STATE MIND

# Symbiotic Middleware

2025-05-30 - 2025-05-30

# Table of contents

# 1. Project brief

| Title | Description |
|---|---|
| Client | Symbiotic |
| Project name | Symbiotic Middleware |
| Timeline | 2025-05-30 - 2025-05-30 |

## Project Log

| Date | Commit Hash | Note |
|---|---|---|
| 2025-05-30 | 5c801a099f95591e676adb7bf3eb4cf17d83b063 | Initial commit |

## Short Overview

## Overview

This is an AI generated report.

## Project Scope

The audit covered the following files:

- Master.sol
- KeyManagerLogic.sol
- ForcePauseSelfRegisterOperators.sol
- InputNormalizer.sol
- SigEddsaCurve25519.sol
- EpochManager.sol
- KeyEcdsaSecp256k1.sol
- SigBlsBn254.sol
- VaultManager.sol
- NetworkManager.sol
- ValSetVerifier.sol
- Tokens.sol
- SettlementLogic.sol
- ExtraDataStorageHelper.sol
- VaultManagerLogic.sol
- OperatorManager.sol
- BN254.sol
- SigEcdsaSecp256k1.sol
- OperatorManagerLogic.sol
- SelfRegisterOperators.sol
- Settlement.sol
- OzEIP712.sol
- KeyEddsaCurve25519.sol
- Replica.sol
- OzEIP712Logic.sol
- ConfigProviderLogic.sol
- KeyBlsBn254.sol
- WhitelistSelfRegisterOperators.sol
- SCL_sqrtMod_5mod8.sol
- SigVerifierBlsBn254Simple.sol
- PermissionManager.sol
- SharedVaults.sol
- PersistentSet.sol
- KeyManager.sol
- SigVerifierBlsBn254ZK.sol
- EpochManagerLogic.sol
- KeyTags.sol
- OzAccessControl.sol

## 2. Finding severity breakdown

All vulnerabilities discovered during the audit are classified based on their potential severity and have the following classification:

| Severity | Description |
|---|---|
| 🟥 Critical | Bugs leading to assets theft, fund access locking, or any other loss of funds to be transferred to any party. |
| 🟧 High | Bugs that can trigger a contract failure. Further recovery is possible only by manual modification of the contract state or replacement. |
| 🟨 Medium | Bugs that can break the intended contract logic or expose it to DoS attacks, but do not cause direct loss of funds. |
| 🟩 Informational | Bugs that do not have a significant immediate impact and could be easily fixed. |

Based on the feedback received from the Client regarding the list of findings discovered by the Contractor, they are assigned the following statuses:

| Status | Description |
|---|---|
| Fixed | Recommended fixes have been made to the project code and no longer affect its security. |
| Acknowledged | The Client is aware of the finding. Recommendations for the finding are planned to be resolved in the future. |

## 3. Summary of findings

| Severity | # of Findings |
|---|---|
| Critical | 4 (0 fixed, 4 acknowledged) |
| High | 23 (0 fixed, 23 acknowledged) |
| Medium | 42 (0 fixed, 42 acknowledged) |
| Informational | 40 (0 fixed, 40 acknowledged) |
| Total | 109 (0 fixed, 109 acknowledged) |

## 4. Conclusion

# 5. Findings report

| CRITICAL-01 | Critical Missing Subgroup Checks for BLS Cryptographic Inputs | Pending |
|---|---|---|

### Description

The **verify** function uses three key cryptographic inputs: **keyG1** (derived from **keyBytes**), **signatureG1** (from **signature**), and **keyG2** (from **extraData**). For BLS or related pairing-based signature schemes to be secure, these elliptic curve points must reside in their respective correct prime-order subgroups.

1. **keyG1**: This point is obtained via **KeyBlsBn254.fromBytes(keyBytes)**, which internally calls **KeyBlsBn254.wrap()**. The **wrap()** function in **KeyBlsBn254.sol** (as per finding **KEYBLS-002** for that file) checks for on-curve properties and coordinate ranges but critically omits the subgroup check. Thus, **keyG1** used in this verification might not be in the G1 prime-order subgroup **r**.
2. **signatureG1**: This point is decoded directly from the **signature** bytes using **abi.decode**. No subsequent check is performed to ensure **signatureG1** is an element of the G1 prime-order subgroup **r**.
3. **keyG2**: This point is decoded directly from the **extraData** bytes using **abi.decode**. No subsequent check is performed to ensure **keyG2** is an element of the G2 prime-order subgroup **q**.

Failure to ensure these points are in the correct subgroups can lead to severe vulnerabilities, such as small subgroup attacks (where an attacker provides points of small order to trivialize certain checks or extract information), rogue key attacks (if **keyG2** is not properly tied to **keyG1** and subgroup checks are missing), or allowing invalid signatures/keys to pass verification. The pairing precompile itself does not perform these subgroup membership checks.

Impact: Undermines the fundamental cryptographic security of the signature verification scheme. This can allow attackers to forge signatures, register keys associated with points not in the correct subgroup leading to security breaks in protocols relying on this (e.g., signature aggregation), or cause other cryptographic failures. The scheme effectively loses its intended security guarantees.

Lines: **SigBlsBn254.sol**, **21, 27, 28, 39-44**

### Recommendation

Implement comprehensive subgroup checks for all critical elliptic curve points used in the verification:

1. For **keyG1**: The **KeyBlsBn254.wrap** function (in **KeyBlsBn254.sol**) should be augmented to include a check that the G1 point is in the prime-order subgroup **r**. This typically involves multiplying the point by **r** and verifying the result is the point at infinity.
2. For **signatureG1**: After decoding **signatureG1** at line 28, add an explicit check to ensure it belongs to the G1 prime-order subgroup **r**.
3. For **keyG2**: After decoding **keyG2** at line 27, add an explicit check to ensure it belongs to the G2 prime-order subgroup **q**.

These checks require access to the subgroup orders (e.g., **BN254.FR_MODULUS** for G1 and the G2 equivalent) and scalar multiplication functions for both G1 and G2 points, which should be available in or added to the **BN254** library. If a point fails its subgroup check, the **verify** function must return **false**.

| CRITICAL-02 | Incorrect Generalized Index Used for SSZ Proof Verification | Pending |
|---|---|---|

## Description

The **ValSetVerifier** library's functions that verify inclusion of specific fields within structs (e.g., **verifyValidatorOperatorLocal**, **verifyKeyTagLocal**) incorrectly use simple base field indices (e.g., **VALIDATOR_OPERATOR_LOCAL_INDEX = 0** which is **VALIDATOR_OPERATOR_BASE_INDEX**) as the **localIndex** parameter for the **processInclusionProofSha256** function. The **processInclusionProofSha256** function's assembly logic for Merkle tree traversal (using **mod(localIndex, 2)** and **shr(1, localIndex)**) strictly requires its **localIndex** parameter to be the SSZ generalized index (g-index) of the leaf relative to the provided root. Using a simple field index (like 0, 1, 2) instead of the correctly calculated SSZ g-index (e.g., **(uint256(1) << CONTAINER_TREE_HEIGHT) + FIELD_BASE_INDEX**) will cause the Merkle proof path reconstruction to be incorrect, leading to verification failures for all valid proofs of struct fields.
Furthermore, for proofs of elements within lists (e.g., **verifyValidatorRootLocal**), the **localIndex** parameter (such as **validatorRootLocalIndex**) is also used directly in **processInclusionProofSha256**. The range checks for these indices use ***_MIN_LOCAL_INDEX** constants (e.g., **VALIDATOR_ROOT_MIN_LOCAL_INDEX**). These ***_MIN_LOCAL_INDEX** constants are calculated in a way that results in 0 (e.g., **VALIDATORS_LIST_LOCAL_INDEX << (1 + VALIDATORS_LIST_TREE_HEIGHT)** where **VALIDATORS_LIST_LOCAL_INDEX** is 0). This means that **validatorRootLocalIndex** can be 0. However, SSZ generalized indices are 1-based from the conceptual root, and the g-index for the first element (index 0) of a list is typically **(uint256(1) << (LIST_TREE_HEIGHT + 1)) + 0**, which is non-zero. Passing **localIndex = 0** to **processInclusionProofSha256** for the first element of a list will also lead to incorrect proof verification.
Impact: All Merkle proof verifications for fields within structs, and for elements within lists, will consistently fail for valid proofs because the proof traversal logic in **processInclusionProofSha256** will use an incorrect path due to the wrong generalized index. This renders the core SSZ proof verification functionality of the library unusable.
Lines: **ValSetVerifier.sol**,
**89, 117, 121, 125, 129, 133, 235-237, 245, 250-261, 294-296, 299, 308-310, 316, 321-324, 331, 344, 354, 367, 394**

## Recommendation

1. For proofs of fields within a container (e.g., **operator** within **Validator** structure): The **localIndex** passed to **processInclusionProofSha256** must be calculated as the SSZ generalized index: **(uint256(1) << CONTAINER_TREE_HEIGHT) + FIELD_BASE_INDEX**. For example, in **verifyValidatorOperatorLocal**, the **localIndex** should be **(uint256(1) << VALIDATOR_TREE_HEIGHT) + VALIDATOR_OPERATOR_BASE_INDEX** instead of just **VALIDATOR_OPERATOR_LOCAL_INDEX**. This correction needs to be applied to all similar functions that verify fields within structs (**verifyValidatorVotingPowerLocal**, **verifyValidatorIsActiveLocal**, **verifyKeyTagLocal**, **verifyKeyPayloadHash**, **verifyVaultChainIdLocal**, **verifyVaultVaultLocal**, **verifyVaultVotingPowerLocal**).
2. For proofs of elements within a list (e.g., a validator root within the **validators** list which is part of **ValidatorSet**): The **localIndex** parameter passed to **processInclusionProofSha256** (e.g., **validatorRootLocalIndex** in **verifyValidatorRootLocal**) must be the SSZ generalized index of that list element relative to the root against which it's being proven (e.g., **validatorSetRoot**). If **validatorSetRoot** is the root of the list itself (i.e., **VALIDATOR_SET_TREE_HEIGHT = 0**), and **validatorRootLocalIndex** is intended to be the 0-based array index of the element in the list, then it must be converted to the generalized index before being passed to **processInclusionProofSha256**. The g-index for element **i** in a list of tree height **LH** is **(uint256(1) << (LH + 1)) + i**. The range check constants like **VALIDATOR_ROOT_MIN_LOCAL_INDEX** and **VALIDATOR_ROOT_MAX_LOCAL_INDEX** must then be defined as bounds for this array index **i**, not for the g-index itself if the parameter is an array index. Alternatively, if the parameter is expected to be the g-index, then **VALIDATOR_ROOT_MIN_LOCAL_INDEX** should be **(uint256(1) << (VALIDATORS_LIST_TREE_HEIGHT + 1)) + 0**.

| CRITICAL-03 | Missing Previous Header Hash Consistency Check in setValSetHeader | Pending |
|---|---|---|

## Description

The **setValSetHeader** function is responsible for storing a new **ValSetHeader**. This header structure contains a **previousHeaderHash** field, which is intended to link it to the preceding header, forming a chain. However, **setValSetHeader** stores **header.previousHeaderHash** (line 370) without verifying its consistency against the actual hash of the previously committed validator set header. While the **commitValSetHeader** function ensures the new **header** (including its **previousHeaderHash** field) is signed by a quorum of validators from a previous epoch, it does not guarantee that the **previousHeaderHash** field correctly corresponds to the on-chain historical header state.
Impact: If this check is missing, a malicious quorum of validators (or a compromised submission process) could commit a new validator set header whose **previousHeaderHash** field does not match the hash of the actual last committed header. This could effectively fork the header chain or create a header that links to an arbitrary or non-existent previous state, breaking the integrity and continuity of the settlement layer's historical state. This could lead to confusion for light clients, disrupt services relying on the header chain's integrity, and potentially enable more complex attacks if parts of the system trust this broken chain.
Lines: **SettlementLogic.sol**, **339-380, 370**

## Recommendation

Implement a check within **setValSetHeader** to validate the **header.previousHeaderHash** field. This check should ensure that **header.previousHeaderHash** correctly matches the hash of the most recently committed validator set header on-chain. Specifically, in **setValSetHeader**:

```
ISettlement.SettlementStorage storage $ = _getSettlementStorage();
uint48 currentEpoch = EpochManagerLogic.getCurrentEpoch(); // This is header.epoch

bytes32 expectedPreviousHeaderHash;
if (currentEpoch == 0) {
  // For the very first header (genesis for epoch 0), previousHeaderHash is typically bytes32(0).
  expectedPreviousHeaderHash = bytes32(0);
} else {
  // For subsequent headers, previousHeaderHash should match the hash of the header
  // from the last successfully committed epoch (_lastCommittedHeaderEpoch).
  // This ensures linking to the true predecessor on-chain, handling skipped epochs.
  ISettlement.ValSetHeader storage lastCommittedValSetHeader = $._valSetHeader[$._lastCommittedHeaderEpoch];
  // Ensure lastCommittedValSetHeader actually exists if _lastCommittedHeaderEpoch is not a sentinel for 'no prior
header'
  if (lastCommittedValSetHeader.version == 0 && $._lastCommittedHeaderEpoch != 0) {
    // This case should ideally not happen if _lastCommittedHeaderEpoch is always valid after epoch 0 genesis
    revert ISettlement.Settlement_InternalError(); // Or a more specific error
  }
  if ($._lastCommittedHeaderEpoch == 0 && currentEpoch > 0 && lastCommittedValSetHeader.version == 0) {
    // This could mean genesis for epoch 0 never actually set _lastCommittedHeaderEpoch properly, or logic is flawed.
    // If epoch 0 header was stored, lastCommittedValSetHeader (for epoch 0) should have version > 0.
    // For epoch 1, it should link to epoch 0's header.
    // This specific edge case implies _lastCommittedHeaderEpoch might not be updated before first non-genesis
commitValSetHeader.
    // A robust way for currentEpoch > 0:
    // require($._valSetHeader[currentEpoch -1].version > 0, "Previous epoch header must exist for linking");
    // expectedPreviousHeaderHash = keccak256(abi.encode($._valSetHeader[currentEpoch -1]));
    // This simpler continuous chain logic needs to be reconciled with PROLONG phase and
_lastCommittedHeaderEpoch.
    // The original suggestion of using _lastCommittedHeaderEpoch is likely more robust for PROLONG.
  }
  // Assuming _lastCommittedHeaderEpoch correctly points to the *actual* last committed header's epoch
  expectedPreviousHeaderHash = keccak256(abi.encode(lastCommittedValSetHeader));
}

if (header.previousHeaderHash != expectedPreviousHeaderHash) {
  revert ISettlement.Settlement_InvalidPreviousHeaderHash();
}
```

The exact logic for determining **expectedPreviousHeaderHash** needs careful consideration of how epoch 0 genesis is handled and how **_lastCommittedHeaderEpoch** is updated relative to **currentEpoch** during **setValSetHeader** execution, especially for the very first call to **commitValSetHeader** after genesis. The core idea is to ensure **header.previousHeaderHash** matches the on-chain reality of the immediate valid predecessor.

| CRITICAL-04 | Public Functions Allow Unauthenticated Key Registration for Arbitrary Operators | Pending |
|---|---|---|

### Description

The **KeyManagerLogic.sol** library, based on its provided summary, contains several **public** functions for setting keys: **setKey(address operator, uint8 keyTag, bytes memory key)**, **setKey32(address operator, uint8 tag, bytes memory key)**, and **setKey64(address operator, uint8 tag, bytes memory key)**. These specific overloads directly allow registering a key payload for any given **operator** without performing any signature verification or other authentication to confirm the caller's authority over the **operator** account. They directly write to the key storage (**_keys32** or **_keys64** checkpoints). While other **setKey** overloads in the library properly implement signature verification, the public visibility of these unauthenticated versions poses a severe risk if the contract delegating to this library (e.g., **KeyManager.sol**) exposes them without its own stringent access controls.
Impact: If these unauthenticated key-setting functions are exposed publicly by the main **KeyManager** contract or any contract that **delegatecall**s to **KeyManagerLogic**, an attacker could call them to register arbitrary keys for any operator. This would allow attackers to impersonate legitimate operators, potentially leading to unauthorized control over assets, manipulation of governance, or disruption of system operations that rely on authentic operator keys.
Lines: **KeyManagerLogic.sol**, **228-239, 241-244, 246-251**

### Recommendation

Change the visibility of **setKey(address operator, uint8 keyTag, bytes memory key)**, **setKey32(address operator, uint8 tag, bytes memory key)**, and **setKey64(address operator, uint8 tag, bytes memory key)** from **public** to **internal**. These functions should only be callable as low-level helpers by other **setKey** functions within the **KeyManagerLogic** library that have already performed necessary authentication (e.g., EIP-712 signature verification) and other high-level checks.

| HIGH-01 | Incorrect Proof Height Used in **verifyKeyTagLocal** | Pending |
|---|---|---|

### Description

The **verifyKeyTagLocal** function is intended to verify a key tag's inclusion within a **keyRoot**. The **keyRoot** is the root of a **Key** struct, which has 2 fields (**tag**, **payloadHash**) and thus a Merkle tree height of **KEY_TREE_HEIGHT = 1**. However, the function incorrectly passes **KEY_ROOT_PROOF_EXPECTED_HEIGHT** as the **expectedHeight** to **processInclusionProofSha256**. **KEY_ROOT_PROOF_EXPECTED_HEIGHT** is defined as **VALIDATOR_TREE_HEIGHT + 1 + KEY_LIST_TREE_HEIGHT** (e.g., 3 + 1 + 7 = 11), which is the height for proving a **keyRoot** within a **validatorRoot**. The correct height for proving a field (like **tag**) within **keyRoot** should be **KEY_TAG_PROOF_EXPECTED_HEIGHT** (which is **KEY_TREE_HEIGHT = 1**).
Impact: This bug will cause **verifyKeyTagLocal** to always return **false** for valid key tag proofs of height 1, because the proof length check **proof.length != expectedHeight** (e.g., **1 != 11**) in **processInclusionProofSha256** will fail. This leads to a denial of service (DoS) as valid keys cannot be verified. If an attacker could craft a specific proof of the incorrect height (e.g., 11) that coincidentally validates, it might lead to incorrect validation, although this is less likely than DoS.
Lines: **ValSetVerifier.sol**, **321-324, 107, 119**

### Recommendation

In **verifyKeyTagLocal**, change the **expectedHeight** argument passed to **processInclusionProofSha256** from **KEY_ROOT_PROOF_EXPECTED_HEIGHT** to the correct constant **KEY_TAG_PROOF_EXPECTED_HEIGHT**.

```
--- a/ValSetVerifier.sol
+++ b/ValSetVerifier.sol
@@ -321,7 +321,7 @@
    function verifyKeyTagLocal(SszProof calldata keyTagProof, bytes32 keyRoot) internal view returns (bool) {
        return processInclusionProofSha256(
-           keyTagProof.proof, keyTagProof.leaf, keyRoot, KEY_TAG_LOCAL_INDEX, KEY_ROOT_PROOF_EXPECTED_HEIGHT
+           keyTagProof.proof, keyTagProof.leaf, keyRoot, KEY_TAG_LOCAL_INDEX, KEY_TAG_PROOF_EXPECTED_HEIGHT
        );
    }
```

| HIGH-02 | Incorrect negation of G1 points with Y-coordinate equal to zero | Pending |
|---|---|---|

### Description

The **negate** function calculates the negation of a G1Point (X, Y) as (X, FP_MODULUS - (Y % FP_MODULUS)). If Y is 0 (for a point on the x-axis, distinct from the point at infinity), Y % FP_MODULUS is 0. The expression for the new Y-coordinate becomes FP_MODULUS - 0 = FP_MODULUS. This results in the Y-coordinate of the negated point being FP_MODULUS. However, the correct negation of a point (X, 0) is (X, 0). Furthermore, precompiled contracts for elliptic curve operations (like ECADD, ECMUL, ECPAIRING) expect point coordinates to be strictly less than FP_MODULUS. A Y-coordinate of FP_MODULUS is an invalid input for these precompiles and will cause them to fail. This bug effectively renders points negated from (X,0) unusable in subsequent precompile operations.
Impact: Any cryptographic operation (e.g., addition, scalar multiplication, pairing) that uses a G1 point resulting from **negate** where the original point had Y=0 (and X!=0) will fail due to the malformed Y-coordinate being passed to an Ethereum precompile. This can lead to denial of service for contract functionalities relying on this library for BN254 operations, or cause transactions to unexpectedly revert.
Lines: **BN254.sol**, **77**

### Recommendation

Modify the negation logic to correctly handle points where Y=0. The Y-coordinate of the negation of (X,0) should be 0. Specifically, line 77 should be changed to correctly compute the negated Y-coordinate. For example: **return G1Point(p.X, (p.Y == 0) ? 0 : (FP_MODULUS - (p.Y % FP_MODULUS)));** Assuming **p.Y** is already canonical (i.e., **0 <= p.Y < FP_MODULUS**), a simpler form could be: **return G1Point(p.X, (p.Y == 0) ? 0 : (FP_MODULUS - p.Y));**

| HIGH-03 | Dependency on Unaudited SqrtMod Implementation | Pending |
|---|---|---|

### Description

The **KeyEddsaCurve25519** library's **decompress** function, which is critical for key validation in **wrap** and **fromBytes**, relies on **SqrtMod** imported from **../../utils/unaudited/SCL_sqrtMod_5mod8.sol**. The path explicitly marks this utility as "unaudited". Bugs, vulnerabilities (e.g., incorrect results, excessive gas consumption for certain inputs, exploitable edge cases) or inefficiencies in an unaudited cryptographic primitive can compromise the security and correctness of the key validation logic.
Impact: If **SqrtMod** is flawed, invalid keys might be accepted as valid, valid keys might be rejected, or denial-of-service could occur due to excessive gas consumption or unexpected reverts. This would undermine any security assumptions based on the validity of Ed25519 keys.
Lines: **KeyEddsaCurve25519.sol**, **7, 32-37, 68, 88**

### Recommendation

The **SCL_sqrtMod_5mod8.sol** library should undergo a thorough security audit. Alternatively, replace it with a well-audited and standard implementation for modular square roots if available. Ensure the chosen **SqrtMod** implementation is correct, secure, and gas-efficient for all possible inputs.

| HIGH-04 | Missing Subgroup Check for BLS Keys | Pending |
|---|---|---|

### Description

The **wrap** function (lines 20-37) validates that a **BN254.G1Point** has coordinates within the field Fp and lies on the elliptic curve **y^2 = x^3 + b**. However, for BLS cryptography, public keys (G1 points) must also reside in the prime-order subgroup **r**. The **wrap** function does not perform this subgroup check. Accepting keys that are not in the correct subgroup can lead to several security vulnerabilities in a BLS scheme, including ineffective signature aggregation, and potentially rogue key attacks, breaking the cryptographic security assumptions of the BLS scheme.
Impact: If keys not in the prime-order subgroup **r** are accepted and used, it can undermine the security of the BLS signature scheme. This may enable attacks such as rogue key attacks in aggregation scenarios or lead to signatures that do not provide the intended security guarantees.
Lines: **KeyBlsBn254.sol**, **20-37**

### Recommendation

In the **wrap** function, after existing validations (coordinates in Fp, point on curve), add a check to ensure that the input point **keyRaw** belongs to the prime-order subgroup **r**. This is typically done by multiplying the point by the subgroup order **r** and verifying that the result is the point at infinity (0,0). This requires **BN254.scalar_mul** function and the constant **r** (the prime order of the G1 subgroup for BN254).
Example addition to **wrap** (conceptual):

```
// ... existing checks ...
// uint256 R_SUBGROUP_ORDER = ...; // Define the BN254 G1 subgroup order
// BN254.G1Point memory pointAtInfinity = BN254.G1Point({X: 0, Y: 0});
// BN254.G1Point memory checkPoint = BN254.scalar_mul(R_SUBGROUP_ORDER, keyRaw); // Requires scalar_mul
// if (checkPoint.X != pointAtInfinity.X || checkPoint.Y != pointAtInfinity.Y) {
//     revert KeyBlsBn254_InvalidKey(); // Not in correct subgroup
// }
key = KEY_BLS_BN254(keyRaw);
```

This assumes **keyRaw** at this stage is a valid point on the curve with coordinates in Fp.

| HIGH-05 | Missing Access Control on **setEpochDuration** Function | Pending |
|---|---|---|

### Description

The **setEpochDuration** function in **Settlement.sol** (lines 279-283) overrides the corresponding function from its parent contract, **EpochManager**. The **EpochManager.setEpochDuration** function is protected by the **checkPermission** modifier, which restricts its invocation to authorized entities. However, the overriding implementation in **Settlement.sol** omits this **checkPermission** modifier. This effectively removes the access control, allowing any external account to call this function. The function delegates its logic to **SettlementLogic.setEpochDuration**, and the summary for **SettlementLogic** indicates that its state-modifying functions, including **setEpochDuration**, require external access control to be enforced by the calling contract (**Settlement.sol** in this case).
Impact: Unauthorized actors could change the system's epoch duration. This can disrupt critical network operations, validator reward calculations, consensus mechanism timings, and other time-sensitive functionalities that depend on a correctly configured epoch duration, potentially leading to instability or exploits.
Lines: **Settlement.sol**, **279-283**

### Recommendation

Reinstate the **checkPermission** modifier on the **setEpochDuration** function in **Settlement.sol** to ensure it aligns with the access control policy of the overridden function and the design expectation of **SettlementLogic**. The function signature should be: **function setEpochDuration(uint48 epochDuration) public virtual override checkPermission { ... }**.

| HIGH-06 | Non-Canonical Point Coordinates Used in Challenge Hash Calculation | Pending |
|---|---|---|

### Description

The **verify** function decodes **signatureG1** from **signature** (line 28) and **keyG2** from **extraData** (line 27) using **abi.decode**. This decoding process does not ensure that the coordinate values of these points are canonical (i.e., less than the field modulus **BN254.FP_MODULUS**). While elliptic curve precompiles typically handle non-canonical coordinates by reducing them modulo the field prime, this function uses the raw, potentially non-canonical, coordinate values of **signatureG1** and **keyG2** as inputs to **abi.encodePacked** for the **keccak256** hash that generates the challenge scalar **alpha** (lines 33-35). In contrast, **keyG1**'s coordinates are canonical due to validation in **KeyBlsBn254.fromBytes**, and **messageG1** from **BN254.hashToG1** is also expected to have canonical coordinates. If an attacker can provide **signature** or **extraData** bytes that decode to non-canonical coordinates (e.g., **X_coord + k*P** where **P** is the modulus), these byte strings would represent the same cryptographic points for pairing purposes (after reduction by precompiles) but would produce different **alpha** values. This malleability in the challenge **alpha** based on the representation of input coordinates can compromise the security of the Fiat-Shamir heuristic used to generate the challenge.

Impact: An attacker could potentially manipulate the challenge scalar **alpha** by submitting **signature** or **extraData** with non-canonical coordinates. If they can find a non-canonical representation that yields a favorable **alpha**, it might allow them to forge a signature, bypass checks, or otherwise break the intended security properties of the signature verification scheme. Using non-canonical inputs in cryptographic hashes for challenges is a deviation from standard secure practices and can lead to vulnerabilities.

Lines: **SigBlsBn254.sol**, **27, 28, 33-35**

### Recommendation

Ensure that only canonical coordinates for **signatureG1** and **keyG2** are used in the calculation of **alpha**. After decoding **signatureG1** and **keyG2**, explicitly validate that all their coordinate components are strictly less than **BN254.FP_MODULUS**. If any coordinate is not canonical, the function should return **false** or revert.

Example for **signatureG1** (a G1 point):

```
// After line 28
if (signatureG1.X >= BN254.FP_MODULUS || signatureG1.Y >= BN254.FP_MODULUS) {
    return false; // Or revert
}
```

Example for **keyG2** (a G2 point, assuming its coordinates are also Fp elements):

```
// After line 27
if (keyG2.X[0] >= BN254.FP_MODULUS || keyG2.X[1] >= BN254.FP_MODULUS ||
    keyG2.Y[0] >= BN254.FP_MODULUS || keyG2.Y[1] >= BN254.FP_MODULUS) {
    return false; // Or revert
}
```

This ensures that the inputs to the **keccak256** hash for **alpha** are standardized, preventing manipulation through non-canonical coordinate representations.

| HIGH-07 | Initializer Can Be Front-Run Leading to Malicious Configuration | Pending |
|---|---|---|

### Description

The **initialize** function in **Replica.sol** is **public** and protected by the **initializer** modifier, ensuring it can only be called once. However, if the contract deployment and its initialization are not performed atomically (e.g., in the same transaction or by a secured, immediate subsequent transaction), an attacker can monitor the deployment of a **Replica** contract instance. The attacker could then call the **initialize** function before the legitimate deployer, providing their own address as **defaultAdmin** and potentially supplying malicious **settlementInitParams**. This would grant the attacker control over the **DEFAULT_ADMIN_ROLE** and allow them to set up the initial parameters of the **Settlement** component to values that could compromise or disrupt the system.

Impact: If an attacker successfully front-runs the **initialize** call, they can set themselves as the **defaultAdmin**. This gives them full control over role management via **OzAccessControl**, allowing them to grant or revoke any roles. Furthermore, by controlling **settlementInitParams**, the attacker could configure the underlying **Settlement** module with malicious parameters (e.g., a compromised signature verifier, incorrect key tags, or disruptive epoch/commit durations), potentially leading to a non-functional or compromised settlement system, including denial of service or theft of assets if the settlement mechanism is subverted.

Lines: **Replica.sol**, **14-22**

### Recommendation

To prevent front-running of the initializer:

1. Atomic Initialization: If deploying via a factory contract, the factory should deploy the **Replica** instance and call its **initialize** function with legitimate parameters in the same transaction.
2. Immediate Initialization: If deploying manually, the deployer (EOA) should call **initialize** in the transaction immediately following the deployment transaction, minimizing the window for front-running.
3. Access Controlled Initializer (Alternative for Proxies): For upgradeable proxy patterns (like UUPS where **Replica** might be an implementation), the **initialize** function is typically public on the implementation. The actual initialization call via the proxy should be restricted to the deployer or proxy admin. Ensure this pattern is correctly followed if **Replica** is used with proxies.
4. Internal Initializer with Constructor: If **Replica** were not intended to be upgradeable or part of such a proxy pattern (less likely given the **initializer** modifier), its initialization logic could be moved to an **internal** function called by a constructor that takes the necessary parameters, ensuring it's set at deployment time.

| HIGH-08 | Incorrect Invariant Check in **setEpochDuration** Can Lead to **epochDuration <= commitDuration** | Pending |
|---|---|---|

## Description

The **setEpochDuration** function (lines 259-266) in **SettlementLogic.sol** aims to ensure that the **epochDuration** is always greater than the **commitDuration**. It performs a check:
**if (epochDuration <= _getSettlementStorage()._commitDuration.latest())**. The **_commitDuration.latest()** method retrieves the **commitDuration** value that is currently active. However, a new **epochDuration** set via **EpochManagerLogic.setEpochDuration(epochDuration)** typically becomes effective from the start of the next epoch.
The issue arises if **commitDuration** is also scheduled to change at the start of that same next epoch (due to a prior call to **SettlementLogic.setCommitDuration**, which pushes changes keyed by **EpochManagerLogic.getNextEpochStart()**). In such a scenario, the check in **setEpochDuration** compares the new **epochDuration** (for the next epoch) against the current **commitDuration** (for the current epoch), not the **commitDuration** that will be active alongside the new **epochDuration** in the next epoch. This can lead to the check passing, but the invariant **epochDuration > commitDuration** being violated in the subsequent epoch.
Impact: If an epoch occurs where **epochDuration** is less than or equal to **commitDuration**, the system's phase logic ( **getCurrentPhase**) will likely break. Commitment windows might effectively be non-existent or misinterpreted. This could prevent new validator set headers from being committed, stall the settlement layer's progression, or lead to other unpredictable behaviors due to violated timing assumptions.
Lines: **SettlementLogic.sol**, **262-264**

## Recommendation

Modify the check within **setEpochDuration** to compare the proposed **epochDuration** against the **commitDuration** that will be effective at the same future time (i.e., at **EpochManagerLogic.getNextEpochStart()**).
This can be achieved by looking up the value of the **_commitDuration** trace at the timestamp corresponding to the start of the next epoch:

```
function setEpochDuration(
    uint48 epochDuration
) public {
    // Determine the timestamp when the new epochDuration will become effective.
    uint48 effectiveTimestamp = EpochManagerLogic.getNextEpochStart();

    // Retrieve the commitDuration that will be active at that effectiveTimestamp.
    // This requires using a method that correctly finds the checkpoint value for a given time.
    // Assuming the Checkpoints.Trace208 from OpenZeppelin or similar, upperLookupRecent is appropriate.
    // Note: Ensure correct error handling if no checkpoint exists before effectiveTimestamp.
    uint48 futureCommitDuration;
    Checkpoints.Trace208 storage commitDurationTrace = _getSettlementStorage()._commitDuration;
    if (commitDurationTrace.length() == 0) {
        revert ISettlement.Settlement_InternalError(); // Or appropriate error, should not happen if initialized
    }
    futureCommitDuration = uint48(commitDurationTrace.upperLookupRecent(effectiveTimestamp, bytes("")));
    // If upperLookupRecent reverts on no value <= key, need try-catch or pre-check.
    // If it returns 0 or some default, ensure this is handled if it means 'not yet set for future'.

    if (epochDuration <= futureCommitDuration) {
        revert ISettlement.Settlement_EpochDurationTooShort();
    }
    EpochManagerLogic.setEpochDuration(epochDuration);
}
```

This ensures the invariant **epochDuration > commitDuration** is checked using values that will be contemporaneously active.

| HIGH-09 | Timestamp Calculation Overflow Leads to Incorrect Epoch Start Times | Pending |
|---|---|---|

### Description

Functions **getCurrentEpochStart()**, **getNextEpochStart()**, and **getEpochStart(epoch, hint)** calculate epoch start timestamps and return them as **uint48**. These calculations involve a term **(epoch_difference) * epochDuration**. If **epoch_difference** (number of epochs elapsed since a reference epoch, e.g., **getCurrentEpoch() - epochDurationIndex**) multiplied by **epochDuration** results in a value that, when added to **epochDurationTimestamp**, exceeds **type(uint48).max**, the final **uint48** result will wrap around due to overflow. This leads to the functions returning incorrect, often much earlier, epoch start times.
Impact: Incorrect epoch start times can severely disrupt system logic that relies on them. This includes determining the current phase of an epoch, eligibility for time-sensitive actions, reward calculations, or synchronization with external events. It can lead to denial of service for operations, incorrect state transitions, failed transactions, or potential economic exploits if financial logic is tied to these faulty timestamps.
Lines: **EpochManagerLogic.sol**, **45-48, 54-56, 69-73**

### Recommendation

Implement overflow checks for timestamp calculations. Before casting the final result to **uint48**, ensure the calculated value (as **uint256**) does not exceed **type(uint48).max**. For example, in **getEpochStart** (and similarly for others):

```
// ... inside getEpochStart, after getting epochDuration, epochDurationTimestamp, epochDurationIndex
if (epoch < epochDurationIndex) {
   revert IEpochManager.EpochManager_InvalidEpoch(); // Or appropriate error
}
uint256 numElapsedEpochs = uint256(epoch) - epochDurationIndex;
uint256 durationOfElapsedEpochs = numElapsedEpochs * epochDuration;

uint256 calculatedEpochStart = uint256(epochDurationTimestamp) + durationOfElapsedEpochs;

if (calculatedEpochStart > type(uint48).max) {
   revert IEpochManager.EpochManager_EpochStartTimeOverflow(); // Define this error
}
return uint48(calculatedEpochStart);
```

This ensures that if an overflow would occur, the function reverts instead of returning a misleading timestamp.

| HIGH-10 | Constructor Reads Uninitialized State Variables for Input Validation | Pending |
|---|---|---|

### Description

The constructor logic iterates and checks lengths based on **verifiers.length** and **maxValidators.length** (state variables) before these state variables are assigned values from the input parameters **verifiers_** and **maxValidators_**. At this point, the state variables are empty (length 0), causing all validation loops (e.g., checking for non-zero **maxValidators** values, strictly increasing order) to be skipped.
Impact: The contract can be deployed with invalid or empty **verifiers** and **maxValidators** arrays. This can lead to **_getVerifier** always reverting or selecting incorrect verifiers if arrays are malformed, effectively rendering **verifyQuorumSig** unusable or insecure.
Lines: **SigVerifierBlsBn254ZK.sol**, **48-58**

### Recommendation

Modify the constructor's validation logic to use the input parameters **verifiers_** and **maxValidators_** (e.g., **verifiers_.length**, **maxValidators_[i]**) for all checks before assigning them to the state variables.

| HIGH-11 | Direct Calls to setKey32/setKey64 Bypass Critical Validation and State Updates | Pending |
|---|---|---|

### Description

The helper functions **setKey32** (line 241) and **setKey64** (line 246) are declared **public**. These functions are intended to be called internally by the main **setKey** logic after signature verification and key reuse checks. If called directly by an external actor, they bypass all these critical validation steps, including signature verification for key ownership and key reuse policies. Additionally, they circumvent updates to several important state mappings (**_operatorByKeyHash**, **_operatorByTypeAndKeyHash**, **_operatorByTagAndKeyHash**), the **_operators** set, and the **_operatorKeyTags** checkpoint. Only the raw key payload (**_keys32** or **_keys64**) is stored in its respective checkpointed trace.
Impact: An attacker could store arbitrary key payloads for any operator and tag without proper authorization (signature), bypassing ownership checks. This leads to a state where **getKey(operator, tag)** might return a key, but other crucial functions like **getOperator(key)** or **getKeyTags(operator)** would not reflect this registration correctly, causing system inconsistency and potentially allowing unauthorized actions if other parts of the system rely on this partial state. Key reuse policies enforced by the main **setKey** function would also be bypassed.
Lines: **KeyManagerLogic.sol**, **241-244, 246-251**

### Recommendation

Change the visibility of **setKey32** and **setKey64** from **public** to **internal**. These functions should only be callable from within the **KeyManagerLogic** library itself, specifically by the **setKey(address operator, uint8 keyTag, bytes memory key)** function, to ensure that all prerequisite checks and state updates are performed.

| HIGH-12 | Direct Call to setKey(operator, keyTag, key) Overload Bypasses Signature Verification and Key Management Logic | Pending |
|---|---|---|

### Description

The **setKey(address operator, uint8 keyTag, bytes memory key)** function (lines 228-239) is declared **public**. While this function is legitimately called by the primary **setKey** overloads (which perform signature verification and comprehensive key state management), making it **public** allows it to be called directly from outside. A direct call to this overload bypasses critical security steps: EIP-712 signature verification for key ownership, checks for key reuse (e.g., preventing the same key from being used by different operators or for the same type under different tags by the same operator), and the logic that updates associated state mappings (**_operatorByKeyHash**, **_operatorByTypeAndKeyHash**, **_operatorByTagAndKeyHash**), the **_operators** set, and the **_operatorKeyTags** checkpoint. It only performs key serialization and stores its compressed form using **setKey32**.
Impact: Attackers can register key payloads for any operator without their consent by bypassing signature verification. Key reuse policies are not enforced, potentially allowing key registrations that the main **setKey** logic would prevent. This can lead to significant state inconsistency: **getKey(operator, tag)** might return a key, but other functions like **getOperator(key)** or **getKeyTags(operator)** would not reflect this registration correctly, undermining the integrity of the key management system.
Lines: **KeyManagerLogic.sol**, **228-239**

### Recommendation

Change the visibility of **setKey(address operator, uint8 keyTag, bytes memory key)** from **public** to **internal**. This will ensure it can only be called by the other **setKey** overloads within the library, after all necessary validations (including signature checks) and preliminary state updates have been completed.

| HIGH-13 | Potential Division by Zero if Epoch Duration is Set to Zero | Pending |
|---|---|---|

### Description

The epoch duration can be set via **__EpochManager_init** (through **EpochManagerInitParams.initialEpochDuration**) and the public **setEpochDuration** function. These calls delegate to **EpochManagerLogic** for state updates. Neither **EpochManager.sol** nor the provided summaries for **EpochManagerLogic.sol** indicate explicit validation preventing the epoch duration from being set to zero. If the epoch duration is set to zero, functions within **EpochManagerLogic** that calculate epoch indices or other time-dependent values (e.g., **getCurrentEpoch**, **getEpochIndex**) by dividing by the **epochDuration** will revert due to a division-by-zero error.
Impact: If the epoch duration is set to zero, core system functionalities that rely on epoch calculations (such as retrieving the current epoch, epoch start times, or indexing by timestamp) will become permanently unavailable, leading to a Denial of Service. This would persist until a non-zero epoch duration can be set, assuming the **setEpochDuration** function itself remains operable and does not also depend on non-faulting epoch calculations.
Lines: **EpochManager.sol**, **20-24, 92-96**

### Recommendation

Modify the underlying **EpochManagerLogic** (specifically the internal function responsible for writing epoch duration checkpoints, e.g., **setEpochDurationInternal**) to validate that the **epochDuration** parameter is strictly greater than zero. For example, add a check like: **require(epochDuration > 0, "EpochManagerLogic: Epoch duration must be positive");**. This check should be applied both during initialization and when **setEpochDuration** is called.

| HIGH-14 | Incomplete Initialization of Inherited Base Contracts in __Settlement_init | Pending |
|---|---|---|

## Description

The **__Settlement_init** function in **Settlement.sol** (lines 35-39) only calls **SettlementLogic.initialize(settlementInitParams)**. It does not call the initializers of its direct base contracts: **__NetworkManager_init**, **__EpochManager_init**, and **__OzEIP712_init**. These base contract initializers are responsible for setting up their own state (including their **Initializable** state for versioning and re-entrancy protection during initialization) and, critically, for initializing their own respective base contracts. For example, **EpochManager**'s **__EpochManager_init** function also calls **__PermissionManager_init_unchained()** (as **EpochManager** inherits **PermissionManager**). By not calling these intermediate base contract initializers, **__Settlement_init** breaks the standard OpenZeppelin upgradeable contract initialization chain. This can lead to parts of the inherited functionality (like **PermissionManager** state within **EpochManager**) being uninitialized or improperly configured, and the **_initialized** versioning flags for **NetworkManager**, **EpochManager**, and **OzEIP712** not being set, potentially causing issues with future upgrades or incorrect behavior of the **onlyInitializing** modifier if it relies on per-contract initialized state.
Impact: Critical components inherited by **Settlement** (e.g., **PermissionManager** via **EpochManager**) may remain uninitialized, leading to their malfunction or security vulnerabilities (e.g., permission checks failing or defaulting incorrectly). The **Initializable** state for base contracts like **NetworkManager**, **EpochManager**, and **OzEIP712** will not be correctly set, which could complicate future upgrades or lead to unexpected behavior if these contracts' **onlyInitializing** logic or **_initialized** version flags are important for their operation or upgradeability. This constitutes a significant deviation from secure upgradeable contract patterns.
Lines: **Settlement.sol**, **35-39**

## Recommendation

Modify **__Settlement_init** to follow the standard OpenZeppelin upgradeable contract initialization pattern by explicitly calling the initializers of its direct base contracts. This typically involves passing specific initialization parameters for each base component.
Example structure for **__Settlement_init**:

```
function __Settlement_init(
    SettlementInitParams memory settlementInitParams,
    NetworkManagerInitParams memory networkManagerInitParams, // Add if NetworkManager needs params
    EpochManagerInitParams memory epochManagerInitParams,     // Add if EpochManager needs params
    OzEIP712InitParams memory ozEIP712InitParams          // Add if OzEIP712 needs params
) internal virtual onlyInitializing {
    // Call base contract initializers first
    __NetworkManager_init(networkManagerInitParams);       // Or _init_unchained if appropriate
    __EpochManager_init(epochManagerInitParams);          // Or _init_unchained
    __OzEIP712_init(ozEIP712InitParams);             // Or _init_unchained
    // MulticallUpgradeable typically does not have a parameterized initializer beyond Initializable

    // Then, initialize Settlement specific logic
    SettlementLogic.initialize(settlementInitParams);
}
```

Ensure that **SettlementInitParams** struct is adjusted if it was previously expected to contain sub-parameters for these base logic modules, as **SettlementLogic.initialize** should then only handle logic unique to **Settlement** not covered by its bases.
This change ensures that each component in the inheritance hierarchy is correctly initialized, including their own **Initializable** state and any further inherited components like **PermissionManager**.

| HIGH-15 | Element Duplication and State Corruption if Initial Addition Key is Zero | Pending |
|---|---|---|

### Description

The **_add(Set storage set, uint48 key, bytes32 value)** function determines if an element is being added for the first time by checking **set._statuses[value].addedAt > 0**. If an element **value** is added for the first time with **key = 0**, **set._statuses[value].addedAt** is consequently set to 0. Any subsequent call to **_add** for the same **value** (even with a non-zero **key'**) will find **set._statuses[value].addedAt > 0** (i.e., **0 > 0**) to be false. This causes the logic to incorrectly re-enter the 'else' branch (lines 34-36) intended for first-time additions. As a result, **value** is pushed to **set._elements** multiple times, violating the expectation that **_elements** contains unique entries. Additionally, **_length** is incremented on each such incorrect 'first' addition, leading to an inflated count. The **addedAt** timestamp is also overwritten, losing the original (zero) first addition time. Furthermore, functions like **_containsAt** and **_contains** will behave incorrectly as **addedAt > 0** will be false until a non-zero key overwrites it.

Impact: This vulnerability leads to critical state corruption:

1. **set._elements** can contain duplicate entries, significantly increasing gas costs for iterating functions like **_valuesAt** and **_values**, potentially leading to Denial of Service.
2. **set._length** will report an incorrect (inflated) number of active elements.
3. The **addedAt** timestamp for an element can be reset, losing historical accuracy.
4. Query functions like **_containsAt** and **_contains** will return incorrect results for elements initially added with **key = 0** until their **addedAt** is updated by a subsequent add with a non-zero key.

Lines: **PersistentSet.sol**, **29, 34-36, 38**

### Recommendation

To prevent this vulnerability, the **key** used in **_add** (and **_remove** for consistency, as keys often represent timestamps or ordered checkpoints) should be strictly positive.

1. Add a requirement at the beginning of the **_add** and **_remove** functions:
   **require(key > 0, "PersistentSet: Key must be greater than zero");**. This ensures that **addedAt** will always be non-zero if an element has been added.
2. If supporting **key = 0** is an absolute requirement (which is atypical for checkpoint keys), the logic for identifying a new element must be changed. For example, use a separate boolean flag like
   **mapping(bytes32 => bool) private _isEverAdded;** instead of relying solely on **addedAt > 0**.

| HIGH-16 | EdDSA Verification Does Not Explicitly Reject Identity Public Key | Pending |
|---|---|---|

### Description

The **verify** function in **SigEddsaCurve25519.sol** relies on **KeyEddsaCurve25519.fromBytes** to provide and validate the EdDSA public key. The function checks if the key is **KeyEddsaCurve25519.zeroKey()** (which corresponds to **bytes32(0)**), but this check does not prevent the use of the actual EdDSA identity point (typically (0,1) in Edwards coordinates) if it's represented by a different byte string (e.g., its standard compressed form **0x01** followed by 31 zero bytes).
RFC 8032, Section 5.1.3, mandates that an EdDSA public key A must not be the neutral element of the group. The summary for **KeyEddsaCurve25519.wrap** (called by **fromBytes**) indicates it performs an on-curve check but does not mention a specific rejection of the identity point (other than **bytes32(0)**). If **KeyEddsaCurve25519.fromBytes** can successfully decode the standard compressed representation of the identity point into a **key** struct, and **SCL_EIP6565.Verify_LE** also does not reject a public key derived from the identity point, then signatures could be verified against the identity public key. This is cryptographically insecure.
Impact: Accepting the identity point as a valid public key can fundamentally break the security of the EdDSA signature scheme. It may allow for trivial signature verifications or make it easier for an attacker to satisfy the verification equation, potentially leading to forged signatures or bypassing security controls that rely on EdDSA's unforgeability.
Lines: **SigEddsaCurve25519.sol**, **17-19, 30**

### Recommendation

Ensure that the EdDSA public key is explicitly validated against being the identity point. This check should occur either within **KeyEddsaCurve25519.fromBytes** (ideally in its **wrap** function) or as an additional check within **SigEddsaCurve25519.verify** after the key is decompressed and before **SCL_EIP6565.Verify_LE** is called. Specifically, after **(uint256 x, uint256 y) = key.decompress();**, check if these **x** and **y** coordinates represent the EdDSA identity point (0,1). If they do, the **verify** function should return **false**. The existing check against **KeyEddsaCurve25519.zeroKey()** at line 18 is insufficient as it only covers the **bytes32(0)** case, not the canonical representation of the identity point.

| HIGH-17 | Missing Subgroup and Identity Checks for Ed25519 Public Keys | Pending |
|---|---|---|

## Description

The **wrap** function validates Ed25519 public keys by decompressing the key and checking if the resulting point (x,y) satisfies the Twisted Edwards curve equation (lines 32-37). However, it omits two crucial validation steps recommended by RFC 8032 for EdDSA public keys:

1. Identity Check: It does not check if the decoded point is the identity element (0,1). The identity element is not a valid public key.
2. Subgroup Check: It does not verify that the decoded point belongs to the prime-order subgroup of order **l**. Ed25519 has a cofactor of 8, meaning the curve contains points of small order (1, 2, 4, 8) that are not in the prime-order subgroup. Using points not in the correct subgroup as public keys can lead to security vulnerabilities.

Impact: Accepting keys that are the identity element or are not in the correct prime-order subgroup can compromise the security of cryptographic protocols relying on these keys. Specifically for EdDSA:

- It may allow certain algebraic attacks or weaken the scheme against signature forgery if the point is not of the correct order.
- Small subgroup points can lead to vulnerabilities in more complex protocols built atop EdDSA, such as certain threshold signature schemes or protocols involving Diffie-Hellman-like operations with these keys.

Lines: **KeyEddsaCurve25519.sol**, **32-37**

## Recommendation

Enhance the **wrap** function to include the following checks after the point is decompressed and verified to be on the curve:

1. Identity Check: Add a check to ensure the decompressed point (x,y) is not the Ed25519 identity element (0,1). If **x == 0** and **y == 1**, the function should revert (e.g., **revert KeyEddsaCurve25519_InvalidKey();**).
2. Subgroup Check: Implement a check to ensure the point lies in the prime-order subgroup. For Ed25519, this is typically done by multiplying the point **P** by the cofactor **h=8** and verifying that **[8]P** is not the identity element **O**. If **[8]P == O** (and **P != O**), then **P** is a small-order point and invalid. This requires Ed25519 point addition and doubling functions. A more computationally intensive but direct check is **[l]P == O** where **l** is the prime order of the main subgroup. Given gas constraints, prioritize at least the identity check and carefully consider the feasibility and implementation of the subgroup check.

| HIGH-18 | keyTag Not Validated Against Required Key Tag for the Epoch | Pending |
|---|---|---|

### Description

The **verifyQuorumSig** function accepts a **keyTag** parameter (uint8) which it uses to construct a key for fetching the **validatorSetHash** from the **settlement** contract. However, the function does not validate if this provided **keyTag** matches the **requiredKeyTag** for the given **epoch** as specified by the **settlement** contract (e.g., via a function like **ISettlement.getRequiredKeyTagAt(epoch)**). An attacker could supply an arbitrary **keyTag**. If this arbitrary **keyTag** causes **ISettlement(settlement).getExtraDataAt(...)** to return a predictable value for **validatorSetHash** (e.g., **bytes32(0)** if the key formed by the arbitrary **keyTag** is not set in the settlement contract) and the attacker can craft a ZK proof valid for this predictable **validatorSetHash**, they could potentially validate a signature for an arbitrary message. This bypasses the check against the true, active **validatorSetHash** intended for the epoch.
Impact: Allows an attacker to potentially bypass signature verification by providing a **keyTag** that leads to a known or default **validatorSetHash**, against which they can generate a valid ZK proof. This undermines the integrity of the signature verification process, as it would not be tied to the correct, cryptographically secured validator set.
Lines: **SigVerifierBlsBn254ZK.sol**, **73, 96-97**

### Recommendation

The **verifyQuorumSig** function should query the **settlement** contract for the officially **requiredKeyTag** for the given **epoch** (e.g., using **ISettlement(settlement).getRequiredKeyTagAt(epoch, bytes(""))** or an equivalent method if available in the **ISettlement** interface). It must then compare this **requiredKeyTag** with the **keyTag** parameter provided to **verifyQuorumSig**. If the provided **keyTag** does not match the **requiredKeyTag** for that epoch, the verification should fail and the function should return **false**.

| HIGH-19 | Loop Bound Mismatch with Decoded Array Lengths in SigVerifierBlsBn254Simple | Pending |
|---|---|---|

### Description

In **SigVerifierBlsBn254Simple.verifyQuorumSig**, an integer **length** is read from **proof** calldata at a fixed offset (224). This **length** is then used as the upper bound for a loop (lines 72-77) that accesses elements from **validatorsData[i]** and **isNonSigners[i]**. The **validatorsData** and **isNonSigners** arrays are decoded from other parts of the **proof** calldata. The actual number of elements in these arrays upon decoding depends on their own ABI-encoded length fields within their respective calldata slices, not necessarily on the externally supplied **length** variable read from offset 224. If the **length** variable from offset 224 is greater than the actual decoded length of **validatorsData** or **isNonSigners**, the loop will attempt to access array elements out of bounds, leading to a revert.
Impact: An attacker can craft a **proof** where the **length** field at offset 224 is inconsistent with (larger than) the actual encoded lengths of the **validatorsData** or **isNonSigners** arrays within the proof. This will cause out-of-bounds access during the loop and make the transaction revert. This can lead to a denial of service for the **verifyQuorumSig** function, preventing valid signatures from being verified.
Lines: **SigVerifierBlsBn254Simple.sol**, **58, 69-70, 72-77**

### Recommendation

After decoding **validatorsData** and **isNonSigners**, their **.length** properties should be used as the authoritative source for loop bounds and consistency checks, rather than relying solely on the **length** variable read from offset 224 for iteration.

1. Use **validatorsData.length** for the main loop bound.
2. Before the loop, add checks: **require(validatorsData.length == length, "Mismatched validatorsData length");** (if **length** from offset 224 is still considered a primary count) and critically, **require(isNonSigners.length == validatorsData.length, "Inconsistent array lengths");** to ensure all arrays to be iterated have the expected and consistent number of elements.

| HIGH-20 | Unclear Message Hashing Responsibility and Method for EdDSA Verification | Pending |
|---|---|---|

### Description

The **verify** function accepts **bytes memory message** and passes it as **string(message)** to **SCL_EIP6565.Verify_LE**. The EdDSA algorithm (specifically Ed25519) requires a specific cryptographic hash of the message, typically SHA-512. It is unclear from the provided code and context whether **SCL_EIP6565.Verify_LE** is responsible for hashing this input message. If it is, it's also unclear whether it uses the correct hashing algorithm (e.g., SHA-512 for standard Ed25519) and correctly handles the conversion from **string** to the precise byte sequence that needs to be hashed (EdDSA operates on byte sequences, not UTF-8 strings directly). If the calling contract (or user) is expected to pre-hash the message and pass the digest, the parameter **bytes memory message** is misleading and should instead be **bytes32 messageDigest**. If **SCL_EIP6565.Verify_LE** does not perform the hashing, or performs it incorrectly (e.g., wrong algorithm, or hashing the string representation rather than the intended byte sequence), the EdDSA verification process will be fundamentally flawed.
Impact: If the message is not hashed, or hashed using an incorrect algorithm or over an incorrect byte representation (due to the **string** cast or internal library logic), valid EdDSA signatures will likely fail verification. More critically, if a flawed hashing process (e.g., no hashing, a weak hash, or hashing a misrepresentation of the message) is employed, it could allow attackers to forge signatures by finding another message that results in the same (incorrectly processed) input to the core EdDSA equation, thereby breaking the unforgeability and integrity properties of the signature scheme.
Lines: **SigEddsaCurve25519.sol**, **13, 30**

### Recommendation

1. Clearly define and document which component is responsible for hashing the message in the EdDSA verification process: the caller of **SigEddsaCurve25519.verify**, the **SigEddsaCurve25519.verify** function itself, or the underlying **SCL_EIP6565.Verify_LE** library.
2. If **SigEddsaCurve25519.verify** or **SCL_EIP6565.Verify_LE** is responsible for hashing, ensure the correct cryptographic hash function (e.g., SHA-512 for standard Ed25519) is used. The input to this hash should be the raw **bytes memory message**, not **string(message)**, unless the **SCL_EIP6565** library specifically requires a string and handles its conversion to bytes for hashing in a standard-compliant way for EdDSA.
3. If the caller is expected to pre-hash the message, the **verify** function signature in **SigEddsaCurve25519.sol** should be changed to accept **bytes32 messageDigest** instead of **bytes memory message** to make this requirement explicit and prevent misuse.

| HIGH-21 | State Inconsistency from Unauthenticated Key Setters | Pending |
|---|---|---|

### Description

The public functions **setKey(address operator, uint8 keyTag, bytes memory key)**, **setKey32(...)**, and **setKey64(...)** in **KeyManagerLogic.sol**, in addition to bypassing authentication, also fail to update several crucial state mappings and sets that are part of the comprehensive key management logic. According to the summary, these functions only update the raw key payload storage (**_keys32** or **_keys64** checkpoints). They do not update associated metadata like **_operatorByKeyHash**, **_operatorByTypeAndKeyHash**, **_operatorByTagAndKeyHash**, the **_operators** persistent set, or the **_operatorKeyTags** checkpoint. This omission means that while a key might be retrievable via **getKey(operator, tag)**, other important views of the key system, such as **getOperator(key)** (which likely uses **_operatorByKeyHash**) or **getKeyTags(operator)**, would not reflect this registration accurately.
Impact: The key management system's state becomes inconsistent and unreliable. Keys set via these unauthenticated functions will be partially registered, leading to conflicting information depending on which getter function is used. This can cause failures in critical system operations, allow for potential bypasses of key reuse policies (which rely on the unupdated mappings), and make auditing or monitoring operator keys extremely difficult. For example, an attacker-registered key might be usable but untraceable to an operator through standard lookup functions.
Lines: **KeyManagerLogic.sol**, **228-239, 241-244, 246-251**

### Recommendation

As with KML-AUTHBYPASS-001, change the visibility of these functions (**setKey(operator, keyTag, key)**, **setKey32**, **setKey64**) to **internal**. They should only be invoked by the primary, authenticated **setKey** function variants, which are responsible for correctly updating all relevant state variables and mappings to maintain consistency and enforce key management policies.

| HIGH-22 | Incorrect Logic in **getCurrentValue** Leads to Wrong Epoch Data Retrieval and Potential DoS | Pending |
|---|---|---|

### Description

The **getCurrentValue** function (lines 146-163) is intended to retrieve the most recent checkpoint value for a given **currentTimepoint** from a **Checkpoints.Trace208** storage. However, its logic is flawed. Specifically, when the latest checkpoint in the trace (**trace.at(length - 1)**) has a key (**_key**) greater than **currentTimepoint**, the function attempts to return the value of the second-to-latest checkpoint (**trace.at(length - 2)**) without verifying if this second-to-latest checkpoint's key is actually less than or equal to **currentTimepoint**.
This can lead to two problematic scenarios:

1. If the second-to-latest checkpoint's key is also greater than **currentTimepoint**, **getCurrentValue** will return data from a future configuration. This can cause **Time.timestamp() - epochDurationTimestamp** (in **getCurrentEpoch**) to underflow (as **epochDurationTimestamp** would be > **Time.timestamp()**), leading to a revert or calculation of a wildly incorrect, large epoch number, likely causing further reverts due to **uint48** overflow.
2. If the correct checkpoint is older than the second-to-latest (e.g., **c1._key <= currentTimepoint < c2._key < c3._key**), **getCurrentValue** would incorrectly return **c2._value** instead of **c1._value**.

The **Checkpoints.Trace208** type, typically based on OpenZeppelin's Checkpoints library, provides an **upperLookupRecent(key, hint)** method which correctly implements this lookup (usually via binary search).
This flawed **getCurrentValue** function is used by **getCurrentEpochDurationData** (line 101), which in turn is used by **getCurrentEpoch**, **getCurrentEpochDuration**, and **getCurrentEpochStart**. Incorrect data from **getCurrentValue** will therefore propagate to these core epoch information functions.
Impact: The flawed logic in **getCurrentValue** can cause critical functions like **getCurrentEpoch()**, **getCurrentEpochStart()**, and **getCurrentEpochDuration()** to return incorrect values or revert due to arithmetic errors (underflow/overflow). This can disrupt any system functionality reliant on accurate current epoch information, potentially leading to denial of service for time-sensitive operations or incorrect state transitions.
Lines: **EpochManagerLogic.sol**, **146-163, 99-102**

### Recommendation

Replace the custom logic in **getCurrentValue** with a direct call to the **upperLookupRecent** method available on the **Checkpoints.Trace208** type. This method is designed to correctly find the checkpoint value associated with the largest key less than or equal to the provided **currentTimepoint**.
Modified **getCurrentValue** function:

```
function getCurrentValue(
   Checkpoints.Trace208 storage trace,
   uint48 currentTimepoint
) public view returns (uint208) {
   if (trace.length() == 0) {
      revert IEpochManager.EpochManager_NoCheckpoint();
   }
   // Use the standard upperLookupRecent from the Checkpoints library.
   // This function is expected to revert if no checkpoint is found (e.g., all keys > currentTimepoint,
   // or currentTimepoint is before the first key), which is appropriate.
   return trace.upperLookupRecent(currentTimepoint, bytes(""));
}
```

Alternatively, remove **getCurrentValue** entirely and call **_getEpochManagerStorage()._epochDurationDataByTimestamp.upperLookupRecent(Time.timestamp(), bytes(""))** directly within **getCurrentEpochDurationData**. If the specific revert **IEpochManager.EpochManager_NoCheckpoint()** is required for cases where **upperLookupRecent** might revert with a different error (e.g., **CheckpointsOutOfBounds**), a try-catch block can be used in **getCurrentEpochDurationData** to translate the error.

| HIGH-23 | Unvalidated Input Array Lengths in Constructor Leads to Potential Out-of-Bounds Read | Pending |
|---|---|---|

### Description

The constructor of **SigVerifierBlsBn254ZK** fails to validate that the input arrays **verifiers_** and **maxValidators_** have the same length. It attempts a check using **verifiers.length != maxValidators.length** (state variables) before these state variables are initialized from the inputs. At this point, **verifiers.length** and **maxValidators.length** are both 0, so the check **0 != 0** is false, and the intended validation for equal input lengths is skipped. Consequently, the contract can be deployed with **verifiers** and **maxValidators** state arrays of differing lengths. If the **maxValidators** array is initialized to be longer than the **verifiers** array, the **_getVerifier** function can iterate with an index **i** that is valid for **maxValidators[i]** but out of bounds for **verifiers[i]**. Accessing **verifiers[i]** in this scenario will lead to an out-of-bounds read, causing the transaction to revert.

Impact: If **maxValidators.length > verifiers.length** due to the skipped validation, any call to **verifyQuorumSig** that results in **_getVerifier** attempting to access an out-of-bounds index in the **verifiers** array will cause a revert. This can render the **verifyQuorumSig** function unusable for certain ranges of **totalActiveValidators**, leading to a denial of service for signature verification.

Lines: **SigVerifierBlsBn254ZK.sol**, **48-50, 61-62, 118-120**

### Recommendation

Modify the constructor to validate the lengths of the input arrays **verifiers_** and **maxValidators_** directly, before assigning them to state variables. The subsequent validation loop should also use the length from these input arrays. Example:

```
constructor(address[] memory verifiers_, uint256[] memory maxValidators_) {
  uint256 verifiersLength = verifiers_.length;
  if (verifiersLength != maxValidators_.length) {
    revert SigVerifierBlsBn254ZK_InvalidLength();
  }
  for (uint256 i; i < verifiersLength; ++i) {
    if (maxValidators_[i] == 0) {
      revert SigVerifierBlsBn254ZK_InvalidMaxValidators();
    }
    if (i > 0 && maxValidators_[i - 1] >= maxValidators_[i]) {
      revert SigVerifierBlsBn254ZK_InvalidMaxValidators();
    }
  }

  verifiers = verifiers_;
  maxValidators = maxValidators_;
}
```

| MEDIUM-01 | Missing Events for Critical State Changes | Pending |
|---|---|---|

### Description

The contract modifies critical state variables such as the whitelist status (**_isWhitelistEnabled**), operator whitelist ( **_whitelisted**), and operator vault whitelist (**_whitelistedVault**) in several functions (**setWhitelistStatus**, **whitelistOperator**, **unwhitelistOperator**, **whitelistOperatorVault**, **unwhitelistOperatorVault**) but does not emit events upon these changes. Events are essential for off-chain services to monitor contract activity, for transparency, and for user interfaces to react to state updates.
Impact: Lack of events makes it difficult to track whitelist changes off-chain, hindering monitoring, auditing, and integration with user interfaces or other services that rely on these states.
Lines: **WhitelistSelfRegisterOperators.sol**, **61-63, 68-75, 80-90, 95-100, 105-113**

### Recommendation

Emit events for all functions that modify the whitelist status or whitelist entries. For example:

- **event WhitelistStatusSet(bool newStatus);** in **setWhitelistStatus**.
- **event OperatorWhitelisted(address indexed operator);** in **whitelistOperator**.
- **event OperatorUnwhitelisted(address indexed operator);** in **unwhitelistOperator**.
- **event OperatorVaultWhitelisted(address indexed operator, address indexed vault);** in **whitelistOperatorVault**.
- **event OperatorVaultUnwhitelisted(address indexed operator, address indexed vault);** in **unwhitelistOperatorVault**.

| MEDIUM-02 | Failure in Unregistration Logic Prevents Force-Pausing | Pending |
|---|---|---|

### Description

In **forcePauseOperator** (lines 54-56) and **forcePauseOperatorVault** (lines 79-81), if an operator or vault is registered, the contract attempts to unregister them by calling **_unregisterOperator** or **_unregisterOperatorVault**. These calls are part of the same transaction that sets the force-pause flag. If these internal unregistration functions revert for any reason (e.g., an internal assertion fails in the parent contract's logic, unexpected state, or an out-of-gas error not covered by gas griefing), the entire **forcePauseOperator** or **forcePauseOperatorVault** transaction will revert. Consequently, the operator/vault will not be marked as force-paused.
Impact: The system may be unable to force-pause an operator or vault if the dependent unregistration logic fails. This compromises the reliability of the force-pause mechanism, potentially allowing a problematic operator to continue operating when administrators intend to stop them.
Lines: **ForcePauseSelfRegisterOperators.sol**, **55, 80**

### Recommendation

The atomicity of pausing and unregistering is a design choice. If this atomicity is strictly required for system consistency, then the **_unregisterOperator** and **_unregisterOperatorVault** functions (and their underlying implementations) must be made exceptionally robust to avoid reverting under conditions where a force-pause is necessary. Thoroughly review and test all potential revert paths in the unregistration logic. Alternatively, if ensuring the pause flag is set is paramount even if unregistration fails, consider decoupling the unregistration step, for example, by using a try/catch block for the unregistration call. This would allow the force-pause flag to be set, and an event could be emitted if unregistration fails, signaling the need for manual intervention or a separate cleanup process. However, this could lead to temporary inconsistencies where an entity is paused but still appears registered in some underlying modules.

| MEDIUM-03 | Misleading **KeyTags_Duplicate** Revert for Invalid Key Tags in **serialize** Function | Pending |
|---|---|---|

## Description

The **serialize** function at lines 57-65 iterates through an array of **keyTags**. For each **keyTag**, it calls **keyTagsSerialized.add(keyTag)** (line 60). The **add** function (lines 46-48) itself does not validate if the input **keyTag** is within the valid range (i.e., **keyTag < TOTAL_KEY_TAGS**). If an invalid **keyTag** (e.g., 128 or greater, given **TOTAL_KEY_TAGS = 128**) is provided to **add**, the expression **uint128(1 << keyTag)** on line 47 evaluates to 0. This is because **1** is promoted to **uint256**, shifted by **keyTag**, and if **keyTag >= 128**, the resulting **uint256** value when cast to **uint128** becomes 0. Consequently, **keyTagsSerialized.add(invalidKeyTag)** will return the original **keyTagsSerialized** value, meaning the addition of an invalid tag has no effect. Back in the **serialize** function, the check **if (oldKeyTagsSerialized == keyTagsSerialized)** (line 61) will then evaluate to true. This causes the function to revert with **KeyTags_Duplicate()**. This error message is misleading because the actual issue is that an invalid key tag was provided, not that a valid key tag was duplicated.
Impact: Callers of the **serialize** function might misinterpret the **KeyTags_Duplicate** error, leading them to believe they have provided a duplicate valid tag when the root cause is an invalid (out-of-range) tag. This can complicate debugging, lead to incorrect assumptions about the state of input data, and potentially cause unexpected behavior in systems relying on specific error types for control flow.
Lines: **KeyTags.sol**, **47, 60, 61**

## Recommendation

Add an explicit check for **keyTag** validity within the **add** function. If **keyTag >= TOTAL_KEY_TAGS**, the **add** function should revert with **KeyTags_InvalidKeyTag()**. This ensures that invalid key tags are rejected with the correct error message before any other logic is performed, and **serialize** will then propagate the correct error.
Modify the **add** function as follows:

```
function add(uint128 keyTagsSerialized, uint8 keyTag) internal pure returns (uint128) {
  if (keyTag >= TOTAL_KEY_TAGS) {
    revert KeyTags_InvalidKeyTag();
  }
  return keyTagsSerialized | uint128(1 << keyTag);
}
```

| MEDIUM-04 | Potential Gas Griefing/DoS via Unbounded Iteration in Value Retrieval Functions | Pending |
|---|---|---|

### Description

The private functions **_valuesAt** and **_values** (and consequently their internal wrappers in **Bytes32Set** and **AddressSet**) iterate over **set._elements**. The **_elements** array stores every unique element ever added to the set. If a very large number of unique elements are added over the contract's lifetime (even if many are subsequently removed or become inactive), the **_elements** array can grow significantly. Iterating this potentially large array and performing checks (**_containsAt** or **_contains**) for each element can lead to substantial gas consumption. Notably, **_containsAt** involves a call to **Checkpoints.Trace208.upperLookupRecent**, which can itself be gas-intensive depending on the checkpoint history length and the quality of the provided hint.
Impact: If a context allows for a large number of unique elements to be added to the set (e.g., by users or an automated process), calls to **valuesAt** or **values** (through the **Bytes32Set** or **AddressSet** wrappers) could consume an excessive amount of gas. This might lead to transactions hitting the block gas limit, rendering these functions unusable and causing a denial of service for any functionality that relies on retrieving the complete list of active set members at a specific time or currently.
Lines: **PersistentSet.sol, 69-88, 90-106**

### Recommendation

Evaluate the expected maximum number of unique elements that will be added to a set.

1. If this number can be very large, consider alternative designs for retrieving all elements, such as implementing pagination, or restricting the conditions under which these functions are called.
2. Document the gas characteristics and potential for high gas costs clearly for developers using this library.
3. If the number of unique elements is inherently bounded to a small or moderate size by the application's logic, this risk may be acceptable.

| MEDIUM-05 | Ambiguous **message** Parameter Assumed to be a 32-Byte Hash | Pending |
|---|---|---|

### Description

The **verify** function accepts a **bytes memory message** parameter. This parameter is decoded into a **bytes32** using **abi.decode(message, (bytes32))** and then passed to **ECDSA.recover**. This implementation implicitly requires **message** to be exactly 32 bytes long (otherwise **abi.decode** reverts) and to represent the cryptographic hash that was signed. The parameter name **message** is misleading, as **messageHash** or **digest** would be more accurate. If a caller provides 32 bytes of raw data (that is not a hash) as **message**, signature verification will be performed against this raw data. This deviates from the standard practice where the message data is typically hashed before signing and recovery, and OpenZeppelin's **ECDSA.recover** explicitly expects a **bytes32 hash**.
Impact: If callers misunderstand this implicit requirement and provide 32 bytes of raw data instead of a pre-computed hash, the signature verification will operate on unintended data. This could lead to vulnerabilities if the broader system relies on messages being hashed before signing (e.g., to prevent replay attacks or issues with structured data). While the primary known caller (**KeyManagerLogic**) uses this correctly by passing an ABI-encoded **bytes32** digest, future or other callers might misuse it due to the lack of clarity. The current approach also adds a slight processing overhead of **abi.decode** and restricts the **message** to be exactly 32 bytes, which might not be flexible for all use cases if arbitrary messages were intended (though **ECDSA.recover** itself requires a hash).
Lines: **SigEcdsaSecp256k1.sol, 14, 23**

### Recommendation

To improve clarity, security posture against misuse, and efficiency:

1. Rename the **message** parameter to **messageHash** (or a similar name like **digest**) to clearly indicate its expected nature.
2. Change its type from **bytes memory** to **bytes32**. This makes the 32-byte hash requirement explicit at the type level and eliminates the need for **abi.decode(message, (bytes32))**. The call to **ECDSA.recover** would then use **messageHash** directly. Callers like **KeyManagerLogic** would then pass their **bytes32 digest** directly instead of **abi.encode(digest)**, simplifying the calling code as well.

| MEDIUM-06 | Incomplete Validation in **deserialize** Allows Non-Canonical Keys | Pending |
|---|---|---|

## Description

The **deserialize** function (lines 55-68) constructs a **KEY_BLS_BN254** struct from a compressed byte representation. While it reconstructs the X and Y coordinates, it does not validate that the **X** coordinate is within the field, i.e., **X < BN254.FP_MODULUS**. The **X** coordinate is derived from **uint256(compressedKey) >> 1**, which can result in a value greater than or equal to **BN254.FP_MODULUS**. The **wrap** function, which is the canonical way to create validated **KEY_BLS_BN254** structs, explicitly checks for **keyRaw.X >= BN254.FP_MODULUS** (line 27) and would revert. Therefore, **deserialize** can produce a **KEY_BLS_BN254** instance that is not fully validated according to the library's own **wrap** function. While precompiled contract calls for elliptic curve operations typically take coordinates modulo the field prime, having struct fields with non-canonical values (**X >= FP_MODULUS**) can lead to inconsistencies, incorrect behavior in other parts of the system that might not expect this (e.g., equality checks, further serializations if underlying EC math handles large X differently), and generally violates the invariants expected of a key type.
Impact: Creation of non-canonical **KEY_BLS_BN254** structs where **value.X >= BN254.FP_MODULUS**. This can lead to inconsistent behavior, failed equality checks against canonical keys representing the same field element, and potential issues if other components consume these structs assuming canonical field elements. It may also cause denial of service if such a key is later processed by a stricter function (like **wrap** itself or **fromBytes**).
Lines: **KeyBlsBn254.sol**, **55-68**

## Recommendation

Modify the **deserialize** function to ensure that the deserialized **X** coordinate is less than **BN254.FP_MODULUS** and that the resulting point is fully valid. The most robust approach is to reconstruct the **BN254.G1Point** from the compressed data and then pass it to the existing **wrap** function. This ensures all validations defined in **wrap** (including coordinate range and on-curve checks) are applied. For example:

```
function deserialize(
  bytes memory keySerialized
) internal view returns (KEY_BLS_BN254 memory key) {
  bytes32 compressedKey = abi.decode(keySerialized, (bytes32));
  if (compressedKey == bytes32(0)) {
    return zeroKey(); // Uses wrap implicitly or should be made to
  }
  uint256 X = uint256(compressedKey) >> 1;
  // It's critical that BN254.findYFromX handles X correctly (e.g., reverts if X >= FP_MODULUS, or if X is not suitable)
  // Assuming findYFromX might not check X range itself, wrap will.
  (, uint256 derivedY) = BN254.findYFromX(X);

  BN254.G1Point memory pointToWrap;
  pointToWrap.X = X;
  if ((uint256(compressedKey) & 1) == 0) { // Y is derivedY
    pointToWrap.Y = derivedY;
  } else { // Y is P - derivedY
    // Assuming BN254.negate primarily changes Y to P-Y for a G1Point(X, derivedY)
    // A direct calculation might be clearer if negate is complex:
    pointToWrap.Y = BN254.FP_MODULUS - derivedY;
  }
  key = wrap(pointToWrap); // This will perform all necessary checks including X range.
}
```

Alternatively, add specific checks within **deserialize** after reconstructing the point:
**if (X_reconstructed >= BN254.FP_MODULUS) { revert KeyBlsBn254_InvalidKey(); }** and ensure all other **wrap** conditions are met.

| MEDIUM-07 | Identity Point Acceptance for G2 Key Component (keyG2) | Pending |
|---|---|---|

### Description

The **verify** function decodes **keyG2** from **extraData** via **abi.decode(extraData, (BN254.G2Point))**. There is no check to ensure that the decoded **keyG2** is not the G2 point at infinity (identity element). If **keyG2** is the point at infinity, the pairing equation **e(A, B) * e(C, D) == 1** simplifies because **e(C, O_G2)** (where **O_G2** is the G2 identity element) evaluates to **1_GT**. The verification would then only depend on **e(A, B) == 1_GT**, which is **e(signatureG1.plus(keyG1.scalar_mul(alpha)), BN254.negGeneratorG2()) == 1_GT**. The challenge **alpha** is computed using the coordinates of **keyG2**, which would be zero if it's the identity point. Allowing **keyG2** to be the identity element changes the cryptographic properties of the verification and might lead to vulnerabilities if an attacker can control **extraData** to supply an identity **keyG2**. Standard cryptographic protocols usually require public key components to be non-identity elements of the correct prime-order subgroup.
Impact: If an attacker can control the **extraData** parameter to make **keyG2** the G2 point at infinity, the verification logic changes significantly. This alters the security properties of the signature scheme, potentially making it easier to satisfy the verification equation or bypass checks intended by the protocol designers. While not leading to an immediate trivial forgery, it represents a deviation from secure cryptographic practice and might weaken the overall system guarantees.
Lines: **SigBlsBn254.sol**, 27

### Recommendation

Explicitly check if **keyG2** is the G2 point at infinity after it is decoded. If it is the identity element, the function should return **false**. A G2 point is at infinity if all its coordinates are zero. For a **BN254.G2Point** struct **{(uint256[2] X, uint256[2] Y)}**, this means checking **keyG2.X[0] == 0 && keyG2.X[1] == 0 && keyG2.Y[0] == 0 && keyG2.Y[1] == 0**. Consider using a helper function from the **BN254** library if available for checking identity.

| MEDIUM-08 | Gas Griefing Vulnerability due to Unbounded **isNonSigners** Array Decoding | Pending |
|---|---|---|

### Description

The **verifyQuorumSig** function decodes a **bool[] memory isNonSigners** from a tail slice of the **proof** calldata ( **proof[256 + length * 96:]**) on line 70. The **abi.decode** operation will attempt to decode the entire **bool[]** as specified by its ABI-encoded length within that slice. An attacker can craft **proof** such that **length** (for **ValidatorData[]**) is small and passes initial checks (including any cap if implemented from the previous recommendation), but the slice **proof[256 + length * 96:]** encodes an extremely large **bool[]** array (e.g., with a large ABI-encoded length field). The **abi.decode** call on line 70 will then consume a very large amount of gas to decode this array, even though the subsequent loop (lines 72-77) only uses **length** elements from **isNonSigners**. This constitutes a gas griefing vulnerability.
Impact: An attacker can cause the **verifyQuorumSig** function to consume excessive gas during the decoding of the **isNonSigners** array, potentially leading to denial of service for legitimate signature verifications by making them hit gas limits or become too expensive. This occurs even if the primary **length** (for **ValidatorData**) is reasonably small.
Lines: **SigVerifierBlsBn254Simple.sol**, **70**

### Recommendation

To mitigate this, restrict the size of the byte slice passed to **abi.decode** for **isNonSigners** based on the already determined (and ideally capped) **length** of **ValidatorData[]**. The ABI-encoded size of **bool[length]** can be calculated, and only a slice of this expected size should be decoded. Example steps:

1. After **length** (for **ValidatorData[]**) is known and validated: **uint256 startOfIsNonSignersSliceOffset = 256 + length * 96;**
2. Calculate the expected ABI-encoded size for **bool[length]**. For **bool[] arr**, this is 64 bytes (for offset and length fields in the ABI encoding) plus **((length + 31) / 32) * 32** bytes for the tightly packed boolean data.
   **uint256 expectedDataBytes = ((length + 31) / 32) * 32;**
   **uint256 expectedIsNonSignersEncodedSize = 64 + expectedDataBytes;**
3. Ensure **proof** is long enough for this expected slice:
   **if (startOfIsNonSignersSliceOffset + expectedIsNonSignersEncodedSize > proof.length) {**
   **revert SigVerifierBlsBn254Simple_ProofFormatError(); // Or specific error }**
4. Manually create a **bytes memory** slice of **expectedIsNonSignersEncodedSize** from **proof** starting at **startOfIsNonSignersSliceOffset**.
   **bytes memory sliceForIsNonSigners = new bytes(expectedIsNonSignersEncodedSize);**
   **for (uint j = 0; j < expectedIsNonSignersEncodedSize; ++j) {**
   **sliceForIsNonSigners[j] = proof[startOfIsNonSignersSliceOffset + j]; }**
5. Decode this controlled-size slice: **bool[] memory isNonSigners = abi.decode(sliceForIsNonSigners, (bool[]));**
6. Finally, verify consistency: **if (isNonSigners.length != length) {**
   **revert SigVerifierBlsBn254Simple_ProofFormatError(); // Or specific error }** This prevents **abi.decode** from processing an arbitrarily large segment of **proof** for **isNonSigners**.

| MEDIUM-09 | Missing Input Validation for **proof** Calldata Length | Pending |
|---|---|---|

### Description

The **verifyQuorumSig** function uses inline assembly to read multiple components (**zkProof**, **commitments**, **commitmentPok**, **signersVotingPower**) from the **proof** calldata argument at fixed offsets. It assumes **proof** has a specific minimum length (416 bytes) to accommodate these components. However, there is no check to ensure **proof.length** meets this requirement.
Impact: If a **proof** shorter than 416 bytes is supplied, **calldataload** operations might read beyond the bounds of the **proof** byte array. This could lead to zero-values or arbitrary data from other parts of calldata being used for **signersVotingPower** or other proof components. This could potentially allow bypassing the quorum threshold check or cause the external ZK verifier to behave unpredictably, possibly accepting invalid proofs or rejecting valid ones.
Lines: **SigVerifierBlsBn254ZK.sol**, **77-92**

### Recommendation

Add a **require** statement at the beginning of **verifyQuorumSig** to ensure **proof.length** is exactly 416 bytes. For example: **require(proof.length == 416, "SigVerifierBlsBn254ZK: Invalid proof length");**.

| MEDIUM-10 | Potentially Unbounded Loops in Getter Functions May Lead to Denial of Service | Pending |
|---|---|---|

### Description

Functions such as **getVotingPowerProviders**, **getVotingPowerProvidersAt**, **getReplicas**, **getReplicasAt**, and consequently **getConfig** and **getConfigAt** (which call these), retrieve all relevant items from **PersistentSet** storage and then iterate over the results to deserialize them. The underlying **PersistentSet.values()** and **PersistentSet.valuesAt()** likely also iterate to collect these items.
Impact: If the number of registered voting power providers or replicas grows significantly, the gas cost to execute these getter functions could exceed the block gas limit. This would make it impossible for users or other contracts to retrieve this configuration data through these functions, leading to a denial of service for reading this state.
Lines: **ConfigProviderLogic.sol**, **72-78, 80-90, 123-128, 130-136, 181-197, 199-210**

### Recommendation

Implement pagination or indexed access for these getter functions to allow retrieval of data in manageable chunks. For example, allow specifying an offset and limit. Alternatively, if the system design ensures these lists remain small and bounded, this risk might be acceptable. The **PersistentSet** library might also need to support paginated retrieval if it's the primary source of unbounded iteration.

| MEDIUM-11 | setCommitDuration Does Not Validate Against Epoch Duration Invariant | Pending |
|---|---|---|

## Description

The **setCommitDuration** function allows setting a new **commitDuration** effective from the next epoch. However, it lacks a check to ensure that this new **commitDuration** remains less than the **epochDuration** that will be active for that same future epoch. The system relies on the invariant **epochDuration > commitDuration** for correct phase transitions and operation. While **initialize** and **setEpochDuration** enforce this, **setCommitDuration** can be called independently, potentially violating this invariant if a large **commitDuration** is set against a smaller, previously set **epochDuration** for the next epoch.
Impact: If **commitDuration** becomes greater than or equal to **epochDuration** for any given epoch, the logic for determining validator set phases (**getCurrentPhase**) and commitment windows can break. This could prevent new validator set headers from being committed, stall the progression of epochs, or lead to other undefined behavior as time-based conditions for **COMMIT** and **PROLONG** phases might become impossible to satisfy correctly.
Lines: **SettlementLogic.sol**, **268-272**

## Recommendation

In the **setCommitDuration** function, add a validation step to ensure the new **commitDuration** is strictly less than the **epochDuration** that will be effective when the new **commitDuration** becomes active (i.e., for the next epoch). This requires fetching the planned **epochDuration** for the next epoch from **EpochManagerLogic**'s storage (e.g., using **EpochManagerLogic.getCurrentValue** with **EpochManagerLogic.getNextEpochStart()** as the timestamp for the lookup in **_epochDurationDataByTimestamp**). The check should be:
**require(futureEpochDuration > commitDuration, "Settlement: Epoch duration too short for commit duration");**.

| MEDIUM-12 | registerOperator Susceptible to Malicious OPERATOR_REGISTRY if Misused by Caller | Pending |
|---|---|---|

## Description

The **registerOperator** function in **OperatorManagerLogic.sol** takes **OPERATOR_REGISTRY** as an address parameter. It then calls **IRegistry(OPERATOR_REGISTRY).isEntity(operator)** to verify the operator. If a calling contract, other than the intended **OperatorManager.sol**, were to use this library function and provide a malicious or compromised **OPERATOR_REGISTRY** address, it could bypass the operator validation check.
Impact: If an attacker can control the **OPERATOR_REGISTRY** instance passed to this function (e.g., through a misconfigured or vulnerable calling contract), they could register arbitrary addresses as operators or prevent legitimate operators from registering. This would compromise the integrity of the operator set managed by this logic.
Lines: **OperatorManagerLogic.sol**, **55-58**

## Recommendation

This library function relies on its calling contract to provide a trusted and secure **OPERATOR_REGISTRY** address. The primary **OperatorManager.sol** contract correctly mitigates this by using an **immutable OPERATOR_REGISTRY** address set at its construction. This finding highlights the importance of such a practice for any contract utilizing **OperatorManagerLogic.registerOperator**. Ensure that any integration of this library function rigorously controls and validates the **OPERATOR_REGISTRY** parameter it passes.

| MEDIUM-13 | Inefficient Memory Allocation and Iteration in _valuesAt and _values | Pending |
|---|---|---|

### Description

The private functions **_valuesAt** and **_values** allocate a new memory array **values_** with a size equal to **set._elements.length**. This **_elements** array stores every unique element ever added to the set and does not shrink upon removal. The functions then iterate **set._elements.length** times to find currently active elements. If the set has experienced high churn (many elements added and subsequently removed), **set._elements.length** can be significantly larger than the number of currently active elements (**set._length**). This leads to allocating a potentially much larger intermediate array than needed and iterating over many inactive elements.

Impact: Increased gas costs for **_valuesAt** and **_values** operations, especially in sets with high churn. While the final returned array is resized to **actualLength** using an assembly **mstore**, the initial memory allocation and the loop iterations are based on the total number of unique elements ever added, not just the currently active ones. This can make these read operations more expensive than necessary and could contribute to hitting gas limits in transactions that call these functions, even if the number of active elements is small.

Lines: **PersistentSet.sol**, **75-78, 93-96**

### Recommendation

To optimize gas usage, consider strategies to reduce the iteration space or initial allocation size if **set._elements.length** is much larger than **set._length**:

1. If feasible, for current value queries (**_values**), it might be more efficient to iterate based on a structure that only tracks active elements, though this would be a significant redesign.
2. For the current design, document this gas characteristic clearly, so users are aware that queries can become more expensive if the total number of unique historical elements is large, irrespective of the number of currently active elements.
3. Evaluate if the cost of allocating memory for **totalLength** items when **actualLength** is much smaller is a significant concern. If so, a two-pass approach (first count active elements, then allocate and populate) could save on memory allocation gas but would require iterating twice or more complex logic.

| MEDIUM-14 | Non-compliance with RFC 8032 Ed25519 Decoding: **y >= p** Check Missing | Pending |
|---|---|---|

### Description

The **decompress** function extracts the y-coordinate from the compressed key bytes (line 86). According to RFC 8032 (Section 5.1.3, "Decoding", Step 2), if the recovered y-coordinate **Y** is greater than or equal to the field prime **p**, the decoding process should fail. The current implementation in **decompress** does not perform this check. While subsequent calculations involving **y** within the library (e.g., **mulmod(y, y, p)** in **decompress** or the curve equation check in **wrap**) will effectively use **y mod p**, the standard explicitly requires rejecting keys that encode a **y >= p**.

Impact: The library may accept Ed25519 key encodings that are considered invalid by the RFC 8032 standard. This can lead to interoperability issues with other Ed25519 implementations that strictly follow the standard. While it might not immediately lead to a direct security vulnerability if all math is consistently done modulo **p** for point validation, it represents a deviation from a cryptographic standard. If any part of a system consuming these keys (or their decompressed coordinates) relies on the **y** coordinate being strictly less than **p** as per the standard, it could lead to unexpected behavior or errors.

Lines: **KeyEddsaCurve25519.sol**, **86, 32-37**

### Recommendation

Modify the **decompress** function to comply with RFC 8032. After the y-coordinate is extracted at line 86 (i.e., **y = kPubC & 0x7f...f;**), add an explicit check: **if (y >= p) { revert KeyEddsaCurve25519_InvalidKey(); }**. This will ensure that any key encoding a y-coordinate not strictly less than **p** is rejected, aligning the library's behavior with the Ed25519 standard.

| MEDIUM-15 | Unvalidated G2 Public Key Component from Proof Passed to Verification | Pending |
|---|---|---|

### Description

The **aggPublicKeyG2** component is taken directly from the **proof** calldata (lines 93-94) and passed as the **extraData** argument (interpreted as **keyG2**) to **SigBlsBn254.verify**. There is no validation performed on **aggPublicKeyG2** within **SigVerifierBlsBn254Simple** (e.g., to check if it's the G2 point at infinity, is on the curve, or if it's in the correct G2 subgroup). The **SigBlsBn254.verify** function itself, as indicated by finding **SIGBLS_ALLOWS_IDENTITY_KEYG2** in its analysis context, does not validate this **keyG2** input for being the identity point.
Impact: If **aggPublicKeyG2** is the G2 point at infinity or otherwise invalid (e.g., not in the correct subgroup), it can compromise the security of the BLS signature verification scheme used in **SigBlsBn254.verify**. This might simplify the pairing equation, potentially weakening the signature verification scheme or making it easier for an attacker to construct a proof that passes verification. This could undermine the integrity of the quorum signature verification.
Lines: **SigVerifierBlsBn254Simple.sol**, **94, 95**

### Recommendation

Validate **aggPublicKeyG2** before passing it to **SigBlsBn254.verify**. At a minimum, decode **aggPublicKeyG2** to a **BN254.G2Point** and check that it is not the G2 point at infinity (all coordinates zero). Ideally, also verify that it is a valid point on the BN254 G2 curve and belongs to the correct prime-order subgroup. This might involve adding helper functions to the **BN254** library or ensuring **SigBlsBn254.verify** performs these checks. Example check for identity:

```
BN254.G2Point memory decodedAggPublicKeyG2 = abi.decode(aggPublicKeyG2, (BN254.G2Point));
if (decodedAggPublicKeyG2.X[0] == 0 && decodedAggPublicKeyG2.X[1] == 0 &&
   decodedAggPublicKeyG2.Y[0] == 0 && decodedAggPublicKeyG2.Y[1] == 0) {
   return false; // Or revert with a specific error
}
// Potentially add on-curve and subgroup checks here.
```

| MEDIUM-16 | Lack of Relational Validation Between minInclusionVotingPower and maxVotingPower | Pending |
|---|---|---|

### Description

The **ConfigProviderLogic.sol** contract allows setting **minInclusionVotingPower** and **maxVotingPower** independently via **setMinInclusionVotingPower** and **setMaxVotingPower** functions respectively. There are no checks to enforce a logical relationship between these two values, specifically that **minInclusionVotingPower** should not exceed **maxVotingPower**.
Impact: If **minInclusionVotingPower** is set to a value greater than **maxVotingPower**, it could create logically impossible conditions for validator inclusion or other consensus-related mechanisms that rely on these parameters. This might lead to stalled network processes, inability to form valid validator sets, or other system malfunctions.
Lines: **ConfigProviderLogic.sol**, **266-270, 272-276**

### Recommendation

Implement a check to ensure that **minInclusionVotingPower** is less than or equal to **maxVotingPower**. This check should be performed whenever either of these values is updated. For example, when **setMinInclusionVotingPower** is called, verify against the current **maxVotingPower**, and vice-versa. Revert the transaction if the condition **new_min_inclusion_vp <= current_max_vp** (or **current_min_inclusion_vp <= new_max_vp**) is violated.

| MEDIUM-17 | Missing Events for Critical Parameter Changes and Header Commits | Pending |
|---|---|---|

**Description**

Several functions that modify critical state or configurations within **SettlementLogic.sol** do not emit events. These include:

- **setEpochDuration**, **setCommitDuration**, **setRequiredKeyTag**, **setSigVerifier**: These functions change fundamental parameters for future epoch operations.
- **setGenesis**, **commitValSetHeader** (which internally calls **setValSetHeader**): These functions establish new validator set headers, which are critical state updates for the system. The absence of events makes it difficult for off-chain services, monitoring tools, and users to track these important changes efficiently and transparently.

Impact: Reduced observability of critical system state changes. This can hinder:

- Off-chain monitoring and alerting.
- Indexing services that rely on events to build historical views of the system state.
- User interfaces that need to react to configuration updates or new header commitments.
- Auditing and debugging processes.

Lines: **SettlementLogic.sol**, **259-266, 268-272, 274-278, 280-285, 287-296, 298-337, 339-380**

**Recommendation**

Emit events for all significant state-changing operations. For example:

- **event EpochDurationSet(uint48 newEpochDuration, uint48 effectiveFrom);** in **setEpochDuration**.
- **event CommitDurationSet(uint48 newCommitDuration, uint48 effectiveFrom);** in **setCommitDuration**.
- **event RequiredKeyTagSet(uint8 newRequiredKeyTag, uint48 effectiveFrom);** in **setRequiredKeyTag**.
- **event SigVerifierSet(address newSigVerifier, uint48 effectiveFrom);** in **setSigVerifier**.
- **event ValSetHeaderCommitted(uint48 indexed epoch, bytes32 headerHash, address committer);** within **setValSetHeader** (called by **commitValSetHeader** and **setGenesis**). The **headerHash** could be **keccak256(abi.encode(headerStorage))** after it's populated.

| MEDIUM-18 | Prolong Window Calculation Uses Current **commitDuration** Instead of Relevant Historical Value | Pending |
|---|---|---|

### Description

The functions **getCurrentValSetTimestamp**, **getCurrentValSetEpoch**, and **getCurrentPhase** determine the duration of the PROLONG phase using a condition like
**Time.timestamp() < getLastCommittedHeaderCaptureTimestamp() + getCommitDuration() + getProlongDuration()**. The **getCommitDuration()** function (lines 105-107) retrieves the **commitDuration** active at the current **Time.timestamp()**. However, **getLastCommittedHeaderCaptureTimestamp()** refers to the capture time of the last successfully committed header, which could be from a significantly older epoch. Using the current **commitDuration** to calculate the prolong window for this older header means that subsequent changes to the **commitDuration** setting retroactively alter the liveness period of previously established states. If **commitDuration** is reduced, the prolong window for an older, last-committed header could shrink unexpectedly.
Impact: The system might prematurely transition to a **FAIL** state if the global **commitDuration** setting is reduced. This is because the prolong window for the last valid committed header would be calculated using this new, shorter duration, potentially cutting short the time available for recovery or committing a header for the current epoch. This could lead to increased operational friction, unexpected **FAIL** states, and a less predictable liveness guarantee for past epochs during stall scenarios.
Lines: **SettlementLogic.sol**, **61-63, 70-72, 83-85, 91-93, 105-107, 152-154, 160-162**

### Recommendation

To ensure that the liveness properties of a given **getLastCommittedHeaderEpoch** are determined by parameters relevant to that epoch, consider calculating the **commitDuration** component of the prolong window based on the **getLastCommittedHeaderCaptureTimestamp**. This would involve changing the condition to use a historical lookup for **commitDuration**:
**uint48 historicalCommitDuration = uint48(_getSettlementStorage()._commitDuration.upperLookupRecent(getLastCommittedHeaderCaptureTimestamp(), bytes("")));**
And then using
**Time.timestamp() < getLastCommittedHeaderCaptureTimestamp() + historicalCommitDuration + getProlongDuration()**.
This approach ensures that the prolong window for a specific historical header is based on the **commitDuration** that was active or relevant at that historical point, making its liveness period independent of future global configuration changes. If the current behavior is intentional, it should be clearly documented with its implications.

| MEDIUM-19 | Getter Functions Returning Unbounded Arrays May Lead to Denial of Service | Pending |
|---|---|---|

### Description

The public view functions **getOperatorsAt(uint48 timestamp, bytes[] memory hints)** (line 63) and **getOperators()** (line 70) return an **address[] memory**. If the number of registered operators becomes very large, the gas cost associated with constructing, returning, and processing this array (which occurs in the **OperatorManagerLogic** contract but is exposed through **OperatorManager**) could exceed the block gas limit. This would render these functions unusable for on-chain callers and potentially problematic for off-chain clients.
Impact: On-chain contracts relying on these functions to fetch the complete list of operators might experience out-of-gas errors, leading to a denial of service for functionalities that depend on this data. Off-chain clients might also face performance issues or timeouts.
Lines: **OperatorManager.sol**, **63-65, 70-72**

### Recommendation

Implement pagination for functions that return potentially unbounded arrays. For instance, modify **getOperators()** to **getOperators(uint256 startIndex, uint256 count)** allowing callers to retrieve the operator list in manageable chunks. Ensure **OperatorManagerLogic** supports this paginated retrieval. Alternatively, if these functions are primarily for off-chain use, ensure that clients are robust against large data responses and consider alternative event-based or indexed data access patterns for on-chain needs.

| MEDIUM-20 | Functions Without Explicit Role Assignment Default to DEFAULT_ADMIN_ROLE Protection | Pending |
|---|---|---|

### Description

The **_checkPermission()** function is designed to be invoked by a **checkPermission** modifier in contracts inheriting from **OzAccessControl**. This function retrieves the role associated with the current function's selector (**msg.sig**) using **getRole(msg.sig)**. If no specific role has been configured for a selector via **_setSelectorRole()**, the **getRole()** function will return **bytes32(0)**. Subsequently, **_checkPermission()** calls **_checkRole(bytes32(0))**. In the context of the inherited OpenZeppelin **AccessControlUpgradeable** contract, **bytes32(0)** corresponds to the **DEFAULT_ADMIN_ROLE**. Consequently, any function protected by the **checkPermission** modifier that lacks an explicitly assigned selector role will implicitly require the caller to possess the **DEFAULT_ADMIN_ROLE**.
Impact: This default behavior can lead to unintended access restrictions. If a role is not explicitly set for a function that is intended to be accessible by specific non-admin roles or more broadly, it will default to being admin-only. This could render functionalities unusable for their intended audience or inadvertently centralize control, potentially disrupting the contract's intended operational flow.
Lines: **OzAccessControl.sol**, **50-52**

### Recommendation

1. Clearly document this default behavior in the contract's documentation: functions using the **checkPermission** modifier without an explicitly mapped selector role will require the caller to have **DEFAULT_ADMIN_ROLE**.
2. Derived contracts should diligently ensure that all functions intended to be protected by this mechanism have their roles explicitly configured using **_setSelectorRole** during initialization or through an administrative function.
3. For functions intended to be public, the **checkPermission** modifier (which relies on this selector-to-role mapping) should not be used. Alternative access control mechanisms or no restriction should be applied in such cases.

| MEDIUM-21 | Missing Events for Shared Vault Registration and Unregistration | Pending |
|---|---|---|

### Description

The public functions **registerSharedVault** and **unregisterSharedVault** in **SharedVaults.sol** do not emit events upon the successful registration or unregistration of a shared vault. While the actual state modification logic is handled by internal functions inherited from **VaultManager** (which in turn call **VaultManagerLogic**), there are no events specified at this layer or in the deeper logic for these specific operations according to the provided summaries.
Impact: The absence of events makes it challenging for off-chain services, monitoring tools, and users to efficiently track the current list of registered shared vaults and changes to this list. This reduces transparency and hinders the ability to build responsive off-chain systems that rely on shared vault status.
Lines: **SharedVaults.sol**, **19-23, 28-32**

### Recommendation

Emit events such as **SharedVaultRegistered(address indexed sharedVault, address indexed actor)** and **SharedVaultUnregistered(address indexed sharedVault, address indexed actor)** upon successful completion of these operations. These events should ideally be emitted as close to the actual state change as possible, for instance, within the **_registerSharedVault** and **_unregisterSharedVault** implementations in **VaultManager.sol** or, even better, in **VaultManagerLogic.sol**. Since **SharedVaults.sol** implements the **ISharedVaults** interface (not provided, but assumed), ensuring that these significant state changes are observable via events is crucial.

| MEDIUM-22 | Identity Point Acceptance for Signature (**signatureG1**) | Pending |
|---|---|---|

### Description

The **verify** function decodes **signatureG1** from the **signature** bytes via **abi.decode(signature, (BN254.G1Point))**. There is no check to ensure that the decoded **signatureG1** is not the G1 point at infinity (identity element). BLS signatures and related cryptographic scheme elements are typically required to be non-identity elements. If **signatureG1** is the point at infinity, its coordinates (X, Y) would be (0,0), which would affect the calculation of the challenge **alpha**. The pairing equation would then be evaluated with **signatureG1** as identity. While this may not lead to a direct trivial forgery in this specific scheme due to **alpha**'s dependency on **signatureG1**'s coordinates, allowing identity elements for signatures is a deviation from standard cryptographic practice and can lead to weaknesses or unexpected behavior in certain protocols.
Impact: Accepting the point at infinity as a valid **signatureG1** can alter the behavior of the verification logic. It might allow scenarios where a seemingly 'empty' or 'null' signature passes verification under specific conditions, or it could interact unexpectedly with other parts of a larger system relying on this verification. This deviates from the security assumptions of many cryptographic schemes, which rely on signatures being valid, non-identity group elements.
Lines: **SigBlsBn254.sol**, **28**

### Recommendation

Explicitly check if **signatureG1** is the G1 point at infinity after it is decoded. If it is the identity element, the function should return **false**. A G1 point is at infinity if its coordinates are zero (i.e., **signatureG1.X == 0 && signatureG1.Y == 0**). This check is similar to the existing one for **keyG1** (derived from **keyBytes**) using **KeyBlsBn254.zeroKey()**. Consider using a helper function from the **BN254** library if available for checking identity.

| MEDIUM-23 | Key Collision for Indexed **getKey** with Index 0 | Pending |
|---|---|---|

## Description

The **ExtraDataStorageHelper** library provides overloaded functions to generate **bytes32** keys, presumably for storage slot identification. The function **getKey(uint32 verificationType, uint8 keyTag, string memory name)** computes a key by hashing its parameters. Another overload, **getKey(uint32 verificationType, uint8 keyTag, string memory name, uint256 index)**, computes a key by taking the hash of **(verificationType, keyTag, name)** (identical to how the first mentioned function computes its base hash) and adding the **index** to it arithmetically: **bytes32(uint256(hash_base) + index)**. When **index** is 0 in the indexed version, the calculation becomes **bytes32(uint256(hash_base) + 0)**, which simplifies to **hash_base**. This is identical to the key generated by **getKey(uint32 verificationType, uint8 keyTag, string memory name)** for the same **verificationType**, **keyTag**, and **name** parameters. This means that using the non-indexed **getKey** and the indexed **getKey** with **index = 0** (for the same base parameters) will result in the same **bytes32** key.
Impact: If a contract utilizes both **getKey(vt, kt, name)** and **getKey(vt, kt, name, 0)** expecting to address distinct data items or storage slots, they will inadvertently target the same slot. This can lead to critical bugs such as data corruption (one value overwriting another), incorrect state reads, and broken application logic. For example, if one key is used for metadata and the other for the first element of an array, the metadata could be overwritten by the array element or vice-versa.
Lines: **ExtraDataStorageHelper.sol**, **14, 23-25**

## Recommendation

To prevent this collision and ensure that the non-indexed key and the indexed key (even with index 0) are always distinct, modify the key generation logic. A common and effective approach is to offset the indexed calculation slightly. For instance, change the return statement in the indexed **getKey** function (lines 23-25) to:
**return bytes32(uint256(keccak256(abi.encode(GLOBAL_KEY_PREFIX, verificationType, KEY_TAG_PREFIX, keyTag, name))) + index + 1);**
This modification ensures that if **index** is 0, the resulting key will be **hash_base + 1**, which is distinct from **hash_base** generated by the non-indexed version **getKey(verificationType, keyTag, name)**. This change preserves the **base_slot + offset** pattern for arrays while ensuring the base slot itself (if represented by the non-indexed function) is not aliased by the first element of the array (index 0). Callers should be aware that this shifts the effective storage slot for each index by one.

| MEDIUM-24 | **processInclusionProofSha256** Incorrectly Handles Proofs of Height Zero | Pending |
|---|---|---|

### Description

The **processInclusionProofSha256** function does not correctly handle the case where **expectedHeight** is 0. If **expectedHeight** is 0, it means the **leaf** itself should be the **root**, and the **proof** array should be empty. The current assembly code, when **expectedHeight** is 0 (and **proof.length** is also 0 due to the preceding Solidity check), proceeds to load **calldataload(proof.offset)**. Since **proof.length** is 0, **proof.offset** points to the location of the length word itself (which is 0). Thus, it effectively loads 0 as the sibling hash. It then computes **sha256(leaf, 0)** or **sha256(0, leaf)** (depending on **localIndex**) and compares this to the **root**. This is incorrect; it should directly compare **leaf** with **root**.
Impact: Currently, none of the **_PROOF_EXPECTED_HEIGHT** constants used in this library resolve to 0. All calls to **processInclusionProofSha256** use an **expectedHeight** of at least 1. Therefore, this bug is not directly exploitable with the current constant definitions and usage patterns. However, it is a latent bug in a core proof verification function. If the library constants change or if this internal function is used in a new context where an **expectedHeight** of 0 is possible, it would lead to incorrect proof validation: valid proofs of height 0 would likely be rejected, and in a highly unlikely scenario, an invalid proof might be accepted if **root == sha256(leaf, 0)**.
Lines: **ValSetVerifier.sol**, **390-443**

### Recommendation

Modify **processInclusionProofSha256** to correctly handle the **expectedHeight = 0** case. This should be done before the assembly block or as the first check within it. If **expectedHeight** is 0, the function should check if **proof.length** is also 0, and if so, return **leaf == root**. If **proof.length** is not 0, it should return **false**.

```
function processInclusionProofSha256(
  bytes32[] calldata proof,
  bytes32 leaf,
  bytes32 root,
  uint256 localIndex,
  uint256 expectedHeight
) internal view returns (bool valid) {
  if (expectedHeight == 0) {
    if (proof.length == 0) {
      return leaf == root;
    }
    return false; // Proof for height 0 must be empty
  }

  if (proof.length != expectedHeight) return false;

  /// @solidity memory-safe-assembly
  assembly {
    // ... (rest of the existing assembly code)
  }
}
```

| MEDIUM-25 | Potential Signature Bypass via External **hashTypedDataV4** Function Pointer in **setKey** | Pending |
|---|---|---|

### Description

The **setKey** functions (lines 170-177 and 179-186) accept an external function pointer **hashTypedDataV4**. This function is used to compute the EIP-712 typed data hash for signature verification (line 196). If the contract integrating this library provides a malicious, flawed, or user-controllable implementation of **hashTypedDataV4**, the signature verification for key ownership can be subverted. For instance, if **hashTypedDataV4** could be made to return a predictable or constant value, an attacker could forge a signature for an arbitrary **(operator, key)** pair.
Impact: An attacker could register a key for an arbitrary operator without their legitimate signature, potentially leading to unauthorized control or actions if these keys are used for authentication or operational purposes in the broader system. The security of key registration relies heavily on the integrity of the provided **hashTypedDataV4** function.
Lines: **KeyManagerLogic.sol**, **170-177, 179-186, 196**

### Recommendation

While this pattern allows flexibility, it delegates a critical security step. It's crucial that any contract using this library ensures that the **hashTypedDataV4** function provided is secure, non-manipulable, and correctly implements EIP-712 hashing (e.g., using **this.hashTypedDataV4** from a standard, audited EIP-712 implementation like OpenZeppelin's **EIP712** contract). Add explicit documentation within **KeyManagerLogic.sol** warning about the security implications of the **hashTypedDataV4** parameter and the requirement for its secure implementation. Consider making the **setKey** functions that accept this pointer **internal** if direct external use with arbitrary function pointers is not an intended use case for the library, forcing usage through a facade contract that provides a known-good hasher.

| MEDIUM-26 | Unwhitelisting Can Fail if Underlying Unregistration Reverts | Pending |
|---|---|---|

### Description

In **WhitelistSelfRegisterOperators.sol**, the **unwhitelistOperator** function (lines 80-90) and **unwhitelistOperatorVault** function (lines 105-113) attempt to unregister the operator or vault by calling internal functions (**_unregisterOperator** or **_unregisterOperatorVault**) if the whitelist is enabled and the entity is registered. These internal unregistration functions are inherited from **SelfRegisterOperators** and eventually delegate to logic in **OperatorManagerLogic** or **VaultManagerLogic**. If these underlying unregistration calls revert for any reason (e.g., out of gas, an internal assertion failure in the logic contracts, or unexpected state), the entire **unwhitelistOperator** or **unwhitelistOperatorVault** transaction will revert. This means the operator/vault will not be removed from the whitelist state if the unregistration step fails.
Impact: An administrator might be unable to remove an operator or vault from the whitelist if the associated unregistration logic fails. This could prevent the removal of an unwanted or malicious entity from the whitelist, potentially allowing it to re-register if it meets other conditions, undermining the purpose of the unwhitelisting feature. This is similar to issue **FPAUSE-003** for **ForcePauseSelfRegisterOperators**.
Lines: **WhitelistSelfRegisterOperators.sol**, **86-88, 109-111**

### Recommendation

The atomicity of unwhitelisting and unregistering is a design choice.

1. If atomicity is required: The **_unregisterOperator** and **_unregisterOperatorVault** functions (and their underlying implementations in logic contracts) must be exceptionally robust to prevent reverts, especially under conditions where unwhitelisting is a necessary administrative action. Review and test all potential revert paths in the unregistration logic.
2. If unwhitelisting is paramount: If setting the whitelist flag to **false** is critical even if unregistration fails, consider decoupling the unregistration. For example, use a try/catch block for the internal unregistration call (if Solidity version allows for internal try/catch, or adapt the pattern). If unregistration fails, the contract could emit an event signaling the failure and requiring manual intervention or a separate cleanup process, while still proceeding to update the whitelist status. This approach, however, may lead to temporary state inconsistencies where an entity is unwhitelisted but still technically registered in underlying modules.

| MEDIUM-27 | Potential Division by Zero in Decompression if Denominator is Zero | Pending |
|---|---|---|

### Description

In the **decompress** function (line 88), the x-coordinate is calculated using the formula **x = SqrtMod((y^2 - 1) * ModInv(d*y^2 + 1, p) mod p)**. The term **d*y^2 + 1** serves as the denominator. If **d*y^2 + 1** is congruent to **0 mod p**, its modular inverse does not exist. While the **ModInv** function (from the potentially unaudited **@crypto-lib/modular/SCL_modular.sol**) is expected to revert or handle this, the **decompress** function does not explicitly check for this condition before calling **ModInv**. This relies entirely on **ModInv**'s behavior.

Impact: If **ModInv** does not correctly handle the case where its first argument is non-invertible (e.g., congruent to **0 mod p**) by reverting, it might return an incorrect value (e.g., 0 or an error code not causing a revert). This could lead to **SqrtMod** being called with an erroneous input, potentially resulting in incorrect x-coordinate calculation, failed key validation for valid keys, or, in a worst-case scenario with a severely flawed **ModInv**, acceptance of an invalid key. The primary risk is a denial of service if **ModInv** reverts correctly, but the lack of an explicit check in **decompress** makes the code less robust and more dependent on the external library's precise error handling.

Lines: **KeyEddsaCurve25519.sol**, **88**

### Recommendation

For improved robustness and clarity, explicitly check if the denominator **denom = addmod(mulmod(d, y2, p), 1, p)** is zero before calling **ModInv(denom, p)**. If **denom** is zero, the **decompress** function should revert (e.g., **revert KeyEddsaCurve25519_InvalidKey();**) as the point is invalid or leads to an undefined x-coordinate. This makes the function's behavior explicit rather than solely relying on the external **ModInv** to revert.

Example addition before line 88:

```
// ...
// uint256 num = addmod(y2, pMINUS_1, p);
// uint256 denom = addmod(mulmod(d, y2, p), 1, p);
// if (denom == 0) {
//     revert KeyEddsaCurve25519_InvalidKey(); // Point leads to division by zero in x-coordinate formula
// }
// x = SqrtMod(mulmod(num, ModInv(denom, p), p));
// ...
```

This recommendation assumes that **ModInv** itself should also robustly handle non-invertible inputs by reverting.

| MEDIUM-28 | Slashing Window Parameter Can Only Be Decreased and May Become Permanently Zero | Pending |
|---|---|---|

### Description

The **setSlashingWindow** function in **VaultManagerLogic.sol** (as per its summary) enforces that a new **slashingWindow** value must be strictly less than the current value, due to the check:
**if (slashingWindow >= getSlashingWindow()) { revert IVaultManager.VaultManager_SlashingWindowTooLarge(); }**. This design means the slashing window can only ever be decreased. A critical consequence is that if the slashing window is set to 0, this condition (**new_value >= 0**) will always be true for any non-negative **new_value**, causing the transaction to revert. Therefore, once the slashing window is 0, it cannot be changed to any other value through this function.
Impact: This restriction means the system's slashing window can only become more stringent over time. If a longer slashing window is ever required due to evolving network conditions or a re-assessment of security parameters, it would be impossible to implement without a contract upgrade or a different mechanism. If the window is set to 0 (e.g., by mistake or for a specific reason), it becomes permanently fixed at 0, which could effectively disable time-based slashing considerations or lead to unintended consequences if a non-zero window is later desired.
Lines: **VaultManagerLogic.sol**, **480-483**

### Recommendation

Re-evaluate the intended flexibility for the **slashingWindow** parameter.

1. If strictly decreasing is the desired behavior, this should be clearly documented, including the terminal state at zero.
2. If it should be possible to increase the window or set it from 0 to a non-zero value, the validation logic **if (slashingWindow >= getSlashingWindow())** must be modified. For instance, to allow setting from 0, the check could be **if (getSlashingWindow() != 0 && slashingWindow >= getSlashingWindow())**.
3. Consider if there should be an absolute maximum for the slashing window or if other constraints are more appropriate than just allowing decreases.

| MEDIUM-29 | Decoded **ValidatorData** Array Length Mismatch with Proof's Specified Length Field | Pending |
|---|---|---|

### Description

The function reads a **length** value (referred to as **L_field** here for clarity) directly from the **proof** calldata at a fixed offset ( **proof[224:256]**). It then uses this **L_field** to determine the slice of **proof** to be decoded as **ValidatorData[]** (i.e., **proof[192 : 256 + L_field * 96]**). The ABI decoding process for **ValidatorData[]** will determine its own length (**L_abi**) based on the ABI-encoded length field found within the slice itself (typically at **slice[32:64]**, if **slice[0:32]** is an offset of **0x20**). The subsequent loop for aggregating non-signer information (lines 72-77) iterates **L_field** times, accessing **validatorsData[i]**. If **L_abi** (the actual length of the decoded **validatorsData** array) is less than **L_field**, an attempt to access **validatorsData[i]** for **i >= L_abi** will result in an out-of-bounds revert.

Impact: A malformed **proof** can cause the **verifyQuorumSig** function to revert due to an out-of-bounds array access when processing **validatorsData**. This can occur if the **length** field in the proof (**L_field**) specifies a larger number of validators than what is actually decodable or present in the **ValidatorData** portion of the proof (resulting in **L_abi < L_field**). This creates a denial of service vector for specific, malformed proofs.

Lines: **SigVerifierBlsBn254Simple.sol**, **58-59, 69, 72-77**

### Recommendation

After ABI-decoding **validatorsData** from the proof slice, explicitly verify that the length of the decoded array ( **validatorsData.length**, which is **L_abi**) is equal to the **length** value (**L_field**) read from the proof structure. If they do not match, the proof is malformed, and the function should revert.

Add the following check after line 69:

```
ValidatorData[] memory validatorsData = abi.decode(proof[192:256 + length * 96], (ValidatorData[]));
if (validatorsData.length != length) {
  revert SigVerifierBlsBn254Simple_ProofFormatError(); // Define a suitable custom error for this case
}
```

| MEDIUM-30 | Failure in Unregistration Logic Prevents Unwhitelisting | Pending |
|---|---|---|

### Description

In **WhitelistSelfRegisterOperators.unwhitelistOperator** (lines 80-90) and **WhitelistSelfRegisterOperators.unwhitelistOperatorVault** (lines 105-113), if the whitelist is enabled and the operator or vault is currently registered, the contract attempts to unregister them by calling **_unregisterOperator(operator)** or **_unregisterOperatorVault(operator, vault)**. These internal unregistration functions, inherited from **SelfRegisterOperators** and its bases, could revert due to issues in **OperatorManagerLogic** or **VaultManagerLogic** (e.g., out-of-gas errors, internal assertions failing, or other unexpected states). If such an unregistration call reverts, the entire unwhitelisting transaction will also revert, preventing the operator or vault from being removed from the whitelist.

Impact: The system may be unable to remove an operator or vault from the whitelist if the dependent unregistration logic fails. This could hinder administrative actions, such as removing a compromised or non-compliant entity from the whitelist, if the underlying unregistration process encounters an error.

Lines: **WhitelistSelfRegisterOperators.sol**, **87-89, 110-112**

### Recommendation

The atomicity of unwhitelisting and unregistering is a design choice. If unwhitelisting is paramount even if the underlying unregistration fails, consider decoupling the unregistration step. For example, use a try/catch block (if Solidity version and context allow, typically for external calls not internal ones) for the unregistration call, or make the unregistration conditional and emit an event if it fails, signaling the need for manual intervention or a separate cleanup process. This would allow the whitelist flag to be updated regardless. However, this could lead to temporary inconsistencies where an entity is unwhitelisted but still appears registered in some underlying modules. If atomicity is strictly required, then the **_unregisterOperator** and **_unregisterOperatorVault** functions (and their underlying implementations) must be made exceptionally robust.

| MEDIUM-31 | Denial of Service via Memory Allocation in **getKeysAt** due to Empty Hints and Large Result Sets | Pending |
|---|---|---|

### Description

The **getKeysAt** functions (lines 100-115 and 127-143) retrieve **keyTags** or **operators**, which can be arrays of significant length. These functions then use **InputNormalizer.normalize** on sub-arrays of the provided **hints** data (e.g., **operatorKeysHints.keyHints**, **operatorsKeysHints.operatorKeysHints**). The **InputNormalizer.normalize** function, when called with an empty input array and a non-zero **length L**, allocates a new array of size **L** (e.g., **new bytes[](L)** or **new bytes[][](L)**). If the **hints** sub-array is empty (e.g., the overall **hints** parameter was empty or the specific sub-array within **hints** was empty) and the determined length (**keyTags.length** or **operators.length**) is very large, the memory allocation for **new bytes[](length)** or **new bytes[][](length)** can consume excessive gas. This can lead to the transaction running out of gas and reverting.
Impact: Callers of **getKeysAt** providing empty or minimal **hints** for operators or timestamps that have a large number of associated keys or sub-operators could experience a denial of service. The transaction may fail due to out-of-gas errors caused by the large memory allocation, rendering these getter functions unusable in such scenarios.
Lines: **KeyManagerLogic.sol**, **100-115, 127-143, 108, 137**

### Recommendation

1. Consider adding a reasonable upper limit to **keyTags.length** and **operators.length** before they are used as the **length** argument for **InputNormalizer.normalize** when the corresponding hint array is empty. If the determined length exceeds this limit, the function could revert early or choose not to normalize (which might require changes to subsequent logic that expects a normalized array).
2. Alternatively, document that providing appropriately sized hint arrays is mandatory for queries involving large result sets to prevent excessive gas usage from memory allocation. Users calling these functions should be aware of this behavior.
3. The **InputNormalizer** library itself could be modified to cap the allocation size or return an error/empty array if an excessively large length is requested for an empty input array, but this is a broader change outside **KeyManagerLogic**.

| MEDIUM-32 | Slashing Window Can Only Be Decreased and Potentially Permanently Set to Zero | Pending |
|---|---|---|

### Description

The **setSlashingWindow** function contains the condition
**if (slashingWindow >= getSlashingWindow()) { revert IVaultManager.VaultManager_SlashingWindowTooLarge(); }**. This logic means that the new **slashingWindow** must be strictly less than the current **slashingWindow**. Consequently, the slashing window can only ever be decreased. More critically, if the current slashing window is 0, any attempt to set a new **slashingWindow** (which must be a non-negative **uint48**) will satisfy **slashingWindow >= 0**, causing a revert. This effectively means that once the slashing window is set to 0, it cannot be changed again through this function.
Impact: The slashing window can only become stricter (shorter) over time. If an increase in the slashing window is ever required (e.g., due to evolving network conditions or protocol needs), it would be impossible without a contract upgrade or redeployment. If the slashing window is set to 0, it becomes permanently fixed at 0, potentially disabling time-based slashing considerations if that was not the long-term intention.
Lines: **VaultManagerLogic.sol**, **480-483**

### Recommendation

Re-evaluate the intended behavior for updating the **slashingWindow**.

1. If the intention is strictly to only allow decreases, the error message could be more precise, e.g., **VaultManager_SlashingWindowCannotBeIncreasedOrUnchanged**.
2. Address the case where **getSlashingWindow()** is 0. If it should be possible to set it from 0 to a non-zero value, the condition **slashingWindow >= getSlashingWindow()** needs adjustment, for example, by adding an explicit check **&& getSlashingWindow() != 0** or by changing the logic to allow setting from 0.
3. If a maximum allowable slashing window is intended, the check should be **if (newSlashingWindow > MAX_ALLOWED_SLASHING_WINDOW)**. Consider if there should be a minimum slashing window or if the current behavior is entirely intentional including the permanent zero state.

| MEDIUM-33 | Gas Inefficiency in EIP-712 Hash Computations | Pending |
|---|---|---|

## Description

The **_EIP712NameHash()** and **_EIP712VersionHash()** functions recompute the **keccak256** hash of the EIP-712 domain name and version strings respectively on every call after the contract has been initialized. This occurs when **hashTypedDataV4()** or **hashTypedDataV4CrossChain()** are invoked. The **initialize()** function stores the raw **name** and **version** strings (in **$._name** and **$._version**) but explicitly sets the **_hashedName** and **_hashedVersion** storage slots to zero. These zeroed slots are used by **eip712Domain()** to ensure that **$._name** and **$._version** are used. The **_hashedName** and **_hashedVersion** slots are otherwise only read by **_EIP712NameHash()** and **_EIP712VersionHash()** as a fallback mechanism if the contract was upgraded from a previous version and **initialize()** has not yet been called on the new version. After **initialize()** is called, **$._name** and **$._version** are read from storage and re-hashed in **_EIP712NameHash()** and **_EIP712VersionHash()** respectively during every signature verification, leading to avoidable gas costs.
Impact: Increased gas consumption for each call to **hashTypedDataV4** or **hashTypedDataV4CrossChain**, which are typically used for every EIP-712 signature verification. This can lead to higher transaction costs for users interacting with contracts that use this EIP-712 logic, especially if domain names or versions are long strings.
Lines: **OzEIP712Logic.sol**, **151-165, 173-187**

## Recommendation

To optimize gas usage, consider caching the hashes of the name and version. This can be achieved by introducing two new storage slots (e.g., **_cachedDomainNameHash** and **_cachedDomainVersionHash**) to the **OzEIP712Storage** struct.
In the **initialize(IOzEIP712.OzEIP712InitParams memory initParams)** function:

1. Set **$._name = initParams.name;** and **$._version = initParams.version;**.
2. Set **$._hashedName = 0;** and **$._hashedVersion = 0;** (as is currently done, this signals to **eip712Domain()** that the new **_name** and **_version** fields are authoritative).
3. Compute and store **$._cachedDomainNameHash = keccak256(bytes(initParams.name));** and **$._cachedDomainVersionHash = keccak256(bytes(initParams.version));** in the new storage slots.

Then, modify **_EIP712NameHash()** to:

```
function _EIP712NameHash() public view returns (bytes32) {
   OzEIP712Storage storage $ = _getOzEIP712Storage();
   // If $._hashedName is 0, it means initialize() has been called for the current contract version (or it's a fresh deployment).
   // In this case, use the cached hash derived from $._name.
   if ($._hashedName == 0) {
      return $._cachedDomainNameHash;
   } else {
      // $._hashedName is non-zero, indicating an upgrade from an older version
      // where initialize() for this version's logic has not yet been run.
      // $._name would be empty, so use the $._hashedName from the old storage.
      return $._hashedName;
   }
}
```

A similar modification should be applied to **_EIP712VersionHash()**. This approach avoids re-hashing **_name** and **_version** after initialization while preserving the existing upgrade path compatibility.

| MEDIUM-34 | _remove with key = 0 Can Lead to Revert Due to Checkpoint Key Order Violation | Pending |
|---|---|---|

### Description

The **_remove** function allows a **key** of 0. If an element **value** has previously had its removal status checkpointed with a positive **key** (e.g., **isRemoved.push(100, 1)**), a subsequent call to **_remove(set, 0, value)** will attempt **set._statuses[value].isRemoved.push(0, 1)**. The underlying OpenZeppelin **Checkpoints.Trace.push(key, value)** function typically requires that **key** be greater than or equal to the last stored key. If the last stored key in the **isRemoved** trace for **value** is positive, passing **key = 0** will violate this condition and cause the **push** operation to revert.
Impact: Attempts to remove elements using **key = 0** after they have been checkpointed with positive keys will fail, leading to a denial of service for that specific removal operation. This could prevent elements from being correctly marked as removed at key 0 if key 0 has a special meaning (e.g., genesis removal), assuming adds also allowed key 0.
Lines: **PersistentSet.sol**, **43-51, 48**

### Recommendation

Enforce that the **key** parameter in **_remove** must be strictly positive and consistent with the ordering requirements of the **Checkpoints** library (typically non-decreasing). The recommendation for a previous finding (PS-004, concerning **_add** and **key=0**) to **require(key > 0, ...)** should be extended to **_remove** as well. If key 0 is intended to have a special meaning for removals, the interaction with the Checkpoints library needs careful design to ensure keys are always pushed in a valid order (e.g., by ensuring key 0 operations happen before any positive key operations for that checkpoint trace).

| MEDIUM-35 | Gas Intensive Default Hint Allocation in _valuesAt | Pending |
|---|---|---|

### Description

In the **_valuesAt** function, if the provided **hints** array is empty (**hints.length == 0**), **InputNormalizer.normalize** allocates a new **bytes[] memory** array of size **set._elements.length**. This new array is initialized with empty **bytes** arrays (**bytes("")**) for each element. If **set._elements.length** (the total number of unique elements ever added) is very large, the gas cost of allocating this large array of default hints can be substantial.
Impact: Calls to **_valuesAt** with an empty **hints** parameter can become unexpectedly expensive for sets with a large history of unique elements, purely due to the memory allocation for default hints. This could lead to out-of-gas errors or higher transaction costs than anticipated, potentially hindering the usability of the function for large sets when specific hints are not available.
Lines: **PersistentSet.sol**, **76, 79-83**

### Recommendation

Callers of **_valuesAt** should strive to provide meaningful, correctly-sized **hints** arrays whenever possible, especially for sets with many elements, to avoid the gas cost of default hint allocation and to benefit from more efficient checkpoint lookups. If providing hints is generally not feasible, users should be aware of this potentially high baseline gas cost for **_valuesAt**. Consider evaluating if **InputNormalizer** could be modified or bypassed in a way that avoids allocating a large default **hints** array if no hints are provided (e.g., by directly passing an empty **bytes** as hint to **_containsAt** if **hints** is empty or out of bounds for the current index).

| MEDIUM-36 | **deserialize** Does Not Verify Reconstructed Point Satisfies Curve Equation | Pending |
|---|---|---|

### Description

The **deserialize** function reconstructs a G1 point **(X, Y)** from a compressed byte representation. After obtaining **X** and **derivedY** (from **BN254.findYFromX(X)** at line 63), it directly forms the point. However, it fails to verify that this reconstructed point actually lies on the elliptic curve by checking if **derivedY^2 = X^3 + b (mod p)**. The **wrap** function, which is the library's standard validation mechanism for G1 points, performs this crucial on-curve check (lines 30-31 by comparing **derivedY^2** with **beta**, where **beta** is **X^3+b**). The **deserialize** function, by using **(, uint256 derivedY) = BN254.findYFromX(X);**, discards the **beta** value that would be needed for this check, or it fails to recompute **beta** and perform the check. This omission in **deserialize** means it can produce a **KEY_BLS_BN254** struct representing a point that is not on the BN254 curve.
Impact: If **deserialize** is used to create **KEY_BLS_BN254** objects, and these objects are then used directly without further validation through **wrap** or **fromBytes**, the system might operate with points that are not valid curve members. This could lead to failed cryptographic operations (e.g., precompile calls for pairings or additions/multiplications would likely revert), incorrect behavior in protocols relying on these keys, or security vulnerabilities if downstream code incorrectly assumes point validity.
Lines: **KeyBlsBn254.sol**, **63-64**

### Recommendation

Modify the **deserialize** function to include an explicit check that the reconstructed point **(X, derivedY)** satisfies the curve equation. The most comprehensive and consistent approach is to reconstruct the **BN254.G1Point** and then pass it to the **wrap** function for full validation, as **wrap** already performs this check. If calling **wrap** is not desired (e.g., for specific gas optimization, though **wrap** is **view**), then **deserialize** must retrieve or recompute **beta = X^3+b** and perform the on-curve check **mulmod(derivedY, derivedY, BN254.FP_MODULUS) == beta** itself, similar to lines 29-32 in the **wrap** function.

| | | |
|---|---|---|
| MEDIUM-37 | Non-compliance with RFC 8032 Ed25519 Decoding: Failure to Reject (x=0, sign_bit=1) Keys | Pending |

### Description

The Ed25519 key validation process in the **wrap** function, which relies on **decompress**, does not fully comply with RFC 8032, Section 5.1.3 ("Decoding"). Specifically, Step 3 of the decoding process states: "Third, if x_0 = 0 and x_255 = 1, then FAIL." Here, x_0 is the canonical x-coordinate (0 <= x_0 < p) and x_255 is the sign bit derived from the compressed key (MSB of the y-coordinate part, effectively the LSB of the original x).
The current **decompress** function correctly computes an x-coordinate whose LSB matches the provided sign bit (x_255). However, neither **decompress** nor **wrap** explicitly checks and rejects the case where the resulting canonical x-coordinate is 0, but the sign bit (x_255) was 1. This means the library will accept non-canonical encodings of points (0,1) (the identity point) and (0, p-1) if they are encoded with their x-sign bit set to 1. The standard mandates that these specific encodings are invalid.
Impact: The library accepts certain non-canonical key encodings that should be rejected according to RFC 8032. This can lead to interoperability issues with other Ed25519 implementations that strictly adhere to the standard. If systems rely on unique, canonical byte representations for keys, accepting these non-canonical forms could cause inconsistencies or allow multiple representations for the same cryptographic key. While the underlying points (0,1) and (0,p-1) are on the curve, their encoding with x_255=1 is invalid. This affects the integrity of key validation with respect to standard compliance.
Lines: **KeyEddsaCurve25519.sol**, **32-37, 82-92**

### Recommendation

Modify the **decompress** function (or the **wrap** function immediately after calling **decompress**) to include the check specified in RFC 8032, Section 5.1.3, Step 3. After the x-coordinate (**x_final**) has been determined such that its LSB matches the sign bit (**sign_bit_x = kPubC >> 255**), add the following validation:

```
// Inside decompress function, after x has been finalized:
// uint256 canonical_x = mulmod(x_final, 1, p); // Ensures x is in [0, p-1]
// if (canonical_x == 0 && sign_bit_x == 1) {
//     revert KeyEddsaCurve25519_InvalidKey();
// }
```

Alternatively, this check can be placed in **wrap** using the **x** and **y** returned by **decompress**, and **uint256(key.value) >> 255** for the sign bit:

```
// Inside wrap function, after (x, y) = key.decompress();
// uint256 sign_bit_x = uint256(key.value) >> 255; // Assuming key.value is big-endian here.
// // If key.value is little-endian (as it should be for Ed25519), then kPubC >> 255 from decompress is needed.
// // Let's assume decompress returns the sign_bit_x as well, or re-calculate kPubC.
// uint256 kPubC_val = SCL_sha512.Swap256(uint256(key.value));
// uint256 sign_bit_x_val = kPubC_val >> 255;
// if (mulmod(x, 1, p) == 0 && sign_bit_x_val == 1) {
//     revert KeyEddsaCurve25519_InvalidKey();
// }
```

It is cleaner to perform this check within **decompress** as it has direct access to **kPubC** and **sign_bit_x**.

| MEDIUM-38 | Public Getter Functions Returning Potentially Unbounded Arrays May Lead to Denial of Service | Pending |
|---|---|---|

### Description

Several public view functions in **KeyManager.sol** delegate calls to **KeyManagerLogic** to retrieve lists of keys or operators. These functions include **getKeysAt(address operator, ...)** (lines 92-98), **getKeys(address operator)** (lines 103-107), **getKeysAt(uint48 timestamp, ...)** (lines 112-114), and **getKeys()** (lines 119-121). These functions return arrays like **Key[] memory** or **OperatorWithKeys[] memory**. If the number of keys associated with an operator, or the total number of operators with registered keys, becomes very large, the gas cost to construct, return, and process these arrays (within **KeyManagerLogic** and then by the EVM for the caller) could exceed the block gas limit or the caller's gas allowance. This could render these functions unusable for on-chain callers.
Impact: On-chain contracts relying on these functions to fetch complete lists of keys or operators might experience out-of-gas errors. This would constitute a denial of service for functionalities that depend on this data. Off-chain clients might also face performance issues or timeouts when dealing with very large responses.
Lines: **KeyManager.sol**, **92-98, 103-107, 112-114, 119-121**

### Recommendation

To mitigate the risk of denial of service for on-chain users, consider the following:

1. Implement pagination for these getter functions in **KeyManagerLogic** and expose these paginated versions through **KeyManager.sol**. For example, a function like **getKeys(address operator, uint256 startIndex, uint256 count)** would allow retrieval in manageable chunks.
2. Clearly document the potential for high gas costs for these functions and advise users to employ them cautiously, especially in on-chain contexts or with potentially large datasets.
3. Encourage the use of emitted events for building off-chain indexed views of key data where feasible, reducing reliance on direct on-chain full list retrieval.

| MEDIUM-39 | Initialization Can Be Front-Run with Attacker-Controlled Parameters | Pending |
|---|---|---|

## Description

The **initialize** function (line 19) is **public** and protected by the **initializer** modifier, allowing it to be called by any address, but only once. If a legitimate deployer submits a transaction to initialize the **Master** contract with specific parameters, an attacker observing the transaction mempool could attempt to front-run it. The attacker could call **initialize** with their own parameters before the deployer's transaction is mined. For instance, they could set **defaultAdmin** (line 18) to their own address, or provide malicious/malformed **settlementInitParams** or **configProviderInitParams** (lines 16-17) to put the contract into an undesirable or non-functional state.
Impact: If an attacker successfully front-runs the initialization, they could gain administrative control of the system by setting themselves as the **defaultAdmin**. Alternatively, they could initialize the contract with invalid parameters for **Settlement** or **ConfigProvider**, potentially rendering the system unusable or misconfigured from the start. This would necessitate a redeployment of the contract system.
Lines: **Master.sol**, **15-25**

## Recommendation

The deployer should take precautions to mitigate the risk of front-running the **initialize** transaction. Options include:

1. Using a private transaction submission service (e.g., Flashbots Protect RPC) to prevent the transaction from being publicly visible in the mempool.
2. Setting a sufficiently high gas price for the legitimate initialization transaction to make it economically challenging for an attacker to front-run profitably.
3. If **Master** is deployed via a factory contract, consider having the factory call **initialize** in the same transaction that deploys the **Master** contract instance. This atomically deploys and initializes, preventing external front-running of the **initialize** call on the new instance.
4. For future or more complex initializations, critical parameters could be required to be signed by a trusted key, and the **initialize** function would verify this signature, rather than accepting raw parameters directly.

| MEDIUM-40 | **deserialize** Can Produce Keys with Y-Coordinate Equal to Field Modulus | Pending |
|---|---|---|

### Description

The **deserialize** function, when reconstructing a G1 point, might produce a **KEY_BLS_BN254** struct where **key.value.Y** is equal to **BN254.FP_MODULUS**. This can happen in two primary scenarios within **deserialize**'s current logic:

1. If **BN254.findYFromX(X)** returns **derivedY = 0** (meaning the point is on the x-axis), and the compression flag **(uint256(compressedKey) & 1) == 1** indicates that the alternative Y value should be used. The function calls **BN254.negate(key.value)**. If **key.value** was **(X,0)**, and **BN254.negate** has a bug where it returns **(X, FP_MODULUS)** for such points (as suggested by finding BN254-001 for the **BN254.sol** library), then **key.value.Y** becomes **FP_MODULUS**.
2. Even if **BN254.negate** is correct for points on the x-axis (i.e., **negate((X,0))** yields **(X,0)**), the logic of choosing the 'other' Y is effectively **Y = P - derivedY**. If **derivedY** is 0, this results in **P - 0 = P**. The **deserialize** function does not explicitly check if the resulting **Y** is **FP_MODULUS**. A Y-coordinate equal to **BN254.FP_MODULUS** is invalid for elliptic curve operations in Ethereum precompiles, which expect coordinates to be strictly less than the modulus. The library's **wrap** function correctly includes the check **keyRaw.Y < BN254.FP_MODULUS** (implicitly by **keyRaw.Y >= BN254.FP_MODULUS** causing a revert). By not performing this check, **deserialize** can produce an invalid key.

Impact: **deserialize** can return a key struct with an invalid Y-coordinate (**Y = FP_MODULUS**). This can lead to failed cryptographic operations if such a key is used with precompiles, incorrect behavior in protocols relying on these keys, or inconsistencies if such keys are stored or used, as they do not represent valid field elements in the canonical sense required by cryptographic operations.
Lines: **KeyBlsBn254.sol**, **64-67**

### Recommendation

Ensure that the **deserialize** function validates that the final **key.value.Y** is strictly less than **BN254.FP_MODULUS**. The most robust and consistent method to achieve this is to call the library's own **wrap** function on the fully reconstructed **BN254.G1Point** before returning from **deserialize**. The **wrap** function already includes the necessary check: **if (keyRaw.X >= BN254.FP_MODULUS || keyRaw.Y >= BN254.FP_MODULUS) { revert KeyBlsBn254_InvalidKey(); }**. This would centralize all key validation logic within **wrap**. Alternatively, if direct use of **wrap** is avoided for some reason, an explicit check **if (key.value.Y >= BN254.FP_MODULUS) { revert KeyBlsBn254_InvalidKey(); }** must be added at the end of the **deserialize** function before returning the key.

| MEDIUM-41 | Potential Denial of Service from Unbounded Array Returns in Basic List Getter Functions | Pending |
|---|---|---|

### Description

The functions **getTokens()** (line 78), **getSharedVaults()** (line 100), and **getOperatorVaults(address operator)** (line 130), along with their **At(timestamp, hints)** counterparts (e.g., **getTokensAt** on line 74), retrieve and return arrays of addresses. If the number of registered tokens, shared vaults, or operator-specific vaults becomes very large, the gas cost to construct and return these arrays can exceed the block gas limit or a caller's gas stipend. This is because these functions iterate or fetch all matching elements from the underlying **PersistentSet** data structure.
Impact: On-chain callers relying on these functions to retrieve complete lists of tokens, shared vaults, or operator vaults might experience out-of-gas errors if the lists are excessively large. This could lead to a denial of service for functionalities that depend on this data. Off-chain clients might also face performance issues or timeouts when querying nodes.
Lines: **VaultManagerLogic.sol**, **74-76, 78-80, 96-98, 100-102, 125-127, 129-133**

### Recommendation

For on-chain scenarios that require accessing these lists, consider implementing and exposing paginated versions of these getter functions (e.g., **getTokens(uint256 startIndex, uint256 count)**). This would allow data retrieval in manageable chunks. For off-chain usage, clearly document the potential for high gas costs or large data responses. The existing **Length** functions (e.g., **getTokensLength**) can help callers anticipate the size of the data.

| MEDIUM-42 | Potential Revert in **_add** When Re-Adding with Out-of-Order Key | Pending |
|---|---|---|

### Description

When an element **value** is re-added (i.e., it was previously added and then removed, so **set._statuses[value].addedAt > 0** and **set._statuses[value].isRemoved.latest() == 1**), the **_add** function calls **set._statuses[value].isRemoved.push(key, 0)** (line 33) to mark it as no longer removed, using the provided **key**. The underlying **Checkpoints.Trace.push** mechanism, used by **isRemoved**, requires that keys be pushed in non-decreasing order. If the element **value** was last marked as removed at **key_remove** and is now being re-added with a new **key_add** such that **key_add < key_remove**, the call to **push(key_add, 0)** will revert. This is because it violates the key order requirement of the checkpoint system for the **isRemoved** trace.
Impact: Transactions attempting to re-add an element with a key (e.g., timestamp) that is earlier than its last removal key will revert. This can lead to denial of service for such operations if keys are not managed strictly in non-decreasing order by the calling contract for an element's removal/re-addition lifecycle. The library does not enforce this specific ordering between a previous removal key and a subsequent re-addition key for the same element.
Lines: **PersistentSet.sol**, **33**

### Recommendation

Callers of the **_add** function (and by extension, **Bytes32Set.add** and **AddressSet.add**) must ensure that if they are re-activating a previously removed element, the **key** provided for the re-addition is greater than or equal to the **key** that was used when the element was last marked as removed. This responsibility lies with the calling contract, as the library does not internally check the last removal key against the new re-addition key. The documentation for **PersistentSet** should clearly state this key ordering requirement for re-adding elements. Additionally, if the fix for PS-004 and PS-ADDITIONAL-001 (requiring **key > 0**) is implemented, this constraint applies to positive keys.

| INFORMATIONAL-01 | Unused **using for** Directives | Pending |
|---|---|---|

### Description

The **OperatorManager** contract includes **using Arrays for address[];** (line 16) and **using Checkpoints for Checkpoints.Trace208;** (line 15). However, methods from the **Arrays** library are not invoked on **address[]** types, and methods from **Checkpoints** are not invoked on **Checkpoints.Trace208** types directly within the **OperatorManager.sol** file. These directives might be intended for use by derived contracts or could be remnants from previous development phases.
Impact: Slightly increases code size and can be minorly confusing for readers, but has no direct security impact or effect on runtime behavior for **OperatorManager** itself.
Lines: **OperatorManager.sol**, **15, 16**

### Recommendation

If these **using for** directives are not utilized by **OperatorManager** and are not specifically intended to be inherited and used by derived contracts in a way that requires their presence in the base, consider removing them to improve code clarity and reduce clutter. If they are indeed for derived contracts, they can be left as is, though it's good practice to keep such declarations closer to their actual use if possible.

| INFORMATIONAL-02 | Critical Parameter Change Lacks Event Emission | Pending |
|---|---|---|

### Description

The **setEpochDuration(uint48 epochDuration)** function, located at lines 92-96, allows a permissioned actor to change the system's epoch duration. This is a significant operational parameter that dictates the length of epochs. Currently, this state change, which is delegated to **EpochManagerLogic.setEpochDuration**, does not result in an event being emitted. The absence of an event makes it more difficult for off-chain services, monitoring tools, and users to track and react to updates of this critical parameter.
Impact: Reduced transparency for a key system parameter change. Off-chain monitoring systems, indexing services, and user interfaces may not be easily notified of epoch duration updates. This can lead to the use of stale information or delayed reactions to important configuration changes within the system.
Lines: **EpochManager.sol**, **92-96**

### Recommendation

Ensure that an event is emitted when the epoch duration is changed. This event should ideally be emitted within the **EpochManagerLogic.setEpochDuration** function, as that is where the state modification actually occurs. The event should include details such as the new epoch duration and, if possible, the epoch number or timestamp from which this new duration becomes effective. For example, an event like
**event EpochDurationUpdated(uint48 newEpochDuration, uint48 effectiveFromTimestamp, uint48 effectiveFromEpochIndex);**
could be emitted.

| INFORMATIONAL-03 | Unused EnumerableSet Import and using for Directive | Pending |
|---|---|---|

### Description

The **PersistentSet.sol** library imports **EnumerableSet** from OpenZeppelin (line 7) and includes the directive **using EnumerableSet for EnumerableSet.Bytes32Set;** (line 11). However, **EnumerableSet.Bytes32Set** is not utilized anywhere within the **PersistentSet** library's **Set** struct or its associated private/internal functions. The library implements its own set-like data structures and logic.
Impact: The unused import and **using for** directive do not affect the runtime functionality or security of the contract. They represent dead code, which can slightly increase deployment size (though modern compilers might optimize this out) and add minor clutter, potentially confusing developers who are reading or maintaining the code.
Lines: **PersistentSet.sol**, **7, 11**

### Recommendation

To improve code clarity and maintainability, remove the **import {EnumerableSet} ...** statement (line 7) and the **using EnumerableSet for EnumerableSet.Bytes32Set;** directive (line 11) if they are not intended for current or future use in this library.

| INFORMATIONAL-04 | **remove** and **contains** Functions Silently Ignore Out-of-Range Key Tags | Pending |
|---|---|---|

### Description

The **remove** (lines 50-52) and **contains** (lines 42-44) functions do not explicitly validate if the input **keyTag** is within the valid range (i.e., **keyTag < TOTAL_KEY_TAGS**).

- In **remove**, if **keyTag >= 128**, **uint128(1 << keyTag)** evaluates to 0. Thus, **keyTagsSerialized & ~uint128(0)** becomes **keyTagsSerialized & type(uint128).max**, which simplifies to **keyTagsSerialized**. The function silently performs no operation for an out-of-range **keyTag**.
- In **contains**, if **keyTag >= 128**, **uint128(1 << keyTag)** is 0. So, **keyTagsSerialized & 0 > 0** will always evaluate to false. The function silently returns **false** for an out-of-range **keyTag**. While this behavior of effectively ignoring invalid tags might not directly lead to incorrect state modifications within the bitmask itself, it lacks explicitness. Callers might expect these functions to revert if an invalid **keyTag** is provided, similar to how **getKeyTag**, **getType**, and **getTag** behave.

Impact: This behavior can lead to subtle bugs or misunderstandings in contracts using this library if they expect these functions to fail on invalid inputs. For example, logic attempting to remove an invalid tag and then checking its absence (which **contains** would report as **false**) might proceed under the false assumption that the removal was successful or that the tag was never there. It reduces the library's explicitness and can make debugging more challenging.

Lines: **KeyTags.sol**, **43, 51**

### Recommendation

For consistency and robustness, add an explicit check at the beginning of both the **remove** and **contains** functions. If **keyTag >= TOTAL_KEY_TAGS**, these functions should revert with **KeyTags_InvalidKeyTag()**.

Example for **remove**:

```
function remove(uint128 keyTagsSerialized, uint8 keyTag) internal pure returns (uint128) {
  if (keyTag >= TOTAL_KEY_TAGS) {
    revert KeyTags_InvalidKeyTag();
  }
  return keyTagsSerialized & ~uint128(1 << keyTag);
}
```

Example for **contains**:

```
function contains(uint128 keyTagsSerialized, uint8 keyTag) internal pure returns (bool) {
  if (keyTag >= TOTAL_KEY_TAGS) {
    revert KeyTags_InvalidKeyTag();
  }
  return keyTagsSerialized & (uint128(1) << keyTag) > 0;
}
```

(Note: **uint128(1)** in **contains** recommendation for explicitness, though **1 << keyTag** promotes **1** correctly).

| INFORMATIONAL-05 | Unreachable code in **hashToG1** function | Pending |
|---|---|---|

### Description

The **hashToG1** function contains a **return G1Point(0, 0);** statement at line 288. This statement is unreachable because it is placed after a **while(true)** loop that can only be exited via an earlier **return G1Point(x, y);** statement within the loop. While modern compilers might optimize out unreachable code, its presence can be confusing to developers reading the code.
Impact: No direct security impact. May slightly increase deployed contract bytecode size if not optimized out by the compiler. Primarily reduces code clarity.
Lines: **BN254.sol**, **288**

### Recommendation

Remove the unreachable **return G1Point(0, 0);** statement at line 288 to improve code clarity and potentially reduce bytecode size.

| INFORMATIONAL-06 | Unused Import **Strings.sol** and **using Strings for string** Directive | Pending |
|---|---|---|

### Description

The contract imports **Strings.sol** from OpenZeppelin contracts (line 4) and includes a **using Strings for string;** directive (line 9). However, no functionalities from the **Strings** library are utilized within the **KeyEcdsaSecp256k1** library.
Impact: This has no direct security impact. It adds a minor amount of clutter to the code and might slightly increase compilation time or deployment size if the compiler doesn't optimize it out completely, though modern compilers are typically good at this for unused imports/code.
Lines: **KeyEcdsaSecp256k1.sol**, **4, 9**

### Recommendation

Remove the unused import statement (**import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";**) and the **using Strings for string;** directive to keep the codebase clean and minimize dependencies if they are not needed.

| INFORMATIONAL-07 | Redundant **serialize** and **toBytes** Functions with Identical Implementation | Pending |
|---|---|---|

### Description

The library defines two functions, **serialize** (lines 32-36) and **toBytes** (lines 44-48), which have identical implementations. Both functions take a **KEY_ECDSA_SECP256K1 memory key** and return **abi.encode(key.value)**. This introduces code duplication.

Impact: Minor code duplication increases the codebase size slightly and can lead to maintenance overhead if changes are needed (though unlikely for such simple functions). It does not pose a direct security risk.

Lines: **KeyEcdsaSecp256k1.sol**, **32-36, 44-48**

### Recommendation

Remove one of the redundant functions and update its usages to call the remaining one. For example, keep **toBytes** and remove **serialize**, or vice-versa, depending on the preferred naming convention within the project. If both names are desired for semantic reasons (e.g., "serialize" for storage/IPC, "toBytes" for general byte conversion), one could call the other to avoid implementation duplication. Example (keeping **toBytes** as the primary):

```
function toBytes(
    KEY_ECDSA_SECP256K1 memory key
) internal view returns (bytes memory keyBytes) {
    keyBytes = abi.encode(key.value);
}

function serialize(
    KEY_ECDSA_SECP256K1 memory key
) internal view returns (bytes memory keySerialized) {
    return toBytes(key); // Call the other function
}
```

| INFORMATIONAL-08 | Unused Import **Strings** | Pending |
|---|---|---|

### Description

The contract imports **Strings** from OpenZeppelin contracts (
**import {Strings} from "@openzeppelin/contracts/utils/Strings.sol";**) and declares **using Strings for string;** (line 11), but **Strings** library functions are not used anywhere in the **KeyBlsBn254.sol** library.

Impact: No direct security impact. Slightly increases contract size and compilation time. Can be confusing for developers maintaining the code.

Lines: **KeyBlsBn254.sol**, **6, 11**

### Recommendation

Remove the import statement for **Strings** and the corresponding **using Strings for string;** declaration if they are not used.

| INFORMATIONAL-09 | Unused constant **powersOfTauMerkleRoot** | Pending |
|---|---|---|

### Description

The internal constant **powersOfTauMerkleRoot** is defined at line 64 but is not referenced or used anywhere within the **BN254** library code provided.
Impact: No direct security impact. Results in a small amount of dead code and may slightly increase the deployed contract's bytecode size. If the constant is intended for future use or for external reference by other contracts, this should be documented.
Lines: **BN254.sol**, **64**

### Recommendation

If the constant **powersOfTauMerkleRoot** is not intended for any current or future use by this library or its consumers, consider removing it to reduce contract bytecode size and improve code clarity. If it serves an external purpose (e.g., for off-chain tools or other contracts to read), add a comment clarifying its intended use.

| INFORMATIONAL-10 | Incorrect Parameter Description in NatSpec | Pending |
|---|---|---|

### Description

The NatSpec documentation for the **self** parameter (line 24) states: "@param self The integer of which to find the modular inverse". This is incorrect; the function **SqrtMod** calculates a modular square root, not a modular inverse.
Impact: Misleading documentation can cause confusion for developers using the function, leading them to misunderstand its purpose or provide incorrect inputs, though the function name **SqrtMod** is fairly clear.
Lines: **SCL_sqrtMod_5mod8.sol**, **24**

### Recommendation

Correct the documentation for the **self** parameter to accurately describe its purpose. Change "@param self The integer of which to find the modular inverse" to "@param self The integer of which to find the modular square root".

| INFORMATIONAL-11 | Commented-Out Redundant Code | Pending |
|---|---|---|

### Description

Line 65 contains a commented-out line: **// result :=addmod(result,0,p)**. The **MODEXP_PRECOMPILE** is expected to return a result **r** such that **0 <= r < M** (where **M** is the modulus **p**). Therefore, this **addmod(result,0,p)** operation would be redundant.
Impact: None, as the code is commented out. If it were active, it would consume a small amount of unnecessary gas.
Lines: **SCL_sqrtMod_5mod8.sol**, **65**

### Recommendation

Remove the commented-out code to improve code clarity and maintainability, as it appears to be correctly identified as unnecessary.

| INFORMATIONAL-12 | Unused Imported Constants | Pending |
|---|---|---|

### Description

The constants **n**, **pMINUS_2**, and **nMINUS_2** are imported from **@crypto-lib/fields/SCL_wei25519.sol** but are not used within the **SqrtMod** function. The function only utilizes **p**, **pp3div8**, and **sqrtm1** from this import.
Impact: Minor increase in potential bytecode size if these constants are not used by any other part of a contract that includes this function. This is generally negligible for library functions but can be optimized.
Lines: **SCL_sqrtMod_5mod8.sol**, **20**

### Recommendation

If this file (or the context where **SqrtMod** is used) does not require **n**, **pMINUS_2**, and **nMINUS_2**, consider removing them from the import statement to keep dependencies minimal and potentially reduce bytecode size. For example, change the import to **import {p, pp3div8, sqrtm1} from "@crypto-lib/fields/SCL_wei25519.sol";**.

| INFORMATIONAL-13 | Unused **extraData** Parameter in **verify** Function | Pending |
|---|---|---|

### Description

The **verify** function declares a parameter **bytes memory extraData** (line 15), but this parameter is not used anywhere within the function's body. The comment **/* extraData */** suggests it might be intentionally unused or a placeholder.
Impact: There is no direct security impact. However, unused parameters can make the code slightly harder to understand, as developers might expect them to have a purpose. It also adds a marginal gas cost for passing an argument that is then ignored if called on-chain from a non-view/pure context.
Lines: **SigEddsaCurve25519.sol**, **11, 15**

### Recommendation

If the **extraData** parameter is genuinely not needed and not part of a required interface signature that this function must match, it should be removed to simplify the code and save a minimal amount of gas. If it is part of a required interface or reserved for future use, consider adding a more explicit comment explaining its status.

| INFORMATIONAL-14 | Misleading Comment Regarding **proof** Structure | Pending |
|---|---|---|

### Description

The comment on line 68:
**// proof is 64 bytes zkProof | 64 bytes commitments | 64 bytes commitmentPok | 32 nonSignersVotingPower** incorrectly describes the layout and size of the **proof** calldata. The actual structure, as inferred from assembly offsets, is 256 bytes for **zkProof**, 64 bytes for **commitments**, 64 bytes for **commitmentPok**, and 32 bytes for **signersVotingPower**.
Impact: This discrepancy can confuse developers reading the code or attempting to construct the **proof** argument externally, potentially leading to integration errors.
Lines: **SigVerifierBlsBn254ZK.sol**, **68**

### Recommendation

Update the comment at line 68 to accurately reflect the **proof** structure: e.g.,
**// proof consists of: zkProof (256 bytes) | commitments (64 bytes) | commitmentPok (64 bytes) | signersVotingPower (32 bytes)**
.

| INFORMATIONAL-15 | Potential Gas Limit Issues in Aggregation Getter Functions | Pending |
|---|---|---|

### Description

Several **view** functions responsible for aggregating voting power or listing entities loop through potentially large sets of data. These include **getOperatorVotingPowerAt** (the overload iterating through vaults), **getOperatorVotingPowersAt**, **getOperatorVotingPowers**, **getTotalVotingPowerAt**, **getTotalVotingPower**, **getVotingPowersAt**, and **getVotingPowers**. If the number of shared vaults, operator-specific vaults, or registered operators becomes very large, these functions could consume significant amounts of gas, potentially exceeding the block gas limit when called.
Impact: If the number of iterated items (vaults, operators) grows excessively large, these view functions may become too costly to execute, effectively becoming unusable for on-chain callers or even off-chain queries if they rely on node execution that respects gas limits. This could hinder the ability to accurately query aggregated voting power or full lists of entities.
Lines: **VaultManagerLogic.sol**, **212-243, 255-274, 292-337, 349-382, 395-408, 414-419, 432-454, 460-475**

### Recommendation

While **view** functions consuming large amounts of gas primarily affect the caller, consider the expected scale of the system. If a very large number of operators or vaults is anticipated, explore alternative patterns for data retrieval, such as:

1. Implementing pagination for list-returning functions.
2. Providing methods to query data for specific subsets rather than entire collections.
3. Off-chain aggregation solutions that can process data incrementally if on-chain full aggregation becomes infeasible. The use of **hints** in underlying **PersistentSet** calls helps optimize individual lookups, but the loops themselves remain a factor for gas consumption.

| INFORMATIONAL-16 | Unused Import of Subnetwork Library | Pending |
|---|---|---|

### Description

The **NetworkManager** contract imports the **Subnetwork** library from **@symbioticfi/core/src/contracts/libraries/Subnetwork.sol** (line 8) but does not appear to use any of its functions or types directly within the **NetworkManager.sol** file. This import might be intended for derived contracts or could be a remnant.
Impact: Slightly increases contract size if the compiler doesn't optimize it out. Adds a minor dependency that isn't actively used by this abstract contract, potentially confusing developers reading the code.
Lines: **NetworkManager.sol**, **8**

### Recommendation

If the **Subnetwork** library is not used by this contract and is not specifically intended to be made available for direct use by immediate derived contracts via this base contract's import, consider removing the import statement. This helps in keeping the codebase clean and reducing unnecessary dependencies.

| INFORMATIONAL-17 | Missing Natspec Documentation | Pending |
|---|---|---|

### Description

The **PermissionManager** contract and its components (modifier **checkPermission** and function **_checkPermission**) lack Natspec comments. Comprehensive documentation is crucial for understanding the contract's purpose, how it's intended to be used, and the responsibilities of derived contracts, especially for an abstract contract designed as a base for permissioning logic.
Impact: Reduced code clarity and maintainability. Developers inheriting from or interacting with this contract might misunderstand its design, intended usage, or the specific requirements for overriding **_checkPermission**, potentially leading to integration errors or insecure implementations in derived contracts.
Lines: **PermissionManager.sol**, **8-17**

### Recommendation

Add detailed Natspec comments to the contract definition, the **checkPermission** modifier, and the **_checkPermission** function. The documentation for **_checkPermission** should clearly state that it's an internal virtual function intended to be overridden by derived contracts to implement specific access control logic, and explain any assumptions or expected behavior (e.g., whether it should revert on failure, its expected gas characteristics, etc.).

| INFORMATIONAL-18 | Inconsistent Handling of Zero-Length Input Array Matching Zero Expected Length | Pending |
|---|---|---|

### Description

When an empty input array **arr** (i.e., **arr.length == 0**) is provided, and the expected **length** is also 0, the **normalize** function unconditionally creates and returns a new zero-length array (**new bytes[](0)** or **new bytes[][](0)**). This differs from the behavior for non-empty arrays where, if **arr.length == length**, the original **arr** is returned. While functionally equivalent in terms of array content (both are empty), this always allocates a new array object in memory for the **arr.length == 0 && length == 0** case.
Impact: Minor gas inefficiency due to an unnecessary memory allocation for the specific case of normalizing an already empty array to an expected length of zero. The caller receives a different memory reference than the input, which might be unexpected if they assumed reference preservation for already-correct inputs, though this is unlikely to cause bugs.
Lines: **InputNormalizer.sol**, **8-10, 18-20**

### Recommendation

For consistency and minor gas optimization, consider modifying the logic to check if **arr.length == length** first. If they match (including the **0 == 0** case), return the original **arr**. If **arr.length != length**, then proceed to check if **arr.length == 0** to allocate a new array, or revert otherwise.
Example modification for the **bytes[]** version:

```
function normalize(bytes[] memory arr, uint256 length) internal pure returns (bytes[] memory) {
   if (arr.length == length) { // Covers arr.length == 0 && length == 0
      return arr;
   }
   // At this point, arr.length != length
   if (arr.length == 0) { // arr is empty, but length is non-zero, so allocate
      return new bytes[](length);
   }
   // arr is non-empty, and arr.length != length, so revert
   revert InvalidLength_InvalidLength();
}
```

Apply a similar change to the **bytes[][]** overload.

| INFORMATIONAL-19 | Redundant Import of BN254 Library | Pending |
|---|---|---|

### Description

The **BN254** library is imported twice from the same path **../../libraries/utils/BN254.sol**. The first import is on line 5, and the second, redundant import is on line 8.
Impact: This is a minor code style issue that increases clutter. It does not affect the functionality or security of the contract, as modern compilers typically handle redundant imports gracefully.
Lines: **SigVerifierBlsBn254Simple.sol**, **5, 8**

### Recommendation

Remove one of the duplicate import statements for the **BN254** library to improve code clarity. For instance, delete line 8.

| INFORMATIONAL-20 | Implicit 32-Byte Length Requirement for message Parameter | Pending |
|---|---|---|

### Description

The **verifyQuorumSig** function accepts a **bytes memory message** parameter, which is subsequently passed to **SigBlsBn254.verify**. According to the context summary for **SigBlsBn254.sol**, its **verify** function decodes its **message** input (also **bytes memory**) into a **bytes32** using **abi.decode(message, (bytes32))**. This operation will revert if the provided **message** is not exactly 32 bytes long. This implicit requirement is not enforced or documented in **SigVerifierBlsBn254Simple.sol**.
Impact: Callers of **SigVerifierBlsBn254Simple.verifyQuorumSig** might be unaware of this strict 32-byte length requirement for the **message**. If a message of a different length is supplied, the transaction will revert deep within the **SigBlsBn254.verify** call, potentially causing confusion or denial of service for interactions expecting to use arbitrary length messages. This is primarily a usability and clarity issue.
Lines: **SigVerifierBlsBn254Simple.sol**, **49, 95**

### Recommendation

To improve clarity and prevent unexpected reverts:

1. If the **message** parameter is always intended to be a 32-byte hash or digest, change its type in **SigVerifierBlsBn254Simple.verifyQuorumSig** from **bytes memory message** to **bytes32 message**. The call to **SigBlsBn254.verify** would then need to pass **abi.encode(message)**.
2. Alternatively, if **SigVerifierBlsBn254Simple** is intended to support arbitrary length messages, it should hash the **message** to a **bytes32** digest (e.g., using **keccak256(message)**) before preparing it for **SigBlsBn254.verify**.
3. At a minimum, clearly document the 32-byte length requirement for the **message** parameter in the NatSpec comments for **verifyQuorumSig**.

| INFORMATIONAL-21 | Timestamp-Based Configuration Change Susceptible to Miner Manipulation | Pending |
|---|---|---|

### Description

The **setEpochDurationInternal** function (line 120) validates that **epochDurationTimestamp >= Time.timestamp()**. **Time.timestamp()** resolves to **block.timestamp**, which can be influenced by miners within a certain range. If a privileged user attempts to set an **epochDurationTimestamp** very close to the current **block.timestamp**, a miner could manipulate the timestamp of the block including the transaction to cause this check to either pass or fail against the user's intent.
Impact: Transactions setting new epoch durations with **epochDurationTimestamp** close to the current time might succeed or fail unpredictably depending on miner actions. This could be an annoyance or minor disruption for administrators. It's unlikely to lead to a direct loss of funds but affects the reliability of time-sensitive configuration updates.
Lines: **EpochManagerLogic.sol**, **120-122**

### Recommendation

When setting epoch configurations, administrators should specify **epochDurationTimestamp** values that are sufficiently far in the future to mitigate the impact of minor **block.timestamp** manipulation by miners (e.g., a few minutes or more). The contract could also enforce a minimum delta between **epochDurationTimestamp** and **Time.timestamp()**, e.g., **require(epochDurationTimestamp >= Time.timestamp() + MIN_DELAY, "Timestamp too soon");**, though this reduces flexibility.

| INFORMATIONAL-22 | Missing Event for EIP-712 Domain Initialization | Pending |
|---|---|---|

### Description

The **__OzEIP712_init** function is responsible for setting up the EIP-712 domain parameters (name and version) by delegating to **OzEIP712Logic.initialize**. This initialization is a significant one-time configuration step. However, neither **OzEIP712.sol** nor the underlying **OzEIP712Logic.initialize** (based on its provided summary) emits an event upon successful initialization of these domain parameters.
Impact: The absence of an event for EIP-712 domain initialization reduces the observability of this critical setup step. Off-chain services, monitoring tools, or auditing processes may find it more difficult to verify and track when and with what parameters the EIP-712 domain was configured for a contract instance.
Lines: **OzEIP712.sol**, **19-23**

### Recommendation

Consider emitting an event when the EIP-712 domain parameters are initialized. This event should ideally be emitted from the **OzEIP712Logic.initialize** function, as that is where the state change for domain parameters occurs. An example event could be: **event EIP712DomainInitialized(string name, string version);** Emitting this event would enhance transparency and facilitate easier off-chain tracking of the contract's EIP-712 configuration.

| INFORMATIONAL-23 | Message Cast to String for Verification | Pending |
|---|---|---|

### Description

The **message** parameter, which is **bytes memory**, is cast to **string** before being passed to **SCL_EIP6565.Verify_LE(string(message), ...)** at line 30. While Solidity strings are UTF-8 encoded byte sequences and can often be used interchangeably with **bytes** for raw data, this explicit cast might be unnecessary. If **SCL_EIP6565.Verify_LE** (or any underlying mechanism it uses) performs specific string manipulations or UTF-8 validation that are not intended for arbitrary byte messages used in signatures, it could lead to unexpected behavior or slight gas overhead.
Impact: Likely low impact. Primarily, it could lead to misinterpretation if **SCL_EIP6565.Verify_LE** has string-specific logic not suitable for raw cryptographic message bytes. Otherwise, it might be a minor gas inefficiency or simply a matter of code clarity. EdDSA standards typically operate on sequences of octets (bytes).
Lines: **SigEddsaCurve25519.sol**, **30**

### Recommendation

Verify that **SCL_EIP6565.Verify_LE** correctly interprets **string(message)** as a raw sequence of bytes, equivalent to passing **bytes memory message**. If the library supports **bytes memory** directly for the message parameter, prefer that to avoid potential ambiguity, ensure byte-perfect processing, and potentially save a minor amount of gas associated with the cast or string handling.

| INFORMATIONAL-24 | setKey64 Function is Unused by the Primary setKey Logic | Pending |
|---|---|---|

### Description

The library includes a **setKey64** function (lines 246-251) designed for handling 64-byte compressed keys. However, the main **setKey(address operator, uint8 keyTag, bytes memory key)** function (lines 228-239), which routes key registration to specific storage functions based on key type, currently only calls **setKey32** for the defined **KEY_TYPE_BLS_BN254** and **KEY_TYPE_ECDSA_SECP256K1**. There is no existing key type or logic within this routing function that would utilize **setKey64**.
Impact: The **setKey64** function is effectively dead code within the primary key registration workflow as currently implemented. While it can be called directly (though this is unsafe due to its **public** visibility, as noted in KML-001), its intended integration through the standard **setKey** mechanism for a 64-byte key type is missing. This does not pose a direct security risk in itself but might indicate an incomplete feature or an orphaned function.
Lines: **KeyManagerLogic.sol**, **246-251, 228-239**

### Recommendation

If 64-byte keys are intended to be supported in the future, update the **setKey(address operator, uint8 keyTag, bytes memory key)** function to include logic for new key types that serialize to 64 bytes and correctly call **setKey64**. The visibility of **setKey64** should also be changed to **internal** (see KML-001). If **setKey64** is deprecated or not planned for use, consider removing it to reduce contract size and complexity, or clearly document its status and intended future use.

| INFORMATIONAL-25 | Missing Event Emission for Operator and Vault Registration/Unregistration | Pending |
|---|---|---|

### Description

The functions **registerOperator**, **unregisterOperator**, **registerOperatorVault**, **unregisterOperatorVault** (and their **WithSignature** counterparts) modify operator and operator-vault registration states by calling internal functions inherited from **VaultManager**. Based on the provided summaries for **OperatorManagerLogic.sol** and **VaultManagerLogic.sol** (which handle the ultimate state changes), these underlying logic layers do not appear to emit specific events for these registration/unregistration actions. The **SelfRegisterOperators.sol** contract itself also does not explicitly define or emit events for these actions. If the **ISelfRegisterOperators** interface mandates such events, this would be a compliance issue. Even if not mandated, missing events reduce the observability of these significant state changes for off-chain systems and UI components.
Impact: Makes it more difficult for off-chain services and user interfaces to track operator and vault registrations/unregistrations in real-time. This can hinder monitoring, indexing, and auditing capabilities. If the **ISelfRegisterOperators** interface (not provided in full) requires specific events for these actions, the contract would not be fully compliant.
Lines: **SelfRegisterOperators.sol**, **40-44, 49-54, 59-61, 66-71, 76-80, 85-96, 101-105, 110-121**

### Recommendation

Define and emit appropriate events (e.g., **OperatorRegistered(address indexed operator)**, **OperatorUnregistered(address indexed operator)**, **OperatorVaultRegistered(address indexed operator, address indexed vault)**, **OperatorVaultUnregistered(address indexed operator, address indexed vault)**) in the respective functions within **SelfRegisterOperators.sol**. Alternatively, ensure these events are emitted by the underlying logic layers ( **OperatorManagerLogic**, **VaultManagerLogic**) and that **SelfRegisterOperators.sol** adheres to any event specifications in the **ISelfRegisterOperators** interface. For example, after a successful call to **_registerOperatorImpl(operator, vault)**, appropriate events for operator registration and, if applicable, vault registration for that operator should be emitted.

| INFORMATIONAL-26 | Unused using Strings for ... Directives and Import | Pending |
|---|---|---|

### Description

The **SigBlsBn254** library imports **Strings** from OpenZeppelin contracts (line 7) and includes **using Strings for bytes;** (line 12) and **using Strings for string;** (line 13). However, no methods from the **Strings** library are actually invoked on **bytes** or **string** types within the **SigBlsBn254.verify** function or any other part of this library.
Impact: This results in minor code clutter and a slightly larger compiled contract size if the compiler and linker do not optimize out the unused import and associated library code completely. It has no direct security or functional impact on the **SigBlsBn254** library itself.
Lines: **SigBlsBn254.sol**, **7, 12, 13**

### Recommendation

To improve code clarity and potentially reduce the deployed bytecode size, remove the unused import statement for **Strings.sol** (line 7) and the associated **using Strings for bytes;** (line 12) and **using Strings for string;** (line 13) directives.

| | | |
|---|---|---|
| INFORMATIONAL-27 | Redundant require Check After Precompile Call Failure Handling | Pending |

### Description

In functions **plus** (line 100), **scalar_mul** (line 164), and **expMod** (line 327), a **require(success, "error message")** check is performed immediately after an assembly block that calls a precompile. The assembly block itself checks the **success** flag returned by the **staticcall** and executes the **invalid()** opcode if **success** is false. The **invalid()** opcode halts execution and reverts state changes, meaning the subsequent **require** statement will only ever be reached if **success** is true. In this scenario, the **require(true, ...)** check will always pass, making it redundant.
Impact: Minor gas cost for an effectively no-op **require** check. Slightly reduces code clarity by having a check that cannot fail when reached.
Lines: **BN254.sol**, **100, 164, 327**

### Recommendation

Remove the redundant **require(success, ...)** statements after the assembly blocks. The **invalid()** opcode in the assembly block sufficiently ensures that execution does not proceed if the precompile call fails.

| | | |
|---|---|---|
| INFORMATIONAL-28 | Redundant Serialization Functions serialize and toBytes | Pending |

### Description

The library defines two functions, **serialize** (lines 46-50) and **toBytes** (lines 58-62), which have identical implementations: **keySerialized = abi.encode(key.value)** and **keyBytes = abi.encode(key.value)**. This introduces code duplication.
Impact: Minor code duplication increases the codebase size slightly and might lead to minor maintenance overhead if changes were ever needed for this logic (though unlikely for such simple functions). It does not pose a direct security risk.
Lines: **KeyEddsaCurve25519.sol**, **46-50, 58-62**

### Recommendation

To reduce duplication, remove one of the functions and update any internal usages to call the remaining one. For instance, keep **toBytes** and remove **serialize**, or vice-versa, based on the preferred naming convention. Alternatively, if both names are desired for semantic clarity, one function can simply call the other to consolidate the implementation logic (e.g., **function serialize(...) internal view returns (bytes memory) { return toBytes(...); }**).

| | | |
|---|---|---|
| INFORMATIONAL-29 | Unused using for Directives and Associated Imports | Pending |

### Description

The **SigVerifierBlsBn254ZK** contract imports **SigBlsBn254.sol** (line 4) and **KeyBlsBn254.sol** (line 6). It also includes **using for** directives for types from these libraries at lines 15, 16, 17, and 18 (e.g., **using KeyBlsBn254 for bytes;**, **using KeyBlsBn254 for KeyBlsBn254.KEY_BLS_BN254;**, **using KeyBlsBn254 for BN254.G1Point;**, **using SigBlsBn254 for bytes;**). However, the functionalities provided by **SigBlsBn254** and **KeyBlsBn254** through these directives are not utilized within the **SigVerifierBlsBn254ZK.sol** contract's code.
Impact: The unused imports and **using for** directives do not affect the runtime behavior or security of the contract. They represent dead code, which can slightly increase the deployed contract size (though modern compilers might optimize some of this away) and add minor clutter, potentially confusing developers who are reading or maintaining the code.
Lines: **SigVerifierBlsBn254ZK.sol**, **4, 6, 15, 16, 17, 18**

### Recommendation

To improve code clarity, reduce potential bytecode size, and minimize dependencies, remove the import statements for **SigBlsBn254.sol** and **KeyBlsBn254.sol**, as well as the corresponding **using KeyBlsBn254 for ...;** and **using SigBlsBn254 for ...;** directives if they are not intended for current or future use in this specific contract or are not essential for contracts inheriting from it.

| INFORMATIONAL-30 | Token Registration and Unregistration Functions Lack Event Emission | Pending |
|---|---|---|

### Description

The **registerToken** and **unregisterToken** functions in **Tokens.sol**, and their underlying implementations in **VaultManager.sol** and **VaultManagerLogic.sol** (where the state change occurs), do not emit events upon the successful registration or unregistration of a token. Events are essential for off-chain monitoring, indexing services, and providing a transparent audit trail of significant state changes within the contract.
Impact: The absence of events makes it more difficult for off-chain services and users to track changes to the set of registered tokens efficiently. This reduces the observability of these critical operations and can complicate debugging or historical analysis.
Lines: **Tokens.sol**, **20-24, 29-33**

### Recommendation

It is recommended to emit events in the **VaultManagerLogic.registerToken** and **VaultManagerLogic.unregisterToken** functions, as these are where the actual state modification for token registration occurs. For example:
Define events:

```
// In IVaultManager or a relevant interface/contract
event TokenRegistered(address indexed token, uint48 timestamp);
event TokenUnregistered(address indexed token, uint48 timestamp);
```

Emit events in **VaultManagerLogic.sol**: In **registerToken**:

```
emit TokenRegistered(token, Time.timestamp()); // Assuming Time.timestamp() is the key used
```

In **unregisterToken**:

```
emit TokenUnregistered(token, Time.timestamp()); // Assuming Time.timestamp() is the key used
```

| INFORMATIONAL-31 | Public Function _getOperatorStakeAt Uses Internal Naming Convention | Pending |
|---|---|---|

### Description

The function **_getOperatorStakeAt** at lines 305-312 is declared **public** but uses an underscore prefix, which is conventionally reserved for **internal** or **private** functions in Solidity. This naming can be misleading to developers interacting with the contract's ABI or reading the source code, as it might incorrectly suggest that the function is not part of the stable public API or has restricted access.
Impact: This deviation from common Solidity naming conventions can reduce code clarity and potentially lead to confusion for developers regarding the function's intended visibility and usage. It does not directly cause a loss of funds or incorrect contract behavior but affects maintainability and understandability.
Lines: **VaultManager.sol**, **305-312**

### Recommendation

If the function **_getOperatorStakeAt** is intended to be part of the contract's public interface, it should be renamed to **getOperatorStakeAt** (i.e., remove the leading underscore) to align with standard naming conventions. If it is primarily intended for use by derived contracts and not as a direct public endpoint, its visibility should be changed to **internal**.

| INFORMATIONAL-32 | Implicit Assumption on **message** Length for **abi.decode** | Pending |
|---|---|---|

### Description

In **verifyQuorumSig**, the **message** argument (type **bytes memory**) is decoded into a **bytes32** using **abi.decode(message, (bytes32))**. This operation will revert if **message.length** is not exactly 32 bytes. This implies that **message** is expected to be a 32-byte hash or value.
Impact: If **message** is passed with a length other than 32 bytes, the function will revert. This behavior might be intended but is not explicitly checked or documented, potentially surprising users of the interface if they expect support for arbitrary length messages.
Lines: **SigVerifierBlsBn254ZK.sol**, **98**

### Recommendation

If **message** must always be 32 bytes, add an explicit check like **require(message.length == 32, "SigVerifierBlsBn254ZK: Message must be 32 bytes");** for improved clarity and a more specific error message. Alternatively, if arbitrary length messages are desired, **message** should be hashed to **bytes32** (e.g., using **keccak256(message)**) before being used in **BN254.hashToG1**. Document the expected format of **message**.

| INFORMATIONAL-33 | Potential Gas Limit Issues for Retrieving All Operators | Pending |
|---|---|---|

### Description

The **getOperators()** function retrieves all currently registered operators by calling **_operators.values()** from the **PersistentSet** library. Similarly, **getOperatorsAt(timestamp, hints)** calls **_operators.valuesAt(timestamp, hints)**. If the number of registered operators becomes very large, the gas cost to construct and return the array of all operator addresses can become very high, potentially exceeding the block gas limit or becoming prohibitively expensive for callers, especially if **hints** are not used or are suboptimal for **getOperatorsAt**.
Impact: Callers (including other smart contracts) may be unable to retrieve the complete list of operators if the set is too large, leading to a denial of service for functionalities that rely on iterating over all operators on-chain within a single transaction's gas limit.
Lines: **OperatorManagerLogic.sol**, **43-49**

### Recommendation

For on-chain use cases requiring iteration over large sets of operators, consider alternative patterns such as allowing iteration in smaller, paginated chunks, or having consumers track operator registrations via events emitted by **registerOperator** and **unregisterOperator**. If on-chain retrieval of the full set is a requirement, document the potential gas limitations. The **getOperatorsLength()** function can be used by callers to estimate the potential cost or size before attempting to retrieve the full list. For **getOperatorsAt**, emphasize the importance of providing correct **hints** to manage gas consumption effectively.

| INFORMATIONAL-34 | Magic Numbers Used for Type and Tag Validation | Pending |
|---|---|---|

## Description

The **getKeyTag** function uses literal values **7** and **15** to validate the **type_** and **tag** parameters, respectively (lines 15 and 18). These values are derived from the bit allocation (3 bits for type, 4 bits for tag) described in the comment on line 11. While the current values are correct, using named constants for these maximum values would improve code readability, maintainability, and make the relationship to the bit allocation explicit in the code.
Impact: Reduced code clarity. If the bit allocation (3 bits for type, 4 bits for tag) were to change, these magic numbers would need to be identified and updated manually, increasing the risk of inconsistency if not all occurrences are found or if the derivation logic is not immediately obvious to a maintainer.
Lines: **KeyTags.sol**, **15, 18**

## Recommendation

Define and use named constants for the maximum type and tag values. For example:

```
// Add these constants to the library
uint8 private constant MAX_KEY_TYPE_VALUE = (1 << 3) - 1; // For 3 bits
uint8 private constant MAX_KEY_TAG_VALUE = (1 << 4) - 1;  // For 4 bits

// In getKeyTag function:
if (type_ > MAX_KEY_TYPE_VALUE) {
   revert KeyTags_InvalidKeyType();
}
if (tag > MAX_KEY_TAG_VALUE) {
   revert KeyTags_InvalidKeyTag();
}
```

This makes the code more self-documenting and easier to maintain if the bit allocations change.

| INFORMATIONAL-35 | Unused Import **Strings** and **using Strings for string** Directive | Pending |
|---|---|---|

## Description

The **KeyEddsaCurve25519** library imports **Strings** from **@openzeppelin/contracts/utils/Strings.sol** (line 4) and declares **using Strings for string;** (line 14). However, no functionalities from the **Strings** library are utilized anywhere within this specific library file.
Impact: This has no direct security impact. It adds minor clutter to the code and might slightly increase compilation time or deployment bytecode size if the compiler and linker do not fully optimize out the unused import and directive. Primarily, it affects code readability and maintainability.
Lines: **KeyEddsaCurve25519.sol**, **4, 14**

## Recommendation

Remove the unused import statement for **Strings** (line 4) and the **using Strings for string;** directive (line 14) to keep the codebase clean, minimize dependencies, and improve clarity.

| INFORMATIONAL-36 | Misleading or Unused **using KeyTags for uint8;** Directive | Pending |
|---|---|---|

### Description

The **KeyManager.sol** contract includes the directive **using KeyTags for uint8;** at line 21. The **KeyTags** library, as per its provided summary, contains **internal pure** utility functions for manipulating key tags and serializing/deserializing them. These functions (e.g., **KeyTags.combine(type, tag)**, **KeyTags.getType(keyTag)**) are designed to be called directly (e.g., **KeyTags.functionName(...)**) and do not take **uint8** as their first **self** parameter in a way that is compatible with the **using for uint8** syntax. For example, one would call **KeyTags.getType(myUint8Tag)** not **myUint8Tag.getType()**. Therefore, this **using** directive is likely unused or represents a misunderstanding of how to use the **KeyTags** library with the **using for** feature.
Impact: This directive, if unused, has no direct security impact or effect on runtime behavior. However, it adds dead code, which can slightly increase deployed bytecode size (if not optimized out by the compiler) and potentially confuse developers reading or maintaining the code.
Lines: **KeyManager.sol**, **21**

### Recommendation

Review the usage of the **KeyTags** library within **KeyManager.sol** and its derived contracts. If the directive **using KeyTags for uint8;** is indeed unused or incorrectly applied, it should be removed to improve code clarity and reduce clutter. Calls to **KeyTags** library functions should be made directly, such as **KeyTags.getType(tagValue)**.

| INFORMATIONAL-37 | Unused **using Checkpoints for Checkpoints.Trace208;** Directive | Pending |
|---|---|---|

### Description

The **EpochManager** contract declares **using Checkpoints for Checkpoints.Trace208;** at line 11. However, no methods from the **Checkpoints** library (e.g., **trace.push()**, **trace.latest()**) are called on **Checkpoints.Trace208** typed variables using this directive directly within the **EpochManager.sol** file. The only function that uses **Checkpoints.Trace208** is **_getCurrentValue**, where it appears as a storage pointer type in the function's parameter, which is then passed directly to **EpochManagerLogic.getCurrentValue**.
Impact: This has no direct effect on runtime behavior or security for **EpochManager.sol** itself. It constitutes minor code clutter and might slightly increase the deployed contract size if not optimized away by the compiler. It could also be misleading to developers reading the code who might expect direct usage of the library's methods on **Checkpoints.Trace208** instances within this contract.
Lines: **EpochManager.sol**, **11**

### Recommendation

If the **using Checkpoints for Checkpoints.Trace208;** directive is not intended for use by derived contracts that might override **_getCurrentValue** (or other future virtual functions) and then use **Checkpoints** methods on **trace** variables, it should be removed to improve code clarity and reduce unnecessary imports. If it is specifically retained for potential use in derived contracts, consider adding a comment to clarify this intent.

| INFORMATIONAL-38 | Missing Events for Force Pause/Unpause Operations | Pending |
|---|---|---|

### Description

The functions **forcePauseOperator**, **forceUnpauseOperator**, **forcePauseOperatorVault**, and **forceUnpauseOperatorVault** modify critical operational states by pausing or unpausing operators and their vaults. However, these functions do not emit events to signal these state changes.

Impact: The absence of events makes it more challenging for off-chain services, monitoring tools, and user interfaces to track these important administrative actions. This can hinder transparency, delay reactions to state changes, and complicate debugging or auditing efforts.

Lines: **ForcePauseSelfRegisterOperators.sol**, **47-57, 62-69, 74-82, 87-92**

### Recommendation

Emit events in each of the force-pause and force-unpause functions. These events should clearly indicate the action taken (pause/unpause), the operator and vault (if applicable) affected, and potentially the address that initiated the action (e.g., **msg.sender**).

Example events:

```
event OperatorForcePaused(address indexed operator, address indexed pauser);
event OperatorForceUnpaused(address indexed operator, address indexed pauser);
event OperatorVaultForcePaused(address indexed operator, address indexed vault, address indexed pauser);
event OperatorVaultForceUnpaused(address indexed operator, address indexed vault, address indexed pauser);
```

These events should then be emitted in the respective functions.

| INFORMATIONAL-39 | Potentially High Gas Costs for Key Operations | Pending |
|---|---|---|

### Description

Functions like **wrap**, **decompress**, and **fromBytes** involve multiple complex cryptographic operations such as modular exponentiation (implicitly in **SqrtMod** and **ModInv**), modular multiplications, and SHA-512 hashing (**SCL_sha512.Swap256**). These operations are inherently gas-intensive on the Ethereum Virtual Machine.

Impact: If these functions are frequently called within transactions, especially with user-provided inputs or in loops, they could lead to high gas costs for users or cause transactions to approach or exceed the block gas limit, potentially resulting in denial-of-service in some scenarios. This is particularly relevant if the library is used in smart contracts that handle a large number of key registrations or verifications on-chain.

Lines: **KeyEddsaCurve25519.sol**, **23-38, 64-72, 82-92**

### Recommendation

1. Analyze the expected usage patterns of this library within the broader system. If on-chain calls to these functions are frequent or involve batch operations, carefully benchmark and consider gas implications.
2. Provide clear gas cost warnings and benchmarks in the documentation for functions involving these cryptographic operations.
3. Callers of these functions should be aware of potential gas implications, especially if inputs can be controlled by external users or if functions are called in batch.

| INFORMATIONAL-40 | Superfluous **using for** Directives for Delegated Logic | Pending |
|---|---|---|

## Description

The **KeyManager.sol** contract declares several **using for** directives (e.g., **using Checkpoints for Checkpoints.Trace...;**, **using KeyBlsBn254 for KeyBlsBn254.KEY_BLS_BN254;**, **using PersistentSet for PersistentSet.AddressSet;**, etc.) on lines 21-29. Since **KeyManager.sol** is an abstract contract that primarily delegates its core logic to the **KeyManagerLogic** library, **KeyManager.sol** itself does not appear to directly invoke methods on these types using the **using for** syntax (e.g., **myTrace.push(...)**). These directives are relevant and likely used within **KeyManagerLogic.sol**. Their presence in **KeyManager.sol** might be intended for convenience if derived contracts were to directly use these library methods, but if derived contracts also do not use them directly in this manner, the directives in **KeyManager.sol** add unnecessary clutter.
Impact: This primarily affects code clarity and maintainability. It can lead to minor increases in contract bytecode size if not optimized out by the compiler. It does not pose a direct security risk.
Lines: **KeyManager.sol**, **21-29**

## Recommendation

Review whether these **using for** directives are actively utilized by contracts deriving from **KeyManager.sol** or if they are remnants. If they are not used in **KeyManager.sol** or its typical derived contract patterns, consider removing them from **KeyManager.sol** to keep its code focused on its facade and abstract responsibilities. The necessary **using for** directives should be maintained in **KeyManagerLogic.sol** where the actual implementation resides and utilizes them.

STATE
MIND