Aztec Rollup Contracts

Security Review

Cantina Managed review by:

Xmxanuel, Lead Security Researcher
Chinmay Farkya, Associate Security Researcher

November 12, 2025

Contents

1 Introduction **2**
1.1 About Cantina . . . 2
1.2 Disclaimer . . . 2
1.3 Risk assessment . . . 2
1.3.1 Severity Classification . . . 2

2 Security Review Summary **3**

3 Findings **4**
3.1 Critical Risk . . . 4
3.1.1 `Outbox` double-consume (replay) via unbounded `leafIndex` in `MerkleLib.verifyMembership` . . . 4
3.2 High Risk . . . 5
3.2.1 Bootstrap minimum rollup validators bypass via refundable invalid deposits . . . 5
3.2.2 `Inbox.consume` deadlock when enabling transactions after ignition phase . . . 6
3.3 Medium Risk . . . 7
3.3.1 `Propose` griefing attack via mutated attestations signatures . . . 7
3.3.2 `MerkleLib.computeRoot` inflates array length without padding (reads adjacent memory) 8
3.3.3 `Validator` sampling from past state can result in inactive committee (liveness risk) . . 9
3.3.4 Attacker can fill Inbox with spam messages at low cost . . . 9
3.3.5 User funds could get stuck in `FeeJuicePortal` if block production halts on L2 . . . 10
3.3.6 Validators' deposits can get stuck in Staking queue . . . 11
3.4 Low Risk . . . 12
3.4.1 `getBlobCommitmentsHash` returns incorrect data for future block numbers . . . 12
3.4.2 `claimProverRewards`: batch reward claim reverts if any epoch already claimed (griefing attack) . . . 12
3.4.3 Reward calculations can be inconsistent for an epoch if sequencerBps or `blockReward` is modified . . . 13
3.4.4 `deposit` front�run using user's `BLS` keys temporarily locks user funds . . . 13
3.4.5 Staking queue can be spammed with invalid requests . . . 14
3.4.6 Public `checkpointRandao` allows opportunistic seed timing (first-caller-wins) . . . 14
3.5 Gas Optimization . . . 15
3.5.1 `computeSampleIndex` early return if index <= 1 . . . 15
3.6 Informational . . . 15
3.6.1 Test-only VM cheatcode path in `BlobLib` . . . 15
3.6.2 `ChainTips.updateProvenBlockNumber` can corrupt pending half for > `uint128` inputs . 15
3.6.3 Add a sanity check for the shift operation in `Outbox.consume` to avoid overflow . . . 16
3.6.4 `_invalidIndex` lacks bounds check in `invalidateBadAttestation` . . . 16
3.6.5 Prevent `Outbox.insert` to be called for proven `_l2BlockNumber` . . . 16
3.6.6 General Code Improvements . . . 17
3.6.7 `_isFirstBlockOfEpoch==true` implicitly requires at least one blob in `BlobLib.calculateBlobCommitmentsHash` . . . 17

1 Introduction

1.1 About Cantina

Cantina is a security services marketplace that connects top security researchers and solutions with clients. Learn more at cantina.xyz

1.2 Disclaimer

Cantina Managed provides a detailed evaluation of the security posture of the code at a particular moment based on the information available at the time of the review. While Cantina Managed endeavors to identify and disclose all potential security issues, it cannot guarantee that every vulnerability will be detected or that the code will be entirely secure against all possible attacks. The assessment is conducted based on the specific commit and version of the code provided. Any subsequent modifications to the code may introduce new vulnerabilities that were absent during the initial review. Therefore, any changes made to the code require a new security review to ensure that the code remains secure. Please be advised that the Cantina Managed security review is not a replacement for continuous security measures such as penetration testing, vulnerability scanning, and regular code reviews.

1.3 Risk assessment

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

1.3.1 Severity Classification

The severity of security issues found during the security review is categorized based on the above table. Critical findings have a high likelihood of being exploited and must be addressed immediately. High findings are almost certain to occur, easy to perform, or not easy but highly incentivized thus must be fixed as soon as possible.

Medium findings are conditionally possible or incentivized but are still relatively likely to occur and should be addressed. Low findings are a rare combination of circumstances to exploit, or offer little to no incentive to exploit but are recommended to be addressed.

Lastly, some findings might represent objective improvements that should be addressed but do not impact the project's overall security (Gas and Informational findings).

2 Security Review Summary

Aztec Labs was founded in 2017, and has a team of +50 leading zero-knowledge cryptographers, engineers, and business experts. Aztec Labs is developing its namesake products: AZTEC (a privacy-first L2 on Ethereum) and NOIR (the universal ZK language).

From Aug 14th to Sep 11th the Cantina team conducted a review of aztec-packages on commit hash 270e2a58. The review focused on the following scope:

- Commit hash 270e2a58ae2a378472f9be791900bbd4342c7881.
- The `EmpireSlashingProposer` PR (PR 16357).

The team identified a total of **23** issues:

Issues Found

Severity	**Count**	**Fixed**	**Acknowledged**
Critical Risk	1	1	0
High Risk	2	1	1
Medium Risk	6	3	3
Low Risk	6	3	3
Gas Optimizations	1	1	0
Informational	7	6	1
Total	**23**	**15**	**8**

3 Findings

3.1 Critical Risk

3.1.1 `Outbox` double-consume (replay) via unbounded `leafIndex` in `MerkleLib.verifyMembership`

Severity: Critical Risk

Context: Outbox.sol#L95

Description: `MerkleLib.verifyMembership` uses the leaf `index` bits to decide left/right concatenation for the hash function at each tree level. The tree height defines how many bits are consumed from the `index` parameter, but there is no input validation that the `index` fits within the tree size (depth). Only the lower `d = _path.length` bits are used, higher bits are ignored. As long as the lower bits match the actual index, higher bits can have any value and verification still succeeds.

```
function verifyMembership(bytes32[] calldata _path, bytes32 _leaf, uint256 _index, bytes32 _expectedRoot)
  internal
  pure
{
  bytes32 subtreeRoot = _leaf;
  /// @notice - We use the indexAtHeight to see whether our child of the next subtree is at the left or the right
  ↪  side
  uint256 indexAtHeight = _index;

  for (uint256 height = 0; height < _path.length; height++) {
    /// @notice - This affects the way we concatenate our two children to then hash and calculate the root, as
    ↪  any odd
    /// indexes (index bit-masked with least significant bit) are right-sided children.
    bool isRight = (indexAtHeight & 1) == 1;

    subtreeRoot = isRight
      ? Hash.sha256ToField(bytes.concat(_path[height], subtreeRoot))
      : Hash.sha256ToField(bytes.concat(subtreeRoot, _path[height]));
    /// @notice - We divide by two here to get the index of the parent of the current subtreeRoot in its own
    ↪  layer
    indexAtHeight >>= 1;
  }

  require(subtreeRoot == _expectedRoot, Errors.MerkleLib__InvalidRoot(_expectedRoot, subtreeRoot, _leaf,
  ↪  _index));
}
```

Because only the lowest `d` bits are read, there are `2^(256 - d)` valid aliases for a given leaf which will return `true` for the `verifyMembership` function. In `Outbox.consume`, the unique nullifier key used to check if a leaf has been consumed is calculated as `leafId = (1 << _path.length) + _leafIndex`. The actual merkle tree proof is based on a balanced subtree and the `leafId` is the unique global tree index. Because `_leafIndex` is not bounded/masked, an attacker can re-consume the same message multiple times by changing the high bits of `_leafIndex`, generating a fresh `leafId` each time.

Impact: Enables repeated consumption of the same L2→L1 message (double-spend), draining token portals and re-executing privileged actions. Direct loss of funds across any consumer of `Outbox.consume`.

Concrete example (depth d = `path.length`): given a valid proof for index 5 (`0b101`), calling with index 13 (`0b1011` = 5 + 8) also verifies. Both produce the same root but different `leafIds`, bypassing the consumed check.

Proof of Concept:

```
function testVerifyMembership(uint256 _idx) public view {
  uint256 leafIndex = bound(_idx, 0, merkle.SIZE() - 1);

  (bytes32[] memory path, bytes32 leaf) = merkle.computeSiblingPath(leafIndex);

  bytes32 expectedRoot = merkle.computeRoot();

  merkleLibHelper.verifyMembership(path, leaf, leafIndex, expectedRoot);

  // Extension = 2^depth
  uint256 depth = path.length;
  uint256 extendedIndex = leafIndex + (1 << depth); // e.g. 5 + 8 = 13
  // Extended index ALSO works (vulnerability which enables double spend in the Outbox.consume)
  merkleLibHelper.verifyMembership(path, leaf, extendedIndex, expectedRoot);

}
```

Recommendation: Add an explicit bound in the Merkle verification:

```
  function verifyMembership(bytes32[] calldata _path, bytes32 _leaf, uint256 _index, bytes32 _expectedRoot)
    internal
    pure
  {
    bytes32 subtreeRoot = _leaf;
    uint256 indexAtHeight = _index;
    for (uint256 height = 0; height < _path.length; height++) {
      bool isRight = (indexAtHeight & 1) == 1;
      subtreeRoot = isRight
        ? Hash.sha256ToField(bytes.concat(_path[height], subtreeRoot))
        : Hash.sha256ToField(bytes.concat(subtreeRoot, _path[height]));
      indexAtHeight >>= 1;
    }
+   // Ensure no high bits remain beyond the provided path depth
+   require(indexAtHeight == 0, MerkleLib__IndexExceedsPathDepth(uint256 index, uint256 depth));
    require(subtreeRoot == _expectedRoot, Errors.MerkleLib__InvalidRoot(_expectedRoot, subtreeRoot, _leaf,
    ↪  _index));
  }
```

Also add a bound in `Outbox.consume` (defense-in-depth) so the nullifier space matches the proof domain:

```
  function consume(
    DataStructures.L2ToL1Msg calldata _message,
    uint256 _l2BlockNumber,
    uint256 _leafIndex,
    bytes32[] calldata _path
  ) external {
    // ...
+   require(_leafIndex < (1 << _path.length), Outbox__InvalidLeafIndex(uint256 index, uint256 pathLen));
    uint256 leafId = (1 << _path.length) + _leafIndex;
    require(!rootData.nullified.get(leafId), Errors.Outbox__AlreadyNullified(_l2BlockNumber, leafId));
    // ...
  }
```

Aztec Labs: The `verifyMembership` was fixed as part of PR 16585, but the extra check in the consume was added in PR 17299

Cantina Managed: Fixed.

3.2 High Risk

3.2.1 Bootstrap minimum rollup validators bypass via refundable invalid deposits

Severity: High Risk

Context: StakingLib.sol#L316

Description: The validator bootstrap phase intends to require an initial cohort of `bootstrapValidatorSetSize` before allowing any activations. The current logic is based on the bootstrap queue length (not on successful validator deposits) and refunds failed deposits in-place.

This failed deposit refund feature can be exploited to skip the intended `bootstrapValidatorSetSize`.

The function `flushEntryQueue` moves validators from the queue into active staking. The `getEntryQueueFlushSize()` returns `0` while `activeAttesterCount == 0 && queueSize < bootstrapValidatorSetSize`. Therefore, the `bootstrapValidatorSetSize` needs to be reached before the first validators can start staking.

A high `bootstrapValidatorSetSize` prevents a malicious set from starting the rollup. Entering the queue already requires depositing the `ACTIVATION_THRESHOLD`.

Exploit (single transaction, flash‑loan friendly):

- Borrow `missing * ACTIVATION_THRESHOLD` tokens.
- Push `missing` deposits with invalid parameters to inflate the queue to `bootstrapValidatorSetSize`.

- Call `flushEntryQueue()`. The "bootstrap" gate opens due to queue size, valid ones (if any) activate; invalid ones revert and are refunded.
- Repay loan with refunds. The rollup now has a small active set (e.g., 5) instead of the required bootstrap minimum (e.g., 50), yet subsequent epochs are in "growth" phase, allowing further flushes without meeting the original threshold.

Since multiple DeFi protocols offer flash loans with no additional fees, the described attack can be executed with minimal or no economic cost, assuming sufficient staking‑token liquidity is available on secondary markets.

Impact:

- Starts the chain with an under‑sized validator set, defeating the intended bootstrap decentralization property.
- Moves the system into "growth" phase after the first small cohort activates, bypassing the initial cohort size requirement for all future epochs.
- The network can start with far fewer than the intended minimum validators, weakening decentralization and increasing capture risk. The attack is permissionless and cheap once a few valid deposits are present.

Recommendation: Enforce the bootstrap requirement on successful deposits, not just queue length. During the first activation (when `activeAttesterCount == 0`), require the same flush to activate at least `bootstrapValidatorSetSize` validators; otherwise, revert the flush.

Additionally, to prevent flash‑loan attacks, store the `block.timestamp` when a validator joins the queue and don't allow them to be processed in `flushEntryQueue` when the `block.timestamp` is the same.

Aztec Labs: Acknowledged. This is not an issue during the initial bootstrap since token liquidity is very limited. However, it is also not possible to flush the full bootstrap set in a single transaction, because the configured `bootstrapFlushSize` will be smaller than the `bootstrapValidatorSetSize`. As a result, only the `bootstrapFlushSize` can be flushed per transaction per epoch, not the entire `bootstrapValidatorSetSize`.

Cantina Managed: Acknowledged.

3.2.2 `Inbox.consume` deadlock when enabling transactions after ignition phase

Severity: High Risk

Context: ProposeLib.sol#L292

Description: During ignition phase (when `FeeLib.getManaTarget() == 0`), `ProposeLib.propose` skips Inbox/Outbox handling:

- `v.isTxsEnabled = FeeLib.isTxsEnabled()` $\rightarrow$ false.
- Inbox `consume(blockNumber)` is not called, so `Inbox.state.inProgress` never advances beyond its constructor value (`INITIAL_L2_BLOCK_NUM + 1`).

When transactions are later enabled (manaTarget > 0), `propose` starts calling:

```
v.inHash = rollupStore.config.inbox.consume(blockNumber);
```

But `Inbox.consume` requires `_toConsume < inProgress`:

```
uint64 inProgress = state.inProgress;
require(_toConsume < inProgress, Errors.Inbox__MustBuildBeforeConsume());
```

If ignition produced N blocks without calling `consume`, then on the first post‑ignition block we have `blockNumber >= inProgress`, so the require reverts. There is no on‑chain path to advance `inProgress`, leaving `propose` permanently stuck after toggling transactions on.

Impact:

- Liveness halt when enabling transactions after an ignition period (rollup can't propose new blocks).
- Requires contract upgrade or manual intervention to recover.

Recommendation: Add a catch‑up path in `Inbox.consume` that safely advances `inProgress` when `_toConsume >= inProgress` (returning the empty tree root), or provide a dedicated `catchUp` method callable only by `ROLLUP`.

Safety note: During ignition, `Inbox.sendL2Message` is gated by `IRollup(ROLLUP).getManaTarget() > 0`, so no L1→L2 messages are inserted. It is therefore safe to "catch up" by returning empty roots for skipped blocks.

- Option A: Make a `consume` call which only increases the `inProgess` when in ignition phase.
- Option B: Add `catchUp(uint256 pendingBlockNumber)` (only `ROLLUP`) to set `inProgress = pendingBlockNumber + 2`, and have `propose` call it the first time transactions are enabled.

Either approach prevents deadlock and preserves safety because no messages can exist while `manaTarget` is zero.

Aztec Labs: Fixed in PR 16637.

Cantina Managed: Fixed.

3.3 Medium Risk

3.3.1 `Propose` griefing attack via mutated attestations signatures

Severity: Medium Risk

Context: ProposeLib.sol#L168

Description: A third party can frontrun a valid `propose` transaction by copying it, keeping the valid proposer signature and committee addresses, but flipping any non‑proposer signature bytes to be invalid. The call to `ValidatorSelectionLib.verifyProposer(...)` only verifies:

- The proposer address for the slot (via committee reconstruction from `_signers` + bitmap), and...
- The proposer's signature over the payload digest.

It does not validate the rest of the committee signatures at propose time by design (to save gas) and it is not required that the `msg.sender` is the acutal `proposerId`.

As a result, the mutated transaction is accepted, the original proposer's transaction reverts ("slot already used"), and the proposed block later needs to be invalidated (or will fail at proof time) due to invalid or insufficient attestations.

Attack sketch:

- Copy a valid `propose` transaction, keep `_signers` and proposer's signature intact.
- Flip any non‑proposer signature bytes in `_attestations` to garbage.
- Submit first. `propose` succeeds because only the proposer signature is checked; the mutated attestations hash is stored.
- The legitimate transaction reverts because the slot is already used.
- Later, the block must be invalidated (bad attestation/insufficient attestations), creating a liveness hiccup and wasted slot/reward.

Impact:

- Temporary liveness degradation: blocks need to be invalidated before progress, or proofs will fail.
- Missed slot and reward for the legitimate proposer; increased operational overhead for committee (they must invalidate).

Recommendation: Bind the `attestation` set to the proposer in `propose` so a frontrunner cannot mutate it. Require an extra proposer‑only signature over both the payload digest and the `attestationsHash`.

Aztec Labs: Fixed in PR 16753.

Cantina Managed: Fixed.

3.3.2 `MerkleLib.computeRoot` inflates array length without padding (reads adjacent memory)

Severity: Medium Risk

Context: MerkleLib.sol#L67

Description: `computeRoot` adjusts the input array length to the next power-of-two via `assembly { mstore(_leafs, treeSize) }` without allocating or initializing additional elements for the array. If any memory was allocated after `_leafs` prior to this call, the function reads those adjacent slots as if they were leaves, producing a different merkle root for the same logical input.

This matches the in-code comment "@todo Must pad the tree" and is reproducible with a simple proof of concept that allocates another array after `_leafs` and before calling `computeRoot`.

Note: `computeUnbalancedRoot` supplies power-of-two arrays to `computeRoot`, so it is not affected. The issue arises when callers pass non-power-of-two leaf arrays directly to `computeRoot`.

Impact:

- Non-deterministic/incorrect Merkle root for the same logical leaf set depending on memory layout.
- Currently, the libary has the function but other contracts only use the `MerkleLib.verifyMembership` function.

Proof of Concept: A simple proof of concept for illustration:

```
function testComputeRootReadMemError() public {
  bytes32 root1 = _callComputeRoot(false);
  bytes32 root2 = _callComputeRoot(true);
  assertEq(root1, root2, "if not the same, computeRoot reads from used memory");
}

function _callComputeRoot(bool constructNextArray) internal returns (bytes32) {
  bytes32[] memory leaves = new bytes32[](3);
  leaves[0] = keccak256("L0");
  leaves[1] = keccak256("L1");
  leaves[2] = keccak256("L2");

  if(constructNextArray) {
    bytes32[] memory nextArray = new bytes32[](1);
    bytes32 nextArrayValue = keccak256(abi.encode("I want ot be part of the merkle tree"));
    nextArray[0] = nextArrayValue;
  }
  assertEq(leaves.length, 3, "length should be 3");

  // computeRoot will read the length value of nextArray and use it as a leaf
  bytes32 root = MerkleLib.computeRoot(leaves);

  assertEq(leaves.length, 4, "post: leaves length now 4");
  return root;
}
```

Recommendation: Either enforce power-of-two input length or explicitly pad into a fresh array before hashing.

- Option A - enforce power-of-two length and remove `mstore`:

  ```
  function computeRoot(bytes32[] memory _leafs) internal pure returns (bytes32) {
    uint256 len = _leafs.length;
    require(len > 0 && (len & (len - 1)) == 0, "MerkleLib: length must be power of two");
    uint256 treeDepth = 0;
    while ((1 << treeDepth) < len) treeDepth++;
    uint256 treeSize = 1 << treeDepth;
    // compute over _leafs in place...
  }
  ```

- Option B - pad into a new array if needed:

  ```
  function computeRoot(bytes32[] memory _leafs) internal pure returns (bytes32) {
    uint256 treeDepth = 0;
    while ((1 << treeDepth) < _leafs.length) treeDepth++;
    uint256 treeSize = 1 << treeDepth;
    if (_leafs.length != treeSize) {
      bytes32[] memory padded = new bytes32[](treeSize);
  ```

```
        for (uint256 i = 0; i < _leafs.length; i++) padded[i] = _leafs[i];
        _leafs = padded; // zeros pad
      }
      // compute over _leafs safely...
    }
```

- Option C - remove `computeRoot` function.

If the `MerkleLib.computeRoot` function is not needed by the rollup contracts remove it from the library.

Aztec Labs: The code has been deleted and was fixed as part of PR 16859.

Cantina Managed: Fixed.

3.3.3 `Validator` sampling from past state can result in inactive committee (liveness risk)

Severity: Medium Risk

Context: ValidatorSelectionLib.sol#L544

Description: The committee for an epoch is sampled using a stable historical snapshot one epoch minus one back:

```
function epochToSampleTime(Epoch _epoch) internal view returns (uint32) {
  return Timestamp.unwrap(_epoch.toTimestamp()).toUint32()
        - uint32(TimeLib.getEpochDurationInSeconds()) - 1;
}
```

This avoids last□minute set changes, but if many validators later move to a new rollup (e.g., staking with ”move with latest”), the sampled committee may include members who are no longer active on the current rollup and thus won't sign. Since proof verification requires >2/3 of the committee to sign, epochs may fail proof submission, forcing invalidation or pruning with ~50% of validators moved, the probability of meeting the threshold is very low.

Since, there are financial incentives for validators to always stake at the latest rollup, it seems realistic that a lot of them will use the ”move with latest” feature for staking.

Impact:

- Provers may be unable to gather >2/3 signatures for affected epochs $\rightarrow$ not possibe to produce new blocks.
- Liveness problems (invalidation/pruning) until a new epoch with a new committee starts. In the worst case, inactivity spans two epochs.

Recommendation: This risk for an old rollup during a transition should be clearly communicated to validators and users if it is an accepted trade□off. Coordinate validator moves at epoch boundaries and add a governance switch (activated before `epochToSampleTime`) to sample only from the rollup instance set during upgrades.

Aztec Labs: Acknowledged. This is acceptable. We are fine with the rollup behaving slightly less like normal right around the time of upgrades.

Cantina Managed: Acknowledged.

3.3.4 Attacker can fill Inbox with spam messages at low cost

Severity: Medium Risk

Context: Inbox.sol#L88-L112

Description: The inbox contract is meant to handle L1 => L2 messages from users, which are stored as merkle tree leaves and eventually consumed when a block is proposed on the L2. Anyone can call `sendL2Message()` function and insert a message of their own at no cost (except the L1 gas fees). The Inbox contract tracks these messages in a fixed height merkle tree for each block number, and if the current tree gets filled the later messages get added to the next tree. Because of this design, spamming the tree with random messages can not cause a DOS.

But an attacker can still post low cost messages by calling `sendL2Message` repeatedly and spam the tree to the point that `trees[inProgress]` gets filled for many future block numbers. This could lead to an unnecessary delay in posting and consumption of actual valid messages by normal users, as their message would be added to future tree.

Impact: Unnecessary delay in posting and consumption of actual valid user messages (for users who really wanted to send a valid message to L2 recipient). Also, the current tree size might be insufficient for normal usage(even if not spammed) in case that aztec rollup and this messageBridge gains enough popularity.

Currently, these message trees are of a fixed size (currently height 4, so 16 leaves). So, if an L1 block is 12s you would need 300 blocks to block 1 hour. (12s * 300 = 3600s = 1h). This means it would require 300 *16 leaves = 4,800 leaves.

Recommendation: Consider increasing the height of the tree, so that message tree is sufficiently big to handle any spam attacks, and real demand on the Inbox. Another solution is to add a small fee in the inbox, which could increase the cost of such a spam attack.

Aztec Labs: Acknowledged. For ignition it is not needed to address as there will be no messages. But for alpha it might make sense and we will be looking into it to increase the tree size. But as there might be other changes we and not used in ignition we allow it for now.

Cantina Managed: Acknowledged.

3.3.5 User funds could get stuck in `FeeJuicePortal` if block production halts on L2

Severity: Medium Risk

Context: *(No context files were provided by the reviewer)*

Description: The `FeeJuicePortal` contract has a function `depositToAztecPublic()` which helps users move their feeAsset tokens from L1 to L2. The way it works is by inserting an L2 message into the inbox with L2 recipient set to "L2_TOKEN_ADDRESS" which will then handle the claiming of these tokens.

```
// Send message to rollup
    (bytes32 key, uint256 index) = INBOX.sendL2Message(actor, contentHash, _secretHash);
```

The key point here is that this amount of feeAsset tokens is pulled into the FeeJuicePortal and kept locked here.

```
// Hold the tokens in the portal
UNDERLYING.safeTransferFrom(msg.sender, address(this), _amount);
```

One outcome of this process is => If a user deposited feeAsset tokens to the portal, but can't claim on the L2, there is no way to cancel the L2 message, and thus his tokens will be stuck on the portal. This could happen when the "L2 message tree" never gets consumed, ie. when the block production stops.

A potential scenario when this can happen is :

- If most of the validators use the moveWithLatestRollup feature, and they move on with the active validator set of the latest rollup.
- There are not enough committee members in the current validator set, as this is no longer the latest rollup instance.
- This means block proposal on the old rollup X is not possible now, inbox.consume() cannot be called, so all remaining Inbox message trees ahead are stuck.

Now Alice's tokens are stuck in the portal as her L2 message cannot be consumed, and cannot be cancelled.

This can also happen with normal L2 messages getting stuck (though with lower impacts) because old rollup is not producing blocks => because of not having enough validator set, or potentially other reasons.

Impact: If block production halts due to any reason, the inbox consumption will halt. The visible impact has been mentioned here as user funds could get locked forever in `FeeJuicePortal` contract if block production does not ever resume (in case enough validators are not interested).

Recommendation: Introduce an exit mechanism for unconsumed L2 messages after a timeout to allow users to recover locked funds in the FeeJuicePortal. For example this could be achieved by encoding a timestamp in the deposit message and letting governance define a cutoff time if block production halts, ensuring that unclaimed funds can be safely withdrawn on L1.

This cutoff timestamp could be set by governance on L1 and L2 based on what latest message got consumed in the last block produced. Note that this solution might not work for other type of L2 message stuck in the Inbox.

Another simpler solution is to design alternate ways to keep legacy rollup instances running : either with Aztec's own validators, or open sourcing block production in extreme situations to prevent this from happening, which would solve all problems related to block production halting.

Aztec Labs: Acknowledged. While possible, it is not expected that users will directly deposit into the fee asset portal due to it being a one-way bridge. So it being stuck due to lack of block production seems less of an issue. At the same time, for at least the ignition deployment it won't be used at all since there won't be transactions, so it does not seem like an issue for this.

An escape hatch to ensure block production should ensure that it is not stalled and thereby allow eventual exit.

Cantina Managed: Acknowledged.

3.3.6 Validators' deposits can get stuck in Staking queue

Severity: Medium Risk

Context: StakingLib.sol#L537-L539

Description: The staking process in rollup contracts follows a two-step routine. At first, validators signal their interest by depositing `ACTIVATION_THRESHOLD` amount of tokens, which inserts them into a queue. This queue is then processed once an epoch, where a dynamic number of participants are selected from the queue, and added to the active validator set.

When `flushEntryQueue()` is called to push depositors to the validator set, it uses `getEntryQueueFlushSize()` to calculate how many queue items should be processed.

The logic in `getEntryQueueFlushSize()` considers many different variables to conclude how many attesters can be added :

```
function getEntryQueueFlushSize() internal view returns (uint256) {
  StakingStorage storage store = getStorage();
  StakingQueueConfig memory config = store.queueConfig.decompress();

  uint256 activeAttesterCount = getAttesterCountAtTime(Timestamp.wrap(block.timestamp));
  uint256 queueSize = store.entryQueue.length();

  // Only if there is bootstrap values configured will we look into bootstrap or growth phases.
  if (config.bootstrapValidatorSetSize > 0) {
    // If bootstrap:
    if (activeAttesterCount == 0 && queueSize < config.bootstrapValidatorSetSize) {
      return 0;
    }
    // If growth:
    if (activeAttesterCount < config.bootstrapValidatorSetSize) {
      return Math.min(config.bootstrapFlushSize, StakingQueueLib.MAX_QUEUE_FLUSH_SIZE);
    }
  }
  // If normal:
  return Math.min(
    Math.max(activeAttesterCount / config.normalFlushSizeQuotient, config.normalFlushSizeMin),
    StakingQueueLib.MAX_QUEUE_FLUSH_SIZE
  );
}
```

Note that the `activeAttesterCount` includes both : attester count of this specific rollup instance + attester count of the bonus instance. If `activeAttesterCount == 0`, this returns zero and the `flushEntryQueue()` ends without adding any queue items into the validator set. This state can happen in the following scenario:

- There are 100 active validators on rollup X, everyone opted for `moveWithLatestRollup`, so all of them exist in the bonus instance registry.
- Alice deposits into rollup X hoping to enter the validator set in near future.
- The rollup gets upgraded.
- `flushEntryQueue()` is now called for the old rollup X, it calls `getEntryQueueFlushSize()`.

- `activeAttesterCount` for the old rollup now returns 0 because all attester deposits automatically moved to the latest rollup ie. bonus address.
- At this point it is guaranteed that Alice's deposit request cannot be processed as no new attesters can be flushed, and her request is stuck in the queue unless the queue is filled which is unlikely as no one wants to take part in a dead rollup.

This can also happen if a rollup was already dead and users start depositing again later in the hope to make it working again, but the bootstrapValidatorSetSize can never be reached and the funds are stuck.

Impact: Any pending requests in the deposit queue of the old rollup version get stuck and there is no way to cancel the deposit to get a refund.

Recommendation: Consider adding a kill switch to the Staking logic, which if turned on by governance, allows users to cancel their deposit requests in order to get refunded with their tokens.

Also, currently, the bootstrap phase (and other phases) can be cyclical and lead to other similar problems, it is suggested to make these phases once-in-a-lifetime for a particular rollup instance.

Aztec Labs: Fixed in PR 17067.

Cantina Managed: Fix verified.

3.4 Low Risk

3.4.1 `getBlobCommitmentsHash` returns incorrect data for future block numbers

Severity: Low Risk

Context: STFLib.sol#L219

Description: `STFLib.getStorageTempBlockLog(_blockNumber)` only checks that a block is not stale (too old) via:

```
bool isStale = _blockNumber + size <= pending;
require(!isStale, Errors.Rollup__StaleTempBlockLog(_blockNumber, pending, size));
```

There is no upper‑bound check for future blocks (`_blockNumber > pending`). For such inputs, the staleness check passes and the code reads `tempBlockLogs[_blockNumber % size]`, which may return unrelated data from the circular buffer.

The calls in the write paths ensure the block number is valid. However, the external getter `Rollup.getBlobCommitmentsHash(uint256)` forwards directly to `STFLib.getBlobCommitmentsHash(_blockNumber)` without a pending‑tip bound, so passing a future block number can return incorrect data rather than reverting.

Impact:

- Read‑only API inconsistency: callers can receive incorrect `blobCommitmentsHash` for non‑existent (future) blocks.
- Off‑chain clients/tools may misinterpret results.

Recommendation: Add an upper‑bound pending‑tip check to external getters (consistent with `getBlock`):

```
// Rollup.sol
function getBlobCommitmentsHash(uint256 _blockNumber) external view override(IRollup) returns (bytes32) {
  uint256 pendingBlockNumber = STFLib.getStorage().tips.getPendingBlockNumber();
  require(_blockNumber <= pendingBlockNumber,
          Errors.Rollup__InvalidBlockNumber(pendingBlockNumber, _blockNumber));
  return STFLib.getBlobCommitmentsHash(_blockNumber);
}
```

Aztec Labs: Fixed in PR 17299

Cantina Managed: Fixed.

3.4.2 `claimProverRewards`: batch reward claim reverts if any epoch already claimed (griefing attack)

Severity: Low Risk

Context: RewardLib.sol#L109

Description: `claimProverRewards(_prover, _epochs)` iterates the requested epochs and uses a `require(!proverClaimed.get(epoch))`. If any one epoch in the batch has already been claimed, the entire transaction reverts. A frontrunner can pre‑claim a single epoch in the target batch to make the user's bulk claim revert, forcing retries with edited inputs.

Recommendation: Make the loop robust and skip already‑claimed epochs instead of reverting the entire call.

Aztec Labs: Fixed in PR 17299

Cantina Managed: Fixed.

3.4.3 Reward calculations can be inconsistent for an epoch if sequencerBps or `blockReward` is modified

Severity: Low Risk

Context: RewardLib.sol#L79-L82

Description: In RewardLib, the `sequencerBps` and `blockReward` are part of the reward config which is used in calculating rewards for the sequencer.

The RewardLib also has a way to modify these values via `setConfig()`, which is callable by the governance. If the governance proposal+voting+delay timeline ends while an epoch was running, the config values will get applied suddenly.

We know that whenever an epoch block proof is submitted via `submitEpochRootProof()`, it calls `RewardLib.handleRewardsAndFees()` to calculate the rewards for both sequencer and prover.

The issue arises for the sequencer fee as it depends on these two variables. This could lead to a certain block proposer earning more/ less than another block proposer in the same epoch, even though they belong to the same epoch and validator set.

Impact: The rewards applied across an epoch can become uneven if the sequencerBps or blockReward is abruptly modified while an epoch is already running. If a sequencer gets lower rewards than they expected (if its proven after reward decreased), it might be unfair to them.

Recommendation: Consider adding logic to align any config updates with new epoch starting. This can be done by associating an epoch with fixed config values stored in a mapping, and setting up config for each epoch when it starts, just like how `setupEpoch()` works. This will ensure modifications only apply starting from the first block of the next epoch.

Aztec Labs: Acknowledged. The rewards impacted is very limited, and the extra cost would be applied all the time. Also, it will anyway be altered ”on the spot” if new rollups are made the canonical from the POV of the reward distributor, so it seems easier to understand if they behave the same way.

Cantina Managed: Acknowledged.

3.4.4 `deposit` front‑run using user's `BLS` keys temporarily locks user funds

Severity: Low Risk

Context: StakingLib.sol#L269

Description: An attacker can copy a user's G1/G2 public keys and PoP from the mempool and front‑run `deposit`. Both deposits enter the queue, but at flush time the attacker's `GSE.deposit` succeeds first and marks `ownedPKs[hashedIncomingPoint] = true`. The user's later `GSE.deposit` then reverts (duplicate key/AlreadyRegistered) and is refunded only during the flush, temporarily locking their capital until the queue is processed.

Recommendation: Add pre‑enqueue uniqueness checks (e.g., consult GSE for existing PK1 ownership or queued duplicates) before transferring funds.

Aztec Labs: Acknowledged. Performing the grief is fairly costly. As the attacker is adding a validator that they do not have keys for, so they would need to exit it right away to ensure that they won't be punished for inactivity etc...

Cantina Managed: Acknowledged.

3.4.5 Staking queue can be spammed with invalid requests

Severity: Low Risk

Context: *(No context files were provided by the reviewer)*

Description: The validator staking routine works as a two-step process : first users deposit funds and enter the staking queue, then the queue is processed once in an epoch, which adds new attesters to the registry. The way this works is:

- `flushEntryQueue()` calls `getEntryQueueFlushSize()` to get the maximum number of queue items that can be processed right now.
- For every validator entry, the logic tries to deposit validator funds to the GSE so that a new attester can be registered.
- If the deposit fails for any reason (except out-of-gas), it refunds the validator address as the deposit failed and goes ahead with processing other entries in the queue.

```
} else {
  // If the deposit fails, we need to handle two cases:
  // 1. Normal failure (data.length > 0): We return the funds to the withdrawer and continue processing
  //    the queue. This prevents a single failed deposit from blocking the entire queue.
  // 2. Out of gas failure (data.length == 0): We revert the entire transaction. This prevents an attack
  //    where someone could drain the queue without making any deposits.
  //    We can safely assume data.length == 0 means out of gas since we only call trusted GSE contract.
  require(data.length > 0, Errors.Staking__DepositOutOfGas());
  store.stakingAsset.transfer(args.withdrawer, amount);
  emit IStakingCore.FailedDeposit(
    args.attester, args.withdrawer, args.publicKeyInG1, args.publicKeyInG2, args.proofOfPossession
  );
```

Since there is a limit on the number of queue entries that can be processed in a single flush, and due to the fact that `flushEntryQueue()` can be called only once in an epoch, it is possible to purposefully spam the queue with invalid parameters such that most deposits are guaranteed to fail.

In such a situation, registration of valid attesters can be delayed by an attacker as the queue is already filled up with invalid entries.

Impact: This attack can cause delays in the growth of the validator set for the chain, as real validators may not get flushed.

Recommendation: It is implied from the docs that limits on queue were placed to limit growth of the actual validator set. Consider applying these checks on the actual growth in the `activeAttesterCount`, instead of just applying them on the queue size.

Aztec Labs: Fixed in PR 17067.

Cantina Managed: Fix verified.

3.4.6 Public `checkpointRandao` allows opportunistic seed timing (first-caller-wins)

Severity: Low Risk

Context: RollupCore.sol#L538

Description: `checkpointRandao()` is public and sets the epoch+2 seed using the current L1 `prevrandao` the first time it's called for that epoch. A validator can monitor `prevrandao` across L1 blocks and call when the derived seed `keccak(epoch, prevrandao)` leads to a favorable committee/proposer outcome, creating a minor timing bias.

Recommendation: Make seeding time-deterministic (derive from a fixed L1 block/slot) and restrict writes to `setupEpoch` and remove the public method, or keep it public but document the policy.

Aztec Labs: We will acknowledge this but not alter it. The first caller locks in the value, and there will always be someone who wins if it is called immediately. In cases of heavy competition, it will be called right away, so the flexibility has no practical effect.

Cantina Managed: Acknowledged.

3.5 Gas Optimization

3.5.1 `computeSampleIndex` early return if index <= 1

Severity: Gas Optimization

Context: SampleLib.sol#L102

Description: The `computeSampleIndex` can automatically return 0, if the `_indexCount` is either 0 or 1:

```
  function computeSampleIndex(uint256 _index, uint256 _indexCount, uint256 _seed) internal pure returns
  ↪  (uint256) {
    // Cannot modulo by 0
-    if (_indexCount == 0) {
+   if (_indexCount <= 1) {
      return 0;
    }

    return uint256(keccak256(abi.encodePacked(_seed, _index))) % _indexCount;
  }
```

Recommendation: Add the early return because modulo by 1 always result in zero.

Aztec Labs: Fixed in PR 17299.

Cantina Managed: Fixed.

3.6 Informational

3.6.1 Test-only VM cheatcode path in `BlobLib`

Severity: Informational

Context: BlobLib.sol#L50

Description: `BlobLib` detects Foundry test environments via `VM_ADDRESS.code.length > 0` and then routes to cheatcodes (`Vm.getBlobBaseFee()` / `Vm.getBlobhashes()`) instead of on-chain sources (`block.blobbasefee` / `blobhash`).

While practical, this couples production code to a testing sentinel and makes the test path selectable only by runtime code presence.

Recommendation:

- Prefer an injected provider interface to fully separate test vs production behavior:
  - Define `IBlobProvider` with `getBlobBaseFee()` and `getBlobHash(uint256)`.
  - Use a production implementation (reads `block.blobbasefee` / `blobhash`) and a test implementation (wraps Foundry VM).
- Store the provider on the consuming contract and pass through to `BlobLib` helpers.
  - If keeping the current approach, add a defensive environment guard on the cheatcode path to reduce risk:
  - Require a known local chain id (e.g., `31337`) when `VM_ADDRESS.code.length > 0`, or assert both conditions:

    `if (VM_ADDRESS.code.length > 0) { require(block.chainid == 31337, "VM cheats only on local"); }`.

Both options keep tests convenient while making production behavior explicit and harder to misconfigure.

Aztec Labs: Acknowledged. We might implement it in the future but currently low in priority.

Cantina Managed: Acknowledged.

3.6.2 `ChainTips.updateProvenBlockNumber` can corrupt pending half for > `uint128` inputs

Severity: Informational

Context: Tips.sol#L44

Description: `updateProvenBlockNumber` clears the low 128 bits and ORs `_provenBlockNumber` into place without masking/casting:

```
uint256 value = CompressedChainTips.unwrap(_c) & ~PROVEN_BLOCK_NUMBER_MASK;
return CompressedChainTips.wrap(value | _provenBlockNumber);
```

If `_provenBlockNumber > type(uint128).max`, its high bits will spill into the upper half and corrupt the pending block number. While such high block numbers are unrealistic, `updatePendingBlockNumber` uses a safe `<< 128`, and `compress` casts to `uint128`, so adding the same guard improves robustness.

Recommendation: Enforce a bound or cast in `updateProvenBlockNumber`: `require(_provenBlockNumber <= type(uint128).max);`.

Aztec Labs: Fixed in PR 17299.

Cantina Managed: Fixed.

3.6.3 Add a sanity check for the shift operation in `Outbox.consume` to avoid overflow

Severity: Informational

Context: Outbox.sol#L89

Description: `leafId = (1 << _path.length) + _leafIndex` can overflow when `_path.length >= 256` because left�shifting a 256�bit value by 256 (or more) wraps to zero, collapsing the depth prefix and risking collisions. While such depths are unrealistic, a small guard makes intent explicit.

Recommendation: Add a simple sanity check before computing `leafId`:

```
require(_path.length < 256, "Outbox: path too deep");
```

Aztec Labs: Fixed in PR 16847.

Cantina Managed: Fixed.

3.6.4 `_invalidIndex` lacks bounds check in `invalidateBadAttestation`

Severity: Informational

Context: InvalidateLib.sol#L90

Description: `invalidateBadAttestation` indexes `_committee[_invalidIndex]` without an explicit bounds check. If `_invalidIndex` is out of range, the call would only revert in the last `require` statement with an array out-of-bounds error rather than a clear domain error.

This is especially relevant here because most of the operation related to attestations are implemented in assembly and would not revert if the `_invalidIndex` is too high.

Recommendation: Add an early bounds check before reading from `_committee`:

```
require(_invalidIndex < _committee.length, "InvalidateLib: invalid index");
```

Aztec Labs: Fixed in PR 17299.

Cantina Managed: Fixed.

3.6.5 Prevent `Outbox.insert` to be called for proven `_l2BlockNumber`

Severity: Informational

Context: Outbox.sol#L50

Description: Currently, the `Outbox.insert` function is only called as part of the proposal process for `pending` blocks. If `insert` were called for an already proven block to overwrite the `root`, the state of the `Outbox` contract would be broken, since the `insert` function would not properly reset existing nullifiers from the Merkle proofs.

This would prevent consuming leaves that were already consumed under the old root with the same global index.

Recommendation: Add a `require` statement to ensure the `_l2BlockNumber` has not been proven when calling `insert` to make the contract more robust.

```
require(_l2BlockNumber > ROLLUP.getProvenBlockNumber());
```

Aztec Labs: Fixed in PR 17299.

Cantina Managed: Fixed.

3.6.6 General Code Improvements

Severity: Informational

Context: *(No context files were provided by the reviewer)*

Description: Here are multiple small recommendations to improve the overall quality of the codebase.

Recommendation:

1. `ProposeLib` timestamp normalization: Normalize `block.timestamp` once (to slot-aligned time) and reuse for epoch/slot conversions; `_args.header.timestamp` already needs to be slot-aligned.
2. `StakingQueue.getLast` naming: Method name is confusing; it returns the next available index (tail), not the last occupied entry - consider renaming or document it.
3. `MerkleLib` test coverage: Only `verifyMembership` appears used by Outbox and has tests. add tests for other functions or remove not needed functions.
4. `EpochProofLib` comment clarity: Say "verifies attestations for the provided end block" (it may or may not be the last block of the epoch).
5. In `ValidatorSelectionLib`, a comment at L#39 says "Committee size is configurable (targetCommitteeSize)" which is not true.
6. RewardLib zero transfer: Guard with `if (accumulatedRewards > 0)` before calling `transfer` to avoid no-op token transfer calls.
7. The name `sequencerShare` in `RewardLib.handleRewardsAndFees` is a bit confusing it is the `sequencerBlockRewards` an not a share value. The `sequencerBps` would be the share in basis points of the `blockRewardsAvailable`.
8. `FeeStructsLib` has incorrect comments which bits are read from the compromissed bytes. for example `getExcessMana` comment says reads the bits 224-256 but it acutally reads `0..31` or `getExcessMana` the comment says reads `176-223` but code reads `32..79`.
9. `preheat` helper function in `FeeStructsLib` isn't used.
10. `FeeConfig` could use the `MASK_128_BITS` constant instead of hardcoded `0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF`.

Aztec Labs: Addressed items 2, 4, 5, 7, 8, 9, 10 in PR 17300 and item 6 in PR 17299.

Cantina Managed: Fixed.

3.6.7 `_isFirstBlockOfEpoch==true` implicitly requires at least one blob in `BlobLib.calculateBlobCommitmentsHash`

Severity: Informational

Context: BlobLib.sol#L207

Description: The `calculateBlobCommitmentsHash` directly access the `_blobCommitments[0]` when `_isFirstBlockOfEpoch = true`:

```
if (_isFirstBlockOfEpoch) {
  // Initialise the blobCommitmentsHash
  currentBlobCommitmentsHash = Hash.sha256ToField(abi.encodePacked(_blobCommitments[i++]));
}
```

This implicitly means that the first block of an epoch can't be empty and needs to include at least one blob. Otherwise, this would be an out of bounds array read operation and a revert. For other blocks this doesn't seem to be the case and blocks can be empty.

Recommendation: If empty blocks with no blobs should be possible allow it also for the first block of an epoch.

Aztec Labs: Fixed in PR 16859.

Cantina Managed: Fixed.