parameters.

Permission functions and invariants

Function: `setPermittedCaller(address caller, bool isPermitted)`

- Caller must be the owner of the contract.
- Set `isPermittedCaller[caller]` to `isPermitted`.
- Emit `PermittedCallerSet` event with parameters.

Test coverage

- `ConsensusValidatorEntrypoint`'s behavior is covered end-to-end by ValidatorManager tests, which verify its event emissions and caller-allowlist enforcement.

5.12. Contract: ConsensusGovernanceEntrypoint

Description

ConsensusGovernanceEntrypoint serves as the EVM-level bridge for governance operations, emitting events that the Cosmos SDK consensus layer consumes.

Core functions and invariants

Function: `execute(string[] calldata messages)`

- Caller must be in `isPermittedCaller`.
- Emit `MsgExecute` event with messages.

Permission functions and invariants

Function: `setPermittedCaller(address caller, bool isPermitted)`

- Caller must be the owner of the contract.
- Set `isPermittedCaller[caller]` to `isPermitted`.
- Emit `PermittedCallerSet` event with parameters.

Test coverage

N/A.

5.13. Contract: AssetManager

Description

AssetManager acts as the hub-chain asset bridge, minting and burning hub assets upon branch-chain requests. It integrates with Matrix and EOL vaults for direct strategy deposits, handling caps and idle liquidity. It supports strategist allocations, reservations, and settlements of yields, losses, and extra rewards via entry point.

Core functions and invariants

Function: `deposit(uint256 chainId, address branchAsset, address to, uint256 amount)`

- Caller must be the entry point.
- Check `branchAsset` pair exists.
- Mint `amount` of hub asset to `to`.
- Emit `Deposited(chainId, hubAsset, to, amount)`.

Function: `depositWithSupplyMatrix(uint256 chainId, address branchAsset, address to, address matrixVault, uint256 amount)`

- Caller must be the entry point.
- Check `branchAsset` pair exists.
- Check `matrixVault` is initialized.
- Mint `amount` of hub asset to `to` if `hubAsset` is different from `branchAsset`. Or mint `amount` of branch asset to this contract.
    - Deposit into `matrixVault`.
    - Transfer excess hub assets to `matrixVault` if cap applies.
- Emit `DepositedWithSupplyMatrix(chainId, hubAsset, to, matrixVault, amount, supplyAmount)`.

Function: `depositWithSupplyEOL(uint256 chainId, address branchAsset, address to, address eolVault, uint256 amount)`

- Caller must be the entry point.
- Check `branchAsset` pair exists.
- Check `eolVault` is initialized.
- Mint `amount` of hub asset to `to` if `hubAsset` is different from `branchAsset`. Or mint `amount` of branch asset to this contract.
    - Deposit into `eolVault`.
- Emit `DepositedWithSupplyEOL(chainId, hubAsset, to, eolVault, amount,`

`amount)`.

Function: `withdraw(uint256 chainId, address hubAsset, address to, uint256 amount)`

- Check `to` is nonzero and `amount > 0`.
- Check `branchAsset` pair exists.
- Check branch liquidity and thresholds are satisfied.
- Burn `amount` from caller.
- Call `entrypoint.withdraw`.
- Emit `Withdrawn(chainId, hubAsset, to, amount)`.

Function: `allocateMatrix(uint256 chainId, address matrixVault, uint256 amount)`

- Caller must be the strategist.
- Check `matrixVault` is initialized.
- Check idle liquidity ≥ `amount`.
- Call `entrypoint.allocateMatrix`.
- Check branch liquidity is available.
- Increase allocations state.
- Emit `MatrixAllocated(operator, chainId, matrixVault, amount)`.

Function: `deallocateMatrix(uint256 chainId, address matrixVault, uint256 amount)`

- Caller must be the entry point.
- Decrease allocations state.
- Emit `MatrixDeallocated(chainId, matrixVault, amount)`.

Function: `reserveMatrix(address matrixVault, uint256 claimCount)`

- Caller must be strategist.
- Check `claimCount > 0` and reservation <= idle liquidity.
- Call `reclaimQueue.sync`.
- Emit `MatrixReserved(operator, matrixVault, claimCount, totalShares, totalAssets)`.

Function: `settleMatrixYield(uint256 chainId, address matrixVault, uint256 amount)`

- Caller must be the entry point.
- Mint yield amount to vault and increase allocations.
- Emit `MatrixRewardSettled(chainId, matrixVault, asset, amount)`.

Function: `settleMatrixLoss(uint256 chainId, address matrixVault, uint256 amount)`

- Caller must be the entry point.
- Burn loss amount from the vault and decrease allocations.
- Emit `MatrixLossSettled(chainId, matrixVault, asset, amount)`.

Function: `settleMatrixExtraRewards(uint256 chainId, address matrixVault, address`

branchReward, uint256 amount)

- Caller must be the entry point.
- Check branch reward and treasury are set.
- Mint reward to the contract and store via treasury.
- Emit `MatrixRewardSettled(chainId, matrixVault, hubReward, amount)`.
- Call `treasury.storeRewards` to transfer to treasury.

Permission functions and invariants

Function: `initializeAsset(uint256 chainId, address hubAsset)`

- Check `hubAsset` is the contract.
- Caller must be the owner of the contract.
- Check branch–asset pair exists.
- Call `entrypoint.initializeAsset`.
- Emit `AssetInitialized(hubAsset, chainId, branchAsset)`.

Function: `setBranchLiquidityThreshold(uint256 chainId, address hubAsset, uint256 threshold)`

- Caller must be the owner of the contract.
- Update threshold.
- Emit `BranchLiquidityThresholdSet(chainId, hubAsset, threshold)`.

Function: `initializeMatrix(uint256 chainId, address matrixVault)`

- Caller must be the owner of the contract.
- Check Matrix vault factory is initialized and `matrixVault` is set in factory.
- Check branch–asset pair exists.
- Check Matrix vault is uninitialized then marked initialized.
- Call `entrypoint.initializeMatrix`.
- Emit `MatrixInitialized(hubAsset, chainId, matrixVault, branchAsset)`.

Function: `initializeEOL(uint256 chainId, address eolVault)`

- Caller must be the owner of the contract.
- Check EOL vault factory is initialized and `eolVault` is set in factory.
- Check `branchAsset` pair exists.
- Check EOL vault is uninitialized then marked initialized.
- Call `entrypoint.initializeEOL`.
- Emit `EOLInitialized(hubAsset, chainId, eolVault, branchAsset)`.

Function: `setAssetPair(address hubAsset, uint256 branchChainId, address branchAsset)`

- Caller must be the owner of the contract.

- Check hub asset factory is initialized and `hubAsset` is set in factory.
- Check `branchAsset` pair exists.
- Update branch-asset mappings.
- Emit `AssetPairSet(hubAsset, branchChainId, branchAsset)`.

Function: `setEntrypoint(address entrypoint_)`

- Caller must be the owner of the contract.
- Set entry-point address.
- Emit `EntrypointSet(entrypoint_)`.

Function: `setReclaimQueue(address reclaimQueue_)`

- Caller must be the owner of the contract.
- Set reclaim-queue address.
- Emit `ReclaimQueueSet(reclaimQueue_)`.

Function: `setTreasury(address treasury_)`

- Caller must be the owner of the contract.
- Set treasury address.
- Emit `TreasurySet(treasury_)`.

Function: `setHubAssetFactory(address hubAssetFactory_)`

- Caller must be the owner of the contract.
- Set hub asset factory.
- Emit `HubAssetFactorySet(hubAssetFactory_)`.

Function: `setMatrixVaultFactory(address matrixVaultFactory_)`

- Caller must be the owner of the contract.
- Set Matrix vault factory.
- Emit `MatrixVaultFactorySet(matrixVaultFactory_)`.

Function: `setEOLVaultFactory(address eolVaultFactory_)`

- Caller must be the owner of the contract.
- Set EOL vault factory.
- Emit `EOLVaultFactorySet(eolVaultFactory_)`.

Function: `setStrategist(address matrixVault, address strategist)`

- Caller must be the owner of the contract.
- Update strategist.
- Emit `StrategistSet(matrixVault, strategist)`.

Test coverage

- **Deposit**
  - `test_deposit` covers successful deposit, mint call, and `Deposited` event.
  - `test_deposit_Unauthorized` reverts when caller is not entry point.
  - `test_deposit_BranchAssetPairNotExist` reverts when branch-asset pair is missing.
- **Deposit with Matrix vault**
  - `test_depositWithSupplyMatrix` covers both "vault asset ≠ hubAsset" and "vault asset == hubAsset" flows, including cap handling and excess transfer.
  - `test_depositWithSupplyMatrix_Unauthorized` reverts for non–entry-point callers.
  - `test_depositWithSupplyMatrix_BranchAssetPairNotExist` and `test_depositWithSupplyMatrix_MatrixNotInitialized` revert on missing pair or uninitialized vault.
- **Deposit with EOL vault**
  - `test_depositWithSupplyEOL` covers both "vault asset ≠ hubAsset" and "vault asset == hubAsset" flows.
  - `test_depositWithSupplyEOL_Unauthorized` reverts for non–entry-point callers.
  - `test_depositWithSupplyEOL_BranchAssetPairNotExist` and `test_depositWithSupplyEOL_EOLNotInitialized` revert on missing pair or uninitialized vault.
- **Withdraw**
  - `test_withdraw` covers successful burn, entry-point call, and `Withdrawn` event.
  - `test_withdraw_BranchAssetPairNotExist`, `test_withdraw_ToZeroAddress`, `test_withdraw_ZeroAmount`, `test_withdraw_BranchAvailableLiquidityInsufficient`, and `test_withdraw_BranchLiquidityThresholdNotSatisfied` cover all invalid-parameter and liquidity-threshold reverts.
- **Matrix allocation**
  - `test_allocateMatrix` covers authorized allocation, state update, and `MatrixAllocated` event.
  - `test_allocateMatrix_Unauthorized` and `test_allocateMatrix_MatrixNotInitialized` revert on invalid caller or uninitialized vault.
  - `test_allocateMatrix_MatrixInsufficient` and `test_allocateMatrix_BranchAvailableLiquidityInsufficient` revert when idle or branch liquidity is insufficient.
- **Matrix deallocation**
  - `test_deallocateMatrix` covers authorized deallocation, state update, and `MatrixDeallocated` event.
  - `test_deallocateMatrix_Unauthorized` reverts for non–entry-point callers.
- **Matrix reservation**

  - `test_reserveMatrix` covers strategist-only reservation via reclaim queue sync and `MatrixReserved` event.
  - `test_reserveMatrix_Unauthorized`, `test_reserveMatrix_MatrixNothingToReserve`, and `test_reserveMatrix_MatrixInsufficient` revert on invalid caller or zero/insufficient reservation scenarios.
- **Settle Matrix yield/loss/extra rewards**
  - `test_settleMatrixYield`, `test_settleMatrixLoss`, and `test_settleMatrixExtraRewards` cover successful mint/burn, approval, treasury storage calls, and events.
  - `test_settleMatrixYield_Unauthorized`, `test_settleMatrixLoss_Unauthorized`, and `test_settleMatrixExtraRewards_Unauthorized` revert on invalid callers.
  - `test_settleMatrixExtraRewards_BranchAssetPairNotExist` reverts on missing reward pair.
- **Initialize asset**
  - `test_initializeAsset` covers entry-point call, state update, and `AssetInitialized` event.
  - `test_initializeAsset_Unauthorized`, `test_initializeAsset_InvalidParameter`, and `test_initializeAsset_BranchAssetPairNotExist` revert on invalid caller, zero/invalid asset, or missing pair.
- **Branch-liquidity threshold**
  - `test_setBranchLiquidityThreshold` and `test_setBranchLiquidityThreshold_batch` cover owner-only updates in single and batch modes, events, and state checks.
  - `test_setBranchLiquidityThreshold_Unauthorized` and `test_setBranchLiquidityThreshold_HubAssetPairNotExist` revert on invalid caller or missing pair.
- **Asset pair and factories**
  - `test_setAssetPair` covers setting a new hub branch-asset mapping.
  - `test_setAssetPair_Unauthorized` and `test_setAssetPair_InvalidParameter` revert on invalid caller or invalid hub asset.
- **Entry point, reclaim queue, treasury, and strategist**
  - `test_setEntrypoint`, `test_setReclaimQueue`, `test_setTreasury`, and `test_setStrategist` cover owner-only address updates, events, and state checks.

5.14. Contract: AssetManagerEntrypoint

Description

AssetManagerEntrypoint serves as the EVM-level gateway for AssetManager, receiving on-chain calls and dispatching cross-chain messages via Hyperlane's GasRouter. It enforces permission guards — only AssetManager may trigger asset operations and only registered chains may be targeted. It provides owner-/registry-controlled enrollment of Hyperlane routers and gas configuration and supports UUPS upgrades.

Core functions and invariants

Function: `initializeAsset(uint256 chainId, address branchAsset)`

- Caller must be AssetManager.
- Check `chainId` must be registered and enrolled.
- Dispatch `MsgInitializeAsset` to branch via `_dispatchToBranch`.

Function: `initializeMatrix(uint256 chainId, address matrixVault, address branchAsset)`

- Caller must be AssetManager.
- Check `chainId` must be registered and enrolled.
- Dispatch `MsgInitializeMatrix` via `_dispatchToBranch`.

Function: `initializeEOL(uint256 chainId, address eolVault, address branchAsset)`

- Caller must be AssetManager.
- Check `chainId` must be registered and enrolled.
- Dispatch `MsgInitializeEOL` via `_dispatchToBranch`.

Function: `withdraw(uint256 chainId, address branchAsset, address to, uint256 amount)`

- Caller must be AssetManager.
- Check `chainId` must be registered and enrolled.
- Dispatch `MsgWithdraw` with encoded parameters via `_dispatchToBranch`.

Function: `allocateMatrix(uint256 chainId, address matrixVault, uint256 amount)`

- Caller must be AssetManager. ‘
- Check `chainId` must be registered and enrolled.
- Dispatch `MsgAllocateMatrix` via `_dispatchToBranch`.

Function: `_dispatchToBranch(uint256 chainId, bytes memory enc)`

- `chainId` must be registered in `_ccRegistry`.
- Quote fee via `_GasRouter_quoteDispatch` and dispatch via `_GasRouter_dispatch` for Hyperlane routers.

Function: `_handle(uint32 origin, bytes32 sender, bytes calldata msg_)`

- Check `chainId` is registered and enrolled using origin.
- Check `sender` is the entry point of the vault of the `chainId`.
- Verify `sender.toAddress()` matches the enrolled vault entry point.
- Parse `msg_` by `msgType()` and decode to the correct `Msg…` struct.
- Forward each decoded call to the corresponding AssetManager method.

Test coverage

- **Withdraw forwarding**
  - `test_withdraw` uses `entrypoint.assertLastCall` to verify that `IAssetManagerEntrypoint.withdraw` is called with `(branchChainId, branchAsset, user, amount)`.
- **Initialize asset dispatch**
  - `test_initializeAsset` uses `entrypoint.assertLastCall` to verify that `IAssetManagerEntrypoint.initializeAsset` is called with `(chainId, hubAsset)` after `initializeAsset`.

5.15. Contract: HubAsset

Description

HubAsset is the contract for the ERC-20 token representing hub assets, with minting and burning controlled by a designated supply manager.

It is upgradable via `Ownable2StepUpgradeable`, securing ownership transfers in two steps. Metadata (`name`, `symbol`, `decimals`) and supply-manager address are stored in upgrade-safe storage.

Core functions and invariants

Function: `mint(address account, uint256 value)`

- Caller must be `supplyManager`.
- Check `value > 0`.
- Invoke `_mint`.

Function: `burn(address account, uint256 value)`

- Caller must be `supplyManager`.
- Check `value > 0`.
- Invoke `_burn`.

Function: `setSupplyManager(address supplyManager_)`

- Caller must be the owner of the contract.
- Update `supplyManager` to `supplyManager_`.
- Emit `SupplyManagerUpdated(oldSupplyManager, supplyManager_)`.

Test coverage

- **Basic ERC-20 functionality test**
  - test basic transfer, approve, transferFrom, and so on.

5.16. Contract: HubAssetFactory

Description

HubAssetFactory deploys and manages upgradable HubAsset contracts via a beacon proxy pattern. It enables the owner to initialize the factory with a base implementation and to create new HubAsset instances with custom metadata and supply managers. It supports direct beacon calls and UUPS upgrades, all guarded by owner-only access.

Permission functions and invariants

Function: `create(address owner_, address supplyManager, string memory name, string memory symbol, uint8 decimals) → address`

- Caller must be the owner of the contract.
- Encode the provided parameters into `HubAsset.initialize` calldata.
- Deploy a new `BeaconProxy` pointing at the factory's beacon.
- Record the new proxy address in the instance registry.
- Return the address of the newly created proxy.

Function: `callBeacon(bytes calldata data) → bytes memory`

- Caller must be the owner of the contract.
- Forward the exact `data` payload to the beacon via `_callBeacon`.

Test coverage

- **Initialization**
  - `test_init` verifies the factory's implementation address, beacon admin slot, beacon owner, and initial implementation on the beacon.
- **Asset creation by owner**
  - `test_create` calls `create` as the `contractOwner` and asserts the new proxy's admin and impl slots are zero, the proxy's beacon matches `base.beacon()`, and `instancesLength` increments to 1 with the correct instance address.

- **Unauthorized creation**
  - `test_create_ownable` reverts when a non-owner attempts to call `create`, enforcing the `onlyOwner` guard.

5.17. Contract: CrossChainRegistry

Description

CrossChainRegistry maintains registry of supported chains with their Hyperlane domain IDs, vault entry points, and governance entry points. It allows the owner to add new chains, set vault implementations, and enroll remote entry points on Hyperlane routers. It provides read-only accessors for chain metadata and enrollment status.

Permission functions and invariants

Function: `setChain(uint256 chainId_, string calldata name, uint32 hplDomain, address mitosisVaultEntrypoint_, address governanceEntrypoint_)`

- Caller must be the owner of the contract.
- `chainId_` and `hplDomain` must not already be registered.
- Append `chainId_` and `hplDomain` to storage arrays.
- Store `name`, `mitosisVaultEntrypoint_`, and `governanceEntrypoint_` for the chain.
- Emit `ChainSet(chainId_, hplDomain, mitosisVaultEntrypoint_, governanceEntrypoint_, name)`.

Function: `setVault(uint256 chainId_, address vault_)`

- Caller must be the owner of the contract.
- `chainId_` must be registered and not yet have a vault.
- Store `vault_` for the given chain.
- Emit `VaultSet(chainId_, vault_)`.

Function: `enrollMitosisVaultEntrypoint(address hplRouter)`

- Caller must be the owner of the contract.
- Iterate over all registered chains, enrolling each `mitosisVaultEntrypoint` with `IRouter.enrollRemoteRouter`.

Function: `enrollGovernanceEntrypoint(address hplRouter)`

- Caller must be the owner of the contract.
- Iterate over all registered chains, enrolling each `governanceEntrypoint` with `IRouter.enrollRemoteRouter`.

Function: `enrollMitosisVaultEntrypoint(address hplRouter, uint256 chainId_)`

- Caller must be the owner of the contract.

- `chainId_` must be registered and its vault entry point not yet enrolled.
- Mark vault entry point as enrolled and call `IRouter.enrollRemoteRouter` for that chain.

Function: `enrollGovernanceEntrypoint(address hplRouter, uint256 chainId_)`

- Caller must be the owner of the contract.
- `chainId_` must be registered and its governance entry point not yet enrolled.
- Mark governance entry point as enrolled and call `IRouter.enrollRemoteRouter` for that chain.

Test coverage

- **Authorization**
  - `test_auth` reverts `setChain` when called by non-owner.
  - Valid owner transfer via `transferOwnership` and `acceptOwnership` then successful `setChain`.
- **Chain registration (`setChain`)**
  - `test_setChain` allows owner to register a new chain with correct metadata.
  - Reverts when attempting to register the same `chainId` twice.
  - Verifies `chainIds()`, `chainName()`, `hyperlaneDomain()`, `mitosisVaultEntrypoint()`, `governanceEntrypoint()`, and reverse lookup `chainId(domain)`.
- **Vault enrollment (`setVault`)**
  - `test_setVault` reverts when `setVault` is called for an unregistered chain.
  - After registering a chain, it allows owner to set a `mitosisVault`.
  - Verifies that `mitosisVault(chainId)` returns the expected vault address.

5.18. Contract: EOLVault

Description

EOLVault is an ERC-4626 vault wrapper for EOL assets, enabling standard deposit/mint and withdraw/redeem flows. It uses `Ownable2StepUpgradeable`, `Pausable`, and `ReentrancyGuardTransient` for safe upgrade, pause, and reentrancy protection. It stores custom `name`, `symbol`, and `decimals` per underlying asset, with sensible defaults when unspecified.

Core functions and invariants

Function: `deposit(uint256 assets, address receiver)`

- Call `ERC4626.deposit(assets, receiver)` and return shares minted.

Function: `mint(uint256 shares, address receiver)`

- Call `ERC4626.mint(shares, receiver)` and return assets deposited.

Function: `withdraw(uint256 assets, address receiver, address owner)`

- Call `ERC4626.withdraw(assets, receiver, owner)` and return shares burned.

Function: `redeem(uint256 shares, address receiver, address owner)`

- Call `ERC4626.redeem(shares, receiver, owner)` and return assets withdrawn.

Test coverage

- **Initialization**
  - `test_init` verifies `name()`, `symbol()`, `decimals()`, `asset()`, and `owner()` are set correctly.
- **Deposit**
  - `test_deposit` covers depositing 100 ETH: wraps to WETH, approves vault, calls `deposit`, and checks `balanceOf`, `totalAssets`, and `totalSupply`.
- **Mint**
  - `test_mint` covers minting 100 shares — wraps to WETH, approves vault, calls `mint`, and checks `balanceOf`, `totalAssets`, and `totalSupply`.
- **Withdraw**
  - `test_withdraw` reuses `test_deposit`, calls `withdraw(100 ETH, owner, user)`, and verifies vault balances reset and WETH is transferred back to `owner`.
- **Redeem**
  - `test_redeem` reuses `test_mint`, calls `redeem(100 ETH, owner, user)`, and verifies vault balances reset and WETH is transferred back to `owner`.

5.19. Contract: EOLVaultFactory

Description

EOLVaultFactory deploys and manages upgradeable EOLVault instances via a beacon proxy pattern. It tracks multiple vault types and their beacon implementations, allowing creation and migration of vault instances. It restricts initialization, beacon calls, creation, and migration to the contract owner.

Permission functions and invariants

Function: `initVaultType(VaultType vaultType, address initialImpl)`

- Caller must be the owner of the contract.
- `vaultType` must be non-`Unset`.
- Vault type must not already be initialized.

- Deploy a new `UpgradeableBeacon` pointing to `initialImpl`.
- Mark `infos[vaultType].initialized == true` and store beacon address.
- Emit `VaultTypeInitialized(vaultType, beacon)`.

Function: `callBeacon(VaultType t, bytes calldata data) returns (bytes memory)`

- Caller must be the owner of the contract.
- Vault type `t` must be initialized.
- Forwards `data` to the beacon via low-level `call`.
- Revert on beacon call failure.
- Emit `BeaconCalled(caller, t, data, success, result)` with returned payload.

Function: `create(VaultType t, bytes calldata args) returns (address instance)`

- Caller must be the owner of the contract.
- Vault type `t` must be initialized.
- Decode `args` into the correct init struct for `t`.
- Proxy creation through `BeaconProxy`, invoking `EOLVault.initialize`.
- Record new instance in `infos[t].instances` and `instanceIndex`.
- Emit `EOLVaultCreated(t, instance, args)`.

Function: `migrate(VaultType from, VaultType to, address instance, bytes calldata data)`

- Caller must be the owner of the contract.
- Both `from` and `to` types must be initialized.
- `instance` must be a tracked instance of the `from` type.
- Remove `instance` from `infos[from]` arrays and mappings.
- Add `instance` to `infos[to].instances` and update its index.
- Call `IBeaconProxy(instance).upgradeBeaconToAndCall(newBeacon, data)`.
- Emit `EOLVaultMigrated(from, to, instance)`.

Test coverage

- **Initialization**
  - `test_init` verifies `owner()` is set correctly after `initialize`.
- **Vault-type setup**
  - `test_initVaultType` (owner) initializes `Basic` vault type, checks nonzero beacon address, and `vaultTypeInitialized` returns true.
- **Basic vault creation**
  - `test_create_basic` initializes `Basic` type then calls `create` with `BasicVaultInitArgs`. It asserts that proxy's admin and implementation slots are zero, proxy's beacon matches stored beacon, `instancesLength` increments to 1, and `instances(0)` returns the new proxy address.

5.20. Contract: MITOGovernance

Description

MITOGovernance provides on-chain governance for gMITO holders by extending OpenZeppelin Governor V5 and TimelockController. It enables proposal creation, voting, queuing, timelocked execution, and cancellation on the Mitosis hub chain. It relays approved governance actions to branch chains via `BranchGovernanceEntrypoint` and to the consensus layer via `ConsensusGovernanceEntrypoint`.

Permission functions and invariants

Function: `propose(address[] targets, uint256[] values, bytes[] calldatas, string description)`

- Caller must be the proposer.
- Invoke OZ's Governor contract's `propose` function.

Test coverage

- N/A.

5.21. Contract: MITOGovernanceVP

Description

MITOGovernanceVP aggregates voting power across multiple gMITO-compatible voting token contracts (`ISudoVotes`), summing current and historical votes for each account. It supports off-chain delegation via EIP-712 signatures and on-chain delegation calls, updating all underlying `ISudoVotes` tokens. Includes owner-updatable token list with safety checks and historical event emission (`TokensUpdated`).

Core functions and invariants

Function: `updateTokens(ISudoVotes[] calldata newTokens_)`

- Caller must be the owner of the contract.
- Check `newTokens_`'s length > 0 and ≤ `MAX_TOKENS`.
- Check each `newTokens_[i]` has nonzero code length.
- Replace stored `tokens` array with `newTokens_`.
- Emit `TokensUpdated`.

Function: `delegate(address delegatee)`

- Invoke `_delegate` using `msg.sender` and `delegatee`.
- Call `sudoDelegate` on each `tokens[i]` with `msg.sender` and `delegatee`.
- Emit `DelegateChanged`.

Function: `delegateBySig(address delegatee, uint256 nonce, uint256 expiry, uint8 v, bytes32 r, bytes32 s)`

- Check `block.timestamp ≤ expiry`.
- Check signer is valid and not expired.
- Invoke `_delegate` using `signer` and `delegatee`.

Test coverage

- **Token updates**
  - `test_updateTokens` reverts when non-owner calls.
  - Reverts on zero-length input or invalid token address.
  - Emits `TokensUpdated` and updates `tokens` on valid owner call.
- **Vote queries**
  - `test_getVotes` returns sum of `getVotes` across all tokens.
  - `test_getPastVotes` returns sum of `getPastVotes` across all tokens.
  - `test_getPastTotalSupply` returns sum of `getPastTotalSupply` across all tokens.
- **Delegation lookup**
  - `test_delegates` returns the first token's delegate for a given account.
- **On-chain delegation**
  - `test_delegate` emits `DelegateChanged` and calls `sudoDelegate` on each token for authorized sender.
- **Off-chain delegation (by signature)**
  - `test_delegateBySig` reverts when signature is invalid or expired.
  - Emits `DelegateChanged` and calls `sudoDelegate` on each token when signature is valid.

5.22. Contract: BranchGovernanceEntrypoint

Description

BranchGovernanceEntrypoint acts as the hub-chain entry point for governance proposals targeting branch chains via Hyperlane. It allows managers to dispatch encoded governance calls to registered remote chains with fee quoting and gas routing. It manages enrollment of remote routers and gas configurations under owner or registry control.

Permission functions and invariants

Function: `dispatchGovernanceExecution(uint256 chainId, address[] calldata targets, bytes[] calldata data, uint256[] calldata values, bytes32 predecessor, bytes32 salt)`

- Caller must be `MANAGER_ROLE`.
- Check `chainId` is registered and enrolled.
- Encoded payload matches inputs and is sent via `_dispatchToBranch`.
- Emit `ExecutionDispatched`.

Test coverage

N/A.

5.23. Contracts: MatrixVault, MatrixVaultBasic, and MatrixVaultCapped

Description

MatrixVault is abstract contract for basic and capped. This contract is an ERC-4626 vault implementation for Matrix strategies, enabling secure deposit/mint and withdraw/redeem flows. Configurable metadata (`name`, `symbol`, `decimals`) is derived from the underlying asset or sensible defaults. It integrates `Ownable2StepUpgradeable`, `Pausable`, and `ReentrancyGuardTransient` for upgrade safety, pausing, and reentrancy protection.

MatrixVaultBasic is same as MatrixVault.

MatrixVaultCapped extends MatrixVault by enforcing a maximum share cap on deposits and mints. It exposes `loadCap` to view the current cap and overrides `maxDeposit`/`maxMint` to respect the cap. It restricts cap updates to the `assetManager` owner and emits `CapSet` on changes.

Core functions and invariants

Function: `deposit(uint256 assets, address receiver)`

- Check `whenNotPaused` and `nonReentrant`.
- Assert `assets <= maxDeposit(receiver)`.
- Call `_deposit(msg.sender, receiver, assets, previewDeposit(assets))`.
- Return the correct `shares` amount.

Function: `mint(uint256 shares, address receiver)`

- Check `whenNotPaused` and `nonReentrant`.
- Assert `shares <= maxMint(receiver)`.
- Call `_deposit(msg.sender, receiver, previewMint(shares), shares)`.

- Return the correct `assets` amount.

Function: `withdraw(uint256 assets, address receiver, address owner)`

- Check `whenNotPaused` and `nonReentrant`.
- Assert caller is the reclaim queue (`_assertOnlyReclaimQueue`).
- Assert `assets <= maxWithdraw(owner)`.
- Call `_withdraw(msg.sender, receiver, owner, assets, previewWithdraw(assets))`.
- Return the correct `shares` burned.

Function: `redeem(uint256 shares, address receiver, address owner)`

- Check `whenNotPaused` and `nonReentrant`.
- Assert caller is the reclaim queue (`_assertOnlyReclaimQueue`).
- Assert `shares <= maxRedeem(owner)`.
- Call `_withdraw(msg.sender, receiver, owner, previewRedeem(shares), shares)`.
- Return the correct `assets` amount.

New feature for MatrixVaultCapped (and invariants)

Function: `setCap(uint256 newCap)`

- Caller must equal `Ownable(assetManager).owner()`.
- Update the cap storage slot to `newCap`.
- Emit `CapSet(setter, previousCap, newCap)`.

Test coverage

N/A.

5.24. Contract: Treasury

Description

Treasury acts as an on-chain reward vault, accepting and holding ERC-20 reward tokens from vaults and dispatching them to distributors. It enforces role-based access: `TREASURY_MANAGER_ROLE` for storing rewards and `DISPATCHER_ROLE` for dispatching. It records a history log of deposits and withdrawals with timestamps and supports UUPS upgrades via admin role.

Core functions and invariants

Function: `storeRewards(address vault, address reward, uint256 amount)`

- Caller must be `TREASURY_MANAGER_ROLE`.
- Check `vault` and `reward` are nonzero addresses and `amount > 0`.
- Transfer `amount` of `reward` from caller into contract.
- Increment stored balance and append a deposit `Log` entry with the current timestamp.
- Emit `RewardStored(vault, reward, sender, amount)`.

Function: `dispatch(address vault, address reward, uint256 amount, address distributor)`

- Caller must be `DISPATCHER_ROLE`.
- Check `vault`, `reward`, and `distributor` are nonzero, `amount > 0`, and stored balance for `(vault, reward)` ≥ `amount`.
- Decrement stored balance and append a withdrawal `Log` entry with the current timestamp.
- Transfer `amount` of `reward` to `distributor`.
- Emit `RewardDispatched(vault, reward, distributor, amount)`.

Test coverage

- **`storeRewards`**
  - `test_storeReward` mints tokens to `rewarder`, approves the treasury, calls `storeRewards(matrixVault, token, 100 ether)`, and verifies the treasury's ERC-20 balance increases by 100 Ether.
- **`dispatch`**
  - `test_dispatch` first runs `test_storeReward`, grants `DISPATCHER_ROLE` to `dispatcher`, calls `dispatch(matrixVault, token, 100 ether, distributor)`, and verifies the treasury's balance returns to zero while `distributor` receives 100 Ether.

5.25. Contract: MerkleRewardDistributor

Description

MerkleRewardDistributor provides Merkle tree–based reward distribution with on-chain proof verification and batch claiming. Managers can stage new reward roots and reserve amounts via `fetchRewards` and `addStage`. Users claim rewards by submitting a valid Merkle proof; reserves are decremented and tokens transferred from the treasury.

Core functions and invariants

Function: `claim(address receiver, uint256 stage, address vault, address[] calldata rewards, uint256[] calldata amounts, bytes32[] calldata proof)`

- Check `!claimed[stage][receiver][vault]`.

- Verify `proof` against stored `root(stage)` for the encoded leaf.
- Mark `(receiver, stage, vault)` as claimed.
- Decrement `reservedRewardAmounts` for each reward token by `amounts[i]`.
- Transfer each `amounts[i]` of `rewards[i]` to `receiver`.
- Emit `Claimed(receiver, stage, vault, rewards, amounts)`.

Function: `claimMultiple(address receiver, uint256 stage, address[] calldata vaults, address[][] calldata rewards, uint256[][] calldata amounts, bytes32[][] calldata proofs)`

- All array lengths (`vaults`, `rewards`, `amounts`, `proofs`) match and ≤ `MAX_CLAIM_VAULT_SIZE`.
- Loop over each vault index and call `claim` with corresponding parameters.

Function: `claimBatch(address receiver, uint256[] calldata stages, address[][] calldata vaults, address[][][] calldata rewards, uint256[][][] calldata amounts, bytes32[][][] calldata proofs)`

- All outer array lengths (`stages`, `vaults`, `rewards`, `amounts`, `proofs`) match and ≤ `MAX_CLAIM_STAGES_SIZE`.
- Loop over each stage index and call `claimMultiple` accordingly.

Function: `fetchRewards(uint256 stage, uint256 nonce, address vault, address reward, uint256 amount)`

- Caller must be `MANAGER_ROLE`.
- Check `stage == lastStage` and `nonce == stages[stage].nonce`.
- Increment `stages[stage].nonce`.
- Call `treasury.dispatch(vault, reward, amount, address(this))`.
- Emit `RewardsFetched(stage, nonce, vault, reward, amount)`.

Function: `fetchRewardsMultiple(uint256 stage, uint256 nonce, address vault, address[] calldata rewards, uint256[] calldata amounts)`

- Caller must be `MANAGER_ROLE`.
- Check `stage == lastStage` and `nonce == stages[stage].nonce`.
- `rewards.length == amounts.length`.
- Loop over each `rewards[i]` calling `_fetchRewards(stage, vault, rewards[i], amounts[i])`.

Function: `fetchRewardsBatch(uint256 stage, uint256 nonce, address[] calldata vaults, address[][] calldata rewards, uint256[][] calldata amounts)`

- Caller must be `MANAGER_ROLE`.
- Check `stage == lastStage` and `nonce == stages[stage].nonce`.
- All outer array lengths match.
- Nested loops over `vaults[i]` and `rewards[i][j]`, calling `_fetchRewards(stage, vaults[i], rewards[i][j], amounts[i][j])`.

Function: `addStage(bytes32 merkleRoot, uint256 stage, uint256 nonce, address[] calldata rewards, uint256[] calldata amounts) → uint256`

- Caller must be `MANAGER_ROLE`.
- Check `stage == lastStage` and `nonce == stages[stage].nonce`.
- Check `rewards.length == amounts.length`.
- Check each `amounts[i] <= availableRewardAmount(rewards[i])`.
- Increment `lastStage` and set new `root`, `rewards`, and `amounts`.
- Emit `StageAdded(newStage, merkleRoot, rewards, amounts)`.
- Return the new stage number.

Test coverage

- **`fetchRewards`**
  - `test_fetchRewards` mints and approves 100 tokens to the treasury, calls `storeRewards`, and then invokes `fetchRewards` on the distributor.
  - Verifies that the treasury's token balance is zero and the distributor's token balance increases by 100.

5.26. Contract: ValidatorManager

Description

ValidatorManager manages hub-chain validator operations.

- Validator setup, allowing operators to create validators (verifying their public keys), update operator settings, commission schedules, and metadata and to query validator state at any epoch.
- Collateral management, supporting staking (`depositCollateral`), timed withdrawals (`withdrawCollateral`), and unjailing after downtime (`unjailValidator`), all routed through the consensus-layer entry point.
- Protocol configuration, where the owner sets global parameters (transaction fees, minimum commission rates, withdrawal delays) with each change emitting an event.

Core functions and invariants

Function: `createValidator(bytes pubKey, CreateValidatorRequest request)`

- Check `pubKey` is not empty.
- Verify `pubKey` is valid and corresponds to `valAddr` using `verifyPubKeyWithAddress`.
- Protocol fee is charged.
- Check validator is not already registered.
- Check initial value is greater than `initialValidatorDeposit`.

- Check commission rate is within global bounds.
- Update `indexByValAddr[valAddr]` and `validators[valAddr]`.
- Emit `ValidatorCreated`.

Function: `depositCollateral(address valAddr)`

- Protocol fee is charged.
- `valAddr` must be a registered validator address.
- Call `depositCollateral` on the entry point.
- Emit `CollateralDeposited`.

Function: `withdrawCollateral(address valAddr, address receiver, uint256 amount)`

- Protocol fee is charged.
- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator.
- Call `withdrawCollateral` on the entry point.
- Emit `CollateralWithdrawn`.

Function: `unjailValidator(address valAddr)`

- Protocol fee is charged.
- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator or validator itself.
- Call `unjail` on the entry point.
- Emit `ValidatorUnjailed`.

Function: `transferCollateralOwnership(address valAddr, address newOwner)`

- Protocol fee is charged.
- `valAddr` must be a registered validator address.
- Call `transferCollateralOwnership` on the entry point.
- Emit `CollateralOwnershipTransferred`.

Permission functions and invariants

Function: `updateOperator(address valAddr, address newOperator)`

- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator.
- Set `validators[valAddr].operator` to `newOperator`.
- Emit `OperatorUpdated`.

Function: `updateWithdrawalRecipient(address valAddr, address recipient)`

- `valAddr` must be a registered validator address.

- Caller must be the operator of the validator.
- Check `recipient` is not zero address.
- Set `validators[valAddr].withdrawalRecipient` to `recipient`.
- Emit `WithdrawalRecipientUpdated`.

Function: `updateRewardManager(address valAddr, address rewardManager)`

- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator.
- Check `rewardManager` is not zero address.
- Set `validators[valAddr].rewardManager` to `rewardManager`.
- Emit `RewardManagerUpdated`.

Function: `updateMetadata(address valAddr, bytes metadata)`

- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator.
- Check `metadata` is not empty.
- Set `validators[valAddr].metadata` to `metadata`.
- Emit `MetadataUpdated`.

Function: `updateRewardConfig(address valAddr, UpdateRewardConfigRequest request)`

- `valAddr` must be a registered validator address.
- Caller must be the operator of the validator.
- Check commission rate is within global bounds.
- Update previous pending rate if condition is met.
- Set the pending commission rate.
- Emit `RewardConfigUpdated`.

Function: `setFee(uint256 fee)`

- Caller must be the contract owner.
- Update `fee`.
- Emit `FeeSet`.

Function: `setGlobalValidatorConfig(SetGlobalValidatorConfigRequest request)`

- Caller must be the contract owner.
- Check commission rate is within global bounds.
- Update `globalConfig`.
- Emit `GlobalValidatorConfigUpdated`.

Test coverage

- **Initialization**

    - `test_init` verifies owner, epoch feeder, entry-point addresses, global config, fee, and empty validator set.
- **Validator creation**
    - `test_createValidator` checks pubkey verification, fee deduction, consensus entry-point call, and registry state.
    - `test_createValidator_with_zero_fee` repeats creation with the fee set to zero.
- **Collateral deposit**
    - `test_depositCollateral` covers operator update, fee handling, and correct entry-point invocation.
    - `test_depositCollateral_with_zero_fee` repeats deposit when fee is zero.
- **Collateral withdrawal**
    - `test_withdrawCollateral` exercises fee gate, revert on insufficient fee, and entry-point call.
    - `test_withdrawCollateral_with_zero_fee` repeats withdrawal with zero fee.
- **Unjailing**
    - `test_unjailValidator` checks operator update, fee handling, revert paths, and entry-point unjail call.
    - `test_unjailValidator_with_zero_fee` repeats unjail with zero fee.
- **Protocol fees**
    - `test_setFee` validates owner-only fee changes, reverts for unauthorized callers, and zero/restore flows.
- **Operator-only updates**
    - `test_updateOperator` confirms operator address change.
    - `test_updateRewardManager` verifies reward-manager update.
    - `test_updateMetadata` checks metadata storage.
- **Commission schedule**
    - `test_updateRewardConfig` ensures delayed commission-rate update and historical epoch queries via `validatorInfoAt`.
- **Global minimum commission**
    - `test_globalMinimumCommissionRateChange` validates that raising the protocol minimum overrides pending lower rates after delay.

5.27. Contract: ValidatorStaking

Description

ValidatorStaking supports the staking, unstaking, redelegation, and claim to validators from the user.

Core functions and invariants

Function: `stake(address valAddr, address recipient, uint256 amount)`

- Check `amount` is not zero.
- Check `amount` is greater than `minStakingAmount`.
- If `_baseAsset` is `NATIVE`, check `msg.value` is the same as `amount`.
- Check `recipient` is not zero address.
- Check `valAddr` is a valid validator address.
- If `_baseAsset` is not `NATIVE`, transfer `amount` from `payer` to this contract.
- Update states:
    - Increase `totalStaked`.
    - Update validator's stake state — increase `staked[valAddr][staker]`, `stakerTotal[staker]`, and `validatorTotal[valAddr]`.
- Call `notifyStake` on `_hub` contract.
- Emit `Staked`.

Function: `requestUnstake(address valAddr, address receiver, uint256 amount)`

- Check `amount` is not zero.
- Check `valAddr` is a valid validator address.
- Check `amount` is greater than `minUnstakingAmount`.
- Check `amount` is less than or equal to `staked[valAddr][staker]`.
- Update `unstakeQueue`.
- Update states:
    - Decrease `totalStaked`.
    - Increase `totalUnstaking`.
    - Update validator's stake state — decrease `staked[valAddr][staker]`, `stakerTotal[staker]`, and `validatorTotal[valAddr]`.
- Call `notifyUnstake` on `_hub` contract.
- Emit `UnstakeRequested`.

Function: `claimUnstake(address receiver)`

- Calculate claimed amount using `unstakeQueue`.
- Transfer assets to `receiver`.
- Update states — decrease `totalUnstaking`.
- Emit `UnstakeClaimed`.

Function: `redelegate(address fromValAddr, address toValAddr, uint256 amount)`

- Check `amount` is not zero.
- Check `fromValAddr` is not the same as `toValAddr`.
- Check `amount` is greater than `minUnstakingAmount`.
- Check `amount` is less than or equal to `staked[fromValAddr][staker]`.

- Check `fromValAddr` is a valid validator address.
- Check `toValAddr` is a valid validator address.
- Check within redelegation cooldown period.
- Update new redelegation cooldown period.
- Update states:
    - Update from validator's stake state.
    - Update to validator's stake state.
- Call `notifyRedelegate` on `_hub` contract.
- Emit `Redelegated`.

Permission functions and invariants

Function: `setMinStakingAmount(uint256 minAmount)`

- Caller must be the owner of the contract.
- Set `minStakingAmount` to `minAmount`.
- Emit `MinimumStakingAmountSet`.

Function: `setMinUnstakingAmount(uint256 minAmount)`

- Caller must be the owner of the contract.
- Set `minUnstakingAmount` to `minAmount`.
- Emit `MinimumUnstakingAmountSet`.

Function: `setUnstakeCooldown(uint48 unstakeCooldown_)`

- Caller must be the owner of the contract.
- Check `unstakeCooldown_` is not zero.
- Set `unstakeCooldown` to `unstakeCooldown_`.
- Emit `UnstakeCooldownUpdated`.

Function: `setRedelegationCooldown(uint48 redelegationCooldown_)`

- Caller must be the owner of the contract.
- Check `redelegationCooldown_` is not zero.
- Set `redelegationCooldown` to `redelegationCooldown_`.
- Emit `RedelegationCooldownUpdated`.

Test coverage

- **Initialization**
    - `test_init` verifies correct initialization of vaults (owner, `baseAsset`, manager, hub, min staking/unstaking amounts, cooldowns) for both ERC-20 and native vaults.
- **Minimum staking/unstaking amounts**

    - `test_minStakingAmount` checks access control and update logic for minimum staking amount.
    - `test_minUnstakingAmount` checks access control and update logic for minimum unstaking amount.
- **Staking**
    - `test_stake` exercises general staking logic for both ERC-20 and native vaults.
    - `test_stake_minAmount` verifies staking fails for amounts below minimum and succeeds for valid amounts.
    - `test_stake_fromAnotherAddress` verifies staking on behalf of another user and state updates.
- **Unstaking**
    - `test_unstake` covers general unstaking flow for both ERC-20 and native vaults.
    - `test_unstake_minAmount` verifies unstaking fails below minimum amount and succeeds for valid amounts.
    - `test_unstake_fromAnotherAddress` tests unstaking on behalf of another user and checks staked/unstaking state.
    - `test_unstake_and_multiple_claim` validates multiple claim flows after unstaking.
- **Redelegation**
    - `test_redelegate` verifies redelegation between validators, checking state transitions.
    - `test_redelegate_minAmount` ensures redelegation fails below minimum amount and succeeds for valid amounts.
    - `test_redelegate_fromAnotherAddress` checks redelegation on behalf of another user and state updates.
    - `test_redelegate_cooldown` validates redelegation fails during cooldown and passes after.
- **Unstake cooldown**
    - `test_setUnstakeCooldown` checks access control and update logic for unstake cooldown.
- **Redelegation cooldown**
    - `test_setRedelegationCooldown` checks access control and update logic for redelegation cooldown.

5.28. Contract: ValidatorStakingHub

Description

ValidatorStakingHub is a stateful contract that tracks staking-related balances (total per staker, per validator, and per validator-staker pair) and maintains time-weighted average balance (TWAB) data for validators and stakers. It processes staking, unstaking, and redelegation notifications from authorized notifiers, and it synchronizes validator voting power with the consensus-layer entry

point.

Core functions and invariants

Function: `notifyStake(address valAddr, address staker, uint256 amount)`

- Check `amount` is not zero.
- Caller must be in `isNotifier`.
- Check `valAddr` is a valid validator address.
- Update checkpoint states — increase `totalStaked`, `validatorTotal[valAddr]`, and `validatorStakerTotal[valAddr][staker]`.
- Emit `NotifiedStake`.

Function: `notifyUnstake(address valAddr, address staker, uint256 amount)`

- Check `amount` is not zero.
- Caller must be in `isNotifier`.
- Check `valAddr` is a valid validator address.
- Update checkpoint states — decrease `totalStaked`, `validatorTotal[valAddr]`, and `validatorStakerTotal[valAddr][staker]`.
- Emit `NotifiedUnstake`.

Function: `notifyRedelegation(address fromValAddr, address toValAddr, address staker, uint256 amount)`

- Check `fromValAddr` is not the same as `toValAddr`.
- Check `amount` is not zero.
- Caller must be in `isNotifier`.
- Update checkpoint states — decrease `validatorTotal[fromValAddr]` and `validatorStakerTotal[fromValAddr][staker]` and increase `validatorTotal[toValAddr]` and `validatorStakerTotal[toValAddr][staker]`.
- Emit `NotifiedRedelegation`.

Permission functions and invariants

Function: `addNotifier(address notifier)`

- Caller must be the owner of the contract.
- Check `notifier` is not zero address.
- Check `notifier` is not already in the list of notifiers.
- Set `isNotifier[notifier]` to true.
- Emit `NotifierAdded` event with parameters.

Function: `removeNotifier(address notifier)`

- Caller must be the owner of the contract.
- Check `notifier` is not zero address.
- Check `notifier` is in the list of notifiers.
- Set `isNotifier[notifier]` to false.
- Emit `NotifierRemoved` event with parameters.

Test coverage

- **Initialization**
  - `test_init` verifies owner address and consensus entry-point address are correctly initialized.
- **Notifier management**
  - `test_addNotifier` checks adding notifiers by the owner and ensures correct `isNotifier` status.
  - `test_removeNotifier` validates removing notifiers by the owner and checks `isNotifier` status update.
- **Staking notifications**
  - `test_notifyStake` verifies staking flow with notifier access control, stake updates, voting power sync, and TWAB calculations for staker, validator, and validator-staker pairs.
- **Unstaking notifications**
  - `test_notifyUnstake` checks unstaking flow with notifier access control, stake reductions, voting power sync, and TWAB updates reflecting unstaking events.
- **Redelegation notifications**
  - `test_notifyRedelegation` validates redelegation flow between validators, notifier access control, stake migration updates, and correct TWAB recalculations for source and destination validators.
- **Access control and revert paths**
  - All notification functions (`notifyStake`, `notifyUnstake`, `notifyRedelegation`) include tests for unauthorized notifier reverts.
- **State invariants**
  - Tests ensure that staking-related state (`stakerTotal`, `validatorTotal`, `validatorStakerTotal`) and TWAB values are correctly updated over time and across events.

5.29. Contract: ValidatorStakingGovMITO

Description

ValidatorStakingGovMITO extends ValidatorStaking to integrate on-chain governance voting (gMITO) via OpenZeppelin's `Votes` abstraction. It enforces nontransferable voting units — minted on stake and burned on unstake claim, ensuring voting power mirrors active stake. It adds sudo

governance capabilities through `SudoVotes` and ties the clock to block timestamps (EIP-6372).

Core functions and invariants

Function: `_stake(..., address valAddr, address payer, address recipient, uint256 amount)`

- Check `recipient` equals `payer`.
- Mint voting units.
- Invoke original `_stake` function of ValidatorStaking.

Function: `_requestUnstake(..., address valAddr, address payer, address receiver, uint256 amount)`

- Check `receiver` equals `payer`.
- Invoke original `_requestUnstake` function of ValidatorStaking.

Function: `_claimUnstake(..., address receiver)`

- Burn voting units.
- Invoke original `_claimUnstake` function of ValidatorStaking.

Test coverage

- **Unsupported delegation**
  - `test_delegate` and `test_delegateBySig` revert for standard `delegate` calls.
- **Delegation manager setup**
  - `test_setDelegationManager` reverts when called by non-owner — emits `DelegationManagerSet` and updates state when called by owner.
- **`sudoDelegate`**
  - `test_sudoDelegate` reverts for unauthorized caller — emits `DelegateChanged` and `DelegateVotesChanged` for valid delegation.
- **Stake voting power**
  - `test_stake` covers staking with and without prior delegation and verifies `getVotes` and `getPastTotalSupply` updates.
- **Nontransferable stake**
  - `test_stake_nonTransferable` reverts when `recipient != payer`.
- **Unstake request**
  - `test_requestUnstake` schedules cooldown correctly; votes remain locked until claim.
- **Nontransferable unstake**
  - `test_requestUnstake_nonTransferable` reverts when `receiver != payer`.
- **Claim unstake**
  - `test_claimUnstake` emits `DelegateVotesChanged`, returns correct amount, and updates voting power.

- **Reentrancy scenario**
  - `test_temporaryVoteUsingClaimCall` demonstrates temporary voting-power inflation via fallback during `claimUnstake`.

5.30. Contract: ValidatorRewardDistributor

Description

ValidatorRewardDistributor handles gMITO reward distribution to validators and their stakers each epoch, based on contribution data. It supports individual and batch claims for staker and operator rewards with configurable limits. It allows claim approvals by stakers or operators and owner control over claim configuration.

Core functions

Function: `setStakerClaimApprovalStatus(address valAddr, address claimer, bool approval)`

- Update `stakerClaimApprovals[_msgSender()][valAddr][claimer]` to `approval`.
- Emit `StakerRewardClaimApprovalUpdated(_msgSender(), valAddr, claimer, approval)`.

Function: `setOperatorClaimApprovalStatus(address valAddr, address claimer, bool approval)`

- Update `operatorClaimApprovals[_msgSender()][valAddr][claimer]` to `approval`.
- Emit `OperatorRewardClaimApprovalUpdated`.

Function: `claimStakerRewards(address staker, address valAddr)`

- Invoke `_claimStakerRewards`.
  - Check caller has approval to claim rewards for `staker` and `valAddr`.
  - Calculate claimable epoch by `_claimRange`.
  - Calculate `claimable` rewards.
  - Call `requestValidatorReward` to get rewards and update `totalClaimed`.
  - Update `lastClaimedEpoch` for `(staker, valAddr)` to the last processed epoch.
  - Emit `StakerRewardsClaimed`.

Function: `batchClaimStakerRewards(address[] calldata stakers, address[][] calldata valAddrs)`

- Check element's length is not greater than `maxStakerBatchSize`.
- Check element's length of `stakers` and `valAddrs` are equal.
- Invoke `_claimStakerRewards` for each `staker` and `valAddr` pair.

Function: `claimOperatorRewards(address valAddr)`

- Invoke `_claimOperatorRewards`.
  - Check caller is the validator's `rewardManager` or approved claimer.
  - Calculate claimable epoch by `_claimRange`.
  - Calculate `claimable` rewards.
  - Call `requestValidatorReward` to get rewards and update `totalClaimed`.
  - Update `lastClaimedEpoch` for `valAddr` to the last processed epoch.
  - Emit `OperatorRewardsClaimed`.

Function: `batchClaimOperatorRewards(address[] calldata valAddrs)`

- Check element's length is not greater than `maxOperatorBatchSize`.
- Invoke `_claimOperatorRewards` for each `valAddr`.

Permission functions and invariants

Function: `setClaimConfig(uint32 maxClaimEpochs, uint32 maxStakerBatchSize, uint32 maxOperatorBatchSize)`

- Check caller is the owner of the contract.
- Update `claimConfig` fields to the given values.
- Emit `ClaimConfigUpdated`.

Test coverage

- **Initialization**
  - `test_init` verifies `owner`, `epochFeeder`, `validatorManager`, `stakingHub`, `contributionFeed`, and `govMITOEmission` addresses and default `claimConfig` values.
- **Single-epoch single-validator claim**
  - `test_claim_rewards` sets up one epoch and one validator and ensures that operator and staker claimable amounts are correct, that `claimOperatorRewards` and `claimStakerRewards` transfer those amounts once and then return zero, and that `lastClaimedEpoch` is updated.
- **Single-epoch multiple-validators claim**
  - `test_claim_rewards_by_multiple_validator` tests two validators with equal weight and verifies per-validator operator and staker claims behave independently and update state correctly.
- **Single-epoch differing weights**
  - `test_claim_rewards_by_multiple_validator_diff_weight` configures 70/30 weight split and checks scaled operator and staker rewards for each validator.
- **Multiple stakers for one validator**
  - `test_claim_rewards_by_multiple_stakers` uses one validator with three

stakers (50/25/25) and ensures each staker's claimable and claimed amounts match their share.

- **Collateral-only or delegation-only edge cases**
  - `test_claim_rewards_validator_collateral_zero` and `test_claim_rewards_validator_delegation_zero` verify correct reward division when one share is zero.
- **Multiple epochs' sequential claims**
  - `test_claim_multiple_epoch` runs two epochs for one validator and ensures cumulative claimable totals and `lastClaimedEpoch` reflect both epochs.
- **Multiple epochs and multiple validators**
  - `test_claim_multiple_epoch_multiple_validator` covers two epochs and two validators and confirms combined and per-validator reward logic across epochs.
- **Batch operator and staker claims**
  - `test_batch_claim_rewards_by_multiple_stakers` exercises `batchClaimOperatorRewards` and `batchClaimStakerRewards` with approvals and verifies aggregated reward amounts and state resets.
- **Unavailable epoch gating**
  - `test_claim_rewards_unavailable` and `test_claim_batch_rewards_unavailable` ensure no claims are allowed and state remains unchanged when reward availability is false.
- **Max-epoch limit enforcement**
  - `test_claim_rewards_gt_32_epochs` simulates 35 epochs and checks that only up to `maxClaimEpochs` are autoclaimed and additional epochs can be claimed separately.
- **Claim-approval mechanics**
  - `test_claim_approval_own` verifies that default operator and staker can claim without explicit approval.
  - `test_claim_approval_delegate` and `test_claim_approval_false` ensure unauthorized claim attempts revert and that setting and clearing approvals allow or disallow delegates correctly.
  - `test_batch_claim_approval` confirms batch-claim calls revert until each validator/staker grants approval, then it succeeds.
- **Approval-status getters**
  - `test_setOperatorClaimApprovalStatus` and `test_setStakerClaimApprovalStatus` verify that setting approval toggles the corresponding `operatorClaimAllowed` and `stakerClaimAllowed` flags.

5.31. Contract: ValidatorContributionFeed

Description

ValidatorContributionFeed manages per-epoch validator weight reports used for reward distribution. It allows a designated feeder role to initialize, push weights, finalize, or revoke reports.

It provides read-only accessors for epoch readiness and individual validator weights.

Core functions and invariants

Function: `initializeReport(InitReportRequest calldata request)`

- Caller has `FEEDER_ROLE`.
- Check next epoch is not initialized.
- Check reward status is `NONE` before initialization.
- Update reward states to `INITIALIZED` and set `checker` values.
- Emit `ReportInitialized`.

Function: `pushValidatorWeights(ValidatorWeight[] calldata weights)`

- Caller has `FEEDER_ROLE`.
- Check `weights.length` is within range.
- Check current report status is `INITIALIZED`.
- Check each `weight.addr` is unique in this epoch.
- Append weights and updates `reward.totalWeight`.
- Emit `WeightsPushed`.

Function: `finalizeReport()`

- Caller has `FEEDER_ROLE`.
- Check current report status is `INITIALIZED`.
- Check accumulated `checker.totalWeight == reward.totalWeight`.
- Check `checker.numOfValidators == _weightCount(reward)`.
- Update `reward.status == FINALIZED` and increment `nextEpoch`.
- Emit `ReportFinalized`.

Function: `revokeReport()`

- Caller has `FEEDER_ROLE`.
- Check current report status is `INITIALIZED` or `REVOKING`.
- Remove up to `MAX_WEIGHTS_PER_ACTION` entries per call, transitioning to `REVOKING` if any remain.
- When all weights are removed, clear `rewards[epoch]` and `checker` and emit `ReportRevoked(epoch)`.
- Emit `ReportRevoked`.

Test coverage

- **Initialization**
  - `test_init` verifies `owner()`, `epochFeeder()`, and `FEEDER_ROLE()` values.

- **Report initialization**
  - `test_report` rejects nonfeeder and emits `ReportInitialized` for valid feeder.
- **Pushing weights**
  - `test_report` within the same function covers unauthorized reverts and multiple `pushValidatorWeights` batches emitting correct `WeightsPushed` events.
- **Finalizing report**
  - `test_report` covers unauthorized reverts and successful `ReportFinalized` emission.
  - `test_finalizeReport_InvalidReportStatus` reverts when status is not `INITIALIZED`.
- **Invalid total weight**
  - `test_finalizeReport_InvalidTotalWeight` rejects finalization when sum of pushed weights ≠ declared `totalWeight`.
- **Invalid validator count**
  - `test_finalizeReport_InvalidValidatorCount` rejects finalization when number of weights ≠ declared `numOfValidators`.
- **Revoke report**
  - `test_revokeReport` emits `ReportRevoking` then `ReportRevoked` and confirms report data has cleared.
  - `test_revokeReport_InvalidReportStatus` reverts if status is not `INITIALIZED` or `REVOKING`.
  - `test_revokeReport_NotFeeder` reverts when caller lacks `FEEDER_ROLE`.
- **Report availability assertions**
  - `test_assertReportReady` ensures `weightCount`, `weightAt`, `weightOf`, and `summary` all revert with `ReportNotReady` before finalization.

5.32. Contract: EpochFeeder

Description

EpochFeeder manages protocol epochs and timing, providing on-chain epoch numbers and timestamps. It supports configurable interval updates for future epochs via owner-only calls. It implements a checkpoint history for constant-time epoch and time lookups.

Permission functions and invariants

`setNextInterval(uint48 interval_)`

- Caller must be the owner of the contract.
- Compute `nextEpoch` and `nextEpochTime` correctly based on last checkpoint and current time.
- Append or update the last checkpoint's interval and timestamp.

- Emit `NextIntervalSet`.

Test coverage

- **Initialization**
  - `test_init` verifies `owner()`, initial `epoch()`, `time()`, and `interval()` are all zero.
- **`setNextInterval` (nonapplied epoch)**
  - `test_setNextInterval_withNonAppliedEpoch` checks `intervalAt(0)` remains zero and `intervalAt(1)` updates to `2×INTERVAL` when called before epoch 1.
- **`setNextInterval` (applied epoch)**
  - `test_setNextInterval_withAppliedEpoch` warps to epoch 1, sets a new interval, and verifies `intervalAt(2)` is updated and that overwriting works for epoch 2.
- **Unauthorized interval update**
  - `test_setNextInterval_unauthorized` reverts when a non-owner calls `setNextInterval`.
- **Epoch-clean boundaries**
  - `test_epoch_clean` asserts `epochAt` and `timeAt` return correct epoch numbers and timestamps just before, at, and after epoch boundaries with original interval.
- **Epoch dirty after interval change**
  - `test_epoch_dirty` warps into later epochs, updates interval, and verifies `epochAt` and `timeAt` computations reflect the new interval for subsequent epochs.

6. Assessment Results

During our assessment on the scoped Mitosis contracts, we discovered 11 findings. One critical issue was found. Two were of high impact, one was of medium impact, one was of low impact, and the remaining findings were informational in nature.

6.1. Disclaimer

This assessment does not provide any warranties about finding all possible issues within its scope; in other words, the evaluation results do not guarantee the absence of any subsequent issues. Zellic, of course, also cannot make guarantees about any code added to the project after the version reviewed during our assessment. Furthermore, because a single assessment can never be considered comprehensive, we always recommend multiple independent assessments paired with a bug bounty program.

For each finding, Zellic provides a recommended solution. All code samples in these recommendations are intended to convey how an issue may be resolved (i.e., the idea), but they may not be tested or functional code. These recommendations are not exhaustive, and we encourage our partners to consider them as a starting point for further discussion. We are happy to provide additional guidance and advice as needed.

Finally, the contents of this assessment report are for informational purposes only; do not construe any information in this report as legal, tax, investment, or financial advice. Nothing contained in this report constitutes a solicitation or endorsement of a project by Zellic.